Computer Architecture 1
WS 2006/2007
Lecture Notes

Mark A. Hillebrand Wolfgang Paul

Date: 2006/12/13 10:10:56
Revision: 1.63

Chapter 1

Introduction

In this lecture we deal with the specification of processors, their implementation down to the gate-level, and the correctness of the implementation against its specification. In particular, we present a variant of the DLX RISC architecture [1], which has support for virtual memory and devices. As such, this architecture is a suitable basis for the implementation of commodity operating systems. We present a pipelined implementation of this architecture. It features common optimizations such as forwarding, caches, branch prediction, and translation look-aside buffers.

1.1 Overview

Lecture 1, 16 Oct 2006

Introduction
- Overview
  - Computer architecture
    - ... deals with specification, implementation, correctness, and evaluation of computer hardware
    - Specification
      - (Functional) description of hardware components
      - For example, for a processor: the instruction set architecture
      - Often very long; here, we make use of mathematics for precision and succinctness
    - Implementation
      - Provide enough detail to act as actual *building plan* for the hardware
      - Usually: digital gates, registers, and their interconnection
        - With a hardware description language (VHDL, Verilog)
        - As a netlist (which is a big DAG)
      - More effort for highly optimized designs:
        - Layout of individual wires and sizing and placement of individual gates / transistors (!)
    - Correctness
      - Verify that the implementation fulfills its specification
      - Non-exhaustive tests do not guarantee correctness

- Instead of tests, we do mathematical correctness proofs
- Example (simple processor implementation)
  - Prove a *simulation theorem*
  - A step of the architecture (usually: the execution of an instruction),
  - is simulated by the processor in 5 steps (hardware cycles!)

- Evaluation
  - Cost
    - Gate count? Silicon area? Power consumption?
  - Performance
    - Gate delay? Wire delay?
    - Relative to a given benchmark / known workload
  - Quality
    - Performance relative to cost
  - Note: two possible approaches to optimization in this context
    - Revise the implementation (and redo the correctness proof and the evaluation)
    - Revise the specification as well!
  - We will not do too much evaluation in the lecture, but we may do some in the exercises

- Literature
  - Main reference: S. M. Mueller and W. J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000
  - More references as we go along...

- Chapter Overview
  - Basics
    - Combinational circuit (without register)
      - For example, adders and multipliers
    - Clocked circuits (with registers)
      - For example, automata
  - DLX instruction set architecture
  - DLX implementation
    - Sequential (maybe)
    - Pipelined implementation
    - Improved pipelined implementation (forwarding, branch prediction)
  - Memory system with caches
    - Fast memory accesses
  - Virtual memory support
    - Essential for multitasking operation systems
    - MMU and TLB
  - I/O Device support
    - Essential for user interaction, networking, or data storage

- And then...? Potential Topics for CA 2
  - Out-of-order execution: Tomasulo scheduler
  - Fixed-point division
  - Unit with high latency, in particular:
  - Floating point
  - Automotive
  - More elaborate hardware models: circuit layout, wire delay, fan-out, power consumption

- Multiple control flows
- Superscalar processors
- Multithreading
- Multiprocessing
- Memory system with 64-bit access
- Devices with DMA and memory system with DMA support

Formalities

- Assistant: Peter Böhm, email `pbm@wjpserver.cs.uni-sb.de`
- Homepage: `http://www-wjp.cs.uni-sb.de/lehre/vorlesung/ws0607/`
  - Registration
  - Bibliography
  - Exercise sheets
  - Lecture script
- Exercise
  - one exercise sheet per week
  - group work allowed (max. group size?)
  - requirements: enough points, successful presentation of exercise solutions and not to many failures in presenting exercise solutions
- Exams: one mid-term, one end-term, sufficient 'success' in exercise groups required for participation
- Details: will be made available on the first exercise sheet and on the homepage
- Tutors: we might need more tutors, if anyone is interested, meet Peter and me

Chapter 2

Basics

Lecture 1, 16 Oct 2006 (cont.)

- Basics
  - Notation
    - Sets
      - Empty set: $\emptyset$
      - Natural numbers: $\mathbb{N} = \{0, 1, 2, \dots\}$ and $\mathbb{N}^+ = \{1, 2, \dots\}$
        - For $n \in \mathbb{N}^+$ we define $\mathbb{N}_{<n} = \{0, 1, \dots, n-1\}$
        - For $n \in \mathbb{N}$ we define $\mathbb{N}_n = \{0, 1, \dots, n\}$
      - Integers $\mathbb{Z} = \{\dots, -2, -1, 0, 1, 2, \dots\}$
      - Booleans: $\mathbb{B} = \{0, 1\}$
      - Operations
        - Union $A \cup B$, intersection $A \cap B$, power set 2^A (set of all subsets of A)
        - Cartesian product:
          - $A \times B$ (set of all pairs of elements in A and B)
          - $A_1 \times \cdots \times A_n$ (set of all n–-tuples of elements in A_i)
          - A^n (set of all n–-tuples of elements in A)
    - Records?
    - Functions $f : A \to B$
      - Predicates $P : A \to \mathbb{B}$ (any such predicate may also be identified with the subset $A' \subseteq A$ that is defined as $A' = \{a \in A \mid P(a) = 1\}$
      - Function update: 'f with $f(x) = y$' represents a new function f' which is equal to f at all locations $x' \neq x$ and returns y at location x.
    - Bits and bit vectors
      - Bit: $\mathbb{B} = \{0, 1\}$ (same as a Boolean)
      - Bit vector $a \in \mathbb{B}^n$ of length $n \in \mathbb{N}$
        - More verbosely denoted as $a[n-1:0] \in \mathbb{B}^n$
        - $\mathbb{B}^1$ is identified with $\mathbb{B}$
      - Switching function $f : \mathbb{B}^n \to \mathbb{B}^m$ (too strict)
        - The domain and range may be constructed from bits by building Cartesian products
      - Operators

Concatenation: $\circ : \mathbb{B}^n \times \mathbb{B}^m \to \mathbb{B}^{n+m}$ with $\circ(a[n-1:0], b[m-1:0]) = (a[n-1], \ldots, a[0], b[m-1], \ldots, b[0])$

Often, we will just omit $\circ$

Bit vector selection

For $a[n-1:0] \in \mathbb{B}^n$, $b \in \mathbb{N}$, and $c \in \mathbb{N}$ with $c > b$ define $a[c:b] = (a[c], a[c-1], \ldots, a[b])$

Unary: Negation: $\neg : \mathbb{B} \to \mathbb{B}$ with $\neg(0) = 1$ and $\neg(1) = 0$

Binary

Disjunction: $\vee : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ with $\vee(a, b) = 1$ iff $a = 1$ or $b = 1$

iff is 'if and only if'

Conjunction: $\wedge : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ with $\wedge(a, b) = 1$ iff $a = 1$ and $b = 1$

Inequality (exclusive-or) $\oplus : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ with $\oplus(a, b) = 1$ iff $a \neq b$

Equality (xnor) ...

These above are all commutative, here is a non-commutative binary example:

Implication: $\Rightarrow$: $\mathbb{B} \times \mathbb{B} \to \mathbb{B}$ with $\Rightarrow (a, b) = 1$ iff $a = 0$ or $b = 1$

Ternary

Choice / selection / if-then-else $ifte : \mathbb{B} \times \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ with $ifte(a, b, c) = b$ if $a = 1$ and $ifte(a, b, c) = c$ otherwise

Negation is usually written in prefix form, all other operators are written in infix form

Generalization for bitvectors

Negation: $\neg : \mathbb{B}^n \to \mathbb{B}^n$ with $\neg(0) = 1$ and $\neg(1) = 0$

Let $* \in \{\vee, \wedge, \oplus\}$

Then, define the following functions:

$* : \mathbb{B} \times \mathbb{B}^n \to \mathbb{B}^n$ with $*(a, b[n-1:0]) = (a * b[n-1], \ldots, a * b[0])$

$* : \mathbb{B}^n \times \mathbb{B} \to \mathbb{B}^n$ with $*(b[n-1:0], a) = *(a, b[n-1:0])$

$* : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{B}^n$ with $*(a[n-1:0], b[n-1:0]) = (a[n-1] * b[n-1], \ldots, a[0] * b[0])$

Similarly, generalize $ifte()$ for second and third operand

Number formats / encodings

Binary: $\langle a[n-1:0] \rangle = \sum_{i=0}^{n-1} a[i] \cdot 2^i$

Range: $\langle a[n-1:0] \rangle \in \{0, 1, \ldots, 2^n - 1\}$

Lemma: $\langle a[n-1:0] \rangle = \langle a[n-1:j] \rangle \cdot 2^j + \langle a[j-1:0] \rangle$ for all $j \in \mathbb{N}_{<n}$

Three-bit addition

a	b	c	c'	s
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

Definition

$s = a \oplus b \oplus c$

$c' = 1 \iff a + b + c \geq 2$

Property: $\langle c', s \rangle = a + b + c$

- Binary addition:
  - Inputs: $a[n-1:0]$, $b[n-1:0]$, and c_{in}
  - Outputs: inductively defined bit vectors $c[n-1:-1]$ and $s[n:0]$
    - $c[-1] = c_{\text{in}}$
    - $\langle c[i], s[i] \rangle = c[i-1] + a[i] + b[i]$ for $i \in \mathbb{N}_{<n}$
    - $s[n] = c[n-1]$
  - Property: $\langle a[n-1:0] \rangle + \langle b[n-1:0 \rangle + c_{\text{in}} = \langle c[n-1], s[n-1:0] \rangle$
    - This is the same as $\langle s[n:0] \rangle$
    - Sanity check: left-hand side is representable by an $(n+1)$-wide bit vector
  - Prove: by induction
    - Induction base $n = 1$: same as three-bit addition
    - Induction step $n \mapsto n+1$:
      - $\langle a[n:0] \rangle + \langle b[n:0] \rangle + c_{\text{in}}$
      - $= (a[n] + b[n]) \cdot 2^n + \langle a[n-1:0] \rangle + \langle b[n-1:0] \rangle + c_{\text{in}}$ (by aux. lemma)
      - $= (a[n] + b[n]) \cdot 2^n + \langle c[n-1], s[n-1:0] \rangle$ (by ind. hypothesis)
      - $= (a[n] + b[n] + c[n-1]) \cdot 2^n + \langle s[n-1:0] \rangle$ (by aux. lemma)
      - $= \langle c[n], s[n] \rangle \cdot 2^n + \langle s[n-1:0] \rangle$ (by def. of $c[i]$ and $s[i]$)
      - $= \langle c[n], s[n:0] \rangle$ (by aux. lemma)

Lecture 2, 18 Oct 2006

- Basics
  - Notation
    - Number formats / encodings
      - Repetition:
        - Binary value: $\langle a[n-1:0] \rangle = \sum_{i=0}^{n} a[i] \cdot 2^i \in \{0, 1, \ldots, 2^n - 1\}$
        - Lemma: $\langle a[n-1:0] \rangle = \langle a[n-1:j] \rangle \cdot 2^j + \langle a[j-1:0] \rangle$ for all $j \in \mathbb{N}_{<n}$
        - Three-bit addition...
        - Binary addition...
          - Inputs: $a[n-1:0]$, $b[n-1:0]$, and c_{in}
          - Outputs: inductively defined bit vectors $c[n-1:-1]$ and $s[n:0]$
          - Property: $\langle a[n-1:0] \rangle + \langle b[n-1:0 \rangle + c_{\text{in}} = \langle s[n:0] \rangle$
            - Carries $c[i]$ same 'auxiliary data' as used in school addition
        - Function add_n is defined as $add_n(a, b, c_{\text{in}}) = s$ with variables as above
      - Two's complement: $[a[n-1:0]] = -2^{n-1} \cdot a[n-1] + \langle a[n-2:0] \rangle$
        - Range: $[a[n-1:0]] \in T_n := \{-2^{n-1}, -2^{n-1}+1, \ldots, 2^{n-1}-1\}$
        - Sign bit: $[a] < 0$ iff $a[n-1] = 1$
        - Properties:
          1. $[0, a] = \langle a \rangle$
          2. $[a] \equiv \langle a[n-2:0] \rangle \mod 2^{n-1}$
          3. $[a] \equiv \langle a \rangle \mod 2^n$
          4. $[a[n-1], a] = [a]$ (sign extension)
          5. $-[a] = [\neg a] + 1$ (important)
        - TODO geometry of binary and two's complement number, picture?
        - Subtraction of binary numbers

Inputs: $a[n-1:0]$ and $b[n-1:0]$, assume $\langle a\rangle - \langle b\rangle \geq 0$
Output: compute $\langle a\rangle - \langle b\rangle$
Note: $\langle a\rangle - \langle b\rangle \geq 0$ implies $\langle a\rangle - \langle b\rangle \in \{0, 1, \ldots, 2^{n-1}\}$, representable by an n-bit number
Claim: $\langle a\rangle - \langle b\rangle \equiv \langle a\rangle + \langle \neg b\rangle + 1 \mod 2^n$
Proof:
$\langle a\rangle - \langle b\rangle$
$= \langle a\rangle - [0, b]$ (Prop. 1)
$= \langle a\rangle + [1, \neg b] + 1$ (Prop. 5)
$\equiv \langle a\rangle + \langle \neg b\rangle + 1 \mod 2^n$ (Prop. 2)

Addition of two's complement numbers
Inputs: $a[n-1:0]$, $b[n-1:0]$, and c_{in}
(so, for $b[n-1] = 1$ we have an effective subtraction)
Outputs: inductively defined bit vectors $c[n-1:-1]$ and $s[n:0]$ as in the binary addition algorithm
We then know already: $\langle a[n-1:0]\rangle + \langle b[n-1:0\rangle + c_{\text{in}} = \langle s[n:0]\rangle$
Claim:
1. $[a] + [b] + c_{\text{in}} \in T_n$ iff $c[n-1] = c[n-2]$
2. $[a] + [b] + c_{\text{in}} \in T_n$ implies
$[a[n-1:0]] + [b[n-1:0] + c_{\text{in}} = [s[n-1:0]]$
Proof:
$[a[n-1:0]] + [b[n-1:0] + c_{\text{in}}$
$= 2^{n-1} \cdot (-a[n-1] - b[n-1]) + \langle a[n-1:0]\rangle + \langle b[n-1:0\rangle + c_{\text{in}}$ (def. TC)
$= -2^{n-1} \cdot (a[n-1] + b[n-1]) + \langle c[n-2], s[n-2:0]\rangle$ (add alg.)
$= -2^{n-1} \cdot (a[n-1] + b[n-1] + c[n-2] - 2 \cdot c[n-2]) + \langle s[n-2:0]\rangle$ (prop. add alg.)
$= -2^{n-1} \cdot (\langle c[n-1], s[n-1]\rangle - 2 \cdot c[n-2]) + \langle s[n-2:0]\rangle$ (def. add alg.)
$= -2^n \cdot (c[n-1] - c[n-2]) + [s[n-1:0]]$ (def. add alg.)
$= 2^n \cdot (-c[n-1] + c[n-2]) + [s[n-1:0]]$ (if you want)
Now: by case distinction get claim 1; then again by claim 1 and substitution get claim 2
Note: adding $\langle a[n-1], a\rangle + \langle b[n-1], b\rangle + c_{\text{in}}$ with an $(n+1)$-bit result always 'works'

