A Pipelined Multi Core MIPS Machine

Hardware Implementation and Correctness Proof

Wolfgang J. Paul

April 17, 2012
# Contents

1 Number Formats and Boolean Algebra .......................... 9  
1.1 Basics .................................................. 9  
1.2 Modulo Computation ..................................... 11  
1.3 Geometric Sums ......................................... 13  
1.4 Binary Numbers ......................................... 14  
1.5 Two's Complement Numbers ............................... 17  
1.6 Boolean Algebra ......................................... 19  
1.6.1 Identities ........................................... 20  
1.6.2 Solving Equations .................................... 22  
1.6.3 Disjunctive Normal Form ............................. 23  

2 Hardware .................................................. 25  
2.1 Gates and Circuits ....................................... 25  
2.2 Some Basic Circuits ..................................... 27  
2.3 Clocked Circuits ........................................ 33  
2.3.1 Digital Clocked Circuits ............................. 34  
2.3.2 The detailed hardware model ........................ 36  
2.3.3 Timing Analysis ..................................... 41  
2.4 Registers ................................................ 44  
2.5 Drivers and Main Memory ................................ 45  
2.5.1 Open collector drivers and active low signal ...... 46  
2.5.2 Tri state drivers and bus contention ............... 47  
2.5.3 The incomplete digital model for drivers .......... 51  
2.5.4 Self destructing hardware ........................... 52  
2.5.5 Clean operation of tri state buses ................ 54  
2.5.6 Specification of main memory ....................... 59  
2.5.7 Operation of main memory via a tri state bus ..... 62  
2.6 Finite State Transducers ................................. 65  
2.6.1 Realization of Moore Automata ...................... 66  
2.6.2 Precomputing Outputs of Moore Automata .......... 68  
2.6.3 Realization of Mealy Automata ...................... 69  
2.6.4 Partial Precomputation of Outputs of Mealy Automata 70
3 Nine Shades of RAM

3.1 Basic Random Access Memory ................................................. 73
3.2 Single Port RAM Designs ................................................. 76
  3.2.1 Read Only Memory (ROM) ............................................. 76
  3.2.2 Combining RAM and ROM .......................................... 77
  3.2.3 Multi Bank RAM .................................................. 77
  3.2.4 Cache State RAM ................................................ 81
  3.2.5 SPR-RAM ...................................................... 82
3.3 Multiport RAM Designs ................................................. 83
  3.3.1 Three port RAM for general purpose registers ............. 83
  3.3.2 General Two Port RAM ....................................... 86
  3.3.3 Two Port Cache State RAM .................................. 87

4 Arithmetic Circuits .................................................... 91

4.1 Adder and Incrementer ................................................ 91
4.2 Arithmetic Unit ....................................................... 92
4.3 ALU ................................................................. 98
4.4 Shifter ................................................................. 99
4.5 Branch Condition Evaluation Unit .................................. 105

5 A Basic Sequential MIPS Machine ...................................... 109

5.1 Tables ........................................................................ 109
  5.1.1 I-Type .......................................................... 109
  5.1.2 R-type .......................................................... 111
  5.1.3 J-type .......................................................... 111
5.2 MIPS ISA .................................................................. 112
  5.2.1 Configuration and instruction fields ......................... 112
  5.2.2 Instruction Decoding ......................................... 115
  5.2.3 ALU-Operations .............................................. 116
  5.2.4 Shift Unit Operations ....................................... 118
  5.2.5 Branch and Jump .............................................. 119
  5.2.6 Sequences of consecutive memory bytes .................. 120
  5.2.7 Loads and Stores .............................................. 122
  5.2.8 ISA Summary .................................................. 124
5.3 A Sequential Processor Design ...................................... 124
  5.3.1 Software Conditions .......................................... 125
  5.3.2 Embedding byte addressable memory into line addressable memory ............................................. 126
  5.3.3 Defining Hardware Correctness for the Processor Design ................................................... 129
  5.3.4 Stages of Instruction Execution ............................. 131
  5.3.5 Initialization ................................................... 132
  5.3.6 Instruction Fetch ............................................... 132
  5.3.7 Instruction Decoder ........................................... 133
  5.3.8 Reading from General Purpose Registers ............... 137
5.3.9 Next pc environment .................................. 137
5.3.10 ALU environment ................................. 141
5.3.11 Shift unit environment .......................... 142
5.3.12 jump and link ........................................ 143
5.3.13 Collecting results .................................. 143
5.3.14 Effective Address .................................. 143
5.3.15 Shift for Store environment ..................... 144
5.3.16 Memory Stage ...................................... 145
5.3.17 Shifter for Load .................................... 148
5.3.18 Writing to the General Purpose Register File .... 149

6 Pipelining 151
6.1 MIPS ISA and basic implementation revisited .... 151
6.1.1 Delayed PC .......................................... 151
6.1.2 Implementing the delayed pc .................... 152
6.1.3 Pipeline stages and visible registers .......... 153
6.2 Basic pipelined processor design ................. 157
6.2.1 Transforming the sequential design into a pipelined
      design ............................................... 157
6.2.2 Scheduling functions ............................. 160
6.2.3 Use of invisible registers ....................... 163
6.2.4 Software condition SC – 1 ....................... 164
6.2.5 Correctness statement ........................... 165
6.2.6 Correctness proof of the basic pipelined design . 166
6.3 Forwarding ............................................. 176
6.3.1 Hits ................................................. 177
6.3.2 Forwarding Circuits ............................... 177
6.3.3 Software condition SC – 2 ....................... 178
6.3.4 Scheduling functions revisited ................. 178
6.3.5 Correctness Proof ................................. 179
6.4 Stalling ................................................ 182
6.4.1 Stall Engine ....................................... 182
6.4.2 Hazard Signals .................................... 184
6.4.3 Correctness statement ........................... 185
6.4.4 Scheduling Functions ............................. 185
6.4.5 Correctness Proof ................................. 189
6.4.6 Liveness ............................................ 190

7 Caches and Shared Memory 191
7.1 Concrete and Abstract Caches ...................... 191
7.1.1 Abstract caches and cache coherence .......... 191
7.1.2 Direct mapped caches ............................. 194
7.1.3 k-way associative caches ....................... 196
7.1.4 Fully associative caches ......................... 198
7.2 Notation ........................................... 201
  7.2.1 Parameters ................................. 201
  7.2.2 Memory and memory systems .............. 201
  7.2.3 Accesses and Access Sequences .......... 202
  7.2.4 Sequential memory semantics .............. 203
  7.2.5 Sequentially consistent memory systems .. 204
  7.2.6 Memory system hardware configurations .. 204
7.3 Atomic MOESI Protocol ......................... 205
  7.3.1 Invariants ................................ 206
  7.3.2 Defining the protocol by tables ........ 208
  7.3.3 Translating the tables into sets of switching functions. 210
  7.3.4 Algebraic specification of the atomic MOESI protocol 211
  7.3.5 Properties of the atomic protocol ......... 214
7.4 Gate Level Design of a Shared Memory System ... 216
  7.4.1 Specification of interfaces ............... 217
  7.4.2 Data paths of caches .................... 220
  7.4.3 Cache Protocol Automata .................. 226
  7.4.4 Automata Transitions and Control Signals 229
  7.4.5 Bus arbiter ................................ 234
7.5 Correctness Proof ............................... 236
  7.5.1 Arbitration ................................ 237
  7.5.2 Silent slaves on the open collector (OC) bus 239
  7.5.3 Automata synchronization .................. 239
  7.5.4 Control of tri state drivers .............. 242
  7.5.5 Protocol Data Transmission ................. 245
  7.5.6 Data Transmission ........................ 248
  7.5.7 Accesses of the Hardware Computation .... 250
  7.5.8 Relating hardware computation with single steps of the atomic protocol ................. 257
  7.5.9 Ordering Hardware Accesses Sequentially . 259
  7.5.10 Sequential Consistency .................... 262

8 A Multicore Processor ............................. 265
  8.1 Multi-core ISA ............................... 265
    8.1.1 ISA specification ........................ 265
    8.1.2 Sequential reference implementation .... 266
    8.1.3 Simulation Relation ....................... 267
    8.1.4 Local configurations and computations .... 268
    8.1.5 Accesses of the reference computation .... 271
  8.2 Shared Memory in the Multicore System ....... 272
    8.2.1 Connecting Interfaces ..................... 272
    8.2.2 Stability of inputs of accesses .......... 273
    8.2.3 Relating updates enable signals and ends of accesses . 274
    8.2.4 Scheduling function ....................... 277
Chapter 1

Number Formats and Boolean Algebra

1.1 Basics

We denote by

\[ N = \{0, 1, 2, \ldots\} \]

the set of natural numbers including zero, by

\[ \mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\} \]

the set of integers and by

\[ \mathbb{B} = \{0, 1\} \]

the set of bits. For bits \( x \in \mathbb{B} \) and natural numbers \( n \in \mathbb{N} \) we denote the string obtained by repeating \( x \) exactly \( n \) times by \( x^n \)

\[ x^n = x \underbrace{\ldots x}_n \]

Strictly speaking, definitions using three dots are never precise; they are intelligence tests, where the author hopes that all readers who are forced to take the test arrive at the same solution. Usually one can easily find in such situations completely precise recursive definitions without three dots. We denote by \( \circ \) the concatenation of bit strings and redefine \( x^n \) by

\[
\begin{align*}
x^1 &= x \\
x^{n+1} &= x \circ x^n
\end{align*}
\]

Thus \( 1^2 = 11 \) and \( 0^4 = 0000 \). For integers \( i, j \) with \( i < j \) we define the interval of integers from \( i \) to \( j \) as an intelligence test by

\[ [i : j] = \{i, i + 1, \ldots, j\} \]
\begin{table}
\begin{tabular}{|c|c|}
\hline
\( \land \) & and \\
\( \lor \) & or \\
\( \neg \) & not \\
\( \oplus \) & exclusive or, \( + \) modulo 2 \\
\( \rightarrow \) & implies \\
\( \iff \) & if and only if \\
\( \forall \) & for all \\
\( \exists \) & exists \\
\hline
\end{tabular}
\end{table}

and formally by
\[
[i : i] = \{i\}
\]
\[
[i : j + 1] = \{j + 1\} \cup [i : i]
\]

In statements and predicates we use the logical connectives and quantifiers from Table 1.1. For \( \neg x \) we also write \( \overline{x} \) or \( /x \). For sets \( A \) and \( n \in \mathbb{N} \setminus \{0\} \) we denote by
\[
A^n = \{(a_{n-1}, \ldots, a_0) \mid \forall i : a_i \in A\}
\]
the set of sequences of length \( n \) with elements from \( A \). Formally, a sequence \( a \in A^n \) can be viewed as a mapping
\[
a : [0 : n - 1] \to A
\]

Sequence elements are denoted in various ways:
\[
a(i) = a_i = a[i]
\]

For sequences \( (a_{n-1}, \ldots, a_0) \in A^n \) several notations are used. We can leave out commas and brackets
\[
a_{n-1} \ldots a_0 = (a_{n-1}, \ldots, a_0)
\]

We also use a notation from computer aided design (CAD) systems for hardware
\[
a[n - 1 : 0] = (a_{n-1}, \ldots, a_0)
\]

For indices \( i \geq j \) we denote the subsequence of elements from index \( i \) down to \( j \) by
\[
a[i : j] = (a_i, \ldots, a_j)
\]

Formally we define for \( j \geq i \)
\[
a[i : i] = a[i]
\]
\[
a[j + 1 : i] = a[j + 1] \circ a[j : i]
\]
1.2. Modulo Computation

For situations, where sequence elements are numbered in increasing order from left to right we also define

\[ a[i : j + 1] = a[i : j] \circ a[j + 1] \]

For \( \circ \in \{ \land, \lor, \oplus \} \), strings \( a[n - 1 : 0] \) and \( b[n - 1 : 0] \) and bits \( c \) we borrow notation from vector calculus to define identical bit-operations on the components of vectors

\[ \overline{a} = (\overline{a_{n-1}}, \ldots, \overline{a_0}) \]
\[ a[n - 1 : 0] \circ b[n - 1 : 0] = (a_{n-1} \circ b_{n-1}, \ldots, a_0 \circ b_0) \]
\[ c \circ b[n - 1 : 0] = (c \circ b_{n-1}, \ldots, c \circ b_0) \]

The Hilbert \( \in \)-Operator picks an element from a set \( A \). Applied to a singleton set, it returns the unique element of the set.

\[ \in(x) = x \]

For finite sets \( A \) we denote by \( \#A \) the cardinality, i.e. the number of elements in \( A \).

1.2 Modulo Computation

There are infinitely many integers and every computer can only store finitely many numbers. Thus computer arithmetic cannot possibly work like ordinary arithmetic. Fixed point arithmetic\(^3\) is usually performed modulo \( 2^n \) for some \( n \). We review basics about modulo computation

**Definition 1.** For integers \( a, b \in \mathbb{Z} \) and natural numbers \( k \in \mathbb{N} \) one defines \( a \) and \( b \) to be congruent \( \mod k \) iff they differ by an integer multiple of \( k \).

\[ a \equiv b \mod k \iff \exists z \in \mathbb{Z} : a - b = z \cdot k \]

**Definition 2.** Let \( R \) be a relation between elements of a set \( A \). We say that \( R \) is reflexive if we have \( aRa \) for all \( a \in A \). We say that \( R \) is symmetric if \( aRb \) implies \( bRa \). We say that \( R \) is transitive if \( aRb \) and \( bRc \) imply \( aRc \). If all three properties hold, \( R \) is called an equivalence relation on \( A \).

An easy exercise shows

**Lemma 1.** Equivalence \( \mod k \) is an equivalence relation.

\(^3\)The only arithmetic considered in this booklet. For the construction of floating point units see [MP00].
CHAPTER 1. NUMBER FORMATS AND BOOLEAN ALGEBRA

Proof:

- Reflexivity: For all \( a \in \mathbb{Z} \) we have \( a - a = 0 \cdot k \), thus \( a \equiv a \mod k \) thus equivalence mod \( k \) is reflexive.

- Symmetry: Let \( a \equiv b \mod k \) with \( a - b = z \cdot k \). Then \( b - a = -z \cdot k \), thus \( b \equiv a \mod k \).

- Transitivity: Let \( a \equiv b \mod k \) with \( a - b = z \cdot k \) and \( b \equiv c \mod k \) with \( b - c = u \cdot k \). Then \( a - c = (z + u) \cdot k \), thus \( a \equiv c \mod k \).

**Lemma 2.** Let \( a \equiv a' \mod k \) and \( b \equiv b' \mod k \). Then

\[
\begin{align*}
a + b & \equiv a' + b' \mod k \\
a - b & \equiv a' - b' \mod k
\end{align*}
\]

Proof: Let \( a - a' = u \cdot k \) and \( b - b' = v \cdot k \) then

\[
\begin{align*}
a + b - (a' + b') &= a - a' + b - b' \\
&= (u + v) \cdot k \\
a - b - (a' - b') &= a - a' - (b - b') \\
&= (u - v) \cdot k
\end{align*}
\]

**Lemma 3.** Two numbers \( r \) and \( s \) in an interval of the form \([i : i + k - 1]\) that are both equivalent to \( a \mod k \) are identical. Let \( r, s \in [i : i + k - 1] \) and \( a \equiv r \mod k \) and \( a \equiv s \mod k \). Then \( r = s \).

Proof: By symmetry we have \( s \equiv a \mod k \) and by transitivity we get \( s \equiv r \mod k \). Thus \( r - s = z \cdot k \) for an integer \( z \). We conclude \( z = 0 \) because \(|r - s| < k\).

**Definition 3.** Let \( R \) be an equivalence relation on \( A \). A subset \( B \subseteq A \) is called a system of representatives if for every \( a \in A \) there is exactly one \( r \in B \) with \( a R r \). The unique \( r \in B \) satisfying \( a R r \) is called the representative of \( a \) in \( B \).

**Lemma 4.** The interval of integers of the form \([i : i + k - 1]\) is a set of representatives for equivalence mod \( k \).

Proof: Let \( a \in \mathbb{Z} \). We define the representative \( r(a) \) by

\[
\begin{align*}
f(a) &= \begin{cases} 
\max \{j \mid a - k \cdot j \geq i\} & : a \geq i \\
\min \{j \mid a + k \cdot j \geq i\} & : a < i
\end{cases} \\
r(a) &= \begin{cases} 
a - f(a) \cdot k & : a \geq i \\
a + f(a) \cdot k & : a < i
\end{cases}
\end{align*}
\]
1.3. GEOMETRIC SUMS

Then \( r(a) \equiv a \mod k \) and \( r(a) \in \{i : i + k - 1\} \). Uniqueness follows from Lemma 3. Note that for \( a \geq i = 0 \) function \( f(a) \) is the result of the integer division of \( a \) by \( k \)

\[
f(a) = \lfloor a/k \rfloor
\]

and

\[
r(a) = a - \lfloor a/k \rfloor \cdot k
\]

is the remainder of this division.

We have to point out that in mathematics the three letter word \( \text{mod} \) is not only used for the relation defined above. It is also used as a binary operator in which case \( a \mod k \) denotes the representative of \( a \) in \([0 : k-1]\).

**Definition 4.**

\[
(a \mod k) = \{ b \mid a \equiv b \mod k \land b \in [0 : k-1] \}
\]

Thus \( a \mod k \) is the remainder of the integer division of \( a \) by \( k \) for \( a \geq 0 \). In order to stress when \( \mod \) is used as a binary operator, we always write \( a \mod k \) in brackets. For later use in the theory of two’s complement numbers we make the following

**Definition 5.** For even numbers \( k \)

\[
(a \text{ tmod } k) = \{ b \mid a \equiv b \mod k \land b \in [-k/2 : k/2-1] \}
\]

From Lemma 3 we infer a simple but useful lemma about the solution of equivalences \( \mod k \)

**Lemma 5.** Let \( k \) be even and \( x \equiv y \mod k \) then

1. \( x \in [0 : k-1] \rightarrow x = (y \mod k) \)
2. \( x \in [-k/2 : k/2-1] \rightarrow x = (y \text{ tmod } k) \)

1.3 Geometric Sums

For \( q \neq 1 \) let

\[
S = \sum_{i=0}^{n-1} q^i
\]

Then

\[
q \cdot S = \sum_{i=1}^{n} q^i
\]

\[
q \cdot S - S = q^n - 1
\]

\[
S = \frac{q^n - 1}{q - 1}
\]

For \( q = 2 \) we get
Lemma 6.
\[ \sum_{i=0}^{n-1} 2^i = 2^n - 1 \]

1.4 Binary Numbers

Definition 6. For strings \( a = a[n-1 : 0] \in \mathbb{B}^n \) we denote by
\[ \langle a \rangle = \sum_{i=0}^{n-1} a_i \cdot 2^i \]
the natural number with binary representation \( a \). A binary number is a string that is interpreted as the binary representation of a natural number.

Examples are
\[ \langle 100 \rangle = 4 \\
\langle 111 \rangle = 7 \\
\langle 10^n \rangle = 2^n \]

With Lemma 6 we get
\[ \langle 1^n \rangle = \sum_{i=0}^{n-1} 2^i = 2^n - 1 \]

Lemma 7. Binary representation of length \( n \) is injective. Let \( a, b \in \mathbb{B}^n \). Then
\[ a \neq b \rightarrow \langle a \rangle \neq \langle b \rangle \]

Proof: Let \( j = \max\{i \mid a_i \neq b_i\} \) be the largest index where strings \( a \) and \( b \) differ. Without loss of generality assume \( a_j = 1 \) and \( b_j = 0 \).

\[ \langle a \rangle - \langle b \rangle = \sum_{i=0}^{j} a_i \cdot 2^i - \sum_{i=0}^{j} b_i \cdot 2^i \]
\[ \geq 2^j - \sum_{i=0}^{j-1} 2^i \]
\[ = 1 \]

by equation 6.
We denote by
\[ B_n = \{ \langle a \rangle \mid a \in \mathbb{B}^n \} \]
the set of numbers that have a binary representation of length \( n \). As

\[ 0 \leq \langle a \rangle \leq \sum_{i=0}^{n-1} 2^i = 2^{n-1} \]

we find

\[ B_n \subseteq [0 : 2^n - 1] \]

As \( \langle \cdot \rangle \) is injective and

\[ \#B_n = \# [0 : 2^n - 1] = 2^n \]

we find that \( \langle \cdot \rangle \) is bijective and thus we have

**Lemma 8.**

\[ B_n = [0 : 2^n - 1] \]

For \( x \in B_n \) we denote the binary representation of \( x \) of length \( n \) by \( \text{bin}_n(x) \).

\[ \text{bin}_n(x) = \{ a \mid a \in \mathbb{B}^n \land \langle a \rangle = x \} \]

It is often useful to decompose \( n \) bit binary representations \( a[n-1 : 0] \) into an upper part \( a[n-1 : m] \) and a lower part \( a[m-1 : 0] \). The connection between the numbers represented is stated in

**Lemma 9.** (Decomposition Lemma) Let \( n \geq m \). Then

\[ \langle a[n-1 : 0] \rangle = \langle a[n-1 : m] \rangle \cdot 2^m + \langle a[m-1 : 0] \rangle \]

Proof:

\[
\langle a[n-1 : 0] \rangle = \sum_{i=m}^{n-1} a_i \cdot 2^i + \sum_{i=0}^{m-1} a_i \cdot 2^i \\
= \sum_{j=0}^{n-1-m} a_{m+j} \cdot 2^{m+j} + \langle a[m-1 : 0] \rangle \\
= 2^m \cdot \sum_{j=0}^{n-1-m} a_{m+j} \cdot 2^j + \langle a[m-1 : 0] \rangle \\
= 2^m \cdot \langle a[n-1 : m] \rangle + \langle a[m-1 : 0] \rangle
\]

We obviously have

\[ \langle a[n-1 : 0] \rangle \equiv \langle a[m-1 : 0] \rangle \mod 2^m \]

and infer with Lemma 5.


\[
\begin{array}{|c|c|c|c|c|}
\hline
a & b & c & c' & s \\
\hline
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 0 & 1 \\
0 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

Table 1.2: Binary addition of 1 bit numbers \(a, b\) with carry \(c\)

**Lemma 10.**

\[
\langle a[m - 1 : 0] \rangle = \langle \langle a[n - 1 : 0] \rangle \mod 2^m \rangle
\]

Thus in order to take a binary number modulo \(2^m\) all one has to do is to throw way the bits with position \(m\) or higher.

Table 1.2 specifies the addition algorithm for binary numbers \(a, b\) of length 1 and a carry \(c\). The binary representation \((s, c')\) of the sum of bits \(a, b\) and \(c\) is computed

\[
\langle c', s \rangle = a + b + c
\]

For the addition of \(n\) bit numbers \(a[n - 1 : 0]\) and \(b[n - 1 : 0]\) with carry in \(c_0\) we first observe for the sum \(S\):

\[
S = \langle a[n - 1 : 0] \rangle + \langle b[n - 1 : 0] \rangle + c_0 \\
\leq 2^n - 1 + 2^n - 1 + 1 \\
= 2^{n+1} - 1
\]

Thus the sum \(S \in \mathbb{B}^{n+1}\) can be represented as a binary number \(\langle s[n : 0] \rangle\) with \(n + 1\) bits. For the computation of the sum bits we use the method for long addition that we learn in elementary school for decimal numbers. We denote by \(c_i\) the carry from position \(i - 1\) to position \(i\) and compute \((c_{i+1}, s_i)\) by the basic binary addition algorithm from Table 1.2

\[
\langle c_{i+1}, s_i \rangle = a_i + b_i + c_i \quad (1.1)
\]

\[
s_n = c_n \quad (1.2)
\]

That this computes indeed the sum of the input numbers is asserted in

**Lemma 11. (Binary Addition)**

\[
\langle c_n, s[n - 1 : 0] \rangle = \langle a[n - 1 : 0] \rangle + \langle b[n - 1 : 0] \rangle + c_0
\]
1.5. **TWO’S COMPLEMENT NUMBERS**

Proof: by induction on \( n \). For \( n = 0 \) this follows directly from equation 1.1. For the induction step we conclude from \( n - 1 \) to \( n \):

\[
\langle a[n-1 : 0] \rangle + \langle b[n-1 : 0] \rangle + c_0
= (a_{n-1} + b_{n-1}) \cdot 2^{n-1} + \langle a[n-2 : 0] \rangle + \langle b[n-2 : 0] \rangle + c_0
= (a_{n-1} + b_{n-1}) \cdot 2^{n-1} + \langle c_{n-1}, s[n-2 : 0] \rangle \text{ (induction hypothesis)}
= (a_{n-1} + b_{n-1} + c_{n-1}) \cdot 2^{n-1} + \langle s[n-2 : 0] \rangle
= \langle c_n, s[n-1] \rangle \cdot 2^{n-1} + \langle s[n-2 : 0] \rangle \text{ (equation 1.1)}
= \langle c_n, s[n-1 : 0] \rangle \text{ (Lemma 9)}
\]

The following simple lemma allows to break the addition of two long numbers into two additions of shorter numbers. It can be used among other things for recursive constructions of adders and incrementers.

**Lemma 12.** For \( a, b \in \mathbb{B}^n \), for \( d, e \in \mathbb{B}^m \) and for \( c_0, c', c'' \in \mathbb{B} \) let

\[
\langle d \rangle + \langle e \rangle + c_0 = \langle c' \rangle m[n-1 : 0]
\]
\[
\langle a \rangle + \langle b \rangle + c' = \langle c'' \rangle s[n-1 : 0]
\]

then

\[
\langle ad \rangle + \langle be \rangle + c_0 = \langle c'' \rangle st
\]

Using repeatedly Lemma 9 we have

\[
\langle ad \rangle + \langle be \rangle + c_0
= \langle a \rangle \cdot 2^{n-1} + \langle d \rangle + \langle b \rangle \cdot 2^{n-1} + \langle e \rangle + c_0
= (\langle a \rangle + \langle b \rangle) \cdot 2^{n-1} + \langle c' \rangle t
= (\langle a \rangle + \langle b \rangle + c') \cdot 2^{n-1} + \langle t \rangle
= \langle c'' \rangle \cdot 2^{n-1} + \langle t \rangle
= \langle c'' \rangle st
\]

1.5 Two’s Complement Numbers

**Definition 7.** For strings \( a[n-1 : 0] \in \mathbb{B}^n \) we denote by

\[ [a] = -a_{n-1} \cdot 2^{n-1} + \langle a[n-2 : 0] \rangle \]

the integer with two’s complement representation \( a \). A two’s complement number is a string that is interpreted as two’s complement representation of an integer.
We denote by
\[ T_n = \{ \langle a \rangle \mid a \in B^n \} \]
the set of integers that have a two's complement representation of length \( n \).
As
\[
T_n = \{ \langle 0b \rangle \mid b \in B^{n-1} \} \cup \{ \langle 1b \rangle \mid b \in B^{n-1} \} \\
= B_{n-1} \cup \{ -2^{n-1} + x \mid x \in B_{n-1} \} \\
= \{ 0 : 2^{n-1} - 1 \} \cup \{ -2^{n-1} + x \mid x \in \{ 0 : 2^{n-1} - 1 \} \} \text{ (Lemma 8)}
\]
we have

**Lemma 13.**
\[ T_n = [-2^{n-1} : 2^{n-1} - 1] \]

By \( twoc_n(x) \) we denote the two's complement representation of \( x \in T_n \).
\[ twoc_n(x) = \{ a \mid a \in B^n \land \langle a \rangle = x \} \]

Several basic properties of two's complement numbers are summarized in

**Lemma 14.** Let \( a = a[n-1 : 0] \), then
\[
\begin{align*}
\langle 0a \rangle &= \langle a \rangle \quad \text{(embedding)} \\
\langle a \rangle &= \langle a \rangle \mod 2^n \\
\langle a \rangle < 0 &\iff a_{n-1} = 1 \quad \text{(sign bit)} \\
\langle a_{n-1}a \rangle &= \langle a \rangle \quad \text{(sign extension)} \\
\langle -a \rangle &= \langle a \rangle + 1
\end{align*}
\]

Proof: The first line is trivial. The second line follows from
\[
\langle a \rangle - \langle a \rangle = -a_{n-1} \cdot 2^{n-1} + \langle a[n-2 : 0] \rangle - (a_{n-1} \cdot 2^{n-1} + \langle a[n-2 : 0] \rangle) \\
= -a_{n-1} \cdot 2^n
\]

If \( a_{n-1} = 0 \) we have \( \langle a \rangle = \langle a[n-2 : 0] \rangle \geq 0 \). If \( a_{n-1} = 1 \) we have
\[
\begin{align*}
\langle a \rangle &= -2^{n-1} + \langle a[n-2 : 0] \rangle \\
&\leq -2^{n-1} + 2^{n-1} + 1 \quad \text{(lemma 8)} \\
&= -1
\end{align*}
\]

This shows the third line.
\[
\begin{align*}
\langle a_{n-1}a \rangle &= -a_{n-1} \cdot 2^n + \langle a[n-1 : 0] \rangle \\
&= -a_{n-1} \cdot 2^n + a_{n-1} \cdot 2^{n-1} + \langle a[n-1 : 0] \rangle \\
&= -a_{n-1} \cdot 2^{n-1} + \langle a[n-1 : 0] \rangle \\
&= \langle a \rangle
\end{align*}
\]
1.6. BOOLEAN ALGEBRA

This shows the fourth line. For the last line we observe \( \bar{x} = 1 - x \) for \( x \in \mathbb{B} \).

Then

\[
\overline{[a]} = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i
\]

\[
= -(1 - a_{n-1}) \cdot 2^{n-1} + \sum_{i=0}^{n-2} (1 - a_i) \cdot 2^i
\]

\[
= -2^{n-1} + \sum_{i=0}^{n-2} 2^i + a_{n-1} \cdot 2^{n-1} - \sum_{i=0}^{n-2} a_i \cdot 2^i
\]

\[
= -1 - [a] \quad \text{(lemma 6)}
\]

We finally state a subtraction algorithm for binary numbers

**Lemma 15.** Let \( a, b \in \mathbb{B}^n \). Then

- \( \langle a \rangle - \langle b \rangle \equiv \langle a \rangle - \langle \bar{b} \rangle + 1 \mod 2^n \)

- if \( \langle a \rangle - \langle b \rangle \geq 0 \) then

\[
\langle a \rangle - \langle b \rangle = \langle \langle a \rangle - \langle \bar{b} \rangle + 1 \mod 2^n \rangle
\]

**Proof:** by Lemma 14 we have

\[
\langle a \rangle - \langle b \rangle = \langle a \rangle - [0b]
\]

\[
= \langle a \rangle + [1\bar{b}] + 1
\]

\[
= \langle a \rangle - 2^n + \langle \bar{b} \rangle + 1
\]

\[
= \langle a \rangle + \langle \bar{b} \rangle + 1 \mod 2^n
\]

The extra hypothesis \( \langle a \rangle - \langle b \rangle \geq 0 \) implies

\[
\langle a \rangle - \langle b \rangle \in B_n
\]

The second claim now follows from Lemma 5

1.6 Boolean Algebra

We consider Boolean expressions with constants 0 and 1, variables \( x_0, x_1, \ldots, a, b, \ldots \) and function symbols \( -, \land, \lor, \oplus, f(\ldots), g(\ldots) \). Four of the function symbols have predefined semantics as specified in Table 1.3 In order to save brackets one uses the convention, that \( - \) binds stronger than \( \land \) and that \( \land \) binds stronger than \( \lor \). Thus \( \overline{x_1} \land x_2 \lor x_3 \) is an abbreviation:

\[
\overline{x_1} \land x_2 \lor x_3 = (\overline{x_1} \land x_2) \lor x_3
\]
We denote expressions $e$ depending on variables $x = x[1:n]$ by $e(x)$. Variables $x_i$ can take values in $\mathbb{B}$, thus $x = x[1:n]$ can take values in $\mathbb{B}^n$. Equations

$$e(x) = e'(x)$$

come in two flavors:

- **Identities.** For the purposes of this subsection we denote them by $e(x) \equiv e'(x)$. An equality holds, if expressions $e$ and $e'$ evaluate to the same value $\in \mathbb{B}$ for any substitution of the variables $a = a[1:n] \in \mathbb{B}^n$:

$$e(x) \equiv e'(x) \iff \forall a \in \mathbb{B}^n : e(a) = e'(a)$$

- **Equations which one wants to solve.** A substitution $a = a[1:n] \in \mathbb{B}^n$ solves equation $e(x) = e'(x)$ if $e(a) = e'(a)$

In Boolean Algebra there is a very simple connection between the solution of equations and equivalences. An equivalence $e(x) \equiv e'(x)$ holds iff equations $e(x) = 1$ and $e'(x) = 1$ have the same set of solutions

**Lemma 16.**

$$e(x) \equiv e'(x) \iff \forall a \in \mathbb{B}^n : (e(a) = 1 \iff e'(a) = 1)$$

**Proof:** The direction from left to right is trivial. For the other direction we distinguish cases:

- $e(a) = 1$. Then $e'(a) = 1$ by hypothesis

- $e(a) = 0$. Then $e'(a) = 1$ would by hypothesis imply the contradiction $e(a) = 1$. Because in Boolean Algebra $e'(a) \in \mathbb{B}$ we conclude $e'(a) = 0$

Thus we have $e(a) = e'(a)$ for all $a \in \mathbb{B}^n$.

**1.6.1 Identities**

Useful identities resp. laws are
1.6. BOOLEAN ALGEBRA

• commutativity:

\[ x_1 \land x_2 \equiv x_2 \land x_1 \]
\[ x_1 \lor x_2 \equiv x_2 \lor x_1 \]
\[ x_1 \oplus x_2 \equiv x_2 \oplus x_1 \]

• associativity:

\[ (x_1 \land x_2) \land x_3 \equiv x_1 \land (x_2 \land x_3) \]
\[ (x_1 \lor x_2) \lor x_3 \equiv x_1 \lor (x_2 \lor x_3) \]
\[ (x_1 \oplus x_2) \oplus x_3 \equiv x_1 \oplus (x_2 \oplus x_3) \]

• distributivity

\[ x_1 \land (x_2 \lor x_3) \equiv (x_1 \land x_2) \lor (x_1 \land x_3) \]
\[ x_1 \lor (x_2 \land x_3) \equiv (x_1 \lor x_2) \land (x_1 \lor x_3) \]

• identity

\[ x_1 \land 1 \equiv x_1 \]
\[ x_1 \lor 0 \equiv x_1 \]

• idempotence

\[ x_1 \land x_1 \equiv x_1 \]
\[ x_1 \lor x_1 \equiv x_1 \]

• annihilation

\[ x_1 \land 0 \equiv 0 \]
\[ x_1 \lor 1 \equiv 1 \]

• absorption

\[ x_1 \lor (x_1 \land x_2) \equiv x_1 \]
\[ x_1 \land (x_1 \lor x_2) \equiv x_1 \]

• complementation

\[ x_1 \land \overline{x_1} \equiv 0 \]
\[ x_1 \lor \overline{x_1} \equiv 1 \]
22  CHAPTER 1.  NUMBER FORMATS AND BOOLEAN ALGEBRA

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_1 \land x_2$</th>
<th>$\overline{x_1 \land x_2}$</th>
<th>$\overline{x_1}$</th>
<th>$\overline{x_2}$</th>
<th>$\overline{x_1 \lor x_2}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 0</td>
<td>1 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 1.4: Verifying the first of de Morgan’s laws

- double negation
  \[ \overline{\overline{x_1}} = x_1 \]

- de Morgan’s laws
  \[ \overline{x_1 \land x_2} = \overline{x_1} \lor \overline{x_2} \]
  \[ \overline{x_1 \lor x_2} = \overline{x_1} \land \overline{x_2} \]

Each of these identities can be proven in a brute force way: if the identity has $n$ variables, then for each of the $2^n$ possible substitutions of the variables the left and right hand sides of the identities are evaluated with the help of Table 1.3. If for each substitution the left hand side and the right hand side evaluate to the same value, then the identity holds. For the first of de Morgan’s laws this is illustrated in Table 1.4

1.6.2  Solving Equations

For $i = 1, 2, \ldots$ we consider expressions $e(x[1: n])$ and $e_i(x[1: n])$ involving a vector of variables $x = x[1: n]$ and derive three basic lemmas about the solution of Boolean equations. For $a \in \mathbb{B}$ we define

\[ e(x)^a = \begin{cases} 
    e(x) & a = 1 \\
    \overline{e(x)} & a = 0
\end{cases} \]

Inspection of the semantics of $-$ in Table 1.3 immediately gives

**Lemma 17.**

\[ e(x)^a = 1 \iff e(x) = a \]

Inspection of the semantics of $\land$ in table 1.3 gives

\[ e_1(x) \land e_2(x) = 1 \iff e_1(x) = 1 \land e_2(x) = 1 \]

Induction on $n$ gives

**Lemma 18.**

\[ \bigwedge_{i=1}^n e_i(x) = 1 \iff \forall i \in [1: n] : e_i(x) = 1 \]
1.6. BOOLEAN ALGEBRA

Inspection of the semantics of $\lor$ in table 1.3 gives

$$e_1(x) \lor e_2(x) = 1 \iff e_1(x) = 1 \lor e_2(x) = 1$$

Induction on $n$ gives

**Lemma 19.**

$$\bigvee_{i=1}^{n} e_i(x) = 1 \iff \exists i \in [1 : n] : e_i(x) = 1$$

### 1.6.3 Disjunctive Normal Form

**Definition 8.** Let $f : \mathbb{B}^n \rightarrow \mathbb{B}$ be a switching function, let $x = x[1 : n]$ and let $e(x)$ be a Boolean expression. We say that $e$ computes $f$ iff the identity $f(x) \equiv e(x)$ holds.

That every switching function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ is computed by some Boolean expression is asserted in

**Lemma 20.**

$$\forall f : \mathbb{B}^n \rightarrow \mathbb{B} \ \exists e(x[1 : n]) : f(x[1 : n]) \equiv e(x[1 : n])$$

**Proof:** Let $b \in B$ and let $x_i$ be a variable. We define the literal

$$x_i^b = \begin{cases} x_i & b = 1 \\ \overline{x_i} & b = 0 \end{cases}$$

Then by lemma 17

$$x_i^b = 1 \iff x_i = b \quad (1.3)$$

Let $a = a[1 : n]$ and let $x = x[1 : n]$ be a vector of variables. We define the monomial

$$m(a) = \bigwedge_{i=1}^{n} x_i^{a_i}$$

Then

$$m(a) = 1 \iff \forall i \in [1 : n] : x_i^{a_i} = 1 \quad (\text{Lemma 18})$$

$$\iff \forall i \in [1 : n] : x_i = a_i \quad (\text{eqn. 1.3})$$

$$\iff x = a$$

Thus

$$m(a) = 1 \iff x = a \quad (1.4)$$

We define the support $S(f)$ of $f$ as the set of arguments $a$, where $f$ takes the value $f(a) = 1$:

$$S(f) = \{ a \mid a \in \mathbb{B}^n \land f(a) \}$$
If the support is empty, then \( e = 0 \) computes \( f \) otherwise we set

\[
e(x) = \bigvee_{a \in S(f)} m(a)
\]

Then

\[
e(x) = 1 \iff \exists a \in S(f) : m(a) = 1 \quad \text{(Lemma 19)}
\]

\[
\iff \exists a \in S(f) : a = x \quad \text{(eqn. 1.4)}
\]

\[
\iff x \in S(f)
\]

\[
\iff f(x) = 1
\]

Thus the equation \( e(x) = 1 \) and \( f(x) = 1 \) have the same solutions and we conclude

\[
e(x) \equiv f(x)
\]

with Lemma 16.

The expression \( e(x) \) constructed in the proof of Lemma 20 is called the \textit{complete disjunctive normal form} of \( f \). The complete disjunctive normal forms of the sum and carry functions \( c' \) defined in table 1.2 are

\[
c'(a, b, c) \equiv \bar{a} \land b \land c \lor a \land \bar{b} \land c \lor a \land b \land \bar{c} \lor a \land b \land c
\]

\[
s(a, b, c) = \bar{a} \land \bar{b} \land \bar{c} \lor a \land b \land c \lor a \land b \land \bar{c} \lor a \land \bar{b} \land c
\]

In ordinary Algebra one often leaves out the multiplication sign in order to simplify notation. Recall that the first binomic formula is usually written as

\[
(a + b)^2 = a^2 + 2ab + b^2
\]

where \( 2ab \) is a shorthand for \( 2 \cdot a \cdot b \) in the same spirit one often leaves out the \&-sign in the conjunction of literals. Thus the above identities are also written as

\[
c'(a, b, c) \equiv \bar{a}bc \lor \bar{a}bc \lor ab\bar{c} \lor abc
\]

\[
s(a, b, c) = \bar{a}bc \lor \bar{a}bc \lor ab\bar{c} \lor abc
\]

Simpler expressions for the same functions are

\[
c'(a, b, c) \equiv ab \lor bc \lor ab \quad (1.5)
\]

\[
s(a, b, c) = a \oplus b \oplus c \quad (1.6)
\]

The correctness can be checked in the usual brute force way by trying all 8 assignments of values in \( \mathbb{B}^3 \) to the variables of the expressions. In the remainder of this text we return to the usual mathematical notation and use the equality sign for identities too. Whether we deal with identities or whether we solve equations will (hopefully) be clear from the context.
Chapter 2

Hardware

In a nutshell, hardware consists of three kinds of components which are interconnected by wires: gates, storage elements and drivers. Gates are: AND-gates, OR-gates, &-gates and inverters. In circuit schematics we use the symbols from Figure 1.

2.1 Gates and Circuits

A circuit $C$ consist of a finite set $G$ of gates, sequences of inputs $x[1 : n]$ and outputs $y[1 : t]$ and a set $N(C)$ of nets (of wires). Also special inputs 0 and 1 of the circuit are always available. The signals $\text{Sig}(C)$ of the circuit consist of the set inputs

$$\text{In} = \{x_1, \ldots, x_n, 0, 1\}$$

and of the (outputs of) the gates

$$\text{Sig}(c) = \text{In} \cup G$$

Certain signals are outputs

$$\forall i \in [1 : t] : z_i \in \text{Sig}(c)$$

Each gate $g \in G$ has one or two inputs $\text{in1}(g), \text{in2}(g)$ which are driven by signals of the circuit. If $g$ is an inverter then $\text{in2}(g)$ is not defined.

$$\text{in1}(g), \text{in2}(g) \in \text{Sig}(C)$$

At first glance is very easy to define how a circuit should work.

Definition 9. For substitutions $a = a[1 : n] \in \mathbb{B}^n$ we specify the value $y(a)$ assumed by signals $y$

1. : inputs $x_i$;

$$\forall i \in [1 : n] : x_i = a_i$$
2. **inverters** $g$:

$$g(a) = \overline{\text{in}1(g)(a)}$$

3. **o-gates with** $\circ \in \{\text{AND}, \text{OR}, \oplus\}$:

$$g(a) = \text{in}1(g)(a) \circ \text{in}2(g)(a)$$

Unfortunately, this is not always a definition. For a counterexample see Figure 2. Due to the cycle one cannot find an order, in which the above definition can be applied. Fortunately defining and then forbidding cycles solves the problem. A path from $y_1$ to $y_m$ in $G$ is a sequence of signals $(y[0 : m])$ such that for all $i < m$ we have

$$y_i = \text{in}1(y_{i+1}) \lor y_i = \text{in}2(y_{i+1})$$

The length $\ell(y[0 : m])$ of this path is

$$\ell(y[0 : m]) = m$$

The path is a **cycle** if $y_0 = y_m$. One requires circuits to be free of cycles and shows

**Lemma 21.** Every path in a circuit with set $G$ of gates has length at most $\#G$

Proof by contradiction: Assume a path $y[0 : k]$ with $k > \#G$ exists in the circuit. All $y_i$ are gates except possibly $y_0$ which might be an input. Thus a gate must be twice on the path:

$$\exists i, j : i < j \land y_i = y_j$$

Then $y[i : j]$ is a cycle.
2.2. SOME BASIC CIRCUITS

Because every path in a circuit has finite length one can define for each signal \( y \) the depth \( d(y) \) of \( y \) as the length of a longest path from an input to \( y \)

\[
d(y) = \max\{m : \exists \text{ path } y[0 : m], y_0 \in In \land y_m = y\}
\]

For later use we also define the length \( sp(y) \) of a shortest such path

\[
sp(y) = \min\{m : \exists \text{ path } y[0 : m], y_0 \in In \land y_m = y\}
\]

The definitions imply that \( d \) and \( s \) satisfy

\[
d(y) = \begin{cases} 
0 & \text{if } y \in In \\
(d(in1(y)) + 1 & \text{if } y \text{ is an inverter} \\
\max\{d(in1(y)), d(in2(y))\} + 1 & \text{otherwise}
\end{cases}
\]

\[
sp(y) = \begin{cases} 
0 & \text{if } y \in In \\
sp(in1(y)) + 1 & \text{if } y \text{ is an inverter} \\
\min\{sp(in1(y)), sp(in2(y))\} + 1 & \text{otherwise}
\end{cases}
\]

By straightforward induction one now obtains

**Lemma 22.** Let \( \text{depth}(y) = n \), then \( y(a) \) in definition 9 is well defined.

Proof by induction on \( n \). If \( n = 0 \), then \( y \) is an input and \( y(a) \) is clearly well defined by the first rule. If \( n > 0 \), then we have \( \text{depth}(in1(y)) < n \). If \( y \) is not an inverter, we also have \( \text{depth}(in2(y)) < n \). By induction hypothesis \( in1(y)(a) \) and - if \( y \) is not an inverter - \( in2(y)(a) \) are well defined. We now conclude, that \( y(a) \) is well defined by the second and third rule.

2.2 Some Basic Circuits

Boolean expressions can be translated into circuits in a very intuitive way. In Figure 3 b) we have translated the simple formulae 1.5 for \( c'(a, b, c) \) and \( s(a, b, c) \) into a circuit. With inputs \( (a, b, c) \) and outputs \( (c', s) \) this circuit satisfies

\[
\langle c', s \rangle = a + b + c
\]

A circuit satisfying this condition is called a full adder. We use for this circuit the symbol from Figure 3 a).

If the \( b \)-input of a half adder is known to be zero, the specification simplifies to

\[
\langle c', s \rangle = a + c
\]
a) Symbol

b) Implementation

Figure 3: Full adder

---

a) Symbol

b) Implementation

Figure 4: Half adder
2.2. SOME BASIC CIRCUITS

![Diagram](image)

a) Symbol  

b) Implementation

**Figure 5: Multiplexer**

![Diagram](image)

a) Symbol  

b) Implementation

**Figure 6: n-bit multiplexer**

The resulting circuit is called a half adder. Symbol and implementation are shown in figure 22. The circuit in Figure 5 b) is called a multiplexer or short: mux. Its inputs and outputs satisfy

\[
    z = \begin{cases} 
    x & s = 0 \\
    y & s = 1 
    \end{cases}
\]

For multiplexers we use the symbol from Figure 5 a)

The n-bit multiplexer or short n-mux in Figure 6 b) consists of n multiplexers with a common select signal s. Its inputs and outputs satisfy:

\[
    z[1:n] = \begin{cases} 
    x[1:n] & s = 0 \\
    y[1:n] & s = 1 
    \end{cases}
\]

For n-muxes we use the symbol from figure Figure 6 a).

Figure 7 a) shows the symbol for an n bit inverter. Its inputs and outputs satisfy

\[
    y[1:n] = \overline{x[1:n]}
\]

n bit inverters are simply realized by n separate inverters as shown in Figure 7 b)
CHAPTER 2. HARDWARE

\[ x[1 : n] \quad y[1 : n] \]
\[ \downarrow \quad \quad \downarrow \]
\[ n \quad n \]
\[ \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \ quasi
2.2. SOME BASIC CIRCUITS

![Diagram of basic circuits]

Figure 9: n-bit \( \circ \) tree of gates for \( \circ \in \{\land, \lor, \oplus\} \)

![Diagram of n-Zero circuit]

Figure 10: n-bit zero tester

The inputs \( a[1 : n] \) and outputs zero of an n-zero tester n-ZERO (figure 10 a)) satisfy

\[
\text{zero} \equiv a = 0^n \\
\text{nzero} \equiv a \neq 0^n
\]

An implementation uses

\[
nzero(a[1 : n]) = \bigwedge_{i=1}^{n} a_i, \quad \text{zero} = \overline{\text{nzero}}
\]

The inputs \( a[1 : n], b[1 : n] \) and output eq, neq of an n bit equality tester (figure 11 a)) satisfy

\[
eq \equiv a = b \\
\text{neq} \equiv a \neq b
\]

An implementation uses

\[
\text{neq}(a[1 : n]) = \text{nzero}(a[1 : n] \oplus b[1 : n]), \quad eq = \overline{\text{neq}}
\]
a) Symbol  
\[
\begin{array}{c}
\text{n-eq} \\
\text{eq} \quad \text{neq}
\end{array}
\]

b) Implementation  
\[
\begin{array}{c}
\text{n-Zero} \\
\text{eq} \quad \text{neq}
\end{array}
\]

Figure 11: n-bit equality tester

\[
x[n-1:k] = x[k-1:0]
\]

Figure 12: Implementation of an n-bit decoder

An \textit{n-decoder} is a circuit with inputs \(x[n-1:0]\) and outputs \(y[2^n-1:0]\) satisfying

\[
\forall i: y_i = 1 \iff \langle x \rangle = i
\]

A recursive construction with \(k = \lceil \frac{n}{2} \rceil\) is shown in Figure 12. For the correctness one argues in the induction step

\[
y[2^k \cdot i + j] = 1 \iff V[i] = 1 \land U[j] = 1 \quad \text{(construction)}
\]

\[
\iff \langle x[n-1:k] \rangle = i \land \langle x[k-1:0] \rangle = j \quad \text{(ind. hypothesis)}
\]

\[
\iff \langle x[n-1:k]|x[k-1:0] \rangle = 2^k \cdot i + j \quad \text{(lemma 9)}
\]

An \textit{n-halfdecoder} is a circuit with inputs \(x[n-1:0]\) and outputs \(y[2^n-1:0]\) satisfying

\[
y = 0^{2^n-(x)} 1(x)
\]

i.e. input \(x\) is interpreted as a binary number and decoded into a unary number. The remaining output bits are filled with zeros.

A recursive construction of \(n\)-half decoders is shown in figure 13.
2.3. Clocked Circuits

For the construction of $n$-half decoders from $(n - 1)$-half decoder we divide the index range into upper and lower half

$$L = [2^{n-1} - 1 : 0], \quad H = [2^{n-1} - 1 : 0]$$

Also we divide $x[n-1 : 0]$ into the leading bit $x_{n-1}$ and the low order bits

$$x' = x[n-2 : 0]$$

Then

$$Y[H] \circ Y[L] = x_{n-1} \wedge U[L] \circ (x_{n-1} \lor U(L))$$

$$= \begin{cases} 0^{2n-1} \circ 0^{2n-1} - (x')11' & x_{n-1} = 0 \\ 0^{2n-1} - (x')1(x') \circ 1^{2n-1} & x_{n-1} = 1 \end{cases}$$

$$= \begin{cases} 0^{2n} - (x')1(x') & x_{n-1} = 0 \\ 0^{2n} - (x_{n-1}x')1^{2n-1}(x') & x_{n-1} = 0 \end{cases}$$

$$= 0^{2n} - (x_{n-1})x')(x')$$

2.3 Clocked Circuits

Here we introduce two computational models in which processors are constructed and their correctness is proven. We begin with the usual digital model, where time is counted in hardware cycles and signals are binary. Then we present a more general model that is motivated by the data sheets of hardware manufacturers. There, time is real valued and signals may assume the digital values in $B$ as well as a third value $\Omega$. This model allows to argue about
• hardware with multiple clock domains, e.g. in the real time systems that control cars or airplanes [JSchmaltz FMCAD, CMüller-Paul CAV 2011]. Here this is not an issue

• the presence and absence of glitches. This is an issue in the construction of memory systems: accesses to dynamic RAM tend to take several hardware cycles and inputs have to be constant in the digital sense and free of glitches during this time. The latter requirement cannot be expressed in the usual digital model, and thus the lemmas establishing their absence in our construction would be isolated from the remainder of the theory without the detailed model,

We explain how timing analysis is performed in the detailed model and then show, that with proper timing analysis the digital model is an abstraction of the detailed model. Thus, in the end all constructions are correct in the detailed model. But where glitches don't matter - i.e. everywhere except the access to dynamic RAM - we can work in the usual and much more comfortable digital model.

2.3.1 Digital Clocked Circuits

A em digital clocked circuit has three components

• a special reset input. It assumes values in \(B\)

• a sequence \(x[1 : n]\) of 1 bit registers assuming values in \(B^n\). Each register \(x[i]\) has two inputs
  - a data input \(x_{\text{din}}[i]\)
  - a clock enable input \(x_{\text{ce}}[i]\)

The values of the registers are identical with the signals at their outputs.

• a circuit with inputs \(x[1 : n] \circ \text{reset}\) and output sequence \(x_{\text{din}}[1 : n] x_{\text{ce}}[1 : n]\)

The semantics of clocked circuits are very simple. A hardware configuration of a clocked circuit is a snapshot of the current values of the registers, but for instance after power up this value might be binary but unknown. For this situation we use the special value \(\Omega\). Thus we have

\[
a[1 : n] \in (B \cup \{\Omega\})^n
\]

A hardware computation is a sequence \(a^t\) of hardware configurations where

The next configuration \(a'[1 : n]\) is computed from the current configuration \(a[1 : n]\) and the reset signal by a next hardware configuration function

\[
a' = \delta_H(a, \text{reset})
\]
Cycles of hardware computations are counted by natural numbers \( t \in \mathbb{N} \). Signals \( y \) during cycle \( t \) are denoted by \( y^t \). The values of the \( \text{reset} \) signal are fixed. It is on in cycle 0 and off ever after

\[
\text{reset}^t = \begin{cases} 
1 & t = 0 \\
0 & t > 0 
\end{cases}
\]

At power up register values are binary but unknown.

\[
x[1:n]^0 = \Omega^n
\]

We abbreviate the set of values of the new ternary logic used by

\[
\mathbb{B}_n = \mathbb{B} \cup \{ \Omega \}
\]

Assume we are in configuration \( x[1:n] \in \mathbb{B}_n^n \). Then the current values \( y(x) \) of circuit signals are defined by the known circuit semantics

\[
y(x) = \begin{cases} 
in1(y(x)) & \text{if } y \text{ is an inverter} \\
in1(y(x)) \circ in2(y(x)) & \text{if } y \text{ is a } \circ \text{-gate}
\end{cases}
\]

Because input signals can have value \( \Omega \) we must extend the semantics for the basic Boolean functions by

\[
\begin{align*}
\overline{\Omega} & = \Omega \\
1 \land \Omega & = \Omega \land 1 = \Omega \\
0 \land \Omega & = \Omega \land 0 = 0 \\
1 \lor \Omega & = \Omega \lor 1 = 1 \\
0 \lor \Omega & = \Omega \lor 0 = \Omega \\
a \oplus \Omega & = \text{omega} \oplus a = \Omega
\end{align*}
\]

The new value \( x'[i] \) of register \( x[i] \) is defined by

\[
x'[i] = \begin{cases} 
xin[i] & xce[i] = 1 \\
x[i] & xce[i] = 0 \\
\Omega & xce[i] = \Omega
\end{cases}
\]

Hardware computations are sequences \( x^t[1:n] \) of configurations satisfying

\[
\forall t : x^{t+1} = \delta_H(x^t, \text{reset}^t)
\]

We define the value \( y^t \) of arbitrary circuit signals during cycle \( t \) by

\[
y^t = y(x^t)
\]
As an example consider figure 15. There is only one register, thus we abbreviate
\[ x = x[0] \]

For cycle 0 we have
\[
\begin{align*}
x^0 &= \Omega \\
reset^0 &= 1 \\
xin^0 &= 1 \\
xce^0 &= 1
\end{align*}
\]
Hence
\[ x^1 = 1 \]

For cycles \( t > 0 \) we have
\[
\begin{align*}
reset^t &= 0 \\
xin^t &= y^t = \overline{x^t} \\
xce^t &= 1
\end{align*}
\]
Hence
\[ x^{t-1} = \overline{x^t} \]

An easy induction on \( t \) shows
\[ x^t = \{t \mod 2\} \]

### 2.3.2 The detailed hardware model

→ Time is real valued. Circuit signals \( y \) (which include register values resp. register outputs) are functions
\[ y : IR \rightarrow \{0, 1, \Omega\} \]

The circuit clock has two parameters
- the position \( \gamma \) of clock edge 0
- the cycle time \( \tau \). For \( c \in \mathbb{N} \) this defines the position \( e(c) \) of clock edge 0 as
\[ e(c) = \gamma + c \cdot \tau \]

Inspired by data sheets register have and gates six timing parameters
- \( \rho \): the minimal propagation delay of register outputs after clock edges
2.3. CLOCKED CIRCUITS

Figure 14: Detailed timing of a register \( x[i] \) with stable inputs and \( ce = 1 \)

- \( \sigma \): the maximal propagation delays of register outputs after clock edges. We require \( 0 \leq \rho < \sigma \).

- \( ts \): setup time of register input and clock enable before clock edges

- \( th \): hold time of register input and clock enable after clock edges

- \( \alpha \): minimal propagation delay of gates

- \( \beta \): maximal propagation delay of gates. We require \( 0 < \alpha < \beta \). \(^1\)

This is a simplification. Setup and hold times can be different for register inputs and clock enable. Also the propagation delays of different types of gates are in general different. Generalizing our model to this situation is very easy but requires more tedious notation.

Let \( y \) be any signal. The requirements that this signal satisfies the setup and hold times of registers at clock edge \( c \) is defined by

\[
\text{stable}(y, c) \leftrightarrow \exists a \in B : \forall t \in [e(c) - ts, e(c) + th] : y(t) = a
\]

The behavior of a register \( x[i] \) with stable input and clock enable at edge \( t \) is illustrated in figure 14.

For \( c \in \mathbb{N} \) and \( t \in (e(c) + \rho, e(c + 1) + \rho) \) we define the register value \( x[i](t) \) and output at time \( t \) by a case distinction:

- regular clocking in of data at edges \( c \geq 0 \) The setup and hold time for the input and clock enable signal are met: \( \text{stable}(xin[i], c) \land \text{stable}(xce[i](t)) \). The clock enable signal is 1 during this period. Then the data input at edge \( e(c) \) becomes the new value of the register, and it becomes visible (at the latest) at time \( \sigma \) after clock edge \( e(c) \).

\(^1\)Defining such delays from voltage levels of electrical signals is nontrivial and can go wrong in subtle ways. For the deduction of a negative propagation delay from the data of a very serious hardware catalogue see [Keller Paul], page...
regular non clocking in of data at edges \( c > 0 \). The output stays unchanged for the entire period

- any other situation, where the voltage cannot be guaranteed to be recognized as a known logical 0 or 1. This includes i) initialization \((c = 0)\), ii) the transition period from \( \rho \) to \( \sigma \) after regular clocking and ii) the entire time interval if there was a violation of the stability conditions of any kind. Usually a physical register will settle in this situation quickly into an unknown logical value, but in rare occasions the register can 'hang' at a voltage level not recognized as 0 or 1 for a long time. This is called ringing or metastability.

\[
x[i](t) = \begin{cases} 
\text{zin}[i](e(c)) & t \in [e(c) + \sigma, e(c + 1) + \rho] \land \text{stable}(\text{zin}[i], c) \\
\wedge \text{stable}(\text{xce}[i](t)) \land \text{xce}[i] = 1 \\
x[i](e(c)) & t \in (e(c) + \rho) \land \text{stable}(\text{xce}[i](t)) \\
\wedge \text{xce}[i] = 0 \\
\Omega & \text{otherwise}
\end{cases}
\]

Notice that during regular clocking in, the output is unknown between \( e(c) + \rho \) and \( e(c) + \sigma \). This is the case even if \( \text{zin}[i](e(c)) = x[i](e(c)) \); i.e. if we clock the value into the register that is already there. If the voltage reaches during that period a value not recognized as 0 or 1 this is called a glitch. The only way we have to guarantee constant register outputs during a time period is not to clock the register during that time.

