Diplomarbeit

Design and Evaluation of a Superscalar RISC Processor

Mark A. Hillebrand
(mah@wjpserver.cs.uni-sb.de)

Lehrstuhl Prof. Dr. W. J. Paul
Fachbereich Informatik
Universität des Saarlandes

April 2000

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, daß ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt und andere als die angegebenen Quellen und Hilfsmittel nicht benutzt habe.

Saarbrücken, den 5. April 2000

Contents

Chapter 1

Introduction

Processor design tries to increase the computing power of microprocessors by the advances in two fields of research. In the first, speed-up is achieved by physical advances in circuitry like reduced gate delay and an increased integration density or wafer size. In the second, refined or new algorithmic strategies are implemented decreasing the cycle-per-instruction (CPI) rate of a processor.

One approach in the second field that has been followed with considerable success is the development of superscalar processors. Such processors execute instructions in parallel while retaining the traditional sequential semantics of programs.

This thesis develops a superscalar processor on formal basis. Our superscalar processor falls into two parts: an instruction fetch mechanism and a superscalar DLX processor core implementing the Tomasulo algorithm. The instruction fetch mechanism loads instruction from the instruction memory and by sorting them in program order produces the so-called *instruction stream*. The processor takes the instruction stream and executes it as fast as possible. This combination of fetch mechanism and processor is extended by speculative execution (in the context of branch prediction) and by a precise interrupt mechanism.

1.1 Overview

The remaining part of this thesis is organized in five chapters:

- Chapter 2 develops the formal framework of our processor. It contains formal approaches and proofs of correctness of a superscalar Tomasulo algorithm, a precise roll-back mechanism and a (speculative) instruction fetch mechanism.

- Chapter 3 describes the implementation of the hardware of our processor design hierarchically descending down to gate-level. The hardware model used for the description is that of [MP95]. It provides a formal background and is easily evaluated.

- Chapter 4 examines the parameter space of the proposed processor design and compares variants of it to existing processor designs from [MP95, MP00, Krö99].

- Chapter 5 defines modules used in the machine description from chapter 3. These modules represent integral functional parts of the processor. Therefore their correctness is of great importance to the correctness of the whole processor design; formal criteria for correctness are specified and proven.

- Chapter 6 draws conclusions from the presented design, the formal approach and its evaluation. It also points out the fields that may be of interest for further research.

1.2 Notes on Reading

We assume that the reader is familiar with the concepts of circuit and processor design as developed in [KP95, MP95, MP00]. For the understanding of a non-superscalar DLX design implementing the Tomasulo algorithm, [Krö99] is of further help.

Chapter 2

Abstract Model

This chapter develops an abstract model for a superscalar hardware in three steps:

- Section 2.1 presents a *superscalar Tomasulo scheduling algorithm*. The algorithm handles the execution of *instruction streams*, sequences of instructions without control flow changes. This limitation is removed later on. The correctness of the algorithm is proven.

- Section 2.2 extends the Tomasulo algorithm by a precise roll-back mechanism. A precise roll-back mechanism stops the execution of an instruction stream at a specified instruction I_n; the instruction following I_n must not modify the state of the machine according to its sequential semantics. Precise roll-back is needed for the implementation of precise interrupts and speculative execution. The correctness of the roll-back mechanism is proven.

- Section 2.3 develops the concepts of an *instruction fetch mechanism* and of a *speculative instruction fetch mechanism*. These concepts are needed to provide the transition from instruction-stream based machines (that nobody builds) to real-life instruction-memory-based machines (with speculative execution). The speculation makes use of the precise roll-back mechanism to take back the effects of falsely executed instructions.

This chapter does not treat data memory and interrupts specifically. The exact manner of their treatment is often only defined by concrete system architectures. This is not the level of generalization we aim for in this chapter; nevertheless, in chapter 3 data memory and interrupts are treated in the environment of the DLX architecture.

2.1 Tomasulo Algorithm

The section describes a superscalar Tomasulo scheduling algorithm. It executes a dynamic instruction stream $I_1, I_2, \ldots$ out-of-order while preserving the sequential semantics.