Unary
Define $unary(a[n-1:0]) = 1$ iff $a[n-1:0] = (0^{n-i-1}, 1, 0^i)$ for some $i \in \{0, \ldots, n-1\}$
For $a[n-1:0]$ with $unary(a)$, define $\langle a\rangle_{\text{u}} = i$ iff $a[i] = 1$ (uniquely defined, same i as above)
Range: $\langle a[n-1:0]\rangle_{\text{u}} \in \{0, \ldots, n-1\}$

Half-Unary
Define $halfunary(a[n-1:0]) = 1$ iff $a[n-1:0] = (0^{n-i}, 1^i)$ for some $i \in \{0, \ldots, n-1\}$
For $a[n-1:0]$ with $halfunary(a)$, define $\langle a\rangle_{\text{hu}} = \min(\{n\} \cup \{i \mid a[i] = 0\})$
or $\langle a\rangle_{\text{hu}} = i$ for $a[n-1:0] = (0^{n-i}, 1^i)$
Range: $\langle a[n-1:0]\rangle_{\text{hu}} \in \{0, \ldots, n\}$

Combinational circuits
Hardware model (Part 1)

For now: without storage elements (register, flip-flops, RAM)
A combinational circuit is a directed acyclic graph (DAG) of gates (and their connections)
Inputs and output correspond (roughly) to sources and sinks of the graph
Cost and delay based on gates

	Cost	Delay
INV	1	1
NAND, NOR	2	1
AND, OR	2	2
XOR, XNOR	4	2
MUX	3	2

([2, Table 2.1] is wrong for AND and OR)
Cost of a circuit: sum of all gate costs
Delay of a path: sum of gate delays on the path
Delay from input to output: maximum of delay of all path from input to output (or 0)
Delay of a circuit: maximum over all delay from input to output (or 0)
Trivial Constructions [2, Section 2.3.1]
Testing for Zero or Equality [2, Section 2.3.2]
Decoders [2, Section 2.3.3]
Only specify half-decoders and decoders
SKIP leading zero counter [2, Section 2.3.4]
Arithmetic circuits [2, Section 2.4]
Half-adder: adds two bits
Full-adder: adds three bits
Adder specification
$add_n : \mathbb{B}^n \times \mathbb{B}^n \times \mathbb{B} \to \mathbb{B}^{n+1}$ with
$add_n(a[n-1:0], b[n-1:0], c_{in}) = s[n:0]$ such that $\langle s \rangle = \langle a \rangle + \langle b \rangle + c_{in}$
Carry-chain adder (a.k.a. ripple-carry adder)
SKIP proofs, maybe also skip implementation
SKIP: Carry-chain incrementer
Conditional sum adder [2, Section 2.4.2]
SKIP conditional sum incrementer
TODO in the exercises we should have at least one incrementer
Compound adder
TODO missing circuits?

Lecture 3, 23 Oct 2006

Basics
Combinational circuits
Note: delay of AND and OR is 2, not 1
Arithmetic circuits [2, Section 2.4]
Parallel-prefix computation

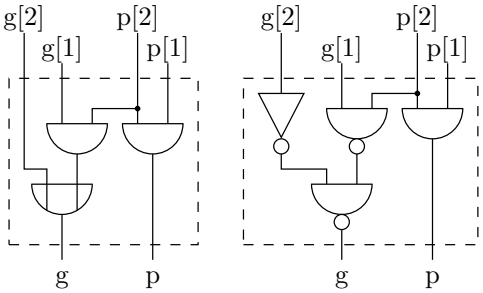
Let $\circ : M \times M \to M$ be associative on some $M = \{0,1\}^m$
Define $PP_\circ(n) = M^n \to M^n$ by $PP_\circ(n)(x_1, \ldots, x_n) = (y_1, \ldots, y_n)$ with $y_i = x_1 \circ \cdots \circ x_i$
Assumption: we have a circuit that computes $\circ$
TODO Circuit implementation ($n = 1$, n even, and n odd)
Claim: circuit correct, i.e., it computes prefixes
Proof by induction.
Here: let's only look at the case $n > 0$ and n even
Let X'_i be inputs to $PP_\circ(n/2)$ and let Y'_i be the outputs of $PP_\circ)(n/2)$ (for $i \in \{0, \ldots, n/2 - 1\}$)
$X'_i = X_{2 \cdot i+1} \circ X_{2 \cdot i}$ (if you will: correctness of $\circ$-circuit)
We have:
$Y'_i = X'_i \circ \cdots \circ X'_0$ (by induction hypothesis)
$= X_{2 \cdot i+1} \circ \cdots \circ X_0$
$= Y_{2 \cdot i+1}$ (correctness for all odd outputs)
Additionally:
$Y_{2 \cdot i} = X_{2 \cdot i} \circ Y_{2 \cdot i-1}$
$= X_{2 \cdot i} \circ \cdots \circ X_0$ (easy)
Cost and delay
Let $C_\circ$ and $D_\circ$ the cost and delay of the $\circ$-circuit
Then
$C_{PP_\circ}(0) = 0$ and $D_{PP_\circ}(0)$
$C_{PP_\circ}(n) = C_{PP_\circ}(\lfloor n/2 \rfloor) + (n-1) \cdot C_\circ$
($\rightsquigarrow$ linear in n)
$D_{PP_\circ}(n) \leq D_{PP_\circ}(\lfloor n/2 \rfloor) + 2 \cdot D_\circ$
($\rightsquigarrow$ logarithmic in n)

Carry-lookahead adder
An application of the parallel-prefix computation
Inputs: $a[n-1:0]$, $b[n-1:0]$, and c_{in}
Define two sets of signals, which we intend compute with a parallel prefix computation:
Let i, $j \in \{0, \ldots, n-1\}$ with $i \leq j$
$p_{i,j}(a,b) = 1$ iff $\langle a[j:i] \rangle + \langle b[j:i] \rangle = \langle 1^{j-i+1} \rangle$
A carry in from position $i-1$ *propagates* to position $j+1$; $c_j = c_{i-1}$
For $i \neq 0$: $g_{i,j}(a,b) = 1$ iff $\langle a[j:i] \rangle + \langle b[j:i] \rangle \geq \langle 10^{j-i+1} \rangle$
A carry out c_j will always be generated
Special case for $i = 0$:
$g_{0,j}(a,b,c_{\text{in}}) = 1$ iff $\langle a[j:0] \rangle + \langle b[j:0] \rangle + c_{\text{in}} \geq \langle 10^{j-i+1} \rangle$
In the following: omit (a,b) and write shortly $p_{i,j}$ and $g_{i,j}$
Properties (single position)
1. $p_{i,i} = a[i] \oplus b[i]$
2. $g_{i,i} = a[i] \wedge b[i]$ for $i > 0$
3. $g_{0,0} = ((a[0] \oplus b[0]) \wedge c_{\text{in}}) \vee (a[0] \wedge b[0])$
(Same as $s[1]$ in a FA; more classical construction than given previously)
4. $g_{0,j} = c[j]$ with $c[j]$ being the carry bit for binary addition

Properties (intervals)
Let $i \leq j < k$ (for intervals $[i:j]$ and $[j+1:k]$)
TODO figure

Then:
4. $p_{i,k} = p_{i,j} \wedge p_{j+1,k}$
5. $g_{i,k} = g_{j+1,k} \vee p_{j+1,k} \wedge g_{i,j}$

Circuit $\circ$ over $M = \mathbb{B}^2$, which is associative (exercise)



Optimization: $g = \neg\neg(g_2 \vee g_1 \wedge p_2) = \neg(\neg g_2 \wedge \neg(g_1 \wedge p_2))$
Cost and delay of optimized version: $C_\circ = 7$ and delay $D_\circ = 2$

Use this circuit in a parallel-prefix computation
with inputs $p[i] = p_{i,i} = a[i] \oplus b[i]$ and $g[i] = a[i] \wedge b[i]$ for $i > 0$
and $g[0] = ((a[0] \oplus b[0]) \wedge c_{\text{in}}) \vee (a[0] \wedge b[0])$
and outputs $G[i]$ and $P[i]$

Thus,
$(G[i], P[i] = (g[i], p[i]) \circ \cdots \circ (g[0], p[0])$
$= (g_{i,i}, p_{i,i}) \circ \cdots \circ (g_{0,0}, p_{0,0})$
$= (g_{0,i}, p_{0,i})$

Carry-lookahead adder (CLA)
Cost and delay
$C_{CLA}(n) = C_{PP-\circ}(n) + 2 \cdot n \cdot C(XOR) + (n+1) \cdot C(AND) + C(OR)$
$D_{CLA}(n) = D_{PP-\circ}(n) + 2 \cdot D(XOR) + D(AND) + D(OR)$

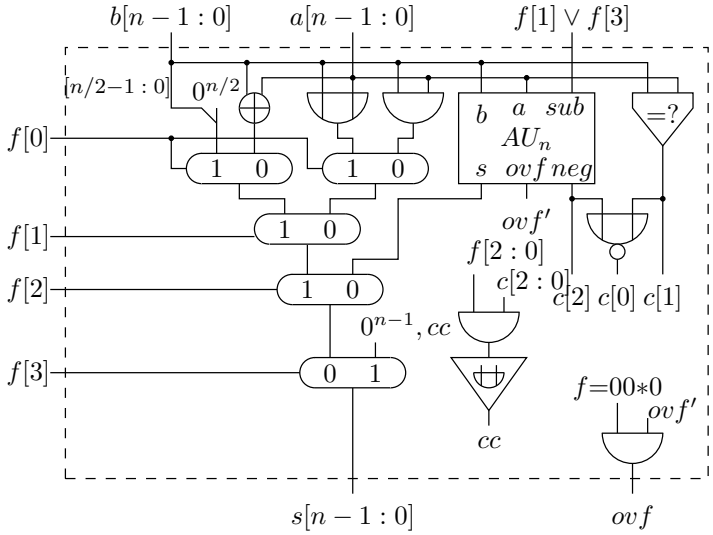
Arithmetic unit [2, Section 2.4.5]
TODO fill in material

Shifter
Definitions

Lecture 4, 25 Oct 2006

Basics
Combinational circuits
Arithmetic circuits [2, Section 2.4]
Fixed-point arithmetic logical unit (ALU or XPU) [2, Section 3.3.4]
Inputs: $a[n-1:0]$, $b[n-1:0]$, function code $f[3:0]$
Outputs: $s[n-1:0] = aluop(f, a, b)$, ovf

$f[3]$	$f[2]$	$f[1]$	$f[0]$	$s[n-1:0]$	ovf
0	0	0	0	$a +_n b$	$[a]+[b] \notin T_n$
0	0	0	1	$a +_n b$	0
0	0	1	0	$a -_n b$	$[a]-[b] \notin T_n$
0	0	1	1	$a -_n b$	0
0	1	0	0	$a \wedge b$	0
0	1	0	1	$a \vee b$	0
0	1	1	0	$a \oplus b$	0
0	1	1	1	$b[n/2-1:0]0^{n/2}$	0
	$<$	$=$	$>$	$s[0]$	ovf
1	0	0	0	0	0
1	0	0	1	$[a] > [b]$	0
1	0	1	0	$[a] = [b]$	0
1	0	1	1	$[a] \geq [b]$	0
1	1	0	0	$[a] < [b]$	0
1	1	0	1	$[a] \neq [b]$	0
1	1	1	0	$[a] \leq [b]$	0
1	1	1	1	1	0



$c[1] = eq = (a[n-1:0] = b[n-1:0])$ (Equality tester)
$c[2] = lt$ (Negation output)
$c[0] = gt = \neg lt \wedge \neg eq = \neg(lt \vee eq)$
Notation: $a[n-1:0] = (b_{n-1}, \ldots, b_0)$ for $b_i \in \mathbb{B} \cup \{\star\}$
True if for all i with $b_i \neq \star$ we have $a[i] = b_i$
Definition $\langle a \rangle +_n \langle b \rangle \equiv \langle s \rangle \mod 2^n$ Definition $\langle a \rangle -_n \langle \neg b \rangle + 1 \equiv \langle s \rangle \mod 2^n$ Shifters and shifter unit
Shift functions
Let $a[n-1:0]$, $i \in \{0, \ldots, n-1\}$
Cyclical left shift: $cls(a,i) = (a[n-1-i:0], a[n-1:n-i])$
Cyclical right shift: $crs(a,i) = (a[i-1:0], a[n-1:i])$
Logical left shift: $lls(a,i) = (a[n-1-i:0], 0^i)$

Logical right shift: $lrs(a,i) = (0^i, a[n-1:i])$
Arithmetical Right Shift: $ars(a,i) = (a[n-1]^i, a[n-1:i])$
Properties:
$\langle lls(a,i)\rangle \equiv \langle a\rangle \cdot 2^i \mod 2^n$
$\langle lrs(a,i)\rangle \equiv \lfloor \langle a\rangle / 2^i \rfloor$
$[ars(a,i)] \equiv \lfloor [a]/2^i \rfloor$

Incrementer
Compute $a +_n 0^{n-1}c_{\text{in}}$!
Exercise?

Half-Decoder (implementation)
Specification:
Input $a[n-1:0]$, output $b[n-1:0]$ with $\langle a\rangle = \langle b\rangle_{\text{hu}}$, which is $b = 0^{2^n-i}1^i$ for $i = \langle a\rangle$
Implementation...