**Lemma 23.** Assume data are regularly clocked into register \( x[i] \) at edge \( e(c) \):

\[
\text{stable}(\text{zin}[i], c) \land \text{stable}(\text{xce}[i], c) \land \text{xce}[i] = 1
\]

Assume further that the register is in a regular way not clocked at the following clock \( K - 1 \) clock edges edges

\[
\forall k \in [1 : K - 1] : \text{stable}(\text{xce}[i](c + k) \land \text{xce}[i] = 0
\]

Then the value \( x[i](e(c)) \) is visible at the output of register \( x[i] \) from time \( e(c) + \sigma \) to \( e(c + K) + \rho \)

\[
\forall t \in [e(c) + \sigma, e(c + K) + \rho] : x[i](t) = xin[i](e(c))
\]

Proof: one shows by an easy induction on \( k \)

\[
\forall t \in [e(c) + \sigma, e(c + 1) + \rho] : x[i](t) = xin[i](e(c))
\]

and

\[
\forall k \in [1 : K - 1] \forall t \in (e(c + k) + \rho, e(c + k + 1) + \rho] : x[i](t) = xin[i](e(c))
\]
2.3. CLOCKED CIRCUITS

Figure 15: Detailed timing of a gate $y$ with two inputs

We require the reset signal to behave like the output of a register, that is always clocked in a regular way

$$\forall c : \exists a \in B : \forall t \in [e(c) + \sigma, e(c + 1) + \rho] : reset(t) = a$$

It assumes value 1 at edge 0 and value 0 otherwise

$$reset(e(c)) = \begin{cases} 1 & c = 0 \\ 0 & c > 0 \end{cases}$$

For the definition of the value $y(t)$ of gates $g$ at time $t$ in the detailed model we distinguish three cases (see figure 15)

- regular signal propagation, where all input signals are binary and stable for the maximal propagation delay $\beta$ before $t$. For inverters $y$ this is captured by the following predicate

$$reg(y, t) \leftrightarrow \exists a \in B : \forall t' \in [t - \beta, t] : in1(y)(t') = a$$

The $y$ outputs $\overline{a}$ at time $t$. For other $o$-gates $y$ we define

$$reg(y, t) \leftrightarrow \exists a, b \in B : \forall t' \in [t - \beta, t] : in1(y)(t') = a \land in2(y)(t') = b$$

Then $y$ outputs $a \circ b$ at time $t$

- signal holding, where signal propagation is not regular any more, but it was regular until at most than the minimal propagation delay $\alpha$ before $t$.

$$hold(y, t) \leftrightarrow \neg reg(y, t) \land \exists t' \in [t - \alpha, t] : reg(y, t')$$

Then $y$ should still hold the old value $y(t')$ at time $t$; we will show that this is well defined for all $t'$.

- in all other cases we cannot give guarantees about $y(t)$
Lemma 24. Assume \( \text{hold}(y, t) \) and \( t_1, t_2 \in [t - \alpha, t] \land \text{reg}(y, t_1) \land \text{reg}(y, t_2) \). Then for all inputs \( z \) of \( y \) we have 
\[
y(t_1) = y(t_2)
\]

The proof is illustrated in figure x:
Without loss of generality we have \( t_1 < t_2 \). Let \( z \in \text{in1}(y), \text{in2}(y) \) be any input of \( y \). From \( \text{reg}(y, t_1) \) we infer 
\[
\exists a \in \mathbb{B} : \forall t' \in [t_1 - \beta, t_1] : z(t') = z(t_1) = a
\]
From 
\[
0 < t_2 - t_1 < \alpha < \beta
\]
we infer 
\[
t_2 - \beta < t_1 < t_2
\]
Thus 
\[
t_1 \in [t_2 - \beta, t_2]
\]
and hence 
\[
z(t_2) = z(t_1) = a
\]
For 2 input gates \( y \) we have 
\[
y(t_1) = \text{in1}(t_1) \circ \text{in2}(t_1) = \text{in1}(t_2) \circ \text{in2}(t_2) = y(t_2)
\]
For inverters the argument is equally simple.
2.3. CLOCKED CIRCUITS

For values $t$ satisfying $hold(y, t)$ we define $lreg(y, t)$ as the last value $t'$ before $t$ when signal propagation was regular

$$lreg(y, t) = \max\{t' : t' < t \land reg(y, t')\}$$

Now we can complete the definition of values $y(t)$

$$y(t) =
\begin{cases}
  in1(y)(t), & \text{reg}(y, t) \land y \text{ is an interer} \\
  in1(y)(t) \circ in2(y)(t), & \text{reg}(y, t) \land y \text{ is an } \circ \text{-gate} \\
  y(lreg(y, t)), & \text{hold}(y, t) \\
  \Omega, & \text{otherwise}
\end{cases}$$

2.3.3 Timing Analysis

Timing analysis is performed in the detailed model in order to ensure, that all register inputs $xin[i]$ and clock enables $xce[i]$ are regular at clock edges. We capture the conditions for correct timing by

$$\forall i, c : \text{reg}(xin[i], c) \land \text{reg}(xce[i], c)$$

After a reminder that $d(y)$ and $sp(y)$ where the length of longest and shortest paths from the inputs to $twe$ define the propagation delay of arbitrary signals $y$ relative to the clock edges

$$t_{min}(y) = \rho + d(y) \cdot \alpha$$
$$t_{max}(y) = \sigma + sp(y) \cdot \beta$$

In what follows we define a sufficient condition for correct timing and show, that with this condition detailed and digital circuits simulate each other in the sense that for all signals $y$ the value $y^c$ in the digital model during cycle $c$ equals the value $y(e(c+1))$ at the end of the cycle. I.e. with correct timing the digital model is an abstraction of the detailed model.

$$y^c = y(e(c + 1))$$

**Lemma 25.** If for signals $y$ we have

$$\forall y : th \leq t_{min}(y) \land t_{max}(y) + ts \leq \tau$$

and if for inputs $xin[i]$ of registers we have

$$\forall xin[i] : th \leq t_{min}(xin[i])$$

then

1. $$\forall y, c : \forall t \in [e(c) + t_{max}(y), e(c + 1) + t_{min}(y)] : y(t) = y^c$$
\[2.\]
\[
\forall i, c : stable(x[i], c) \land stable(xce[i], c)
\]

Proof by induction on \(c\). For each \(c\) we show the theorem by induction on the depth \(d(y)\) of signals.

For \(c = 0\) and the registers \(x[i]\) with \(d(x[i]) = 0\) we have

\[tmin(x[i]) = \rho \land tmax(x[i]) = \sigma\]

From the initialization rules in the digital and detailed model we get for

\[\forall t \in (e(0) + \rho, e(1) + \sigma) : x[i](t) = \Omega = x[i]^0\]

For \(c \geq 0\) assume

\[\forall t \in (e(c) + \sigma, e(c + 1) + \rho) : x[i](t) = x[i]^c\]

i.e. we have statement 1 for all signals \(y\) with depth 0. Assume it holds for signals of depth \(d - 1\) and let \(y\) be a \(o\)-gate of depth \(d\). We show that it holds for \(y\). Consider figure 17

There are inputs \(z_1, z_2\) of \(y\) such that

\[d(y) = d(z_1) + 1 \land sp(z) = sp(z_2) + 1\]

Hence

\[tmax(y) = tmax(z_1) + \beta\]
\[tmin(y) = tmin(z_2) + \alpha\]

By induction we have for all inputs \(z\) of \(y\)

\[\forall t \in [e(c) + tmax(z), e(c + 1) + tmin(z)] : z(t) = z^c\]
2.3. Clocked Circuits

Because
\[ t_{\min}(z_2) \leq t_{\min}(z) \wedge t_{\max}(z) \leq t_{\max}(z_1) \]
we get
\[ [e(c) + t_{\max}(y) - \beta, e(c + 1) + t_{\min}(y) - \alpha] \]
\[ = [e(c) + t_{\max}(z_1), e(c + 1) + t_{\min}(z_2)] \]
\[ \subseteq [e(c) + t_{\max}(z), e(c + 1) + t_{\min}(z)] \]
and thus
\[ \forall t \in [e(c) + t_{\max}(y) - \beta, e(c + 1) + t_{\min}(y) - \alpha]: z(t) = z^c \]
We conclude
\[ \forall t \in [e(c) + t_{\max}(y), e(c + 1) + t_{\min}(y) - \alpha]: \text{reg}(y, t) \]
and
\[ y(t) = \text{in1}(y)(t) \circ \text{in2}(y)(t) \]
\[ = \text{in1}(y)^c \circ \text{in2}(y)^c \] (ind. hypothesis)
\[ = y^c \]
We also get
\[ \forall t \in (e(c) + t_{\min}(y) - \alpha, e(c + 1) + t_{\min}(y))]: \text{hold}(y, t) \]
with
\[ l_{\text{reg}}(y(t)) = e(c + 1) + t_{\min}(y) - \alpha \]
We conclude from above
\[ y(t) = y(e(c + 1) + t_{\min}(y) - \alpha) \]
\[ = y^c \]
This shows that part 1 of the induction step holds for \( c \) and all \( y \) if it holds for \( c \) and the registers, i.e. the signals with \( d(y) = 0 \).
Consider figure 18.
For register inputs \( y = \text{xin}[i] \) we have shown
\[ \forall t \in [e(c) + t_{\max}(y), e(c + 1) + t_{\min}(y)]: y(t) = y^c \]
From the lemmas hypothesis
\[ \forall y : th \leq t_{\min}(y) \wedge t_{\max}(y) + ts \leq \tau \]
we get
\[
\begin{align*}
  e(c) + t_{\text{max}}(y) & \leq e(c) + \tau - ts \\
  & = e(c + 1) - ts \\
  e(c + 1) + t_{\text{min}} & \geq e(c + 1) + th
\end{align*}
\]
Thus
\[
e(c + 1) - ts, e(c + 1) + th \subseteq [e(c) + t_{\text{max}}(y), e(c + 1) + t_{\text{min}}(y)) : y(t) = y^c
\]
We conclude \(reg(y, c + 1)\). This shows part 2 for \(c\) if part 1 holds for the registers.
and hence for all \(t \in [e(c + 1) + \sigma, e(c + 2) + \rho]\):
\[
x[i](t) = \begin{cases} 
  xin[i](e(c + 1)) & xce[i](e(c + 1)) = 1 \\
  x[i](e(c + 1)) & xce[i](e(c + 1)) = 0 
\end{cases}
\]
\[
= \begin{cases} 
  xin[i]^c & xce[i]^c = 1 \\
  x[i]^c & xce[i]^c = 0 
\end{cases}
= x[i]^{c+1}
\]
This shows part 1 for \(c+1\) and the registers if it holds for \(c\) and the registers.

\section{2.4 Registers}

So far we have shown that there is one basic hardware model, namely the detailed one, but with correct timing it can be abstracted to the digital model (Lemma 26). From now on we assume correct timing and stick to the usual digital model unless we need to prove properties not expressible in this model like the absence of glitches.
2.5. **DRivers AND MAIN MEMORY**

![Diagram](image)

Figure 19: n-bit register

Although all memory components can be built from 1 bit registers, it is inconvenient to refer to all memory bits in a computer by numbering them with an index $i$ of a clocked circuit input $x[i]$. It is more convenient to deal with hardware configurations $h$ and to gather groups of such bits into certain memory components $h.M$. For $M$ we introduce here $n$-bit registers $h.R$. In chapter 3 we add to this no less than 9 (nine) random access memory (RAM) designs. As before we write for the next hardware configuration $h'$:

$$h' = \delta_H(h, \text{reset})$$

An $n$-bit register $R$ consists simply of $n$ many 1 bit registers $R[i]$ with a common clock enable signal $Rce$ as shown in figure 19.

Register configurations are $n$-**tuples**

$$h.R \in \mathbb{B}_n^n$$

With inputs signals $Rin(h)$ and $Rce(h)$ we can derive from the semantics of the basic clocked circuit model

$$h'.R = \begin{cases} Rin(h) & Rce(h) = 1 \\ h.R & Rce(h) = 0 \end{cases}$$

Recall from the initialization rules also that after power up register content is binary but unknown (metastability is extremely rare)

$$h^0.R = \Omega^n$$

### 2.5 Drivers and Main Memory

In order to deal with main memory and its connection to caches and processor cores we introduce several new hardware components: tri state drivers, open collector drivers and main memory. For hardware only consisting of gates, inverters and registers we have shown in lemma x that a design that works in the digital model also works in the detailed hardware model. For Tri state drivers and main memory this will not be the case any more.
2.5.1 Open collector drivers and active low signal

A single open collector driver $y$ and its detailed timing is shown in figure 20. Viewed in isolation such a driver with input $y_{in}$ simply computes the identity function as long as it is not switching. For the propagation delay we use the same parameters $\alpha$ and $\beta$ as for gates. A situation with regular signal propagations is defined as for inverters:

$$reg(y, t) \iff \exists a \in B : \forall t \in [t - \alpha, t]: y_{in}(t) = a$$

The signal $y$ generated by a single open collector driver is then defined as

$$y(t) = \begin{cases} y_{in}(t) & \text{if } \text{reg}(y, t) \\ y(t_{reg}(y, t)) & \text{if } \text{hold}(y, t) \\ \Omega & \text{otherwise} \end{cases}$$

In contrast to other gates it is allowed to connect the outputs of drivers by wires which are often called buses. Figure 21 shows $n$ open collector drivers $y_i$ with inputs $y_{in}$ driving a bus $b$. The rule for determining the bus value $b(t)$ from the driver values $y_i(t)$ is simple: compute the AND of the $y_i(t)$ using the rules introduced earlier

$$\text{bus}(t) = \bigwedge_i y_i(t)$$

with

$$\Omega \land x = \begin{cases} 0 & \text{if } x = 0 \\ \Omega & \text{otherwise} \end{cases}$$

In the digital model we simply have

$$\text{bus}^t = \bigwedge_i y_i^t$$

but this abstracts away an important detail: glitches on a driver input can propagate to the bus, for instance when the signals of other drivers are 1.
2.5. DRIVERS AND MAIN MEMORY

\[
y_{i1} \quad \ldots \quad y_{ik}
\]

\[
\begin{align*}
\text{OC} & \quad \text{OC} \\
y_i & \quad y_k
\end{align*}
\]

Figure 21: Open collector drivers \( y_i \) connected by a bus \( b \)

This will not be an issue for the open collector busses constructed here. It is, however, an issue in the control of real time busses [MP 2011].

By de Morgan’s law one can use open collector buses together with some inverters to compute the logical OR of signals \( u_i \):

\[
b = \bigwedge_i \lnot u_i = \bigvee_i u_i
\]

(2.1)

In control logic it often equally easy to generate or use an 'active high' signal \( u \) or its inverted 'active low' version \( /u \). By equation 2.1 open collector buses compute an active low OR \( /b \) of control signals \( u_i \) without extra cost, if the active low versions \( /u_i \) are available.

\( n \)-bit open collector drivers are simply \( n \) open collector drivers in parallel. Symbol and construction are shown in figure 3.

2.5.2 Tri state drivers and bus contention

Tri state drivers \( y \) are controlled by output enable signals \( yoe \). Scheme and timing are shown in figure 22. Only with active output enable signals does a tristate driver propagate the data input \( yin \) to the output \( y \). Like ordinary switching, enabling and disabling drivers involves propagation delays. In detailed timing diagrams an undefined value due to disabled outputs is usually drawn as a horizontal line in the middle between 0 and 1. In the

\[\text{\footnote{Like clock enable signals we model them as active high, but in data sheets for real hardware components they are usually active low.}}\]
The jargon of hardware designers this is called the high impedance state or high Z or simply Z. In order to specify behavior and operating conditions of tri
state drivers we have to permit Z as a signal value for tri state drivers y, so we have

\[ y : \mathbb{R} \rightarrow \{0, 1, \Omega, Z\} \]

Ignoring propagation delays a tri state driver then computes the following function

\[ tr(in, oe) = \begin{cases} in & \text{if } oe = 1 \\ Z & \text{if } oe = 0 \end{cases} \]

For simplicity we use the same timing parameters as for gates. Regular signal propagation is defined as for gates:

\[ reg(y, t) \leftrightarrow \exists a, b \in B : \forall t \in [t - \alpha, t] : yin(t) = a \land yoe(t) = b \]

The signal y generated by a single open collector driver is then defined as

\[ y(t) = \begin{cases} tr(yin(t), yoe(t)) & \text{if } reg(y, t) \\ y(tr(y, y)) & \text{if } hold(y, t) \\ \Omega & \text{otherwise} \end{cases} \]

Observe that a glitch on an output enable signal can produce a glitch in signal y. In contrast to glitches on open collector busses this will be an issue in our designs involving main memory.

As open collector drivers the outputs of tri state drivers can be connected via so called tri state busses. The clean way to operate tri state busses b with drivers y, as shown in figure 23 is to allow at any time t at most one driver to produce a signal different from Z.

\[ y_i(t) \neq Z \land y_j(t) \neq Z \Rightarrow i = j \] (2.2)

If this invariant is maintained the following definition of the bus value b(t) at time t is well defined
2.5. **DRIVERS AND MAIN MEMORY**

![Diagram of circuit with enable signals](image)

Figure 24: Switching enable signals of drivers at the same clock edge

![Timing diagram](image)

Figure 25: Possible timing when enable signals are switched at the same clock edge

\[ b(t) = \begin{cases} y_i(t) & y_i(t) \neq Z \\ Z & \text{otherwise} \end{cases} \]

The invariant excludes a design like figure 24, where drivers \( y_0 \) and \( y_1 \) are switched on and off at the same clock edge.\(^3\) In order to understand the possible problem with such a design consider a rising clock edge when \( R_0 = y_0oe \) is turned on and \( R_1 = y_1oe \) is turned off. This can lead to a situation as shown in figure 25.

There, we assume that the propagation delay of \( R_0 \) is \( \rho = 1 \) and the propagation delay of \( R_1 \) is \( \sigma = 2 \). Similarly, assume that the enable time of \( y_0 \) is \( \alpha = 1 \) and the disable time of \( y_1 \) is \( \beta = 2 \). The resulting signals at a rising edge of clock \( ck \) are shown in the detailed timing diagram in figure y. Note that for \( 2 \leq t \leq 4 \) we have \( y_0(t) = 0 \) and \( y_1(t) = 1 \). This happens to produce more problems than just a temporarily undefined bus value.

\(^3\)This is not unheard of in practice
The output circuitry of a driver or gate can be envisioned as a pair of adjustable resistors as shown in figure 26. Resistor $R_1$ is between the supply voltage $VCC$ and the drivers output $y$. The other resistor $R_2$ is between the output and ground $GND$. Logical values 0 and 1 as well as and high impedance state $Z$ can now be realized by adjusting the values of the resistors as shown in table 2.1.2

Of course the circuitry of a well designed single driver will never produce a short circuit by adjusting both resistors to 'low'. However, as shown in figure 27 the short circuit is still possible via the low resistance path

\[
GND - y_0 - b - y_1 - VCC
\]

if two drivers are simultaneously enabled and one of the drivers drives 0 whereas the other driver drives 1. Exactly this situation occurs temporarily in the real valued time interval $[r+2, r+3]$ after each rising clock edge $r$. In the jargon of hardware designers this is called - temporary - *bus contention*, which clearly sounds much better than 'temporary short circuit'. But even with the nicer name it remains of course a short circuit. In the best case it increases power consumption and shortens the life time of the driver. The spikes in power consumption can have the side effect that power supply voltage falls under specified levels; maybe not always but sporadically when power consumption in other parts of the hardware is high. Insufficient supply voltage then will tend to produce sporadic non reproducible failures in other parts of the hardware.
2.5. DRIVERS AND MAIN MEMORY

\[
\begin{align*}
\text{VCC} & \quad y_0 \quad b \quad y_1 \\
\text{GND} & \quad \text{low} \quad \text{high} \\
\end{align*}
\]

Figure 27: Short circuit via the bus \( b \) when two drivers are enabled at the same time

2.5.3 The incomplete digital model for drivers

Observe that there is a deceptively natural looking digital model of tri state drivers which has a good and a bad part. The good part is

\[
y = \begin{cases} 
y_{\text{in}}(y) & \text{yoe} = 1 \\
Z & \text{otherwise}
\end{cases} \tag{2.3}
\]

The bad part - as we will demonstrate later - is the very natural looking condition

\[
y^i \neq Z \land y^j \neq Z \rightarrow i = j \tag{2.4}
\]

The good part, i.e. equation 2.3 correctly models the behavior of drivers for times after clock edges where all propagation delays have occurred and when registers are updated. Indeed, if we consider a bus \( b \) driven by drivers \( y_i \) as a gate with depth

\[
d(b) = \max_i d(y_i)
\]

we can immediately extend lemma 26 to circuits with busses and drivers of both kinds

**Lemma 26.** Assume that invariant 2.2 holds for all tri state buses and assume

\[
\forall y : th \leq t_{\text{min}}(y) \land t_{\text{max}}(y) + ts \leq \tau
\]

then

1. \[\forall y, c : \forall t \in [e(c) + t_{\text{max}}(y), e(c + 1) + t_{\text{min}}(y)] : y(t) = y^c\]

2. \[\forall i, c : \text{reg}(x_{\text{in}}[i], c) \land \text{reg}(x_{\text{ce}}[i], c)\]

This justifies the use of the digital model as far as register update is concerned. It has however a hypothesis coming from the detailed model.
Figure 28: Generating a pulse of arbitrary width by a sufficiently long delay line

Replacing it simply by what we call the bad part of the digital model, i.e. invariant 2.4, is the highway to big trouble. First of all observe that our design in figure 25, which switched enable signals at the same clock edge, satisfies it. But in the detailed model (and the real world) we can do worse. We will construct hardware that destroys itself by the short circuits caused by bus contention but which is contention free according to the (bad part of) the digital model.

2.5.4 Self destructing hardware

In what follows we will do some arithmetic on time intervals \([a, b]\) where signals change. In our computations of these time bounds we use the following rules

\[ c + [a, b] = [c + a, c + b] \]
\[ c \cdot [a, b] = [c \cdot a, c \cdot b] \]
\[ [a, b] + [c, d] = [a + c, b + d] \]

**Lemma 27.** For any \( \epsilon > 0 \) there is a design satisfying invariant 2.4 which produces continuous bus contention for at least a fraction \( \alpha/\beta - \epsilon \) of the total time.

For common technologies the fraction \( \alpha/\beta \) is around 1/3. Thus we are talking about a short circuit for roughly 1/3 of the time. We should mention that this will overheat drivers to an extent that the packages of the chips tend to explode.

Proof: the key to the construction is the parameterized design of figure 28. The timing diagram in figure 2 shows, that the entire design produces a pulse of length growing with \( c \); hence we call it a \( c \)-pulse generator.

Signal \( u \) goes up at time \( t \). The chain of \( c \) AND gates just serves as a delay line. The result is finally inverted. Thus signal \( u' \) falls in time interval with

\[ t_1 = t + (c + 1) \cdot [\alpha, \beta] \]

The final AND gate produces a pulse \( v \) with a rise time in interval \( t_2 \) and a fall time in interval \( t_3 \) satisfying

\[ t_2 = t + [\alpha, \beta] \]
\[ t_3 = t + (c + 2) \cdot [\alpha, \beta] \]
2.5. **Drivers and Main Memory**

![Image](image.png)

Figure 29: Timing diagram for a pulse generator

![Diagram](diagram.png)

Figure 30: Generating contention with two pulse generators

Note that in the digital model we have for all cycles \( t \)

\[
v^t = u^t \land \lnot u^t = 0
\]

which is indeed correct after propagation delays are over, and that is all the digital model captures. Now consider the design in figure 30. In the digital model \( v_1 \) and \( v_2 \) are always zero. The only driver ever enabled in the digital model is \( y_3 \). Thus the design satisfies the digital invariant 2.4.

Now consider the timing diagram in figure 31. At each clock edge \( T \), one of registers \( R_i \) has arising edge in time interval

\[
t = T + [\rho, \sigma]
\]

which generates a pulse with rising edge in time interval \( T_2 \) and falling edge in time interval \( T_3 \) satisfying

\[
t_2 = T + [\rho, \sigma] + [\alpha, \beta]
\]

\[
t_3 = T + [\rho, \sigma] + (c + 2) \cdot [\alpha, \beta]
\]
Driver $y_t$ then enables in time interval $t_4$ and disables in time interval $t_5$ satisfying

$$t_4 = T + |\rho, \sigma| + 2 \cdot [\alpha, \beta]$$
$$t_5 = T + |\rho, \sigma| + (c + 3) \cdot [\alpha, \beta]$$

We choose a cycle time

$$\tau(c) = \rho + (c + 3) \cdot \beta$$

so that the timing diagram fits exactly into one clock cycle. In the next cycle we then have the same situation for the other Register and driver. We have contention on bus $b$ at least during time interval $C = T + |\sigma + 2 \cdot \beta, \rho + (c + 3) \cdot \alpha|$ of length

$$\ell(c) = \rho + (c + 3) \cdot \alpha - (\sigma + 2 \cdot \beta)$$

Asymptotically we have

$$\ell(c)/\tau(c) \rightarrow_c \alpha/\beta$$

Thus we choose $c$ such that

$$\ell(c)/\tau(c) \geq \alpha/\beta - \epsilon$$

and the lemma follows.

### 2.5.5 Clean operation of tri state buses

We now construct control logic for tri state buses. We begin with a digital specification, construct a control logic satisfying this (incomplete) specification and then show in the detailed model i) that the bus is free of contention and ii) that signals are free of glitches while we guarantee their presence on the bus.

As a building block of the control we use the set-clear-flip flops whose symbol is shown in figure 32 a) a), and whose implementation is shown in
2.5. DRIVERS AND MAIN MEMORY

![Diagram of a set clear flip flop](image)

a) Symbol  

b) Implementation

Figure 32: Symbol and implementation of a set clear flip flop

![Diagram of registers connected to a bus](image)

Figure 33: Registers $R_j$ connected to a bus $b$ by tri state drivers $y_j$

figure 32 a) b). This is simply a 1 bit register which is set to 1 by activation of the set signal and to 0 by activation of the clr signal (without activation of the set signal). During reset, i.e. during cycle 0, the flip flops are forced to zero.

$$R^1 = 0$$

$$Rt + 1 = \begin{cases} 
1 & \text{set}^t \\
0 & \text{set}^t \land \text{clr}^t \\
R_t & \text{otherwise} 
\end{cases}$$

We consider a situation as shown in figure 33 with registers $R_j$ connected to a bus $b$ by tri state drivers $y_j$ for $j \in [0 : k - 1]$.

For $i \in \mathbb{N}$ we aim at intervals

$$T_i = [a_i : b_i]$$

of cycles and a function

$$send : \mathbb{N} \rightarrow [0 : k - 1]$$
specifying for each $i \in \mathbb{N}$ the unique index $j = s(i)$ such that $R_j$ is 'sending' on the bus during 'time' interval $T_i$:

$$y^t = \begin{cases} R_j & \exists i : j = s(i) \land t \in T_i \\ Z & \text{otherwise} \end{cases}$$

We require

$$1 < a_0 \land \forall i : a_i \leq b_i \leq a_{i+1} - 2$$

As illustrated in the idealized timing of figure 34, between the end $b_i$ of interval $T_i$ and the start $a_{i+1}$ of the next interval $T_{i+1}$, there is at least one cycle, where no driver is enabled in the digital model.

As shown in figure 35 control signals $yoe_j$ are generated as outputs of set-clear-flip flops which in turn are controlled by signals $yoeset_j$ and $yoeclr_j$. The rule for generation the latter signals is simple: for intervals $T_i$ during which $y_j$ is enabled ($j = s(i)$), the output enable signal is set in cycle $a_i - 1$ and cleared in cycle $b_i$:

$$yoeset^t_j \equiv \exists i : s(i) = j \land t = a_i - 1$$
$$yoeclr^t_j \equiv \exists i : s(i) = j \land t = b_i$$
In the digital model we immediately conclude

\[
yoe_j^t = \exists i : \text{send}(i) = j \land t \in T_i
\]

\[
b_j^t = \begin{cases} R_j & \exists i : \text{send}(i) = j \land t \in T_i \\ Z & \text{otherwise} \end{cases}
\]

as required in the digital specification. In the detailed model we can however show more. Before we do that, recall that \(e(t)\) is the time of the clock edge starting cycle \(t\). Because we are arguing about cycles and time simultaneously we denote cycles with \(q\) and times with \(t\).

**Lemma 28.**  
- After time \(e(1) + \sigma + \beta\) there is no bus contention

\[
t \geq e(1) + \sigma + \beta \land y_i(t) \neq Z \land y_j(t) \neq Z \rightarrow i = j
\]

- if \(j = \text{send}(i)\) and register \(R_j\) is not clock enabled during \(T_i\), then the content of \(R_j\) is glitch free on the bus roughly during \(T_i\)

\[
(j = \text{send}(i) \land \forall q \in T_i : /R_{c_j}^q) \rightarrow \\
\forall t \in [e(a_i) + \sigma + \beta, e(b_i + 1) + \rho + \alpha] : b(t) = R_j^{a_i}
\]

Note that the hypotheses of this lemma are all digital. Thus we can prove them if necessary entirely in the digital world.

Proof: Consider the timing diagram in figure 36. For the outputs of the set-reset flip-flops \(yoe_j\) we get after reset

\[
e(1) + \sigma \leq te(2) + \rho \rightarrow yoe_j(t) = 0
\]
For $t > e(2) + \rho$ we get

$$yoe_j(t) = \begin{cases} 
\Omega & \exists i : \text{send}(i) = j \land t \in e(a_i) + (\rho, \sigma) \\
1 & \exists i : \text{send}(i) = j \land t \in [e(a_i + \sigma), e(b_i + 1) + \rho) \\
\Omega & \exists i : \text{send}(i) = j \land t \in e(b_i + 1) + (\rho, \sigma) \\
0 & \text{otherwise} 
\end{cases}$$

For the outputs $y_j$ of the drivers follows after reset

$$e(1) + \sigma + \beta \leq te(2) + \rho + \alpha \rightarrow y_j(t) = Z$$

For $t > e(2) + \rho$ we get

$$y_j(t) \neq Z \rightarrow \exists i : \text{send}(i) = j \land t \in (e(a_i), e(b_i + 2))$$

Hence

$$y_j(t) \neq Z \rightarrow \exists i : \text{send}(i) = j \land t \in (e(a_i), e(b_i + 2))$$

The first statement of the lemma now follows because

$$e(2) \leq e(a_1) \land e(b_i + 2) \leq e(a_{i+1})$$

For the second statement of the lemma we use lemmas 23 and 26 to conclude from the extra hypothesis

$$j = \text{send}(i) \land t \in [e(a_i) + \sigma, e(b_i + 1) + \rho] \rightarrow R_j(t) = R_j(e(a_i + 1)) = R^{a_i}_j$$

We have shown already about the output enable signals

$$j = \text{send}(i) \land t \in [e(a_i) + \sigma, e(b_i + 1) + \rho] \rightarrow yoe_j(t) = 1$$

Thus we get for the driver values

$$j = \text{send}(i) \land t \in [e(a_i) + \sigma + \beta, e(b_i + 1) + \rho + \alpha] \rightarrow y_j(t) = R^{a_i}_j \neq Z$$

From the first part of the lemma we conclude for the value of the bus

$$j = \text{send}(i) \land t \in [e(a_i) + \sigma + \beta, e(b_i + 1) + \rho + \alpha] \rightarrow b(t) = R^{a_i}_j$$

Figure 37 a) shows symbol and implementation on an n-tri state driver. This driver consists simply of n tri state drivers with a common output enable signal.
2.5. DRIVERS AND MAIN MEMORY

\[ x[n-1:0] \]
\[ y[n-1:0] \]

\[ \text{a) Symbol} \]
\[ \text{b) Implementation} \]

Figure 37: Symbol and construction of an n-tri state driver

2.5.6 Specification of main memory

As a last building block for hardware we introduce main memory h.mm. It is line addressable memory

\[ h.mm : \mathbb{B}^{29} \to \mathbb{B}^{64} \]

It is accessed via a tri state bus b with the following components:

- \( b.data \in \mathbb{B}^{64} \). In write operations this is a cache line to be stored in main memory. In (the last cycle of) read operations it contains the data read from main memory.
- \( b.a \in \mathbb{B}^{29} \). The line address of main memory operations
- \( b.mmreq \in \mathbb{B} \). The request signal for main memory operations
- \( b.mmw \in \mathbb{B} \). The main memory write signal that the current main memory request is a write.
- \( b.mmack \in \mathbb{B} \). The main memory acknowledgement signal. The main memory activates it in the last cycle of a main memory operation

An incomplete digital specification of main memory accesses can be read off the idealized timing diagram in figure 1.

Operating conditions of main memory are formulated in the following definitions and requirements:

1. **Stable Inputs.** In general, accesses to main memory last several cycles. During such an access, main memory requires the inputs to be stable:

\[ mmreq^t \land \neg mmack^t \Rightarrow mmw^t = mmw^{t+1} \land b.ad^t = b.ad^{t+1} \land (mmw^t \Rightarrow b.data^t = b.data^{t+1}) \]

This is an incomplete part of the digital specification.
2. Main memory ready predicate. We define \( \text{mmready} \) as an auxiliary predicate:

\[
\text{mmready}^t = (\forall t' < t : \neg \text{mmreq}^{t'}) \\
\vee (\exists t'' < t : \text{mmack}^{t''} \land \forall t' \in (t'', t] : \neg \text{mmreq}^{t'})
\]

For the memory to be ready, we have two possible conditions: either, there has not been a memory request since reset or the last memory request was acknowledged and the request signal was off since then.

3. No Spurious Acknowledgements. The main memory will never raise a \( \text{mmack} \) signal unless the \( \text{mmreq} \) signal is set:

\[
\neg \text{mmreq}^t \rightarrow \neg \text{mmack}^t
\]

4. Liveness and request execution If the inputs are stable, we may assume liveness for the main memory, i.e., every request is eventually served:

\[
\text{stable_inputs} \land \text{mmready}^{t-1} \land \text{mmreq}^t \rightarrow (\exists t' > t : \text{mmack}^{t'} \\
\land (\text{mmw}^t \rightarrow \text{mm}^{t+1}(b.ad^t) = b.data^t) \\
\land (\neg \text{mmw}^t \rightarrow b.d^t = \text{mm}^t(b.ad^t))
\]

We denote the cycle in which the acknowledge for the main memory request from cycle \( t \) by

\[
\text{ackc}(t) = \min\{x > t : b.mmack(h^x) = 1\}
\]

The effect of read and write operations is specified in the following definitions, where we assume \( \text{mmreq}^0 \land /\text{mmreq}^{2-1} \)
2.5. **DRIVERS AND MAIN MEMORY**

1. **Effect of write operations** In the cycle after the end of a write cycle the \( b.data \) component of the bus is written to main memory at the address specified by the address component \( b.a \) of the bus

\[
mmw^q \rightarrow mm^a_{\text{ack}(q)+1}(x) = \begin{cases} 
  b.data^q & x = b.a^q \\
  mm^q(x) & \text{otherwise}
\end{cases}
\]

2. **Effect of read operations.** In the last cycle of a read the data component \( b.data \) of the bus is the content of main memory (at the start of the access) at address specified by the address component \( b.a \) of the bus

\[
/mmw^q \rightarrow b.data_{\text{ack}(q)} = mm^q(b.a^q)
\]

3. **Tri state driver enable** The driver \( mmbd \) connecting the main memory to the \( b.data \) bus is never enabled outside of a read access.

\[
\exists q : b.mmreq^q \land /b.mmw^q \land t \in [q, \text{ack}(q)] \rightarrow mmbd^q = Z
\]

This is the second incomplete part of the digital specification.

The specification is completed in the detailed hardware model by three conditions.

1. We require that during a main memory access inputs to main memory have to be free of glitches. Consider an access initiated in cycle \( q \) by raising the \( b.mmreq \) signal. Then the set of input components \( b.X \) of the access are defined as

\[
mmin(q) = \{b.a, b.mmreq, b.mmw\} \cup \begin{cases} 
  b.data & b.mmw(h^q) \\
  \emptyset & \text{otherwise}
\end{cases}
\]

The detailed specification has a new timing parameter, namely a main memory set up time \( mmts \). This setup time has to be large enough to permit a reasonable control automaton (as specified in section 2.6) to compute a next state and a response (the \( b.mmbusy \)-signal) before the next clock edge.

We require input components \( b.X \) of the bus to have the digital value \( b.X^q \) from time \( mmts \) before edge \( e(q+1) \) until hold time \( th \) after edge \( e(\text{ack}(q) + 1) \)

\[
X \in mmin(q) \land t \in [e(q + 1) - mmts, e(\text{ack}(q) + 1) + th] \quad (2.5)
\]

\[
\rightarrow b.X(t) = b.X^q
\]
2. We also have to specify the timing of responses given by main memory. The set of output components of an access initiated in cycle $q$ is defined as

$$mmout(q) = \{\text{ack}\} \cup \begin{cases} b.\text{data} & /b.\text{mmw}(h^q) \\ \emptyset & \text{otherwise} \end{cases}$$

We require output components $b.X$ of the bus to have digital value $b.X^{\text{ackc}(q)}$ from time $mmts$ after edge $e(\text{ackc}(q) + 1)$ until hold time $th$ after edge $e(\text{ackc}(q) + 1)$

$$X \in mmout(q) \land t \in [e(\text{ackc}(q) + 1) - mmts, e(\text{ackc}(q) + 1) + th]$$

$$\rightarrow b.X(t) = b.X^{\text{ackc}(q)}$$

3. Finally we have to define the absence of bus contention in the detailed model so that clean operation of the tri state bus can be guaranteed. The $mmbd$-driver can only be outside the high $Z$ from start of the cycle starting a read access until the end of the cycle following a read access:

$$mmbd(t) \neq Z \land \exists q : b.\text{mmreq}^q \lor /b.\text{mmw}^q$$

$$\land t \in (e(q), \text{ackc}(q) + 2) \rightarrow mmbd(t) = Z$$

2.5.7 Operation of main memory via a tri state bus

We extend the control of the tri state bus from subsection 2.5.5 to a control of the four components of the main memory bus. We consider $k$ units $U(j)$ with $j \in [0 : k - 1]$ capable of accessing main memory. Each has registers $\text{mmreq}_j, \text{mmw}_j, a_j$ and $\text{data}_j$. They are connected to the bus $b$ accessing main memory in the obvious way: bus components $b.X$ with $X \in$
\{a, mmreg, mmw\} occur only as inputs to the main memory. The situation shown in figure 39 for unit \(U(j)\) is simply a special case of figure 33 with

\[
\begin{align*}
R_j &= X_j \\
y_j &= Xbd_j \\
b &= b.X
\end{align*}
\]

As shown in figure 40 bus components \(b.data\) can be driven both by the units and by the main memory. If main memory drives the data bus, the data on the bus can be clocked into input register \(Q_j\) of unit \(U(j)\). If the data bus is driven by a unit, the data on the bus can be stored in main memory. We treat main memory simply as unit \(k\). Then we almost have a special case of figure 33 with

\[
\begin{align*}
R_j &= \text{data}_j & \text{if } j \leq k - 1 \\
y_j &= \begin{cases} 
\text{data}_jbd & j \leq k - 1 \\
\text{mmbd} & j = k
\end{cases} \\
b &= b.data
\end{align*}
\]

Signal \(b.mmack\) is broadcast by main memory. Thus bus control is not necessary for this signal. We want to extend the proof of lemma 28 to show that all four tri state buses above operated in a clean way. We also use the statement of lemma 28 to show that the new control produces memory input without glitches in the sense of the main memory specification. The crucial signals governing the construction of the control are the main memory request signals \(mmreq\). We compute them in set clear flip flops; they are cleared at reset.

\[\forall y : mmreq^1_y = 0\]

For the set and clear signals of the request signals we use the following discipline:
CHAPTER 2. HARDWARE

- a main memory request signal is only set, when all request signals are off

\[ \text{mmreq}_j.set^x \rightarrow \forall y : \text{mmreq}^x_y = 0 \]

- at most one request signal is turned on at a time (this requires some sort of bus arbitration):

\[ \text{mmreq}_j.set^x \land \text{mmreq}_{j'} set^x \rightarrow j = j' \]

- a request which starts in cycle \( q \) is kept on until the corresponding acknowledgement in cycle \( \text{ackc}(q) \) and is turned off

\[ \text{mmreq}_j.set^{x-1} \rightarrow (\forall x \in [q : \text{ackc}(q) - 1] : /\text{mmreq}_j.clr^x) \land \text{mmreq}_j.clr^{\text{ackc}(q)} \]

Now we can define access intervals \( T_i = [a_i, b_i] \): The start cycle \( a_i \) of interval \( T_i \) is occurrence number \( i \) of the event that any signal \( \text{mmreq}_j \) turns on. The end cycle \( b_i \) just occurs with the corresponding acknowledgement

\[
\begin{align*}
    a_0 &= \min\{x \geq 1 : \exists j : \text{mmreq}_j.set^x\} + 1 \\
    a_{i+1} &= \min\{x > \text{ackc}(a_i) : \exists j : \text{mmreq}_j.set^x\} + 1 \\
    b_i &= \text{ackc}(a_i)
\end{align*}
\]

For bus components \( b.X \) with \( X\{a, \text{mmreq}, \text{mmw}\} \) we define a unit \( U(j) \) to be sending in interval \( T - i \) if its request signal is on at the start of the interval.

\[ \text{send}(i) = j \leftrightarrow \text{mmreq}_j^{a_i} = 1 \]

Now things are easy. For bus components \( b.X \) with \( X \in \{a, \text{mmreq}, \text{mmw}\} \) occurring only as inputs we control registers and drivers as prescribed in lemma 28 and conclude

\[ \forall t \in [e(a_i) + \sigma + \beta, e(b_i + 1) + \rho + \alpha] : b.X(t) = X_{send(i)}^{a_i} \]

For the data component \( b.data \) of the bus we define unit \( U(j) \) to be sending if its request signal is on in cycle \( a_i \) and the request is a write request. We define the main memory to be sending \( (send(i) = k) \), if the request in cycle \( a_i \) is a read request

\[ \text{send}(i) = \begin{cases} 
    j & \text{mmreq}_j^{a_i} = 1 \land \text{mmw}_j^{a_i} \\
    k & \exists j : \text{mmreq}_j^{a_i} = 1 \land /\text{mmw}_j^{a_i}
\end{cases} \]

Now control for \( j \in [0 : k - 1] \) all registers \( data_j \) as prescribed in lemma 28. Absence of bus contention for component \( b.data \) follows from the proof of lemma 28 and equation 2.7 in the specification of main memory. For write operations \( (send(i) < k) \) we conclude with lemma 28

\[ \forall t \in [e(a_i) + \sigma + \beta, e(b_i + 1) + \rho + \alpha] : b.X(t) = X_{send(i)}^{a_i} \]
2.6. **FINITE STATE TRANSDUCERS**

Under reasonable assumptions for timing parameters and cycle time \( \tau \) this completes the proof of equation 2.5 of the main memory specification requiring that glitches are absent in main memory input

**Lemma 29.** Let \( \rho + \alpha \geq th \) and \( \sigma + \beta + \text{mmts} \leq \tau \). Then

\[ \text{mmin}(a_i) \land t \in [e(a_i + 1) - \text{mmts}, e(ackc(q) + 1) + th] \rightarrow b.X(t) = b.X^0 \]

Equation 2.6 is needed for timing analysis. In order to meet set up times for the data of input \( Q_j \) in of registers \( Q_j \), on bus \( b.data \) it obviously suffices if

\[ \text{mmts} \geq ts \]

However a larger lower bound for parameter \( \text{mmts} \) will follow from the construction of particular control automata in chapter 7

### 2.6 Finite State Transducers

Control automata resp. finite state transducers are finite automata which produce an output in every step. Formally, a finite state transducer is defined by a 6-tuple \( (Z, z_0, I, O, \delta, \eta) \), where \( Z \) is a finite set of states; \( I \subseteq \{0, 1\}^\sigma \) is a finite set of input symbols; \( z_0 \in Z \) is called the initial state; \( O \subseteq \{0, 1\}^\gamma \) is a finite set of output symbols;

\[ \delta_A : Z \times I \rightarrow Z \]

is the transition function, and

\[ \eta : Z \times I \rightarrow O \]

is the output function.

Such an automaton works step by step according to the following rules:

- The automaton is started in state \( z_0 \).
- If the automaton is in state \( z \) and reads input symbol \( in \), then it outputs symbol \( \eta(z, in) \) and goes to state \( \delta_A(z, in) \).

If the output function does not depend on the input, i.e., if it can be written as

\[ \eta : Z \rightarrow O \]

then the automaton is called a **Moore automaton**. Otherwise, it is called a **Mealy automaton**.

Automata are very often visualized in graphical form. We will do this too in subsection 7.4.3 when we construct several automata for the control
of a cache coherence protocol. In this case states \( z \) are drawn as cycles with a \( z \) written inside. A state transition

\[
z' = \delta_A(z, i)
\]

is visualized as an arrow from \( z \) to \( z' \) with label \( i \) as shown in figure 41. Initial states are sometimes drawn as a double cycle.

In what follows we show how to implement control automata by switched circuits. We start with the simpler Moore automata and the generalize the construction to Mealy automata.

### 2.6.1 Realization of Moore Automata

Let \( k = \#Z \) be the number of states of the automaton. Then states can be numbered from 0 to \( k-1 \), and we can rename the states with numbers from 0 to \( k-1 \) with 0 as initial state.

\[
Z = \{0, \ldots, k-1\}, \quad z_0 = 0
\]

We code the current state \( z \) in a register \( S \in B^k \) by the simple unary coding

\[
S = \text{code}(z) \quad \leftrightarrow \\
\forall i : S[i] = \begin{cases} 
1 & z = i \\
0 & \text{otherwise}
\end{cases}
\]

A completely straightforward and naive implementation is shown in figure 42. By the reset logic we get

\[
h^1.S = \text{code}(0)
\]

Circuits \textit{out} (like output) and \textit{nexts} are constructed such that the automaton is simulated in the following sense: if \( h.S = z \), i.e. state \( z \) is encoded by the hardware, then i) \( \text{out}(h) = \eta(z) \), i.e. automaton and hardware produce the same output. In the next cycle the hardware \( h'.S \) encodes the next state \( \text{delta}_A(z, \text{in}(h)) \).
Lemma 30. Let 

\[ h.S = code(z) \land \delta_A(z, in(h)) = z' \]

Then 

\[ out(h) = \eta(z) \land h'.S = code(z') \]

For all \( i \in [0 : \gamma - 1] \) we construct the \( i \)'th output simply by OR-ing together all bits \( S[x] \) such \( \eta(x)[i] = 1 \), i.e. such that the \( i \)'th output is on in state \( x \) of the automaton 

\[ out(h)[i] = \bigvee_{\eta(x)[i]=1} h.S[x] \]

A straightforward argument shows the first claim of the lemma. Assume \( h.S = z \). Then 

\[ h.S[x] = 1 \iff x = z \]

Hence 

\[ out(h)[i] = 1 \] 

\[ \iff \bigvee_{\eta(x)[i]=1} h.S[x] = 1 \] 

\[ \iff \exists x : \eta(x)[i] = 1 \land h.S[x] = 1 \] 

\[ \iff \eta(z)[i] = 1 \]

Lemma 16 gives 

\[ out(h)[i] = \eta(z)[i] \]

Figure 42: Naive implementation of a Moore automaton
For for states $i, j$ we define auxiliary switching functions

$$
\delta_{i, j} : \mathbb{B}^r \to \mathbb{B}
$$

from the transition function $\delta_A$ of the automaton by

$$
\delta_{i, j}(in) = 1 \leftrightarrow \delta_A(i, in) = j
$$

i.e. function $\delta_{i, j}(in)$ is on if input $in$ takes the automaton from state $i$ to state $j$. Boolean formulas for functions $\delta_{i, j}$ can be constructed by lemma 20. For each state $j$ component $\text{nexts}[j]$ of the next state function is turned on in states $i$ such that the input $in$ takes the automaton to state $j$:

$$
\text{nexts}(h)[j] = \bigvee_x h.S[x] \land \delta_{x, j}(in(h))
$$

For the second claim of the lemma let

$$
\begin{align*}
    h.S &= \text{code}(z) \\
    \delta_A(z, in(h)) &= z'
\end{align*}
$$

For any next state $j$ we then have

$$
\begin{align*}
    \text{nexts}(h)[j] &= 1 \\
    \leftrightarrow \bigvee_x h.S[x] \land \delta_{x, j}(in(h)) = 1 \\
    \leftrightarrow \delta_{x, j}(in(h)) = j \\
    \leftrightarrow \delta_A(z, in(h)) = j
\end{align*}
$$

Hence

$$
\text{nexts}(h)[j] = \begin{cases} 1 & j = z' \\ 0 & \text{otherwise} \end{cases}
$$

Thus

$$
\begin{align*}
    \text{code}(z') &= \text{nexts}(h) \\
    &= h'.S
\end{align*}
$$

### 2.6.2 Precomputing Outputs of Moore Automata

The previous construction has the disadvantage that the propagation delay of circuit $\text{out}$ tends to contribute to the cycle time of the circuitry controlled by the automaton. This can be avoided by precomputing the output signals of a Moore automaton as a function of the next state signals as shown in figure 43.

As above one shows

$$
\text{Sin}(h) = \text{code}(z) \to \text{out}(h) = \eta(z)
$$
2.6. FINITE STATE TRANSDUCERS

For $h = h^0$ the reset signal is active and we have

$$Sin(h) = 0^{k-1}1 = code(0) \land out(h) = \eta(0)$$

Thus

$$h^1.S = code(0) \land h^1.outR = \eta(0)$$

We show

**Lemma 31.** For $h = h^t, t \geq 1$ let

$$h.S = code(z) \land \delta_A(z, in(h)) = z'$$

Then

$$h'.S = code(z') \land h'.outR = \eta(z')$$

We have $reset(h) = 0$ and hence $Sin(h) = nexts(h)$. From above we have

$$h'.S = nexts(h) = code(z')$$

and

$$h'.outR = out(h) = \eta(z')$$

### 2.6.3 Realization of Mealy Automata

Figure 44 shows a simple implementation of Mealy automata. Compared to Moore automata only the generation of output signals changes; the next state computation stays the same. Output $\eta(z, in)$ now depends both on
the current state $z$ and the current input $\text{in}$. For states $z$ and indices $i$ of outputs we derive from function $\eta$ a set of switching function $f_{z,i}$ by

$$f_{z,i}(\text{in}) = 1 \leftrightarrow \eta(z, \text{in})[i] = 1$$

Output is generated by

$$\text{out}(h)[i] = \bigvee_x h.S[x] \land f_{z,i}(\text{in}(h))$$

This generates the outputs of the automaton in the following sense.

**Lemma 32.**

$$h.S = \text{code}(z) \rightarrow \text{out}(h) = \eta(z, \text{in}(h))$$

Again, the proof is straightforward:

$$\text{out}(h)[i] = 1$$

$$\leftrightarrow \bigvee_x h.S[x] \land f_{z,i}(\text{in}(h)) = 1$$

$$\leftrightarrow f_{z,i}(\text{in}(h)) = 1$$

$$\leftrightarrow \eta(z, \text{in}(h))[i] = 1$$

### 2.6.4 Partial Precomputation of Outputs of Mealy Automata

We describe two optimizations that can reduce the delay of outputs of Mealy automata. The first one is trivial. We devide the output components $\text{out}[j]$
into two classes: i) Mealy components $\eta[k](z, in)$ have a true dependency on the input variables. ii) Moore components that can be written as $\eta[k](z)$, i.e. that only depend on the current state. Suppose we have $\alpha$ Moore components and $\beta$ Mealy components with $\gamma = \alpha + \beta$. The obviously one can precompute the Moore components as in a Moore automaton and realize the Mealy components as in the previous construction of Mealy automata. The resulting construction is shown without further correctness proof in figure 45.

However, very often more optimization is possible, because Mealy components usually depend only on very few input bits $in[j]$ of the automaton. As an example consider a Mealy output

$$f(z, in[1 : 0]) = \eta[k](z, in[1 : 0])$$

depending only two input bits. For $x, y \in B$ we derive Moore outputs $f_{x,y}$ that precompute $\eta[k]$ if $in[1 : 0] = xy$:

$$f_{x,y}(z) = \eta(z, x, y)$$

Now we precompute for the automata outputs $f_{x,y}$ circuit outputs $g_{x,y}(Sin(h))$ and store them in registers $g_{x,y}R$ as shown in figure 46.

As for precomputed Moore signals one shows

$$h.S = code(z) \rightarrow h.g_{x,y}R = f_{x,y}(z)$$

For the output $f$ of the multiplexer tree we conclude
Figure 46: Partial precomputation of a Mealy component depending on two input bits

\[
f(z, \text{in}[1:0]) = h \cdot g_{\text{in}[1:0]} R \\
= f_{\text{in}[1:0]}(z) \\
= \eta(z, \text{in}[1:0])
\]

This construction has the advantage, that only the multiplexers contribute to the delay of the control signals generated by the automaton. In general for Mealy signals which depend on \(k\) input bits, we have \(k\) levels of multiplexers.
Chapter 3

Nine Shades of RAM

The processors of multicore machines communicate via a shared memory in a highly nontrivial way. Thus, not surprisingly, memory components play an important role in the construction of such machines. We start with a basic construction of (static) random access memory (RAM). Next we derive 5 specialized designs: read only memory (ROM), combined ROM and SRAM, cache state RAM, cache data RAM and special purpose register RAM (SPR-RAM). Then we generalize the construction to multiport RAM; this is RAM with more than one address and data port. We will need multiport RAMs in 4 flavors: 3 port RAM for the construction of general purpose register files, general two port RAM, two port cache sstate RAM and two port cache data RAM.

3.1 Basic Random Access Memory

As shown in figure 47 an \((n, a)\)-static RAM \(S\) or SRAM is a portion of a clocked circuit with the following groups of inputs and outputs:

- an \(n\) bit data input \(S_{in}\)
- an \(a\) bit address inputs \(S_{a}\)
- write signal \(S_{w}\)
- an \(n\) bit data output \(S_{out}\)

Internally this static RAM contains \(2^a\) many \(n\)-bit registers \(S(a) \in \mathbb{B}^n\); thus it is modeled by a function

\[ h : S : \mathbb{B}^a \to \mathbb{B}_{\text{Omega}}^n \]

The initial content is unknown

\[ \forall a : h^0, R(a) = \Omega^n \]
We only define output and next state of SRAM for situations where both addresses and write signal are binary. The output of a RAM is the register content selected by the address input.

$$Sout(h) = h.S(Sa(h))$$

For addresses $x \in \mathbb{B}^n$ we define

$$h'tS(x) = \begin{cases} Sin(h) & Sa(h) = x \land Sw(h) = 1 \\ h.S(x) & \text{otherwise} \end{cases}$$

For the implementation we first define $(n, A)$-OR trees for powers of two $A$. As shown in figure 48 have $n \cdot A$ inputs vectors $b[i]$ with $i \in [A - 1 : 0]$, each consisting of $n$ bits $b[i][j]$ with $j \in [n - 1 : 0]$. the outputs $out[j \cdot n - 1 : 0]$ satisfies

$$out[i] = \bigvee_{i=0}^{A-1} b[i]$$
The implementation in figure 4 generalizes the definition of 1 bit wide OR trees from figure ... in an obvious way.

The implementation of an SRAM is shown in figure 50. We use $2^a$ many $n$ bit registers $R^{(i)}$ with $i \in [0 : 2^a - 1]$ and an $a$-decoder with outputs $Y[0 : 2^a - 1]$ satisfying

$$X(i) = 1 \iff i = \langle Sa(h) \rangle$$

The inputs of register $R^{(i)}$ are

$$h.R^{(i)}_{\text{din}} = Sin$$
$$h.R^{(i)}_{\text{ce}} = Sw \land X[i]$$

For the next state we get

$$h'.R^{(i)} = \begin{cases} 
  Sin(h) & i = \langle Sa(h) \rangle \\
  h.R^{(i)} & \text{otherwise}
\end{cases}$$

The $i^{th}$ input vector $b[i]$ to the OR tree is constructed as

$$b[i] = X[i] \land h.R^{(i)}$$
$$= \begin{cases} 
  h.R^{(i)} & i = \langle Sa \rangle \\
  0^n & \text{otherwise}
\end{cases}$$

Thus

$$S\text{out} = \bigvee_{i=0}^{2^a-1} b[i] = h.R^{\langle Sa(h) \rangle}$$
Figure 50: Construction of an \((n,a)\)-SRAM

\[
\begin{array}{c}
S_a \quad a-\text{Dec} \\
\downarrow \\
X[0] \\
\downarrow \\
X[a] \\
\downarrow \\
X[2^a - 1] \\
\downarrow \\
R(0) \\
\downarrow \\
\vdots \\
\downarrow \\
R(n) \\
\downarrow \\
\vdots \\
\downarrow \\
R(2^a-1) \\
\downarrow \\
S_{out}
\end{array}
\]

Figure 51: Symbol of an \((n,a)\)-ROM

Thus, for
\[h.S(x) = h.R^i(x)\]
the construction implements an SRAM.

### 3.2 Single Port RAM Designs

#### 3.2.1 Read Only Memory (ROM)

An \((n,a)\)-ROM is is a memory with a drawback and an advantage. The drawback: it can only be read. The advantage: its content is known after power up. It is modeled by a mapping \(S : B^n \rightarrow B^m\) which does not depend on the hardware configuration \(h\). The realization is obtained by a trivial variation of the basic RAM design: replace each register \(R^i\) by the constant input \(S(i_a) \in B^m\). There are no data in, write or clock enable signals. Indeed the hardware constructed in this way is a circuit. Symbol and construction are shown if figures 51 and 52.
3.2. Single Port RAM Designs

3.2.2 Combining RAM and ROM

It is very often desirable to realize some small portion of memory by ROM and the remaining large part as RAM. The standard use for this is ROM for boot code. After power up memory content is unknown. That makes it impossible to start computation in a meaningful way unless at least some portion of memory contains code that is known after power up. The reset mechanism as to ensure that processors start execution of programs in this region. It usually contains a so called boot loader. This is code that accesses a large and slow memory device - like a disk - and loads from the device further programs to be executed.