Our approach is in three steps. First, we define the semantics of an *instruction stream* by a sequential instruction-stream-based machine IS_{seq}. Second, a superscalar Tomasulo machine TM is defined by description of the global structure and the scheduling protocols. In the third step, it is shown that TM simulates IS_{seq}. This is done by showing the data consistency theorem and the termination lemma.

2.1.1 Sequential Instruction-Stream-Based Machine

We define the model of an instruction-stream-based sequential machine, IS_{seq}. The machine IS_{seq} has $\#reg$ registers $\mathsf{R}_1, \ldots, \mathsf{R}_{\#\mathrm{reg}}$ over the finite domain DOM. These registers are referenced by the indices $\mathcal{R} := \{1, \ldots, \#reg\}$. Instructions are executed from the *instruction stream*. In cycle n the machine executes I_n, a tuple:

$$\mathsf{I}_\mathrm{n} \quad = \quad (\mathsf{op}, \quad \delta, \mathsf{dop}_1.\mathsf{A}, \ldots, \mathsf{dop}_\delta.\mathsf{A}, \quad \sigma, \mathsf{sop}_1.\mathsf{A}, \ldots, \mathsf{sop}_\sigma.\mathsf{A})$$

The function $\mathsf{op} : DOM^\sigma \longrightarrow DOM^\delta$ computes the results of the operation; δ is the number of destination operands and $\mathsf{dop}_1.\mathsf{A}, \ldots, \mathsf{dop}_\delta.\mathsf{A} \in \mathcal{R}$ are their identifiers (pairwise distinct); σ is the number of source operands and $\mathsf{sop}_1.\mathsf{A}, \ldots, \mathsf{sop}_\sigma.\mathsf{A} \in \mathcal{R}$ are their identifiers. With

$$(\mathsf{result}_1, \ldots, \mathsf{result}_\delta) \quad = \quad \mathrm{op}\,(\mathsf{R}_{\mathsf{sop}_1.\mathsf{A}}, \ldots, \mathsf{R}_{\mathsf{sop}_\sigma.\mathsf{A}})$$

and primed registers denoting "new", i.e. next-cycle values of registers, the semantics of the instruction I_n is defined as

$$\mathsf{R'}_\mathsf{A} \quad := \quad \begin{cases} \mathsf{result}_1 & \text{if } \mathsf{A} = \mathsf{dop}_1.\mathsf{A} \\ \mathsf{result}_2 & \text{if } \mathsf{A} = \mathsf{dop}_2.\mathsf{A} \\ \vdots & \vdots \\ \mathsf{result}_\delta & \text{if } \mathsf{A} = \mathsf{dop}_\delta.\mathsf{A} \\ \mathsf{R}_\mathsf{A} & \text{otherwise} \end{cases} .$$

Note, that a generalization on multiple, orthogonal destination operands ($\delta > 1$) is not used in real machines. Usually just one general destination operand and additional fixed-addressed destination operands (like operation flags) are sufficient. However, we allow $\delta \geq 1$ for two reasons: first, our approach to fetch mechanisms takes advantage of this fact, justifying the extra effort. Second, the simplifications used in machines having additional fixed-address destinations can be justified on the basis of algorithms for $\delta > 1$.

For ease notation, we consider σ and δ fixed for the instruction set of the machine.

A configuration of the machine stores (the values of) the registers. Instruction I_n is supposed to arrive in cycle n and is therefore not stored in the configuration.

2.1.2 Tomasulo Machine

Informal description of the Tomasulo algorithm

This algorithm was originally published by Tomasulo in [Tom67] in the year 1967. Written first for a very specific environment, this algorithm can be easily adopted for more general architectures. The algorithm associates each instruction and its destination registers on execution with a state-unique identifier, a small natural number, called tag.[1] On requesting a source register, an instruction either receives the (correct) value of this register, or the tag of a previous instruction computing it. On receiving a tag, the instruction has to wait until the result becomes available by a global result broadcast system. If all operands are gathered, an instruction may start execution. On completing execution, the instruction broadcasts tag and results in the whole machine for the benefit of instructions awaiting their source operands. Note that instructions awaiting execution need only to be connected to the global result broadcast system without taking up any other machine resources. A data structure, called reservation station, serves this purpose for a single instruction.