Shifter implementations
Assume $n = 2^k$
Cyclic left shift by constant or identity: $(n,i) - CLS$
Cyclic left shift: $n - CLS$
Proof: induction on shift depth, exercise
Cylic left-right shift
Specification: $c[n-1:0] = \begin{cases} cls(a,\langle b\rangle) & \text{if } r = 0 \\ crs(a,\langle b\rangle) & \text{otherwise} \end{cases}$
Implementation
Proof (only right shift):
Case $b = 0^k$ we get $b' = 0^k$ and thus $cls(a,\langle b'\rangle) = cls(a,0) = crs(a,0) = crs(a,\langle b\rangle$
Case $b \neq 0^k$ we get $\langle b'\rangle = 2^k - \langle b\rangle$ because
$\langle \neg b\rangle + 1 < 2^k$ (no carry out)
$\langle b'\rangle = \langle \neg b\rangle + 1 = \langle 0, \neg b\rangle + 1 = [0, \neg b] + 1 = -[1,b] = 2^k - \langle b\rangle$

Shifter Unit

$f[1]$	$f[0]$	$c[n-1:0]$
0	0	$lls(a,\langle b\rangle)$
0	1	(unspecified)
1	0	$lrs(a,\langle b\rangle)$
1	1	$ars(a,\langle b\rangle)$

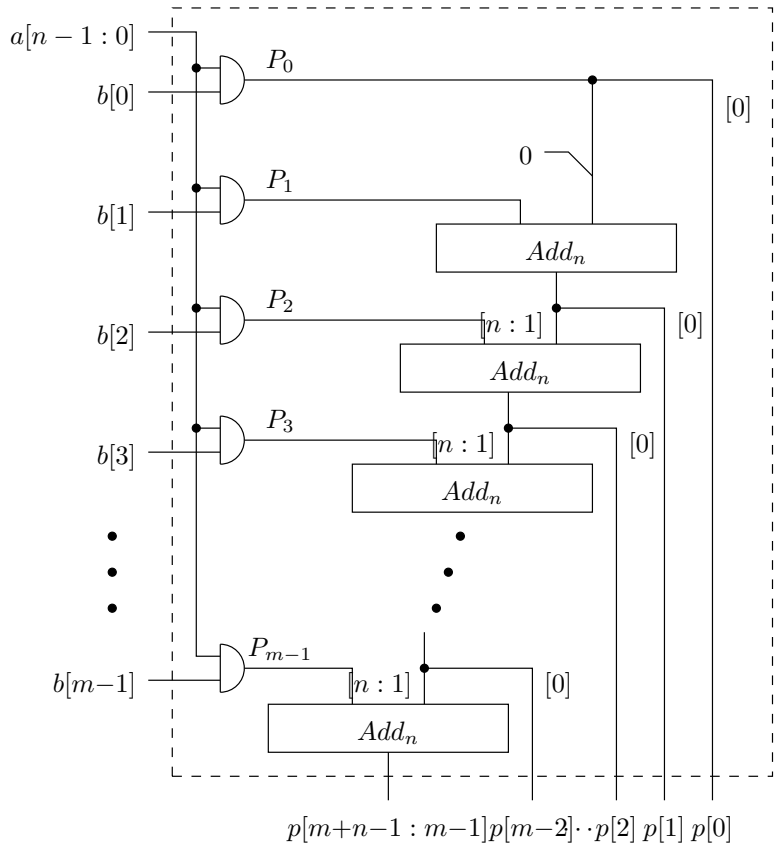
TODO Implementation

Arithmetic circuits: Multipliers
Let $a[n-1:0]$ and $b[m-1:0]$ for $n, m \in \mathbb{N}$
Since, $\langle a\rangle \cdot \langle b\rangle \leq (2^n-1)\cdot(2^m-1) \leq 2^{n+m}-1$, the product of $\langle a\rangle$ and $\langle b\rangle$ can be represented as an $(n+m)$-bit binary number.
An (n,m)-multiplier is a circuit with inputs $a[n-1:0]$ and $b[m-1:0]$ that computes an output $p[(n+m)-1:0]$ with $\langle p\rangle = \langle a\rangle \cdot \langle b\rangle$.
School method
Sum of partial products: $\langle a\rangle \cdot \langle b\rangle = \sum_{t=0}^{m-1} \langle a\rangle \cdot b[t] \cdot 2^t$
Partial product t: $\langle a\rangle \cdot b[t] \cdot 2^t = \langle a \wedge b[t], 0^t\rangle$
(Partial) sum of partial product for $j + k < m$
$S_{j,k} = \sum_{t=j}^{j+k-1} \langle a\rangle \cdot b[t] \cdot 2^t$
$= \langle a\rangle \cdot \langle b[j+k-1:j]\rangle \cdot 2^j < 2^{n+k+j}$
is a multiple of 2^j with an $(n+k)$-bit representation

TODO figure
Properties
$S_{j,1} = \langle a \rangle \cdot b[j] \cdot 2^j$
$\langle a \rangle \cdot \langle b \rangle = S_{0,m}$
$S_{j,k+h} = S_{j,k} + S_{j+k,h}$
$S_{0,t} = S_{0,t-1} + S_{t-1,1}$
$\rightsquigarrow$ multiplier with $(m-1)$ many n-adders

Lecture 5, 30 Oct 2006

Basics
Combinational circuits
Arithmetic circuits: Multipliers
(n, m)-multiplier: inputs $a[n-1:0]$ and $b[m-1:0]$, output $p[n+m-1:0]$ with $\langle p \rangle = \langle a \rangle \cdot \langle b \rangle$
School method
$S_{j,k} = \sum_{t=j}^{j+k-1} \langle a \rangle \cdot b[t] \cdot 2^t < 2^{n+k+j}$
Properties
$S_{j,1} = \langle a \rangle \cdot b[j] \cdot 2^j$
$\langle a \rangle \cdot \langle b \rangle = S_{0,m}$
$S_{j,k+h} = S_{j,k} + S_{j+k,h}$
$S_{0,t} = S_{0,t-1} + S_{t-1,1}$
Hence:
$S_{0,m} = S_{0,m-1} + S_{m-1,1}$
$= S_{0,m-2} + S_{m-2,1} + S_{m-1,1} = S_{0,0} + S_{1,0} + \cdots + S_{m-2,1} + S_{m-1,1}$
Define: $P_t = a \wedge b[t]$
$\rightsquigarrow$ multiplier with $(m-1)$ many n-adders



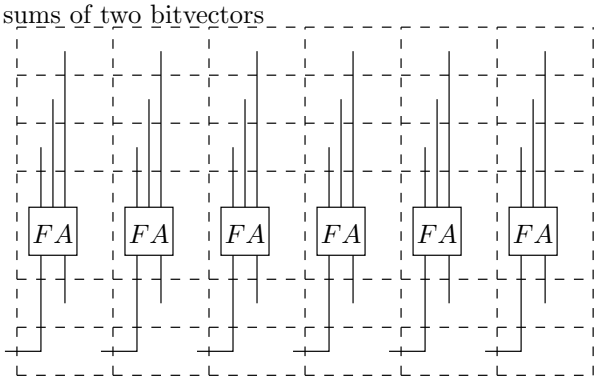
With carry-chain adders, cost is in $O(n \cdot m)$ and delay is in $O(n + m)$. Too slow!

3/2-Adders or Carry Save Adders

Bit vectors s and t are called a carry-save representation of $x \in \mathbb{N}$ if $\langle s \rangle + \langle t \rangle = x$

3/2-adders is a circuit with inputs $a[n-1:0]$, $b[n-1:0]$, and $c[n-1:0]$ and outputs $s[n-1:0]$ and $t[n:0]$ such that $\langle a \rangle + \langle b \rangle + \langle c \rangle = \langle s \rangle + \langle t \rangle$

Hence, 3/2 adders allow to 'compress' sums of three bitvectors into sums of two bitvectors



Proof:

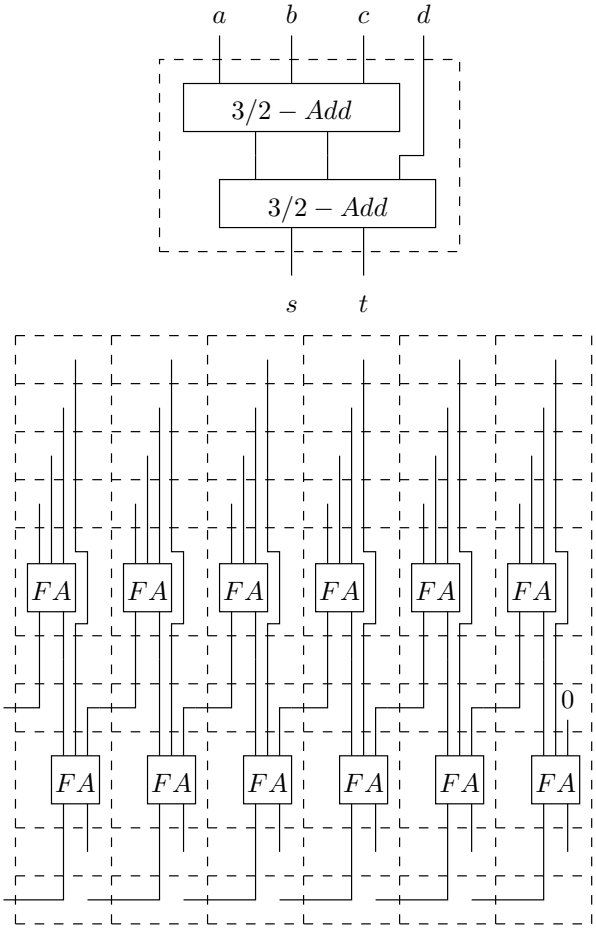
$\langle a \rangle + \langle b \rangle + \langle c \rangle = \sum_{i=0}^{n-1} (a[i] + b[i] + c[i]) \cdot 2^i$

$= \sum_{i=0}^{n-1} \langle t[i+1], s[i] \rangle \cdot 2^i$
$= \sum_{i=0}^{n-1} (2 \cdot t[i+1] + s[i]) \cdot 2^i$
$= \langle s \rangle + \langle t \rangle$

Cost: linear in n

Delay: full-adder (constant!)

4/2-Adders



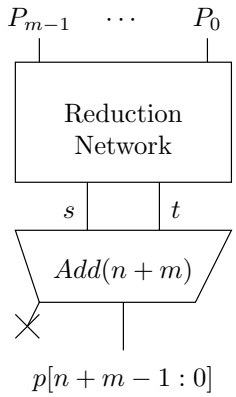
Specification: $\langle a \rangle + \langle b \rangle + \langle c \rangle + \langle d \rangle \equiv \langle s \rangle + \langle t \rangle \mod 2^n$

(Modulo suffices later on because of the bound on $S_{i,j}$)

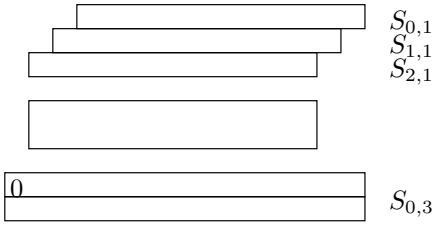
Cost: linear in n

Delay: 2 full-adders (constant!)

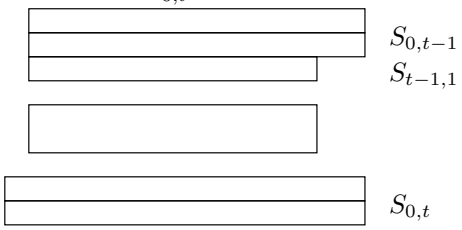
Multiplication arrays (simple)



Compute $S_{0,t}$ in carry-save representation with 3/2-adders
Start with $S_{0,3}$

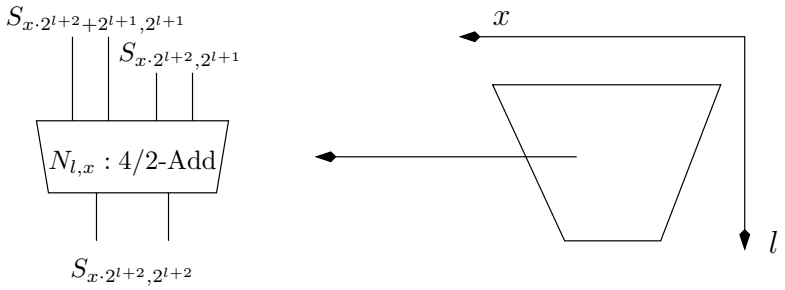


Continue with $S_{0,t}$



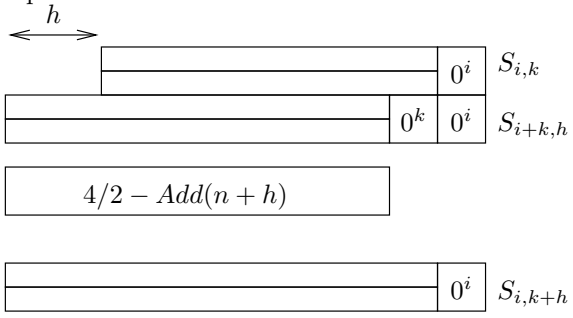
Cost: $n \cdot m \cdot C(AND) + (m-2) \cdot C(3/2 - ADD(n)) + C(CLA(n+m))$
Delay: $D(AND) + (m-2) \cdot D(FA) + D(CLA(n+m))$, linear in m, logarithmic in $n+m$

Multiplication arrays (balanced tree)
Let $m = 2^k$, only for $k \geq 2$ (e.g., for 32-bit we will have $k = 5$)
Build a completely balanced tree of 4/2-adders with levels 0 to $(k-2)$.
Let $N_{l,x}$ denote 4/2-adder x in level l
In level $l \in \{0, \ldots, k-2\}$ we have
$m/2^l = 2^{k-l}$ inputs,
$m/2^{l+2} = 2^{k-l-2}$ 4/2-adder nodes, and
$m/2^{l+1} = 2^{k-l-1}$ outputs.
The output of $N_{l,x}$ encodes $S_{x \cdot 2^{l+2}, 2^{l+2}}$, which is computed from $S_{x \cdot 2^{l+2}+2^{l+1}, 2^{l+1}}$ and $S_{x \cdot 2^{l+2}, 2^{l+1}}$
(same as: $S_{(2 \cdot x+1) \cdot 2^{l+1}, 2^{l+1}}$ and $S_{2 \cdot x \cdot 2^{l+1}, 2^{l+1}}$



$\rightsquigarrow$ overlap of 2^{l+1} bits

Computation at each node:



Delay: logarithmic in m, logarithmic in $n+m$

Cost

How much nodes? $\sum_{l=0}^{k-2} 2^{k-l-2} = \sum_{l=0}^{k-2} 2^l = 2^{k-1} - 1$

Now, each 4/2-adder node must have width n to start with, i.e., $2 \cdot (2^{k-1} - 1) \cdot n = n \cdot (m-2)$ full adders are needed