For \( r < a \) we define a combined \((n, r, a)\)-RAM-ROM \( S \) as a device that behaves for small addresses \( a = 0^n \leftrightarrow b \) with \( b \in \mathbb{B}^r \) like ROM and on the other addresses like RAM. As ordinary \((n, a)\)-RAM it is modeled as \( h.S : \mathbb{B}^a \rightarrow \mathbb{B}^n \) and output

\[
Sout(h) = h.S(Sa(h))
\]

Write operations however affect only addresses larger than \( 0^r \leftrightarrow 1^r \)

\[
h'.S(x) = \begin{cases} 
    h^0.S(x) & x[a-1 : r] = 0^r \\
    Sin(h) & x = Sa(h) \land Sw(h) \land x[a-1 : r] \neq 0^r \\
    h.S(h) & \text{otherwise}
\end{cases}
\]

The symbol for an \((n, r, a)\)-RAM-ROM and a straight forward implementation involving an \((n, a)\)-SRAM, an \((n, r)\)-ROM and an \((a-r)\)-zero tester is shown in figures 53 and 54

3.2.3 Multi Bank RAM

Let \( n = 8k \) be a multiple of 8. An \((n, a)\)-multi-bank RAM \( S : \mathbb{B}^a \rightarrow \mathbb{B}^{8k} \) is basically an \((n, a)\)-RAM with separate bank write signals \( bw(i) \) for each byte \( i \in [0 : k-1] \) (see figure 55). It has
Figure 53: Symbol of an \((n, r, a)\)-RAM-ROM

- data input \(Sin \in \mathbb{B}^{8k}\)
- data output \(Sout \in \mathbb{B}^{8k}\)
- address input \(Sa \in \mathbb{B}^a\)
- bank write signals \(bw[i] \in \mathbb{B}\) for \(i \in [k - 1 : 0]\)

Data output is defined exactly as for ordinary RAM

\[ Sout(h) = h.S(Sa(h)) \]

For the definition of the next state we define an important auxiliary function

\[ modify : \mathbb{B}^{8k} \times \mathbb{B}^{8k} \times \mathbb{B}^k \rightarrow \mathbb{B}^{8k} \]

Let \(y, z \in \mathbb{B}^{8k}\) and \(bw[k - 1 : 0] \in \mathbb{B}^k\). Then for all \(i \in [k - 1]\)

\[ byte(i, modify(y, z, bw)) = \begin{cases} \text{byte}(y, i) & bw[i] = 1 \\ \text{byte}(z, i) & bw[i] = 0 \end{cases} \]

i.e. for all \(i\) with active \(bw[i]\) one replaces byte \(i\) of \(y\) by byte \(i\) of \(z\). The next state of multi-bank RAM is then defined as

\[ h'.S(x) = \begin{cases} modify(h.S(x), Sin(h), bw(h)) & x = \langle Sa(h) \rangle \\ h.S(x) & \text{otherwise} \end{cases} \]

The straightforward construction uses \(k\) separate so called \(banks\). This are \((8, a)\)-RAMs \(S^{(i)}\) for \(i \in [k - 1 : 0]\). For each \(i\) bank \(S^{(i)}\) is wired as shown in figure 56:
3.2. SINGLE PORT RAM DESIGNS

\[ S^{(i)}_{in} = byte(i, Sin) \]
\[ S^{(i)}_{out} = byte(i, Sout) \]
\[ S^{(i)}_{w} = Sbw[i] \]

We abstract the state \( h.S \) for this construction as

\[ byte(i, h.S(x)) = h.S^{(i)}(x) \]

Correctness now follows in a lengthy but completely straight forward way from the specification of ordinary RAM. For the outputs:

\[ byte(i, Sout(h)) = S^{(i)}_{out}(h) \quad \text{(construction)} \]
\[ = h.S^{(i)}(S^{(i)}_{a}(h)) \quad \text{(RAM spec)} \]
\[ = h.S^{(i)}(Sa(h)) \quad \text{(construction)} \]
\[ = byte(i, h.S(Sa(h)) \quad \text{(state abstraction)} \]

For the new state and address \( x \neq Sa(h) \)
CHAPTER 3. NINE SHADES OF RAM

Figure 55: Symbol of an \((n, a)\)-multi-bank RAM

\[
\begin{align*}
\text{byte}(i, \text{Sin}) & \\
\text{Sa} & \quad a \\
S & \quad k \\
\text{Sbw} & \quad 8k \\
\text{Sout} & \quad 8k
\end{align*}
\]

Figure 56: Bank \(i\) of an \((n, a)\)-multi-bank RAM

\[
\begin{align*}
\text{byte}(i, \text{Sin}) & \\
\text{Sa} & \quad a \\
S(i) & \\
\text{Sbw}[i] & \\
\text{byte}(i, \text{Sout})
\end{align*}
\]

\[
\begin{align*}
\text{byte}(i, h'.S(x)) & = h'.S^{(i)}(x) \quad \text{(state abstraction)} \\
& = h.S^{(i)}(x) \quad \text{(RAM spec)} \\
& = \text{byte}(i, h.S(x))
\end{align*}
\]

For the new state and address \(x = Sa(h)\)
3.2. SINGLE PORT RAM DESIGNS

![Diagram of cache-state-RAM]

**Figure 57:** Symbol of an \((n, a)\)-cache-state-RAM

\[
\text{byte}(i, h'.S(x)) = h'.S^{(i)}(x) \quad \text{(state abstraction)}
\]

\[
= \begin{cases} 
S^{(i)}\text{in}(h) & S^{(i)}\text{w}(h) = 1 \\
.5S^{(i)}(x) & S^{(i)}\text{w}(h) = 0 
\end{cases} \quad \text{(RAM spec)}
\]

\[
= \begin{cases} 
\text{byte}(i, \text{Sin}(h)) & S\text{bw}[i](h) = 1 \\
h.S^{(i)}(x) & S\text{bw}[i](h) = 0 
\end{cases} \quad \text{(construction)}
\]

\[
= \begin{cases} 
\text{byte}(i, \text{Sin}(h)) & S\text{bw}[i](h) = 1 \\
\text{byte}(i, h.S(x)) & S\text{bw}[i](h) = 0 
\end{cases} \quad \text{(state abstraction)}
\]

i.e. we have

\[
h'.S(x) = \text{modify}(h.S(x), \text{Sin}(h), S\text{bw}(h))
\]

### 3.2.4 Cache State RAM

The symbol of an \((n, a)\)-cache-state RAM or CS-RAM is shown in figure 57. This type of RAM is used later for holding the status bits of caches. It has two extra inputs

- control signal \(\text{Sin}\): on activation, a special value is forced into all registers of the RAM. Later this indicates that all cache lines are invalid.

- \(n\)-bit input \(\text{Sinv}\) providing this special value. This input is usually wired to a constant value in \(B^n\) and signals that data in a cache line is invalid\(^1\)

\(^1\)i.e. not a copy of meaningful data in same programming model. We explain this in much more detail later.
Activation of $Sin$ takes precedence over ordinary write operations.

$$h'.S(x) = \begin{cases} 
Sinv(h) & Sinv(h) = 1 \\
Sin(h) & x = Sa(h) \land Sw(h) = 1 \\
h.S(x) & \text{otherwise}
\end{cases}$$

The changes in the implementation for each register $R^{(i)}$ are shown in figure 58. The clock enable is also activated by $Sin$ and the data input comes from a multiplexer

$$R^{(i)}ce = Sin \lor Y[i] \land Sw$$

$$R^{(i)}in = \begin{cases} 
Sinv & Sinv = 1 \\
Sin & \text{otherwise}
\end{cases}$$

**3.2.5 SPR-RAM**

An $(n, a)$-SPR-RAM as shown in figure ?? is used for the realization of special purpose register files. It behaves both as an $(n, a)$-RAM and as a set of $2^a$
many \( n \) bit registers. It has the following inputs and outputs

- an \( n \) bit data input \( S\text{in} \)
- an \( n \) bit data output \( S\text{out} \)
- a write signal \( Sw \)
- for each \( i \in [0 : 2^n - 1] \) an individual \( n \) bit data input \( S\text{din}[i] \) for register \( R^{(i)} \)
- for each \( i \in [0 : 2^n - 1] \) an individual \( n \) bit data output \( S\text{dout}[i] \) for register \( R^{(i)} \)
- for each \( i \in [0 : 2^n - 1] \) an individual clock enable signal \( S\text{ce}[i] \) for register \( R^{(i)} \)

Ordinary data output is generated as usual, and the individual data output simply are the output of the internal registers

\[
S\text{out}(h) = h.S(S\text{ad}(h))
\]
\[
S\text{dout}[i](h) = h.S(i_a)
\]

Register updates to \( R^{(i)} \) can be either by \( S\text{in} \) for regular writes or by \( S\text{din}[i] \) if the special clock enables are activated. Special writes take precedence over ordinary writes

\[
h'.S(x) = \begin{cases} 
S\text{din}[i](h) & i = \langle x \rangle \land S\text{ce}[i](h) = 1 \\
S\text{in}(h) & i = \langle x \rangle \land S\text{ce}[i](h) = 0 \land Sw(h) = 1 \\
h.S(x) & \text{otherwise}
\end{cases}
\]

A single address decoder with outputs \( Y[i] \) and a single OR-tree suffices. Figure 60 shows the construction satisfying

\[
R^{(i)}_{ce} = S\text{ce}[i] \lor X[i] \land Sw(h)
\]
\[
R^{(i)}_{in} = \begin{cases} 
S\text{din}[i](h) & S\text{ce}[i] = 1 \\
S\text{in} & \text{otherwise}
\end{cases}
\]

### 3.3 Multiport RAM Designs

#### 3.3.1 Three port RAM for general purpose registers

An \((n, a)\)-gpr-RAM is a three port RAM that we use later for general purpose registers. As shown in figure 61 it has the following inputs and outputs

- an \( n \) bit data input \( S\text{in} \)
CHAPTER 3. NINE SHADES OF RAM

![Diagram of (n, a)-spr-RAM](image)

Figure 60: Construction of an \( (n, a) \)-spr-RAM

![Diagram of (n, a)-gpr-RAM](image)

Figure 61: Symbol of an \( (n, a) \)-gpr-RAM

- three \( a \) bit address inputs \( Sa, Sb, Sc \)
- write signal \( Sw \)
- two \( n \) bit data outputs \( Sout_a, Sout_b \)

As for ordinary SRAM the state of the three port RAM is a mapping

\[
h.S : \mathbb{B}^a \rightarrow \mathbb{B}^n
\]

Outputs are addresses by address inputs \( Sa(h) \) and \( Sb(h) \)

\[
Sout_a(h) = h.S(Sa(h)) \\
Sout_b(h) = h.S(Sb(h))
\]
3.3. MULTIPORT RAM DESIGNS

Figure 62: Construction of an \((n, a)\)-gpr-RAM

Writing is performed under control of address input \(Sc(a)\)

\[
h' S(x) = \begin{cases} 
Sinc(h) & Sc(h) = x \land Sw(h) = 1 \\
h.S(x) & \text{otherwise} \end{cases}
\]

The implementation shown in figure 62 is a straightforward variation of the design for ordinary SRAM. One uses three different \(a\)-decoders with outputs \(X[0 : 2^a - 1], Y[0 : 2^a - 1], Z[0 : 2^a - 1]\) satisfying

\[
X[i] = 1 \iff i = \langle Sa \rangle \\
Y[i] = 1 \iff i = \langle Sb \rangle \\
Z[i] = 1 \iff i = \langle Sc \rangle
\]

Clock enable signals are derived from the decoded \(Sc\) address

\[
R^{(i)ce} = Z[i] \cdot Sw
\]
Outsuts $Souta, Soutb$ are generated by two $(n,2^a)$-Or trees with inputs $a[i], b[i]$ satisfying

\[
\begin{align*}
a[i] &= X[i] \land R^{(i)} \\
Souta &= \bigvee a[i] \\
b[i] &= Y[i] \land R^{(i)} \\
Soutb &= \bigvee b[i]
\end{align*}
\]

3.3.2 General Two Port RAM

A general $(n, a)$-2-port-RAM is shown in figure 63. This is a RAM with the following inputs and outputs

- two data inputs $Sina, Sinb$
- two addresses $Sa, Sb$
- two write signals $Swa, Swb$

The data outputs are determined by the addresses as in the three port RAM for general purpose registers

\[
\begin{align*}
Souta(h) &= h.S(Sa(h)) \\
Soutb(h) &= h.S(Sb(h))
\end{align*}
\]

Simultaneous writes to two addresses are now also possible. In case both write signals are active and both addresses point to the same port we have to resolve the conflict: the write via the $a$ port will take precedence.
3.3. MULTIPORT RAM DESIGNS

\begin{center}
\begin{tikzpicture}
    \node (Ri) at (0,0) {$R^{(i)}$};
    \node (x) at (-1,1) {$X[i] \land Swa$};
    \node (y) at (-1,2) {$X[i] \land Swa \lor Y[i] \land Swb$};
    \node (sa) at (-2,1) {$Sina(h)$};
    \node (sb) at (-2,0) {$Sinb(h)$};
    \node (h) at (-2,-1) {$h.S(x)$};
    \node (1) at (-1,2) {$1$};
    \node (0) at (-1,1) {$0$};
    \node (n) at (0,0) {$n$};
    \node (n2) at (-2,2) {$n$};
    \draw[->] (n2) -- (0,0);
    \draw[->] (0,0) -- (x);
    \draw[->] (x) -- (y);
    \draw[->] (1) -- (sa);
    \draw[->] (0) -- (sb);
    \draw[->] (h) -- (n2);
\end{tikzpicture}
\end{center}

Figure 64: Construction of an \((n, a)\)-two-port-RAM

\[
h'.S(x) = \begin{cases} 
    Sina(h) & x = Sa(h) \land Swa(h) = 1 \\
    Sinb(h) & x = Sb(h) \land Swb(h) = 1 \land \overline{x = Sa(h) \land Swa(h) = 1} \\
    h.S(x) & \text{otherwise}
\end{cases}
\]

Only two address decoders with outputs \(x[0 : 2^n - 1], Y[0 : 2^n - 1]\) are necessary. They satisfy

\[
X[i] = 1 \leftrightarrow i = \langle Sa \rangle \\
Y[i] = 1 \leftrightarrow i = \langle Sb \rangle
\]

Figure 64 shows the changes to each register \(R^{(i)}\). Clock enable is activated in case of a write via the a-Address or via the b-Address occurs. The input is chosen from one of the data inputs by a multiplexer

\[
R^{(i)ce} = Swa \land X[i] \lor Swb \land Y[i]
\]

\[
R^{(i)in} = \begin{cases} 
    Sina(h) & Swa[i] \land X[i] \\
    Sinb & \text{otherwise}
\end{cases}
\]

In this implementation writes via port \(a\) take precendence over writes via port \(b\) to the same address.

Output is generated as for gpr-rams.

3.3.3 Two Port Cache State RAM

Exactly as the name indicates, an \((n, a)\)-two-port-cs-RAM is a RAM with all features of a two-port-RAM and a cs-RAM. Its symbol is shown in figureInputs and outputs are

- two data inputs \(Sina, Sinb\)
Figure 65: Symbol of an \((n, a)\)-two-port-es-RAM

- two addresses \(Sa, Sb\)
- two write signals \(Swa, Swb\)
- control signal \(Sinv\)
- \(n\)-bit input \(Sinv\) providing a special data value

Address decoding, data output generation and execution of writes is as for two port RAMs. In write operations activation of signal \(Sinv\) takes precedence over everything else

\[
h'.S(x) = \begin{cases} 
Sinv(h) & Sinv(h) = 1 \\
Sina(h) & Sinv(h) = 0 \land x = Sa(h) \land Swa(h) = 1 \\
Sinv(b) & Sinv(h) = 0 \land x = Sb(h) \land Swb(h) = 1 \\
\land x = Sa(h) \land Swa(h) = 1 \\
h.S(x) & \text{otherwise}
\end{cases}
\]

The changes in the implementation for each register \(R^{(i)}\) are shown in figure 10. The signals thus generated are

\[
R^{(i)}_{ce} = Sinv \lor X[i] \land Swa \lor Y[i] \land Swb
\]

\[
R^{(i)}_{in} = \begin{cases} 
Sinv(h) & Sinv(h) = 1 \\
Sina(h) & Sinv(h) = 0 \land X[i] \land Swa(h) \\
Sinv(b) & \text{otherwise}
\end{cases}
\]
Figure 66: Construction block of an \( (n,a) \)-two-port-os-RAM
Chapter 4

Arithmetic Circuits

For later use in processors with the MIPS instruction set architecture (ISA) we construct several circuits: adder and incrementer, an arithmetic unit (AU), an arithmetic logic unit (ALU), a shift unit (SU) and a branch condition evaluation unit (BCE)

4.1 Adder and Incrementer

An $n$-adder is a circuit with inputs $a[n-1:0] \in \mathbb{B}^n$, $b[n-1:0]$, $c_0 \in \mathbb{B}$ and outputs $c_n \in \mathbb{B}$ and $s[n-1:0] \in \mathbb{B}^n$ satisfying

$$\langle c_n, s[n-1:0] \rangle = \langle a[n-1:0] \rangle + \langle b[n-1:0] \rangle + c_0$$

We use for $n$-adders the symbol from figure 67.

A full adder is obviously a 1-adder. A recursive construction of a very simple carry chain adder is shown in figure 68. The correctness follows directly from the correctness of the basic addition algorithm for binary numbers (lemma 11)

An $n$-incrementer is a circuit with inputs $a[n-1:0] \in \mathbb{B}^n$, $c_0 \in \mathbb{B}$ and outputs $c_n \in \mathbb{B}$ and $s[n-1:0] \in \mathbb{B}^n$ satisfying

$$\langle c_n, s[n-1:0] \rangle = \langle a[n-1:0] \rangle + c_0$$

We use for $n$-incrementers the symbol from figure 69.

Obviously incrementers can be constructed from $n$-adders by tying the $b$-input to $0^n$. As shown in section 2.2 a full adders whose $b$ input is tied to zero can be replaced to a half adder. This yields the construction in figure 70 of carry chain incrementers.

We introduce special symbols $+_n$ and $-_n$ to denote addition and subtraction of $n$ bit binary numbers mod$2^n$

$$a +_n b = \text{bin}_n((a) + \langle b \rangle \mod 2^n)$$

$$a -_n b = \text{bin}_n((a) - \langle b \rangle \mod 2^n)$$
\[ n = 1: \] \[ n > 1: \]

4.2 Arithmetic Unit

The symbol of an \( n \)-arithmetic unit or short \( n \text{-AU} \) is shown in figure 71. It is a circuit with the following inputs

- operand inputs \( a = a[n-1:0], b = b[n-1:0] \) with \( a, b \in \mathbb{B}^n \)
- control input \( u \) distinguishing between unsigned (binary) and signed (two's complement) numbers
- control input \( \text{sub} \) indicating whether input \( b \) should be subtracted from or added to input \( a \)

and the following outputs

- result \( s[n-1:0] \in \mathbb{B}^n \)
- overflow \( \text{ovf} \in B \)
4.2. ARITHMETIC UNIT

\[ n = 1: \quad n > 1: \]

\[ S = \begin{cases} 
[a] + [b] & (u, \text{sub}) = 00 \\
[a] - [b] & (u, \text{sub}) = 01 \\
\langle a \rangle + \langle b \rangle & (u, \text{sub}) = 10 \\
\langle a \rangle - \langle b \rangle & (u, \text{sub}) = 11 
\end{cases} \]

For the result of the ALU we pick the representative of the exact result in \( B_\text{n} \) resp. \( T_\text{n} \) and represent it in the corresponding format

\[ s = \begin{cases} 
\text{twoc}_{\text{n}}(S \mod 2^n) & u = 0 \\
\text{bin}_{\text{n}}(S \mod 2^n) & \text{else} 
\end{cases} \]

resp.

\[ [s] = (S \mod 2^n) \quad \text{if } u = 0 \]

\[ \langle s \rangle = (S \mod 2^n) \quad \text{if } u = 1 \]

- negative bit \( \text{neg} \in \mathbb{B} \)
Overflow and negation signals are defined with respect to the exact result

\[
\text{ovf} \leftrightarrow \begin{cases} 
S \not\in T_n & u = 0 \\
S \not\in B_n & u = 1 
\end{cases}
\]

\[
\text{neg} \leftrightarrow S < 0
\]

**Data Paths** The following somewhat slick lemma asserts, that for signed and unsigned numbers the sum bit \( s \) can be computed in exactly the same way:

**Lemma 33.** Compute the sum bits as

\[
s = \begin{cases} 
 a +_n b & \text{sub} = 0 \\
 a -_n b & \text{sub} = 1 
\end{cases}
\]

then

\[
\begin{align*}
|s| &= (S \mod 2^n) \quad \text{if} \quad u = 0 \\
\langle s \rangle &= (S \mod 2^n) \quad \text{if} \quad u = 1
\end{align*}
\]

Proof: for \( u = 1 \) this follows directly from the definitions. For \( u = 0 \) we have from lemma 14 and lemma 2

\[
\begin{align*}
|s| &= (s) \mod 2^n \\
&= ( (a) + (b) \quad \text{sub} = 0 ) \mod 2^n \\
&= ( (a) - (b) \quad \text{sub} = 1 ) \mod 2^n \\
&= ( |a| + |b| \quad \text{sub} = 0 ) \mod 2^n \\
&= ( |a| - |b| \quad \text{sub} = 1 ) \mod 2^n \\
&= S \mod 2^n
\end{align*}
\]

From \( |s| \in T_n \) and lemma 5 we conclude

\[
|s| = (S \mod 2^n)
\]
4.2. ARITHMETIC UNIT

The main data paths of an \( n \)-AU are shown in figure 72. That the sum bits are correctly computed is asserted in

**Lemma 34.** The sum bits \( s[n-1:0] \) in figure 1 satisfy

\[
s = \begin{cases} 
  a +_n b & \text{sub} = 0 \\
  a -_n b & \text{sub} = 1 
\end{cases}
\]

Proof: We have

\[
d = b \oplus \text{sub} \\
  = \begin{cases} 
  b & \text{sub} = 0 \\
  \bar{b} & \text{sub} = 1 
\end{cases}
\]

From the specification of an \( n \)-adder, lemma 10, and the subtraction algorithm for binary numbers (lemma15) we conclude

\[
\langle s \rangle = \langle \langle (a) + (b) \text{ sub} = 0 \rangle \text{ mod } 2^n \rangle \\
  = \langle \langle (a) + (\bar{b}) + 1 \text{ sub} = 0 \rangle \text{ mod } 2^n \rangle \\
  = \langle \langle (a) - (b) \text{ sub} = 0 \rangle \text{ mod } 2^n \rangle 
\]

Application of \( bin_n(\ ) \) to both sides gives the lemma.

**Negative Bit** We start with the case \( u = 0 \), i.e. two's complement numbers.

We have
\[ S = \lfloor a \rfloor \pm \lfloor b \rfloor \\
= \lfloor a \rfloor + \lfloor d \rfloor + \text{sub} \\
\leq 2^{n-1} - 1 + 2^{n-1} - 1 + 1 \\
= 2^n - 1 \\
S \geq -2^{n-1} - 2^{n-1} \\
= -2^n \\
\]
Thus
\[ S \in T_{n+1} \]

Therefore, according to lemma 14 we use sign extension to extend operands to \( n + 1 \) bits

\[
\lfloor a \rfloor = \lfloor a_{n-1}a \rfloor \\
\lfloor d \rfloor = \lfloor d_{n-1}d \rfloor
\]

We compute an extra sum bit \( s_n \) by the basic addition algorithm

\[ s_n = a_{n-1} \oplus d_{n-1} \oplus c_n \]

and conclude

\[ S = \lfloor s[n : 0] \rfloor \]

Again by lemma 14 this is negative if and only if the sign bit \( s_n \) is 1

\[ S < 0 \iff s_n = 1 \]

and we have

**Lemma 35.**

\[ u = 0 \rightarrow \text{neg} = a_{n-1} \oplus d_{n-1} \oplus c_n \]

For \( u = 1 \), i.e. binary numbers, a negative result can only occur in the case of subtraction, i.e. if \( \text{sub} = 1 \). In this case we argue along the lines of the correctness proof for the subtraction algorithm

\[ S = \langle a \rangle - \langle b \rangle \\
= \langle a \rangle - \langle 0b \rangle \\
= \langle a \rangle + \lfloor 1 \overline{0} \rfloor + 1 \\
= \langle a \rangle + \langle \overline{0} \rangle - 2^n + 1 \\
= \langle c_n s[n-1 : 0] \rangle - 2^n \\
= 2^n (c_n - 1) + \langle s[n-1 : 0] \rangle \in B^n \]
4.2. ARITHMETIC UNIT

If \( c_n = 1 \) we have \( S = \langle s \rangle \geq 0 \). If \( c_n = 0 \) we have

\[
S = -2^n + \langle s[n-1 : 0] \rangle \\
\leq -2^n + 2^n - 1 = -1
\]

Thus

\[ u = 1 \rightarrow neg = \text{sub} \land \overline{c_n} \]

and together with lemma 35 we get

Lemma 36.

\[
\begin{align*}
\text{neg} & = \overline{u} \land (a_{n-1} \oplus d_{n-1} \oplus c_n) \lor \\
& \quad u \land \text{sub} \land \overline{c_n}
\end{align*}
\]

Overflow Bit  If \( u = 0 \) we have

\[
S = |a| + |d| + \text{sub} = -2^{n-1}(a_{n-1} + d_{n-1}) + \langle a[n - 2 : 0] \rangle + \langle d[n - 2 : 0] \rangle + \text{sub}
\]

\[
= -2^{n-1}(a_{n-1} + d_{n-1}) + \langle c_{n-1}s[n - 2 : 0] \rangle - c_{n-1}2^{n-1} + c_{n-1}2^{n-1}
\]

\[
= -2^{n-1}(a_{n-1} + d_{n-1} + c_{n-1}) + 2^{n-1}(c_{n-1} + c_{n-1}) + \langle s[n - 2 : 0] \rangle
\]

\[
= -2^{n-1}(c_{n}s_{n-1}) + 2^n c_{n-1} + \langle s[n - 2 : 0] \rangle
\]

\[
= -2^n c_n - 2^{n-1}s_{n-1} + 2^n c_{n-1} + \langle s[n - 2 : 0] \rangle
\]

\[
= 2^n(c_{n-1} - c_n) + [s[n - 1 : 0]]
\]

We claim

\( S \in T_n \iff c_{n-1} = c_n \)

If \( c_n = c_{n-1} \) we obviously have \( S = \langle s \rangle \), thus \( S \in T_n \)

If \( c_n = 1 \) and \( c_{n-1} = 0 \) we have

\[
-2^n + |s| \leq -2^n + 2^{n-1} - 1 = -2^{n-1} - 1 < -2^{n-1}
\]

and if \( c_n = 0 \) and \( c_{n-1} = 1 \), we have

\[
2^n + |s| \geq 2^n - 2^{n-1} > 2^{n-1} - 1 < /math >
\]

Thus, in the two latter cases, we have \( S \notin T_n \). Because

\[
c_n \neq c_{n-1} \iff c_n \oplus c_{n-1} = 1
\]

we conclude

Lemma 37.

\[ u = 0 \rightarrow ovf = c_n \oplus c_{n-1} \]
For \( u = 1 \) we get an overflow if we add and have a carry out or if we subtract and the result is negative. From lemma 36 we conclude

\[
\begin{align*}
    u = 1 & \rightarrow \\
    neg &= \overline{\text{sub}} \land c_n \lor \text{sub} \land \overline{c_n} \\
    &= \text{sub} \oplus c_n
\end{align*}
\]

Together with lemma 37 we get

**Lemma 38.**

\[
    neg = u \land (c_n \oplus c_{n-1}) \lor u \land (\text{sub} \oplus c_n)
\]

### 4.3 ALU

Figure 73 shows a symbol for the \( n \)-ALU constructed here. \( n \) should be even. It has the following inputs

- operand inputs \( a = a[n - 1 : 0], b = b[n - 1 : 0] \) with \( a, b \in \mathbb{B}^n \)
- control inputs \( f[3 : 0] \in \mathbb{B}^4 \) and \( i \in \mathbb{B} \) specifying the operation that the ALU performs with the operands

and the following outputs

- result \( alures[n - 1 : 0] \in \mathbb{B}^n \)
- overflow \( ovfalu \in \mathbb{B} \)

The results that must be generated are specified in table 4.3. There are three groups of operations.

- arithmetic operations
4.4. **SHIFTER**

<table>
<thead>
<tr>
<th>a[i&lt;3:0]</th>
<th>i</th>
<th>alures[31:0]</th>
<th>ovfalu</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>*</td>
<td>a + b</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>*</td>
<td>a + b</td>
<td>[a] + [b] (\in T_n)</td>
</tr>
<tr>
<td>0010</td>
<td>*</td>
<td>a - b</td>
<td>0</td>
</tr>
<tr>
<td>0011</td>
<td>*</td>
<td>a - b</td>
<td>[a] - [b] (\in T_n)</td>
</tr>
<tr>
<td>0100</td>
<td>*</td>
<td>a &amp; b</td>
<td>0</td>
</tr>
<tr>
<td>0101</td>
<td>*</td>
<td>a \lor b</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>*</td>
<td>a + b</td>
<td>0</td>
</tr>
<tr>
<td>0111 0</td>
<td></td>
<td>a \lor b</td>
<td>0</td>
</tr>
<tr>
<td>0111 1</td>
<td></td>
<td>b[n/2 - 1 : 0]0^{n/2}</td>
<td>0</td>
</tr>
<tr>
<td>1010</td>
<td>*</td>
<td>0^{n-1}[(a) &lt; [b] : 1 : 0]</td>
<td>0</td>
</tr>
<tr>
<td>1011</td>
<td>*</td>
<td>0^{n-1}[(a) &lt; [b] : 1 : 0]</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.1: Specification of ALU operations

- **Logical operations.** At first sight, the result \(b[n/2 : 0]0^{n/2}\) might appear odd. This ALU function is later used to construct the upper half of an \(n\) bit constants using the immediate fields of an instruction.

- **Test and set instructions.** They compute an \(n\) bit result \(0^{n-1}z\) where only the last bit is interesting. It can obviously be computed by performing a subtraction in the AU and then testing the negative bit.

Figure 7.4 shows the fairly obvious data paths of an \(n\)-ALU. The missing signals are easily constructed. We subtract if \(a[i\geq 2]/f[1] = 01\). For test and set operations with \(a[i\geq 3] = 1\) we use for \(z\) the negative bit \(neg\) that we compute for unsigned numbers if \(a[i\geq 0] = 1\) and for signed numbers otherwise. The overflow bit can only differ from zero if \(a[i\geq 3]/a[i\geq 0] = 001\). The overflow bit is computed for unsigned numbers if \(a[i\geq 0] = 0\). Thus we have

\[
\begin{align*}
\text{sub} & = a[i\geq 2]a[i] \\
\text{z} & = \text{neg} \\
\text{u} & = a[i\geq 3]a[i\geq 0] \lor a[i\geq 3]/a[i\geq 0] \\
\text{ovfalu} & = \text{ovf} \land a[i\geq 3]/a[i\geq 2]a[i]
\end{align*}
\]

4.4 **Shifter**

\(n\)-shift operations have two operands:

- a bit vector \(a|n-1 : 0| \in B^n\) that is shifted

- a shift distance \(i \in [0 : n-1]\)

Shifting comes in five flavors: cyclcal left shift \(sl\), cyclical right shift \(src\), logical left shift \(sll\), logical right shift \(srl\) and arithmetic right shift \(sra\). The
result \( r \) of such an \( n \)-shift has \( n \) bits \( r[j] \) defined as

\[
\begin{align*}
slc(a, i)[j] &= a[j - i \mod n] \\
src(a, i)[j] &= a[j + i \mod n] \\
sll(a, i)[j] &= \begin{cases} a[j - i] & j \geq i \\ 0 & \text{otherwise} \end{cases} \\
srl(a, i)[j] &= \begin{cases} a[j + i] & j \leq n - 1 - i \\ 0 & \text{otherwise} \end{cases} \\
sra(a, i)[j] &= \begin{cases} a[j + i] & j \leq n - 1 - i \\ a_{n-1} & \text{otherwise} \end{cases}
\end{align*}
\]
or, equivalently

\[
\begin{align*}
slc(a,i) & = a[n - i - 1 : 0]a[n - 1 : n - i] \\
src(a,i) & = a[i - 1 : 0]a[n - i : i] \\
sll(a,i) & = a[n - i - 1 : 0]0^i \\
srl(a,i) & = 0^i a[n - i - 1 : 0] \\
sra(a,i) & = a_{n-1}a[n - 1 : i]
\end{align*}
\]

From the definition we immediately conclude

**Lemma 39.**

\[
src(a,i) = slc(a, n - i \mod n)
\]

**Proof**

\[
\begin{align*}
j + i & = j - (-i) \\
& \equiv j - (n - i) \mod n
\end{align*}
\]

Here we build only shifters for numbers \(n\), which are a power of two.

\[
n = 2^k, \ k \in \mathbb{N}
\]

Basic building blocks for all shifter constructions here are \((n,b)\)-cyclical left shifters or short \((n,b)\)-SLCs for \(b \in [1:n-1]\). They have

- inputs \(a[n - 1 : 0]\); the data to be shifted
- input \(s \in \mathbb{B}\) indicating whether to shift or not
- data outputs \(a'[n - 1 : 0]\) satisfying

\[
a' = \begin{cases} 
slc(a,b) & s = 1 \\
\text{a} & \text{otherwise}
\end{cases}
\]

Figure 75 shows a construction.

A cyclical \(n\)-left shifter or short \(n\)-SLC is a circuit with

- data inputs \(a[n - 1 : 0]\)
- control inputs \(b[k - 1 : 0]\); the binary representation of the shift distance.
- data outputs \(r[n - 1 : 0]\) satisfying

\[
r = slc(a, \langle s \rangle)
\]
\[a[n - 1 : n - l] \quad a[n - l - 1 : 0]\]

![Diagram](image)

Figure 75: Implementation of an \((n, b)\)-cyclical-left shifter

\[
a
\xrightarrow{n}
(n, 1)\text{-SLC}
\xrightarrow{n}
(b_0)
\]

\[
r^{(0)}
\]

\[
\vdots
\]

\[
(n, 2^i)\text{-SLC}
\xrightarrow{n}
b_i
\]

\[
r^{(i)}
\]

\[
\vdots
\]

\[
(n, n/2)\text{-SLC}
\xrightarrow{n}
b_{k-1}
\]

\[
r^{(k-1)}
\]

Figure 76: Implementation of a cyclic \(n\)-left shifter

Figure 76 shows a construction of cyclic \(n\)-left shifters as a stack of \((n, 2^i)\)-cyclical left shifters. An easy induction on \(i \in [0 : k - 1]\) shows

\[r^{(i)} = slc(a, \langle b[i : 0] \rangle)\]

A cyclic \(n\)-right-left shifter \(n\)-SRLC is a circuit with

- data inputs \(a[n - 1 : 0]\)
- data inputs \(s[k - 1 : 0]\); the binary representation of the shift distance.
- control input \(f \in \mathbb{B}\) indicating the shift direction
- data outputs \(r[n - 1 : 0]\) satisfying

\[
r = \begin{cases} 
  slc(a, \langle b \rangle) & f = 0 \\
  src(a, \langle b \rangle) & f = 1 
\end{cases}
\]
4.4. **SHIFTER**

![Diagram](image)

**Figure 77:** Implementation of an $n$-right-left shifter

**Figure 78:** Symbol of an $n$-shift unit

Figure 77 shows a construction. The output $c[k-1:0]$ of the $k$-incrementer satisfies

$$
\langle c \rangle = (\langle b \rangle + 1 \mod n)
= (n - \langle b \rangle \mod n)
$$

by the subtraction algorithm for binary numbers (lemma 15).

The output $d$ of the multiplexer then satisfies

$$
\langle d \rangle = \begin{cases} 
\langle b \rangle & f = 0 \\
 n - \langle b \rangle \mod n & f = 1 
\end{cases}
$$

The correctness of the construction now follows from lemma 39. An $n$-shift (see figure 78) unit $n$-SU has

- inputs $a[n-1:0]$; the data to be shifted
- inputs $s[k-1:0]$ determining the shift distance
• inputs $sf[1:0]$ determining the kind of shift to be executed
• outputs $sures[n-1:0]$ satisfying

$$suress = \begin{cases} 
    sl(a, b) & sf = 00 \\
    srl(a, b) & sf = 10 \\
    sra(a, b) & sf = 11 
\end{cases}$$

A construction is shown in figures 79 to c). Let

$$i = \langle b \rangle$$

The cyclic right left shifter in part a) produces output

$$r = \begin{cases} 
    a[n - i - 1 : 0]a[n - 1 : n - i] & sf[1] = 0 \\
    a[i - 1 : 0]a[n - i : i] & sf[1] = 1 
\end{cases}$$

The output of the circuit in figure 80 produces a mask

$$mask = \begin{cases} 
    0^{n-i} & sf[1] = 0 \\
    1^{n-i} & sf[1] = 1 
\end{cases}$$
For each index \( j \in [0 : n - 1] \) the multiplexer in figure 81 replaces the shifter output \( r[j] \) by the \( fill \) bit if this is indicated by the mask bit \( mask[j] \). Thus we get

\[
sures = \begin{cases} 
  a[n - i - 1 : 0]fill^i & sf[1] = 0 \\
  fill'a[n - i : i] & sf[1] = 1 
\end{cases}
\]

Setting

\[ fill = sf[0] \land a_{n-1} \]

we conclude

\[
sures = \begin{cases} 
  sll(a,i) & sf = 00 \\
  srl(a,i) & sf = 10 \\
  sra(a,i) & sf = 11 
\end{cases}
\]

### 4.5 Branch Condition Evaluation Unit

An \( n\)-BCE (see figure 82) has

- inputs \( a[n - 1 : 0], b[n - 1 : 0] \in \mathbb{B}^n \)
- inputs \( bf[3 : 0] \in \mathbb{B}^4 \) selecting the condition to be tested
- output \( bcrest \in \mathbb{B} \) specified by table 4.5
<table>
<thead>
<tr>
<th>b[3:0]</th>
<th>bcre</th>
<th>( \text{bcre} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>0</td>
<td>( a &lt; 0 )</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>( a \geq 0 )</td>
</tr>
<tr>
<td>100</td>
<td>*</td>
<td>( a = b )</td>
</tr>
<tr>
<td>101</td>
<td>*</td>
<td>( a \neq b )</td>
</tr>
<tr>
<td>110</td>
<td>*</td>
<td>( a &lt; 0 )</td>
</tr>
<tr>
<td>111</td>
<td>*</td>
<td>( a &gt; 0 )</td>
</tr>
</tbody>
</table>

Table 4.2: Specification of branch condition evaluation

![Diagram of circuit](image)

Figure 83: Computation of auxiliary signals in an \( n \)-branch condition evaluation unit

The auxiliary circuit in figure 83 computes obvious auxiliary signals satisfying

\[
\begin{align*}
    d &= b \land (b[3] \lor b[2]) \\
    &= \begin{cases} 
        b & \text{if } b[3 : 2] \neq 01 \\
        0 & \text{otherwise}
    \end{cases} \\
    eq &= \begin{cases} 
        a = b & \text{if } b[3 : 2] \neq 01 \\
        a = 0 & \text{otherwise}
    \end{cases} \\
    neq &= \overline{eq} \\
    lt &= [a] < 0 \\
    le &= [a] < 0 \lor a = b
\end{align*}
\]

The result can then be computed as
\[ \text{bcres} = \quad (b[3:1]) = 001 \land (lt \land b[0]) \lor \text{bcon}[0] \land \overline{b[0]} \\
\lor (b[3:2]) = 10 \land (b[1] \land eq \lor b[1] \land \overline{eq}) \\
\lor (b[3:2]) = 11 \land (b[1] \land le \lor b[1] \land \overline{le}) \\
= (b[3] \land b[2] \land b[1] \land (b[0] \oplus lt)) \\
\lor (b[3] \land b[2] \land (b[1] \oplus eq)) \\
\lor (b[3] \land b[2] \land (b[1] \oplus le)) \]
Chapter 5

A Basic Sequential MIPS Machine

We define the basic MIPS instruction set architecture (ISA) without delayed branch, interrupt mechanism and devices. The first section of this chapter is very short. It contains a very compact summary of the instruction set (and the assembly language) in the form of tables, which define the ISA if one knows how to interpret them. In the second section we provide a succinct and completely precise interpretation of the tables, leaving out only the coprocessor instructions and the system call instruction. From this we derive in the third section the hardware of a sequential - i.e. non pipelined - MIPS processor and provide a straightforward proof, that this processor construction is correct.

5.1 Tables

5.1.1 I-Type

In the following table: \( m = m_d(ea(c)) \) with \( ea(c) = rs(c) +_{32} sztimm(c) \)
## Chapter 5. A Basic Sequential MIPS Machine

<table>
<thead>
<tr>
<th>opc</th>
<th>Mnemonic</th>
<th>Assembler-Syntax</th>
<th>d</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data Transfer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 000</td>
<td>lb</td>
<td><code>lb rt rs imm</code></td>
<td>1</td>
<td>rt = sxt(m)</td>
</tr>
<tr>
<td>100 001</td>
<td>lh</td>
<td><code>lh rt rs imm</code></td>
<td>2</td>
<td>rt = sxt(m)</td>
</tr>
<tr>
<td>100 011</td>
<td>lw</td>
<td><code>lw rt rs imm</code></td>
<td>4</td>
<td>rt = m</td>
</tr>
<tr>
<td>100 100</td>
<td>lbu</td>
<td><code>lbu rt rs imm</code></td>
<td>1</td>
<td>rt = 0\textsuperscript{23}m</td>
</tr>
<tr>
<td>100 101</td>
<td>lhu</td>
<td><code>lhu rt rs imm</code></td>
<td>2</td>
<td>rt = 0\textsuperscript{16}m</td>
</tr>
<tr>
<td>101 000</td>
<td>sb</td>
<td><code>sb rt rs imm</code></td>
<td>1</td>
<td>m = rt[7:0]</td>
</tr>
<tr>
<td>101 001</td>
<td>sh</td>
<td><code>sh rt rs imm</code></td>
<td>2</td>
<td>m = rt[15:0]</td>
</tr>
<tr>
<td>101 011</td>
<td>sw</td>
<td><code>sw rt rs imm</code></td>
<td>4</td>
<td>m = rt</td>
</tr>
<tr>
<td></td>
<td>Arithmetic, Logical Operation, Test-and-Set</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>001 000</td>
<td>addi</td>
<td><code>addi rt rs imm</code></td>
<td></td>
<td>rt = rs + sxt(imm)</td>
</tr>
<tr>
<td>001 001</td>
<td>addiu</td>
<td><code>addiu rt rs imm</code></td>
<td></td>
<td>rt = rs + sxt(imm)</td>
</tr>
<tr>
<td>001 010</td>
<td>slti</td>
<td><code>slti rt rs imm</code></td>
<td></td>
<td>rt = (rs &lt; sxt(imm) ? 1 : 0)</td>
</tr>
<tr>
<td>001 011</td>
<td>sltu</td>
<td><code>sltu rt rs imm</code></td>
<td></td>
<td>rt = (rs &lt; sxt(imm) ? 1 : 0)</td>
</tr>
<tr>
<td>001 100</td>
<td>andi</td>
<td><code>andi rt rs imm</code></td>
<td></td>
<td>rt = rs &amp; zxt(imm)</td>
</tr>
<tr>
<td>001 101</td>
<td>ori</td>
<td><code>ori rt rs imm</code></td>
<td></td>
<td>rt = rs \lor zxt(imm)</td>
</tr>
<tr>
<td>001 110</td>
<td>xori</td>
<td><code>xori rt rs imm</code></td>
<td></td>
<td>rt = rs \oplus zxt(imm)</td>
</tr>
<tr>
<td>001 111</td>
<td>lui</td>
<td><code>lui rt imm</code></td>
<td></td>
<td>rt = imm\textsuperscript{16}</td>
</tr>
<tr>
<td></td>
<td>Branch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000 001</td>
<td>00000</td>
<td>bltz</td>
<td><code>bltz rs imm</code></td>
<td>pc = pc + (rs &lt; 0 ? imm00 : 4)</td>
</tr>
<tr>
<td>000 001</td>
<td>00001</td>
<td>bgez</td>
<td><code>bgez rs imm</code></td>
<td>pc = pc + (rs \geq 0 ? imm00 : 4)</td>
</tr>
<tr>
<td>000 100</td>
<td>00000</td>
<td>beq</td>
<td><code>beq rs rt imm</code></td>
<td>pc = pc + (rs = rt ? imm00 : 4)</td>
</tr>
<tr>
<td>000 101</td>
<td>00000</td>
<td>bne</td>
<td><code>bne rs rt imm</code></td>
<td>pc = pc + (rs \neq rt ? imm00 : 4)</td>
</tr>
<tr>
<td>000 110</td>
<td>00000</td>
<td>blez</td>
<td><code>blez rs imm</code></td>
<td>pc = pc + (rs \leq 0 ? imm00 : 4)</td>
</tr>
<tr>
<td>000 111</td>
<td>00000</td>
<td>bgtz</td>
<td><code>bgtz rs imm</code></td>
<td>pc = pc + (rs &gt; 0 ? imm00 : 4)</td>
</tr>
</tbody>
</table>
### 5.1.2 R-type

<table>
<thead>
<tr>
<th>opcode</th>
<th>fun</th>
<th>Mnemonic</th>
<th>Assembler-Syntax</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>000 00</td>
<td>sli</td>
<td>sli rd rt sa</td>
<td>rd = sli(rt,sa)</td>
</tr>
<tr>
<td>000000</td>
<td>000 01</td>
<td>srl</td>
<td>srl rd rt sa</td>
<td>rd = srl(rt,sa)</td>
</tr>
<tr>
<td>000000</td>
<td>000 11</td>
<td>sra</td>
<td>sra rd rt sa</td>
<td>rd = sra(rt,sa)</td>
</tr>
<tr>
<td>000000</td>
<td>000 10</td>
<td>slv</td>
<td>slv rd rt rs</td>
<td>rd = slv(rt,rs)</td>
</tr>
<tr>
<td>000000</td>
<td>000 11</td>
<td>srlv</td>
<td>srlv rd rt rs</td>
<td>rd = srlv(rt,rs)</td>
</tr>
<tr>
<td>000000</td>
<td>000 111</td>
<td>srav</td>
<td>srav rd rt rs</td>
<td>rd = srav(rt,rs)</td>
</tr>
</tbody>
</table>

### Arithmetic, Logical Operation

<table>
<thead>
<tr>
<th>opcode</th>
<th>fun</th>
<th>Mnemonic</th>
<th>Assembler-Syntax</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>100 00</td>
<td>add</td>
<td>add rd rs rt</td>
<td>rd = rs + rt</td>
</tr>
<tr>
<td>000000</td>
<td>100 01</td>
<td>addu</td>
<td>addu rd rs rt</td>
<td>rd = rs + rt</td>
</tr>
<tr>
<td>000000</td>
<td>100 10</td>
<td>sub</td>
<td>sub rd rs rt</td>
<td>rd = rs - rt</td>
</tr>
<tr>
<td>000000</td>
<td>100 11</td>
<td>subu</td>
<td>subu rd rs rt</td>
<td>rd = rs - rt</td>
</tr>
<tr>
<td>000000</td>
<td>100 100</td>
<td>and</td>
<td>and rd rs rt</td>
<td>rd = rs &amp; rt</td>
</tr>
<tr>
<td>000000</td>
<td>100 101</td>
<td>or</td>
<td>or rd rs rt</td>
<td>rd = rs \lor rt</td>
</tr>
<tr>
<td>000000</td>
<td>100 110</td>
<td>xor</td>
<td>xor rd rs rt</td>
<td>rd = rs \oplus rt</td>
</tr>
<tr>
<td>000000</td>
<td>100 111</td>
<td>nor</td>
<td>nor rd rs rt</td>
<td>rd = rs \neg rt</td>
</tr>
</tbody>
</table>

### Test Set Operation

<table>
<thead>
<tr>
<th>opcode</th>
<th>fun</th>
<th>Mnemonic</th>
<th>Assembler-Syntax</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>101 01</td>
<td>slt</td>
<td>slt rd rs rt</td>
<td>rd = (rs &lt; rt ? 1 : 0)</td>
</tr>
<tr>
<td>000000</td>
<td>101 11</td>
<td>sltu</td>
<td>sltu rd rs rt</td>
<td>rd = (rs &lt; rt ? 1 : 0)</td>
</tr>
</tbody>
</table>

### Jumps, System Call

<table>
<thead>
<tr>
<th>opcode</th>
<th>fun</th>
<th>Mnemonic</th>
<th>Assembler-Syntax</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>001 00</td>
<td>jr</td>
<td>jr rs</td>
<td>pc = rs</td>
</tr>
<tr>
<td>000000</td>
<td>001 01</td>
<td>jalr</td>
<td>jalr rd rs</td>
<td>rd = pc + 4 pc = rs</td>
</tr>
<tr>
<td>000000</td>
<td>001 100</td>
<td>syesc</td>
<td>syesc</td>
<td>System Call</td>
</tr>
</tbody>
</table>

### Coprocessor Instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>fun</th>
<th>Mnemonic</th>
<th>Assembler-Syntax</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>010000</td>
<td>10000</td>
<td>011 000</td>
<td>eret</td>
<td>eret</td>
</tr>
<tr>
<td>010000</td>
<td>001000</td>
<td>movg2s</td>
<td>movg2s rd rt</td>
<td>spr[rd] := spr[rt]</td>
</tr>
<tr>
<td>010000</td>
<td>000000</td>
<td>movs2g</td>
<td>movs2g rd rt</td>
<td>gpr[rt] := spr[rd]</td>
</tr>
</tbody>
</table>

### 5.1.3 J-type

<table>
<thead>
<tr>
<th>opc</th>
<th>Mnemonic</th>
<th>Assembler-Syntax</th>
<th>Effect</th>
</tr>
</thead>
</table>
| Jumps
| 000 010 | j | j iindex | pc = bin32(pc+4)[31:28]iindex00 |
| 000 011 | jal | jal iindex | R31 = pc + 4 pc = bin32(pc+4)[31:28]iindex00 |
5.2 MIPS ISA

5.2.1 Configuration and instruction fields

A basic MIPS configuration $c$ has only three user visible data structures (figure 84):

- $c.pc \in \mathbb{B}^{32}$: the program counter
- $c.gpr : \mathbb{B}^{5} \rightarrow \mathbb{B}^{32}$: the general purpose register file consisting of 32 registers, each 32 bits wide. For register addresses $x \in \mathbb{B}^{5}$ the content of general purpose register $x$ in configuration $c$ is denoted by $c.gpr(x) \in \mathbb{B}^{32}$
- $c.m : \mathbb{B}^{32} \rightarrow \mathbb{B}^{8}$: the processor memory. It is byte addressable; addresses have 32 bits. Thus for memory addresses $a \in \mathbb{B}^{32}$ the content of memory location $a$ in configuration $c$ is denoted by $c.m(a) \in \mathbb{B}^{8}$

Program counter and general purpose registers belong to the central processing unit (CPU).

Let $K$ be the set of all basic MIPS configurations. A mathematical definition of the ISA will be given by a function

$$\delta : K \rightarrow K$$

where

$$c' = \delta(c, reset)$$
is the configuration reached from configuration \( c \), if one instruction is executed. An ISA computation is a sequence \( (c') \) of ISA configurations with \( i \in \mathbb{N} \setminus \{0\} \) satisfying

\[
\begin{align*}
  c^1.pc &= 0^{32} \\
  c^{i+1} &= \delta (c^i)
\end{align*}
\]

i.e. initially the program counter points to address \( 0^{32} \) and in each step one instruction is executed. In the remainder of this section we specify the ISA simply by specifying function \( \delta \), i.e. by specifying \( c' = \delta (c) \) for all configurations \( c \).

For numbers \( y \in \mathbb{B}_n \) we abbreviate the binary representation of \( y \) with \( n \) bits as

\[ y_n = \text{bin}_n(y) \]

c.g. \( 1_8 = 00000001 \) and \( 3_8 = 00000011 \). For memories \( m : \mathbb{B}^{32} \to \mathbb{B}^8 \), addresses \( a \in \mathbb{B}^{32} \) and numbers \( d \) of bytes we denote the content of \( d \) consecutive memory bytes starting at address \( a \) by

\[ m_d(a) = m(a + 32 \cdot d_{32 - 32} 1_{32}) \circ \ldots \circ m(a + 32 1_{32}) \circ m(a) \]

The current instruction \( I(c) \) to be executed in configuration \( c \) is defined by the 4 bytes in memory addressed by the current program counter

\[ I(c) = c.m_4 (c.pc) \]

Because all instructions are 4 bytes long, one requires, that instructions are aligned on 4 word boundaries, or, equivalently that

\[ c.pc[1 : 0] = 00 \]

In case this condition is violated a so called misalignment interrupt is raised.

The six high order bits of the current instructions are called the op-code

\[ \text{opc}(c) = I(c)[31 : 26] \]

There are three instruction types: R-, J- and I-type. The current instruction type is determined by the following predicates

\[
\begin{align*}
  rtype(c) &\equiv \text{opc}(c) = 0^6 \\
  jtype(c) &\equiv \text{opc}(c) = 0^410 \lor \text{opc}(c) = 0^411 \\
  itype(c) &\equiv rtype(c) \lor jtype(c)
\end{align*}
\]

Depending on the instruction type, the bits of the current instruction are subdivided as shown in figure 85. Register addresses are specified in the following fields.
For R-type instructions, ALU-functions to be applied to the register operands can be specified in the function field

\[ \text{fun}(c) = I(c)[5 : 0] \]

Three kinds of immediate constants can be specified: the shift amount \( sa \) in R-type instructions, the immediate constant \( \text{imm} \) in I-type instructions and an instruction index \( \text{idex} \) in J-type (like jump) operations

\[ \text{sa}(c) = I(c)[10 : 6] \]
\[ \text{imm}(c) = I(c)[15 : 0] \]
\[ \text{idex}(c) = I(c)[25 : 0] \]

The immediate constant has 16 bits. In order to apply ALU functions to it, the constant can be extended with 16 high order bits in two ways: zero extension and sign extension

\[ \text{zxtimm}(c) = 0^{16}\text{imm}(c) \]
\[ \text{sztimm}(c) = I(c)[15]^{16}\text{imm}(c) \]

In case of sign extension, the value if the constant interpreted as a two's complement number does not change

\[ [\text{sztimm}(c)] = [\text{imm}(c)] \]
5.2. MIPS ISA

5.2.2 Instruction Decoding

For every mnemonic $mn$ of a MIPS instruction from the tables above we define a predicate $mn(c)$ which is true, if an $mn$ instruction is to be executed in configuration $c$. For instance

\[
\begin{align*}
lw(c) & \equiv \text{opc}(c) = 100011 \\
bitz(c) & \equiv \text{opc}(c) = 0^51 \land rt(c) = 0^5 \\
add(c) & \equiv \text{rtype}(c) \land \text{fun}(c) = 100000
\end{align*}
\]

The remaining predicates directly associated to the mnemonics of the assembly language are derived in the same way from the tables. We group the basic instruction set into 5 groups and define for each group a predicate that holds, if an instruction from that group is to be executed:

- **ALU-operations of I-type** are recognized by the leading three bits of the opcode, resp. $I(c)[31:29]$. ALU-operations of $R-type$ by the two leading bits of the function code, resp. $I(c)[5:4]$

\[
\begin{align*}
alu(c) & \equiv \text{rtype}(c) \land I(c)[31:29] = 001 \\
alu(c) & \equiv \text{itype}(c) \land I(c)[5:4] = 10 \\
alu(c) & = \alu(c) \lor \aluui(c)
\end{align*}
\]

- **shift unit operations** are of R-type and are recognized by the three leading bits of the function code. If bit $\text{fun}[2]$ of the function code is on, the shift distance is taken from a register content.

\[
\begin{align*}
su(c) & \equiv \text{rtype}(c) \land I(c)[5:3] = 000 \\
suw(c) & \equiv \text{su}(c) \land \text{fun}(c)[3]
\end{align*}
\]

- **loads and stores** are of I-type and are recognized by the three leading bits of the opcode

\[
\begin{align*}
l(c) & \equiv I(c)[31:29] = 100 \\
s(c) & \equiv I(c)[31:29] = 101 \\
ls(c) & \equiv l(c) \land s(c) \\
& = I(c)[31:30] = 10
\end{align*}
\]

- **Branches** have I-Type and are recognized by the three leading bits of the opcode.

\[
b(c) \equiv \text{itype}(c) \land I[31:29] = 000
\]

We define jumps in a brute force way

\[
\begin{align*}
jump(c) & \equiv jr(c) \lor jalr(c) \lor j(c) \lor jal(c) \\
jb(c) & \equiv jump(c) \lor b(c)
\end{align*}
\]
5.2.3 ALU-Operations

We can now go through the ALU-operations in the tables one by one and give them precise interpretations. We do this for two examples:

**add(c):** The table specifies the effect as \( rd = rs + rt \). This is to be interpreted as the corresponding register contents: on the right hand side of the equation for \( c \), i.e. before execution of the instruction; on the left hand side for \( c' \)

\[
c'.gpr(rd(c)) = c.gpr(rs(c)) +_{32} c.gpr(rt(c))
\]

Other register contents and the memory content do not change

\[
c'.gpr(x) = c.gpr(x) \quad \text{for} \quad x \neq rd(c)
\]

\[
c'.m = c.m
\]

The program counter is advanced by four bytes to the next instruction

\[
c'.pc = c.pc +_{32} 4_{32}
\]

**addi(c):** The second operand is now the sign extended immediate constant

\[
c'.gpr(x) = \begin{cases} 
  c.gpr(rs(c)) + 32 \text{sext}(c) & x = rt(c) \\
  c.gpr(x) & \text{otherwise}
\end{cases}
\]

\[
c'.m = c.m
\]

\[
c'.pc = c.pc +_{32} 4_{32}
\]

It is clear how to derive precise specifications for the remaining ALU-operations, but we take a shortcut exploiting the fact that we have already constructed an ALU that was specified in table 4.3.
This table defines functions \( alures(a, b, f, i) \) and \( ovf(a, b, f, i) \). As we do not treat interrupts (yet) we use only the first of these functions here. We observe that in all ALU operations a function of the ALU is performed. The left operand is always

\[
lop(c) = c.gpr(rs(c))
\]

For R-type operations, the right operand is the register specified by the \( rt \) field for R-type instructions. For I-type instructions it is the sign extended immediate operand if \( I(opc(c))[2] = I(c)[28] = 0 \) or zero extended immediate operand if \( I(c)[28] = 1 \). Thus we define an immediate fill bit \( ifill(c) \) extended immediate constant \( xtimm(c) \) by

\[
ifill(c) = \begin{cases} 
  \text{imm}(c)[15] & I(c)[28] = 0 \\
  0 & I(c)[28] = 1 
\end{cases}
\]

\[xtimm(c) = ifill(c)^{16}\text{imm}(c)\]

\[
rop(c) = \begin{cases} 
  c.gpr(rt(c)) & rtype(c) \\
  xtimm(c) & \text{otherwise} 
\end{cases}
\]

Comparing table 4.3 with the tables for I-type and R-type instructions we see that bits \( af[2 : 0] \) of the ALU control can be taken from the low order fields of the opcode for I-type instructions and from the low order bits of the function field for I-type instructions.

\[
af(c)[2 : 0] = \begin{cases} 
  I(c)[2 : 0] & rtype(c) \\
  I(c)[28 : 26] & \text{sonst} 
\end{cases}
\]

For bit \( f[3] \) things are more complicated. For R-type instructions it can be taken from the function code. For I-type instructions it must only be forced to 1 for the two test and set operations, which can be recognized by \( IR[28 : 27] = 01 \)

\[
af(c)[3] = \begin{cases} 
  rtype(c) \land I(c)[3] \lor /rtype(c) \land I(c)[28 : 27] = 01 \\
  rtype(c) \land I(c)[3] \lor /rtype(c) \land /I(c)[26] \land I_{27} 
\end{cases}
\]

The \( i \)-input of the ALU distinguishes for \( f[3 : 0] = 1111 \) between the \textit{lui}-instruction of I-type for \( i = 0 \) and the \textit{nor}-instruction of R-type for \( i = 1 \). Thus we set it to \( itype(c) \). The result of the ALU computed with these inputs is denoted by

\[
ares(c) = alures(lop(c), rop(c), alucon(c), itype(c))
\]
Depending on instruction type the destination register \( rdes \) is specified by the \( rd \) field or the \( rt \) field

\[
    rdes(c) = \begin{cases} 
        rd(c) & \text{type(c)} \\
        rt(c) & \text{sonst}
    \end{cases}
\]

A summary of all ALU operations is then

\[
    \begin{align*}
        alu(c) & \rightarrow \\
        c'.gpr(x) & = \begin{cases} 
            ares(c) & x = rdes(c) \\
            c.gpr(x) & \text{otherwise}
        \end{cases} \\
        c'.m & = c.m \\
        c'.pc & = c.pc + 32 \times 432
    \end{align*}
\]

### 5.2.4 Shift Unit Operations

Shift operations come in two flavors: i) for \( f[3] = 0 \) the shift distance \( sdist(c) \) is an immediate operand specified the \( sa \) field of the instruction. For \( f[3] = 1 \) the shift distance is specified by the last bits of the register specified by the \( rt \) field

\[
    sdist(c) = \begin{cases} 
        sa(c) & \text{fun(c)[3]} = 0 \\
        c.gpr(rs(c))[4 : 0] & \text{fun(c)[3]} = 1
    \end{cases}
\]

The left operand that is shifted is always the register specified by the \( rt \)-field

\[
    slop(c) = c.gpr(rt(c))
\]

and the control bits \( sf[1 : 0] \) are taken from the low order bits of the function field

\[
    sf(c) = I(c)[1 : 0]
\]

The result of the shift unit is computed with these inputs is denoted by

\[
    sres(c) = sure(slop(c), sdist(c), sf(c))
\]

For shift operations the destination register is always specified by the \( rd \) field. Thus the shift unit operations can be summarized as

\[
    \begin{align*}
        su(c) & \rightarrow \\
        c'.gpr(x) & = \begin{cases} 
            sres(c) & x = rd(c) \\
            c.gpr(x) & \text{otherwise}
        \end{cases} \\
        c'.m & = c.m \\
        c'.pc & = c.pc + 32 \times 432
    \end{align*}
\]

\(^1\text{Mnemonics with suffix v as 'variable'; one would expect instead for the other shifts a suffix i as 'immediate'}\)
5.2. MIPS ISA

5.2.5 Branch and Jump

A branch condition evaluation unit $BU$ was specified in table 4.5. It computes a function $bcres(a, b, bf)$. We use this function with the following parameters

$$
\begin{align*}
blop(c) &= c.gpr(rs(c)) \\
brop(c) &= c.gpr(rt(c)) \\
bf(c) &= I(c)[28; 26] \circ rt(c)[0] \\
&= I(c)[28 : 26]I[16]
\end{align*}
$$

and define the result of a branch condition evaluation as

$$
bres(c) = bcres(blop(c), brop(c), bf(c))
$$

The next pc $c'.pc$ is usually computed as $c.pc +_{32} 4_{32}$. This order is only changed in jump instructions or in brach instructions, where the branch is taken, i.e. the branch condition evaluates to 1. We define

$$
jbtaken(c) \equiv jump(c) \lor b(c) \land bres(c)
$$

In case of a jump or a branch taken, there are three possible jump targets

**Branch instructions** involve a relative branch. The pc is incremented by a branch distance

$$
b(c) \land bcres(c) \rightarrow \\
bdist(c) = imm(c)[15]^{14}imm(c)00 \\
btarget(c) = c.pc +_{32} bdist(c)
$$

Note that the branch distance is kind of a sign extended immediate constant, but due to the alignment requirement the low order bits of the jump distance must be 00. Thus one uses the 16 bits of the immediate constant for bits $[17:2]$ of the jump distance. Sign extension is used for the remaining bits. Note also that address arithmetic is modulo $2^n$. We have

$$
\langle c.pc \rangle + \langle bdist(c) \rangle = [c.pc] + |bdist(c)|\ mod\ 2^n
\quad = [c.pc] + |imm(c)|00
$$

Thus backward jumps are realized with negative $|imm(c)|$

**R-type jumps** for instructions $jr$ and $jalr$. The branch target is specified by the $rs$ field of the instruction.

$$
jr(c) \lor jalr(c) \rightarrow \\
btarget(c) = c.gpr(rs(c))
$$
J-Type jumps \( j \) and \( jal \). The branch target is computed in a rather peculiar way: i) the pc is incremented by 4. Then bits \([27:0]\) are replaced by the \( \text{index} \) field of the instruction

\[
j(c) \lor jal(c) \rightarrow \\
btarget(c) = (c.pc + 32 \ 4_{32})[31 : 29]\text{\text{index}}(c)
\]

Now we can define the next pc computation for all instructions as

\[
btarget(c) = \begin{cases} 
  c.pc + 32 \ \text{imm}(c)[15:14]\text{imm}(c)00 \ b(c) \\
  c.gpr(rs(c)) \ jr(c) \lor jalr(c) \\
  (c.pc + 32 \ 4_{32})[31 : 29]\text{\text{index}}(c) \ j(c) \lor jal(c)
\end{cases}
\]

\[
c'.pc = \begin{cases} 
  btarget(c) \ jbtaken(c) \\
  c.pc + 32 \ 4_{32}
\end{cases}
\]

Jump and Link The two jump instructions \( jal \) and \( jalr \) are used to implement calls of procedures. Besides setting the pc to the branch target they prepare the so called link address return by saving the incremented pc

\[
\text{linkad}(c) = c.pc + 32 \ 4_{32}
\]

in a register. For the R-type instruction \( jalr \) this register is specified by the \( rs \) field. J-type instruction \( jal \) does not have a \( rs \) field, and the incremented pc is stored in register \( 31 \) (\( = (1^5) \)). Branch and jump instructions do not change the memory.

For the update of registers in branch and jump instructions we therefore have

\[
bj(c) \rightarrow \\
\begin{cases} 
  \text{linkad}(c) \ jalr(c) \land x = rs(c) \lor jal(c) \land x = 1^5 \\
  \text{gpr}(x) \ \text{otherwise}
\end{cases}
\]

\[
c'.m = c.m
\]

5.2.6 Sequences of consecutive memory bytes

A byte is a string \( x \in \mathbb{B}^8 \). Let \( n = 8 \cdot k \) be a multiple of 8, let \( a \in \mathbb{B}^n \) be a string consisting of \( k \) bytes. For \( i \in [k-1:0] \) we define byte \( i \) of string \( a \) as

\[
\text{byte}(i, a) = a[8 \cdot (i + 1) - 1 : 8 \cdot i]
\]

A trivial observation is

Lemma 40. Let \( a \in \mathbb{B}^8 \), let \( b \in \mathbb{B}^d \) and let \( c = a \circ b \). Then

\[
\forall i \in [0 : d] : \text{byte}(i, c) = \begin{cases} 
  a & i = d \\
  \text{byte}(i, b) & i < d
\end{cases}
\]
Proof:

\[
\text{byte}(i, c) = a[8 \cdot (i + 1) - 1 : 8 \cdot i]
\]

\[
= \begin{cases} 
  a & i = d \\
  b[8 \cdot (i + 1) - 1 : 8 \cdot i] & i < d 
\end{cases}
\]

\[
= \begin{cases} 
  a & i = d \\
  \text{byte}(i, b) & i < d 
\end{cases}
\]

The state of byte addressable memory with 32 bit addresses is modeled as a mapping

\[
m : \mathbb{B}^{32} \rightarrow \mathbb{B}^8
\]

where for each address \( x \in \mathbb{B}^{32} \) one interprets \( m(x) \in \mathbb{B}^8 \) as the current value of memory location \( x \). We define the content \( m_d(x) \) of \( d \) consecutive locations starting at address \( x \) informally by

\[
m_d(x) = m(x + 32 (d - 1)_{32}) \circ \ldots \circ m(x)
\]

and formally by

\[
m_1(x) = m(x)
\]

\[
m_{d+1}(x) = m(x + 32 d_{32}) \circ m_d(x)
\]

The following simple lemma allows to localize bytes in sequences of consecutive memory.

**Lemma 41.**

\[
\forall i < d : \text{byte}(i, m_d(x)) = m(x + 32 i_{32})
\]

Proof by induction on \( d \). For \( d = 1 \) we have \( i = 0 \). Thus \( i_{32} = 0_{32} \) and

\[
\text{byte}(0, m_1(x)) = m(x) = m(x + 32 0_{32}) = m(x + 32 i_{32})
\]

For the induction step from \( d \) to \( d + 1 \) we have by lemma 40 and the induction hypothesis for all \( i < d + 1 \)

\[
\text{byte}(i, m_{d+1}(x))
\]

\[
= \begin{cases} 
  m(x + 32 d_{32}) & i = d \\
  \text{byte}(i, m_d(x)) & i < d 
\end{cases}
\]

\[
= \begin{cases} 
  m(x + 32 i_{32}) & i = d \\
  m(x + 32 i_{32}) & i < d 
\end{cases}
\]

\[
= m(x + 32 i_{32})
\]
5.2.7 Loads and Stores

Load and Store operations access a certain number $d(c) \in \{1, 2, 4\}$ of bytes of memory starting at a so called effective address $ea(c)$. Letters $b, h$ and $w$ in the mnemonics define the width: $b$ stands for $d = 1$ resp. a byte access; $h$ stands for $d = 2$ resp. a half word access and $w$ stands for $d = 4$ resp. a word access. Inspection of the instruction tables gives

$$
\begin{align*}
    d(c) = & \begin{cases} 
        1 & IR[26] = 0 \\
        2 & IR[27 : 26] = 01 \\
        4 & IR[27 : 26] = 11 
    \end{cases}
\end{align*}
$$

Addressing is always relative to a register specified by the $rs$-field. The offset is specified by the immediate field

$$
ea(c) = c.gpr(rs(c)) +_{32} sxtimm(c)
$$

Note that the immediate constant is sign extended, thus negative offsets can be realized in the same way as negative branch distances. Effective addresses are required to be aligned. If we interpret them as binary numbers they have to be divisible by the width

$$d(c)|(ea(c))$$

or, equivalently

$$ls(c) \land d(c) = 2 \rightarrow ea(c)[0] = 0 \quad , \quad ls(c) \land d(c) = 4 \rightarrow ea(c)[1 : 0] = 00$$

If this condition is violated a misalignment interrupt $mal$ is raised.

Stores Recall that for words $r \in \mathbb{B}^{32}$ and $i \in [3 : 0]$ we defined byte $i$ of $r$ as

$$\text{byte}(i, r) = r[8 \cdot (i + 1) - 1 : 8 \cdot i]$$

A store instruction takes the low order $d(c)$ bytes of the register specified by the $rt$-field and stores them as $m_{d(c)}(ea(c))$. Other memory bytes and register values are not changed. The pc is incremented by 4 (but we have already defined that).

$$\begin{align*}
    s(c) & \rightarrow \\
    c'.m(x) & = \begin{cases} 
        \text{byte}(i, c.gpr(rt(c))) & x = ea(c) +_{32} i_{32} \land i < d(c) \\
        c.m(x) & \text{otherwise}
    \end{cases} \\
    c'.gpr & = c.gpr
\end{align*}
$$

\footnote{A word of caution in case you plan to enter this into a CAV system: the first case...}
5.2. MIPS ISA

**Loads** Loads like stores access $d(c)$ bytes of memory starting at address $ea(c)$. The result is stored in the low order $d(c)$ bytes of the destination register, which is specified by the $rt$-field of the instruction. This leaves $32 - 8 \cdot d(c)$ bits of the destination register to be filled by some bit $fill(c)$. For unsigned loads (with a suffix u in the mnemonics) the fill bit is zero; otherwise it is sign extended by the leading bit of $c.m_{d(c)}(ea(c))$. In this way a load result $lres(c) \in \mathbb{B}^{32}$ is computed and the general purpose register file.

Other registers and the memory are not updated

\[
\begin{align*}
    u(c) &= I[28] \\
    fill(c) &= \begin{cases} 
        0 & \text{if } u(c) \\
        c.m(ea(c) + 32 \cdot d(c) - 32 \cdot l32)[7] & \text{otherwise}
    \end{cases} \\
    lres(c) &= fill(c)^{32-8 \cdot d(c)} \cdot c.m_d(ea(c)) \\
    s(c) &\rightarrow \\
    c'.gpr(x) &= \begin{cases} 
        lres(c) & x = rt(c) \\
        c.gpr(x) & \text{otherwise}
    \end{cases} \\
    c'.m &= c.m
\end{align*}
\]

of the 'definition' of $c'.m(z)$ is very well understandable for humans, but actually it is a shorthand for the following: if

\[
\exists i : z = ea(c) + 32 \cdot i_{32}
\]

then update $c.m(z)$ with the hopefully unique $i$ satisfying this condition. In this case we can compute this $i$ by solving the equation

\[
z = ea(c) + 32 \cdot i_{32}
\]

resp.

\[
\langle x \rangle = \langle (ea(c) + i \mod 2^{12}) \rangle
\]

From alignment we conclude

\[
\langle ea(c) \rangle + i \leq 2^{12} - 1
\]

Hence

\[
\langle (ea(c) + i \mod 2^{12}) \rangle = \langle (ea(c)) + i \rangle
\]

And we have to solve

\[
\langle x \rangle = \langle ea(c) \rangle + i
\]

as

\[
i = \langle x \rangle - \langle ea(c) \rangle
\]

This turns the above definition into

\[
c'.m(z) = \begin{cases} 
    \text{byte}(\langle x \rangle - \langle ea(c) \rangle), c.gpr(rt(c)) & \langle x \rangle - \langle ea(c) \rangle \in [0 \cdot d(c) - 1] \\
    c.m(z) & \text{otherwise}
\end{cases}
\]

which is not so readable for humans.
5.2.8 ISA Summary

We collect all previous definitions of destination registers for the general purpose register file into

\[
cad(c) = \begin{cases} 
1^6 & jal(c) \\
rd(c) & alu(c) \land rtype(c) \\
rt(c) & \text{otherwise}
\end{cases}
\]

Also we collect the data gprin to be written into the general purpose register file. For technical reasons we define on the way an intermediate result \( C \).

\[
C(c) = \begin{cases} 
\text{sres(c)} & \text{su(c)} \\
\text{linkad(c)} & jal(c) \lor jalr(c) \\
\text{ares(c)} & \text{otherwise}
\end{cases}
\]

\[
gprin(c) = \begin{cases} 
\text{lres(c)} & l(c) \\
\text{C(c)} & \text{otherwise}
\end{cases}
\]

Finally we collect in a general purpose register write signal all situations, when some general purpose register is updated

\[
gprw(c) \equiv alu(c) \lor su(c) \lor l(c) \lor jal(c) \lor jalr(c)
\]

Now we can summarize the MIPS ISA in three rules concerning the updates of pc, general purpose registers and memory:

\[
c'.pc = \begin{cases} 
b\text{target}(c) & b\text{taken}(c) \\
c.pc + 32 \times 32 & \text{otherwise}
\end{cases}
\]

\[
c'.gpr(x) = \begin{cases} 
g\text{prin}(c) & x = cad(c) \land gprw(c) \\
g\text{pr}r(c) & \text{otherwise}
\end{cases}
\]

\[
c'.m(x) = \begin{cases} 
\text{byte}(i, g\text{pr}(r\text{s}(c)))) & x = ea(c) + 32 \times 32 \land i < d(c) \\
c.m(x) & \text{otherwise}
\end{cases}
\]

5.3 A Sequential Processor Design

From the ISA spec we derive a hardware implementation of the basic MIPS processor. It will execute every MIPS instruction in a single hardware cycle, and it will be so close to the ISA specification that the correctness proof is reduced to a very simple bookkeeping exercise. This basic implementation however, is far from naive. In the following chapter we turn this implementation into a provably correct pipelined processor design with almost ridiculously little effort.
5.3. A SEQUENTIAL PROCESSOR DESIGN

Figure 86: Line address x.l and offset x.o of a byte address x

5.3.1 Software Conditions

As was required in the ISA specification, the hardware implementation only needs to work if all memory accesses of the ISA computation (c') are aligned, i.e.

\[ \forall i > 0 : \quad c'.pc[1 : 0] = 00 \]
\[ \land s(c') \rightarrow \]
\[ (d(c') = 1 \rightarrow ea(c')[0] = 0 \]
\[ \land d(c') = 2 \rightarrow ea(c')[1 : 0] = 00) \]

As suggested by figure 86 we divide addresses \( a \in \mathbb{B}^{32} \) into line address \( a.l \in \mathbb{B}^{29} \) and offset \( a.o \in \mathbb{B}^{3} \) by

\[ a.l = a[31 : 3] \]
\[ a.o = a[2 : 0] \]

For the time being we will assume that there is a code region \( CR \subset \mathbb{B}^{29} \) such that all instructions are fetched from addresses with a line address in \( CR \). We also assume that there is a data region \( DR \subset \mathbb{B}^{29} \) such that all addresses off loads and stores have a line address in \( DR \)

\[ \forall i : c'.pc.l \in CR \]
\[ \forall i : ls(c') \rightarrow ea(c').l \in DR \]

For the time being we will also assume that these regions are disjoint

\[ DR \cap CR = \emptyset \]

The hardware \( h \) of the implementation will have four components

- program counter \( h.ps \in \mathbb{B}^{32} \)
- general purpose register file \( h.gpr : \mathbb{B}^{5} \rightarrow \mathbb{B}^{32} \)
- a double word addressable data memory \( h.dm : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64} \). In later constructions it is replaced by a data cache. Here it is a multi-bank-RAM.
- a double word addressable instruction memory \( h.im : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64} \). In later constructions it is replaced by a data cache. Here it is a ROM-RAM.
Due to the assumption that code and data region are disjoint we will not have to worry (yet) about keeping instruction and data memory consistent. Wider memory speeds up aligned loads and stores of half words and words and will later speed up communication between caches and main memory. For the hardware, this comes at the price of shifters for loading or storing words, half words or bytes. We also need to develop some machinery for tracking the byte addressed data in line addressable memory.

5.3.2 Embedding byte addressable memory into line addressable memory

Let \( m : \mathbb{B}^{32} \rightarrow \mathbb{B}^8 \) be a byte addressable memory like \( c.m \) in the ISA specification, and let \( cm : \mathbb{B}^{39} \rightarrow \mathbb{B}^{54} \) be a line addressable memory like \( h.im \) and \( h.dm \) in the intended hardware implementation. Let \( A \subseteq \mathbb{B}^{20} \) be a set of line addresses like \( CR \) and \( DR \). We define in a straightforward way a relation \( cm \sim_A m \) stating that with respect to the addresses in \( A \) memory \( m \) is embedded in memory \( cm \) by

\[
\text{for all } a \in A : cm(a) = m_{8}(a0^3)
\]

thus - illustrating with dots - each line of memory \( cm \) contains 8 consecutive bytes of memory \( m \) namely

\[
cm(a) = m(a + 32732) \ldots m(a)
\]

We are interested to localize the single bytes of sequences \( m_d(x) \) in the line addressable memory \( cm \). We are only interested in access widths, which are powers of two nd at most 8

\[
d \in \{2^k : k \in [0 : 3]\}
\]

Also we are only interested in so called accesses \( (x, d) \) which are aligned in the following sense: if \( d = 2^k \) with \( k \geq 1 \) (i.e. to more than a single byte), then the last \( k \) bits of address \( x \) must all be zero

\[
d = 2^k \land k \geq 1 \rightarrow x[k - 1 : 0] = 0^k
\]

For accesses of this nature and \( i < d \) the expressions \( x + 32 i_{32} \), that are used in lemma 41 to localize bytes of \( m_d(x) \) in byte addressable memory have three very desirable properties: i) their numerical value is at most 7, hence ii) computing their representative mod 8 in \( B_3 \) gives the right result and iii) all bytes are embedded in the same cache line. This is shown in the following technical lemma

**Lemma 42.** Let \((x, d)\) be aligned and \( i < d \). Then

1. \[
\langle x. a \rangle + i < 7
\]
2. \[
\langle x.0 +_3 i_3 \rangle = \langle x.o \rangle + i
\]

3. \[
x +_{32} i_{32} = x.l \circ (x.0 +_3 i_3)
\]

Proof:
1. by alignment and because \(i < d = 2^k\) we have
\[
\langle x.o \rangle + i = \langle x[2 : k] \circ x[k - 1 : 0] \rangle + i
\]
\[
= \langle x[2 : k] \circ 0^k \rangle + i \leq 7 - (2^k - 1) + d - 1
\]
\[
= 7
\]

2. by the definition of \(+_3\) and part 1 of the lemma we have
\[
\langle x.o + i_3 \rangle = \langle (x.o) + (i_3) \mod 8 \rangle
\]
\[
= \langle (x.o) + i \mod 8 \rangle
\]
\[
= \langle x.o \rangle + i
\]

3. We write
\[
x = x.l \circ x.o
\]
\[
i_{32} = 0^{29} \circ i_3
\]

Adding the offset- and the line components separately we get by part 2 of the lemma
\[
\langle x.o \rangle + (i_3) = \langle 0 \circ (x.o +_3 i_3) \rangle
\]
\[
\langle x.l \rangle + (0^{29}) = \langle x.l \rangle
\]

Because the carry of the addition of the offsets to position 4 is 0, we get from lemma 12:
\[
\langle x \rangle + (i_{32}) = \langle x.l \circ (x.o +_3 i_3) \rangle < 2^{32}
\]

Hence
\[
\langle \langle x \rangle + (i_{32}) \rangle \mod 2^{32} = \langle x.l \circ (x.o +_3 i_3) \rangle
\]

Applying \(bin_{32}(\quad)\) to this equation proves part three of the lemma

In lemma 41 we showed for all accesses (aligned or not)
\[
\forall i < d : \text{byte}(i, m_d(x)) = m(x +_{32} i_{32})
\]

For aligned accesses \((x, d)\) we can specialize with the last lemma this to
\[
\forall i < d : \text{byte}(i, m_d(x)) = m(x.l \circ (x.o +_3 i_3) \quad (5.1))
\]

This allows a reformulation of the embedding relation \(\sim_A\):
Lemma 43. Relation \( cm \sim_A m \) holds iff for all byte addresses \( x \in B^{32} \) with \( x.l \in A \) and for all \( i < 8 \):

\[
byte((x.o), cm(x.l)) = m(x)
\]

Proof. Assume for line addresses \( a \in B^{29} \) we have

\[
a \in Acm(a) = m_b(a0^3)
\]

Then access \( (a0^3, 8) \) is aligned and equation 5.1 can be reformulated for the single bytes:

\[
\forall i < 8 : 
byte(i, dm(a)) = byte(i, m_b(a0^3)) = m(a \circ i_3)
\]

Now we rewrite byte address \( a \circ i_3 \) as

\[
a \circ i_3 = x = x.l \circ x.o
\]

and get

\[
byte((x.o), cm(x.l)) = m(x)
\]

Finally, we can formulate for aligned accesses \((x, d)\), how the single bytes of consecutive sequences \( m_d(x) \) are embedded in memory \( cm \)

Lemma 44. Let \((x, d)\) be aligned and \( i < d \). Then

\[
byte(i, m_d(x)) = byte((x.o) + i, cm(x.l))
\]

Proof:

\[
byte(i, m_d(x)) = m(x.l \circ (x.o + i \cdot i_3)) \quad (eq. 5.1)
\]

\[
= byte((x.o + i \cdot i_3), cm(x.l)) \quad (\text{lemma 43})
\]

\[
= byte((x.o) + i, cm(x.l)) \quad (\text{lemma 42})
\]

For aligned word accesses \((d = 4)\) and indices \( i < 3 \) we get an important special case

\[
byte(i, m_4(x)) = byte((x[2]00) + i, cm(x.l)) = byte(4 \cdot x[2] + i, cm(x.l))
\]

\[
= \begin{cases} 
byte(i, cm(x.l)) & x[2] = 0 \\
byte(4 + i, cm(x.l)) & x[2] = 1 
\end{cases}
\]

Concatenating bytes we get

Lemma 45. Let \( x \in B^{32} \) and \( x[1:0] = 00 \). Then

\[
m_4(x) = \begin{cases} 
\text{cm(x.l)[31:0]} : x[2] = 0 \\
\text{cm(x.l)[63:32]} : x[2] = 1 
\end{cases}
\]
5.3.3 Defining Hardware Correctness for the Processor Design

We define in a straightforward way a simulation relation $\text{sim}(c, h)$ stating that hardware configuration $h$ encodes or ISA configuration $c$ by

$$\text{sim}(c, h) \equiv$$

1. $h.\text{pc} = c.\text{pc}$
2. $h.\text{gpr} = c.\text{gpr}$
3. $c.m \sim_{CR} h.\text{im}$
4. $c.m \sim_{DR} h.\text{dm}$

i.e. every hardware memory location $h.\text{im}(a)$ for $a000 \in CR$ and $h.\text{dm}(a)$ for $a000 \in DR$ contains the contents of eight ISA memory locations:

$$c.m(a1110, \ldots, c.m(a000) = \begin{cases} h.\text{im}(a) & a000 \in CR \\ h.\text{dm}(a) & a000 \in DR \end{cases}$$

By lemma 43 that the last condition is equivalent to

$$\forall x \in \mathbb{B}^{32} : x.l \in DR \rightarrow c.m(x) = \text{byte}((x[2 : 0], h.\text{dm}(x[31 : 3])))$$

We will construct the hardware such that one ISA instruction is emulated in every hardware cycle and that we can show

**Lemma 46.**

$$\text{sim}(c, h) \rightarrow \text{sim}(c', h')$$

This is obviously the inductions step in a proof of

**Lemma 47.** There is an initial ISA configuration $c^0$ such that

$$\forall i > 0 : \text{sim}(c^i, h^i)$$

At first glance it seems that the lemmas are utterly useless, because after power up hardware registers come up with unknown binary content. With unknown content of the instruction memory content of the code region one does not know the program that is executed, and thus one cannot prove alignment and the disjointness of code and data regions. That is of course a very real problem whenever one tries to boot any real machine. We have already seen the solution: for the purpose of booting the code region occupies the bottom part of the address range

$$CR = \{0^{32-r}b : b \in \mathbb{B}^r\}$$
for some small $r < 32$. The instruction memory is therefore realized as a combined $(64, r, 32)$-RAM-ROM. The content of the ROM-portion is known after power up and contains the boot loader. Thus lemma 47 works at least literally for programs in ROM. After the boot loader has loaded more known programs in some new code region $CR'$ one can discharge with extra arguments the hypotheses of lemma 46 for the new code region.

As already mentioned earlier, our hardware construction will closely parallel the ISA specification, and there will be many signals $X$ occurring both in the ISA specification and in the hardware implementation. We will distinguish them by their argument. $X(c)$ is the ISA signal whereas $X(h)$ is the hardware signal.
5.3.4 Stages of Instruction Execution

Aiming at pipelined implementations later on, we construct the basic implementation of a MIPS processor in a very structured way. We split instruction execution into 5 stages, that we describe first an a preliminary and somewhat informal way (see figure 87).

- : IF: instruction fetch. The program counter \( pc \) is used to access the instruction memory \( im \) in order to fetch the current instruction \( I \)

- : ID: instruction decode.
  
  - In an instruction decoder predicates \( p \) and functions \( f \) depending only on the current instruction are computed. The predicates \( p \) correspond to the predicates in the ISA specification. Functions \( f \) include the function bits \( af \) and \( sf \) controlling ALU and shift unit as well as the extended immediate constant \( extimm \). Some trivial functions \( f \) select simply of some fields \( F \) of the instruction \( I \) like \( rs \) and \( rt \).
  
  - the general purpose register file is accessed with addresses \( rs \) and \( rt \). For the time being we call the result \( A = gpr(rs) \) and \( B = gpr(rt) \).
  
  - Result \( A \) is used to compute the next program counter \( h'.pc \) in a next-pc environment

- : EX execute. Using only results from the ID stage the following results are computed by the following circuits
  
  - the link address \( pc \) linkad in the link-instructions by an incrementer
  
  - the result \( ares \) of the ALU by an ALU-environment
  
  - the result \( sures \) of the shift unit by a shift unit environment
  
  - the preliminary input \( C \) for the general purpose register file from \( linkad, ares \) and \( sures \) by a small multiplexer tree.
  
  - effective address \( ea \) for loads and stores by an adder
  
  - the shifted operand \( dmin \) and the byte write signals \( bw[i] \) of the multi-bank-RAM \( dm \) for store instructions in an sh4s-environment\(^3\)

- : M: memory. Store instructions update the data memory \( dm \). For store instructions, line \( ea.l \) of the data memory is accessed. The result is visible in \( dmout \).

\(^3\)sh4l is a shorthand for 'shift for load'
**WB**: write back. For load instructions the output $dmout$ of the data memory load is shifted in an sh4l-environment and if necessary modified by a fill bit. The result $bres$ is combined with $C$ to the data input $gprin$ of the general purpose register file. The $gpr$ is updated.

### 5.3.5 Initialization

PC initialization and the instruction memory environment (with memory $im$ and a multiplexer) is shown in figure 88. We have

$$h^1.pc = 0^{32} = c^1.pc$$

Thus condition 1 of relation $sim$ holds for $i = 1$. We take no precautions to prevent writes to $h.gpr$ or $h.dm$ during cycle 0 and define

$$c^1.gpr = h^1.gpr$$

$$c^1_k(a000) = \begin{cases} h^0.im(a) & a000 \in CR \\ h^1.dm(a) & a000 \in DR \end{cases}$$

As the code region lies in ROM we have

$$\forall i : h^0.im = h^i.im$$

and can conclude

$$sim(c^1, h^1)$$

From now on let $i > 0$ and $c = c^1$, $h = h^i$ and assume $sim(c, h)$. When we invoke part $k$ of the simulation relation for $k \in [1 : 4]$, we will abbreviate this as $(sim.k)$. When we argue about hardware construction and semantics of hardware components like memories, we abbreviate by $(II)$.

### 5.3.6 Instruction Fetch

The treatment of the instruction fetch stage is short. The instruction memory is addressed with bits

$$ima(h) = h.pc[31 : 3] = h.pc.l$$
5.3. A SEQUENTIAL PROCESSOR DESIGN

It satisfies

\[
\begin{align*}
\text{h.pc}[31 : 3] &= \text{c.pc}[31 : 3] \quad \text{(sim.1)} \\
\in CR \\
\text{h.im}(\text{h.pc}[31 : 3]) &= \text{c.m}_8(\text{c.pc}[31 : 3]000) \quad \text{(sim.3)} \\
&= \text{c.m}_4(\text{c.pc}[31 : 3]100 \circ \text{c.m}_4(\text{c.pc}[31 : 3]000)
\end{align*}
\]

Using lemma 45 we conclude that the hardware instruction \(I(h)\) fetched by the circuitry in figure 1 is

\[
I(h) = \begin{cases} 
\text{h.im}(\text{h.pc}[31 : 3])[63 : 32] & \text{h.pc}[2] = 1 \\
\text{h.im}(\text{h.pc}[31 : 3])[31 : 0] & \text{h.pc}[2] = 0
\end{cases}
\]

\[
= \begin{cases} 
\text{c.m}_4(\text{c.pc}[31 : 3]100) & \text{c.pc}[2] = 1 \\
\text{c.m}_4(\text{c.pc}[31 : 3]000) & \text{c.pc}[2] = 0
\end{cases}
\]

\[
= \text{c.m}_4(\text{c.pc}) \quad \text{(alignment)}
\]

\[
= I(c)
\]

Thus we have

**Lemma 48.**

\[
I(h) = I(c)
\]

5.3.7 Instruction Decoder

The instruction decoder belongs to the instruction decode stage. As shown in figure 89 it computes the hardware version of functions \(f(c)\) that only
depend on the current instruction $I(c)$, i.e. which can be written as
\[ f(c) = f'(I(c)) \]

For example
\[ rtype(c) \equiv I(c)[31 : 26] = 0^6 \]
\[ rtype'(x[31 : 0]) \equiv x[31 : 26] = 0^6 \]
\[ rtype(c) = rtype'(I(c)) \]

or
\[ rd(c) = I(c)[15 : 11] \]
\[ rd'(x[31 : 7]) = x[15 : 11] \]
\[ rd(c) = rd'(I(c)) \]

**Predicates** This trivial transformation however allows in a straightforward way to construct circuits for all predicates $p(c)$ from the ISA specification that depend only on the current instruction:

- construct a boolean formula for $p'$. This is always possible by lemma 20. In the above example
  \[ rtype'(x) \equiv /x[32]/x[31]/x[30]/x[29]/x[28]/x[27]/x[26] \]

- translate the formula into a circuit and connect the inputs of the circuit to the hardware instruction register. The output $p(h)$ of the circuit satisfies
  \[ p(h) = p'(I(h)) \]
  \[ = p'(I(c)) \quad \text{(lemma 48)} \]
  \[ = p(c) \]

Thus we have

**Lemma 49.** For all predicates $p$ depending only on the current instruction:
\[ p(h) = p(c) \]

**Instruction Fields** All instruction fields $F$ have the form
\[ F(c) = I(c)[m : n] \]

Compute the hardware version as
\[ F(h) = I(h)[m : n] \]
\[ = I(c)[m : n] \quad \text{(lemma 48)} \]
\[ = F(c) \]

and we have
5.3. A SEQUENTIAL PROCESSOR DESIGN

Figure 90: c address computation

Lemma 50. For all instruction fields F:

\[ F(h) = F(c) \]

C Address The output \( cad(h) \) in figure 90 computes the C address for the general purpose register file. By lemmas 49 and 50 it satisfies

\[
cad(h) = \begin{cases} 
15 & jal(h) \\
rd(h) & alu(h) \wedge rtype(h) \\
rt(h) & \text{otherwise}
\end{cases}
\]

\[
cad(c) = \begin{cases} 
15 & jal(c) \\
rd(c) & alu(c) \wedge rtype(c) \\
rt(c) & \text{otherwise}
\end{cases}
\]

Extended immediate constant The fill bit \( ifill(c) \) is a predicate and \( imm(c) \) is a field of the instruction. Thus we can compute the extended immediate constant in hardware as

\[
xtimm(h) = ifill(h)^{15}imm(h) = ifill(c)imm(c) \quad (\text{lemmas 49 and 50}) = xtimm(c)
\]

Thus we have

Lemma 51.

\[ xtimm(h) = xtimm(c) \]
Function fields for ALU, SU and BCE  
Figure 91 shows the computation of the function fields $af, i, sf$ and $bf$ for the ALU, the shift unit and the branch condition evaluation unit.

Outputs $af(h)[2 : 0]$ satisfy by lemmas 49 and 50:

$$af(h)[2 : 0] = \begin{cases} 
I(h)[2 : 0] & \text{rtype}(h) \\
I(h)[28 : 26] & \text{otherwise}
\end{cases}$$

$$= \begin{cases} 
I(c)[2 : 0] & \text{rtype}(c) \\
I(c)[28 : 26] & \text{otherwise}
\end{cases}$$

$$= af(c)$$

One shows:

$$i(h) = i(c)$$

$$sf(h) = sf(c)$$

$$bf(h) = bf(c)$$

in the same way. Bit $af[3](c)$ a predicate, thus $af(h)$ is computed in the function decoder as a predicate and we get by lemma 49:

$$af[3](h) = af[3](c)$$

We summarize

Lemma 52.

$$cad(h) = cad(c)$$

$$af(h) = af(c)$$

$$i(h) = i(c)$$

$$sf(h) = sf(c)$$

$$bf(h) = bf(c)$$
5.3. A SEQUENTIAL PROCESSOR DESIGN

That finishes - fortunately - the bookkeeping of what the instruction decoder does.

5.3.8 Reading from General Purpose Registers

The general purpose register file \( h.gpr \) of the hardware is shown in figure 92 is a 3 port gpr-RAM with two read and one write port. The \( a \) and \( b \) addresses off the file are connected to \( rs(h) \) and \( rt(h) \). For the data outputs \( gprouta \) and \( gproutb \) we introduce the shorthands \( A \) and \( B \)

\[
\begin{align*}
A(h) & = gprouta(h) \\
& = h.gpr(rs(h)) \quad \text{(H)} \\
& = c.gpr(rs(h)) \quad \text{(sim.2)} \\
& = c.gpr(rs(c)) \quad \text{(lemma 50)} \\
B(h) & = c.gpr(rt(c)) \quad \text{(similarly)}
\end{align*}
\]

Thus we have

Lemma 53.

\[
\begin{align*}
A(h) & = c.gpr(rs(c)) \\
B(h) & = c.gpr(rt(c))
\end{align*}
\]

5.3.9 Next pc environment

Branch Condition Evaluation Unit  The BCE-unit is wired as shown in figure 93
By lemmas 53 and 52 as well as the correctness of the BCE implementation from section 4.5 we have

\[
br{\text{res}}(h) = b{\text{cres}}(A(h), B(h), b_f(h)) = b{\text{cres}}(c.gpr(rs(c), c.gpr(rt(c), b_f(c)) = \text{res}(c)
\]

Thus we have

**Lemma 54.**

\[
br{\text{res}}(h) = \text{res}(c)
\]

**Incremented PC** The computation of an incremented pc as needed for next pc environment as well as for the link instructions is shown in figure 94. Because the pc can be assumed to be aligned\(^4\) the use of a 30-incrementer suffices. Using the correctness of the incrementer from section 4.1

\(^4\)Otherwise a misalignment interrupts would be signalled.
5.3. A SEQUENTIAL PROCESSOR DESIGN

Figure 95: Next-pc environment

\[
\begin{align*}
pcinc(h) & = h.pc[32 : 2] +_{30} 1_{30}00 \\
& = (c.pc[32 : 2] +_{30} 1_{30})00 \quad (\text{sim.1}) \\
& = c.pc[31 : 2]00 +_{32} 1_{30}00 \quad (\text{lemma 12}) \\
& = c.pc +_{32} 4_{32} \quad (\text{alignment})
\end{align*}
\]

Thus we have

Lemma 55.

\[pcinc(h) = c.pc +_{32} 4_{32}\]

Next pc computation  The circuits computing the next pc input, which was left open in fig ?? when we treated the instruction fetch, are shown in figure 95

Predicates \(p \in \{jir, jafr, jum, b\}\) are computed in the instruction decoder, thus we have

\[p(c) = p(h)\]

by lemma 49

We compute \(jbtaken\) in the obvious way and conclude with lemma 54

\[
\begin{align*}
jbtaken(h) & = \text{jump}(h) \lor b(h) \land \text{bres}(h) \\
& = \text{jump}(c) \lor b(h) \land \text{bres}(c) \\
& = jbtaken(c)
\end{align*}
\]
We have by lemmas 53, 55 and 50

\[ A(h) = c.gpr(rs(c)) \]
\[ \text{nextpc}(h) = c.pc + 32 \ 4_{32} \]
\[ \text{imm}(h)[15]^{14} \text{imm}(h)00 = \text{imm}(c)[15]^{14} \text{imm}(c)00 \]
\[ = bdist(c) \]

For the computation of the 30-bit adder we argue as in lemma 55

\[ s(h)00 = h.pc[32 : 2] + 30 \text{imm}(h)[15]^{14} \text{imm}(h)00 \]
\[ s(h)00 = c.pc[32 : 2] + 30 \text{imm}(c)[15]^{14} \text{imm}(c)) \quad \text{(sim.1)} \]
\[ s(h)00 = c.pc[32 : 2]00 + 32 \text{imm}(c[15]^{14} \text{imm}(c))00 \quad \text{(lemma 12)} \]
\[ = c.pc + 32 \ bdist(c) \quad \text{(alignment)} \]

We conclude

\[
\text{btarget}(h) = \begin{cases} 
  c.pc + 32 \ bdist(c) \\
  c.gpr(rs(c)) \\
  (c.pc + 32 \ 4_{32}[31 : 29]\iindex(c) \ j(c) \lor j\text{al}(c) \\
\end{cases}
\]
\[ = \text{btarget}(c) \]

Exploiting

\[ \text{reset}(h) \neq 0 \]

and the semantics of register updates we conclude

\[ h'.pc = \text{nextpc}(h) \]
\[ = \begin{cases} 
  \text{btarget}(c) \\
  c.pc + 32 \ 4_{32} \\
\end{cases} \\
\]
\[ = c'.pc \]

Thus we have shown

**Lemma 56.**

\[ h'.pc = c'.pc \]

This is sim.1 for the next configuration.
5.3.10 ALU environment

We begin with the treatment of the execute stage. The ALU environment is shown in figure 96. For the ALU’s left operand we have

\[
lop(h) = A(h) = c.gpr(rs(c)) \quad \text{(lemma 53)}
\]

For the right operand follows with lemmas 53 and 51

\[
rop(h) = \begin{cases} 
B(h) & \text{rtype}(h) \\
xtimm(h) & \text{otherwise}
\end{cases}
\]

\[
= \begin{cases} 
\text{c.gpr(rt(c))} & \text{rtype}(c) \\
xtimm(c) & \text{otherwise}
\end{cases}
\]

\[
= rop(c)
\]

For the result \(ares\) of the ALU we get

\[
ares(h) = alures(lop(h), rop(h), itype(h), af(h)) \quad \text{(section 4.3)}
\]

\[
= alures(lop(c), rop(c), itype(c), af(c)) \quad \text{(lemmas ??, 52, 49)}
\]

\[
= ares(c)
\]

We summarize

**Lemma 57.**

\[
ares(h) = ares(c)
\]

Note that in contrast to previous lemmas the proof of this lemma is not just bookkeeping; it involves the not so trivial correctness of the ALU implementation from section 4.3.
5.3.11 Shift unit environment

The computation of the operands of the shift unit is shown in figure 97. The left operand of the shifter is tied to $B$. Thus

$$
slop(h) = B(h) = c.gpr(rt(c)) \quad \text{lemma 53}
$$

$$
slop(c)
$$

For the shift distance we have by lemmas 50 and 53

$$
sdist(h) = \begin{cases} 
  sa(h) & fun(h)[3] = 0 \\
  A(h)[4:0] & fun(h)[3] = 1 
\end{cases}
$$

$$
sdist(c) = \begin{cases} 
  sa(c) & fun(c)[3] = 0 \\
  c.gpr(rs(c))[4:0] & fun(c)[3] = 1 
\end{cases}
$$

Using the non trivial correctness of the shift unit implementation from section 4.4

$$
sres(c) = \text{sures}(slop(h), sdist(h), f(h)) \quad \text{(section 4.4)}
$$

$$
sres(loc(c), sdist(c), sf(c)) \quad \text{(lemma 52)}
$$

$$
sres(c)
$$

We summarize

Lemma 58.

$$
sres(h) = sres(c)
$$
5.3. A SEQUENTIAL PROCESSOR DESIGN

5.3.12 jump and link

The value \( \text{linkad} \) that is saved in jump and link instructions is here (but not in later designs) identical with the incremented pc \( \text{pcinc} \) from the next-pc environment. Although we could re-use the result from the next pc-environment, we compute \( \text{linkad}(h) \) instead by a second copy of the circuit in figure 94 as a placeholder for a different circuit in later designs. We have

\[
\text{linkad}(h) = \text{pcinc}(h) = h.pc + 32 4_{32} = \text{linkad}(c) \tag{5.2}
\]

5.3.13 Collecting results

Figure 98 shows a small multiplexer-tree collecting results \( \text{linkad}, \text{ares} \) and \( \text{sres} \) into an intermediate result \( C \). Using lemmas Using lemma 57, 58 and 49 as well as equation ?? we conclude

\[
C(h) = \begin{cases} 
\text{sres}(h) & \text{su}(h) \\
\text{linkad}(h) & \text{jal}(h) \lor \text{jalr}(h) \\
\text{ares}(h) & \text{otherwise}
\end{cases}
\]

Thus we have

**Lemma 59.**

\[
C(h) = C(c)
\]

5.3.14 Effective Address

The effective address computation is shown in figure 99. We have

\[
ea(h) = A(h) + 32 \text{imm}(h)[15:16] \text{imm}(h) \quad \text{(section 4.1)}
\]

\[
ea(h) = c.gpr(rs(c)) + 32 \text{siximm}(c) \quad \text{(lemmas ?? and 50)}
\]

Thus we have

**Lemma 60.**

\[
ea(h) = ea(c)
\]
5.3.15 Shift for Store environment

Figure 100 shows a shifter construction and the data memory, which is realized as a 64-multi-bank-RAM with bank write signals \( bw[7 : 0] \) and address \( ea[31 : 3] \). The shifter construction serves to align the the \( B \)-operand with the 64 bit wide memory. A second small shifter construction generating the byte write signals is shown in figure 100.

The initial mask signals are generated as

\[
smask(h)[3 : 0] = s(h) \land (I(h)[27] I(h)[26])
\]

One easily verifies

Indeed we have

\[
smask(h) = \begin{cases} 
0000 & s(c) = 0 \\
0001 & s(c) \land d(c) = 1 \\
0011 & s(c) \land d(c) = 2 \\
1111 & s(c) \land d(c) = 4 
\end{cases}
\]

resp.

\[
smask(h)[i] = 1 \leftrightarrow s(c) \land i < d(c)
\]

By alignment we have

\[
d(c) = 2 \rightarrow ea(c)[0] = 0 \land d(c) = 4 \rightarrow ea(c)[1 : 0] = 00
\]

For the shifted versions of the mask signals we conclude for \( s(c) = 0 \):

\[
bw(h) = 0^8
\]
Using $ea(c) = ea(h)$ from lemma 60 we conclude for $s(c) = 1$:

\[ e(h)[j] = 1 \iff j = ea(c)[0] + i \land i < d(c) \]
\[ f(h)[j] = 1 \iff j = (ea(c)[1 : 0]) + i \land i < d(c) \]
\[ bw(h)[j] = 1 \iff j = (ea(c)[2 : 0]) + i \land i < d(c) \]

Similarly we have for the large shifter and $i < d(c)$

\[
\begin{align*}
\text{byte}(i, B(h)) &= \text{byte}(i + ea(c)[0], D(h)) \\
&= \text{byte}(i + (ea(c)[1 : 0]), E(h)) \\
&= \text{byte}(i + (ea(c)[2 : 0]), dmin(h))
\end{align*}
\]

Using $B(h) = c.gpr(rt(c))$ from lemma 53, we summarize for the shifters supporting the store operations

**Lemma 61.** If $s(c) = 1$, i.e. if a store operation is performed in ISA configuration $c$, then

\[ bw(h)[j] = 1 \iff j = (ea(c).a) + i \land i < d(c) \]

\[ \forall i < d(c) : \text{byte}(i, c.gpr(rt(c))) = \text{byte}(i + (ea(c).a), dmin(h)) \]

This concludes the treatment of the execute stage.

### 5.3.16 Memory Stage

In the memory we only have the data memory $dm$, a 64-multi-port RAM addressed by $ea.l$ and controlled by the byte write signals $bw(h)[7 : 0]$ constructed above. We proceed to prove the induction step for the two memories.
Lemma 62.

\[ \forall a \in CR : h'.im = c.m_s(a000) \]
\[ \forall a \in DR : h'.dm(a) = c.m_s(a000) \]

By lemma 43 the second condition is equivalent to

\[ \forall x \in B^{32} : c'.m(x) = \text{byte}(\langle x[2 : 0] \rangle, h'.dm(x[31 : 3])) \]

and we will prove the lemma for \( dm \) in this form.

By induction hypotheses \( \text{sim.3} \) and \( \text{sim.4} \) we have

\[ \forall a \in CR : h.im(a) = c.m_s(a000) \]
\[ \forall a \in DR : d.im(a) = c.m_s(a000) \]

For \( s(c) = 0 \) no store is executed and in the ISA computation we have \( c'.m = c.m \). In the hardware computation we have \( b\text{mask}(h) = 0^4 \) and \( bw(h)[7 : 0] = 0^8 \); hence \( h'.dm = h.dm \). With the induction hypothesis we conclude trivially for all \( a \in CR \)

\[ h'.im(a) = h.im(a) \]
\[ = c.m_s(a000) \quad \text{(sim.3)} \]
\[ = c'.m_s(a000) \]
and for \( a \in DR \)

\[
\begin{align*}
h'.dm(a) &= h.dm(a) \\
&= c.m_8(a000) \quad \text{(sim.4)} \\
&= c'.m_8(a000)
\end{align*}
\]

For \( s(c) = 1 \) the ISA specifies

\[
c'.m(x) =
\begin{cases} 
\text{byte}(i, c.gpr(rt(c))) & x = ea(c) + 32 i_{32} \land i < d(c) \\
\text{c.m}(x) & \text{otherwise}
\end{cases}
\]

and the specification of multi-bank-RAM gives for all \( a \in \mathbb{B}^{29} \)

\[
h'.dm(a) = \begin{cases} 
\text{modify}(h.dm(a), dmin(h), bw(h)) & a = ea(c).l \\
h.dm(a) & \text{otherwise}
\end{cases}
\]

We know \( ea(c).l \in DR \). Thus for any \( a \in CR \) we have

\[
h'.dm(a) = \begin{cases} 
\text{h.dm}(a) \\
c.m_8(a000) & \text{(sim.3)} \\
= c'.m_8(a000)
\end{cases}
\]

With \( x \in \mathbb{B}^{33}, a = x.l \) and \( j = (x.o) \in \mathbb{B}_3 \) and the definition of function \( \text{modify} \) we rewrite equation 5.3 as

\[
\text{byte}(\langle x.o \rangle, h'.dm(x.l))
= \text{byte}(j, h'.dm(a))
= \text{byte}(j, dmin(h)) \\
= \text{byte}(j, h.m(a)) \quad \text{otherwise}
\]

\[
= \begin{cases} 
\text{byte}(j, dmin(h)) & bw(h)[j] \land a = ea(c).l \\
\text{byte}(j, h.m(a)) & \text{otherwise}
\end{cases}
\]

\[
= \begin{cases} 
\text{byte}(j, dmin(h)) & j = (ea(c).o) + i \land i < d(c) \\
\text{byte}(j, h.m(a)) & \text{otherwise} \quad \text{(lemma 61)}
\end{cases}
\]

\[
= \begin{cases} 
\text{byte}(i, dmin(h)) & j = (ea(c).o + 3 i_3) \land i < d(c) \\
\text{byte}(j, h.m(a)) & \text{otherwise} \quad \text{(lemma 42)}
\end{cases}
\]

\[
= \begin{cases} 
\text{byte}(i, c.gpr(rt(c))) & x.o = ea(h).o + 3 i_{32} \land i < d(c) \\
\text{byte}(\langle x.o \rangle, h.m(x.l)) & \text{otherwise} \quad \text{(lemma 61)}
\end{cases}
\]

\[
= \begin{cases} 
\text{byte}(i, c.gpr(rt(c))) & x = ea(c) + 32 i_{32} < d(c) \\
\text{byte}(\langle x.o \rangle, h.m(x.l)) & \text{otherwise} \quad \text{(lemma 42)}
\end{cases}
\]

\[
= c'.m(x)
\]
5.3.17 Shifter for Load

The only remaining stage is the write back stage. A shifter construction supporting load operations is shown in figure ?? . Assume \( l(c) \) holds, i.e. a load instruction is executed. Because \( c.m \sim_{DR} h.dm \) holds by induction hypothesis, we can use lemma 44 to locate for \( i < d(c) \) the bytes to be loaded in \( h.dm \) and subsequently - using memory semantics - in \( dmout(h) \). Then we simply track the effect of the two shifters taking into account, that the 24 bit left shift is a 8 bit right shift.

\[
byte(i, c.m_d(ea(c))) = \begin{cases} 
\text{byte}(\{ea(c).l\} + i, h.dm(ea(c).l) & \text{if } i < 8 \text{ and } h.dm(ea(c).l) \\
\text{byte}(\{ea(c).l\} + i, dmout(h)) & \text{if } i \geq 8 \\
\text{byte}(\{ea(c).[1 : 0]\} + i, G(h)) & \text{if } i \geq 8 \\
\text{byte}(ea(c).[0] + i, II(h)) & \text{if } i \geq 8 \\
\text{byte}(i, J(h)) & \text{if } i \geq 8 
\end{cases}
\]

We conclude

**Lemma 63.**

\( J(h)[8d - 1 : 0] = c.m_d(ea(c)) \)

Setting \( fill(h) = J(h)[7] \land lb(h) \lor J(h)[15] \land lh(h) \) we conclude

\[
fill(h) = fill(c)
\]

Similar to the mask \( smask \) for store operations we generate a load mask

\[
lmask(h) = I(h)^{16}[27]^3 I(h)[26]^{18}
\]

In case of load operations \( (l(c) \) holds) it satisfies

\[
lmask(h) = \begin{cases} 
0^{24}18 & d(c) = 1 \\
0^{16}16 & d(c) = 2 \\
1^{32} & d(c) = 4 \\
0^{32 - 8.d(c)}1^8 & d(c)
\end{cases}
\]

As shown in figure 103 we insert the fill bit at positions \( i \) where the corresponding mask bit \( lmask[i] \) is zero

\[
lres(h)[i] = \begin{cases} 
fill(h)[i] & lmask(h)[i] = 0 \\
J(i) & lmask(i) = 1
\end{cases}
\]

We conclude with lemma 63
Figure 102: Shifter for load operations in the sh4l-environment

Figure 103: Fill bit computation for loads

Lemma 64.

\[ lres(h) = fill(c)^{32-8d(c)c.m_d(c)} \]

5.3.18 Writing to the General Purpose Register File

Figure 104 shows a last multiplexer connecting the data input of the general purpose register file with intermediate result \( C \) and the result \( lres \) coming from the sh4l-environment. The write signal \( gpwrw \) of the general purpose register file and the predicates \( su, jal, jalr, l \) controlling the muxes are predicates \( p \) computed in the instruction decoder. By lemma 49 we have for them

\[ p(c) = p(h) \]

Using lemmas 59 and 64 we conclude
Figure 104: Computing the data input of the GPR

\[ gprin(h) = \begin{cases} 
  lres(h) & l(h) \\
  C(c) & \text{otherwise}
\end{cases} \]

\[ = gprin(c) \]

Using RAM semantics, induction hypothesis \textit{sim.2} and lemma 52 we complete the induction step for the general purpose register file

\[ h'.gpr(x) = \begin{cases} 
  gprin(h) & gprw(h) \wedge x = cad(h) \\
  h.gpr(x) & \text{otherwise}
\end{cases} \]

\[ = \begin{cases} 
  gprin(c) & gprw(c) \wedge x = cad(c) \\
  c.gpr(x) & \text{otherwise}
\end{cases} \]

\[ = c'.gpr(x) \]

This concludes the proof of lemma 46 as well the correctness proof of the entire (simple) processor.
Chapter 6

Pipelining

6.1 MIPS ISA and basic implementation revisited

6.1.1 Delayed PC

What we have presented so far - both in the definition of the ISA and in the implementation of the processor - was a sequential version of MIPS. For pipelined machines the ISA is changed in two ways.

- so far in an ISA computation \( c^i \) new program counters \( c^{i+1}.pc \) are computed by instruction \( I(c^i) \) and the next instruction

\[
I(c^{i+1}) = c^{i+1}.m_4(c^{i+1}.pc)
\]

was fetched with this pc. In the new ISA the instruction fetch after a new pc computation is delayed by 1 instruction. This is achieved by leaving the next pc computation unchanged but i) introducing a delayed pc \( c.dpc \) which simply stores the pc of the previous instruction and ii) fetching instructions with this delayed pc. At the start of computations the two program counters are initialized such that the first two instructions are fetched from addresses \( 0_{32} \) and \( 4_{32} \)

\[
\begin{align*}
  c^1.dpc &= 0_{32} \\
  c^1.pc &= 4_{32} \\
  c^{i+1}.dpc &= c^i.pc \\
  I(c^i) &= c^i.m_4(c^i.dpc)
\end{align*}
\]

The reason for this change of ISA are technical and stem from the fact, that in basic 5 stage pipelines instruction fetch and next pc computation are distributed over two pipeline stages.\(^1\). The introduction of the delayed pc permits to make the effect of this in the sequential

\(^1\)The reasons for this will be explained later
ISA. In a nutshell pc and dpc are a tiny bit of visible pipeline in an otherwise completely sequentially programming model.

The 4 bytes after a jump or branch instruction are called a delay slot, because the instruction in the delay slot is always executed before the branch or jump takes effect.

- the semantics of instructions jal and jalr have also to be changed. One saves the current pc incremented by 8, i.e. one saves as return address the instruction after the delay slot

\[
\text{linkad}(c) = c.pc +_{32} 8_{32}
\]

### 6.1.2 Implementing the delayed pc

The changes in the simple non pipelined implementation for the new ISA are completely obvious

- introduce the delayed pc as shown in figure 105
- compute the link address linkad now by means of a 29 bit incremeneter as shown in figure 106

For the increment of the pc by 8 one now uses a 29-incrementer as shown in figure 106. One proves in the style of lemma 55

\[
\text{linkad}(h) = h.pc +_{32} 8_{32}
\]

and concludes

**Lemma 65.**

\[
\text{linkad}(h) = \text{linkad}(c)
\]

The resulting new design \( \sigma \) is a sequential implementation of the MIPS ISA for pipelined machines. We denote hardware configurations of this machine by \( h_{\text{sigma}} \). For ISA computations \( (c^t) \) of the new pipelined instruction set one shows in the style of the previous chapter under the same software conditions the correctness of the modified implementation for the new (and real) instruction set.

**Lemma 66.**

\[
\forall t \geq 1 : \text{sim}(h_{\text{sigma}}^t, c^t)
\]
6.1.3 Pipeline stages and visible registers

When designing processor hardware one tries to solve a fairly well defined optimization problem, that is formulated and studied at considerable length in [MP00]. In this text we focus on correctness proofs and only remark that one tries i) to spend (on average) as few as possible hardware cycles per executed ISA instruction and ii) to keep the cycle time (as e.g. introduced in the detailed hardware model) as small as possible. In the first respect the present design is excellent. With a single processor one cycle per instruction is hard to beat. As far as cycle time is concerned, it is a perfect disaster: the circuits of every single stage contribute to the cycle time.

In a basic 5 stage pipeline one partitions the circuits of the sequential design into 5 circuit stages cir(i) with i ∈ [0 : 4] such that

- the circuit stages have roughly the same delay which then is roughly 1/5 of the original cycle time and
- connections between circuit stages are as simple as possible

We have already introduced the stages. That the cycle times in each stage are roughly equal cannot be shown here, because we have not introduced a detailed and realistic enough delay model. The interested reader is referred to [MP00].

Simplicity of inter stage connections is desirable, because in pipelined implementations most of theses connections are realized as register stages. And registers cost money without computing anything new. For a study how much relative cost is increased by such registers we refer the reader again to [MP00].

We conclude this section by a bookkeeping exercise about the interconnections between the pipeline stages. We stress that we do almost nothing at all yet. We simply add the delayed pc to figure 87 of chapter 5 and redraw the figure according to some very simple rules:
CHAPTER 6. PIPELINING

Figure 107: Arranging the sequential MIPS design into pipeline stages

1. Whenever a signal crosses downwards from one stage to the next: draw a dotted box around it and rename it (before or after it crosses the boundary)

2. forget about the circuits between stages and collapse them into circles labeled cir(i)

The result is shown in figure y. We observe two kinds of stages. i) circuit stages cir(i) or cir'(i) and pipeline stages reg(k) consisting either of registers or memories of the sequential design or of dotted boxes for renamed signals. Most of the figure should be self explaining, we add a few remarks.

- Circuit stage cir(1) and pipeline stage reg(1) are the IF stage. cir(1)
consists only of the instruction memory environment, which is presently read only and hence behaves like a circuit. Signal $I$ contains the instruction, that was fetched.