TODO pic

Furthermore, at node $N_{l,x}$ we must take into account the 'excess full adders', $\rightsquigarrow$ exercise

Clocked circuits

Hardware model (Part 2) – Storage elements

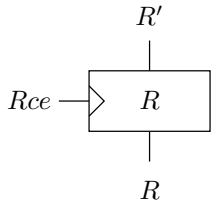
Clocked circuit contain storage elements, which may be updated each cycle

We reason about clocked circuits over time; we use s^t to denote the value of a wire, signal, input, output, or storage element at time t

Registers (also know as flip-flops or latches)

Store single bit $R \in \mathbb{B}$ or bit vector $R[n-1:0] \in \mathbb{B}^n$

Naming convention (input, output, clock enable)



Semantics

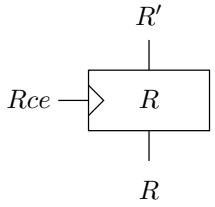
$$R^{t+1} = \begin{cases} R^t & \text{if } \neg Rce^t \\ R'^t & \text{otherwise} \end{cases}$$

Cost: $C(FF) = 8$, delay $D(FF) = 4$

On update, an additional delay of $\delta = 1$ is needed (setup and hold time)

Lecture 6, 6 Nov 2006

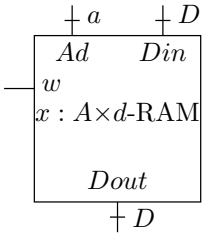
Basics
Clocked circuits
Hardware model (Part 2) – Storage elements
Clocked circuit contain storage elements, which may be updated each cycle
Notation: use s^t to denote the value of a wire, signal, input, output, or storage element at time t
Register / Flip-Flop / Latch
Store single bit $R \in \mathbb{B}$ or bit vector $R[n-1:0] \in \mathbb{B}^n$
Drawn as follows (with data input, clock enabled input, and data output)



Semantics

$$R^{t+1} = \begin{cases} R^t & \text{if } \neg Rce^t \\ R'^t & \text{otherwise} \end{cases}$$

Cost: $C(FF) = 8$, delay $D(FF) = 4$, but see below
RAM
$A = 2^a$: number of addresses
D: data width
Stores mapping $x : A \to \mathbb{B}^D$
Connections
$Ad[a-1:0]$: input address
$Din[D-1:0]$: written data
$Dout[D-1:0]$: read data
w: write enable signal



Semantics for $w^t = 0$:
$Dout^t = x^t(Ad^t)$ (read in the same cycle)
$x^{t+1} = x^t$
Semantics for $w^t = 0$:

$Dout^t$ undefined

$$x^{t+1}(a) = \begin{cases} Din^t & \text{for } a = Ad^t \\ x^t(a) & \text{otherwise} \end{cases}$$

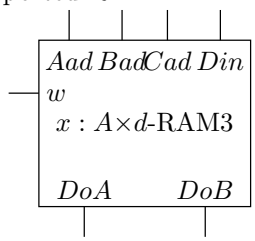
(update in the next cycle)

Cost: $C(RAM(A, D)) = 2 \cdot (A + 3) \cdot (d + \log\log d)$

Delay: $D(RAM(A, D)) = \begin{cases} \lceil \log d \rceil + \lceil A/4 \rceil & \text{if } A \leq 64 \\ 3 \cdot \lceil \log A \rceil + 10 & \text{otherwise} \end{cases}$

(but see below)

Multiported RAM



Stores mapping $x : A \to \mathbb{B}^D$

Semantics

$DoA^t = x^t(Aad^t)$ if $Aad^t \neq Cad^t$, unspecified else

$DoB^t = x^t(Bad^t)$ if $Bad^t \neq Cad^t$, unspecified else

$x^{t+1} = x^t$ if $w^t = 0$

$$x^{t+1}(a) = \begin{cases} Din^t & \text{for } a = Cad^t \\ x^t(a) & \text{otherwise} \end{cases} \quad \text{if } w^t = 1$$

Cost: $C(RAM3(A, D)) = 1.6 \cdot C(RAM(A, D))$

Delay: $D(RAM3(A, D)) = 1.5 \cdot D(RAM(A, D))$ (but see below)

Accumulated Delay and cycle time (depends on inputs and outputs)

Delay of RAMs is taken on reads, no delay on reads for registers

Accumulated delay $A(s)$: maximum delay of a path from a register or input to s

Cycle time: minimum length of a cycle; inverse of clock frequency

Cycle time is determined by the accumulated to any RAM or register input *plus* the delay to update that component

Cycle time is determined by the delay of longest paths; it is greatest number of the following three:

1. $\max\{A(s) + D(FF) + \delta \mid s \text{ input to a register}\}$
2. $\max\{A(s) + D(RAM(A, d)) + \delta \mid s \text{ input to a } RAM(A, d)\}$
3. $\max\{A(s) + D(RAM3(A, d)) + \delta \mid s \text{ input to a } RAM3(A, d)\}$

(2. and 3. are related to the write case of RAMs)

The constant $\delta = 1$ accounts for setup and hold times of registers

Since, the data outputs of RAMs are clocked into register somehow, 2. and 3. do usually not dominate the delay

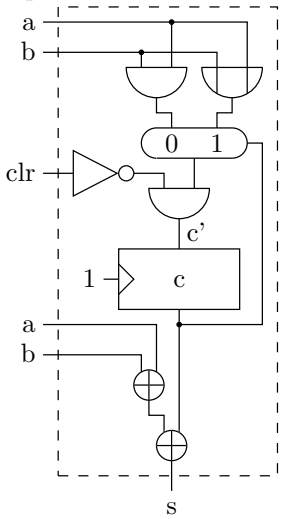
(A separate analysis for RAM reads and writes might be possible, though)

However, must be careful to take into account the accumulated delay of inputs;

Assuming $A(i) = 0$ for all inputs only yields a lower bound on cycle time

$\rightsquigarrow$ cycle time and accumulated delay of the outputs usually are a function of the accumulated delay of the inputs

Example of a clocked circuit (sequential adder):



Claim: for $t > t'$ with $clr^t = 1$ and $clr^{t''} = 0$ for all $t < t'' < t'$.

Then, $\langle s^{t'}, \ldots, s^{t+2}, s^{t+1} \rangle = \langle a^{t'}, \ldots, a^{t+2}, a^{t+1} \rangle + \langle b^{t'}, \ldots, b^{t+2}, b^{t+1} \rangle$

Accumulated delay: $A(s) = \max\{A(a), A(b)\} + 2 \cdot D(XOR)$

Accumulated delay: $A(c') = D(AND) + \max\{A(clr) + D(INV), \max\{A(a), A(b)\} + \max\{D(AND), D(OR)\} + D(MUX)$

Cycle time: $T = A(c') + D(FF) + 1$

Control Automata

(Deterministic) finite automata (DFA) with outputs also called transducer

States $Z = \{0, 1, \ldots, k-1\}$ for some k with 0 being the initial state

Inputs signals $In = \mathbb{B}^\sigma$ for some σ

Output signals $Out = \mathbb{B}^\gamma$ for some γ

Transition function: $\delta : Z \times In \to Z$

Output function: $\eta : Z \times In \to Out$

Deterministic: given current state and inputs, the next state is uniquely determined

Total: for any input there is always a next state

Here (in contrast to [2]): always assume that δ and η are fully specified

Classification

If η does not depend on the inputs, the automaton is called a Moore automaton and the output function may be written as $\eta : Z \to Out$

Otherwise, the automaton is called a Mealy automaton

Automata may be drawn as a labelled directed graphs (V, E) with nodes $V = Z$ representing the states and edges $E \subseteq V \times V$ representing (possible) transitions.

Furthermore, edges may be labeled by transition conditions:

Define $\delta_{z,z'} : In \to \mathbb{B}$ as $\delta_{z,z'}(in) = 1$ iff $\delta(z, in) = z'$

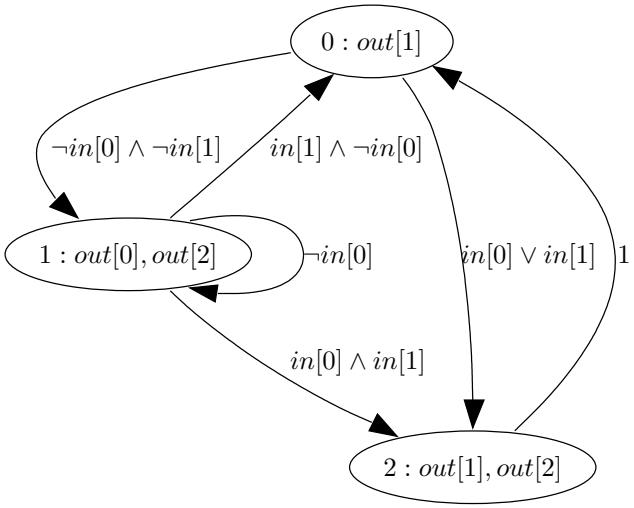
Because the automaton is deterministic and total, for all $z \in Z$ and $in \in In$, there exists a unique $z' \in Z$ such that $\delta_{z,z'}(in)$ holds.

$\rightsquigarrow$ label edge $z \to z'$ with $\delta_{z,z'}$; omit edges with $\delta_{z,z'} \equiv 0$

For a Moore automaton, each node z may be labelled with the output $\eta(z)$ in that state; usually though: just list the *active* output signals

(For a Mealy automaton, may label each node z with functions $\eta_z : In \to Out$)

Examples



Moore:

Implementation

State encoding

Unary, i.e., we have a register $S[k-1:0]$ such that $\langle S \rangle_{\mathrm{u}}$ indicate the current state of the automaton

Binary: maybe in the exercises

Generating outputs for Moore Automata

Implement circuit O, which computes $out[\gamma - 1 : 0] = \eta(\langle S \rangle_{\mathrm{u}})$ from an input $S \in \mathbb{B}^k$

Define $Z_j = \{z \in Z \mid \eta(z)[j] = 1\}$

Obviously, $out[j] = \bigvee_{z \in Z_j} S[z] = \eta(z)[j]$ (or-trees)

Abbreviate $\nu_j = \#Z_j$, the *frequency* of output signal j

Abbreviate $\nu_{\max} = \max\{\nu_j \mid 0 \le j < \gamma\}$ and $\nu_{\mathrm{sum}} = \sum_{j=0}^{\gamma-1} \nu_j$

In the example: $\nu_{\max} = 2$ and $\nu_{\mathrm{sum}} = 5$

$C(O) = \sum_{j=0}^{\gamma-1} (\nu_j - 1) \cdot C(OR) = (\nu_{\mathrm{sum}} - \gamma) \cdot C(OR)$

$D(O) = D(OR) \cdot \max\{\lceil \log v_j \rceil \mid 0 \le j < \gamma\} = \lceil \log \nu_{\max} \rceil \cdot D(OR)$

Computing the next state

Circuit CN with inputs $in \in In$ and $S \in \mathbb{B}^k$ and outputs $N \in \mathbb{B}^k$ such that $\langle N \rangle_{\mathrm{u}} = \delta(\langle S \rangle_{\mathrm{u}}, in)$

Recall: $\delta_{z,z'} : In \to \mathbb{B}$ with $\delta_{z,z'}(in) = 1$ iff $\delta(z, in) = z'$

Let $D(z, z')$ be the disjunctive normal form (DNF) of $\delta_{z,z'}$

Recall: DNF is a disjunction of monomials and a monomial is a conjunction of literals and a literal is an input variable or its negation

Let $M(z, z')$ denote all the monomials in $D(z, z')$

Let $M = \bigcup_{(z,z') \in E} M(z, z')$ denote all monomials

Then:

1. Compute all literals $in[i]$ and $\neg in[i]$

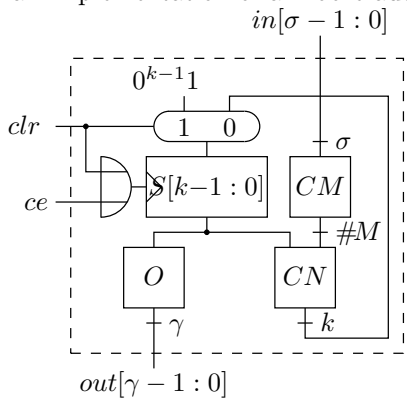
2. Compute all monomials $m \in M$
3. Compute $N[k-1:0]$

Lecture 7, 8 Nov 2006

Basics
Clocked circuits
Control Automata
Implementation
Computing the next state
Then: continue with the book, but also compute $N[0]$!
Recall:
Goal currently: compute N from S and in with $\langle N \rangle_{\mathrm{u}} = \delta(\langle S \rangle_{\mathrm{u}}, in)$
$\delta_{z,z'} : In \to \mathbb{B}$ with $\delta_{z,z'}(in) = 1$ iff $\delta(z, in) = z'$
$D(z, z')$: the DNF of $\delta_{z,z'}$
$M(z, z')$: all monomials in $D(z, z')$
$M = \bigcup_{(z,z') \in E} M(z, z') \backslash \{1\}$: all monomials (without the trivial monomial)
1. Compute negations $\neg in[i]$ of all inputs $in[i]$ (with inverters)
2. Compute all monomials $m \in M$ (with AND-trees)
3. Compute $N[k-1:0]$ (with OR-trees):
$N[z] = \bigvee_{(z',z) \in E} (S[z'] \wedge \bigvee_{m \in M(z',z)} m)$
$= \bigvee_{(z',z) \in E} \bigvee_{m \in M(z',z)} S[z'] \wedge m$
Cost and delay for 1. and 2. (circuit CM)
Let $l(m)$ denote the number of literals in a monomial m
Define $l_{\max} = \max\{l(m) \mid m \in M\}$
Define $l_{\mathrm{sum}} = \sum_{m \in M} l(m)$
Cost $C(CM) = \sigma \cdot C(INV) + (l_{\mathrm{sum}} - \#M) \cdot C(AND)$
Delay $D(CM) = D(INV) + \lceil \log l_{\max} \rceil \cdot D(AND)$
Cost and delay for 3. (circuit CN)
Define $fanin(z) = \sum_{(z',z) \in E} \#M(z', z')$
Define $fanin_{\max} = \max\{fanin(z) \mid 0 \le z < k\}$
Define $fanin_{\mathrm{sum}} = \sum_{z=0}^{k-1} fanin(z)$
Cost $C(CN) = fanin_{\mathrm{sum}} \cdot (C(AND) + C(OR)) - k \cdot C(OR)$
Cost $D(CN) = \lceil \log fanin_{\max} \rceil \cdot C(AND) - D(AND)$
Cost and delay of 1. to 3. (circuit NS)
Cost $C(NS) = C(CM) + C(CN)$
Delay $D(NS) = D(CM) + D(CN)$