- Circuit stage $cir(2)$ and pipeline stage $reg(2)$ are the $ID$ stage. The circuit stage consists of the instruction decoder and the next-pc environment. Signals $A$ and $B$ have been renamed before they enter circuit stage $cir(2)$. Signal $Bin$ is only continued under the synonym $B$, but signal $Atm$ is both used in the next-pc environment and continued under the synonym $A$. Pipeline stage 2 contains the program counter $pc$ and $dpc$, the operands $A$ and $B$ fetched from the GPR and the signals $t2ex$ going from the instruction decoder to the $EX$ stage

$$t2ex = (xtimm, af, sf, i, sa),$$

For some signals $x$ there exist versions in various pipeline stages $k$. In such situations we denote the version in register stage $reg(k)$ by $x.k$. In this sense we find in all pipeline stages $k \geq 2$ versions $con.k$ of control signals that were precomputed in the instruction decoder. This group of signals comprises predicates $p.k$, instruction fields $F.k$ and the C-address $cad.k$

$$con.k = (\ldots, p.k, \ldots, F.k, \ldots, cad.k)$$

- Circuit stage $cir'(3)$ and pipeline stage $reg(3)$ are the execute stage. The circuit stage comprises the ALU-environment, the shift unit environment, an incrementer for the computation of $linkad$, multiplexers for the collection of $ares$, $sures$ and $linkad$ into intermediate result $C$, an adder for the computation of the effective address and the shift-for-store environment.

Pipeline stage 3 contains a version $C.3$ of intermediate result $C$, the effective address $ea.3$, the data input $dmin$ for the data memory and copy $con.3$ of the control signals.

- circuit stage $cir'(4)$ and pipeline stage $reg(4)$ are the $M$ stage. The circuit stage consists only of wires; so we have not drawn it. pipeline stage 4 contains a version $C.4$ of $C$, the data memory data output $dmout.4$ as well as versions $con.4$ and $ea.4$ of the control signals and the effective address (only the offset $ea.4.o$ is used in this stage to control the shift-for-load environment). Note that we also have included the data memory $dm$ itself in this pipeline stage.

- Circuit stage $cir(5)$ and pipeline stage $reg(5)$ are the $WB$ stage. The circuit stage contains the shift-for-load environment (controlled by
ea.4.o) and a multiplexer collecting C.4 and result lres of the shift-for-load-environment into the data input gprin of the general purpose register file. Pipeline stage 5 consists of the general purpose register file.

For the purpose of constructing a first pipelined implementation of a MIPS processor we can simplify this picture even further:

- We distinguish in pipeline stages $k$ only between visible registers $pc$, $dpc$ and memories $dm$, $gpr$ from the ISA on one side and other signals $x$, $k$ on the other side

- We include straight connections by wires into the circuits $cir(i)$.  

- for $k \in [1:5]$ circuit stage $cir(k)$ is input for pipeline stage $k + 1$ and for $k \in [1:4]$ pipeline stage $k$ is input to circuit stage $cir(k + 1)$. We only hint these connections with small arrows and concentrate on the other connections.

We obtain figure 108. In the next section we will transform this simple figure with very little effort into a pipelined implementation of a MIPS processor.

---

2This does not change circuits $cir(0)$ and $cir(4)$
6.2 Basic pipelined processor design

6.2.1 Transforming the sequential design into a pipelined design

We transform the sequential processor design $\sigma$ of the last section into a pipelined design $\pi$ whose hardware configurations we will denote by $h_{pi}$. We also introduce some shorthands for registers or memories $R$ and circuit signals $X$ in either design:

$$R_\sigma^t = h_\sigma^t \cdot R$$
$$R_\pi^t = h_\pi^t \cdot R$$
$$X_\sigma^t = X(h_\sigma^t)$$
$$X_{pi}^t = X(h_{pi}^t)$$

For signals or registers only occurring in the pipelined design $\pi$ we drop the subscript $\pi$. If an equations holds for all cycles (like equations describing hardware construction) we drop the index $t$.

The changes to design $\sigma$ are explained very quickly:

- turn all dotted boxes of all pipeline stages into pipeline registers with the same name. Because their names do not occur in the ISA, they are only visible in the hardware design but not to the ISA programmer. Therefore they are called non visible or implementation registers. We denote visibility of a register or memory $R$ by predicate $\text{vis}(R)$

$$\text{vis}(R) \equiv R \in \{ pc, dpc, dm, gpr \}$$

- For indices $k$ of pipeline stages, collect in $\text{reg}(k)$ all registers and memories of pipeline stage $k$. Use a common clock enable $ue_k$ for all registers of $\text{reg}(k)$.

- initially after reset all pipeline stages except the program counters and circuit stage 0 contain no meaningful data. In the next 5 cycles they are filled one after another. We introduce the hardware from figure 109 to keep track of this. There are 5 full bits $\text{full}[0 : 4]$.

Formally

$$\text{full}_0 \equiv 1$$
$$\forall k \geq 1$$
$$\text{full}^t_k = 0$$
$$\text{full}^{t+1}_k = \text{full}^t_{k-1}$$

We show

$$\text{full}[0 : 4]^t = \begin{cases} 1^{t_0} 0^{5-t} & t \leq 4 \\ 1^5 & t \geq 5 \end{cases} \quad (6.1)$$
Table 6.1: Full bits track the filling of the pipeline stages

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>≥ 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$full_1^t$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$full_2^t$</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$full_3^t$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$full_4^t$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$full_5^t$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

by the simple

Lemma 67.  

$$\forall k, t \geq 1 : full_k^t = \begin{cases} 
0 & t \leq k \\
1 & t > k 
\end{cases}$$

Proof: for $k = 0$ we have for all $t \geq 1$:  

$$t \geq 1 > 0 = k , \quad full_0^t = 1$$

Thus the lemma holds for $k = 0$. For $k \geq 1$ the lemma is shown by induction on $t$. For $t = 1$ we have  

$$t \leq k , \quad full_1^t = 0$$

Thus the lemma holds for after reset. Assume the lemma holds for $t$. 
6.2. BASIC PIPELINED PROCESSOR DESIGN

Figure 110: Updating register stage \(k\) under control of the full bits of stage \(k - 1\)

Then

\[
full_{k+1} = full_{k-1}
\]

\[
= \begin{cases} 
0 & t \leq k - 1 \\
1 & t > k - 1 
\end{cases}
\]

\[
= \begin{cases} 
0 & t + 1 \leq k \\
1 & t + 1 > k 
\end{cases}
\]

Full bits which are 0 are used to prevent the update of pipeline stages. This is also called stalling a pipeline stage; we call the hardware therefore a basic stall engine. Other stall engines are introduced later.

- for any pipeline stage \(k\) we update registers and memories in \(\text{reg}(k)\) only if their input contains meaningful data, which is the case when the previous stage is full. As illustrated in figure 110 we set for registers

\[
ue_k = full_{k-1}
\]

For memories \(dm\) and \(gpr\) we take the precomputed write signals \(dmw.3\) and \(gprw.4\) from the precomputed control and AND it with the corresponding update enable bit to get the new write signals

\[
dmw_{\pi} = dmw.3 \land ue_{4}
\]

\[
gprw_{\pi} = gprw.4 \land ue_{5}
\]

- the address of the instruction memory is now computed as shown in figure 111 as

\[
ima_{\pi} = \begin{cases} 
dpc_{\pi} & full_{1} = 0 \\
pc_{\pi} & full_{1} = 1 
\end{cases}
\]
This has the remarkable effect, that we fetch from the pc in all cycles except the very first one. Thus the important role of the delayed pc is not in the hardware but in the ISA, where it exposes the effect of the fact, that instruction fetch and next-pc computation are distributed over two pipeline stages. If we would join the two stages int one (by omitting the I-register) we would gain back the original instruction set, but we would ruin the efficiency of the design by roughly doubling the cycle time.

- the input of the 29-incrementer for the computation of signal linkad taken from the dpc instead of the pc as shown in figure x. We do not blame readers who are highly suspicious about the last two

This are all changes we make to the sequential design \( \sigma \).

### 6.2.2 Scheduling functions

In the sequential design, there was a trivial correspondence between the hardware cycle \( t \) and the instruction \( I(c^t) \) executed in that cycle. In the
6.2. BASIC PIPELINED PROCESSOR DESIGN

<table>
<thead>
<tr>
<th>t</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>I(1,t)</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>I(2,t)</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>I(3,t)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>I(4,t)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>I(5,t)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 6.2: Scheduling functions for the first 6 cycles

pipelined design π the situation is more complicated, because in 5 stages there are up to 5 different instructions which are in various stages of completion. For instructions \( I(c^i) \) of the sequential computation we use the shorthand

\[
I_i = I(c^i)
\]

We introduce scheduling functions

\[
I : [1 : 5] \times N \rightarrow N
\]

which for each pipeline stage \( k \), and hardware cycle \( t \) keep track of the index

\[
i = I(k, t)
\]

of the instruction \( I_i \) that is processed in stage \( k \) in cycle \( t \). Formally the functions are defined with the help of the update enable function \( u_{ek} \) in the following way

\[
\forall k : I(k, 1) = 1
\]

\[
I(1, t + 1) = I(1, t) + 1
\]

\[
\forall k \geq 2 : I(k, t + 1) = \begin{cases} I(k - 1, t + 1) & u_{ek}^t = 1 \\ I(k - 1, t) & \text{otherwise} \end{cases}
\]

i.e. after reset we are in each stage before execution of instruction \( I_1 \). In stage 1 we fetch every cycle a new instruction. A stage \( k \) which has a predecessor stage \( k - 1 \) is updated or not in cycle \( t \) as indicated by the \( u_{et} \) signal. If it is updated, the instruction from the predecessor stage enters stage \( k \). Otherwise the scheduling functions stays the same. Table 6.2.2 shows the development of the scheduling functions in our basic pipeline for the first 5 cycles

The definition of the scheduling functions can be viewed in the following way: imagine we extend each register stage \( reg(k) \) by a so called ghost register \( I(k, \_ \_ \_ ) \) that can store arbitrary natural numbers. In real machines that is of course impossible because registers are finite, but for the purpose of mathematical argument we can add the ghost registers to the construction and update them like all other registers of their stage by \( u_{ek} \). If we initialize
the ghost register of stage $I(1, \ )$ of stage 1 with 1 and increase it by 1 every cycle, then the pipeline of ghost registers simply clocks the index of the current instruction through the pipeline together with the real data.

Augmenting real configurations for the purpose of mathematical argument by ghost components is a useful proof technique. No harm is done to the real construction as long as no information flows from the ghost components to the real components.

With the help of lemma 67 we show

**Lemma 68.** For all $k \geq 1$ and for all $t$:

$$ I(k, t) = \begin{cases} 1 & t \leq k \\ t - k + 1 & t > k \end{cases} $$

Proof by induction on $t$. For $t = 1$ we have for all $k$

$$ t \leq k \quad , \quad I(k, t) = 1 $$

This shows the base case of the induction. Assume the lemma holds for $t$. In the induction step we consider two cases:

- $k = 1$: the claim of the lemma simplifies to

$$ I(1, t) = t $$

and we have by the definition of $I(1, \ )$:

$$ I(1, t + 1) = I(1, t) + 1 $$

$$ = t + 1 $$

- $k \geq 2$: we have by lemma 67

$$ u^t_k \equiv \text{full}^t_{k-1} \equiv (k - 1 < t) $$

Thus

$$ I(k, t + 1) = \begin{cases} I(k - 1, t) & u^t_k \\ I(k, t) & /u^t_k \end{cases} $$

$$ = \begin{cases} I(k - 1, t) & k - 1 < t \\ I(k, t) & k - 1 \geq t \end{cases} $$

$$ = \begin{cases} t - (k - 1) + 1 & k < t + 1 \\ I(k, t) & k > t \end{cases} $$

$$ = \begin{cases} t + 1 - k + 1 & k < t + 1 \\ 1 & k \geq t + 1 \end{cases} $$
6.2. BASIC PIPELINED PROCESSOR DESIGN

The following lemma relates the indices $I(k, t)$ and $I(k-1, t)$ in adjacent pipeline stages. They differ by 1 iff stage $k - 1$ is full.

**Lemma 69.**

$$I(k-1, t) = \begin{cases} 
I(k, t) & \text{full}_{k-1}^t = 0 \\
I(k, t) + 1 & \text{full}_{k-1}^t = 1
\end{cases}$$

Proof: by lemmas 68 and 67 we have

$$I(k-1, t) - I(k, t) = \begin{cases} 
1 & 1 \leq k - 1 \\
t - (k - 1) + 1 & t > k - 1
\end{cases}$$

$$= \begin{cases} 
1 & 1 \leq k \\
t - k + 1 & t > k
\end{cases}$$

$$= \begin{cases} 
1 - 1 & t \leq k - 1 \\
2 - 1 & t = k
\end{cases}$$

$$= \begin{cases} 
0 & t \leq k - 1 \\
1 & t > k - 1
\end{cases}$$

$$= \text{full}_{k-1}^t$$

### 6.2.3 Use of invisible registers

Not all registers or memories $R$ are used in all instructions $I(c^i)$. In the correctness theorem we need to show correct simulation of invisible registers only in situations where they are used. Therefore we define for each invisible register $X$ a predicate $\text{used}(X, i)$ which must at least be true for all configurations when $X$ is used. Some invisible registers will always be correctly simulated, though not all of them are always used. We define

$$\forall X \in \{I, i2ex, con.2, con.3, con.4\} : \text{used}(X, i) = 1$$

Invisible register $A$ is used when the $gpr$ is addressed with $rs$, and $B$ is used, when the $gpr$ is accessed with $rt$.

We first define auxiliary predicates $A - \text{used}(c)$ and $B - \text{used}(c)$ depending on ISA configurations $c$, that we will need later. Inspection of the tables summarizing the MIPS ISA gives

$$A - \text{used}(c) \equiv (\text{itype}(c) \land /\text{lui}(c))$$

$$\lor (\text{alu}(c) \lor \text{su}(c) \land \text{fun}(c)[2] \lor \text{jr}(c) \lor \text{jalr}(c))$$

$$B - \text{used}(B) \equiv \text{s}(c) \lor \text{beq}(c) \lor \text{bne}(c) \lor \text{su}(c) \land \text{alu}(c) \lor \text{movg2s}(c)$$
Now we simply define

\[ \text{used}(A, i) = A - \text{used}(c^i) \]
\[ \text{used}(B, i) = B - \text{used}(c^i) \]

Registers C.3 and C.4 are used when the \( gpr \) is written but no load is performed

\[ \forall X \in \{C.3, C.4\} : \text{used}(X, i) \equiv gprw(c^i) \land \neg l(c^i) \]

Registers ea.3 and ea.4 are used in load or store operations

\[ \forall X \in \{ea.3, ea.4\} : \text{used}(X, i) \equiv l(c^i) \lor s(c^i) \]

Registers dmin and bw are used in stores:

\[ \forall X \in \{dmin, bw\} : \text{used}(X, i) \equiv s(c^i) \]

Finally \( dmout \) is used in loads

\[ \text{used}(dmout, c) \equiv l(c^i) \]

6.2.4 Software condition \( SC - 1 \)

We keep the software conditions of the sequential construction: alignment and no self modification due to disjoint code and data regions.

On new condition comes from the connection from circuit stage 2 to register stage 5 by the \( rs \) and \( rt \) signals. The scheduling functions for stages 2 and 5 are

\[ I(2, t) = \begin{cases} 
1 & t \leq 2 \\
 t - 1 & t \geq 3 
\end{cases} \]
\[ I(5, t) = \begin{cases} 
1 & t \leq 5 \\
 t - 4 & t \geq 6 
\end{cases} \]

Thus the indices of the instructions in stages \( ID \) and \( WB \) differ by

\[ I(2, t) - I(5, t) = \begin{cases} 
1 - 1 & t \leq 2 \\
 t - 1 - 1 & 3 \leq t \leq 5 \\
 t - 1 - (t - 4) & t \geq 6 
\end{cases} \]

\[ = \begin{cases} 
0 & t \leq 2 \\
 t - 2 & 3 \leq t \leq 4 \\
3 & t \geq 5 
\end{cases} \]

Hence

\[ I(2, t) - I(5, t) \leq 3 \quad (6.2) \]
6.2. BASIC PIPELINED PROCESSOR DESIGN

Assume in cycle $t$ instruction $I(2,t) = i$ is in circuit stage 2, i.e. the ID stage. Then signals $rs$ and $rt$ of this instruction overtake up to 3 instructions in pipeline stages 2, 3 and 4. If any of these overtaken instructions write to some general purpose register $x$ and instruction $i$ tries to read it - as in our basic design directly from the general purpose register file, then the data read will be stale; more recent data from an overtaken instruction is on the way to the GPR but has not reached it yet. For the time being we will simply formulate a software condition $SC - 1$ saying, that this situation does not occur and the prove that the basic pipelined design $\pi$ only works for ISA computations ($c^i$) which obey this condition. In later sections we will improve the design and get rid of the condition.

Therefore we formalize for $x \in \mathbb{B}^d$ and ISA configurations $c$ two predicates

- $\text{writesgpr}(x, i)$: meaning ISA configuration $c^i$ writes $gpr(x)$.
  \[
  \text{writesgpr}(x, i) \equiv gprw(c^i) \land \text{cad}(c^i) = x
  \]

- $\text{readsgpr}(x, i)$: meaning ISA configuration $c^i$ reads $gpr(x)$. Reading $gpr(x)$ can occur via $rs$ or the $rt$ address, i.e. if $A$ or $B$ are used with address $x$
  \[
  \text{readsgpr}(x, i) \equiv \text{used}(A, i) \land rs(c^i) = x \lor \text{used}(B, i) \land rt(c^i) = x
  \]

Now we can define the new part of software condition $SC - 1$: for all $i$ and $x$, if $I_i$ writes $gpr(x)$, then instructions $i_{i-1}, i_{i+2}, i_{i+3}$ don't read $gpr(x)$:

\[
\text{writesgpr}(x, i) \rightarrow \forall j \in [i + 1 : i + 3] \neg \text{readsgpr}(x, j)
\]

6.2.5 Correctness statement

Now that we can express what instruction $I(k, t)$ is in stage $k$ in cycle $t$ and whether an invisible register is used in that instruction, we can formulate the invariant coupling states $h^\pi_t$ of the pipelined machine with the set of states $h^\sigma_t$ of the sequential machine, that are processed in cycle $t$ of the pipelined machine, i.e. the set

\[
\{h^\pi_{I(k,t)} : k \in [1 : 5]\}
\]

We intend to prove by induction on $t$:

**Lemma 70.** Assume software condition $SC - 1$. For $k \in [1 : 5]$ let $R \in \text{reg}(k)$ be a register or memory of pipeline stage $k$ and for $k \in [0 : 5]$ let $X \in \text{cir}(k)$ be a signal of circuit stage $k$. Then

\[
R^t_\pi = \begin{cases} 
R^I_{\sigma_{I(k,t)}} & \text{vis}(R) \\
R^I_{\sigma_{I(k,t)-1}} & \text{full}_k^\pi \land \neg \text{vis}(R) \land \text{used}(R, c^I_{k,t})
\end{cases}
\]
By lemma 66 we know already \( \text{sim}(h^i_x, c^i) \). Thus lemma 70 establishes a simulation between the pipelined computation \((h^i_x)\) and the ISA computation \((c^i)\) too. In particular we have for predicates \( p \) only depending on the current instruction \( I \):

\[
p(c^i) = p(h^i_x)
\]

Except for the subtraction of 1 from \( I(k, t) \) for non visible registers, the induction hypothesis is quite intuitive: pipelined data \( h^i_x.R \) or \( X(h^i_x) \) in stage \( k \) in cycle \( t \) are identical to the corresponding sequential data \( h^i_x.R \) resp. \( c^i.R \) or \( X(h^i_x) \) resp. \( X(c^i) \), where \( i = I(k, t) \) is the index of the sequential instruction, that is in cycle \( t \) in stage \( k \) of the pipelined machine.

The subtraction of 1 can be motivated by the fact, that in the pipelined machine instructions \( i \) pass the pipeline from stage 1 to 5 unloading their results into the visible registers of stage \( k \), when they are clocked into \( \text{reg}(k) \). Now assume pipeline stage \( \text{reg}(k) \) contains a visible register \( R \) and a non visible register \( Q \) and let \( I(k, t) = i \). Then by the intuitive portion of the induction hypothesis \( R^i_x = R^i_y \). Thus the previous instruction \( i - 1 \) is completed for the visible register \( R_x \) in stage \( k \) and the content of \( R_x \) is the content of register \( R_y \) after execution instruction \( i - 1 \) resp. before execution of instruction \( i \). The data in the invisible register however are still intermediate results of instruction \( i - 1 \) (!) that are used in later cycles \( t' \) to update other visible registers deeper down in the pipeline. Of course this is just motivation and is not to be confused with proof.

6.2.6 Correctness proof of the basic pipelined design

We denote by \( \text{inv}(k, t) \) the statement of the lemma for stage \( k \) and cycle \( t \)

**Initialization of register stages** For \( t = 1 \) we have

\[
\text{full}^1_x = 1 \leftrightarrow k = 0
\]

. Thus there is nothing to show for invisible registers. Initially we also have for

\[
\forall k : I(k, 1) = 1
\]

For visible registers one gets

\[
\text{pc}^1_x = 4 = \text{pc}^1_y = \text{pc}^1_x^{(2,1)}
\]

\[
\text{dpc}^1_x = 0 = \text{dpc}^1_y = \text{dpc}^1_x^{(2,1)}
\]

The initial content of general purpose registers and data memory of the sequential machine is defined by the content of the pipelined machine after reset:

\[
\text{dm}^1_x = \text{dm}^1_y = \text{dm}^1_x^{(4,1)}
\]
\[ gpr^1_{\sigma} = gpr^1_{\sigma} = gpr^{t(5,1)}_{\sigma} \]

Thus we have

\[ \forall \sigma : Inv(k, 1) \]

No updates. Assume the lemma holds for \( t \). We show for each stage \( k \) separately that the lemma holds for stage \( k \) and \( t + 1 \), but we always proceed in the same way. There are two cases: The easy case is

\[ u_k^t = 0 \]

i.e. register stage \( k \) is not updated in cycle \( r \). By the definition of full bits we know

\[ full_{k+1}^t = full_{k-1}^t = u_k^t = 0 \]

Thus for invisible registers \( R \in reg(k) \) there is nothing to show either. For the scheduling functions \( u_k^t = 0 \) implies

\[ I(k, t + 1) = I(k, t) \]

For visible registers or memories \( R \) of \( reg(k) \) we have by induction hypothesis \( inv(k, t) \):

\[ R_{\sigma}^{t+1} = R_{\sigma}^t = R_{\sigma}^{I(k,t)} = R_{\sigma}^{I(k,t+1)} \]

This shows \( inv(k, t + 1) \) for stages \( k \) that are not updated in cycle \( t \)

Scheduling functions for updated stages

Lemma 71.

\[ u_k^t \rightarrow I(k, t + 1) = I(k, t) + 1 \]

Proof: \( u_k^t = full_{k-1}^t = 1 \). For \( k = 1 \) we have by definition of the scheduling functions

\[ I(1, t + 1) = I(1, t) + 1 \]

For \( k \geq 2 \) we have by lemma 69 and the definition of scheduling functions

\[ I(k, t + 1) = I(k - 1, t) = I(k, t) + full_{k-1}^t = I(k, t) + 1 \]
Proof obligations for the induction step. The case \( \text{ue}_k^i = \text{full}_{k-1} = 1 \) has is handled for each stage separately. There are however for stages \( k \in [1:5] \) and cycles \( t \) general proof obligations \( P(k, t) \) we have to show for visible registers, invisible registers and memories in each pipeline stage \( \text{reg}(k) \), which will allow us to prove the induction step.

Proof obligation \( P(k, t) \): let \( R \in \text{reg}(k) \) and let cycles \( t \) and instruction indices \( i \) corresponding via

\[
I(k, t) = i
\]

Then

- visible registers of pipeline stage \( \text{reg}(k) \) in machines \( \sigma \) and \( \pi \) have identical inputs

\[
R \in \{ \text{pc, dpc} \} \rightarrow \text{Rin}^t_{\sigma} = \text{Rin}^i_{\sigma}
\]

- invisible registers of pipeline stage \( \text{reg}(k) \) in machines \( \sigma \) and \( \pi \) that are used have identical inputs

\[
\neg \text{vis}(R) \land \text{used}(R, i) \rightarrow \text{Rin}^t_{\sigma} = \text{Rin}^i_{\sigma}
\]

- Memories \( R \) have the same write signal \( Rw \). For writes they have the same write addresses \( ea \) or \( cad \) and the same data input \( \text{Rin} \)

\[
\begin{align*}
dmwa^t_{\sigma} &= dmwa^i_{\sigma} \\
gprw^i_{\sigma} &= gprw^i_{\sigma} \\
s(c^t) &\rightarrow \\
ea^t_{\sigma} &= ea^i_{\sigma} \land \text{dmin}^t_{\sigma} = \text{dmin}^i_{\sigma} \\
gprw(c^t) &\rightarrow \\
cad^t_{\sigma} &= cad^i_{\sigma} \land gprin^t_{\sigma} = gprin^i_{\sigma}
\end{align*}
\]

Very simple arguments show, that \( P(k, t) \) implies \( \text{inv}(k, t+1) \), i.e. proving \( P(k, t) \) suffices to complete the the induction step for stage \( k \).

Lemma 72.

\[
\text{ue}_k^i \land P(k, t) \rightarrow \text{inv}(k, t+1)
\]

The proof hinges on lemma 71 and splits cases in the obvious way

- \( R \) is an invisible register. Because the register has in both machines the same input it gets updated in both machines in the same way

\[
\begin{align*}
R^{t+1}_{n} &= \text{Rin}^t_{n} \\
&= \text{Rin}^i_{\sigma} \\
&= \text{Rin}^{i+1}_{t} \\
&= \text{Rin}^{i(k, t+1)}
\end{align*}
\]
6.2. BASIC PIPELINED PROCESSOR DESIGN

- \( R \) is invisible and used in \( c^t \). Then \( R_\pi \) is updated, but in the sequential machine \( \text{Rin} \) and \( R \) are just synonyms

\[
R_{\pi}^{t+1} = \text{Rin}_\pi^t
= R_{\sigma}^t
= R_{\sigma}^{(k,t+1) - 1}
\]

- \( R \in \{ gpr, dmin \} \) is a memory. We use a common notation \( Rwa \) for the write address

\[
Rwa = \begin{cases} 
ea & R = dm 
\text{cad} & R = gpr
\end{cases}
\]

Then

\[
R_{\pi}^{t+1}(x) = \begin{cases} 
\text{Rin}_\pi^t & \text{Rin}_\pi^t \land x = Rwa_{\pi}^t \\
R_\pi(x)^t & \text{otherwise}
\end{cases}
\]

\[
= \begin{cases} 
\text{Rin}_\sigma^t & \text{Rin}_\sigma^t \land x = Rwa_{\sigma}^t \\
R_\sigma(x)^t & \text{otherwise}
\end{cases}
\]

\[
= R_{\sigma}^{t+1}
= R_{\sigma}^{(k+1,t)}
\]

It remains to prove hypothesis \( P(k,t) \) of lemma 72 for each stage \( k \) separately. These proofs also have a common pattern. For each circuit stage \( cir(k) \) we identify a set of inputs \( in \) of the stage which are identical in cycle \( t \) of \( \pi \) and instruction \( i \) of \( \sigma \).

\[ in_{\pi}^t = in_{\sigma}^t \]

We then show that these inputs determine the relevant outputs \( \text{Rin}, dmin \) etc. of the circuit stage. Because the circuit stages are identical in both machines, this suffices to show that the outputs have identical values. Unfortunately the proofs require simple but tedious bookkeeping about the invisible registers used. The only real action is in the proofs for signals \( \text{Ain}, Bin, ima \) and \( inc - in \).

\( k = 1; \text{stage IF} \): We only have to consider the address input \( ima \) of the instruction memory. We consider the multiplexer in figure 111, which selects between visible registers \( pc, dpc \in \text{reg}(2) \), and distinguish two cases:
CHAPTER 6. PIPELINING

• $t = 1$. Then $full_i^1 = 0$ and $I(1, 1) = I(2, 1) = 1$. We conclude with induction hypothesis $inv(2, 1)$

$$ima^1_{\pi} = dpc^1_{\pi} = dpc^1_{\sigma} = ima^1_{\sigma} = ima^1(1, 1)_{\sigma}$$

• $t \geq 2$. Then $full_i^1 = 1$. By lemma 69 we have

$$i = I(1, t) = I(2, t) + full_i^1 = I(2, t) + 1$$

Using induction hypothesis $inv(2, t)$ and the definition of delayed pc we conclude

$$ima^t_{\pi} = pc^t_{\pi} = pc^t_{\sigma} = dpc^t_{\sigma} = dpc^t(2, t) = 1$$

Thus the instruction memory environment has in both machines the same input; therefore it produces the same output

$$Iin^i_{\pi} = Iin^i_{\sigma}$$

i.e. we have shown $P(1, t)$ and hence by lemma 72 $inv(1, t + 1)$

$k = 2; \text{stage ID}$: Let

$$i = I(2, t) = I(1, t) - 1$$

There are three kinds of inputs signals for the circuits $cir(2)$ of this stage.

• invisible register $I \in reg(1)$. It is always used. By induction hypothesis $inv(1, t)$ we get

$$I^i_{\pi} = I^i_{\sigma}$$

This already determines the inputs of invisible registers $con.2$ and $i2ex$

$$R \in \{con.2, i2ex\} \rightarrow R_{in}^i = R_{in}^i$$

It also determines the signals $i2nextpc$ from the instruction decoder to the next-pc environment so that we have for these signals

$$i2nextpc^i_{\pi} = i2nextpc^i_{\sigma}$$
6.2. BASIC PIPELINED PROCESSOR DESIGN

- Visible registers \( pc, dpc \in reg(2) \) which are inputs to the next-pc environment. From induction hypothesis \( inv(2, t) \) we get immediately

\[
R \in \{pc, dpc\} \rightarrow R^t_n = R^t_o
\]

- For inputs \( Ain \) and \( Bin \) of circuit stage \( cir(2) \) we have to make use of software condition \( SC - 1 \). Assume \( used(A, i) \), i.e. \( A \) is used in instruction \( i \) by access to \( gpr \in reg(5) \) via \( rs \). By equation 6.2 we have

\[
I(5, t) \leq I(2, t) + 3 = i + 3
\]

Let

\[
x = rs^t_o = rs^t_n
\]

i.e. instruction \( I(2, t) \) reads \( gpr(x) \):

\[
readsgpr(x, I(2, t))
\]

if any of instructions \( I(3, t), I(4, t), I(5, t) \) would write \( gpr(x) \) this would violate software condition \( SC - 1 \). Thus

\[
\forall k \in [3 : 5] : \neg write gpr(x, k)
\]

Hence

\[
gpr_o(x)^l(2, t) = gpr_o(x)^l(5, t)
\]

Using induction hypothesis \( inv(5, t) \) we conclude

\[
Ain^l_n = gpr_o(x)^l_{\pi i}
\]

\[
= gpr_o(x)^l(5, t)
\]

\[
= gpr_o(x)^l(2, t)
\]

\[
= Ain^l_o
\]

Arguing about signal \( Bin = B' \) in the same way we conclude

\[
used(A, i) \rightarrow Ain^l_n = Ain^l_i
\]

\[
used(B, i) \rightarrow Ain^l_n = Ain^l_o
\]

It remains to argue about the inputs of visible registers \( pc \) and \( dpc \), i.e. about signals \( nextpc \) and register \( pc \) which is the input of \( dpc \).

For the input \( pc \) of \( dpc \) we have by induction hypothesis \( inv(2, t) \) and because \( t \geq 1 \):

\[
dpcin^l_n = pc^l_o = pc^l_o = dpcin^l_o
\]

For the computation of the \( nextpc \) signal are four cases:
• be($c_i$) ∨ bne($c_i$). This is the easiest case, because it implies used($A, i$) ∧ used($B, i$). and we have
\[ \text{in}_n^t = \text{in}_\sigma^t \]
for all inputs $in \in \{A, B, t2nxtpc\}$ of the next-pc environment. Because the environment is identical in both machines we conclude
\[ \text{pcin}_n^t = \text{nxtpc}_n^t = \text{nxtpc}_\sigma^t = \text{pcin}_\sigma^t \]
and are done.

• b($c_i$) ∧ (be($c_i$) ∨ bne($c_i$)). Then we have used($A, i$) and for signal $d$ in the branch evaluation unit we have
\[ d_n^t = 0^{32} = d_\sigma^t \]
And hence
\[
\begin{align*}
\text{jtaken}_n^t &= \text{jtaken}_\sigma^t \\
\text{btarget}_n^t &= \text{btarget}_\sigma^t \\
\text{nxtpc}_n^t &= \text{nxtpc}_\sigma^t
\end{align*}
\]

• jr($c_i$) ∨ jalr($c_i$). Then used($A, i$) and
\[ \text{nxtpc}_n^t = \text{Ain}_n^t = \text{Ain}_\sigma^t = \text{nxtpc}_\sigma^t \]

• in all other cases we have
\[ \text{nxtpc}_n^t = \text{pc}_n^t + 32 \ 432 = \text{pc}_\sigma^t + 32 \ 432 = \text{nxtpc}_\sigma^t \]

This concludes the proof of $P(2, t)$

$k = 3$; stage EX: Let
\[ i = I(3, t) = I(2, t) - 1 \]

We have to consider three kinds of input signals for the circuits cir(3) of this stage:

• invisible registers t2ex and con.2. They are always used. By induction hypothesis inv(2, t) we get
\[ X \in \{\text{t2ex, con.2}\} \to X_n^t = X_\sigma^t = X_{I(2, t) - 1}^t = X_\sigma^t \]

Because con.2 = con.3in this shows $P(3, t)$ for the pipelined control register con.3.
• denote the input of the 29-incrementer computing signal linkad as \( inc - in^i \). By construction of the sequential and pipelined machines, induction hypothesis \( inv(2, t) \) and the delayed pc semantics we have

\[
\begin{align*}
inc - in^i & = dpc^i_{\pi} \quad \text{(constr. of } \pi) \\
& = dpc^i_{\sigma(2, t)} \quad \text{(} inv(2, t) \text{)} \\
& = pc^i_{\sigma(2, t)-1} \quad \text{(} dpc \text{ semantics)} \\
& = pc^i_{\sigma} \\
& = inc - in^i_\sigma
\end{align*}
\]

We conclude that we always have

\[\text{linkad}^i_{\pi} = \text{linkad}^i_{\sigma}\]

• invisible registers \( A \) and \( B \) for which we have a nontrivial induction hypothesis \( inv(2, t) \) only when they are used.

\[X \in \{A, B\} \land \text{used}(i, X) \rightarrow X^i_{\pi} = X^i_{\sigma}\]

We proceed to show \( P(3, t) \) for registers \( mdin, bw \), register \( ea \) and register \( C.3 \) separately.

• \( mdin \) and \( bw \): we have

\[
\begin{align*}
\text{used}(mdin, i) & \equiv \text{used}(bw, i) \\
& \equiv s(c^i) \\
s(c^i) & \rightarrow itype(c^i) \land \text{tui}(c^i)
\end{align*}
\]

and hence

\[\text{used}(mdin, i) \lor \text{used}(bw, i) \rightarrow \text{used}(A, i) \land \text{used}(B, i)\]

By induction hypothesis \( inv(2, t) \) we conclude

\[X^i_{\pi} = X^i_{\sigma}\]

for all inputs \( X \) of \( cir(3) \) and conclude trivially

\[\text{mdinin}^i_{\pi} = \text{mdinin}^i_{\sigma}\]

\[\text{bwin}^i_{\pi} = \text{bwin}^i_{\sigma}\]

• \( ea \): we have

\[
\begin{align*}
\text{used}(ea, i) & = l(c^i) \lor s(c^i) \\
l(c^i) \lor s(c^i) & \rightarrow \text{used}(A, i)
\end{align*}
\]

Because \( ea \) depends only on \( A \) and \( i2ex \) we conclude

\[\text{eain}^i_{\pi} = \text{eain}^i_{\sigma}\]
• C.3. This needs a larger case split. We have
\[ used(C.3, i) = alu(c^i) \lor su(c^i) \lor jal(c^i) \lor jalr(c^i) \]
This results in 4 subcases
- \( alu(c^i) \lor su(c^i) \land \text{fun}(c^i)[3] \). Then
\[ used(A, i) \land used(B, i) \]
By induction hypothesis \( \text{inv}(2, t) \) we trivially conclude as above
\[ C.3in_t^i = C.3in_s^i \]
- \( alui(c^i) \). Then \( used(A, i) \) and
\[ rop_t^i = xtimm_t^i = xtimm_s^i = rop_s^i \]
Hence \( alures \) is independent of \( B \) and we conclude
\[ C.3in_t^i = alures_t^i = alures_s^i = C.3in_s^i \]
- \( su(c^i) \land /\text{fun}(c^i)[3] \). Then \( used(A, i) \) and
\[ sdist_t^i = sa_t^i = sa_s^i = sdist_s^i \]
Hence \( sures \) is independent of \( B \) and we conclude
\[ C.3in_t^i = sures_t^i = sures_s^i = C.3in_s^i \]
- \( jal(c^i) \lor jalr(c^i) \): then
\[ C.3in_t^i = linkad_t^i = linkad_s^i = C.3in_s^i \]
This concludes the proof of \( P(3, t) \)

\[ k = 4; \text{ stage } M: \quad \text{Let} \]
\[ i = I(4, t) = I(3, t) - 1 \]
We have to make three arguments.
• : \( X \in \{ dmin, \text{con}.3, \text{ea}.3, C.3 \} \). By induction hypothesis \( \text{inv}(3, t) \) we have
\[ used(X, i) \rightarrow X^i_t = X^i_s^{(i+1)-1} = X^i_s \]
This shows \( P(3, t) \) for the data inputs of registers \( \text{con}.4, \text{ea}.4 \) and \( C.4 \).
6.2. BASIC PIPELINED PROCESSOR DESIGN

- \textit{dmout}. We have

\[
\textit{used}(\textit{dmout}.4, i) \rightarrow \textit{load}(c^i) \land \textit{used}(\textit{ea}, i) \land \textit{used}(\textit{dmin}, i)
\]

Using induction hypothesis \textit{inv}(3, t) for \textit{ea} and \textit{dmin} as well as induction hypothesis \textit{inv}(4, t) for \textit{dm} we get

\[
\begin{align*}
dmout^i_{\sigma} &= dm^i_{\sigma}(ea.l^i_{\sigma}) \\
&= dm^i_{\sigma}(ea.l^{i(1+1)-1}_{\sigma}) \\
&= dmout^i_{\sigma}
\end{align*}
\]

This shows \(P(3, t)\) for the input \textit{dmout} of register \textit{dmout}.4.

- \textit{dm}. The data memory write signal \textit{dmw}.3 is a component of \textit{con}.4 which is used for all instructions. Thus

\[
\begin{align*}
dmw.A^t &= dmw^i_{\sigma} \\
dmw^i_{\sigma} &= dmw.A^t \land \textit{uc}^i_{\sigma} \\
&= dmw.A^t \land \textit{full}^i_{\sigma} \\
&= dmw^i_{\sigma}
\end{align*}
\]

because \textit{dmw}.4 is a component of \textit{con}.4 and \textit{used}(\textit{con}.4, i) holds for all \(i\). We have

\[
s(c^i) \rightarrow \textit{used}(\textit{ea}.3, i) \land \textit{used}(\textit{dmin}.3, i)
\]

As shown above this implies

\[
\textit{dmin}.3^i_{\sigma} = \textit{dmin}^i_{\sigma} \quad \text{and} \quad \textit{ea}.3^i_{\sigma} = \textit{ea}^i_{\sigma}
\]

\(k = 5; \text{ stage } WB:\) Let

\[
i = I(5, t) = I(4, t) - 1
\]

We only have to consider the input registers of the stage and to show \(P(4, t)\) for the general purpose register file

- all input registers are invisible thus let \(X \in \{\textit{C}.4, \textit{dmout}.4, \textit{ea}.4, \textit{con}.4\}\).

By induction hypothesis \textit{inv}(4, t) we have

\[
\textit{used}(X, i)X_{\sigma}^i = X_{\sigma}^i
\]
• Signal \( gprw.4 \) is a component of \( con.4 \). Thus we have

\[
gprw.4^t = gpr_{\sigma}^i \\
gpr_{\pi}^i = gprw.A^t \land \text{we}_b^i \\
= gprw.A^t \land \text{full}_b^t \\
= gprw_{\pi}^i
\]

Signal \( cad.4 \) is component of \( con.4 \). Thus

\[
cad.A^t_{\pi} = cad_{\pi}^i
\]

Assume \( gprw(c^i) \), i.e. the general purpose register file is written. We have to consider two subcases

- a load is performed. Then \( dmout \) and \( ea.4 \) are both used, load result \( tres \) is identical for both computations and the data input \( gprin \) for the general purpose register file comes for both computations from \( tres \):

\[
s(c^i) \rightarrow \text{used}(dmout.A, i) \land \text{used}(ea.A, i) \\
dmin_{\pi}^t = tres^t_{\pi} \\
= tres_{\pi}^i \\
= dmin_{\pi}^t
\]

- no load is performed. Then \( C.A \) is used and is the data input \( gprin \):

\[
/s(c^i) \rightarrow \text{used}(C.A, i) \\
dmin_{\pi}^t = C.A^t \\
= C_{\pi}^i \\
= dmin_{\pi}^t
\]

This completes the proof of \( P(5, t) \) and the induction step.

### 6.3 Forwarding

Software condition \( SC - 1 \) forbids to read a general purpose register \( gpr(x) \) that has been written in instruction \( i \) in the following three instructions \( i + 1, i + 2 \) and \( i + 3 \). We needed this condition because with the basic pipelined machine constructed so far we had to wait until the written data had reached the general purpose register file, simply because that’s where we accessed them. This situation is greatly improved by the forwarding circuits studied in this section.
6.3. FORWARDING

6.3.1 Hits

The improvement is based on two very simple observations. First, it is easy to recognize a cycle \( t \) where we want in to fetch in circuit stage \( \text{cir}(2) \) a register content \( gpr(x) \) into register \( A \) or \( B \) that is written by an instruction \( I(k, t) \) in the deeper register stages \( \text{reg}(k) \) with \( k \in [2:4] \). The stage must be full:

\[
\text{full}_k^t
\]

Otherwise it contains no meaningful data. The \( C-\text{adress} \) must coincide with the \( rs \) address or the \( rt \) address (note that these addresses are signals of circuit stage \( \text{cir}(2) \)):

\[
\text{cad}.k^t = rs^t \quad \text{or} \quad \text{cad}.k^t = rt^t
\]

Finally the instruction in stage \( k \) must write to the general purpose register file

\[
gprw.k^t
\]

We introduce for registers \( A \) and \( B \) separate predicates characterizing this situation

\[
\text{hit}_A[k] \equiv \text{full}_k \land \text{cad}.k = rs \land gprw.k
\]

\[
\text{hit}_B[k] \equiv \text{full}_k \land \text{cad}.k = rt \land gprw.k
\]

Second, in case we have a hit in stage 2 or 3 and the instruction is not a load instruction, then the data we want to fetch into \( A \) or \( B \) can be found as the input of the \( C \) register of the following circuit stage, i.e. as \( C.3in \) or \( C.4in \). In case of a hit in stage 4 we can find the required data at the data input \( grpin \) of the general purpose register even for loads.

6.3.2 Forwarding Circuits

All we have to do now is to construct circuits recognizing hits and forwarding the required data - where possible - to circuit stage \( \text{cir}(2) \). In case of simultaneous hits in several stages we are interested in the data of the most recent instruction producing a hit. This is the 'top' instruction in the pipe (i.e. with the smallest \( k \)) producing a hit

\[
\text{top}_A[k] = \text{hit}_A[k] \land \bigwedge_{j > k} /\text{hit}_A[j]
\]

\[
\text{top}_B[k] = \text{hit}_B[k] \land \bigwedge_{j > k} /\text{hit}_B[j]
\]

Obviously top hits are unique, i.e. for \( X \in \{A, B\} \) we have

\[
\text{top}_X[i] \land \text{top}_X[j] \rightarrow i = j
\]
Figure 113 shows the forwarding circuit $\text{For}_X$ for $X \in \{A, B\}$. If we find nothing to forward we access the general purpose register file as in the basic design. We have

$$X_{\text{in}} = \begin{cases} 
C.3\text{in} & \text{top}_X[2] \\
C.4\text{in} & \text{top}_X[3] \\
gprin & \text{top}_X[4] \\
gprout X & \text{otherwise}
\end{cases}$$

### 6.3.3 Software condition $SC - 2$

Forwarding will only fail if in case instruction $i$ is a load with destination $gpr(x)$ and this general purpose is read by one of the next two instructions $i + 1$ or $i + 2$.

$$l(c^i) \land cad(c^i) = x \land j \in [i + 1 : i + 2] \rightarrow \text{/readgpr}(x, j)$$

The correctness statement formulated in lemma 70 stays the same as before. Only software condition $SC - 1$ is replaced by the weaker conditions $SC - 2$.

### 6.3.4 Scheduling functions revisited

For the correctness proof we need a very technical lemma which states in a nutshell, that in the pipeline instructions are not lost.

**Lemma 73.** Let $k \geq 2$ be a pipeline stage, let $1 \leq s$ and

$$i = I(2, t) = I(k, t) + s$$

and let $1 \leq j < s$. Then

$$I(2 + j, t) = i - j \land \text{full}_{2+j}$$

i.e. any instruction $i - j$ between $i$ and $i - s$ is found in the full pipeline stage $2 + j$ between stages 2 and $k$.
6.3. **FORWARDING**

Proof: We rewrite the claim of lemma 69 as

\[ I(x, t) = \begin{cases} 
1 & t \leq x \\
\l t - k + 1 & t > x 
\end{cases} \]

and get

\[ I(2, t) = \begin{cases} 
1 & t < 2 \\
\l t - 1 & t \geq 2 
\end{cases} \]

\[ = I(k, t) + s \]

\[ \geq 2 \]

Hence \( t \geq 2 \) and

\[ I(2, t) = t - 1 \geq 1 + s \]

Thus

\[ t \geq s + 2 > 2 + j \]

Applying again lemma 69 we get

\[ I(2 + j, t) = \begin{cases} 
1 & t < 2 + j \\
\l t - (2 + j) + 1 & t \geq 2 + j 
\end{cases} \]

\[ = t - 1 - j \]

\[ = i - j \]

From lemma 67 we get

\[ full_{i+2}^j = t > j + 2 = 1 \]

6.3.5 **Correctness Proof**

The only case in the proof affected by the addition of the two forwarding circuits \( For_A \) and \( For_B \) is the proof of proof obligation \( P(2, t) \) in the induction step for signals \( Ain \) and \( Bin \). Also the order in which proof obligations \( P(k, t) \) are shown becomes important: one proves \( P(2, t) \) after \( P(3, t) \), \( P(4, t) \) and \( P(5, t) \).

We present the modified proof for \( Ain \). The proof for \( Bin \) is completely analogous. Assume

\[ w_e^i = full_1^i = 1 \]

and let

\[ i = I(2, t) \]
and consider some full stage \( k \in [2 : 4] \)

\[
k \in [2 : 4] \land \text{full}_k^t
\]

Then by lemma 67 stage \( k \) and all preceding stages must be full in cycle \( t \)

\[
\forall j \leq k : \text{full}_j^t
\]

and we can use induction hypothesis \( \text{inv}(j, t) \) for the invisible registers stages. Set

\[
k = 2 + \alpha \quad \text{with} \quad \alpha \in [0 : 2]
\]

For the scheduling function for stages \( k \) and \( k + 1 \) we get by lemma 69

\[
\begin{align*}
I(2 + \alpha, t) &= I(k, t) \\
&= \begin{cases} 
I(k, t) & k = 2 \\
I(2, t) - \sum_{j=2}^{k-1} \text{full}_j^t & k > 2
\end{cases} \\
&= i - (k - 2) \\
&= i - \alpha
\end{align*}
\]

\[
I(3 + \alpha, t) = I(k + 1, t)
\]

\[
= I(k, t) - \text{full}_k^t
\]

\[
= i - \alpha - 1
\]

**Lemma 74.** Let

\[
x = rs^i_\sigma \land k = 2 + \alpha \land \text{full}_k^t
\]

Then

\[
\text{hit}_{\sigma}^t[k] \equiv \text{writesgpr}(x, i - \alpha - 1)
\]

**Proof:** for the hit signal under consideration we can conclude with \( \text{inv}(k, t) \) and \( \text{inv}(2, t) \) for the invisible registers \( \text{cad} \) \( k \) and \( \text{gprw} \) \( k \):

\[
\begin{align*}
\text{hit}_{\sigma}^t[k] &\equiv \text{full}_k^t \land \text{cad}^t \land \text{gprw}^t \\
&= rs^t \land \text{gprw}^t \\
&= \text{cad}_{\sigma}^{(k, t)-1} \land rs^t \land \text{gprw}_{\sigma}^{(k, t)-1} \\
&= \text{cad}_{\sigma}^{i-\alpha-1} = x \land \text{gprw}_{\sigma}^{i-\alpha-1} \\
&= \text{writesgpr}(x, i - \alpha - 1)
\end{align*}
\]

Now assume

\[
\text{hit}_{\sigma}^t[k] \land k = 2 + \alpha
\]

Then by lemma 74 we have \( \text{writesgpr}(x, i - \alpha - 1) \) and for \( \alpha \in [0 : 1] \) we can also conclude from software condition \( SC = 2 \) that instruction \( i - \alpha \) is not a load instruction.

\[
\exists \alpha \in [0 : 1] : \text{hit}_{\sigma}^t[2 + \alpha] \rightarrow /l(c^{i-\alpha-1})
\]
6.3. FORWARDING

This in turn implies that registers $C.3$ and $C.4$ are used by instruction $i - \alpha - 1$
i.e.

$$used(C.(3 + \alpha), i - \alpha - 1)$$

and that the content of these registers are written into register $x$ by this
instruction. Thus we can apply $P(3, t)$ and $P(4, t)$ to conclude

$$C.(3 + \alpha)in^t = C_{i}^{t(3+\alpha,t)}$$
$$= C_{i}^{t-\alpha-1}$$
$$= gpr(x)^{i-\alpha}$$

If we have $hit_A[2 + \alpha]^t$ for $\alpha = 2$ we conclude from $gprw_{i}^{i+\alpha}$ and the
proof of $P(5, t)$

$$gprin_{i}^{t} = gprin_{i}^{t(3+\alpha,t)}$$
$$= gprin_{i}^{t-\alpha-1}$$
$$= gpr(x)^{i-\alpha}$$

The proof of $P(2, t)$ for $Ain$ can now be completed. There are two major
cases: hit or no hit.

- hit: $\exists \alpha \in [0 : 2] : hit_A[2 + \alpha]^t$. In this case we have

  $$top_A[2 + \alpha]^t$$

for the smallest such $\alpha$.

For the output $Ain$ of forwarding circuit $For_A$ we conclude

$$Ain_{i}^{t} = \begin{cases} C.(3 + \alpha)in^t & \alpha \leq 1 \\ gprin_{i}^{t} & \alpha = 2 \end{cases}$$

$$= gpr(x)^{i-\alpha}$$

If $\alpha = 0$ we have

$$gpr(x)^{i-\alpha} = gpr(x)^{i}$$

and we are done. Otherwise we have

$$I(2 + \alpha) = I(2, t) - \alpha \land \alpha \geq 1$$

Thus we can apply lemma 73 to conclude for

$$j \in [1 : \alpha - 1] \rightarrow full_{i+j} \land I(2 + j, t) = i - j$$

From $/hit_A[2 + j]^t$ we conclude by lemma 74 for all such $j$

$/writesgpr(x, i - j)$

This implies again

$$gpr(x)^{i-\alpha} = gpr(x)^{i}$$

and we are done.
• no hit: $\forall \alpha \in [0 : 2] : /\text{hit}_A[2 + \alpha]^t$. For the output $Ain$ of the forwarding circuit we have

$$Ain^t_\pi = gpr(x)^t_\pi$$

$$= gpr(x)^t_{i(5, t)}$$

If $I(5, t) = i$ we are done. Otherwise

$$\exists s \geq 1 : I(5, t) = i - s$$

Applying again lemma 73 we conclude for the instructions $i - j$ between $i$ and $i - s$

$$j \in [1 : s - 1] \rightarrow \text{full}_{i+j}^t \land I(2 + j, t) = i - j$$

From $/\text{hit}_A[2 + j]^t$ we conclude by lemma 74 for all such $j$

$$/\text{writegpr}(x, i - j)$$

This implies

$$gpr(x)^t_{i(5, t)} = gpr(x)^t_{i - s}$$

$$= gpr(x)^t_\pi$$

and we are done.

6.4 Stalling

In this last section of the pipelining chapter we use a non trivial stall engine, which permits to improve the pipelined machine $\pi$ such that we can drop software condition $SC - 2$. As shown in figure 114 the new stall engine receives from every circuit stage $\text{cir}(k)$ an input signal $\text{haz}_k$ indicating that register stage $\text{reg}(k)$ should not be clocked, because correct input signals are not available.

In case a hazard signal $\text{haz}_k$ is active the improved stall engine will stall the corresponding circuit stage $\text{cir}(k)$, but it will keep clocking the other stages if this is possible without overwriting instructions. Care has to be taken, that the resulting design is live, i.e. that stages generating hazard signals are not blocking each other.

6.4.1 Stall Engine

The stall engine we use here was first presented in [Krönig thesis] Is is quickly described but far from trivial. The signals involved for stages $k$ are
6.4. **STALLING**

- full signals $full_k$ for $k \in [0 : 4]$
- update enable signals $ue_k$ for $k \in [1 : 5]$
- stall signals $stall_k$ indicating that stage $k$ should presently not be clocked for $k \in [1 : 6]$. The stall signal for stage 6 is only introduced to make definitions more uniform.
- hazard signal $haz_k$ generated by circuit stage $k$ for $k \in [1 : 5]$

As before, stage 0 is always full and stages 1 to 4 are initially empty. Register stage $reg(6)$ does not exist, and thus it is never stalled

\[
\begin{align*}
full_0 &= 1 \\
full[1 : 4] &= 0^4 \\
stall_6 &= 0
\end{align*}
\]

We specify the new stall engine with 3 equations. Only full stages $k$ with full input registers (in stage $reg(k-1)$) are stalled. This happens in two situations: if a hazard signal is generated in stage $k$ or if the subsequent stage $k + 1$ is stalled and clocking stage $k$ would overwrite data needed in the next stage

\[
stall_k = full_{k-1} \land (haz_k \lor stall_{k+1})
\]

Stage $k$ is updated, when the preceding stage $k - 1$ is full and stage $k$ itself is not stalled

\[
ue_k = full_{k-1} \land /stall_k
\]
A stage is full in cycle \( t + 1 \) in two situations i) if new data were clocked in during the preceding cycle or ii) if it was full before and the old data had to stay where they are because the next stage was stalled.

\[
full_{k}^{t+1} = ue_{k} \lor full_{k}^{t} \land stall_{k+1}^{t}
\]

Because

\[
stall_{k+1} \land full_{k} = stall_{k+1}^{t}
\]

this can be simplified to

\[
full_{k}^{t+1} = ue_{k} \lor stall_{k+1}^{t}
\]

The corresponding hardware is shown in figure 115.

### 6.4.2 Hazard Signals

In the new design only stage 1 generates a hazard signal, namely \( A \) resp. \( B \) is used and forwarding is desirable but not possible due to a hit in stage 2 or 3 which corresponds to a load:

\[
\begin{align*}
\text{haz}_2 &= \text{haz}_A \lor \text{haz}_B \\
\text{haz}_A &= A - used \land (top_A[2] \land l.2 \lor top_A[3] \land l.3) \\
\text{haz}_B &= B - used \land top_B[2] \land l.2 \lor top_B[3] \land l.3
\end{align*}
\]
For the time being we set all other hazard signals to zero

\[ k \neq 2 \rightarrow haz_k = 0 \]

This completes the construction of the new design

### 6.4.3 Correctness statement

The correctness statement formulated in lemma 70 stays the same as before. Software conditions \( SC - 1 \) resp \( SC - 2 \) are completely dropped. Only alignment and disjoint coda and data regions are assumed.

### 6.4.4 Scheduling Functions

The correctness proof follows the pattern of previous proofs, but due to the non trivial stall engine the arguments about scheduling functions now become considerably more complex. Before we can adapt the overall proof we have to show the counter parts of lemmas 69 and 73 for the new stall engine. We begin with three auxiliary technical results

**Lemma 75.** Let \( k \geq 2 \). Then

\[ full_{k-1}^i \land ue_{k-1} \rightarrow ue_k \]

i.e. if a full stage \( k - 1 \) is clocked, then the previous data are clocked into the next stage

Proof by contradiction. Assume

\[
0 = ue_k \\
= full_{k-1} \land /stall_k \\
= /stall_k
\]

Thus

\[
stall_k = 1 \\
stall_{k-1} = full_{k-2} \land (haz_{k-1} \lor stall_k) \\
= full_{k-2} \\
ue_{k-1} = full_{k-2} \land /stall_{k-1} \\
= stall_{k-1} \land /stall_{k-1} \\
= 0
\]

**Lemma 76.**

\[ /full_k \land /ue_k \rightarrow /full_{k+1}^i \]

i.e. an empty stage \( k \) that is not clocked, stays empty
Proof:

$$\text{full}^{t+1}_k = \text{ue}_k^t \lor \text{stall}^{t+1}_k$$
$$= \text{stall}^{t+1}_k$$
$$= \text{full}^t_k \land (\text{haz}_k \lor \text{stall}^{t+1}_k)$$
$$= 0$$

Lemma 77.

$$\text{full}^{t+1}_k \lor /\text{ue}_k^t \rightarrow I(k, t + 1) = I(k, t)$$

i.e. the scheduling function of a stage $k$, that does not have a full input stage $k-1$ or that is not clocked, stays the same.

Proof: by the definitions of the scheduling functions we have

$$/\text{ue}_k^t \rightarrow I(k, t + 1) = I(k, t)$$

By the definition of the update enable functions we have

$$\text{full}^{t+1}_k \rightarrow /\text{ue}_k^t$$

We can now state the crucial counter part of lemma 69.

Lemma 78. Let $k \geq 2$. Then

$$I(k-1, t) = I(k, t) + \text{full}^{t+1}_{k-1}$$

Proof by induction on $t$. For $t = 1$ the lemma is obviously true because initially we have $\text{full}^{t+1}_0 = 0$ and $I(k-1, 1) = I(k, 1) = 1$ for all $k \geq 2$.

For the induction step from $t$ to $t+1$ assume that the lemma holds for $t$ we prove an auxiliary result.

Lemma 79.

$$\text{ue}_k^t \rightarrow I(k, t + 1) = I(k, t) + 1 \land \text{full}^{t+1}_k$$

i.e. after a stage $k$ was clocked, it is full and its scheduling function has increased by one.

In the proof we distinguish two cases. If $k = 1$ then $\text{ue}_k^t$ implies

$$I(1, t + 1) = I(1, t) + 1$$

by the definition of the scheduling functions. Now let $k \geq 2$. By the definitions of functions $\text{ue}$ and $\text{full}$ we have

$$\text{ue}_k^t \rightarrow \text{full}^{t+1}_{k-1} \land \text{full}^{t+1}_k$$
Thus we have by the definition of scheduling functions and the induction hypothesis

\[ I(k, t + 1) = I(k - 1, t) = I(k, t) + full_{k-1}^t \]

The lemma for \( t + 1 \) is now proven by a case split. Let

\[ I(k, t) = i \]

The major case split is according to bit \( full_{k-1}^t \).

- \( full_{k-1}^t = 0 \). By lemma 77 and the induction hypothesis we have

\[ I(k, t + 1) = I(k, t) = I(k - 1, t) = i \]

We consider subcases according to bit \( ue_{k-1}^t \)

- \( ue_{k-1}^t = 0 \). By lemma 76 and the definitions of the scheduling functions we conclude

\[ full_{k-1}^{t+1} \land I(k - 1, t + 1) = I(k, t) = i \]

- \( ue_{k-1}^t = 1 \). By lemma 79 and the induction hypothesis we get

\[ full_{k-1}^{t+1} \land I(k - 1, t + 1) = I(k - 1, t) + 1 = i + 1 \]

In both subcases we have

\[ I(k - 1, t + 1) = I(k, t + 1) + full_{k-1}^{t+1} \]

- \( full_{k-1}^t = 1 \). By induction hypothesis we have

\[ I(k - 1, t) = I(k, t) = i + 1 \]

By the definition of scheduling functions and lemma 79 we get

\[ (I(k - 1, t + 1), I(k, t + 1)) = \begin{cases} (i + 1, i) & \text{ue}[k - 1 : k]^t = 00 \\ (i + 1, i + 1) & \text{ue}[k - 1 : k]^t = 01 \\ (i + 2, i + 1) & \text{ue}[k - 1 : k]^t = 11 \end{cases} \]

We consider subcases according to bits \( \text{ue}[k - 1 : k]^t \in B^2 \), where

\[ \text{ue}_{k-1}^t \rightarrow \text{ue}_k^t \]

by lemma 76. Thus

\[ \text{ue}[k - 1 : k]^t \neq 10 \]
- $ue_{k-1}^i = 1$. Then
\[ full_{k-1}^{i+1} = ue_{k-1}^i \lor stall_k^i = 1 \]

- $ue_{k-1}^i = 0$. Then
\[
\begin{align*}
full_{k-1}^{i+1} &= stall_k^i \\
u_e^i &= full_{k-1}^i \land /stall_k^i \\
    &= /stall_k^i \\
    &= /full_{k-1}^{i+1}
\end{align*}
\]

In both subcases we have
\[ I(k - 1, t + 1) = I(k, t + 1) + full_{k-1}^{i+1} \]

The counter part of lemma 73 can now easily be shown

**Lemma 80.** Let $k \geq 2$ be a pipeline stage, let $1 \leq s$ and
\[ i = I(2, t) = I(k, t) + s \]

For $0 \leq j < s$ we define numbers $a(j)$ by
\[
\begin{align*}
a(0) &= 2 \\
a(j + 1) &= \min\{x : x > a(j) \land full_x^i\}
\end{align*}
\]

Then
\[ \forall j \in [0 : s - 1] : full_{a(j)}^i \land I(a(j), t) = i - j \]

The lemma follows by an easy induction on $j$. For $j = 0$ there is nothing to show. Assume the lemma holds for $j$. By the minimality of $a(j + 1)$ we have
\[ a(j) < x < a(j + 1) \rightarrow /full_x^i \]

By lemma 78 we get
\[
I(a(j + 1), t) = I(a(j), t) - \sum_{x=a(j)}^{a(j+1)-1} full_x^i
\]
\[
= I(a(j), t) - 1
\]
\[
= I(2, t) - j - 1
\]
\[
= I(2, t) - (j + 1)
\]
6.4.5 Correctness Proof

The correctness proof for the pipelined processor with forwarding and stalling follows the lines of previous proofs. The reduction of the induction step to the proof obligations $P(k, t)$ and the subsequent proofs of $P(3, t), P(4, t)$ and $P(5, t)$ relied only on lemma 69 which is now simply replaced by lemma 78.

The proof of $P(1, t)$ is simpler. Let

$$i = I(1, t)$$

We have with lemma 78

$$ima^t_i = \begin{cases} pc^t_i & full^1_i \\ dpc^t_i & /full^1_i \end{cases}$$

$$= \begin{cases} pc^t_\sigma(2, t) & full^1_i \\ dpc^t_\sigma(2, t) & /full^1_i \end{cases}$$

$$= \begin{cases} pc^t_\sigma + 1 & full^1_i \\ dpc^t_\sigma & /full^1_i \end{cases}$$

$$= dpc^t_\sigma$$

$$= ima^t_\sigma$$

In the proof of $P(2, t)$ for $Atm$ recall proof obligations $P(k, t)$ have only to be shown for cycles with active enable signals $ue^t_k$. For $k = 2$ we have

$$ue^t_2 \rightarrow /haz^t_2$$

This permits to conclude without the use of software condition $SC - 2$

$$C.(3 + \alpha)in^t = gpr(x)^t_\sigma - \alpha$$

We use lemma 80 instead of lemma 73 in two places:

- in the hit case we apply lemma 80 to show

  $$j \in [1: \alpha - 1] \rightarrow full^t_{a(j)} \land I(a(j), t) = i - j$$

  and then use lemma 74 to conclude from $/hit_A[a(j)]$:

  $$/writesgpr(x, i - j)$$

- in the case of no hit we apply lemma 80 to show

  $$j \in [1: s - 1] \rightarrow full^t_{a(j)} \land I(a(j), t) = i - j$$

  and then use lemma 74 to conclude from $/hit_A[a(j)]$:

  $$/writesgpr(x, i - j)$$
6.4.6 Liveness

We have to show that all active hazard signals are eventually turned off, so that no stage is stalled forever. By the definitions of the stall signals we have

\[
\lnot\text{stall}_{k+1} \land \lnot\text{haz}_k \rightarrow \lnot\text{stall}_k
\]
i.e. a stage, whose successor stage is not stalled and whose hazard signal is off is not stalled either. From

\[
\text{stall}_5 = \text{haz}_5 = \text{haz}_4 = \text{haz}_3 = 0
\]
we conclude

\[
k \geq 3 \rightarrow \lnot\text{stall}_k
\]
i.e. stages \(k \geq 3\) are never stalled. Stages \(k\) with empty input stage \(k-1\) are never stalled. Thus it suffices to show

**Lemma 81.**

\[
\text{full}_1^t \land \text{haz}_2^t \land \text{haz}_2^{t+1} \rightarrow \lnot\text{haz}_2^{t+2}
\]
i.e. with a full input stage, stage 2 is not stalled for more than 2 successive cycles.

Proof: from the definitions of the signals in the stall engine we conclude successively:

\[
\text{stall}_2^t = \text{stall}_2^{t+1} = 1
\]

\[
\text{full}_2^{t+1} = 1
\]

\[
\text{ue}_2^t = \text{ue}_2^{t+1} = 0
\]

Using

\[
\text{stall}_3 = \text{stall}_4 = 0
\]
we conclude successively

\[
\text{full}_3^{t+1} = \text{full}_3^{t+2} = 0
\]

\[
\text{ue}_3^{t+1} = 0
\]

\[
\text{full}_3^{t+2} = 0
\]

Thus in cycle \(t+2\) stages 2 and 3 are both empty, hence the hit signals of these stages are off

\[
X \in \{A, B\} \rightarrow \text{hit}_X[2]^{t+2} = \text{hit}_X[3]^{t+2} = 0
\]

which implies

\[
\text{haz}_2^{t+2} = 0
\]
Chapter 7

Caches and Shared Memory

We introduce caches, specify the MOESI cache coherence protocol, implement it and give a correctness proof.

7.1 Concrete and Abstract Caches

Caches are small and fast memories between the fast processor and the large but slow main memory. Transporting data between main memory and cache costs extra time, but this time is usually gained back because once data are in cache they are usually accessed several times (this is called locality) and each of these accesses is much faster than an access to main memory. Also caches are extra hardware units which increase cost. Because we do not deal here with accurate enough hardware and time models we cannot give quantitative arguments here why adding caches is cost effective. We refer the interested reader to [MP00]. Here we are interested why they work.

There are three standard cache constructions: i) direct mapped ii) k-way associative and iii) fully associative. In this section we review these three constructions and then show that - as far as their memory content is concerned - they all can be abstracted to what we call abstract caches. The correctness proof of the shared memory construction of the subsequent sections will then to a very large extent be based on abstract caches.

7.1.1 Abstract caches and cache coherence

We use very specific parameters: an address length of 32 bits, line addresses as of 29 bits, a line size of 8 bytes. If line size would be larger than the width of the memory bus, one would have to use sectored caches. This would mildly complicate the control automata. When it comes to states of cache lines we will exclusively consider the 5 states of the MOESI protocol []. We code the 5 states of the MOESI protocol in unary in the state set

\[ S = \{00001, 00010, 00100, 01000, 10000\} \]


\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
s & synonym & name \\
\hline
10000 & M & modified \\
01000 & O & owned \\
00100 & E & exclusive \\
00010 & S & shared \\
00001 & I & invalid \\
\hline
\end{tabular}
\caption{Synonyms and names of cache states $s$}
\end{table}

For the states we use the synonyms and names from table 7.1.1.

In the digital model main memory is simply a line addressable memory with configuration

$$mm : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$$

An abstract cache configuration $aca$ has the following components

- data memory $aca.data : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$. Thus, this component is simply a line addressable memory

- state memory $aca.s : \mathbb{B}^{29} \rightarrow S$ where mapping each line address $a$ to its current state $aca.s(a)$.

We denote the set of abstract cache configurations by $K_{aca}$.

If a cache line $a$ with $a \in \mathbb{B}^{32}$ has state $I$ i.e. $aca.s(a) = I$, then the data $aca.data(a)$ of this cache line is considered invalid or meaningless, otherwise it is considered valid. When cache line $a$ has valid data, we also say that we have an abstract cache hit in cache line $a$

$$ahit(aca, a) \equiv aca.s(a) \neq I$$

In case of a hit we require the data output $dout(aca, a)$ of an abstract cache to be $aca.data(a)$ and the state output $sout(aca, a)$ be $aca.s(a)$

$$ahit(aca, a) \rightarrow acadout(aca, a) = aca.data(a) \land acasout(aca, a) = aca.s(a)$$

From a single abstract cache $aca$ and a main memory $mm$ as sketched in figure 116 one can define an implemented memory $m : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$ by

$$m(a) = \begin{cases} aca.data(a) & ahit(aca, a) \\ mm(a) & \text{otherwise} \end{cases}$$

In this definition valid data in the cache hide the data in main memory. A much more practical and interesting situation arises if $P$ many abstract caches $aca(i)$ are coupled with a main memory $mm$ as shown in figure 117 to get the abstraction of a shared memory. We intend to connect such a
7.1. **CONCRETE AND ABSTRACT CACHES**

![Diagram](image)

Figure 116: A cache $ca$ and a main memory $mm$ are abstracted to a single memory $m(h)$

![Diagram](image)

Figure 117: Many caches $ca(i)$ and a main memory $mm$ are abstracted to a shared memory $m(h)$

shared memory system with $p$ processors. The number of caches will be $P = 2p$. For $i \in [0 : p - 1]$ we will connect processor $i$ with caches $ca(2i)$, which will replace the instruction memory, and with cache $ca(2i + 1)$, which will replace the data memory.

Again we want to get a memory abstraction by hiding the data in main memory by the data in caches. But this only works if we have an invariant stating coherence resp consistency of caches, namely that valid data in different caches are identical

$$aca(i).s(a) \neq I \land aca(j).s(a) \neq I \rightarrow aca(i).data(a) = aca(j).data(a)$$

The purpose of cache coherence protocols like the one considered in this chapter is to maintain this invariant. With this invariant the following definition of an implemented memory $m$ is well defined

$$m(a) = \begin{cases} aca(i).data(a) & \exists i : ahit(aca(i), a) \\ mm(a) & \text{otherwise} \end{cases}$$
### 7.1.2 Direct mapped caches

All cache constructions considered here use the decomposition of byte addresses \( a \in \mathbb{B}^{32} \) into three components as shown in figure 118.

- line offset \( ad.o \in \mathbb{B}^3 \) within lines
- cache line address \( ad.c \in \mathbb{B}^l \). This is the (short) address used to address the (small) RAMs constituting the cache.
- tag \( ad.t \in \mathbb{B}^{\tau} \) with

\[
\tau + l + 3 = 32
\]

It completes cache line addresses to line addresses

\[
ad.l = ad.t \circ ad.c
\]

For line addresses \( a \in \mathbb{B}^{29} \) this gives a decomposition into two components as shown in figure 119

- cache line address \( a.c \in \mathbb{B}^l \).
- tag \( a.t \in \mathbb{B}^{\tau} \) with

\[
a = a.t \circ a.c
\]

We structure the hardware configurations \( h \) of our constructions by introducing cache components \( h.ca \). Direct mapped caches have the following cache line addressable components

- data memory \( h.ca.data : \mathbb{B}^c \rightarrow \mathbb{B}^{64} \); a multi bank RAM.
- tag memory \( h.ca.tag : \mathbb{B}^c \rightarrow \mathbb{B}^{\tau} \); an ordinary static RAM
- state memory \( h.ca.s : \mathbb{B}^c \rightarrow \mathbb{B}^5 \); a cache state RAM
The standard construction of the data paths of a direct mapped cache is shown in figure 120. Note that cache states are stored in a cache state RAM. This permits to make all cache lines invalid by activation of the inv signal. Any data with line address \( a \) are stored in cache line address \( a.c \). At any time this is only possible for one address \( a \). The tag \( a.t \) completing the cache line address \( a.c \) to a line address \( a.l \) is stored in \( ca.tag(a.c) \).

The hardware hit signal is computed as

\[
\text{hhit}(h.ca, a) \equiv h.sa.s(a.c) \neq I \land h.ca.tag(a.c) = a.tag
\]

We define the abstract cache \( aca(h) \) for a direct mapped cache by

\[
aca(h).s(a) = \begin{cases} h.ca.s(a.c) & \text{hhit}(h.ca, a) \\ I & \text{otherwise} \end{cases}
\]

\[
aca(h).data(a) = \begin{cases} h.ca.data(a.c) & \text{hhit}(h.ca, a) \\ * & \text{otherwise} \end{cases}
\]

where * simply indicates a don’t care entry for invalid data.

**Lemma 82.** \( aca(h) \) is an abstract cache.
Figure 121: Connection of way $i$ to the data paths of a $k$-way associative cache

Proof: The hardware hit signal $hhit(h, ad)$ is active for the addresses where the abstract hit signal is on:

\[
\begin{align*}
hhit(h, ad) \equiv & \ h.sa.s(a,c) \neq I \land h.ca.tag(a,c) = a.tag \\
\equiv & \ aca(h).s(a) \neq I \\
\equiv & \ ahit(aca(h), a)
\end{align*}
\]

In case of an abstract hit $ahit(aca(h), a)$ we also have a concrete hit $hhit(h, ca, a)$. For the data and state outputs of the direct mapped cache we conclude

\[
\begin{align*}
cadout(h, ca, a) & = h.ca.data(a,c) \\
& = aca(h).data(a) \\
casout(h, ca, a) & = h.ca.s(a,c) \\
& = aca(h).s(a)
\end{align*}
\]

7.1.3 $k$-way associative caches

As shown in figure 121 these caches consist of $k$ copies $h.ca(i)$ of direct mapped caches for $i \in [0 : k - 1]$ which are called ways. Individual hit signals $hhit(i)$, cache data out signals $cadout(i)$ and cache state out signals
7.1. CONCRETE AND ABSTRACT CACHES

\( \text{casout}^{(i)} \) are computed in each way \( i \) as

\[
\begin{align*}
\text{hhit}^{(i)}(h.ca, a) &= hhit(h.ca^{(i)}, a) \\
\text{cadout}^{(i)}(h.ca, a) &= cadout(h.ca^{(i)}, a) \\
\text{casout}^{(i)}(h.ca, a) &= casout(h.ca^{(i)}, a)
\end{align*}
\]

A hit in any of the individual caches constitutes a hit in the set associative cache:

\[
\text{hhit}(h.ca, a) = \bigvee_i \text{hhit}^{(i)}(h.ca, a)
\]

Joint data output \( \text{cadout}(h.ca, a) \) and state output \( \text{casout}(h.ca, a) \) are obtained by multiplexing the individual data and state outputs under control of the individual hit signals

\[
\begin{align*}
\text{cadout}(h.ca, a) &= \bigvee_i \text{cadout}^{(i)}(h.ca) \land \text{hhit}^{(i)}(h.ca) \\
\text{casout}(h.ca, a) &= \bigvee_i \text{casout}^{(i)}(h.ca) \land \text{hhit}^{(i)}(h.ca)
\end{align*}
\]

Initialization and update of the cache must maintain the invariant, that valid tags in different ways belonging to the same cache line address are distinct

\( i \neq j \land h.ca^{(i)}.s(a.c) \neq I \land h.ca^{(i)}.s(a.c) \neq I \rightarrow h.ca^{(i)}.tag(a.c) \neq \text{h.c.a}^{(i)}.tag(a.c) \)

This implies that for every line address \( a \) hits can occur in at most one way

**Lemma 83.**

\[
\text{hhit}^{(i)}(h.ca, a) \land \text{hhit}^{(j)}(h.ca, a) \rightarrow i = j
\]

**Proof:** assume

\[
\text{hhit}^{(i)}(h.ca, a) \land \text{hhit}^{(j)}(h.ca, a) \land i \neq j
\]

Then

\[
h.ca^{(i)}.s(a.c) \neq I \land h.ca^{(i)}.s(a.c)
\]

and

\[
h.ca^{(i)}.tag(a.c) = a.t = h.ca^{(i)}.tag(a.c)
\]

This contradicts the invariant.
We now can define $aca'(h)$ by

$$aca'(h).s(a) = \begin{cases} 
  h.ca^{(i)}.s(a.c) & \exists i : hhit(h.ca^{(i)}, a) \\
  I & \text{otherwise}
\end{cases}$$

$$aca'(h).data(a) = \begin{cases} 
  h.ca^{(i)}.data(a.c) & \exists i : hhit(h.ca^{(i)}, a) \\
  * & \text{otherwise}
\end{cases}$$

This is well defined by lemma 83.

**Lemma 84.** $aca'(h)$ is an abstract cache

Proof: We have

$$hhit(h.ca, a) \equiv \exists i : hhit(h.ca^{(i)}, a)$$

$$\equiv aca'(h).s(a) \neq I$$

$$= ahit(aca'(h), a)$$

In case of an abstract hit $ahit(aca'(h), a)$ we also have by lemma 83 a unique concrete hit $hhit(h.ca^{(i)}, a)$ For the data and state outputs of the direct mapped cache we conclude

$$cadout(h.ca, a) = h.ca^{(i)}.data(a.c)$$

$$= aca(h).data(a)$$

$$casout(h.ca, a) = h.ca^{(i)}.s(a.c)$$

$$= aca(h).s(a)$$

### 7.1.4 Fully associative caches

These RAMs have the same components $h.ca.s, h.ca.tag$ and $h.ca.data$ as direct mapped caches, but data for any line address $a$ can be stored at any cache line address $a.c$. Hence the tag RAM has width 29 so that it can store entire line addresses.

Figure 122 shows data paths of a fully associative cache. All three RAMs are realized as SPR RAMS. The RAMs are addressed by a cache line address $b$ which is only used for updating the cache. For each of the RAMs $X$ one needs simultaneous access to all register contents $X[b]$. Thus they are all realized as SPR-RAMs. This together with the $2^{t}$ equality testers, that we will find in the hit signal computation, makes fully associative caches expensive.

A hit occurs at cache line address $b$ if line address $a$ can be found in the tag RAM at a valid line address

$$hhit^{(b)}(h.ca, a) = h.ca.tag(b) = a \land h.ca.s(b) \neq I$$
7.1. **Concrete and Abstract Caches**

![Diagram of a fully associative cache](image)

**Figure 122**: Data paths of a fully associative cache

A hit for the entire fully associative cache can occur at cache line address

\[ hhit(h.ca, a) \equiv 5 \bigvee b hhit^b(h.ca, a) \]

For the simultaneous equality test of the tag RAM contents \( h.ca.tag(b) \) with \( a \) one needs to realize the tag RAM as an SPR RAM.

One maintains the invariant that valid tags are distinct

\[ b \neq b' \land h.ca.s(b) \neq I \land h.ca.s(b') \neq I \rightarrow h.ca.tag(b) \neq h.ca.tag(b') \]

Along the lines of the proof of lemma 83 this permits to show the uniqueness of cache lines producing a hit.
Lemma 85.

\[ \text{hhit}^{(b)}(h.ca, a) \land \text{hhit}^{(b')}(h.ca, a) \rightarrow b = b' \]

Outputs are constructed as

\[
\begin{align*}
\text{cadout}(h.ca, a) &= \bigvee_b \text{ca.data}(b) \land \text{hhit}^{(b)}(h.ca, a) \\
\text{casout}(h.ca, a) &= \bigvee_b \text{ca.s}(b) \land \text{hhit}^{(b)}(h.ca, a)
\end{align*}
\]

We define \( \text{aca}''(h) \) by

\[
\begin{align*}
\text{aca}''(h).s(a) &= \begin{cases} 
\text{h.ca.s}(b) & \exists b : \text{hhit}^{(b)}(h.ca, a) \\
I & \text{otherwise}
\end{cases} \\
\text{aca}''(h).data(a) &= \begin{cases} 
\text{h.ca.data}(b) & \exists b : \text{hhit}^{(b)}(h.ca, a) \\
\ast & \text{otherwise}
\end{cases}
\end{align*}
\]

and show

Lemma 86. \( \text{aca}''(h) \) is an abstract cache.

Proof: We have

\[
\begin{align*}
\text{hhit}(h.ca, a) &\equiv \bigvee_b : \text{hhit}^{(b)}(h.ca, a) \\
&\equiv \text{aca}''(h).s(a) \neq I \\
&\equiv \text{ahit}(\text{aca}''(h), a)
\end{align*}
\]

In case of an abstract hit \( \text{ahit}(\text{aca}''(h), a) \) we also have by lemma 85 a unique concrete hit \( \text{hhit}(h.ca, a, b) \). For the data and state outputs of the direct mapped cache we conclude

\[
\begin{align*}
\text{cadout}(h.ca, a) &= \text{h.ca.data}(b) \\
&= \text{aca}(h).data(a) \\
\text{casout}(h.ca, a) &= \text{h.ca.s}(b) \\
&= \text{aca}(h).s(a)
\end{align*}
\]

So far we have not explained yet how to update caches. For different types of concrete caches this is done in different ways. In what follows we only elaborate details for direct mapped caches.
7.2 Notation

We summarize a large portion of the notation we are going to use in the remainder of this book.

7.2.1 Parameters

- $p$ denotes the number of processors. The set of processor IDs is $[0 : p - 1]$

- $P = 2p$ denotes the number of caches; an instruction cache and a data cache per processor. The set of cache indices is $[0 : P - 1]$

7.2.2 Memory and memory systems

The user visible memory model we aim at is line addressable multi bank RAM, i.e. memory configurations are mappings

$$m : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$$

The set of memory configurations is denoted by $K_m$.

A user visible memory will be realized by several flavors of memory systems. A memory system configuration has components

- $ms.mm : \mathbb{B}^{29} \rightarrow \mathbb{B}^{64}$. This is simply line addressable memory

- $ms.aca : [0 : P - 1] \rightarrow K_{aca}$. This is simply a sequence of abstract cache configurations

The set of memory system configurations is denoted by $K_{ms}$. In memory systems we will always keep data caches are consistent, i.e. we maintain the invariant

$$ms.aca(i).s(a) \neq I \land ms.aca(j).s(a) \neq I \rightarrow ms.aca(i).data(a) = ms.aca(j).data(a)$$

From memory systems $ms$ we abstract memories $m(ms)$ in a way described before

$$m(ms)(a) = \begin{cases} 
ms.aca(i).data(a) & ms.aca(i).s(a) \neq I \\
ms.mm(a) & \text{otherwise}
\end{cases}$$

For line addresses $a \in \mathbb{B}^{29}$ we project all components $ms.mm(a)$ and $aca(i).X(a)$ with $X \in \{data, s\}$ belonging to address $a$ in the memory system slice $II(ms, a)$
\[
\Pi(m_s, a) = (ms.aca(0).data(a), ms.aca(0).s(a),
\ldots
ms.aca(P - 1).data(a), ms.aca(P - 1).s(a)
mm(a))
\]

This definition would be much nicer if memory systems were tensors. Then a slice would simply be the submatrix with coordinate \( a \). \(^1\)

7.2.3 Accesses and Access Sequences

Memories will be accessed sequentially by accesses. Memory systems will be accessed sequentially or in parallel by accesses.

An access \( acc \) has the following components:

- processor address \( acc.a[31:3] \). A line address.
- processor data \( acc.data[63:0] \). A cache line. The input data in case of a write.
- write signal \( acc.w \)
- the byte write signals \( acc.bw[7:0] \)
- read signal \( acc.r \)
- flush request \( acc.f \). A request to flush a cache line.

At most one of bits \( w, r, f \) must be on. For technical reasons be also require the byte write signals to be off in read accesses

\[
acc.r \rightarrow acc.bw = 0^8
\]

All accesses in this section will have the property, that exactly one of bits \( w, r, f \) is on. The set of accesses is defined as \( K_{acc} \).

As the name suggests, access sequences are finite or infinite sequences of accesses. As with caches and abstract caches we use the same notation \( acc \) both for single accesses and access sequences. Access sequences come in two flavors.

---

\(^1\) Actually we could choose notation coming closer to this if we define an abstract cache configuration for single caches as

\[
aca(a) = (aca(a), s, aca(a).data)
\]

Then the abstract cache component of a a memory system slice be defined like a row of a matrix:

\[
\Pi(m_s, a) = (ms.aca([0 : P - 1], a), ms.mm(a))
\]
7.2. **NOTATION**

- sequential access sequences. This are simply mappings \( acc : \mathbb{N} \rightarrow K_{acc} \)
in the infinite case and \( acc : [0 : n - 1] \rightarrow K_{acc} \) for some \( n \) in the finite case.

- multi port access sequences

\[
acc : [0 : P - 1] \rightarrow K_{acc}
\]

where \( acc(i, k) \) denotes access number \( k \) to cache (port) \( i \).

### 7.2.4 Sequential memory semantics

Semantics of single accesses \( acc \) operating on a memory \( m \) is specified by

- a memory update function

\[
delta_M : K_m \times K_{acc} \rightarrow K_m
\]

Let

\[
m' = \delta_M(m, acc)
\]

Then memory is updated like a multi bank memory

\[
m'(a) = \begin{cases} 
\text{modify}(m(a), acc.data, acc.bw) & \text{acc.w} \land \text{acc.a} = a \\
m(a) & \text{otherwise}
\end{cases}
\]

- the answers \( \text{dataout}(m, acc) \in \mathbb{B}^{64} \) of read accesses

\[
acc.r \rightarrow \text{dataout}(m, acc) = m(acc.a)
\]

The change of memory state by sequential access sequences \( acc \) of accesses and the corresponding outputs \( \text{dataout}[i] \) are defined in the obvious way by

\[
\Delta_M^0(m, acc) = m \\
\Delta_M^{i+1}(m, acc) = \delta_M(\Delta_M^i(m, acc), acc[i]) \\
\text{dataout}(m, acc)[i] = \text{dataout}(\Delta_M^i(m, acc), acc[i])
\]

An easy induction on \( y \) shows

**Lemma 87.** Let

\[
m' = \Delta_M^x(m, acc[0 : x - 1])
\]

then

\[
\delta_M^{x+y}(m, acc[0 : x + y - 1]) = \Delta_M^y(m', acc[x : x + y - 1])
\]
7.2.5 Sequentially consistent memory systems

For multi port access sequences acc we denote by $pdataout(m, acc, i, k)$ the answer of the system to read access $acc(i, k)$ if the initial memory configuration is $m$. We often drop arguments $m$ and acc and simply write $pdataout(i, k)$.

A sequential ordering of the accesses is simply a bijective mapping

$$seq : [0 : P − 1] \times \mathbb{N} \rightarrow \mathbb{N}$$

which respects the local order of accesses, i.e. which satisfies

$$k < k' \rightarrow seq(i, k) < seq(i, k')$$

A memory system is called sequentially consistent if for any inputs $acc(i, k)$ the answers $pdataout(m, acc, i, k)$ there is a sequential ordering $seq$ satisfying the following condition.

Define the sequential access sequence $acc'$ by

$$acc'[seq(i, k)] = acc(i, k)$$

Then for read accesses $acc(i, j)$ the answer $pdataout(m, acc, i, k)$ to access $acc(i, k)$ of the multi port access sequence acc is the same as the answer $dataout(\Delta_M(m, acc')[seq(i, k)])$ of the sequential access sequence $acc'$

$$pdataout(m, acc, i, k) = dataout(\Delta_m(m, acc')[seq(i, k)])$$

By the definition of function $dataout$ this is equivalent to

$$pdataout(m, acc, i, k) = \Delta_M^{seq(i,k)}(m, acc')(acc'(seq(i, k)).a) = \Delta_M^{seq(i,k)}(m, acc')(acc(i, k)).a$$

7.2.6 Memory system hardware configurations

We collect the components of a hardware configuration of a memory system into the following components

- main memory component $h.mm$
- cache components $h.ca(i)$. In theses components we collect cache RAMs $h.ca(i).X$ for $X \in \{data, s, tag\}$ which we have already introduced, but later also for registers $h.ca(i).Y$ of the cache control and the data paths of cache $i$.

We denote by

$$aca(i) = aca(h.ca(i))$$
the abstract cache abstracted from cache RAMs $h.ca(i).X$ of cache $i$ as explained in section 7.1. For hardware cycles $t$ the states of hardware cache $i$ and abstract cache $i$ in cycle are denoted by

$$\begin{align*}
ca(i)^t &= h^t.ca(i) \\
aca(i)^t &= aca(h^t.ca(i))
\end{align*}$$

For components $X \in \{data, s\}$ of abstract caches and components $Y$ of hardware cache $h.ca(i)$ we use the notations

$$\begin{align*}
ca(i).Y^t &= h^t.ca(i).Y \\
aca(i).X^t &= aca(h^t.ca(i))
\end{align*}$$

The hardware constitutes a memory system

$$ms(h) = (ms(h).mm, ms(h).aca)$$

with

$$\begin{align*}
ms(h).mm &= h.mm \\
ms(h).aca &= aca(h.ca(i))
\end{align*}$$

which in turn permits the definition of a memory abstraction

$$m(h) = m(ms(h))$$

### 7.3 Atomic MOESI Protocol

We specify the MOESI protocol in six steps

1. for any memory system $ms$ system of abstract caches $ms.aca(i)$ and main memory $ms.mm$ we formulate the state invariants for the five states $M, O, E, S, I$ involved in the protocol.

2. we specify the format of memory accesses $acc(i, k)$ where $i$ specifies the cache accessed and $k$ numbers the accesses to this cache.

3. we present the protocol in a way that is common in literature, namely by tables prescribing how to run the protocol one access at a time. We give this version of the protocol a special name and call it atomic, because it performs each access in an atomic way without interference of any other access and by the way...sequentially.

4. as a first step to formalize the protocol we translate the master and slave tables into switching functions $C1, C2$ and $C3$
5. using functions $C_i$ we give an algebraic specification of the atomic MOESI protocol.

6. we specify how the caches and possibly the main memory exchange data after the protocol information of step 3 has been exchanged.

We then review the classical definition of *sequentially consistent shared memory* and the classical proof that a system of caches ca(s) and main memory $mm$ running the atomic MOESI protocol behaves like memory. Observe that the atomic system is sequentially consistent for completely trivial reasons: it runs sequentially.

There are bad news and good news about the classical result. The bad news first: for practical purposes a literal translation of the atomic protocol into hardware would be plain madness: if one sequentializes memory accesses anyway, one should simply use a single cache with a main memory; end of story. The good news and the beauty of the protocol as introduced in [1] is, that it permits a parallel implementation which nevertheless simulates the atomic protocol. In the later sections of this chapter we give to the best of our knowledge the first such construction in the open literature and prove (on paper) that it works. Indeed the classical result, that state invariants are preserved in the atomic protocol will be a crucial lemma in our proof. It will however be only one of 11 statements in the main induction hypothesis.

### 7.3.1 Invariants

For the memory system $ms$ under consideration we abbreviate

$$mm = ms:mm$$
$$aca = ms:aca$$

One calls the data in a cache line clean if this data is known to be the same as in the main memory, otherwise it is called dirty. A line is exclusive if the line is known to be only in one cache, otherwise it is called shared. The intended meaning of the states is:

- $E$: exclusive clean (the data is in one cache and is clean)
- $S$: shared (the data might be in other caches and might be not clean)
- $M$: exclusive modified (the data is in one cache and might be not clean)
- $O$: owned (the data might be in other caches and might be not clean; the cache with this line in owned state is responsible for writing it back to the memory or sending it on demand to other caches)
- $I$: invalid (the data is meaningless)
7.3. **ATOMIC MOESI PROTOCOL**

This intended meaning is formalized in a crucial set of state invariants:

1. states E, M are exclusive; in other caches the line is invalid.

\[
aca(i).s(a) \in \{E, M\} \land j \neq i \rightarrow aca(j).s(a) = I
\]

2. state E is clean:

\[
aca(i).s(a) = E \rightarrow aca(i).data(a) = mm(a)
\]

3. shared lines, i.e. lines in state S are clean or they have an owner

\[
aca(i).s(a) = S \rightarrow (aca(i).data(a) = mm(a) \quad \forall j \neq i : aca(i).s(a) = O)
\]

4. data in lines in nonexclusive state are identical:

\[
aca(i).s(a) = S \land aca(j).s(a) \in \{O, S\} \quad \rightarrow aca(i).data(a) = aca(j).data(a)
\]

5. if a line is non-exclusive, i.e. in state S or O, other copies must be invalid or in a non exclusive state. Moreover the owner is unique.

\[
aca(i).s(a) = S \land j \neq i \rightarrow aca(j).s(a) \in \{I, O, S\}
aca(i).s(a) = O \land j \neq i \rightarrow aca(j).s(a) \in \{I, S\}
\]

We introduce the notation \(\text{sinv}(ms)(a)\) to denote that the state invariants hold for cache line address \(a\) with a system \(aca\) of abstract caches and main memory \(mm\). For cycle numbers \(t\) we denote by \(SINV(t)\) the fact that the state invariants hold for the memory system \(ms(h)\) abstracted from the hardware for all cycles \(t' \in [0 : t]\), i.e. from cycle 0 after reset until \(t\).

\[
\text{sinv}(ms) \leftrightarrow \forall a : \text{sinv}(ms)(a)
SINV(t) \leftrightarrow \forall t' \in [0 : t] : \text{sinv}(ms(h'))
\]

One easily checks, that the state invariants hold if all cache lines are invalid. In the hardware construction, this will be the state of caches after reset.

**Lemma 88.**

\[
\forall a, i : aca(i).s(a) = I \rightarrow \text{sinv}(ms)
\]

The accesses operate on a memory system of the atomic protocol with main memory configurations \(mm1\) and abstract cache configurations \(ca1(i)\), where the 1 indicates atomic or one step execution of accesses.
7.3.2 Defining the protocol by tables

We stress the fact that the atomic protocol is a sequential protocol operating on a multi port memory system \( ms \). Its semantics is defined by a two functions:

- a transition function

\[
\delta_1 : K_{ms} \times K_{acc} \times [0 : P - 1] \rightarrow K_{ms}
\]

where

\[
ms' = \delta_1(ms, acc, i)
\]

defines the new memory system if single access \( acc \) is applied to port (cache) \( i \) of memory system \( ms \).

- an output function

\[
dataout1 : K_{ms} \times K_{acc} \times [0 : P - 1] \rightarrow B^{64}
\]

where

\[
d = dataout1(ms, acc, i) =
\]

specifying for read accesses (i.e. accesses with \( acc.r \)) cache line \( d \) the memory system outputs as response to access \( acc \) at port \( i \) in memory system configuration \( ms \).

We abbreviate

\[
\begin{align*}
mm' &= ms'.mm \\
aca' &= ms'.aca
\end{align*}
\]

The processing of accesses is summarized in tables 123 a) and 123 b).

We first describe somewhat informally how the tables are interpreted. In subsection 7.3.4 we will translate description into into an algebraic specification.

Every access \( acc(i, k) \) is processed by cache \( ca1(i) \) which is called the master of the access. Actions of the master are specified in table 123 a). The master determines the local state \( aca(i).s(acc(i, k).a) \) of cache line \( acc.a \) and the type of the access, i.e. whether the access is a read, write or flush. The state determines the row of the table to be used. The type of the access determines the column.

There are two kinds of table entries: i) single states and ii) others.

A single state indicates that a cache can handle the access without contacting the other caches; for some flushes it still may have to write back a cache line to main memory. In case i) the table entry specifies the next state of the cache line. The table does not explicitly state how data are to
be processed; this is implicitly specified by the fact that we aim at a memory construction and by the state invariants. We will make this explicit in subsection 7.3.4.

In case there is more than a single state in the master table entry, the protocol is run in four steps. Three steps concern the exchange of signals belonging to the memory protocol and the next state computation. The fourth step involves the processing of the data and is only implicitly specified.

1. out of three master protocol signals $Ca$, $im$, $bc$ the master activates the ones specified in the table entry. These signals are broadcast to the other caches $ca(j), j \neq i$ which are called the slaves of the access. The intuitive meaning of the signals is

   • $Ca$: intention of the master to cache line $acc.a$ after the access is processed
   • $im$: intention of the master to modify (write) the line
   • $bc$: intention of the master to broadcast the line after the write has been performed. This signal is activated after a write hit with non exclusive data.

2. the slaves $j$ determine the state local state $aca(i),s(acc.a)$ of cache line $acc.a$, which determines the row of the slave table 123 b) to be used.
The column is determined by the values of the master protocol signals ca, im and bc. Each slave aca(j) goes to a new state as prescribed in the slave table entry and activates two slave protocol signals ch(j) and di(j) as indicated by the slave table entry used. The intuitive meaning of the signals is

- ch(j): cache hit in slave aca(j)
- di(j): data intervention by slave aca(j). Slave aca(j) has the cache line needed by the master and will put it on a bus, where the master can access i.

The individual signals are OREd together (in active low form on an open collector bus) and made accessible to the master as

\[ ch = \bigvee_j ch(j), \quad di = \bigvee_j di(j) \]

3. The master determines the new state of the cache line accessed as a function of the slaves responses as indicated by the table entry used. The notation ch?X/Y is an expression borrowed from C and means

\[ ch?X/Y = \begin{cases} X & \text{ch} \\ Y & /\text{ch} \end{cases} \]

4. data processing as implicitly specified

### 7.3.3 Translating the tables into sets of switching functions.

We extract from the tables three sets of switching functions. They correspond to phases of the protocol, and we specify them in the order, in which they are used in the protocol.

- **C1**: this function is used by the master. It depends on a state \( s \in S \) and the write signal \( w \). It computes the master protocol signals C1.Ca, C1.im and C1.bc. Thus

\[ C1 : S \times B \rightarrow B^3 \]

The component functions C1.X are defined by translating the master protocol table, i.e. looking up the corresponding cell \((state, w)\) in table 123a) and choosing the necessary protocol bits accordingly.

\[ \forall X \in \{Ca, im, bc\} : C1.X(s, w) = 1 \Leftrightarrow \text{master_table}(s, w) \text{ contains } X \]

Using the construction of lemma 20 for each component C1.X the above switching function can be turned into a switching circuits that we also call C1. A symbol for this circuit is shown in figure 124.
7.3. ATOMIC MOESI PROTOCOL

- **C2**: This function is used by slaves. It depends on a cache state \( s \in S \) and the master protocol signals \( ca, im \) and \( bc \). It computes slave protocol signals \( C2.ch \) and \( C2.di \), i.e. the slave response and the next state \( C2.ss' \) for slaves. Thus

\[
C2 : S \times \mathbb{B}^3 \to \mathbb{B}^2 \times S
\]

For \( x \in \{ch, di\} \) component functions \( C2.X \) are defined by translating the slave protocol table, i.e. looking up the corresponding cell \((state, Ca, i, bc)\) in table 123b) and choosing the necessary protocol bits accordingly.

\[
\forall X \in \{ch, di\} : f_X(code(s), mprotin) = 1 \leftrightarrow slave_table(s, Ca, im, bc) \text{ contains } X
\]

\( C2 \) also computes the next state of the slave

\[
C2.ss' = s' \leftrightarrow slave_table(s, Ca, im, bc) \text{ contains } s'
\]

A symbol for the corresponding circuit is also shown in figure 124. TODO: add input \( f \) to circuit \( C3 \)

- **C3**: This function depends on a state \( s \in S \), the write signal \( w \), the flush signal \( f \) and the slave response \( ch \). It computes the next state \( C3.ps' \) of the master. Thus

\[
C3 : S \times \mathbb{B}^3 \to S
\]

The function is defined by translating the master protocol table

\[
ps' = s' \leftrightarrow master_table(s, ch, w) \text{ contains } s'
\]

\[
\forall \exists s^w : master_table(s, w) \text{ contains } ch?s : s^w' \wedge ch
\]

\[
\forall \exists s^v : master_table(s, w) \text{ contains } ch?s : s \wedge /ch
\]

The corresponding circuit symbol is also shown in figure 124

7.3.4 Algebraic specification of the atomic MOESI protocol

For the following definitions we assume \( sim(ms) \), i.e. that the state invariants hold for the memory system \( ms \) before the (sequential and atomic) processing of access \( acc \) at port \( i \).

For all components \( x \) of access \( acc \) we abbreviate

\[
 x = acc.x
\]
Figure 124: Symbols for circuits C1, C2 and C3 computing the protocol signals and next state functions of the MOESI protocol

Note, that a in this section and below where applicable denotes the line address acc.a. Also, the functions we define depend on arguments ms.aca and ms.mmm. For brevity of notation we will omit these arguments most of the time - but not always - in the remainder of this section. We now proceed
to define the effect of applying accesses acc to port i of memory system ms
by specifying functions ms' = δ1(ms, acc, i) and d = dataout1(ms, acc, i)

We only specify the components that do change. We define a hit at
atomic abstract cache aca(i) by

\[ \text{hit}(i, a) = aca(i).s(a) \neq I \]

We say that a read or write access (an access with w ∨ r) to an atomic
cache system configuration aca at prot i is local, if it can be processed by
accessing the local cache only, i.e. if it is a read hit or a write hit in exclusive
state. It is global otherwise

\[
\begin{align*}
\text{local}(aca, acc, i) & \equiv \text{hit}(i, a) \land (r \lor w \land aca(i).s(a) \in \{E, M\}) \\
\text{global}(aca, acc, i) & \equiv \neg\text{local}(aca, acc, i)
\end{align*}
\]

Now we define the transition function for every possible type of access.

We aim

- to maintain the state invariants, i.e. for sinv(ms') and

- that the resulting memory abstraction m(ms') behaves, as if access acc
  would have applied with ordinary memory semantics to the previous
  memory abstraction m(ms)

\[ m(ms') = \delta_M(m(ms), acc) \]

- that the response d to read accesses is is response given by the memory
  abstraction m(ms)

\[ \text{dataout1}(ms, acc, i) = \text{dataout}(m(ms), acc) = m(ms)(acc.a) \]
7.3. ATOMIC MOESI PROTOCOL

flush

A flush invalidates abstract atomic cache line \( a \) and writes back the cache line in case it is modified or owned

\[
f \implies aca'(i).s(a) = I \land (aca(i).s(a) \in \{M, O\} \implies mm'(a) = aca(i).data(a))
\]

Local write accesses

Local writes update the local cache line addressed by \( a \) and change the state to \( M \)

\[
\begin{align*}
\text{local}(aca, acc, i) \land w & \implies \\
aca'(i).data(a) &= \text{modify}(aca(i).data(a), data, bw) \\
aca'(i).s(a) &= M
\end{align*}
\]

Global read write accesses

For global accesses we run the MOESI protocol in an atomic way.

\[
\begin{align*}
mprot &= C1(aca(i).s(a), w) \\
\forall j : sprot(j) &= C2(mprot, aca(i).s(a)).(ch, di) \\
sprot &= \bigvee_j sprot(j)
\end{align*}
\]

\[
\forall j : aca'(j).s(a) = \begin{cases} 
C3(aca(i).s(a), sprot, w).ps' & i = j \\
C2(aca(j).s(a), mprot).ss' & \text{otherwise}
\end{cases}
\]

Next we specify the data \( bdata \) broadcast via the bus during a global transaction. For write hits \((hit(i, a) \land w)\) the master broadcasts the modified result \( \text{modify}(aca(i).data(a), data, bw) \). For misses without data intervention \( \neg(hit(i, a) \land \neg sprot.dl) \) the missing line \( mm(a) \) is provided by the memory. For misses with intervention \( \neg(hit(i, a) \land sprot(j).di) \) the intervening slave \( j \) is unique by the state invariants \( sinv(ms) \); the intervening slave provides the missing line \( aca(j).data(a) \):

\[
\begin{align*}
bdata &= \begin{cases} 
\text{modify}(aca(i).data(a), data, bw) & mprot.bc \\
mm(a) & \neg sprot.dl \land \neg mprot.bc \\
aca(j).data(a) & sprot(j).dl
\end{cases}
\end{align*}
\]

After a read miss \( \neg hit(i, a) \land \neg w \) the master copies the missing cache line from the bus

\[
aca'(j).data(a) = \begin{cases} 
\text{bdata} & j = i \\
aca(j).data(a) & \text{otherwise}
\end{cases}
\]
CHAPTER 7. CACHES AND SHARED MEMORY

After a write hit \( \text{hit}(i, a) \land w \) the master \( i \) and the caches signalling a cache hit \( \text{sprot}(j).ch \) store the modified result

\[
\text{aca}'(j).\text{data}(a) = \begin{cases} 
\text{bdata} & j = i \lor \text{sprot}(j).ch \\
\text{aca}(j).\text{data}(a) & \text{otherwise}
\end{cases}
\]

Note, that after a write hit the master and the affected slaves store the same data for address \( a \).
After a write miss \( \neg \text{hit}(i, a) \land w \) the master reads the data from the bus or from the memory and modifies it.

\[
\text{aca}'(j).\text{data}(a) = \begin{cases} 
\text{modify(bdata, data, bw)} & j = i \\
\text{aca}(j).\text{data}(a) & \text{otherwise}
\end{cases}
\]

Answer of read

After a read request we run the protocol and return either the copy of the data from the intervening cache or from the memory:

\[
\text{dataout}(ms, acc, i) = \begin{cases} 
\text{aca}(i).\text{data}(a) : \text{hit}(i, a) \\
\text{aca}(j).\text{data}(a) : \text{sprot}(j).di \\
\text{mm}(a) : \text{otherwise}
\end{cases}
\]

Iterated transitions

For memory systems \( ms \), linear access sequences \( acc' \), sequences \( i \) of ports and step numbers \( n \) we define the effect of \( n \) steps of the atomic protocol in the obvious way

\[
\Delta^0_1(ms, acc', i) = ms \\
\Delta^{n+1}_1(ms, acc', i) = \delta_1(\Delta^n_1(ms, acc', i), acc'(n), i(n))
\]

The following lemma is proven by an easy induction on \( y \)

**Lemma 89.** Let

\[ ms' = \Delta^y_1(ms, acc', i) \]

Then

\[ \Delta^{x+y}_1(ms, acc', i) = \Delta^y_1(ms', acc'[x : x + y - 1], i[x : y - 1]) \]

### 7.3.5 Properties of the atomic protocol

**Lemma 90.** In the atomic execution of the MOESI protocol the state invariants are preserved,

\[ sinv(ms) \implies sinv(ms') \]
7.3. ATOMIC MOESI PROTOCOL

Proof. The proof of this lemma is error prone, so it is usually shown by model checking.[TODO: Literature] □

An easy proof shows that we have achieved two more goals that were stated before

**Lemma 91.**
- the resulting memory abstraction \(m(ms')\) behaves, as if access acc would have applied with ordinary memory semantics to the previous memory abstraction \(m(ms)\)

\[
m(ms') = \delta_M(m(ms), acc)
\]

- the response \(d\) to read accesses is is response given by the memory abstraction \(m(ms)\)

\[
dataout1(ms, acc, i) = dataout(m(ms), acc)) = m(ms)(acc.a)
\]

By induction we get

**Lemma 92.**

\[
m(\Delta^y_M(ms, acc', i') = \Delta^y_M(m(ms), acc')
\]

The following technical lemma formalizes the fact that the abstract protocol with an access acc only operates on memory system slice \(\Pi(ms, acc.a)\). The reader might have observed that this address does not even occur in the tables specifying the protocol, because everybody understands, that line address \(aca.a\) (alone) is concerned in each cache. Readers familiar with cache designs will of course observe, that read or write accesses acc can trigger flushes evicting cache lines with line addresses \(a' \neq acc.a\); but these are treated as separate accesses in our arguments.

**Lemma 93.** Let

\[
ms' = \delta_1(ms, acc, i) \quad \text{and} \quad a = acc.a
\]

Then

1. read hits don’t change the memory system

\[
\text{hit}(a, ms) \land acc.r \rightarrow ms' = ms
\]

2. slices different from slice \(acc.a\) of the memory system are not changed

\[
b \neq acc.a \rightarrow \Pi(ms'a) = \Pi(ms, a)
\]

3. possible changes to slice \(acc.a\) only depend on slice \(acc.a\)

\[
\Pi(ms_1, a) = \Pi(ms_2, a) \rightarrow \Pi(\delta_1(ms_1, acc, i), a) = \Pi(\delta_1(ms_2, acc, i), a)
\]
4. The answer of reads to address \(a\) depends only on \(a\)

\[
\text{acc.a0a0aII}(ms_1, a) = \text{II}(ms_2, a) \rightarrow \text{dataout1}(ms_1, acc, i) = \text{dataout1}(ms_1, acc, i)
\]

Proof. 1. for read hits we specified no change of \(ms\)

2. In the definition of function \(\delta_1\) we only specified components, that change. Slices other than slice \(acc.a\) were not among them.

3. This is a simple bookkeeping exercise, where one has to compare all parts of the definition of function \(\delta_1\) for the two memory system configurations \(ms_1\) and \(ms_2\)

4. bookkeeping exercise

\[\square\]

7.4 Gate Level Design of a Shared Memory System

We present the construction of a gate level design of a shared memory system in the following order.

1. we specify in this section the interface between processors \(p(j)\) and caches \(ca(i)\) and the interface between caches \(ca\) and and the main memory bus \(b\). Bus \(b\) is extended by a component \(b.mp\) for the exchange of protocol signals. This component is an open collector bus.

2. we specify the data paths of each cache \(ca(i)\). These data paths have three obvious components for the data, tag and state RAMs of the cache. The third part contains circuits \(C1, C2\) and \(C3\) introduced in subsection 7.3.3 implementing the tables of the MOESI protocol. Each cache \(ca(i)\) may have to serve two purposes simultaneously: i) serving its processor as a master of accesses and ii) participating as a slave in the protocol. Therefore, all RAMS \(ca(i).data, ca(i).tag\) and \(ca(i).s\) will be implemented as dual ported RAMs.

3. we present control automata. Each cache \(ca(i)\) has two such automata: one for accesses where \(ca(i)\) is master and one for accesses when \(ca(i)\) is slave. Thus in a system with \(P\) caches we have \(2P\) control automata. Showing that master and slave automata are in some sense synchronized while they are handling the same access will be a crucial part of the correctness proof.

\[\text{This proof can be avoided if one defines function } \delta_1 \text{ directly as a function of slice } \text{II}(ms, acc:a), \text{ but this definition does not match the hardware design so well.}\]
4. accesses requiring cooperation of caches via the memory bus \( b \) are called \textit{global} accesses. In case several caches want to initiate a global access at the same time (as masters) a \textit{bus arbiter} has to grant the bus to one of them and deny it to the others.

### 7.4.1 Specification of interfaces

We need to establish interfaces between

1. processors \( p \) and their caches. This is done by signals
2. the caches \( ca(i) \) and the bus \( b \). This is done via dedicated registers.

\( p \rightarrow ca \) \textbf{Interface}

Signals from a processor \( p \) to cache \( ca(i) \)

- \( ca(i).pdat \) - processor data coming into the cache (for writes)
- \( ca(i).pa \) - processor address bus
- \( ca(i).pw \) - processor write signal
- \( ca(i).bw[7:0] \) - byte write signals. They must be off for read requests:

\[ ca(i).preq \land \neg ca(i).pw \rightarrow ca(i).bw[7:0] = 0^8 \]

- \( ca(i).preq \) - processor request signal

\( ca \rightarrow p \) \textbf{Interface}

Signals from cache to processor:

- \( ca(i).mbusy \) - memory system is busy (generated by control automaton)
- \( ca(i).Dout \) - data out to processor

\( ca \rightarrow b \) \textbf{Interface}

The following dedicated registers are used between cache \( ca(i) \) and bus \( b \)

Data:

- \( ca(i).bdataout \) - cache data out to bus
- \( ca(i).bdatalin \) - cache data in from bus

Address:
- $ca(i).badout$ - master address out (for signalling a line address to other caches)
- $ca(i).badin$ - slave address in (for snooping, triggers data intervention/forwarding)

Protocol (for cache coherence protocol signals):
- $ca(i).mprot\text{out}$, $ca(i).sprot\text{out}$ - protocol data out on bus
- $ca(i).mprot\text{in}$, $ca(i).sprot\text{in}$ - protocol data in from bus

Memory bus

The memory bus $b$ is subdivided into 4 sets of bus lines. The first three are already known from bus connecting to the main memory. The corresponding bus components are tri state buses The fourth set of lines supports the cache protocol and is an open collector bus.

- $b.data$ - for transmitting data contained in a cache line
- $b.ad$ - memory line address ($tag \cdot line$)
- $b.mmreq, b.mmw, b.ack$ - memory protocol lines
- $b.prot$ - cache protocol lines

We use the 5 obvious protocol signals which are dedicated either to the master or slave role in the protocols:

$$b.\text{prot}[4:0] = b.\text{mprot}[2:0] \circ b.\text{sprot}[1:0]$$

The following synonyms are used:

\[
\begin{align*}
b.Ca & = b.\text{mprot}[2] \\
b.im & = b.\text{mprot}[1] \\
b.bc & = b.\text{mprot}[0] \\
b.ch & = b.\text{sprot}[1] \\
b.di & = b.\text{sprot}[0]
\end{align*}
\]

As shown in figure 7.4.1 (TODO: right part only; delete left part; $bus.prot$ should be $b.prot$) memory protocol signals are inverted before they are put on the open collector bus and before they are clocked from the bus into a register. Thus by de Morgan’s law we have for every component $x \in [0:4]$ by de Morgan’s law

$$/b.\text{prot}[x] = / \bigwedge_{j} /ca(j).bprot\text{out}[x] = \bigvee /ca(j).bprot\text{out}[x]$$
7.4. Gate Level Design of a Shared Memory System

![Diagram](image)

Figure 125: Using de Morgan's law to compute the OR of active high signals on the open collector bus $b_{prot}$

When several slaves signal a data intervention, further bus arbitration appears to be necessary, since only one cache should access the bus at a time. However, arbitration is not necessary as long as only one slave will forward the required cache line. This is guaranteed by the cache coherency protocol, where we do not raise DI in case of a miss on data in state $S$. However the protocol provides that all caches keep the same data when it is shared, so that we could in principle forward the data if we arbitrate the data intervention. A possible arbitration algorithm for data intervention in a “shared clean miss” case would be to select $ca(i)$ with the smallest $i$ s.t. $DI$ is active. This can be efficiently implemented using a parallel-prefix-OR circuit.

$p \leftrightarrow ca$ Protocol

We need to define a protocol for interaction between a processor and its caches (data & instruction cache). It uses the following signals: $mbusy$, $preq$, $pa$, $pdatin$, $stallin$. The timing diagram for a $k$-cycle cache access with an inactive $stallin$ signal is depicted in figure 126. An active stall in signal inhibits the start of an access.

The following observations can be made:

- $p$ starts a request by activating $preq$
- $ca$ acknowledges by raising $mbusy$ (a Mealy signal)
- $ca$ finishes with lowering $mbusy$, $p$ disables $preq$ in the next cycle
- start of access in cycle $t$: $/mbusy^{t-1} \land preq^t$
- end of access in cycle $t + k$: $/mbusy^{t+k} \land preq^{t+k}$
- Observe, that 1-cycle accesses are desirable and indeed possible (in a read hit). Then mbusy is not raised at all and the processor can immediately start a new request in cycle $t + 1$.

Once the processor request signal is raised, inputs from the processor must be stable until the cache takes away the mbusys signal. In order to formalize this condition we identify the cache input signals of cache $ca(i)$ in cycle $t$ as

$$cain(i, t) = \{pa, pw, bw, preq\} \cup \begin{cases} \{pdin\} & ca(i).preq^t \land ca(i).w^t \\ \emptyset & \text{otherwise} \end{cases}$$

and then require

$$ca(i).preq^t \land ca(i).mbusy^t \land X \in cain(i, t) \rightarrow ca(i).X^{t+1} = ca(i).X^t$$

### 7.4.2 Data paths of caches

The data paths for the Data RAM, State RAM and Tag RAM are presented on Figures 7.4.2, 7.4.2, 7.4.2 respectively.

The control signals for the data paths are generated by the control automata described in the following subsection. Let us try to get a first understanding of the designs.

#### Data paths of the data RAM

TODO: bus should be b. 64 - Bit - Modifier should be modify. Maybe we should put index [c] on the address lines. data RAM data inputs and outputs should start with $d$.

\footnote{Stability in the digital sense is enough; the processors never access main memory directly}
Figure 127: Datapaths for the Data-RAM of a cache. Only components\n\textit{b.ad.c} and \textit{ca(i).pa.c} are used by the data RAM.
CHAPTER 7. CACHES AND SHARED MEMORY

\[
\text{byte}(i, x) \quad \text{byte}(i, y) \quad \text{byte}(i, \text{modify}(x, y, \text{bw}))
\]

\[
\begin{array}{c}
8 \\
4 \\
0 \\
\end{array}
\begin{array}{c}
8 \\
4 \\
0 \\
\end{array}
\begin{array}{c}
\text{bw}[i] \\
\text{byte}(i, \text{modify}(x, y, \text{bw})) \\
\end{array}
\]

Figure 128: Computation of output byte \(i\) of a \text{modify} circuit by a multiplexer controlled by byte write signal \(\text{bw}[i]\)

In general RAMs are controlled from two sides: i) from the processor side using signals ending with \(A\) and ii) from the bus side using signals ending with \(B\) The data paths in figure ??must support the following operations:

- read hit. The processor addresses the \(A\) side with \(pa\). The hit is signalled by a processor hit signal \(\text{phit}\). Data RAM output \(dout A\) is routed to the data output \(pdout\) at the processor side.

- write hit. It requires two cycles which together perform a read-modify-write operation. The cache line addressed by \(pa\) is read out and temporarily stored in register \(X\). From there it becomes an input to a \text{modify} circuit which computes the modify function previously defined in section 3.2.3

\[
\text{byte}(i, \text{modify}(x, y, \text{bw})) = \begin{cases} 
\text{byte}(x, i) & \text{bw}[i] = 1 \\
\text{byte}(y, i) & \text{bw}[i] = 0 
\end{cases}
\]

As shown in figure 128 each byte of the output of a modifier circuit is simply computed by an 8 bit wide multiplexer. For any kind of write, data to be written \(y\) and byte write signals \(\text{bw}\) come from the processor. For a write hit, the cache line, that is modified comes from register \(X\). The result is written to the data RAM via port \(dina\). In case the line addressed was not exclusive the result of the modifier is also broadcast on the bus via register \(bdout\).

- flush. Except for times when the cache is filling up a cache miss is generally preceded by a flush: a so called em victim line with some eviction line address \(va\) is evicted from the cache in order to make space for the missing line. In a direct mapped cache the eviction address has cache line address

\[
va.c = pa.c
\]

In case of a miss the victim line is taken from output \(dout A\) of the data RAM and put on the bus via register \(bdout\).
7.4. GATE LEVEL DESIGN OF A SHARED MEMORY SYSTEM  223

- write miss. The missing line is clocked from the bus into register \(bdin\). From there it becomes input to the modifier. The output of the modifier is written back to the data RAM at input \(DinA\).

- read miss. The missing line is clocked from the bus into register \(bdin\). The modifier with byte write signals \(bw = 0^d\) is used as a data path for the missing cache line. It is output to the processor via signal \(pdout\) and written into the data RAM via input \(DinA\).