Parameter	Description
σ	Number of input signals
k	Number of states
γ	Number of output signals
$\nu_{\max}$	Maximal frequency of all outputs
ν_{sum}	Sum of all output frequencies
$\#M$	Number of (non-trivial) monomials M in the DNFs of the transition conditions
$l_{\max}$	Maximal number of literals of monomials in M
l_{sum}	Number of literals in the monomials in M
$fanin_{\max}$	Maximal fanin into a state
$fanin_{\text{sum}}$	Sum of the fanins into all states

Overall implementation of a Moore automaton

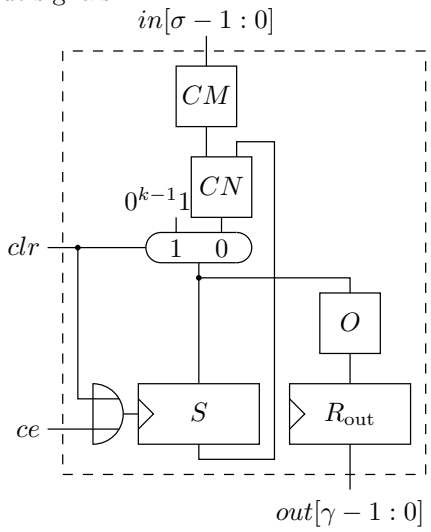


Cost $C(Moore) = k \cdot C(FF) + C(O) + C(NS) + k \cdot C(MUX) + C(OR)$

Accumulated output delay $A(out) = D(O)$

Cycle time $T(Moore) = \max\{\max\{A(clr), A(ce)\} + D(OR), A(in) + D(NS)\} + D(MUX) + D(FF) + \delta$

Overall implementation of a Moore automaton with precomputed output signals



Cost $C(pMoore) = (k+\gamma)\cdot C(FF)+C(O)+C(NS)+k\cdot C(MUX)+C(OR)$
Accumulated output delay $A(out) = 0$
Cycle time $T(pMoore) = \max\{\max\{A(clr), A(ce)\}+D(OR), A(in)+D(NS)\} + D(MUX) + D(O) + D(FF) + \delta$

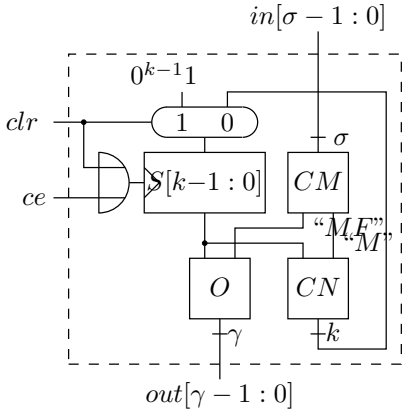
Mealy automata

Without loss of generality, assume that there exists some γ' such that $out[j]$ for $j < \gamma'$ *does not* depend on the inputs and $out[j]$ for $j \geq \gamma'$ *does* depend on the inputs
The former are called Moore outputs, the latter are called Mealy outputs.
Idea: compute the function η_z for $z \geq \gamma'$ similar to the regular transition function
Let $f_{z,j} : In \to \mathbb{B}$ compute output j in state z, i.e., $f_{z,j}(in) = \eta(z, in)[j]$ (for all $k \geq \gamma'$)
Let $Z'_j = \{z \mid f_{z,j} \not\equiv 0\}$ denote all states for which the Mealy output $out[j]$ may become active
Let $F(z, j)$ denote the DNF of $f_{z,j}$
Let $MF(z, j)$ denote the monomials in $F(z, j)$
Define $MF = \bigcup_{j=\gamma'}^{\gamma-1} MF(z, j) \setminus \{1\}$: all monomials (without the trivial monomial)
Define $M' = M \cup MF$
Implementation

1. Extend circuit CM to also compute all monomials in M'
2. Extend circuit O to compute the outputs $out[j]$ for $j \geq \gamma'$ as
   $out[j] = \bigvee_{z\in Z'_j}(S[z] \wedge \bigvee_{m\in MF(z,j)} m)$
   $= \bigvee_{z\in Z'_j} \bigvee_{m\in MF(z,j)} S[z] \wedge m$

Cost and delay of CM
Define $l_{\max} = \max\{l(m) \mid m \in M\}$
Define $lf_{\max} = \max\{l(m) \mid m \in MF\}$
Define $l_{\text{sum}} = \sum_{m\in M'} l(m)$
Cost $C(CM) = \sigma \cdot C(INV) + (l_{\text{sum}} - \#M') \cdot C(AND)$
Delay $D(CM)(M) = D(INV) + \lceil \log l_{\max} \rceil \cdot D(AND)$
Delay $D(CM)(MF) = D(INV) + \lceil \log lf_{\max} \rceil \cdot D(AND)$
Cost and delay of O
Define $\nu_j = \sum_{z\in Z'_k} \#MF(z, j)$ for $j \geq \gamma'$, the *frequency* of Mealy output signal j
Define ν_{sum} and $\nu_{\max}$ as before
$C(O) \leq \nu_{\text{sum}} \cdot (C(AND) + C(OR)) - \gamma \cdot C(OR)$
$D(O) = D(AND) + \lceil \log \nu_{\max} \rceil \cdot D(OR)$

Overall



Cost $C(Mealy) = k \cdot C(FF) + C(O) + C(NS) + k \cdot C(MUX) + C(OR)$

Accumulated output delay $A(out[\gamma' - 1 : 0]) = D(O)$

Accumulated output delay $A(out[\gamma - 1 : \gamma']) = A(in) + D(CM)(MF) + D(O)$

Cycle time $T(Mealy) = \max\{\max\{A(clr), A(ce)\} + D(OR), A(in) + D(NS)\} + D(MUX) + D(FF) + \delta$

Datapaths and control

Suppose we have datapaths controlled by a Mealy automaton

Computations in the datapaths and computation of the Mealy outputs may depend on each other in a complex manner (path in the combined circuit may often cross the boundary between the datapaths and the automaton; of course *no* cycles are allowed)

To get a good bound on cycle time, a more refined delay analysis by partitioning the datapaths and the Mealy output computation is required.

Example (not presented):

Partition of the datapaths into p parts $DP(1)$ to $DP(p)$

Partition of the output circuit into p parts $O(1)$ to $O(p)$

Partition of the output signals into p parts $out(1)$ to $out(p)$

Partition of the input signals into p parts $in(1)$ to $in(p)$

Such that:

1. $O(i)$ depends on register inputs and inputs from $DP(j)$ with $j < i$ only

2. The outputs $out(i)$ are only fed into $DP(i)$

Then,

$A(O(i)) = D(O(1)) + \sum_{j=2}^{i}(D(DP(j-1)) + D(O(j)))$

Chapter 3

Instruction Set Architecture

Lecture 7, 8 Nov 2006 (cont.)

Instruction Set Architecture (ISA)
- Based on the DLX architecture from [1]
- 32-bit Reduced Instruction Set Computer (RISC) architecture, load / store architecture
- Note: We only define the ISA partially first, it will get more complex later
- Informal model:
  - 32 general-purpose register (GPR) of width 32.
  - These registers are denoted $GPR(a)$ for register indices $a[4:0]$
  - Register $GPR(0^5)$ is always equal to 0^{32}
  - byte-oriented memory with 32-bit addresses
  - two (!) program counters DPC and PC' (the latter also often written as PCP?)
    - $\rightsquigarrow$ one delay slot, cf. below
  - 32-bit instructions, read from the DPC address
  - For now, the architecture has
    - fixed-point arithmetic, test, bit-wise logical and shift instructions,
    - register-indexed load / store operations,
    - unconditional and conditional relative or computed control-flow instructions, and
  - Most instructions have two source operands and one destination operand.
  - Source operands are *usually* two general-purpose registers or one general-purpose register and a (sign-extended) immediate constant, which is part of the instruction word.

Lecture 8, 13 Nov 2006

Instruction Set Architecture (ISA)
- Formal definition

TODO Change back (I) to simplify definition?
Configurations $c \in C$ are quadruples $c = (c.DPC, c.PCP, c.gpr, c.m)$ with the following components:

Program counters $c.DPC \in \mathbb{B}^{32}$ and $c.PCP \in \mathbb{B}^{32}$

$c.DPC[1:0]$ and $c.PCP[1:0]$ will be equal to 0^2 all the time

General purpose register file $c.gpr : \mathbb{B}^5 \to \mathbb{B}^{32}$

$c.gpr(0^5)$ will be equal to 0^{32} all the time

Memory $c.m : \mathbb{B}^{32} \to \mathbb{B}^8$

Notation $m_d(a)$ for $d \in \mathbb{N}$:

$m_d(a) = (m(a_{d-1}), m(a_{d-2}), \ldots, m(a_0))$ for $\langle a_i \rangle \equiv \langle a \rangle + i \mod 2^{32}$

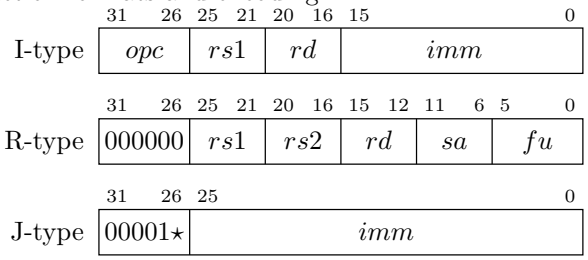
Now define transition function $\delta : \mathbb{B} \times C \to C$

Apart from the reset signal (boolean flag), no inputs and outputs (I/O devices!) considered yet

$\delta(reset, c) = c'$

Current instruction $I(c) = c.m_4(c.DPC) \in \mathbb{B}^{32}$

Instruction formats and encoding



Note: not all instruction are 'legal', illegal instructions are given below

Table of I-type and J-type instructions (with $I[31:26] \neq 0^6$)

	$I[31:29]$							
$I[28:26]$	000	001	010	011	100	101	110	111
000	$(*)$			*clri*	*lb*	*sb*		
001		*addi*		*sgri*	*lh*	*sh*		
010	*j*			*seqi*				
011	*jal*	*subi*		*sgei*	*lw*	*sw*		
100	*beqz*	*andi*		*slsi*	*lbu*			
101	*bnez*	*ori*		*snei*	*lhu*			
110		*xori*	*jr*	*slei*				
111		*lhgi*	*jalr*	*seti*				

↑jump rel. / branch load↑ ↑store
arithmetic / logical↑ ↑jump reg.
↑tests

Table of R-type instructions (with $I[31:26] = 0^6$)

	$I[5:3]$							
$I[2:0]$	000	001	010	011	100	101	110	111
000	*slli*					*clr*		
001					*add*	*sgr*		
010	*srli*					*seq*		
011	*srai*				*sub*	*sge*		
100	*sll*				*and*	*sls*		
101					*or*	*sne*		
110	*srl*				*xor*	*sle*		
111	*sra*				*lhg*	*set*		
	shifts↑		arithmetic / logical↑			↑tests		

Empty table cells correspond to illegal instructions

$illegal(I)$ = "disjunction over all empty table cells"

(Primary) opcode $opc(I) = I[31:26]$

Recognize register-type instruction: $rtype(I) = (opc(I) = 0^6)$

Recognize jump-type instruction: $jtype(I) = (opc(I) = 00001\star)$

Recognize immediate-type instruction: $itype(I) = \neg(rtype(I) \vee jtype(I))$

Register source 1: $rs1(I) = I[25:21]$

Register source 2: $rs2(I) = I[20:16]$

Register destination: $rd(I) = \begin{cases} I[20:16] & \text{if } itype(I) \\ I[15:11] & \text{otherwise} \end{cases}$

Function, secondary opcode: $fu(I) = I[5:0]$

(Sign-extended) immediate constant: $simm(I) = \begin{cases} (I[15]^{16}, I[15:0]) & \text{if } itype(I) \\ (I[25]^6, I[25:0]) & \text{otherwise} \end{cases}$

Shift amount: $sa(I) = I[10:6]$

Constant $co(I) = \begin{cases} (0^{27}, sa(I)) & \text{if } rtype(I) \\ simm(I) & \text{otherwise} \end{cases}$

Left operand $lop(c) = c.gpr(rs1(I(c)))$

$shifti(I) = rtype(I) \wedge ((fu(I) = 000000) \vee (fu(I) = 00001\star))$

(But see below)

Right operand $rop(c) = \begin{cases} c.gpr(rs2(I(c))) & \text{if } rtype(I(c)) \wedge \neg shifti(I(c)) \\ co(c) & \text{otherwise} \end{cases}$

Arithmetic, logical, and test instructions

$alui(I) = (opc(I) = 011\star\star\star) \vee (opc(I) = 0011\star\star) \vee (opc(I) = 0010\star 1)$

(everything but the addition and subtraction that generate overflow signals)

$alu(I) = rtype(I) \wedge ((fu(I) = 101\star\star\star) \vee (fu(I) = 1001\star\star) \vee (fu(I) = 1000\star 1))$

$aluf(I) = \begin{cases} (opc(I)[4], opc(I)[2:0]) & \text{if } itype(I) \\ fu(I)[3:0] & \text{otherwise} \end{cases}$

$c'.gpr(rd(I(c)) = aluop(aluf(I(c)), lop(c), rop(c))$

Memory unchanged, PC see below

Shift instructions

$shifti(I) = rtype(I) \wedge ((fu(I) = 000000) \vee (fu(I) = 00001\star))$

$shift(I) = rtype(I) \wedge ((fu(I) = 000100) \vee (fu(I) = 00011\star))$

$shf(I) = fu(I)[1:0]$

$c'.gpr(rd(I(c)) = shop(shf(I(c)), lop(c), rop(c))$
Memory unchanged, PC see below