- data intervention. The line address is clocked from the bus into register \(badin\). The intervention line missing in some other cache is taken from output \(doutB\) of the data RAM and put on the bus via register \(bdout\).

Data paths of the tag RAM

TODO: [line] should be [c]. \(tag\) should be \(t\). bus width \(t\) should be \(r\). In data ports \(D\) should be \(d\).

The tag RAM is very much wired like a tag RAM in an ordinary direct mapped cache. It is address from the processor side by signal \(pa\) and from the bus side by register \(bdin\).

- New tags are only written into the tag RAM from the processor side.

- Hits signals \(phit\) for the processor side and \(bhit\) for the bus side are computed from outputs \(doutA\) and \(doutB\).

- For global accesses the processor address can be put on the bus via register \(bdout\).

- For flushes the tag of the victim address taken from output \(doutA\) of the tag RAM. The victim line address is then

\[va = doutb \odot pa.c\]

is put on the bus via register \(bdout\).

Data paths of the state RAM

TODO: renaming signals as above. I think the top multiplexer is not needed. setting the state to \(M\) in case of a local write should be handled by circuit \(C3\). Also the second mux should go. Circuit \(C3\) now has an \(f\) input for flush. Connect \(flush \land (6)\) there. drivers should be \(OC\).

As before, addressing from the processor side is by signal \(pa\) and from the bus side by register \(bdin\). Some control signals come from the control automata and are explained in subsection 7.4.3. The data paths of the state RAM use the circuits \(C1, C2\) and \(C3\) from subsection 7.3.3 which compute the the memory protocol signals and the next state of cache lines.
Figure 129: Datapaths for the Tag-RAM of a cache

CHAPTER 7. CACHES AND SHARED MEMORY
Figure 130: Datapaths for the State-RAM of a cache
• the current master state is read from state RAM output \( doutA \)

• in case of a local read or write or a flush the new state \( ps' \) is computed by circuit \( c3 \) and written back to input \( dinA \) of the state RAM.

• otherwise the master protocol signals are computed by circuit \( C1 \) and put on the bus via register \( mprotout \). The mux on top of \( C1 \) forwards the effects of flushes. The mux on top of register \( mprotout \) allows to clear the master protocol signals after a run of the protocol.

• if the cache works as a slave, it determines the slave response with circuit \( C2 \) using the state from output \( doutB \) of the state RAM and puts in on the bus via register \( sprotout \). The mux on top of circuit \( C2 \) forwards the effect of local writes whose line address conflicts with the line address of the current global access. The mux on top of register \( sprotout \) allows to clear the slave response after a run of the protocol.

• in a global read or write access the master determines its new state \( ps' \) based on its own state and the slave response.

7.4.3 Cache Protocol Automata

We define state automata for the master and the slave case in order to implement the cache coherency protocol. In general the protocol is divided in 3 phases:

• master phase 1: \( CA, IM, BC \) are computed and put on the bus

• slave phase: slave responds by computing and sending \( CH, DI \), generating new slave state \( ss' \)

• master phase 2: master computes new state \( ps' \)

The state diagrams for the master and slave automata are presented on Figures 7.4.3 and 7.4.3.

Automata states

The overview on the states is given in the Table 7.2.

We define the following sets of automata states (Master, Slave, Local, Global, Warm, Hot):

\[
M = \{idle, localw, wait, flush, m0, m1, m2, m3, mdata, w\}
\]

\[
S = \{sidle, sidle', s1, s2, s3, sdata, sw\}
\]

\[
L = \{idle, localw\}
\]

\[
G = M \setminus L
\]

\[
W = G \setminus \{wait\}
\]

\[
H = W \setminus \{flush\}
\]
<table>
<thead>
<tr>
<th>#</th>
<th>master state</th>
<th>intended work</th>
<th>slave state</th>
<th>intended work</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>idle</td>
<td>read hits (unless colliding with global transaction on bus)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>localW</td>
<td>exclusive write hit (unless collision)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>wait</td>
<td>wait for arbiter to get bus access</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>flush</td>
<td>write back dirty line to mm</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>m0</td>
<td>put Ca, im, bc on b.mprot (can be pre-computed)</td>
<td>sidle</td>
<td>snooping on bus</td>
</tr>
<tr>
<td>5</td>
<td>m1</td>
<td>wait for slave response</td>
<td>s1</td>
<td>check for bhit, compute slave response</td>
</tr>
<tr>
<td>6</td>
<td>m2</td>
<td>wait for slave response</td>
<td>s2</td>
<td>transmit response on b.sprot</td>
</tr>
<tr>
<td>6'</td>
<td></td>
<td>sidle'</td>
<td></td>
<td>wait until CA is lowered on the bus</td>
</tr>
<tr>
<td>7</td>
<td>m3</td>
<td>analyse slave signals, prepare memory access</td>
<td>s3</td>
<td>do nothing</td>
</tr>
<tr>
<td>8</td>
<td>mdata</td>
<td>read data from bus in hot phase</td>
<td>sdata</td>
<td>transmit data on bus or read data from master or do nothing</td>
</tr>
<tr>
<td>9</td>
<td>w</td>
<td>write data, tag, s</td>
<td>sw</td>
<td>write data, s (if necessary)</td>
</tr>
</tbody>
</table>

Table 7.2: The overview on the automata states.
We denote by $z(i)$ and $zs(i)$ the state of a master or a slave automaton $i$ respectively.

For states $x \in M$ we mean by $x(i)^t$ the statement that master automaton $i$ is in state $x$ during cycle $t$. Similarly for $x \in S$ we mean by $x(i)^t$ the statement that slave automaton $i$ is in state $x$ during cycle $t$.

$$x(i)^t \equiv \begin{cases} \quad z(i)^t = x & : x \in M \\ zs(i)^t = x & : x \in S \end{cases}$$

We use the following notation for the set of states $A \in \{M, S, L, G, W, H\}$:

$$A(i)^t := z(i)^t \in A$$

$$A[i\rightarrow t'] := \forall q \in [t : t'] : A^q$$

For $X \in \{L, G, W, H\}$ we denote by $X(i)^t$ the fact, that master automaton $i$ is in phase $X$:

$$X(i)^t \equiv z(i)^t \in X$$

Statements without index $t$ are implicitly quantified for all cycles $t$. Statements without index $i$ are implicitly quantified for all states $i$ For transitions numbered with $(n)$ in an automaton we mean with $(n)(i)^t$ that the condition holds for the automata of cache $i$ in cycle $t$. 

---

**Figure 131: Master Automaton**
7.4. AUTOMATA TRANSITIONS AND CONTROL SIGNALS

Before we consider the transition and control signals of the master and slave automata we first introduce some auxiliary signals.

The signal \( phit \) indicates a processor hit, i.e., processor \( i \) issued a request on data which is present in the cache \( ca(i) \) and is not invalid. It is computed by means of the datapaths depicted in the Figure at page 224.

\[
phit := pa.tag = tag(pa.cha) \land \neg s(pa.cha).I
\]

A local computation is indicated by the signal \( local \). It is active if \( phit \) is active and if either a read or an exclusive write is performed. Formally, this looks like the following:

\[
local := phit \land (\neg pw \lor s(pa.cha) \in \{ E, M \})
\]

If a processor issues a request for a data, which is currently being processed in a global transaction, handling of the request locally is not possible. In this case \( snoopconflict \) is raised.

\[
snoopconflict := /sidle \land pa = badin.la
\]

Note, that snoop conflict is discovered one cycle after the address is actually on the bus (we have to clock data from the bus to \( badin \) register first).

With these prerequisites at hand, we now continue with the actual state transitions of the master automaton, starting with state \( idle \).

Now we consider transitions and control signals generated in every state of the automata.
State idle

In the idle state the signal \( \text{mbusy} \) is deactivated if either a local read is performed (which can be finished in one cycle) or there is no processor request at all. In case of a snoop conflict or a write we raise \( \text{mbusy} \):

\[
\text{/mbusy} = \text{/preq} \lor (\text{local} \land \text{/pw} \land \text{/snoopconflict})
\]

Note that \( \text{mbusy} \) is a Mealy signal and thus does not need to be changed in sync with the clock edges. In idle we also transmit the content of \( \text{ca}(i).\text{data}(\text{ca}(i).\text{pca}.\text{cla}) \) via \( \text{ca}(i).\text{pdout} \) back to the processor which waits for \( \text{/mbusy} \).

There are three possible transitions starting from state idle:

1. Transition (1): \( \text{idle} \rightarrow \text{idle} \)
   
   This transition is taken if there is a snoop conflict or if we have a read hit or there is no request from the processor to its cache at all:

   \[
   \text{/preq} \lor \text{snoopconflict} \lor (\text{phit} \land \text{/pw})
   \]

2. Transition (2): \( \text{idle} \rightarrow \text{localW} \)

   This transition is taken if there is a exclusive write hit and no global transaction currently accesses the respective data (snoop conflict):

   \[
   \text{/preq} \land \text{/snoopconflict} \land \text{phit} \land \text{pw} \land \text{ca}(i).s \in \{E,M\}
   \]

3. Transition (3): \( \text{idle} \rightarrow \text{wait} \)

   This transition is taken if the processor request is not local and no other master currently accesses the data:

   \[
   \text{preq} \land \text{/snoopconflict} \land \text{/localw}
   \]

   With the transition into state \( \text{wait} \) we activate the clock signals \( \text{ca}(i).\text{pdatace} \) in order to buffer the data input from the processor and \( \text{ca}(i).\text{reqset} \) to issue a request for the bus to the arbiter (cf. Arbiter, Section 7.4.5).

State localW

In state localW the master changes its state \( \text{ps} \) for the given cache line from \( E \) to \( M \), see Figure at page 225. The activated signals are \( \text{swA} \) (used in State-RAM, Figure at page 225) and \( \text{dataA} \) (used in Data-RAM, Figure at page 221). In state localW we lower \( \text{mbusy} \) and directly go back to idle in the next cycle.
7.4. Gate Level Design of a Shared Memory System

State wait

In state wait, the processor waits for his request to be granted by the bus arbiter. There are three transitions starting from state wait:

1. Transition (4): wait → m0
   If the request is granted and there is no cache line that needs to be evicted, we go to m0 directly:
   \[ grant(i) \land (\text{phit} \land s(pa.cla) \in \{S, 0\}) \lor (\text{phit} \land s(pa.cla) \in \{I, E, S\}) \]

2. Transition: wait → wait
   While /grant, the automaton stays in wait.

3. Transition (5): wait → flush
   When the request is granted, but the cache line is occupied and dirty, the automaton goes to state flush.
   \[ grant(i) \land /\text{phit} \land s(pa.cla) \in \{O, M\} \]

The following signals are set in this state:

- \( ca(i).bdoute := (5) \)
- \( ca(i).badoute := (5) \) or (4)
- \( ca(i).mmreqset := (5) \)
- \( ca(i).mmwset := (5) \)
- \( ca(i).bdoutoeset := (5) \)
- \( ca(i).badoutoeset := (5) \) or (4)
- \( ca(i).mmreqoeset := (5) \)
- \( ca(i).mmwoeset := (5) \)
- \( ca(i).mproteoute := (4) \)

Here, (5) represents the same predicate that triggers the transition to state flush. In case of (4) we have to load master data for transmission on the bus.

State flush

In state flush, we write the cache line that needs to be evicted to memory. The following signals are set:

- \( ca(i).bdouteclear := (6) \)
- \( ca(i).mmreqeclear := (6) \)
- \( ca(i).mmwoeclear := (6) \)
- \( ca(i).badoute := (6) \)
- \( ca(i).mprotoute := (6) \)
- \( ca(i).swA := (6) \)
When we leave the state flush we have to load master data for transmission on the bus. After the flush is done we write I to the state ram (this is not necessary for implementation, but makes the proofs much easier).

There are two transitions starting from state flush:

1. Transition: flush → flush
   While /b.mnack, we stay in flush since the memory is still busy.

2. Transition (6): flush → m0
   When the mnack signal gets active, the memory access finished and the automaton proceeds to state m0.

States m0 and s1

During m0 phase (1 cycle) master protocol data is transmitted on the bus.

The slave leaves the s1t state iff some master is in m0 phase:

1. Transition (7): s1t → s1
%(7) := b.mprot.CA

The following signals are raised in the slave:

ca(i).mprotinc := (7)
ca(i).badinc := (7)

States m1 and s1

During this states (1 cycle), the slave computes response signals. If it doesn’t have the requested data or its master automaton is starting to act, it goes to idle state, where it waits until CA signal is removed from the bus. Snoop-conflict starts to be visible in this phase.

1. Transition (8): s1 → idle'
   We move to (8) := /bhit ∨ grant[i]

   The following signal is raised in the slave:
   ca(i).sprotoutc := bhit

State idle'

The slave waits until CA is removed from the bus, then moves to idle.

States m2 and s2

During this states (1 cycle), the slave response signals are transmitted on the bus. The following signals is raised in the master:

ca(i).sprotinc
7.4. GATE LEVEL DESIGN OF A SHARED MEMORY SYSTEM

States $m3$ and $s3$

During this states (1 cycle), the master makes a decision whether to perform a memory access or not in the $mdata$ phase. This depends on whether $DI$ was active on the bus. In case of a write hit the master prepares the data for broadcasting. The following signals are raised in the master:

\[
ca(i).mmreqset := /ca(i).mprotout.BC \land /ca(i).sprotout.DI
\]
\[
ca(i).mmreqreset := /ca(i).mprotout.BC \land /ca(i).sprotout.DI
\]
\[
ca(i).bddataoutce := ca(i).mprotout.BC
\]
\[
ca(i).bddataoutceset := ca(i).mprotout.BC
\]

The following signals are raised in the slave (preparing the data intervention):

\[
ca(i).bddataoutce := /ca(i).sprotout.DI
\]
\[
ca(i).bddataoutceset := /ca(i).sprotout.DI
\]

States $mdata$ and $sdata$

During this phase the master reads the data from the bus (when it is ready). The data is either provided by the slave or is taken from the main memory. Leaving this state, the master clears control signals. The following signals are raised in the master:

\[
ca(i).bddatainc := /ca(i).mprotout.BC \land b.mmack \lor ca(i).sprotout.DI
\]
\[
ca(i).mmreqclear := /ca(i).mprotout.BC \land /ca(i).sprotout.DI \land b.mmack
\]
\[
ca(i).bdoutoeCLEAR := b.mmack
\]
\[
ca(i).bddataoutoeCLEAR := b.mmack
\]
\[
ca(i).mprotoutce
\]
\[
ca(i).mprotz
\]

The slave has to clock the broadcasted data:

\[
ca(i).bddatainc := ca(i).mprotin.BC
\]

States $w$ and $sw$

During this phase (1 cycle) the master and the slave write the results of the transaction into their registers (data, tag and state). The following signals are raised in the master:

\[
ca(i).datawA
\]
\[
ca(i).tagwA
\]
\[
ca(i).swA
\]
\[
ca(i).reجلclear
\]
\[
/\ca(i).mbusy
\]
The following signals are raised in the slave:

\[ ca(i).dataB := ca(i).mprotin.BC \land ca(i).sprotout.ch \]
\[ ca(i).sprotoutce \]
\[ ca(i).sprotz \]

### 7.4.5 Bus arbiter

We need bus arbitration for masters trying to get control of the bus (accessing the main memory or communicating with other caches). We could also do arbitration of slave DI signals when they all are in a Shared \((S)\) state. (Currently we assume that no DI signals are raised in this case and the master reads the main memory).

**Slave arbitration (optional)**

We choose the slave with the smallest index that has the DI signal raised. We collect all DI signals: \(d[i] = ca(i).sprotout.di\). The result of arbitration is defined by:

\[ GDI[i] = 1 \text{ if } i = \min\{j \mid d[i] = 1\} \]

This grant DI signal is then used to turn on exactly one driver: \(ca(i).bdoutoeset = s3 \land GDI[i]\) \(GDI\) is computed by the function \(f1\). \(f1\) is specified as: \(f1(X)[i] = 1 \Leftrightarrow \min\{j \mid X[i] = 1\}\) It finds the first 1 in a bitstring looking from the right. \(f1(X)\) is computed as follows:

1. Apply OR parallel-prefix on input \(X\):

\[
Y[i] = \begin{cases} 
X[0] & : i = 0 \\
X[i] \lor Y[i-1] & : i \neq 0 
\end{cases}
\]

2. Compute result bitstring \(Z[i]\) as follows:

\[
f1(X)[i] = Z[i] = \begin{cases} 
Y[0] & : i = 0 \\
Y[i] \land /Y[i-1] & : i \neq 0 
\end{cases}
\]

For GDI we then have \(GDI = f1(DI[2p - 1 : 0])\).

**Master arbitration**

In case of Master arbitration we have to ensure fairness. Fairness means that every request to access the bus (and thus to become master) is finally granted. The arbiter inputs the request \(req[i]\) from the caches (the caches raise them when they go to wait stage) and chooses exactly one cache that will get the permission to run on the bus. The winner is identified by the
7.4. Gate Level Design of a Shared Memory System

active $grant[i]$ signal. We say that the bus is busy if a node with a request is granted access:

$$bbusy = \bigvee_i (grant[i] \land req[i])$$

The implementation of a fair arbiter is presented on Figure 133. For the implementation of the nextgrant circuit we again use the circuit $f1$, s.t.

$$f1(X)[i] = 1 \leftrightarrow \min\{j \mid X[i] = 1\}$$

![Figure 133: Arbiter for masters](image)

The computation is done the following way:

1. Compute $X$ as OR parallel-prefix of $grant$
2. Compute conjunction $Y$: $Y = X \land req$
3. Apply function $f1$ constructed above to compute nextgrant:

$$nextgrant = \begin{cases} f1(Y) : \bigvee_i Y[i] \\ f1(req) : \text{otherwise} \end{cases}$$

We clock the bus every cycle when we have an active request:

$$grantce = \bigvee_i req[i]$$

Note, that if a cache $i$ gets a permission to run the bus, it will maintain this permission until it lowers its $req$ signal (it will always be the winner in the $f1$ circuit). A cache may get access to the bus in two consecutive memory accesses, however only if there are no waiting requests as we lower $req$ in the $w$ state. Thus, when we return to idle a nextgrant is computed and another cache may start its bus access in the next cycle.
Fairness of master arbiter

Statement \( \text{req}[i]^t \Rightarrow \exists t' \geq t : \text{grant}[i]^{t'} \)

Holds only under two conditions:

1. request stays stable:
   \[ \text{req}[i]^t \land \text{/grant}[i]^t \Rightarrow \text{req}[i]^{t+1} \]

2. granted request eventually taken away:
   \[ \text{grant}[i]^t \Rightarrow \exists t' \geq t : \text{/req}[i]^{t'} \]

First condition is true, since in state \( \text{mwait} \) signal \( \text{req}[i] \) stays active and we do not leave the state before \( \text{grant}[i] \) holds. Second condition holds due to system liveness: after \( \text{flush} \), \( \text{m0} \) and the other states of the hot phase, finally the state \( \text{idle} \) is reached in which \( \text{/req}[i] \) holds.

Proof: show that the distance between the index of the current master node and that of any any requesting node \( i \) is strictly monotonic decreasing with each arbitration.

- We define \( \text{one}(x) = \varepsilon\{i \mid X[i] = 1\} \), where \( \varepsilon \) denotes the Hilbert choice operator (for all sets \( A \), \( \varepsilon A = a \Rightarrow a \in A \)).

- \( \text{one}(\text{nextgrant}) = \begin{cases} 
\min\{j \geq \text{one}(\text{grant}) \mid \text{req}[j] = 1\} : & \text{if exists} \\
\min\{j \mid \text{req}[j]\} : & \text{otherwise}
\end{cases} \)

- We define the distance measure \( M \):

\[
M(i, t) = \begin{cases} 
i - \text{one}(\text{grant}^t) : & i \geq \text{one}(\text{grant}^t) \\
i - \text{one}(\text{grant}^t) + 2P : & \text{otherwise} \\
i - \text{one}(\text{grant}^t) \mod 2P & \end{cases}
\]

- Show by induction: \( M(i, t) \) is decreasing

7.5 Correctness Proof

We proceed in the following order:

1. we show properties of the bus arbitration guaranteeing that the warm phases of global transactions don't overlap

2. we show that slaves not involved in global accesses output ones to the open collector buses, i.e. they do not disturb signal transmission by other caches

3. we show that control automata run in sync during global accesses
4. this permits to show that tri state buses are properly controlled
5. we show that protocol data are exchanged in the intended way
6. this permits to show that data are exchanged in the intended way
7. aiming at a simulation between hardware and the atomic MOESI cache system we identify the accesses of the hardware computation.
8. we prove a technical lemma stating that accesses \( \text{acc}(i, k) \) of the atomic protocol only depend on cache lines with line address \( \text{acc}(i, k).a \) and only changes such cache lines.
9. we order hardware accesses by their end cycle and show that the hardware computation simulates the atomic computation with this order. In particular we establish in statement 5 of theorem (1 step simulation), that the hardware memory system is a sequentially consistent shared memory.

TODO: maybe we should define 'sequentially consistent shared memory' in a prominent place; like the still missing introduction

7.5.1 Arbitration

Lemma (grant unique):

\[
grant(i) \land grant(j) \implies i = j
\]

Proof. Proof trivial by construction of the arbiter. The output has form \( f1(x) \).

Lemma (grant stable) during an active request a grant is not taken away

\[
grant(i)^t \land req(i)^t \implies grant(i)^{t+1}
\]

Proof. Proof by construction of the arbiter.

Lemma (request at global) Automata in a global phase request access to the bus

\[
G(i) \implies req(i)
\]

Proof. By induction on \( t \). Trivially true for \( t = 0 \) because \( idle(i)^0 \) and thus \( \neg G(i)^0 \).

In the induction step we consider cycles \( t \) satisfying \( G(i)^t \) (because otherwise there is nothing to show) and argue here - and many times later - with a very typical case distinction.
• \( \neg G(i)^{t-1} \). By construction of the master automaton (A) we conclude
\[
\text{id}le(i)^{t-1} \land (3)(i)^{t-1} \land \text{wait}(i)^t \land \text{reqset}(i)^{t-1}
\]
By hardware construction (HW) of set/clear flipflops we conclude
\[
\text{req}(i)^t
\]
• \( G(i)^{t-1} \). Then by (A) \( \neg w(i)^{t-1} \) and hence \( \neg \text{reqclear}(i)^{t-1} \). Using the induction hypothesis (I) and hardware construction (HW) we conclude
\[
\text{req}(i)^t = \text{req}^{t-1} \quad (HW)
\]
\[
= 1 \quad (I)
\]
\[ \square \]

**Lemma (grant at warm):** a master can only be in the warm phase if he is granted access to the bus
\[
W(i) \implies \text{grant}(i)
\]

**Proof.** Nothing to show for \( t = 0 \). For the induction step consider \( t \) such that \( W(i)^t \).

• \( \neg W(i)^{t-1} \). By automata construction (A) we conclude
\[
\text{wait}(i)^{t-1} \land \neg \text{wait}(i)^t
\]
which by automata construction (A) implies \( \text{grant}(i)^{t-1} \) and by lemma (req at global) \( \text{req}(i)^{t-1} \). By lemma (grant stable) we conclude \( \text{grant}(i)^t \)

• \( W(i)^{t-1} \). By lemma (request at global) we get \( \text{req}(i)^{t-1} \). By induction hypothesis (I) we get \( \text{grant}(i-1) \) and with lemma (grant stable) we get \( \text{grant}(i)^t \).

\[ \square \]

Now we state the very crucial lemma.

**Lemma (warm unique):** only one processor at a time can be in a warm phase
\[
W(i) \land W(j) \implies i = j
\]

**Proof.** \( W(i) \land W(j) \) implies \( \text{grant}(i) \land \text{grant}(j) \) by lemma (grant at warm).
With lemma (grant unique) one concludes \( i = j \).

\[ \square \]
7.5. CORRECTNESS PROOF

7.5.2 Silent slaves on the open collector (OC) bus

**Lemma (silent slave):**
when a slave is not participating in the protocol, it puts slave response 00 on the control bus:

\[ zs(i) \in \{ \text{idle, sidle, s1} \} \implies sprotout(i) = 00 \]

**Proof.** Proof by induction on \( t \). Reset ensures \( \text{idle}^0 \). And activates signal \( sprotoutz \) which by hardware construction (IIW) clears the register. Thus we have \( sprotout(i)^0 = 00 \) and the lemma holds for \( t = 0 \).

Let \( t > 0 \) and \( zs(i)^t \in \{ \text{idle, sidle, s1} \} \implies mprotout(i) = 00 \). We consider two cases:

- **zs\( ^{t-1} \notin \{ \text{idle, sidle, s1} \} \):** then by automata construction (A)

\[ zs^{t-1} = sw^{t-1} \land sprotz^{t-1} \]

Thus the lemma holds by hardware construction.

- **zs\( ^{t-1} \in \{ \text{idle, sidle, s1} \} \):** Thus we have \( \neg(s1^{t-1} \land s2(i)^t) \). Therefore \( sprotout \) is not clocked \( (\neg sprotoutc^{t-1}) \) (IIW) and we get by induction hypothesis (I) and register semantics

\[
\begin{align*}
    sprotout(i)^t & = sprotout(i)^{t-1} \quad (HW) \\
                   & = 00 \quad (I)
\end{align*}
\]

In exactly the same way one shows the next lemma.

**Lemma (silent master)**

\[ \neg H(i) \implies mprotout(i) = 000 \]

7.5.3 Automata synchronization

This section contains two lemmas. We prove both of them simultaneously by induction on the number of cycles \( t \). Thus the statements of both lemmas in this section form together a single induction hypothesis.

**Lemma (idle slaves):** If no automaton is in a hot phase, then all slaves are idle.

\( \forall i : \neg H(i) \implies \forall j : \text{sidle}(j) \)

For all \( i \) we have after reset \( \text{idle}(i)^0 \notin H \) and \( \text{sidle}(i)^0 \). Thus the lemma holds initially. The induction step requires to argue about all states and can only be completed at the end of the section.
The next lemma explains how in a hot phase the master and the slave states are synchronized.

**Lemma (sync).** Consider a hot phase of master \( i \) lasting from cycles \( t \) to \( t' \), i.e. we have
\[
\neg H(i)^{t-1} \land H(i)^{t+1} \land \neg H(i)^{t'+1}
\]
then

1. for the master \( i \) we have
\[
m0(i)^t \land m1(i)^{t+1} \land m2(i)^{t+2} \land m3(i)^{t+3} \land mdata(i)^{t+4:t'-1} \land w(i)^{t'} \land idle(i)^{t'}
\]
2. for not affected slaves, i.e. for slaves \( j \) with \( \neg bhit(i)^{t+1} \lor j = i \) we have
\[
sidle(j)^t \land s1(j)^{t+1} \land sidle'(j)^{t+2:t'-1} \land sidle(j)^{t'}
\]
3. the affected slaves, i.e. the slaves \( j \) with \( bhit(i)^{t-1} \) run in sync with the master of the transaction
\[
sidle(j)^t \land s1(j)^{t+1} \land s2(j)^{t+2} \land s3(j)^{t+3} \land sdata(j)^{t+4:t'-1} \land sw^{t'} \land sidle^{t'+1}
\]

**Proof.** Part 1 follows directly from the construction of the master automaton (A).

For the proof of parts 2 and 3 recall that we are proving both lemmas together by induction on \( t^i \) thus for the proof of lemma (sync) we can assume lemma (idle slaves) at cycle \( t - 1 \).

We have \( H(i)^t \land \neg H(i)^{t-1} \) and conclude
\[
(wait(i)^{t-1} \lor flush(i)^{t-1}) \land grant(i)^{t-1}.
\]

From grant unique it follows \( \forall j \neq i : \neg grant(j)^{t-1} \) and \( \neg W(j)^{t-1} \) (grant at warm) and \( \neg H(j)^{t-1} \). Applying (idle slaves) as part of III we get \( \forall j : sidle(j)^{t-1} \). Using (silent masters) we then conclude for the cycles \( q \in [t - 1 : t' + 1] \):

\[
CA^q = \bigvee_j mprotout(j).Ca^q
\]
\[
= \begin{cases} 
0 : & q \in \{t - 1, t' + 1\} \quad \text{silent masters} \\
1 : & q \in [t : t']
\end{cases}
\]

Parts 2 and 3 follow now by construction of the slave automata and observing that the exit conditions for states \( mdata \) and \( sdata \) are identical.

For the induction step of the proof of (idle slaves) we consider a cycle \( t \) such that \( \forall i : \neg H(i)^t \). By lemma (unique warm master) this \( i \) is unique. We make the usual case distinction

\(?observe that for lemma (sync) \( t \) is the start time of the hot phase\)
7.5. CORRECTNESS PROOF

- \( \forall i : \neg H(i)^{t-1} \); by induction hypothesis (I) we have \( \forall j : \text{idle}(j)^{t-1} \).
  By lemma (silent masters) we have \( CA^{t-1} = 0 \) and the lemma follows by construction of the slave automata (A).

- \( \exists i : H(i)^{t-1} \); by lemma (warm unique) this \( i \) is unique. By construction of the master automaton (A) we conclude \( w(i)^{t-1} \). This is the end of a hot phase which started before \( t \). Therefore we can apply parts 2 and 3 of lemma (sync) as part of the induction hypothesis (I) to conclude \( \forall (j) : \text{idle}(j)^t \).

\( \square \)

Now we are able to argue about the uniqueness of the DI signal put on the bus by the slaves.

**Lemma (di unique):**

\[
SINV(t) \land d^t(i) \land d^t(j) \implies i = j
\]

**Proof.** Proof by induction \( t - 1 \implies t \).

- **Case** \( \neg d^{t-1} \). This implies \( s2(i)^t \) (from A). Applying lemma (sync) we get that all other slaves are either in \( s2 \) or are in \( \text{idle} \). If a slave \( j \) is in \( \text{idle} \), it doesn’t have active DI (from A and HW). If a slave is in \( s2(j)^t \), that means it was in \( s1(j)^{t-1} \) (from A). From \( SINV(t - 1) \) we can conclude that only one cache was in cycle \( t - 1 \) in O, E or M state. Since we know \( d^t(i) \) holds, then

\[
aca(i).s(ca(i).badin^{t-1})^{t-1} \in \{O, E, M\} \quad (A)
\]

From (HW) we also know that \( ca(j).badin^{t-1} = ca(i).badin^{t-1} \). From \( SINV(t - 1) \) it follows

\[
aca(j).s(ca(i).badin^{t-1})^{t-1} \notin \{O, E, M\} \quad (A)
\]

And thus

\[
C2(aca(j).s(ca(i).badin^{t-1})^{t-1}, aca(j).mprotin^{t-1}).di = 0 \quad (A)
\]

**Case** \( d^{t-1} \). Trivial using (IH) and lemma (sync).

\( \square \)
7.5.4 Control of tri state drivers

We are now ready to characterize for each register $X(i)$ connected via a tri state driver to a component $bY$ of the bus the cycles $t$ during which $X(i)$ is on the bus, i.e.

$$X(i)^t = bY^t$$

This is more subtle than one would expect. Obviously we have to identify the set of cycles

$$Cy(X, i) = \{ t : Xoe(i)^t \}$$

when $X(i)$ is put on the bus. For each of the signals $X$ concerned we will formulate a lemma (X) characterizing $Cy(X, i)$.

We also have to show the absence of bus contention. This involves a case distinction. The easy case deals with output enable signals $Xoe$ which are only set and cleared in the hot phases of master states, i.e. satisfying $Cy(X, i) \implies W(i)$. It will turn out that this is all signals except $bdataout$. We deal with bus contention for the latter signal at the end of the section.

For the easy case

**Lemma (no contention):**
Assume signal $X$ satisfies $Cy(X, i) \implies W(i)$ and $X! = bdataout$. Then

$$i \neq j \implies Cy(X, i) \cap Cy(X, j) = \emptyset$$

*Proof.* The proof by contradiction is trivial. For $i \leq j$ assume $t \in Cy(X, i) \cap Cy(X, j)$. By hypothesis we have $t \in W(i)^t \cap W(j)^t$. By lemma (warn unique) we conclude $i = j$. \qed

As the specification of accesses to main memory involves the detailed hardware model we have to show the absence of spikes for signals $X$ of the form $mmreq(i), mmw(i), bdataout(i)$ and $badout(i)$. This involves two statements. For showing the absence of spikes on the output of an enabled we observe that both $X$ and $Xoe$ are outputs of registers and it suffices to show

**Lemma (no spikes, enabled):**
Let $X \in \{ mmreq(i), mmw(i), bdataout(i), badout(i) \}$ and let $t, t+1$ be consecutive cycles in $Cy(X, i)$. Then in the first of these cycles registers $X$ and $Xoe$ are not clocked:

$$t \in Cy(X, i) \land t + 1 \in Cy(X, i) \implies \neg Xoe(i)^t \land \neg Xoe tếar(i)^t \land \neg Xoe set(i)^t$$

The only exception is the $badoutce(i)$, which might be clocked when $flush(i)^t \land m0(i)^{t+1}$ holds.

*Proof.* For each of the signals $X$ concerned this lemma follows directly from lemma $X$ characterizing $Cy(X, i)$ and the construction of the automata (A). \qed
7.5. **CORRECTNESS PROOF**

It turns out that we also have to show the absence of spikes on enabled drivers; in the detailed hardware model a spike on the output enable signal \(Xoe\) might propagate to the output of the corresponding driver and thus on the bus. In order to prevent this we also need

**Lemma (no spikes, disabled):**

Let \(X \in \{mmreq(i), mmw(i), bdataout(i), badout(i)\}\) and let \(t, t+1\) be consecutive cycles not in \(Cy(X, i)\). Then the output enable signal \(Xoe\) is not redundantly cleared

\[
t \notin Cy(X, i) \land t + 1 \notin Cy(X, i) \implies \neg Xoe\text{clear}(i)^t
\]

**Proof.** For each signal concerned, the proof follows again directly form lemma \((X)\) and automata construction \((A)\)

**mmw and mmrq**

The specification of the main memory unit requires that there are no spikes on \(b.req\) (when \(b.mmack\) is active) and no spikes on \(b.data, b.ad\) and \(b.mmw\) when \(b.req\) is on. The lemmas given in this section are used to derive these properties.

**Lemma (mmw):**

We write to memory in state \(flush\)

\[
t \in Cy(mmw, i) \leftrightarrow flush(i)^t
\]

**Proof.** We first show

\[
t \in Cy(mmw, i) \implies flush(i)^t.
\]

Consider any maximal interval \([t : t'] \subset Cy(mmw, i)\), i.e.

\[
\neg mmwoe(i)^{t-1} \land \forall q \in [t : t'] : mmoe(i)^q \land \neg mmoe(i)^{t+1}
\]

By hardware construction \((HW)\) we have \(mmwoeset(i)^{t-1}\). By automata construction \((A)\) we have

\[
wait(i)^{t-1} \land (5)(i)^t \land flush(i)^t
\]

For \(q \in [t : t']\) we show by induction \(flush(i)^q\): for \(q > t\) we have \(flush(i)^{t-1}\) by induction hypothesis \((I)\) and \(mmwoe(i)^q\), hence

\[
\neg b.mmack^{q-1} \land flush(i)^q
\]

by automata construction \((A)\).

Finally, again by automata construction \((A)\) we conclude from

\[
flush(i)^{t'} \land \neg mmwoe(i)^{t'+1}
\]
by automata construction (A)

\[ m_{\text{mwoeclear}}(i)^t \land m_0(i)^{t+1} \]

This shows that \( t \in C_y(\text{mmw}, i) \implies \text{flush}(i)^t \). The inverse direction

\[ \text{flush}(i)^t \implies t \in C_y(\text{mmw}, i) \]

follows by automata construction (A) with a trivial induction on \( t \). \( \square \)

The proofs of all other lemmas characterizing sets \( C_y(\mathbb{X}, i) \) follow very similar patterns and we therefore just formulate the lemmas without proof. For many of the following lemmas it will be convenient to define for each cycle \( t \) the state \( ez(t, i) \in M \) from which the master automaton entered the state \( z^t \) it holds

\[ ez(t, i) = \max\{t' : t' < t \land z(i)^{t'} \neq z(i)^t\} \]

**Lemma (mmreq):** We request a memory access when we flush or after a (write) miss of the master with no data intervention from any of the slaves

\( t \in C_y(\text{mmreq}, i) \iff \text{flush}(i)^\ell \lor \text{bdata}(i)^\ell \land \neg \text{phit}(i)^{ez(t, i)} \land \neg \text{mprotin}(i).D^t \)

**bdataout and badout**

**Lemma (badout):**
The bus address always comes from the master during the entire hot phase

\( t \in C_y(\text{badout}, i) \iff H(i)^t \)

Observe that the output enable signal \( \text{badoutoe} \) for this signal stays constantly 1 during an entire hot phase, the content of the address register \( \text{badout} \) changes after \( \text{flush} \). The last signal \( \text{bdataout} \) treated here can be activated both by masters and by slaves.

**Lemma (bdataout):**

Signal \( \text{bdataout}(i) \) is put on the bus by the master in state \( \text{bdata} \) after a (write) hit or by the slave in state \( \text{sdare} \) if it intervenes after a miss.

\[ t \in C_y(\text{bdataout}, i) \iff \text{bdata}(i)^t \land \text{mprotout}(i).bcez(t, i) \]
\[ \lor \text{sdata}(i)^t \land \text{sprotout}(i).diz(t, i) \]
\[ \lor \text{flush}(i)^t \]

We see that with the exception of \( X = \text{bdata} \) all signals satisfy the hypothesis of lemma no contention, thus we can summarize:

**Lemma (no contention 2):**

For \( x \neq \text{bdata} \) we have

\[ i \neq j \land X \neq \text{bdata} \implies C_y(X, i) \cap C_y(X, j) = \emptyset \]
7.5. CORRECTNESS PROOF

The corresponding result for $X = bdata$ happens to depend on certain data transmitted during the MOESI protocol. As these data are not transmitted via $bdata$, we can show the correct transmission of these data using the lemmas we already have.

7.5.5 Protocol Data Transmission

For states $z \in \{m0, m1, m2, m3\}$ we identify what data is processed and transmitted during state $z$. We refer here by $C1, C2, C3$ to the output signals of the corresponding circuits.

**Lemma (before $m0$):**
In the cycle before entering $m0$ registers $mprotout$ and $badout$ are loaded with the processor address and the output of circuit $C1$. Let $m0(i)^t \land \neg m0(i)^{t-1}$. Then

$$badout(i)^t = pa(i)^{t-1}$$
$$mprotout(i)^t = C1(soutA', pw)(i)^{t-1}$$

**Proof.** By automata construction (A) we have

$$wait(i)^{t-1} \land grant(i)^{t-1} \lor flush(i)^{t-1} \land b.mmac^t$$

The lemma now follows directly by automata (A) and hardware construction (HW). \qed

**Lemma ($m0$):**
During $m0$ register $mprotout(i)$ does not change and the protocol data and the bus address of the master are broadcasted. Let $m0(i)^t$. Then for all $j$:

$$mprotout(i)^{t+1} = mprotout(i)^t$$
$$mprotin(j)^{t+1} = mprotout(i)^t$$
$$bdain(j)^{t+1} = bdainout(i)^t$$
$$badin(j)^{t+1} = badout(i)^t$$

**Proof.** For $mprotout$ this follows directly from automata and hardware construction (A, HW). For the $mprotin$ we have

$$mprotin(j)^{t+1} = b.mprot^t \quad (HW, A)$$
$$= \bigvee_{k} mprotout(k)^t \quad (HW)$$
$$= mprotout(i)^t \quad (warm \ unique, \ silent \ masters)$$

For the bus address data $bad$ we have
badin(j)_{t+1} = b.ad^t (HW,A)  
= badout(i)^t (warm unique, silent masters, no contention 2, A, HW)

\[\]

Lemma (m1):
During m1 register mprotout(i) does not change. The affected slaves load their answer sprotout with the output of circuit C2.

Let m1(i)^t. Then for all j, s.t. s1(j)^t \land \neg 8(j)^t

\[\]
\begin{align*}
\text{mprotout}(i)_{t+1}^t & = \text{mprotout}(i)^t \\
\text{sprotout}(j)_{t+1}^t & = C2(soutB'(j), \text{mprotin}(j)^t)(j) \\
& = C2(ca(j).s(badin(j)^t)_{t+1}^t, \text{mprotin}(j)^t)(j)
\end{align*}

Proof. Proof analogous to lemma (before m0). One has to argue about the construction of the slave automata (A) and has to use lemma (sync). \[\]

Lemma (m2):
During m2 register mprotout(i) does not change. The protocol answer of the slaves is broadcast. Let m2^t. Then

\[\]
\begin{align*}
\text{mprotout}(i)_{t+1}^t & = \text{mprotout}(i)^t \\
\text{sprotin}(i)_{t+1}^t & = \bigvee_k \text{sprotout}(k)^t
\end{align*}

Proof. Proof analogous to lemma (m0). \[\]

With the above lemmas we can conclude a crucial lemma about the data intervention signals

Lemma (no DI after BC)
If the master signals a write hit during m(i)2 with mprotout.bc(i), then no slave signals intervention with sprotout.di(j). Let m2(i)^t. Then for all j:

\[\]
\[\]
\[\]

Proof. m2(i)^t implies m0(i)^{t-2} by automata construction (A). Thus we have:

\[\]
\[\]
\[\]
7.5. **CORRECTNESS PROOF**

From the protocol \( P \) and its correct implementation in circuit \( C2 \) we conclude for all slaves \( j \)

\[
sprotout(j).di^t = C2(soutB'(j), mprotin(j)).di = 0
\]

We show a series of lemmas \( (x, t) \) for cycles \( t \) having \( SINV(t - 1) \) as hypothesis, as well as

\[
SINV(t - 1) \land \bigwedge_x \text{lemma}(x, t) \implies SINV(t)
\]

The main induction hypothesis is then simply

\[
SINV(t) \land \bigwedge_x \text{lemma}(x, t)
\]

We are now able to show the absence of contention for \( bdataout \)

**Lemma (bdataout contention).**

Assume \( SINV(t - 1) \). Then there is no contention on the \( b.data \) bus until cycle \( t \).

\[
\forall q \leq t : \forall j \neq i : q \in C_y(bdataout, i) \implies q \notin C_y(bdataout, j)
\]

**Proof.** Proof by induction on \( t \). We omit the start of the induction. Assume \( t \in C_y(bdataout, i) \). We make the usual case distinction

- \( t - 1 \in C_y(bdataout, i) \). The easy case. By automata construction \( (A) \) we conclude \( flush(i)^t \), hence \( W(i)^t \land \neg H(i)^t \). Assume \( bdataoutoe(j)^t \) (i.e. \( t \in C_y(bdataout, j) \) for a different cache \( j \). By automata construction \( (A) \) we conclude \( flush(j)^t \lor sdata(j)^t \). By lemma (warm unique) \( flush(j)^t \) is impossible. By lemma (silent slaves) \( sdata(j)^t \) is impossible too

- \( t - 1 \notin C_y(bdataout, i) \). By automata construction \( (A) \) we conclude

\[
flush(i)^t \lor bdata(i)^t \lor sdata(i)^t.
\]

Again for \( j \neq i \) assume \( bdataoutoe(j)^t \). Thus

\[
flush(i)^t \lor bdata(i)^t \lor sdata(i)^t.
\]

As shown above case \( flush(i)^t \) and \( flush(j)^t \) holds at the same time is impossible. The case \( bdata(i)^t \land bdata(j)^t \) is impossible by lemma (warm unique). The case \( bdata(i)^t \) implies \( m2^{t-1} \) and \( mprotout(j).bc^{t-1} \) by automata construction \( (A) \). The case \( sdata(j)^t \) implies \( s2^{t-1} \) and \( sprotout.di(j) \). Thus \( bdata(i)^t \land sdata(j)^t \) and the case with reversed roles of indices \( i \) and \( j \) is excluded by lemma (no DI after BC). Finally, in the case \( sdata(i)^t \land sdata(j)^t \) two different data intervention signals are active which is impossible by lemma (di unique) and \( INV(t - 1) \).

\( \square \)
7.5.6 Data Transmission

Now that we know that tri state drivers are properly controlled it is very easy to state the effect of data transferred via the buses.

**Lemma (flush transfer):**

Assume $SINV(t - 1)$ and consider a maximal time interval $[s : t]$ when master $i$ is in state flush

$$\neg flush(i)^{s-1} \land \forall q \in [s : t]: flush(i)^q \land \neg flush(i)^{t+1}$$

Then $bdataout(i)^s$ is written to line $bdataout(i)^s$ of the main memory

$$mm(bdataout(i)^s)^{t+1} = bdataout(i)^s$$

**Proof.** By automata construction (A) and hardware construction (HW) we have for the start cycle $s$ of the time interval

$$\text{wait}(i)^{s-1} \land (5)(i)^{s-1} \land mmreq(i)^s \land mmw(i)^s$$

Let $x \in \{ mmreq, mmw, bdataout, bdata \}$. Then we have by lemma (x)

$$\forall q \in [s : t - 1]: x(i)^q = x(i)^{q+1}$$

By lemmas (no contention 2) and (bdataout contention) we get for the bus component $b.x$

$$\forall q \in [s, t]: b.x^q = x(i)^q$$

By lemma (no spikes enabled) and (no spikes disabled) we get the freedom of spikes of $b.x$ during the interval of real valued time $[s \ast \tau, t \ast \tau]$ where $\tau$ is the cycle time. The lemma follows now from the specification of main memory. \(\square\)

**Lemma (m0 transfer):**

Assume $SINV(t - 1)$. If the master $i$ is in state $m0$, then the address $bdataout(i)$ is broadcast to all caches $j$

$$m0^0(i)^t \Rightarrow \forall j: bdatain(j)^{t+1} = bdataout(i)^t$$

**Proof.** By automata construction (A) we have

$$bdataoutec(i)^t$$

By lemma (badout) we have

$$b.ad^t = bdataout(i)^t$$

By lemma (sync) we have

$$\forall j: s1(j)^t$$

By automata construction (A) and hardware construction (HW) we get

$$bdatain(j)^t = b.ad^t$$

and the lemma follows. \(\square\)
7.5. **Correctness Proof**

**Lemma (bdata write hit):**
Assume $SINV(t - 1)$. Let $mprotout(i).bc^{t-1} \land bdata(i)^t$. Then $bdataout(i)^t$ is broadcast to all slaves which are in state $sdata$

$$\forall j : sdata(j)^t \implies bdatain(j)^{t+1} = bdataout(i)^t$$

**Proof.** By lemma (bdataout) we have

$$bdataoutoe(i)^t$$

By lemma (bdataout contention) we conclude

$$b.data^t = bdataout(i)^t$$

By automata construction (A) and hardware construction (HW) we conclude

$$\forall j : sdata(j)^t \implies bdatain(j)^{t+1} = b.data^t$$

□

**Lemma (bdata data intervention):**
Assume $SINV(t - 1)$. Let $bdata(i)^t \land sprotout(j).di^{t-1}$. Then $ca(j).bdataout^t$ is transferred to the master

$$bdatain(i)^{t+1} = bdataout(j)^t$$

**Proof.** Proof along the lines of the previous two lemmas. This is the case where the state invariants are really needed in the proof of lemma (bdataout no contention) □

**Lemma (bdata miss no intervention):**
Assume $SINV(t - 1)$ and consider a maximal time interval $[s : t]$ when the master is in state $bdata$:

$$\neg bdata(i)^{s-1} \land bdata(i)^s \land \neg bdata(i)^{t+1}$$

Assume the absence of a write hit and of data intervention in cycle $s - 1$:

$$\neg mprotout(i).bc^{s-1} \land \neg sprotin(i).DI^{s-1}.$$  

Then line $mm^s(badout(i))^s$ is sent to the master

$$bdatain(i)^{t+1} = mm^s(badout(i)^s)$$

**Proof.** This lemma is proven along the lines of lemma (flush) transfer. □
7.5.7 Accesses of the Hardware Computation

Given the hardware computation, we construct a series of accesses \( acc(i, k) \). We start by defining the hardware cycle \( e(i, k) \) when the hardware access corresponding to \( acc(i, k) \) ends. A read or write access to cache \( i \) ends in cycle \( t \) when the processor request signal \( \text{preq}(i)^t \) is on and the busy signal \( \text{mbusy}(i)^t \) is off. A flush access ends in cycle \( t \) when the master leaves state flush, i.e. when \( \text{flush}(i)^t \land -\text{flush}(i)^{t+1} \).

\[
\text{someend}(i, t) \equiv \text{preq}(i)^t \land -\text{mbusy}(i)^t \lor \text{flush}(i)^t \land -\text{flush}(i)^{t+1}
\]

The definition of the end cycles \( e(i, k) \) for cache \( i \) is obviously

\[
e(i, k) = \begin{cases} 
\min \{ t : \text{someend}(i, t) \} & k = 0 \\
\min \{ t : t > e(i, k - 1) \land \text{someend}(i, t) \} & k > 0 
\end{cases}
\]

Note, that from (A) it follows that

\[
\text{idle}(i)^{e(i,j)} \lor \text{localw}(i)^{e(i,j)} \lor \text{flush}(i)^{e(i,j)} \lor \text{w}(i)^{e(i,j)}.
\]

The corresponding start cycles \( s(i, k) \) are defined in the following way: Read hits start and end in the same cycle. Local writes start 1 cycle before they end. Global reads or writes start in the cycle when their hot phase begins. Flushes begin when the master enters state flush.

Let \( t = e(i, k) \)

\[
s(i, k) = \begin{cases} 
t & \text{idle}(i)^t \\
t - 1 & \text{localw}(i)^{t-1} \\
\max \{ q : q \leq t \land \text{wait}(i)^{q-1} \} & \text{flush}(i)^t \\
\max \{ q : q < t \land \text{m0}(i)^q \} & \text{otherwise}
\end{cases}
\]

Note, that the access starts only when there is no snoop conflict on the bus. From (A) we conclude

\[
\text{idle}(i)^{s(i,j)} \lor \text{flush}(i)^{s(i,j)} \lor \text{m0}(i)^{s(i,j)}.
\]

One easily shows the following lemma

**Lemma (local order):**

\[
\forall k : s(i, k) \leq e(i, k) < s(i, k + 1)
\]

With the help of the end cycles \( e(i, k) \) alone we define the parameters of the \( acc(i, k) \) of the sequential computation. We start with flush accesses \( \text{flush}(i)^{e(i,k)} \). The address comes from \textit{badout} at the end of the access. The rest is obvious.
7.5. **CORRECTNESS PROOF**

\[
\begin{align*}
acc(i, k).a &= \text{badout}(i)^{s(i,k)} \\
acc(i, k).f &= 1 \\
acc(i, k).r &= acc(i, k)_w = 0
\end{align*}
\]

For all other accesses we construct \(acc(i, k)\) from the processor input at the end of the access \(t = e(i, k)\) (Note, that the processor inputs don’t change during the access):

\[
\begin{align*}
acc(i, k).a &= p_a(i)^t \\
acc(i, k).data &= pdatain(i)^t \\
acc(i, k).bw &= pbw(i)^t \\
acc(i, k).w &= pw(i)^t \\
acc(i, k).r &= \neg pw(i)^t \\
acc(i, k).f &= 0
\end{align*}
\]

For accesses \(acc(i, k)\) we also define the last cycle \(li(i, k)\) before or during (in case of read hit) the access, when master \(i\) was idle

\[
li(i, k) = \max \{ q : q \leq s(i, k) \land \text{idle}(i)^q \}
\]

In this cycle the master automaton makes crucial decisions for the entire access (either to go to a global transaction or local). We aim at lemmas stating that the outcome of these tests is stable during an access: if we would perform the based on the cache content later during the access we would get the same result.

We now show some crucial lemmas for these accesses:

**Lemma (global end cycle):**

Let \(acc(i, k)\) be a global read or write access and \(li(i, k)\) be the cycle when the master automata makes a decision to go global. Then it holds

\[
global(i, k, acc^{li(i,k)}) \implies w(i)^{s(i,k)}
\]

**Proof.** The proof is simple. We show \(w(i)^{s(i,k)}\) by contradiction. Let \(idle(i)^{s(i,k)} \lor localw(i)^{s(i,k)}\). Then \(idle(i)^{s(i,k)} \land idle(i)^{li(i,k)}\) which contradicts to \(global^{li(i,k)} \land \neg mbusy^{li(i,k)}\) by (A). From \(\neg acc(i, k).g\) we get \(\neg flush(i)^{s(i,k)}\).

In the very same way one show the lemma for the local accesses.

**Lemma (local end cycle):**

Let \(acc(i, k)\) be a local read or write access and \(t\) be the cycle when the master automata makes a decision to go local. Then it holds

\[
local(i, k, acc^{li(i,k)}) \implies acc(i, k).r \land idle(i)^{s(i,k)} \lor acc(i, k).w \land localw(i)^{s(i,k)}
\]
The lemmas global/local end cycle are implicitly applied to (almost) all the lemmas proven below.

**Lemma (stable master):**
Assume $acc(i, k).f \lor global(i, k, acc(i, k))$, i.e. $acc(i, k)$ is a flush or a global read or global write access. Then during the entire access abstract cache $i$ does not change:

$$\forall q \in [s(i, k) : e(i, k)] : aca(i)^q = aca(i)^{s(i, k)}$$

**Proof.** The master automaton activates write signals $xwA$, where $x \in \{data, tag, s\}$ only in cycle $q = e(i, k)$. These writes update the cache only after the end of the access. Any cycle $q$ under consideration belongs to a warm phase: $W(i)^q$. If access $acc(i, j)$ is a flush, then by lemmas (warm unique, idle slaves) all slaves are in state idle. If access $acc(i, k)$ is a global read or write, then by lemma (sync) we have for the slave of the master

$$sidle(i)^q \lor s1(i)^q \lor sidle’(i)^q$$

In none of these states the slave automaton activates a write signal $xwB$.  

The following lemma states that in the last cycle of $wait$ the RAMs of a waiting cache are not updated. Thus tests (4) and (5) would have the same outcome if performed one cycle later

**Lemma (last cycle of wait):**
Let $wait(i)^q \land \neg wait(i)^{q+1}$. Then the abstract cache does not change.

$$aca(i)^q = aca(i)^{q+1}$$

**Proof.** Let $x \in \{data, tag, s\}$. $xwA^q = 0$ is obvious because the master does not write in state $wait$. Also it obviously holds

$$req[i]^q \land \neg grant[i]^q \land req[i]^{q+1} \land grant[i]^{q+1}$$

Now we split cases

- Let $\forall j : \neg (grant[j]^q \land req[j]^q)$. Using (grant at warm) and (request at global) we get $\forall j : \neg W(j)^q$ and from (idle slaves) $\forall j : sidle(j)^q$ and conclude the lemma by (A).

- Let $\exists j : (grant[j]^q \land req[j]^q)$. We know, that $i \neq j$ and from (grant unique) $\neg grant[j]^{q+1}$ we get a contradiction by from the arbiter construction.

**Lemma (overlapping accesses with flush):**
Assume $SINV(t - 1)$. Let $acc(i, k)$ be a flush with address $a = acc(i, k).a$ and ending at cycle $e(i, k) = t - 1$. Let $acc(r, s)$ be a global read/write or
7.5. **CORRECTNESS PROOF**

a local write or a flush with address \(a\). Then the time intervals of the two accesses are disjoint. Thus only read hits with address \(a\) can overlap with flushes.

\[
SINV(t-1) \land (i, k) \neq (r, s) \land \text{acc}(i, k).f \land e(i, k) = t-1 \\
\land \ (\text{acc}(r, s).f \lor \text{global}(r, s, \text{acc}^u(r, s)) \lor \text{local}(r, s, \text{acc}^l(r, s)) \land \text{acc}(r, s).w \\
\land \ \text{acc}(r, s).a = \text{acc}(i, k).a \implies \ [s(i, k) : e(i, k)] \cap [s(r, s) : e(r, s)] = \emptyset
\]

**Proof.** Proof by contradiction. Assume

\[(r, s) \neq (i, k) \land q \in [s(i, k) : e(i, k)] \cap [s(r, s) : e(r, s)]\]

\(r = i\) is impossible by Lemma (local order). Hence \(r \neq i\). Cycle \(q\) is in the warm phase of access \(\text{acc}(i, k)\). Hence \(W(i)^q\) and by Lemma (warm unique)

\[W(i)^q \land \neg W(r)^q.\]

Thus access \(\text{acc}(r, s)\) is a local write at cache \(r\). By automata construction it started in cycle \(q \geq s(i, k)\) or cycle \(q-1 \geq s(i, k-1)\) and with a cache hit in an exclusive state

\[s(r, s) \in \{q-1, q\} \land \text{aca}(r).a^{s(r, s)} \in \{E, M\}\]

On the other hand flush accesses like \(\text{acc}(i, k)\) are preceded by a cache hit in state wait at the eviction address in cycle \(s(i, k) - 1:\)

\[\text{wait}(i)^{s(i, k)-1} \land \text{aca}(i).s(a)^{s(i, k)-1} \neq I\]

By Lemma (stable master) and (last cycle of wait) we get

\[\forall u \in [s(i, k) - 1 : e(i, k)] : aca(i).s(a)^u \neq I\]

For \(u = q\) or \(u = q - 1\) this contradicts the state invariants. \(\square\)

**Lemma (overlapping accesses with global):**

Let \(\text{acc}(i, k)\) be a global read or write with address \(a = \text{acc}(i, k).a\). Let \(\text{acc}(r, s)\) be a read or write with address \(a\) overlapping \(\text{acc}(i, k)\). Then \(\text{acc}(r, s)\) is a local write, and the overlap is in cycle \(\{s(i, k), s(i, k) + 1\}\) or \(\text{acc}(r, s)\) is a local read and the overlap is in cycle \(s(i, k)\)

\[
\text{global}(i, k, \text{acc}^u(i, k)) \land (i, k) \neq (r, s) \land \neg \text{acc}(r, s).f \\
\land \ \text{acc}(i, k).a = \text{acc}(r, s).a \land u \in [s(i, k) : e(i, k)] \cap [s(r, s) : e(r, s)] \\
\implies \text{acc}(r, s).w \land u \in \{s(i, k), s(i, k) + 1\} \\
\lor \text{acc}(r, s).r \land u = s(i, k) = s(r, s)
\]
Proof. Access acc(r, s) cannot by on the same master be lemma (local order) and it cannot be global by lemma (warm unique). Thus it is a local read or write. It cannot start later than s(i, k) because by lemma m0 in cycle s(i, k) + 1 slave r has already clocked in address a

$$badin(r)^{s(i,k)+1} = a$$

This gives a snoop conflict with access acc(r, s) and it cannot start. ☐

Lemma (stable local):
Let acc(i, k) be a local write. Then during the access abstract cache i does not change

$$local(i, k, acc^{s(i,k)}) \implies acc(i)^{s(i,k)} = acc(i)^{s(i,k)+1}$$

Proof. For x \in \{data, tag, s\} we have to show xwA^{s(i,k)} = xwB^{s(i,k)} = 0. This is trivial for signals xwA because they are not activated in state idle. Signals xwB^{s(i,k)} are active if slave automaton i is in state sw, i.e. sw(i)^{s(i,k)}.