Load / store instructions
$ea(c) = c.gpr(rs1(I(c))) +_{32} simm(I(c))$
$load(I) = (opc(I) = 10000\star) \vee (opc(I) = 100011) \vee (opc(I) = 10010\star)$
$store(I) = (opc(I) = 10100\star) \vee (opc(I) = 101011)$
$ls(I) = (opc(I) = 100\star0\star) \vee (opc(I) = 10\star0\star1) \vee (opc(I) = 10\star00\star)$
Access width $d(I) = \langle I[27:26] \rangle + 1$
Alignment: access width must divide effective address, $d(I(c)) \mid \langle ea(c) \rangle$
Store: $c'.m_{d(I(c))}(ea(c)) = c.gpr(rd(I(c)))[8 \cdot d(I(c)) - 1 : 0]$ (register unchanged, PC see below)
Load:
$u(I) = I[28]$ (unsigned bit)
$fill(c) = \neg u(c) \wedge c.m_{d(I(c))}(ea(c))[8 \cdot d(I(c)) - 1]$
$c'.gpr(rd(I(c))) = (fill(c)^{32-8 \cdot d(I(c))}, c.m_{d(I(c))}(ea(c)))$ (memory unchanged, PC see below)

Control-flow and control-flow instructions
DPC update
$$c'.dpc = \begin{cases} 0^{32} & \text{if } reset \\ c.pcp & \text{otherwise} \end{cases}$$
PCP update
Next sequential PC: $nspc(c) = c.pcp +_{32} 0^{29}100$ (note: $\langle 0^{29}100 \rangle = 4$)
$jump(I) = (opc(I) = 00001\star) \vee (opc(I) = 01011\star)$
$branch(I) = (opc(I) = 00010\star)$
$btaken(c) = opc(c)[0] \oplus (c.gpr(rs1(I(c)) = 0^{32}))$
$bjtaken(c) = jump(c) \vee branch(I(c)) \wedge btaken(c)$
$jumpr(I) = I[30]$ (given that $branch(I) \vee jump(I)$)
$$target(c) = \begin{cases} c.gpr(rs1(I(c))) & \text{if } jumpr(I(c)) \\ c.pcp +_{32} simm(c) & \text{otherwise} \end{cases}$$
$$c'.pcp = \begin{cases} 0^{29}100 & \text{if } reset \\ nspc(c) & \text{if } \neg reset \wedge (illegal(I(c)) \vee \neg bjtaken(c)) \\ target(c) & \text{otherwise} \end{cases}$$
Link instructions save a return address in the last register
$link(I) = jal(I) \vee jalr(I)$
If $link(I(c))$, also update $c'.gpr(1^5) = nspc(c)$

3.1 Differences to [2, Chapter 3]

- Delayed PC
- Illegalize the 'overflow' variants of addition and subtraction instructions
- Behaviour for illegal instructions is fully described
- No drivers
- States *test* and *alu* as well as *testI* and *aluI* have been merged; the glue logic for the ALU environment is simplified

3.2 Differences to previous scripts

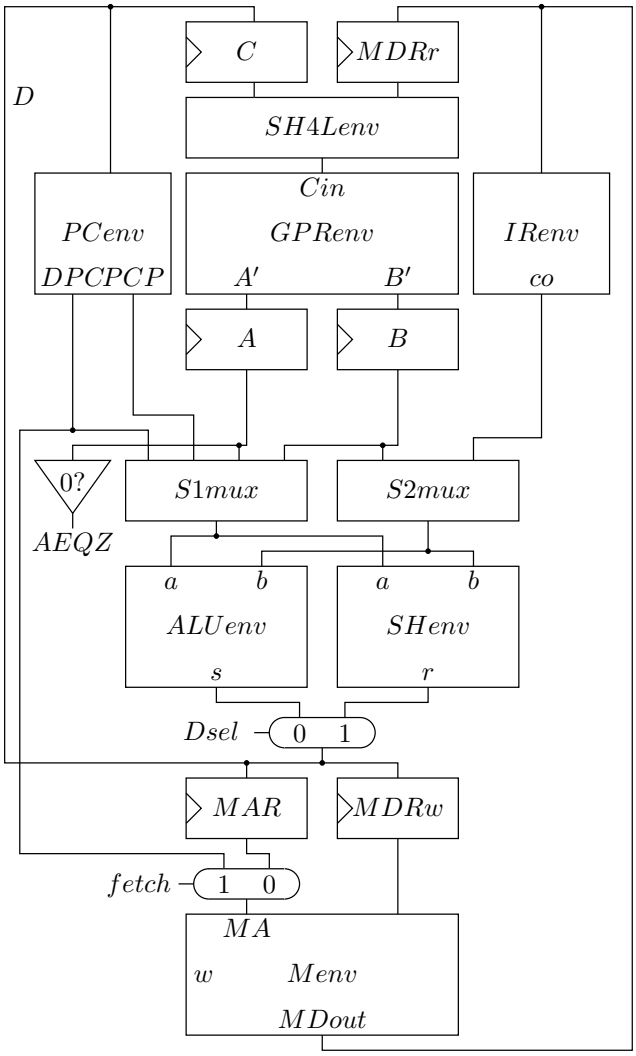
- In the previous version of the Computer Architecture 1 lecture, the first processor that was presented was fully sequential, i.e., apart from the visible registers it had no additional register stages.
- Some functions and predicates are defined on instruction words rather than on full configurations. This allows to reuse these predicates easier in the implementation.

Chapter 4

Sequential Implementation

Lecture 8, 13 Nov 2006 (cont.)

Sequential Implementation
Datapaths
Top-level datapaths



Lecture 9, 15 Nov 2006

Sequential Implementation

Datapaths

GPRenv

Inputs: $IR[25:11]$, $Cin[31:0]$, $Jlink$, $Rtype$, $GPRw$

Outputs: $A'[31:0]$, $B'[31:0]$

Let $h.gpr : \mathbb{B}^5 \to \mathbb{B}^{32}$ denote the hardware's register file contents (and $h'.gpr$ the next cycle's contents)

Spec:

$GPRw = 0$

$$A' = \begin{cases} h.gpr(rs1(IR)) & \text{if } rs1(IR) \neq 0^{32} \\ 0^{32} \text{otherwise} \end{cases}$$

$$B' = \begin{cases} h.gpr(rs2(IR)) & \text{if } rs2(IR) \neq 0^{32} \\ 0^{32} \text{otherwise} \end{cases}$$

($rs1(IR)$ and $rs2(IR)$ need $IR[25:11]$ and $Rtype$)

$GPRw = 1$

$$h'.gpr(a) = \begin{cases} h.gpr(a) & \text{if } a \neq rd(IR)s \\ Cin \text{otherwise} \end{cases}$$

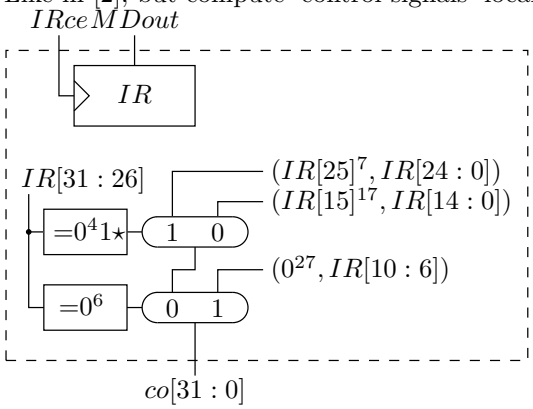
Implementation: [2], exercise (almost)

IRenv

Inputs: $MDout[31:0]$, $IRce$

Outputs: $co[31:0]$, $IR[31:0]$

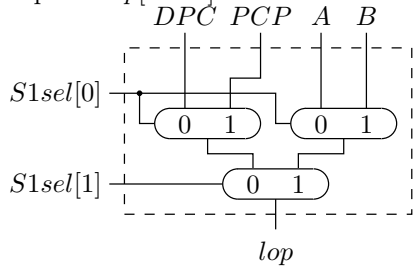
Like in [2], but compute 'control signals' locally



S1mux

Inputs: $DPC[31:0]$, $PCP[31:0]$, $A[31:0]$, $B[31:0]$, $S1sel[1:0]$

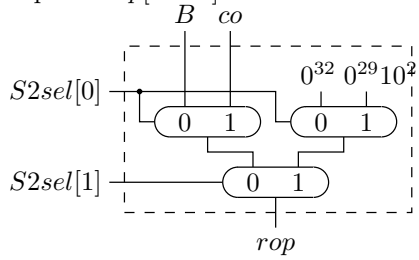
Outputs: $lop[31:0]$



S2mux

Inputs: $B[31:0]$, $co[31:0]$, $S2sel[1:0]$

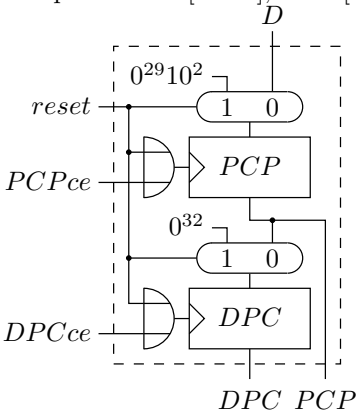
Outputs: $rop[31:0]$



PCenv

Inputs: $reset$, $D[31:0]$, $PCPce$, $DPCce$

Outputs: $DPC[31:0]$, $PCP[31:0]$



ALUenv

Inputs: $lop[31:0]$, $rop[31:0]$, $IR[31:0]$, add, $Rtype$

Outputs: $s[31:0]$

Use regular $ALU(32)$

Discard ovf bit for now

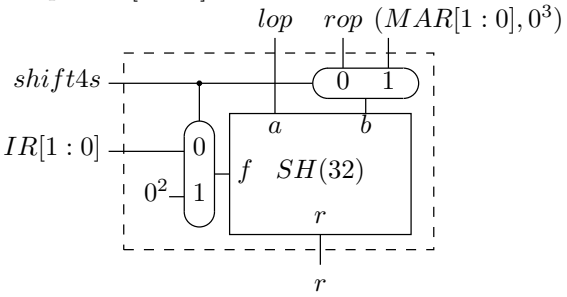
Compute function code f as follows:

$$f[3:0] = \begin{cases} (0^3, 1) & \text{if } add \\ IR[3:0] & \text{if } \neg add \wedge Rtype \\ (IR[30], IR[28:26]) & \text{otherwise} \end{cases}$$

SHenv

Inputs: $lop[31:0]$, $rop[31:0]$, $MAR[1:0]$, $shift4s$

Outputs: $r[31:0]$

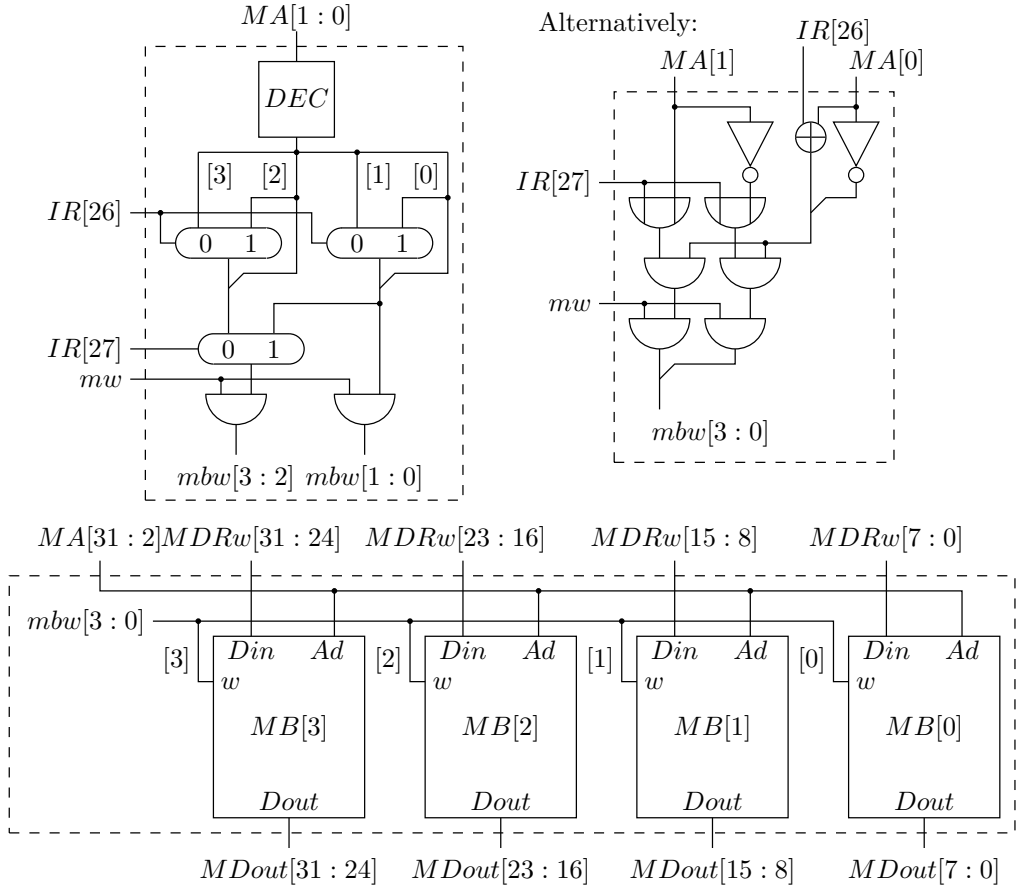


Menv

Inputs: $MA[31:0]$, $MDRw[31:0]$, $IR[27:26]$, mw

Outputs: $MDout[31:0]$

$I[27:26]$	$d(I)$	$MAR[1:0]$	$mbw[3:0]$
00	1	00	0001
		01	0010
		10	0100
		11	1000
01	2	00	0011
		10	1100
11	4	00	1111

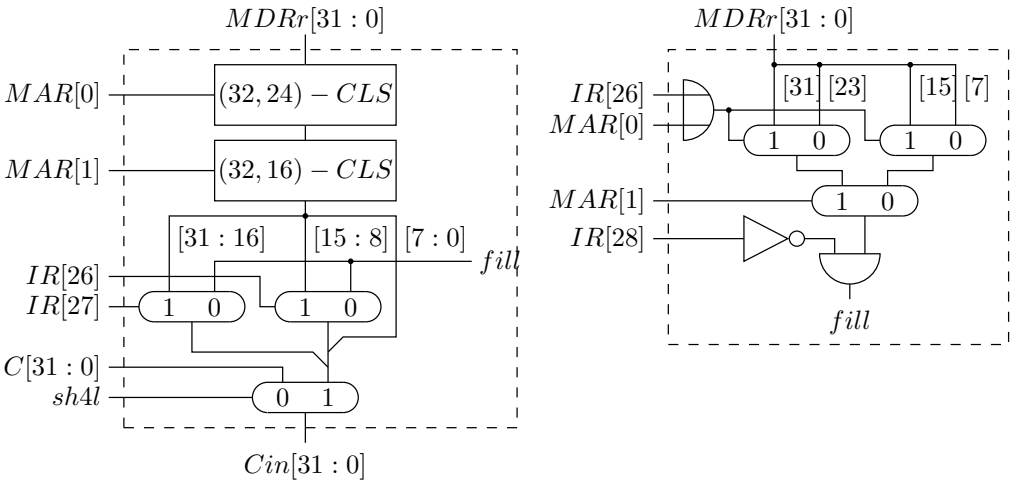


SH4Lenv

Inputs: $C[31:0]$, $MDRr[31:0]$, $MAR[1:0]$, $shift4l$

Outputs: $C'[31:0]$

$I[28]$	$u(I)$	$I[27:26]$	$d(I)$	$MAR[1:0]$	$fill$
0	0	00	1	00	$MDRr[7]$
				01	$MDRr[15]$
				10	$MDRr[23]$
				11	$MDRr[31]$
		01	2	00	$MDRr[15]$
				10	$MDRr[31]$
		11	4	00	$\star$
1	1	00	1	00	0
		01	2	00	0
		01	2	10	0



Control automaton

fetch (initial state)

Active signals: $fetch$, mr, $IRce$

→ *decode*

Active signals: Ace, Bce, $DPCce$, $PCPce$, $S1sel[0]$, $S2sel[1:0]$, add

→ $fetch$ if $illegal(IR)$

→ alu if $alu(IR)$

Active signals: $S1sel[1]$, , Cce, $Rtype$

→ $shift$ if $shift(IR)$

Active signals: $S1sel[1]$, , Cce, $Dsel$, $Rtype$

→ $shiftI$ if $shifti(IR)$

Active signals: $S1sel[1]$, $S2sel[0]$, Cce, $Dsel$

$\{shift, alu, shiftI\} \rightarrow wbR$

Active signals: $GPRw$, $Rtype$

→ $fetch$

→ $aluI$ if $alui(IR)$

Active signals: $S1sel[1]$, $S2sel[0]$, Cce

$aluI \rightarrow wbI$

Active signals: $GPRw$

→ $fetch$

→ $addr$ if $ls(IR)$

Active signals: $S1sel[1]$, $S2sel[0]$, add, $MARce$

→ $load$ if $\neg IR[29]$

Active signals: mr, $MDRrce$

→ $sh4l$

Active signals: $shift4l$, $GPRw$

→ $fetch$

→ $sh4s$ if $IR[29]$

Active signals: $S1sel[1:0]$, $Dsel$, $shift4s$, $MDRwce$

→ $store$

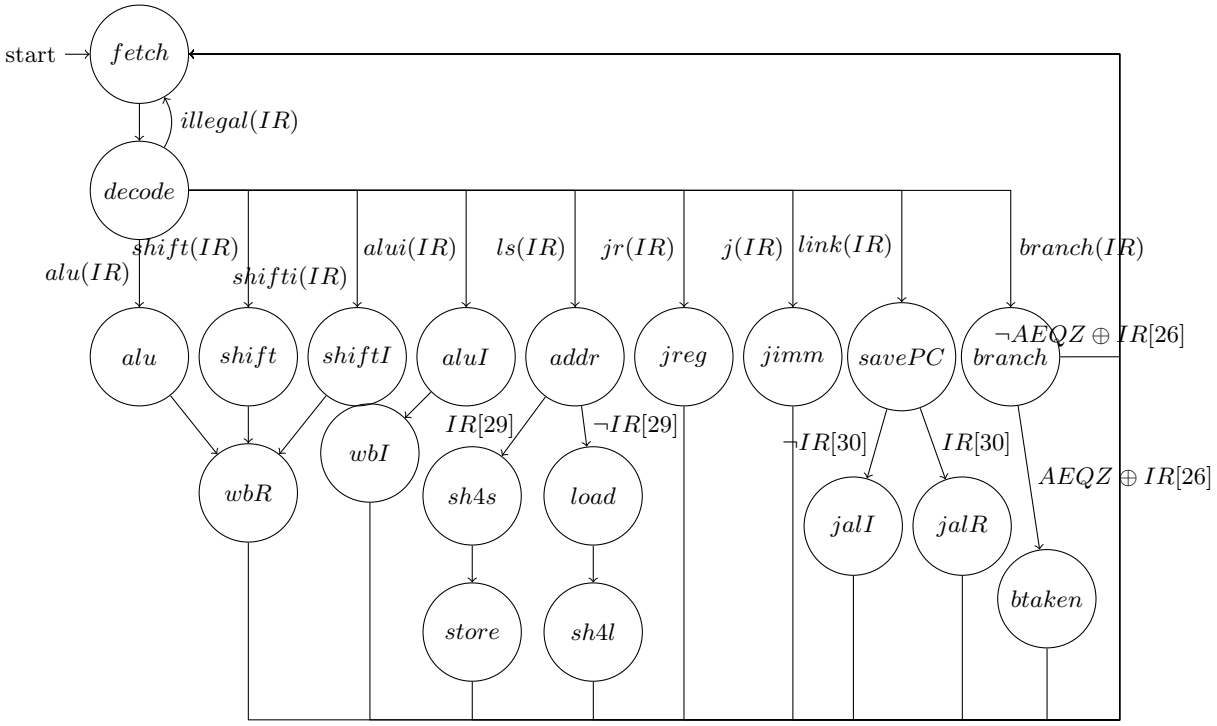
Active signals: mw

→ $fetch$

→ $jreg$ if $jr(IR)$

Active signals: $S1sel[1]$, $S2sel[1]$, add, $PCPce$

→ $fetch$
→ $savePC$ if $link(IR)$
Active signals: $S1sel[0]$, $S2sel[1]$, add, Cce
→ $jalR$ if $IR[30]$
Active signals: $S1sel[1]$, $S2sel[1]$, add, $PCPce$, $GPRw$, $Jlink$
→ $jalI$ if $\neg IR[30]$
Active signals: , $S2sel[0]$, add, $PCPce$, $GPRw$, $Jlink$
$\{jalR, jalI\} \rightarrow fetch$
→ $branch$ if $branch(IR)$
Active signals: none
→ $btaken$ if $AEQZ \oplus IR[26]$
Active signals: , $S2sel[0]$, add, $PCPce$
→ $fetch$
→ $fetch$ if $\neg(AEQZ \oplus IR[26])$
→ $jimm$ if $j(IR)$
Active signals: , $S2sel[0]$, add, $PCPce$, (not needed anymore: $Jjump$)
→ $fetch$



Lecture 10, 20 Nov 2006

Sequential Implementation
Correctness
Let h denote the hardware configurations
$h = (h.dpc, h.pcp, h.gpr, h.MB[3], \ldots, h.MB[0]$, (visible registers)

$h.IR, h.MAR, h.MDRr, h.MDRw, h.A, h.B, h.C, h.cs$) (invisible registers)

where $h.cs \in \mathbb{B}^{20}$ is the control state

We use abbrevations like $h.cs = fetch$ or $h.cs \neq wbR$

δ_{H} next state function hardware ($\delta_{\mathrm{H}}(h) = h'$)

h^0: initial configuration of the hardware (after a reset)

afterwards: no reset

$h^{t+1} = \delta_{\mathrm{H}}(h^t)$ for $t > 0$ (successor configuration)

h^t: hardware computation

Similarly for the architecture

c^0 initial configuration (after a reset)

c^i initial configuration (after a reset)

$c^{i+1} = \delta_{\mathrm{P}}(c^i)$ for $i > 0$ (successor configuration)

(Please forget about reset input of δ_{P})

Define simulation relation $sim(c, h)$ on components of c and *visible* registers in h

$sim(c, h) =$

$(c.dpc = h.DPC) \wedge (c.pcp = h.PCP) \wedge$

$(\forall a[4:0] : a \neq 0^5 \Rightarrow c.gpr(a) = h.gpr(a)) \wedge$

$(\forall a[31:0] : c.m(a) = h.MB[\langle a[1:0] \rangle](a[31:2]))$

TODO Picture: data embedding convention

Let $s(c) = \begin{cases} 2 & \text{if } illegal(I(c)) \\ 3 & \text{if } jr(I(c)) \vee j(I(c)) \vee (branch(I(c)) \wedge \neg btaken(c)) \\ 4 & \text{if } rtype(I(c)) \vee aluI(I(c)) \vee link(I(c)) \vee (branch(I(c)) \wedge btaken(c)) \\ 5 & \text{if } ls(I(c)) \end{cases}$

Theorem (correctness of the sequential processor):

If (TODO assumptions, see below) and $sim(c, h^t)$ and $h.cs = fetch$ then $sim(\delta_{\mathrm{P}}(c), \delta_{\mathrm{H}}^{s(c)}(h))$.

Proof:

By distinction over $I(c)$;

determine the path that is taken through the control automaton;

compute the updates of all implementation registers;

TODO example case: load instruction

Note for the proof: always works, *no* restrictions on the program

Software conditions (formulated on the c^i computation, since the programmer does not know the implementation!)

SC1: Instruction alignment: $\forall i : c^i.dpc[1:0] = 0^2$

SC2: Load / store alignment: $\forall i : ls(I(c^i)) \Rightarrow d(I(c^i)) \mid \langle ea(c^i) \rangle$

Chapter 5

Pipelined Implementation

Lecture 10, 20 Nov 2006 (cont.)

Pipelining
- Introduction
  - Decompose the hardware into stages:
    - Stage 1 Instruction Fetch (IF): load the instruction
    - Stage 2 Instruction Decode (ID): read source operands, compute control signals, compute next program counters
    - Stage 3 Execute (EX): execute ALU and shift instruction, compute the effective address of memory instructions
    - Stage 4 Memory (MEM): memory access for load / store instruction (do nothing else)
    - Stage 5 Write-Back (WB): write back the instruction result into the general-purpose register file
  - Perfect pipelining

	Time t						
Stage k	0	1	2	3	4	5	6
$IF = 0$	I_0	I_1	I_2	I_3	I_4	I_5	I_6
$ID = 1$		I_0	I_1	I_2	I_3	I_4	I_5
$EX = 2$			I_0	I_1	I_2	I_3	I_4
$MEM = 3$				I_0	I_1	I_2	I_3
$WB = 4$					I_0	I_1	I_2

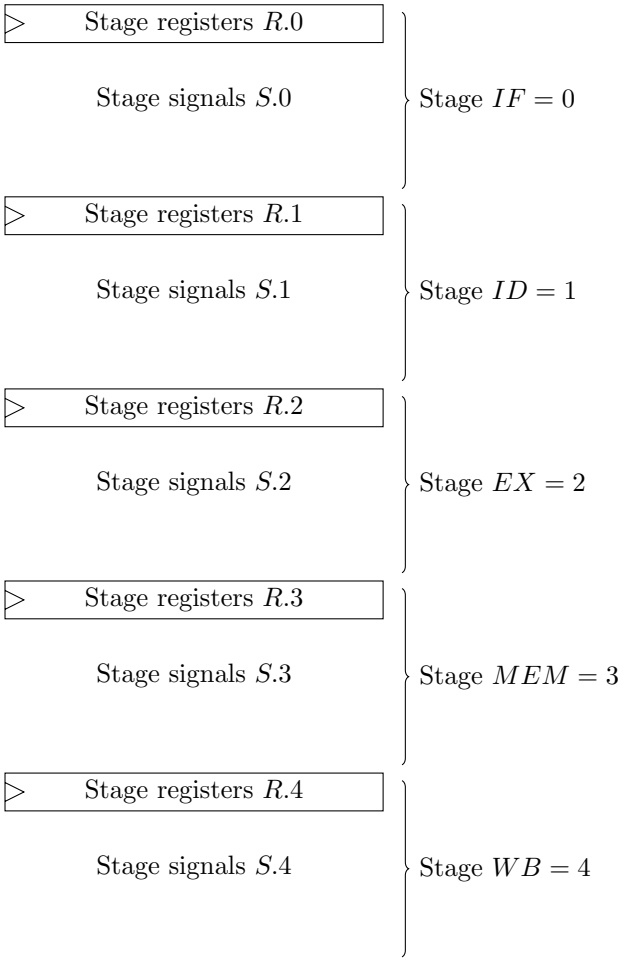
    with $I_i = I(c^i)$
  - If you look forward / below in the pipeline, you see the past
- Problems
  - Structural hazards
    - Hardware resources may not be used simultaneously by many instructions; must duplicate hardware
    - Example:
      - In the sequential processor, the ALU was used to
        compute the new PCs, compute the result of ALU instructions, pass the link target to the C register
      - Memory

- Use two memories, the data and the instruction memory
- Implement with an arbiter, that multiplexes the memory requests for a single memory (for now)
- Let's assume that arbiter has priority on data memory access
- Arbiter: similar to the exercises
- However, according to the architecture, this should be only a single memory;
- so this is really two ports to the same memory
- Problem: self-modification (a forwarding problem, see below)
- Example implementation (very inefficient): instruction memory port, data memory port with arbiter-with-priority from the exercises

Data hazards

- Problem (example)
  - $I_i =$ `add r3, r2, r1`
  - $I_{i+1} =$ `add r5, r3, r4`
  - Instruction I_{i+1} needs a results that is still being computed
- Solutions: see next lecture

Lecture 11, 22 Nov 2006



TODO literature notes?