In this case by lemma (sync) there is a master j \neq i and an access acc(j, v) such that u^{s(i,k)}. Thus the accesses overlap while the master is in state w. But this contradicts lemma (overlapping accesses with global read or write): the accesses can only overlap when the master is in state m0. ☐

Lemma (overlapping accesses with local write):
Assume SINV(t − 1). Let acc(i, k) be a local write with address a = acc(i, k).a and ending at cycle e(i, k) = t − 1. Let acc(r, s) be a local read or write with address a. Then it cannot overlap with acc(i, k).

$$SINV(t - 1) \land (i, k) \neq (r, s) \land local(i, k, acc^{s(i,k)}) \land acc(i, k).w$$
$$\land e(i, k) = t - 1 \land acc(i, k).a = acc(r, s).a$$
$$\land local(r, s, acc^{s(r,s)})$$
$$\implies [s(i, k) : e(i, k)] \cap [s(r, s) : e(r, s)] = \emptyset$$

Proof. Assume intervals overlap in cycle q. We have i \neq r by lemma (local order). By lemma (stable local) we have

$$aca(i).s(a)^{s(i,k)} = aca(i).s(a)^q$$
$$aca(r).s(a)^{s(r,s)} = aca(r).s(a)^q$$

By hypothesis we have

$$aca(i).s(a)^{s(i,k)} \in \{E, M\}$$
$$aca(r).s(a)^{s(r,s)} \neq I$$
Thus

\[
aca(i).s(a)^q \in \{E, M\} \\
aca(r).s(a)^q \neq I
\]

which contradicts the state invariants. \hfill \square

The last three lemmas can be summarized as follows. The only possible
overlaps between accesses to the same cache address \(a\) are (see figure x)

1. a flush with local reads

2. a global read or write with local reads and local writes. In this case
   the global read or write ends later.

3. a local read with other local reads

If we are interested in accesses to the same address \(a\) and ending at the
same cycle \(t\) we are left only with the first and third case. Formally let

\[
E(a, t) = \{(i, k) : e(i, k) = t \land acc(i, k),a = a\}
\]

be the set of accesses with address \(a\) ending in cycle \(t\). Then we have

**Lemma 94.** For any \(a\) and \(t\) either set \(E(a, t)\) contains a single element,
or all accesses in \(E(a, t)\) are read hits or one access in \(E(a, t)\) is a flush
and all other accesses are read hits.

\[
\#E(a, t) = 1 \lor \\
(\forall (i, k) \in E(a, t) : \text{acc}(i, k) = r \land local(i, k, aca^{l(i,k)})) \lor \\
(\exists (i, k) \in E(k, t) : (\text{acc}(i, k),f \land \forall (i', k') \in E(k, t) : \\
((i', k') \neq (i, k) \rightarrow \text{acc}(i, k) = r \land local(i, k, aca^{l(i,k)})))
\]

Let predicate \(P(i, k, a, t)\) be true if access \((i, k)\) ends at cycle \(t\), accesses
address \(a\) and is not a read hit:

\[
P(i, k, a, t) \equiv e(i, k) = t \land \text{acc}(i, k),a = a \land \neg (\text{acc}(i, k) = r \land local(aca^{l(i,k)}, \text{acc}(i, k), i))
\]

We conclude a carefully phrased technical lemma

**Lemma 95.** 1. Memory system slice \(a\) is only changed in cycle \(t\) if \(P(i, k, a, t)\)
   holds for some \((i, k)\)

\[
(-\exists (i, k) : P(i, k, a, t)) \rightarrow \Pi(ms(h^{t+1}, a) = \Pi(ms(h^t, a))
\]
2. If $P(i, k, a, t)$ does not hold, then in the atomic protocol access $acc(i, k)$ applied to port $i$ does not change slice $a$ of memory system $ms(h^t)$

$$\neg P(i, k, a, t) \rightarrow \Pi_1(\delta (h^t), acc(i, k), i, a) = \Pi_2(\delta (h^t), a)$$

3. At most one access ending in cycle $t$ can change slice $a$ both in the hardware computation and in the atomic protocol

$$\langle P(i, k, t, a) \wedge P(r, s, t, a) \rightarrow (i, k) = (r, s) \rangle$$

**Lemma (stable local decision):** Let $acc(i, k)$ be a local read or write, i.e. in cycle $Li(i, k) = s(i, k)$ we have $local(i, k, acc^{Li(i,k)})$. Then we could have reached the same decision in cycle $e(i, k)$.

$$local(acc^{Li(i,k)}, acc(i, k), i) \equiv local(acc^{e(i,k)}, acc(i, k), i)$$

**Proof.** Expand the definition of local and apply lemma (stable local). □

An important consequence is a reformulation of predicate $P(i, k, a, t)$

$$P(i, k, a, t) \equiv e(i, k) = t \wedge acc(i, k), a = a \wedge \neg (acc(i, k) = r \wedge local(acc^t, acc(i, k), i))$$

(7.1)

**Lemma (stable global decision):**

Let $acc(i, k)$ be a global read or write, i.e. in cycle $Li(i, k)$ we have $global(i, k, acc^{Li(i,k)})$. Then we could have reached the same decision in cycle $e(i, k)$.

$$global(i, k, acc^{Li(i,k)}) \implies global(i, k, acc^{e(i,k)})$$

**Proof.** Assume $global(i, k, acc^{Li(i,k)})$. We expand the definition of global and observe

$$global(i, k, acc) \equiv acc(i).s.a = I \vee acc(i, k).w \wedge acc(i).s.a \in \{S, O\}$$

We first consider cycles $q \in [li(i, k) + 1 : s(i, k) - 1]$ when the master is in state $wait$ or $flush$, i.e. $wait(i)^q \vee flush(i)^q$. If in any such cycle an access with slave $i$ and address $a$ ends (i.e. $sw(i)^q$) we can conclude from the protocol

$$acc(i).s(a)^q = I \implies acc(i).s(a)^{q+1} = I$$

$$acc(i).s(a)^q \in \{S, O\} \implies acc(i).s(a)^{q+1} \in \{I, S, O\}$$

Which permits to conclude

$$wait(i)^q \vee flush(i)^q \implies global(i, k, acc^{q+1})$$

Thus we have

$$global(i, k, acc^{e(i,k)})$$

By lemma (stable master) we have

$$acc(i)^{e(i,k)} = acc(i)^{e(i,k)}$$

and conclude the lemma. □
7.5. CORRECTNESS PROOF

7.5.8 Relating hardware computation with single steps of the atomic protocol

We are now ready to derive a crucial simulation result between the sequential computation of the atomic protocol and the hardware computation. Essentially it states that an access acc(i, k) of the hardware computation ending in cycle t has the same effect as the same access acc(i, k) applied to port i and memory system ms(h^t) of the atomic protocol.

**Lemma (1 step):**
Assume SINV(t). Then

1. \[
\Pi(ms(h^{t+1}), a) = \begin{cases} 
\Pi(\delta_1(ms(h^t), acc(i, k), i) : \exists(i, k) : P(i, k, a, t) \\
\Pi(ms(h^t), a) : \text{otherwise} 
\end{cases}
\]

2. let acc(i, k).r \wedge acc(i, k).a = a \wedge e(i, k) = t then
\[
pdataout(i)^t = pdataout1(ms(h^t), acc(i, k), i) = m(h^t)(a)
\]

**Proof.** The second statement is trivial for read hits. We will show the second statement for global reads together with the first statement.

By lemma simultaneously ending accesses 1 II(ms(h^t), a) only changes in cycles t + 1 following cycles t when a flush or write access or a global access with address a ends. Thus, for \(\neg \exists(i, k) : P(i, k, t, a)\) there is nothing left to show. Next we observe by Lemma simultaneously ending accesses, part 3, that in any cycle t there is at most one access acc(i, k) satisfying the conditions of the predicate P(i, k, t, a). Thus the statement of the lemma is well defined.

Now we split cases on the kind of access to address a ending in cycle t.

- if a local write access acc(i, k) ends in cycle s(i, k) + 1 = t then by lemma (stable local decision) we have local(acc^t, acc(i, k), i). By lemma (stable local) we have
\[
aca(i)_{e(i,k)} = aca(i)_{e(i,k)+1}
\]

Again by lemma (simultaneously ending non read hits) we find that no other write access to address acc(i, k).a ends in cycle t. By the construction of automata (A) and the data paths of the hardware (HW) we conclude
\[
\Pi(ms(h^{t+1}), a) = \Pi(\delta_1(ms(h^t), acc(i, k), i)
\]

- a global read or write access acc(i, k) ends in cycle e(i, k) = t. By lemma (stable global decision) we have
\[
\text{global}(aca^t, acc(i, k), i)
\]
Using lemmas (stable master, last cycle of wait) we get
\[ \forall q \in [s(i, k) - 1 : e(i, k)] : aca(i)^q = aca(i)^t \]

Using lemma (overlapping accesses with global) we get for all slaves \( j \)
between \( s(i, k) + 2 \) and \( e(i, k) \)
\[ aca(j)^{s(i,k)+2} = aca(j)^t \]

Using the data transfer lemmas this gives for all slaves \( j \)
\[
\begin{align*}
mprotin(j)^{s(i,k)+1} &= mpoutout(j)^{s(i,k)} \\
&= C1(code(ca(i), s^t), ca(i), pw^t) \\
spoutout(j)^{s(i,k)+2} &= C2(ca(j), s(acc(i, k), a)^{s(i,k)+2}, mpoutout(j)^{s(i,k)+1}) \\
&= C2(ca(j), s(acc(i, k), a)^t, mpoutout(j)^{s(i,k)+1}) \\
spoutin(j)^{s(i,k)+3} &= \bigvee_j spoutout(j)^{s(i,k)+2}
\end{align*}
\]

The statement
\[ \Pi(ms(h^{t-1}, a) = \Pi(\delta_1(ms(h^t), acc(i, k), i) \]

now follows from the data transfer lemmas. If \( acc(i, k) \) is a read access, the statement
\[ pdoutout(i)^t = pdoutout1(ms(h^t), acc(i, k), i) = m(h^t)(a) \]
also follows from the data transfer lemmas.

- if a flush \( acc(i, k) \) ends in cycle \( t \) then by lemma (overlapping accesses with flush) only local reads to address \( a \) can overlap with \( acc(i, k) \). Together with lemma (last cycle of wait) we conclude for all cache RAMs \( X \) of the master
\[ \forall q \in [s(i, k) - 1 : e(i, k)] : aca(i).X(a)^q = aca(i).X^t \]

and conclude the statement
\[ \Pi(ms(h^{t-1}, a) = \Pi(\delta_1(ms(h^t), acc(i, k), i) \]

with lemma (flush transfer) 
\[ \Box \]
7.5. Correctness Proof

7.5.9 Ordering Hardware Accesses Sequentially

We define the set of accesses $E(t)$ with end time $t$

$$E(t) = \{(i, k) : c(i, k) = t\}$$

Then $\#E(t)$ is the number of accesses ending at cycle $t$, and the number $NE(t)$ of accesses that have ended before cycle $t$ is defined by

$$NE(0) = 0$$
$$NE(t + 1) = NE(t) + \#E(t)$$

We number accesses $acc(i, j)$ according to their end time and accesses with the same end time arbitrarily. Thus accesses ending before $t$ get sequential numbers $[0 : NE(t) - 1]$ and accesses ending at $t$ get numbers $Q(t) = [NE(t) : NE(t + 1) - 1]$. Thus

$$\text{seq}(E(0)) = [0 : NE(0) - 1]$$
$$\text{seq}(E(t)) = [NE(t - 1) : NE(t) - 1]$$

If a flush access and one or more read hits to the same address end in cycle $t$, we order the flush access last.

$$\text{acc}(i, k).a = \text{acc}(i, k') = a \land (i, k), (i', k') \in E(t) \land \text{acc}(i, k) \rightarrow \text{seq}(i', k') < \text{seq}(i, k)$$

(7.2)

The resulting sequentialized access sequence $acc'$ is defined by

$$acc'[\text{seq}(i, k)] = acc(i, k)$$

For the accesses $acc(i, k)$ ending in cycle $t$ this gives

$$acc'[\text{seq}(E(t))] = acc(E(t))$$

The sequence $i'[NE(t) : NE(T + 1) - 1]$ of corresponding port indices is defined as follows. Index $i'(n)$ is the index $i$ such that $\text{seq}(i, k) = n$:

$$i'(n) = \{ i : \text{seq}(i, k) \}$$

We now can relate the hardware computation with the the computation of the atomic protocol and show that the state invariants hold for the hardware computation.

Lemma 96. 1. The first $NE(t)$ sequential accesses lead exactly to the same abstract memory system configuration $(aca, mm)$ as the first $t$ cycles of the hardware computation

$$ms(h^t) = \Delta_{\Delta}^{NE(t)} (ms(h^0), acc'[0 : NE(t) - 1], i'[0 : NE(t) - 1])$$
2. The state invariants hold until cycle $t$

$$SINV(t)$$

3. 

$$m(h^t) = \Delta_{m}^{NE(t)}(m(h^0), acc'[0 : NE(t) - 1])$$

Proof. By induction on $t$. For $t = 0$ the first statement is trivial. Also, for $t = 0$, i.e. after reset we have for all $a$ and $i$ 

$$aca(i)^0.s(a) = I$$

and we have $sinv(ms(h^0))$ by lemma 88.

For the induction step we assume that the lemma hold for $t$. For $x \in [0 : \#E(t + 1) - 1]$ we set 

$$n_x = NE(t) + x$$

. Then 

$$seq(E(t + 1)) = [NE(t) : NE(t + 1) - 1] = \{n_x : x \in [0 : \#E(t + 1) - 1]\}$$

For $x \in [0 : \#E(t + 1) - 1]$ we define the pair $(i_x, k_x)$ of indices by 

$$seq(i_x, k_x) = n_x$$

Then 

$$acc(i_x, k_x) = acc'(n_x) \quad \text{and} \quad i_x = i'(n_x)$$

We also define a sequence of memory system configurations $ms_x$ by 

$$ms_0 = ms(h^t)$$

$$ms_{x+1} = \delta_1(ms_x, acc'(n_x), i_x)$$

Using the induction hypothesis and lemma 89 we get 

$$ms_x = \Delta_{1}(ms_0, acc'[NE(t) : n_x], i'[NE(t) : n_x])$$

$$= \Delta_{1}(ms(h^t), acc'[NE(t) : n_x], i'[NE(t) : n_x])$$

$$= \Delta_{1}^{NE(t)+x}(ms(h^0), acc'[NE(t) : n_x], i'[NE(t) : n_x])$$

For $x = \#E(t)$ this gives 

$$ms_{\#E(t)} = \Delta_{1}^{NE(t+1)}(ms(h^0), acc'[NE(t) : NE(t+1) - 1], i'[NE(t) : NE(t+1) - 1])$$

By part 2 of the induction hypothesis the state invariants hold for $ms_x$: 

$$SINV(t) \rightarrow sinv(ms_0)$$

Using lemma 90 we conclude by induction that the state invariants hold for all memory systems $ms_x$ under consideration 

$$\forall x \leq \#E(t) : sinv(ms_x)$$

We proceed to characterize the slices $\Pi(ms_x, a)$ as a function of $a$ and $x$. We split cases
7.5. **CORRECTNESS PROOF**

- \( \forall x : \lnot P(i_x, k_x, a, t) \). Then by part 2 of lemma 95 slice \( a \) does not change
  \[ \Pi(ms_x, a) = \Pi(ms_0, a) \]
- \( \exists x : P(i_x, k_x, a, t) \). By part 3 of lemma 95 index \( x \) is unique.
  \[ \forall y \neq x : \lnot P(i_y, k_y, a, t) \]
By part 2 of lemma 95 no other access \( acc'(n_y) \) with smaller indices ending in cycle \( t \) changes slice \( a \) in the atomic protocol
  \[ \Pi(ms_x, a) = \Pi(ms_0, a) \]
By the construction of the sequential ordering, equation 7.2 no accesses \( acc'(n_y) \) with \( y < x \) accesses address \( a \):
  \[ y > x \rightarrow acc'(n_y).a \neq a \]
Using parts 2 and 3 of lemma 93 we conclude
  \[ \Pi(ms_{x+1}, a) = \Pi(\delta_1(ms_x, acc'(n_x), i_x), a) \]
  \[ = \Pi(\delta_1(ms_0, acc'(n_x), i_x), a) \]
  \[ = \Pi(ms_{\#E(t)}, a) \]
Using the definition of \( ms_0 \) this can be summarized in
  \[ \Pi(ms_{\#E(t)}, a) = \begin{cases} 
  \Pi(\delta_1(ms(h^t), acc'(n_x), i_x), a) & \exists x : P(i_x, k_x, a, t) \\
  \Pi(ms(h^t), a) & \text{otherwise} 
\end{cases} \]
Using the definition of \( acc'(n_x) \) and lemma (1 step) we conclude
  \[ \Pi(ms_{\#E(t)}, a) = \begin{cases} 
  \Pi(\delta_1(ms(h, acc(ni_x, k_x), i_x), a) & \exists x : P(i_x, k_x, a, t) \\
  \Pi(ms_0, a) & \text{otherwise} 
\end{cases} \]
  \[ = \Pi(ms(h^{t+1}), a) \]
Hence
  \[ ms_{\#E(t)} = ms(h^{t+1}) \]
This proves the first and second statement.
For the third statement we conclude with lemma 91
  \[ m(ms_{x+1}) = m(\delta_1(ms_x, acc'(n_x), i_x) \]
  \[ = \delta_M(m(ms_x), acc'(n_x)) \]
By induction on \( x \) we get
  \[ m(ms_x) = \Delta^x(m(ms_0), acc'[NE(t) : n_x] \]

and in particular
\[
m(h^{t+1}) = m(ms_{\#E(t)})
\]
\[
= \Delta_M^{#E(t)}(m(s_0), acc'[NE(t) : NE(t + 1) - 1])
\]
\[
= \Delta_M^{#E(t)}(h^t, acc'[NE(t) : NE(t + 1) - 1])
\]

With lemma 87 and part 3 of the induction hypothesis we conclude
\[
m(h^{t+1}) = \Delta_M^{# E(t)}(m(h^t), acc'[NE(t) : NE(t + 1) - 1])
\]
\[
= \Delta_M^{# E(t)}(\Delta_M^{NE(t)}(m(h^0), acc'[0 : NE(t) - 1]), acc'[NE(t) : NE(t + 1) - 1])
\]
\[
= \Delta_M^{NE(t+1)}(m(h^0), acc'[0 : NE(t) - 1]), acc'[0 : NE(t + 1) - 1])
\]

7.5.10 Sequential Consistency

With the notations of the proof of lemma 96 we now can show

**Lemma 97.** Let acc \((i_x, k_x)\) be a read access with address \(a\) ending in cycle \(t\), i.e., we have acc \((i_x, k_x).r\) and acc \((i_x, k_x).a = a\). Then the answer \(p_{dataout}(i_x)^t\) produced by the hardware at port \(i_x\) in cycle \(t\) is the content of the memory system produced by \(n_x\) steps of the atomic protocol at address \(a\).

\[p_{dataout}(i_x)^t = m(ms_x)(a)\]

**Proof.** We consider cases:

- \(#E(a, t) = 1\. No other access with address \(a\) ends at \(t\). Hence
  \[\forall y \neq x : \neg(P(i_y, k_y, a, t)\]

- \(#E(a, t) \geq 2\). By lemma 94 access acc \((i_x, k_x)\) is a local read access. By the ordering seq as specified in equation 7.2 we get
  \[\forall y < x : \neg(P(i_y, k_y, a, t)\]

As in the proof of lemma 96 we conclude

\[\Pi(ms_x, a) = \Pi(ms_x, a)\]

Using part 4 of lemma 93 and part 2 of lemma 91 we get

\[p_{dataout}(i_x)^t = p_{dataout1}(ms_0, acc(i_x, k_x), i_x)\] (lemma 1 step)
\[= p_{dataout1}(ms_x, acc(i_x, k_x), i_x)\]
\[= m(ms_x)(a)\]
7.5. **CORRECTNESS PROOF**

We finally can show

**Lemma 98.** The hardware memory is sequentially consistent

\[ pd_{\text{dataout}}(i_x)^t = \delta_M^n(m(h^0), acc')(acc(i_x, k_x).a) \]

**Proof.** Using lemma 97, lemma 92 and recalling the definition \( m(h) = m(ms(h)) \) of the hardware memory we get

\[
\begin{align*}
\text{pd}_{\text{dataout}}(i_x)^t & = m(ms_x)(acc(i_x, k_x).a) \\
& = m(\Delta^{ts}_1(m(ms(h^0)), acc', i')(acc(i_x, k_x).a)) \\
& = \Delta^n_M(m(h^0), acc')(acc(i_x, k_x).a)
\end{align*}
\]
Chapter 8

A Multicore Processor

8.1 Multi-core ISA

8.1.1 ISA specification

Recall that MIPS configurations \( c \) have components \( c.pc, c.dpc, c.gpr \) and \( c.m \). For the purpose of defining the programmers view of a multi core MIPS machine we collect the first three components of \( c \) into a processor configuration

\[
c.p = (c.pc, c.p.dpc, c.p.gpr)
\]

We denote by \( K_p \) the set of processor configurations. A MIPS configuration now consists of a processor configuration and memory configuration.

\[
c = (c.p, c.m)
\]

The next state function \( c' = \delta(c) \) is split into a next processor component \( \delta_p \) and a next memory component \( \delta_m \):

\[
c' = \delta(c) = (c'.p, c'.m) = (\delta_p(c.p, c.m), \delta_m(c.p, c.m))
\]

A multi-core configuration \( mc \) with \( p \) processors consists of the following components:

- \( mc.p : [0 : p - 1] \to K_p \). For processor numbers \( q \) the configuration of processor \( q \) in configuration \( mc \) is \( mc.p(q) \)

- \( mc.m \) is the memory shared by all processors

We introduce a step function \( s : \mathbb{N} \to [0 : P - 1] \), which maps step numbers \( n \) of the multi-core configuration to the ID \( s(n) \) of the processor

265
making a step in configuration $mc^n$. We require the step function to be fair in the sense that every processor $q$ is stepped infinitely often

$$\forall n, q \exists m > n : s(m) = q$$

Note that this function unknown to programmer; we will eventually construct it from the hardware. Programs thus have to perform well for all fair step functions.

Initially we require

$$mc^0 . p(q). pc = 4_{32}$$
$$mc^0 . p(q). dpc = 0_{32}$$

We now define the multi-core computation ($mc^n$) where $mc^n$ is the configuration before step $n$:

$$mc^{n+1} . p(x) = \begin{cases} 
\delta_p(mc^n . p(x), mc^n . m) & x = s(n) \\
mc^n . p(x) & x \neq s(n)
\end{cases}$$
$$mc^{n+1} . m = \delta_m(mc^n . p(s(n)), mc^n . m)$$

An equivalent definition is given in

**Lemma 99.**

$$(mc^{n+1} . p(s(n)), mc^{n+1} . m) = \delta(mc^n . p(s(n)), mc^n . m)$$
$$q \neq s(n) \rightarrow mc^{n+1} . p(q) = mc^n . p(q)$$

### 8.1.2 Sequential reference implementation

We define a sequential multicore reference 'implementation'. It is almost hardware and it could easily be turned into hardware, but we don’t bother. Recall that a hardware configuration $h$ of the sequential processor had components

$$h = (h . pc, h . dpc, h . gpr, h . im, h . dm)$$

The hardware construction of the sequential processor defines a hardware transition function

$$h' = \delta_H(h, reset)$$

We collect components $pc, dpc, gpr$ into a processor component

$$h . p = (h . pc, h . dpc, h . gpr)$$

and components $im, dm$ into a memory component

$$h . m = (h . im, h . dm)$$
Now we write the hardware transition function as

\[ h' = \delta_H(h.p, h.m, \text{reset}) \]

For the definition of the reference implementation we duplicate the processor component \( h.p \) of the hardware for every processor ID. Thus the new hardware has components \( h.m \) and \( h.p(q) \) for each processor ID \( q \). The computation \((h^n)\) of the reference implementation is simply defined by

\[ (h_n+1.p(s(n)^1), h_n+1.m) = \delta_H(h_1.p(s(n)), h^n.im, \text{reset}) \]

and

\[ h_{n+1}.p(q) = h^n.p(q) \quad \text{for} \quad q \neq s(n) \]

### 8.1.3 Simulation Relation

As in chapter 5 we assume alignment, disjoint code and data regions and hence no self modifying code. The basic sequential simulation relation \( sim(c, h) \) is extended to multicore machines by

\[
\begin{align*}
msim(mc, h) & \equiv m.c.m \sim_{CR} h.im \\
& \land m.c.m \sim_{DR} h.dm \\
& \land \forall q : (m.c.p(q).pc) = h.p(q).pc \\
& \land m.c.p(q).dpc = h.p(q).dpc \\
& \land m.c.p(q).gpr = h.p(q).gpr
\end{align*}
\]

The correctness of the reference implementation is asserted in

**Lemma 100.** There is an initial multicore ISA configuration \( mc^0 \) such that for

\[ \forall n : msim(mc^n, h^n) \]

**Proof.** This is a straight forward bookkeeping exercise. Assuming reset to be on in cycle \( n = -1 \) we set

\[ mc^0.p(q).gpr = h^0.p(q).gpr \]

\[ mc^0.m8(a000) = \begin{cases} h^0.im(a) & a \in \text{CR} \\ h^0.dm(a) & a \in \text{DR} \end{cases} \]

and obtain

\[ msim(mc^0, h^0) \]

For the induction step we conclude for processor \( s(n) \) from the induction hypothesis

\[ sim((mc^n.p(s(n)), mc^n.m), (h^n.p(s(n)), h^n.im, h^n.dm)) \]
With lemma 46, i.e. the correctness of the basic sequential hardware for one step, we conclude

\[ \text{sim}(mc^{n+1}.p(s(n)), mc^{n+1}.m), (h^{n+1}.p(s(n)), h^{n+1}.m)) \]

For processors \( q \neq (s(n)) \) that are not stepped we have by induction hypothesis

\[
\begin{align*}
mc^n.p(q).pc & = h^n.p(q).pc \\
mc^n.p(q).dpc & = h^n.p(q).dpc \\
mc^n.p(q).gpr & = h^n.q_p.gpr
\end{align*}
\]

By the definitions of multicore ISA and the reference implementation program counters, delayed program counters and general purpose register files do not change, so we have

\[ X \in \{pc, dpc, gpr\} \rightarrow mc^{n+1}.p(q).X = h^{n+1}(q).X \]

\[ \square \]

### 8.1.4 Local configurations and computations

For processor IDs \( q \) and local step numbers \( i \) we define the step numbers \( pseq(q, i) \) when local step \( i \) is executed on processor \( i \)

\[
\begin{align*}
pseq(q, 0) & = \min\{n : s(n) = q\} \\
pseq(q, i) & = \min\{n : n > pseq(q, i-1), s(n) = q\}
\end{align*}
\]

We also define a function \( ic(q, n) \) which counts how often processor \( q \) was stepped before step \( n \) resp. the number of instructions completed on processor \( q \) before step \( n \) by

\[
\begin{align*}
ic(q, 0) & = 0 \\
ic(q, n + 1) & = \begin{cases} 
ic(q, n) + 1 & s(n) = q \\ 
ic(q, n) & \text{otherwise} \end{cases}
\end{align*}
\]

An easy induction on \( n \) shows

**Lemma 101.**

\[ ic(q, n) = \#\{j : j < n, s(j) = q\} \]

A simple relation between functions \( pseq \) and \( ic \) is established in
Lemma 102.

\[ ic(q, n) = i \land s(n) = q \rightarrow \text{pseq}(q, i) = n \]

Proof. Let

\[ \{ j_0, \ldots j_{i-1} \} = \{ j : j < n, s(j) = q \} \]

and

\[ j_0 < \ldots < j_{i-1} \]

A trivial induction shows

\[ \forall x \leq i - 1 : j_x = \text{pseq}(q, x) \]

Because

\[ m \in [j_{i-1} : n - 1] \rightarrow s(m) \neq q \]

we conclude

\[ n = \min \{ m : m > \text{pseq}(q, i - 1), s(m) = q \} = \text{pseq}(q, i) \]

For processor IDs \( q \) and step numbers \( i \) we define the local hardware configurations \( h^{q,i} \) relevant for step \( i \) of processor \( q \) by

\[ h^{q,i} = (h^{\text{seq}(q,i)}.p(q), h^{\text{seq}(q,i)}.m) \]

Thus we start with multi processor hardware configuration \( h^{\text{pseq}(q,i)} \) when processor \( q \) makes step \( i \); then we construct single processor configuration \( h^{q,i} \) by taking the processor component of the processor, that is stepped, i.e. \( q \), and the memory component from the shared memory. We abbreviate

\[ p^{q,i} = h^{q,i}.p \]

The following lemma asserts for every \( q \), that as far as the processor components are concerned, the local configurations \( h^{q,i} \) behave as in an ordinary single single processor hardware computations; the shared memory of course can change between steps \( i \) and \( i+1 \) of the same processor.

Lemma 103.

\[
\begin{align*}
    p^{q,0} & = h^0.p(q) \\
    p^{q,i+1} & = \delta_p(p^{q,i}, h^{\text{pseq}(q,i)}.m)
\end{align*}
\]
Proof. By the definition of \( seq(q, 0) \) processor \( q \) is not stepped before step \( seq(q, 0) \)
\[
n < seq(q, 0) \rightarrow s(n) \neq q
\]
Thus processor configuration \( q \) is not changed in these steps and we get
\[
\begin{align*}
p^{0,0} &= h^{seq(q, 0)} \cdot p(q) \\
      &= h^0 \cdot p(q)
\end{align*}
\]
By the definition of \( pseq(q, i + 1) \) processor \( q \) is also not stepped between steps \( pseq(q, i) \) and \( pseq(q, i + 1) \)
\[
n \in [pseq(q, i) + 1 : pseq(q, i + 1) - 1] \rightarrow s(n) \neq q
\]
As above we conclude that processor configuration \( q \) does not change in these steps
\[
\begin{align*}
p^{q,i+1} &= h^{pseq(q,i+1)} \cdot p(q) \\
      &= h^{pseq(q,i)+1} \cdot p(q) \\
      &= \delta_H(h^{seq(q,i)} \cdot p(q), h^{pseq(q,i)} \cdot m) \\
      &= \delta_H(p^{q,i}, h^{pseq(q,i+1) - 1} \cdot m)
\end{align*}
\]

Next we show a technical result relating the local computations with the overall computation

**Lemma 104.**
\[
h^n \cdot p(q) = p^{q, ic(q, n)}
\]

**Proof.** by induction on \( n \). For \( n = 0 \) we have
\[
h^0 \cdot p(q) = p^{0,0} = p^{q, ic(q, 0)}
\]
For the induction step assume the lemma holds for \( n \). Let \( i = ic(q, n) \). We distinguish two cases

- \( q = s(n) \). Then \( ic(q, n+1) = i + 1 \)

By induction hypothesis and lemma 102 we get
\[
\begin{align*}
h^{n+1} \cdot p(q) &= \delta_H(h^n \cdot p(q), h^n \cdot m) \\
      &= \delta_H(p^{q,i}, h^{pseq(q,i)} \cdot m) \\
      &= p^{q,i+1} \\
      &= p^{q, ic(q, n+1)}
\end{align*}
\]
8.1. MULTI-CORE ISA

• \( q \neq s(n) \). Then
\[
\text{ic}(q, n + 1) = \text{ic}(q, n) = i
\]

and by induction hypothesis we get
\[
\begin{align*}
h^{n+1}.p(q) &= h^n.p(q) \\
&= p^{q,i} \\
&= p^{q,\text{ic}(q,n+1)}
\end{align*}
\]

\[\square\]

8.1.5 Accesses of the reference computation

For registers, memories or circuit signals \( X \) in processors of the reference machine, processor IDs \( q \) and instruction numbers \( i \) we abbreviate
\[
X^{q,i} = \begin{cases} 
&h^{q,i}.X \quad \text{if } X \text{ is a register or memory} \\
&X(h^{q,i}.p(q)) \quad \text{otherwise}
\end{cases}
\]

We define the instruction fetch access \( i - \text{acc}(q, i) \) in local step \( i \) of processor \( q \) as the access \( \text{acc} \) with
\[
\begin{align*}
\text{acc}.a &= \text{ima}^{q,i}.l \\
\text{acc}.r &= 1
\end{align*}
\]

We define the load store access \( ls - \text{acc}(q, i) \) in local step \( i \) of processor \( q \) as the access \( \text{acc} \) with
\[
\begin{align*}
\text{acc}.a &= \text{ea}^{q,i}.l \\
\text{acc}.r &\text{equiv } l^{q,i} \\
\text{acc}.w &= s^{q,i} \\
\text{acc}.data &= d\text{min}^{q,i} \\
\text{acc}.bw &= \begin{cases} b\text{w}^{q,i} & \text{sq }, i \\
&0^8 \quad \text{otherwise}
\end{cases}
\end{align*}
\]

In case instruction \( I^{q,i} \) is neither a load or a store all bits \( f, w \) and \( r \) of access \( le - \text{acc}(q, i) \) are of. We call such an access \textit{void}. A void access does not update memory and does not produce an answer.

Lemma 105. 1. for fetch accesses
\[
\text{imout}^{q,i} = \text{dataout}(h^{\text{seq}(q,i)}.im, i - \text{acc}(q, i))
\]

2. for loads
\[
l^{n,i} \rightarrow \text{dmout}^{q,i} = \text{dataout}(h^{\text{seq}(q,i)}.dm, ls - \text{acc}(q, i))
\]
3. \textit{updates of h.dm} \\
\[ h^{\text{seq}(q,i) + 1}.dm = \delta_M(h^{\text{seq}(q,i)}.dm, ls - \text{acc}(q,i)) \]

\textit{Proof.} by simple unfolding of definitions

\textbullet \hspace{1cm} imout^{q,i} = h^{\text{seq}(q,i)}.im(ima^{q,i}.l) \\
\hspace{1cm} = \text{dataout}(h^{\text{seq}(q,i)}.im, i - \text{acc}(q,i))

\textbullet \hspace{1cm} Let \( l(p^{q,i}, h^{\text{seq}(q,i)}.dm) \). Then \\
\[ dmout^{p,q,i} = h^{\text{seq}(q,i)}.dm(ea^{q,i}.l) \]
\hspace{1cm} = \text{dataout}(h^{\text{seq}(q,i)}.dm, ls - \text{acc}(q,i))

\textbullet \\
\[ h^{\text{seq}(q,i) + 1}.dm(b) = \begin{cases} \\
\text{modify}(h^{\text{seq}(q,i)}.dm(b), \text{dmin}^{q,i}, bw^{q,i}) & b = ea^{q,i}.l \land s(p^{q,i}, h^{\text{seq}(q,i)}) \\
h^{\text{seq}(q,i)}.dm(b) & \text{otherwise} \end{cases} \]
\hspace{1cm} = \delta_M(h^{\text{seq}(q,i)}.dm, ls - \text{acc}(q,i))

\[ \square \]

In the absence of self modifying code the code region of the memory does not change

\textbf{Lemma 106.} If \\
\[ \forall q, i : ls - \text{acc}(q,i).w \rightarrow ea^{q,i} \notin \text{CR} \]
then \\
\[ \forall n : h^n.im = h^0.im \]

\textit{Proof.} By induction on \( n \) and using that \( \text{pseq} \) is bijective. \[ \square \]

\section*{8.2 Shared Memory in the Multicore System}

\subsection*{8.2.1 Connecting Interfaces}

Every MIPS processor in the multi-core system has an instruction cache and a data cache. We connect the instruction cache to the MIPS processor in the following way:

\[ \begin{align*}
\text{ica.pa} & \leftarrow \text{ima} \pi \\
\text{ica.w} & \leftarrow 0 \\
\text{ica.req} & \leftarrow /\text{stall}_2 \\
\text{ica.pdataout} & \rightarrow \text{imout}_2 \\
\text{ica.mbusy} & \rightarrow \text{hazard}_1
\end{align*} \]
8.2. SHARED MEMORY IN THE MULTICORE SYSTEM

The data cache is connected in the following way:

\[ dca.pa \leftarrow ea.3 \]
\[ dca.w \leftarrow con.3.w \]
\[ dca.bw \leftarrow con.3.bw \]
\[ dca.pdin \leftarrow dmin_\pi \]
\[ dca.req \leftarrow full_3 \land (con.3.w \lor con.3.r) \]
\[ dca.Dout \rightarrow d\text{mout}_\pi \]
\[ dca.mbusy \rightarrow hazard_4 \]

Recall how the stall engine is defined as:

\[
\begin{align*}
stall_k &= \text{full}_{k-1} \land (\text{haz}_k \lor \text{stall}_{k+1}) \\
\text{ue}_k &= \text{full}_{k-1} \land \neg \text{stall}_k \\
\text{full}_{k+1} &= \text{ue}_k \lor \text{stall}_{k+1}
\end{align*}
\]

Thus, in stage 1 we start the memory access only when we don’t have a \( stall_2 \) signal coming from the stage below. This will turn out to be crucial for the liveness of the machine: we can write the data from the memory to the instruction register as soon as the access ends. In stage 4 the problem does not arise because stage 5 never is stalled.

8.2.2 Stability of inputs of accesses

Lemma 107. • for data caches \( 2q+1 \): if the request signal \( \text{mreq}(2q+1) \)
and the memory busy signal \( \text{mbusy}(2q+1) \) are both on, stage \( \text{reg}(3) \)
which contains the inputs of the access, is not updated

\[
\text{mreq}(2q + 1)^t \land \text{mbusy}(2q + 1)^t \rightarrow \text{ue}_3^{q,t} = 0
\]

• for instruction caches \( 2q \): if the request signal \( \text{mreq}(2q) \) and the mem-
ory busy signal \( \text{mbusy}(2q + 1) \) are both on, the inputs to the access
remain stable

\[
\text{mreq}(2q)^t \land \text{mbusy}(2q)^t \rightarrow \text{mreq}(2q)^{t+1} \land \text{ima}_\pi^{t+1} = \text{ima}_\pi^t
\]

Proof. For data caches we have

\[ hazard_4^t = \text{mbusy}(2q + 1)^t = 1 \]

and

\[ \text{mreq}(2q + 1)^t = 1 \rightarrow \text{full}_3^t \]
Hence

\[ \text{stall}_4^t = \text{full}_3^t \land \text{hazard}_4 = 1 \]

Thus

\[ \text{ue}_2^t = \text{full}_2^t \land \text{/stall}_4 = 0 \]

For instruction caches we have

\[
\begin{align*}
\text{ima}^{q,t}_2 &= \begin{cases} 
\text{pc}^{q,t}_2 & \text{full}^{q,t}_1 \\
\text{dpc}^{q,t}_2 & \text{}/\text{full}^{q,t}_1 
\end{cases} \\
\text{haz}^{q,t}_1 &= \text{mbusy}(2q)^t = 1 \\
\text{stall}^{q,t}_2 &= \text{full}_0 \land (\text{haz}^{q,t}_1 \lor \text{stall}^{q,t}_2) = 1 \\
\text{ue}^{q,t}_2 &= \text{full}_0 \land \text{}/\text{stall}^{q,t}_1 = 0 \\
\text{stall}^{q,t}_2 &= \text{full}^{q,t}_1 \land (\text{haz}^{q,t}_2 \lor \text{stall}^{q,t}_3) \\
&= \text{}/\text{mreq}(2q)^t = 0 \\
\text{full}^{q,t+1}_1 &= \text{ue}^{q,t}_2 \lor \text{stall}^{q,t}_2 = 0 \\
\text{stall}^{q,t+1}_2 &= \text{full}^{q,t+1}_1 \land (\text{haz}^{q,t+1}_2 \lor \text{stall}^{q,t+1}_3) = 0
\end{align*}
\]

We split cases on the value of \( \text{full}^{q,t}_1 \):

- if \( \text{full}^{q,t}_1 = 0 \) we have

  We conclude successively

  \[
  \begin{align*}
  \text{ue}^{q,t}_2 &= \text{full}^{q,t}_1 \land \text{stall}^{q,t}_2 = 0 \\
  \text{ima}^{q,t+1} &= \text{dpc}^{q,t+1} \\
  &= \text{dpc}^{q,t+1} \\
  &= \text{ima}^{q,t}
  \end{align*}
  \]

- if \( \text{full}^{q,t}_1 = 1 \) we have

  \[
  \begin{align*}
  \text{ue}^{q,t}_2 &= \text{full}^{q,t}_1 \land \text{stall}^{q,t}_2 = 1 \\
  \text{ima}^{q,t+1} &= \text{pc}^{q,t+1} \\
  &= \text{dpc}^{q,t} \\
  &= \text{ima}^{q,t}
  \end{align*}
  \]

\[\square\]

8.2.3 Relating updates enable signals and ends of accesses

By the definition of function \( \text{someend}(i, t) \), read or write accesses to cache \( i \) end in cycles \( t \) when \( \text{mreq}(i)^t \land \text{/busy}(i)^t \).
8.2. SHARED MEMORY IN THE MULTICORE SYSTEM

Lemma 108. 1. for data caches 2q + 1: if the update enable signal of stage 4 is activated and stage 3 contains a memory request, then a read or write access ends

\[ \text{ue}_4^{q,t} \land \text{mreq}(2q + 1)^t \rightarrow \exists k : e(2q + 1, k) \land /\text{acc}(2q + 1, k).f \]

2. for instruction caches 2q: if the update enable signal of stage 1 is activated, then a read or write access ends

\[ \text{ue}_1^{q,t} \rightarrow \exists k : e(2q, k) \land /\text{acc}(2q, k).f \]

Proof. For data caches we have by hypothesis

\[ \text{ue}_4^{q,t} = \text{full}_3^{q,t} \land /\text{stall}_4^{q,t} = 1 \]

Hence

\[ \text{full}_3^{q,t} = 1 \land /\text{stall}_4 = 1 \]

Also by hypothesis we have

\[ \text{mreq}(2q + 1)^t = \text{full}_3^{q,t} \land (\text{con}.3.w^{q,t} \lor \text{con}.3.r^{q,t}) = 1 \]

Hence

\[ (\text{con}.3.w^{q,t} = 1 \lor \text{con}.3.r^{q,t}) = 1 \]

Thus the update is due to a memory access and does not come from an instruction that does not use memory. We have

\[ \text{stall}_4^{q,t} = \text{full}_3^{q,t} \land (\text{haz}_4^{q,t} \lor \text{stall}_5) = 0 \]

Because \( \text{stall}_5 = 0 \) and \( \text{full}_3^{q,t} = 1 \) we conclude

\[ \text{haz}_4^{q,t} = \text{mbusy}(2q + 1)^t = 0 \]

Thus we have \( \text{someend}(2q + 1, t) \). The access ending cannot be a flush access, because by the construction of the control automata of the caches the \( \text{mbusy} \) signal stays active during flush accesses.

For instruction caches we have by hypothesis

\[ \text{ue}_1^{q,t} = \text{full}_0^{q,t} \land /\text{stall}_1^{q,t} = 1 \]

Hence

\[ \text{stall}_1^{q,t} = 0 \]

Because

\[ \text{stall}_1^{q,t} = \text{full}_0 \land (\text{haz}_1^{q,t} \lor \text{stall}_2^{q,t}) \]

and \( \text{full}_0 = 1 \) we conclude

\[ \text{haz}_1^{q,t} = \text{mbusy}(2q)^t = 0 \]

\[ \text{stall}_2^{q,t} = /\text{mreq}(2q)^t = 0 \]

Thus we have \( \text{someend}(2q, t) \). We argue as above that the access ending is not a flush access, and write accesses do not occur at instruction caches.

\[ \square \]
We come to a subtle point. When a read or write access ends, the corresponding stage is updated. This is crucial for the liveness of the system.

**Lemma 109.**

1. For data caches $2q + 1$:
   \[ (acc(2q + 1, k).r \lor acc(2q + 1, k).w) \land e(2q + 1, k) = t \rightarrow ue_4^{q,t} \]

2. For instruction caches $2q$:
   \[ (acc(2q + 1, k).r \lor acc(2q + 1, k).w) \land e(2q + 1, k) = t \rightarrow ue_4^{q,t} \]

**Proof.**

For the data cache we have by hypothesis
\[ mreq(2q + 1)^t = 1 \land mbusy(2q + 1)^t = 0 \]
Because
\[ mreq(2q + 1)^t = full^{q,t}_3 \land (con.3.r^{q,t} \lor con.3.w^{q,t}) \]
we conclude
\[ full^{q,t}_3 = 1 \]
Because
\[
stall^{q,t}_4 = full^{q,t}_3 \land (has^{q,t}_4 \lor stall_5) \\
= has^{q,t}_4 \\
= mbusy(2q + 1)^t \\
= 0
\]
We conclude
\[ ue_4^{q,t} = full^{q,t}_3 \land /stall^{q,t}_4 = 1 \]

For the instruction cache we do not have the counter part of equation $stall_5 = 0$, because stage 2 can be stalled. However we postponed raising the request signal until stage 2 is not stalled any more. By hypothesis we have
\[ mreq(2q)^t = 1 \land mbusy(2q)^t = 0 \]
Because
\[ mreq(2q)^t = /stall^{q,t}_2 \]
we conclude
\[
stall^{q,t}_2 = 0 \\
has^{q,t}_1 = mbusy(2q)^t \\
= 0 \\
stall^{q,t}_1 = full_0 \land (has^{q,t}_1 \lor stall^{q,t}_2) \\
= 0 \\
ue^{q,t}_1 = full_0 \land /stall^{q,t}_1 \\
= 1
\]
\[ \square \]
8.2. SHARED MEMORY IN THE MULTICORE SYSTEM

8.2.4 Scheduling function

The scheduling function for a processor $q$ of the pipelined multi-core system is defined analogous to the single-core processor. $I(q,k,t) = i$ means that instruction $i$ is in circuit stage $k$ of processor $q$ in cycle $t$. Note, that $i$ is the local index of the instruction.

$$I(q,k,0) = 0$$

$$I(q,1,t+1) = \begin{cases} I(q,1,t) + 1 & u_{q,t}^k \\ I(q,1,t) & \text{otherwise} \end{cases}$$

$$I(q,k,t+1) = \begin{cases} I(q,k-1,t) & u_{q,t}^k \\ I(q,k,t) & \text{otherwise} \end{cases}$$

8.2.5 Stepping function

In what follows we distinguish as before between the pipelined multi core machine $\pi$ and the sequential multi core reference implementations $\sigma$. For every hardware cycle $t$ of the pipelined machine $\pi$ we define the set $PS(t)$ of processors stepped at cycle $t$ by

$$PS(t) = \{ q : u_{q,t}^k = 1 \}$$

i.e. a processor $q$ of the reference implementation $\sigma$ is stepped whenever an instruction is clocked out of the memory stage of processor $q$ of the pipelined machine. The number $NS$ of processors stepped before cycle $t$ is defined as

$$NS(0) = 0$$

$$NS(t+1) = NS(t) + \#PS(t)$$

Thus in every cycle $t$ we step $\#PS(t)$ processors. For every $t$ we will define the values $s(m)$ of the step function $s$ for $m \in [NS(t) : NS(t+1) - 1]$ such that

$$s([NS(t) : NS(t+1) - 1]) = PS(t)$$

Any step function with this property would work, but we will later choose a particular function which makes the proof (slightly) easier. For any function with the above property the following easy lemma holds

**Lemma 110.** For every processor $q$ the scheduling function $I(q,4,t)$ of the pipelined machine at time $t$ counts the instructions completed $ic(q,t)$ on the sequential reference implementation

$$ic(q,NS(t)) = I(q,4,t)$$
Proof. By induction on $t$. For $t = 0$ both sides of the equation are 0. For the induction step we assume

$$ic(q, NS(t)) = I(q, 4, t)$$

and unfold definitions for $t + 1$:

$$ic(q, NS(t + 1)) = ic(q, NS(t)) + 1$$

$$\leftrightarrow q \in PS(t)$$

$$\leftrightarrow u_{e_4}^{q,t} = 1$$

$$\leftrightarrow I(q, 4, t + 1) = I(q, 4, t) + 1$$

For $y \in [0 : \#PS(t) - 1$ we define

$$m_y = NS(t) + y$$

$$q_y = s(m_y)$$

Then

$$z < y \rightarrow s(m_z) \neq q_y$$

and hence

$$ic(q_y, m_y) = ic(q_y, NS(t)) \cap s(m_y) = q_y$$

By lemma 102 we get

$$pseq(q_y, I(q_y, 4, t)) = pseq(q_y, ic(q_y, NS(t))) = m_y$$

(8.1)

Thus

$$pseq(q_0, I(q_0, 4, t)) = m_0 = NS(t)$$

and

$$pseq(q_{\#PS(t) - 1}, I(q_{\#PS(t) - 1}, 4, t)) = NS(t + 1) - 1$$

We define the linear ls-access sequence $ls - acc'$ by

$$ls - acc'(y) = ls - acc(q_y, I(q_y, 4, t))$$

and conclude with part 3 of lemma 105

$$h_{\sigma_{q_y}}^{m_y + 1}.dm = h_{\sigma_{q_y}}^{m_y + 1}.dm$$

$$= p_{seq(q_y, I(q_y, 4, t) + 1).dm}$$

$$= \delta_M(h_{\sigma_{q_y}}^{pseq(q_y, I(q_y, 4, t))}.dm, ls - acc(q_y, I(q_y, 4, t))$$

$$= \delta_M(h_{\sigma_{q_y}}^{m_y}.dm, ls - acc'(y))$$

With lemma 87 we get

Lemma 111.

$$h_{\sigma_{NS(t+1)}^{\#PS(t)}(h_{\sigma_{NS(t)}^{\#PS(t)}}.dm, ls - acc'[0 : \#PS(t) - 1])$$
8.2. SHARED MEMORY IN THE MULTICORE SYSTEM

8.2.6 Correctness

For the correctness result of the multi-core system we assume as before alignment and the absence of self modifying code. Recall that for $R \in \text{reg}(k)$ the single-core system simulation (correctness) theorem had the form

$$ R^t_{\pi} = \begin{cases} R^t_{\sigma} & \text{R visible} \\ R^{(k,t)-1}_{\sigma} & \text{R not visible} \land full_t \\ \end{cases} $$

For the multi-core machine we aim at a theorem of the same kind. We have, however, to couple it with an additional statement correlating the memory abstraction $m(h^t_{\pi})$ of the pipelined machine with the instruction memory $h_\sigma.im$ and the data memory $h_\sigma.dm$ of the sequential reference implementation. We will correlate the memory $m(h^t_{\pi})$ of the pipelined machine $\pi$ with the memories of the sequential machine $\sigma$ after $NS(t)$ sequential steps. Thus we aim for

$$ a \in CR \rightarrow m(h^t_{\pi})(a) = h_{\sigma}^{NS(t)}.im(a) $$
$$ a \in DR \rightarrow m(h^t_{\pi})(a) = h_{\sigma}^{NS(t)}.dm(a) $$

We abstract from $h_\sigma$ the memory system $m(h_\sigma)$ by

$$ m(h_\sigma) = \begin{cases} h_{\sigma}^{NS(t)}.im(a) & a \in CR \\ h_{\sigma}^{NS(t)}.dm(a) & a \in DR \\ \end{cases} $$

and reformulate this as

$$ a \in CR \cup DR \rightarrow m(h^t_{\pi})(a) = m(h_{\sigma}^{NS(t)})(a) $$

The main result of this book the asserts the simulation of the sequential multi-core reference implementation $\sigma$ by the pipelined multi-core machine $\pi$:

**Lemma 112.** For $a \in CR \cup DR$ there are initial values $h^0_{\sigma}.im(a)$ and $h^0_{\sigma}.dm(a)$ and for every $t$ there is a step function

$$ s : [0 : NS(t) - 1] \rightarrow [0 : p - 1] $$

such that

- For all stages $k$, registers $R \in \text{reg}(k)$ and all processor IDs $q$ let

  $$ I(q, k, t) = i $$

  then

  $$ R^t_{\pi} = \begin{cases} R^t_{\sigma,i} & \text{R visible} \\ R^{(k,t)-1}_{\sigma} & \text{R not visible} \land full_t \\ \end{cases} $$
\[ a \in CR \cup DR \rightarrow m(h^t_\pi)(a) = m(h^{NS(t)}_\pi)(a) \]

**Proof.** By induction on \( t \). For \( t = 0 \) all cache lines of \( \pi \) are invalid, thus the memory abstraction of \( \pi \) is defined by main memory:
\[ m(h^0_\pi(a) = h^0_\pi.mm(a) \]

For \( a \in CR \) and \( b \in DR \) we choose initial values of the memories of \( \sigma \) by
\[
 h^{NS(t)}_\sigma.im(a) = h^0_\pi.mm(a) \\
 h^{NS(t)}_\sigma.dm(b) = h^0_\pi.mm(b)
\]

A meaningful initial program can only be guaranteed if the initial code region \( CR \) is realized in the main memory as ROM (which it is in real machine).

Compared to the proof for a single pipelined processor the proof of the instruction step changes only for the fetch stage and the memory stage, i.e. for \( k = 1 \) and \( k = 4 \) in cycles \( t \) with \( u_{e_k^t} \), i.e. when the stage on processor \( q \) is clocked. We first consider stage \( k = 4 \) for processors \( q \) with \( u_{e_4^t} = 1 \) resp. with \( q \in PS(t) \). Then \( u_{e_4^t} = 1 \) which implies \( full_3^t \). Let \( i = \bar{I}(q, k, t) \) by lemmas 78 and 110 we get
\[
 I(q, 3, t) = I(q, 4, t) + 1 \\
 = i + 1 \\
 = ic(q, t) + 1
\]

All registers \( R \) of stage \( reg(3) \) are invisible. From the induction hypothesis we get
\[
 R^t_i, q, t = R^{i, I(q, 3, t) - 1}_3 \\
 = R^{i, t}_3 \\
 = R^{ic(q, t)}_3
\]

We split cases on the values \( mreq(2q + 1)^t \) of the memory request signal of the data cache of processor \( q \). Recall that is is defined as
\[
 mreq(2q + 1)^t = (con.3.l^{q,t} \lor con.3.s^{q,t}) \land full_3^t
\]

By definition of the stall engine we have
\[
 u_{e_4^t} \rightarrow full_3^t
\]

Thus
\[
 mreq(2q + 1)^t = con.3.l^{q,t} \lor con.3.s^{q,t}
\]
8.2. SHARED MEMORY IN THE MULTICORE SYSTEM

- if \( mreq(2q + 1)^t = 0 \) instruction \( I^R \) is neither a load nor a store. Then access \( ls - acc(q, i) \) is void. The instruction does not use the memory and the induction step is performed as in the single processor case.

- if \( mreq(2q + 1)^t = con.3.I^R \vee con.3.s^R = 1 \) we conclude with part 1 of lemma 108 that a read or write access \( acc(2q + 1, k(q)) \) ends in cycle \( t \). Because for the input registers \( R \) of the memory stage we showed

\[
R^t_{s, q} = R^t_{s, ic(q, t)}
\]

we can conclude

\[
acc(2q + 1, k(q)) = ls - acc(q, i)
\]

Setting \( q = q_v \) and using the definition of the sequential ls-access sequence \( ls - acc' \) we get

\[
acc(2q + 1, k(q_v)) = ls - acc(q_v, I(q_v, 4, t)) = ls - acc'(m_y)
\]

Now things are easy. By induction hypothesis we have

\[
a \in CR \cup DR \rightarrow m(h^l_n)(a) = m(h^{NS(t)}_n)(a)
\]

By the proof of lemma 96 we have for \( a \in CR \cup DR \)

\[
m(h^{t-1}_n)(a) = \Delta^t_M(m(h^l_n), acc'\{NE(t) : NE(t + 1) - 1\})(a)
\]

\[
= \Delta^t_M(m(h^{NS(t)}_n), acc'\{NE(t) : NE(t + 1) - 1\})(a)
\]

Let \( acc''[0 : u-1] \) be the subsequence of \( acc'[NE(t) : NE(t+1)-1] \) consisting exactly of the write accesses. Because reads and flushes don’t change the memory abstraction we get

\[
m(h^{t+1}_n)(a) = \Delta^t_M(m(h^{NS(t)}_n), acc'')(a)
\]

By lemma 111 we have

\[
h^{NS(t+1)}_n.dm = \Delta^t_M(h^{NS(t)}_n.dm, ls - acc''[0 : \#PS(t) - 1])
\]

Let \( ls - acc''[0 : u-1] \) be the subsequence of ls-access sequence \( ls - acc'[NS(t) : NS(t + 1) - 1] \) consisting only of the write accesses. Because reads and void accesses don’t change the memory abstraction we get

\[
h^{NS(t+1)}_n.dm = \Delta^t_M(h^{NS(t)}_n.dm, ls - acc'')
\]

Sequences \( acc'' \) and \( ls - acc'' \) consist of exactly the same accesses \( acc(2q + 1, k(q)) \) with \( (2q + 1, k(q)) \in E(t) \), possibly in a different order. By lemma
94 write accesses ending in the same cycle have different addresses. Thus the two access sequences have the same effect on memory. Thus for \( a \in DR \)
we have

\[
\Delta_M^t(m(h^{NS(t)}_\sigma, acc'^\nu))(a) = \Delta_M^t(h^{NS(t)}_\sigma.dm, ls - acc'^\nu)(a)
\]

which implies

\[
m(h^{t+1}_\sigma)(a) = h^{NS(t+1)}_\sigma.dm
\]

This shows the second statement for the data region.

Because for write accesses \( \sigma a^{q_y,I(q_y,4,t)} \in DR \) and \( DR \cap DR = \emptyset \) we know for write accesses

\[
acc(2q_y + 1, k(q_y)).a \notin CR
\]

which implies for \( a \in CR \)

\[
m(h^{t+1}_\sigma)(a) = m(h^t_\sigma)(a)
= m(h^{NS(t)}_\sigma)(a)
= m(h^{NS(t+1)}_\sigma)(a)
\]

This shows the second statement for the code region.

Next, for the data outputs of data caches of \( \pi \) we consider load accesses

\[
acc(2q_y + 1, k(q_y)) = ls - acc(q_y, I(q_y,4,t)) = ls - acc'(y)
\]

Let

\[
a = acc(2q_y + 1, k(q_y)).a \quad \text{and} \quad i = I(q_y,4,t)
\]

By lemma 94 read accesses and write accesses ending in the same cycle have different addresses. Hence

\[
m(h^{NS(t)}_\sigma)(a) = m(h^{NS(t)+\nu}_\sigma)(a) = m(h^{m\nu}_\sigma)(a)
\]

By lemma (1-step), equation 8.1 and part 2 of lemma 105 we get

\[
dataout_\pi(2q_y + 1)^t = m(h^t_\sigma)(a)
= m(h^{NS(t)}_\sigma)(a)
= m(h^{m\nu}_\sigma)(a)
= m(h^{pseq(q_y,i)}_\sigma)(a)
= dataout(m(h^{pseq(q_y,i)}_\sigma), ls - acc(q_y, i)
= dmout^{q_y,i}
\]

Finally, for the outputs of instruction caches 2q in stage \( k = 1 \) we consider processors \( q \) with \( we^{q,d}_1 = 1 \). Then a read access \( acc(2q, r(q)) \) ends in cycle \( \ell \), i.e. \( (2q, r(q)) \in E(\ell) \). Let

\[
a = acc(2q, r(q)) \quad \text{and} \quad i = I(q,1,\ell)
\]
8.2. SHARED MEMORY IN THE MULTICORE SYSTEM

By the argument for single pipelined processors we conclude

\[ ima^q_t = ima^q_{i} \]

Thus the access ending at the instruction cache of processor \( q \) in cycle \( t \) is fetch access \( i - acc(q, i) \)

\[ acc(2q, r(q)) = i - acc(q, i) \]

By lemma (1 step), lemma 106 and part 3 of lemma 105 we get

\[
\begin{align*}
pdataout_{p}(2q)^t & = m(h_{p})(a) \\
& = m(h_{p}^{NS(i)})(a) \\
& = m(h_{p}^{0})(a) \\
& = m(h_{p}^{0}(q,i))(a) \\
& = dataout(m(h_{p}^{0}(q,i)), i - acc(q, i) \\
& = imout_{p}^{q,i}
\end{align*}
\]

\[ \square \]