Pipelining

- Introduction
  - Pipelining Picture (see above)
- Problems
  - Structural Hazards
    - ALU
    - Memories
      - Instruction and data memory (port)
      - Memory protocol: memory accesses are performed over several cycles, because:
        1. One memory access per cycles is an unrealistic assumption (memory clock is slower than processor clock)
        2. We will connect instruction and data memory port to the same memory (with priority on the data memory access);
           - therefore instruction memory accesses need to be delayed, if there is a simulatenous data memory access

(Note however: later on there will be caches, speeding up memory accesses)

For now: multiplexed, i.e., instruction and data memory are connected to the a single memory with the same protocol over an arbiter (with priority on the data memory)

TODO protocol handshake example picture

Interfaces:

One port for the instruction, one port for the data memory access
Handshake signals: request signal mr or mw for read and write; acknowledgment signals $busy$ (which is lowered if the access is complete)

—

Inputs: mr, mw, $din[31:0]$, $mwb[3:0]$
Outputs: $busy$, $dout[31:0]$
Inputs: mr, mw, $din[31:0]$, $mwb[3:0]$
Outputs: $busy$, $dout[31:0]$

—

Inputs: imr, imw, $idin[31:0]$, $imwb[3:0]$
Outputs: $ibusy$, $idout[31:0]$
Inputs: dmr, dmw, $ddin[31:0]$, $dmwb[3:0]$
Outputs: $dbusy$, $ddout[31:0]$

—

Will also use the following names for the instruction port (which only has to read):
Inputs: imr
Outputs: $ibusy$, $inst[31:0]$
Inputs: mr, mw, $din[31:0]$, $mwb[3:0]$
Outputs: $dbusy$, $ddout[31:0]$

—

Formalize protocol conditions (similar to the exercises)

Data Hazards

Occurs when data needed for an instruction is not available (not yet updated)

In our case:

Memory request (instruction or data memory) not yet acknowledged
Program counters (delayed PC guarantees that this hazards does not occur to often)
Source register

Example:

$I_i =$ `add r3, r2, r1`
$I_{i+1} =$ `add r5, r3, r4`

TODO definition register source 1 operand, register source 2 operand

Solutions / options:

1. Continue with possibly old data (infeasible with memory access)
   (Observation: may use 'forwarding', some instruction results are already computed prior to the write-back stage)
   Software conditions may be used to guarantee that old data is never seen
   Define $readsrs1(I) = \neg jtype(I)$
   Define $readsrs2(I) = \neg illegal(I)(rtype(I) \wedge \neg shifti(I) \vee store(I))$

(Store instructions read from $c.gpr(rd(I))$ and $store(I) \Rightarrow rd(I) = rs2(I)$)

(Note: it does not really hurt with respect to these software conditions, if these predicates are too strong, i.e., if they identify more reads than necessary)

Define $rwr(I) = \begin{cases} 1^5 & \text{if } link(I) \\ rd(I) & \text{otherwise} \end{cases}$

Define $writesrwr(I) = \neg illegal(I) \wedge \neg branch(I) \wedge \neg store(I) \wedge (\neg jump(I) \vee link(I))$

Attention:

Some pipeline implementation guarantee only a maximum delay in numbers of instruction until a new result is available, some guarantee an exact delay

In the first case, the ISA must be non-deterministic!

If software conditions are just assumed in the correctness proof, nothing is known about the processor if these assumptions are violated; only a 'partial' correctness theorem

Now for the software condition:

Between register update and read, there must be at least x instructions (for a small number x related to the pipeline depth of the processor)

Formally:

ISA computation $c^0, c^1, \ldots$

$\forall i : \forall i - x < i' < i :$

$(readsrs1(I(c^i)) \Rightarrow$

$(\neg writesrwr(I(c^{i'})) \vee (rwr(I(c^{i'})) \neq rs1(I(c^i)))))\wedge$

$(readsrs2(I(c^i)) \Rightarrow$

$(\neg writesrwr(I(c^{i'})) \vee (rwr(I(c^{i'})) \neq rs2(I(c^i)))))$

2. Hardware interlock / wait

Let hardware resolve the situations automatically (by inserting bubbles into the pipeline)

Consider the example with the two `add` instructions above

	Time						
Stage	t	$t+1$	$t+2$	$t+3$	$t+4$	$t+5$	$t+6$
0 (IF)	I_i	I_{i+1}	I_{i+2}	I_{i+2}	I_{i+2}	I_{i+2}	I_{i+3}
1 (ID)		I_i	I_{i+1}	I_{i+1}	I_{i+1}	I_{i+1}	I_{i+2}
2 (EX)			I_i				I_{i+1}
3 (MEM)				I_i			
4 (WB)					I_i		

Only after intruction I_i has written back its result into the register file, instruction I_{i+1} is allowed to continue its computation.

Thus, hardware interlock will automatically insert 'bubbles' into the pipeline.

With a technique called *forwarding*, the number of bubbles may be reduced.

Forwarding is based on the observation that the data to be computed by an instruction is often already available / computed in a stage prior to the write-back stage.

Hence, the data may also be forwarded from this stage at an earlier time.

In the example, data may be forwarded from the ALU output in time $t + 2$ so that no bubble at all is inserted into the pipeline.

3. Guess data and check later (speculation and rollback)
   speculation & rollback: detect the problem late, 'squash' the incorrect instructions and try again (roll-back)
   Similar technique needed for interrupt handling, cf. later this lecture

Stall engine and stall computation

Stall engine control the data / instruction flow through a pipelined machine

A stall engine consists of stages, there is one stall computation circuit per stage

Output to the stage above (the stage with the next lower index):
stallOut (cannot accept data / instruction from the stage above)

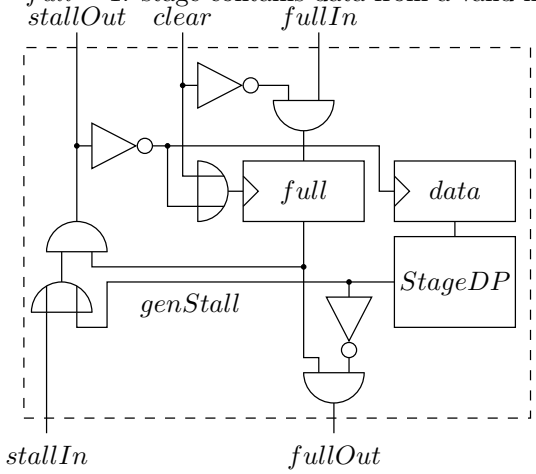
Input from the stage above (the stage with the next lower index):
$fullIn$ (stage above will deliver data / instruction if $\neg stallOut$)

Input from the stage below (the stage with the next higher index):
stallIn (stage below cannot accept data / instruction from this stage)

Output to the stage below (the stage with the next higher index):
$fullOut$ (will deliver new data / instruction to stage below if $\neg stallIn$)

Other inputs:
clear (flushs the stage; for now it is activated on reset)
genStall (local stall signal of the stage; stage is not able to deliver data)

The stall computation also maintains a state register $full$:
$full = 0$: stage contains a 'bubble'
$full = 1$: stage contains data from a valid instruction



Appendix A

Instruction Set

A.1 R-Type Instructions

$I[31:26]$	$I[5:0]$	Mnemonic	Effect
Shift Operation			
000000	000000	*slli*	$RD = RS1 \ll SA[4:0]$
000000	000010	*srli*	$RD = RS1 \gg SA[4:0]$
000000	000011	*srai*	$RD = RS1 \gg SA[4:0]$ (arithmetic shift)
000000	000100	*sll*	$RD = RS1 \ll RS[4:0]$
000000	000110	*srl*	$RD = RS1 \gg RS[4:0]$
000000	000111	*sra*	$RD = RS1 \gg RS[4:0]$ (arithmetic shift)
Arithmetic / Logical Operation			
000000	100001	*add*	$RD = RS1 +_{32} RS2$ (no overflow)
000000	100011	*sub*	$RD = RS1 -_{32} RS2$ (no overflow)
000000	100100	*and*	$RD = RS1 \wedge RS2$
000000	100101	*or*	$RD = RS1 \vee RS2$
000000	100110	*xor*	$RD = RS1 \oplus RS2$
000000	100110	*lhg*	$RD = (RS2, 0^{16})$
Test Set Operation			
000000	101000	*clr*	$RD = 0^{32}$
000000	101001	*sgr*	$RD = ([RS1] > [RS2]?0^{31}1 : 0^{32})$
000000	101010	*seq*	$RD = ([RS1] = [RS2]?0^{31}1 : 0^{32})$
000000	101011	*sge*	$RD = ([RS1] \geq [RS2]?0^{31}1 : 0^{32})$
000000	101100	*sls*	$RD = ([RS1] < [RS2]?0^{31}1 : 0^{32})$
000000	101101	*sne*	$RD = ([RS1] \neq [RS2]?0^{31}1 : 0^{32})$
000000	101110	*sle*	$RD = ([RS1] \leq [RS2]?0^{31}1 : 0^{32})$
000000	101111	*set*	$RD = 0^{31}1$

A.2 J-Type Instructions

$I[31:26]$	Mnemonic	Effect
Control Flow Operation		
000010	*j*	$PCP = \langle PCP \rangle +_{32} simm$
000011	*jal*	$R31 = \langle PCP \rangle +_{32} 4, \quad PCP = \langle PCP \rangle +_{32} simm$

A.3 I-Type Instructions

$I[31:26]$	Mnemonic	Effect
Load / Store, $mem[31:0] = m_4(ea)$		
100000	*lb*	$RD = (mem[7]^{25}, mem[6:0])$
100001	*lh*	$RD = (mem[15]^{17}, mem[14:0])$
100011	*lw*	$RD = mem[31:0]$
100100	*lbu*	$RD = (0^{24}, mem[7:0])$
100101	*lhu*	$RD = (0^{16}, mem[15:0])$
101000	*sb*	$mem[7:0] = RD[7:0]$
101001	*sh*	$mem[15:0] = RD[15:0]$
101011	*sw*	$mem[31:0] = RD$
Arithmetic / Logical Operation		
001001	*addi*	$\langle RD \rangle = RS1 +_{32} simm$ (no overflow)
001011	*subi*	$\langle RD \rangle = RS1 -_{32} simm$ (no overflow)
001100	*andi*	$RD = RS1 \wedge simm[31:0]$
001101	*ori*	$RD = RS1 \vee simm[31:0]$
001110	*xori*	$RD = RS1 \oplus simm[31:0]$
001110	*lhgi*	$RD = (simm[15:0], 0^{16})$
Test Set Operation		
011000	*clri*	$RD = 0^{32}$
011001	*sgri*	$RD = ([RS1] > [simm]?0^{31}1 : 0^{32})$
011010	*seqi*	$RD = ([RS1] = [simm]?0^{31}1 : 0^{32})$
011011	*sgei*	$RD = ([RS1] \geq [simm]?0^{31}1 : 0^{32})$
011100	*slsi*	$RD = ([RS1] < [simm]?0^{31}1 : 0^{32})$
011101	*snei*	$RD = ([RS1] \neq [simm]?0^{31}1 : 0^{32})$
011110	*slei*	$RD = ([RS1] \leq [simm]?0^{31}1 : 0^{32})$
011111	*seti*	$RD = 0^{31}1$
Control Flow Operation		
000100	*beqz*	$\langle PCP \rangle =_{2^{32}} \langle PCP \rangle + (RS = 0^{32}?[simm] : 0)$
000101	*bnez*	$\langle PCP \rangle =_{2^{32}} \langle PCP \rangle + (RS \neq 0^{32}?[simm] : 0)$
010110	*jr*	$PC = RS1$
010111	*jalr*	$\langle R31 \rangle =_{2^{32}} \langle PCP \rangle + 4, \quad PCP = RS1$

Appendix B

Changelog

This list describes the published versions of the lecture notes for this lecture. Also, changes of already published material are described from version to version.

Revision 1.17 Added notes for Lecture 1, 16 Oct 2006, and Lecture 2, 18 Oct 2006. This is the first published version.

Revision 1.19 Added notes for Lecture 3, 23 Oct 2006, and Lecture 4, 25 Oct 2006.

Revision 1.24 Added notes for Lecture 5, 30 Oct 2006 (note: 1 Nov 2006 was a holiday). Added and started maintaining (this) changelog. Started using a new XFig schematics library, resulting in layout changes of a couple of schematics.

Revision 1.30 Added notes for Lecture 6, 6 Nov 2006, and Lecture 7, 8 Nov 2006. Also, cost and delay formulae ($C()$ and $D()$) have been rewritten in a more consistent way.

Revision 1.38 Added notes for Lecture 8, 13 Nov 2006, and Lecture 9, 15 Nov 2006. Changed the predicates $alu(I)$ and $alui(I)$ to illegalize addition and subtraction with overflow.

Revision 1.50 Added notes for Lecture 10, 20 Nov 2006, and Lecture 11, 22 Nov 2006.

Correction for Lecture 8, 13 Nov 2006: $c.m : \mathbb{B}^{32} \rightarrow \mathbb{B}^{32}$ was corrected to $c.m : \mathbb{B}^{32} \rightarrow \mathbb{B}^{8}$ since we have byte-addressable memory in the ISA (thanks to Juan Alvarez and Christian Müller).

Correction for Lecture 10, 20 Nov 2006: the simulation relation $sim(c, h)$ should *not* include general-purpose register 0, since it is only specified that reads of the general-purpose register 0^5 will return 0^{32} but it is not specified that the GPR-RAM stores 0^{32} at address 0^5. Therefore, the condition $c.gpr = h.gpr$ was corrected to $\forall a[4:0] \neq 0^5 : c.gpr(a) = h.gpr(a)$.

Correction for Lecture 11, 22 Nov 2006: the definition of $writesrwr(I)$, 22 Nov 2006, was corrected not to include jump instructions that do not link.

Revision 1.55 Correction for Lecture 11, 22 Nov 2006: fixed the pipeline bubble example, which had an additional column.

Corrections for Lecture 9, 15 Nov 2006: in the states $jimm$, $branch$, and $jalI$ of the control automaton the active control signals where changed to select DPC as a left operand (instead of PCP as it was before). Corrected the table for the byte write signals: for the word write case $I[27:26]$ is equal to 11, not equal to 10 (thanks to Christian Müller).

Revision 1.56 More corrections for Lecture 9, 15 Nov 2006: in the state $savePC$ of the control automaton the active control signals where changed to select PCP as a left operand (instead of DPC as it was in Revision 1.55, originally it had been correct). In the table listing the cases for the SH4Lenv, the last three rows were corrected to have $I[28] = u(I)$ active (thanks to Christian Müller). Correctly listed the output of the shift environment as $r[31:0]$ not as $s[31:0]$ (thanks to Juan Alvarez). Removed the state wbL from the (textual) description of the control automaton (thanks to Juan Alvarez); write back the C register to the $GPRenv$ in the states $jalR$ and $jalI$ already.

In Lecture 10, 20 Nov 2006, corrected the claim for the simulation theorem. Added missing register $h.MAR$ to the invisible registers of the hardware configuration h of the sequential processor implementation (thanks to Juan Alvarez).

Revision 1.62 In Lecture 2, 18 Oct 2006, corrected the application of Proposition 5 in the proof for the subtraction of binary numbers (i.e., $\langle a\rangle - [1, \neg b] + 1$ now reads $\langle a\rangle + [1, \neg b] + 1$). Thanks to Michael Karl.

In Lecture 8, 13 Nov 2006, the mnemomic of the shift-left-logical-immediate instruction was corrected (from $slri$ to $srli$).

Revision 1.63 Fixed a couple of typos in Lectures 1 to 9. In particular, the proof for binary addition is hopefully fixed now. (Thanks to Christian Müller)

Bibliography

[1] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, San Mateo, CA, second edition, 1996.

[2] S. M. Mueller and W. J. Paul. *Computer Architecture: Complexity and Correctness*. Springer, 2000.