[1] State-unique means that in each cycle there is at most one instruction in execution holding the identifier. In the machine, the tags are *recycled* if not used; so the same tag may identify different instructions in different states.

Basic Data Structures and Paths of a Tomasulo Machine

The figure 2.1 shows the basic data structures and paths of a superscalar Tomasulo machine. The figures are annotated with the scheduling phases of an instruction.

The following components are present in a Tomasulo machine:

- The *instruction window* buffers incoming instruction in slots i_1 to i_β. Each round, the machine tries to start the execution of as many of these instructions as possible. According to the definition of the abstract machine, i_i is a tuple of source operand addresses $\mathsf{i}_\mathsf{i}.\mathsf{sop}_1$ to $\mathsf{i}_\mathsf{i}.\mathsf{sop}_\sigma$, an operation code $\mathsf{i}_\mathsf{i}.\mathsf{op}$ and destination operand addresses $\mathsf{i}_\mathsf{i}.\mathsf{dop}_1$ to $\mathsf{i}_\mathsf{i}.\mathsf{dop}_\delta$.

- The *register file* contains a tuple $(\mathsf{valid}, \mathsf{tag}, \mathsf{st}, \mathsf{data})$ for each register R_i. Two invariants will hold for the register file:

  If $\mathsf{R}_\mathsf{i}.\mathsf{valid} = 1$ then $\mathsf{R}_\mathsf{i}.\mathsf{data}$ contains the data of the last instruction writing to R_i up to the current machine cycle.

  If $\mathsf{R}_\mathsf{i}.\mathsf{valid} = 0$, then $\mathsf{R}_\mathsf{i}.\mathsf{tag}$ holds the tag of the *producing instruction*, i.e. the newest instruction having R_i as destination operand. The item $\mathsf{st} \in \{1, \ldots, \delta\}$ then specifies the index of the destination operand having address i.

- The *common data busses* are the global result broadcast system for the machine. Each common data bus bears a tuple $(\mathsf{tag}, \mathsf{result}_1, \ldots, \mathsf{result}_\delta)$ of a tag and the associated results.

- The *reorder buffer* contains for each instruction currently in execution a record, addressed by the instruction's tag, with the following items: the items $\mathsf{dop}_1.\mathsf{A}$ to $\mathsf{dop}_\delta.\mathsf{A}$ buffer the destination addresses of the instruction. The valid flag signals 1, if the instruction has already broadcast its results on a common data bus. In this case, the items $\mathsf{dop}_1.\mathsf{data}$ to $\mathsf{dop}_\delta.\mathsf{data}$ store the results. The reorder buffer writes results back to the register file in program order.

  The reorder buffer is organized as a simple wrap-around queue. The variable $\mathsf{ROB.head}$ points to the head of the queue, which is also the oldest instruction. The variable $\mathsf{ROB.tail}$ points to the next free entry of the queue, unless $\mathsf{ROB.tail} = \mathsf{ROB.head}$ which signals a full reorder buffer. The constant ROB-SIZE denotes the size of the reorder buffer.

- *Reservation station queues* are collections of buffers, called *reservation stations*, for instructions waiting for their source operands to be broadcast on the common data busses. A reservation station contains the following items: the item full indicates valid reservation station contents; the item op is an operation identifier; the tag of the instruction that the reservation holds is stored in the item tag; a tuple sop_σ for each source operand.

  The source operand tuple contains the correct operand data, if $\mathsf{valid} = 1$; otherwise it holds the tag and subtag of the producing instruction.

An instruction i_i is executed from the machine in the following way. On *issue* it is taken from the instruction window, associated with a tag from which the reorder buffer can reconstruct program order and put it an appropriate reservation station. Its source operands may either contain valid data or correct tag and subtag information. The instruction will remain in the reservation station until it has gathered all its missing source operands from the common data busses in a process called *snooping*. Having valid source operand data, the instruction will eventually leave its reservation station and be *dispatched* to its functional unit for *execution*. The functional unit computes the result busses and passes them via an interface called producer to one of the common data busses. This step is called *completion*. The

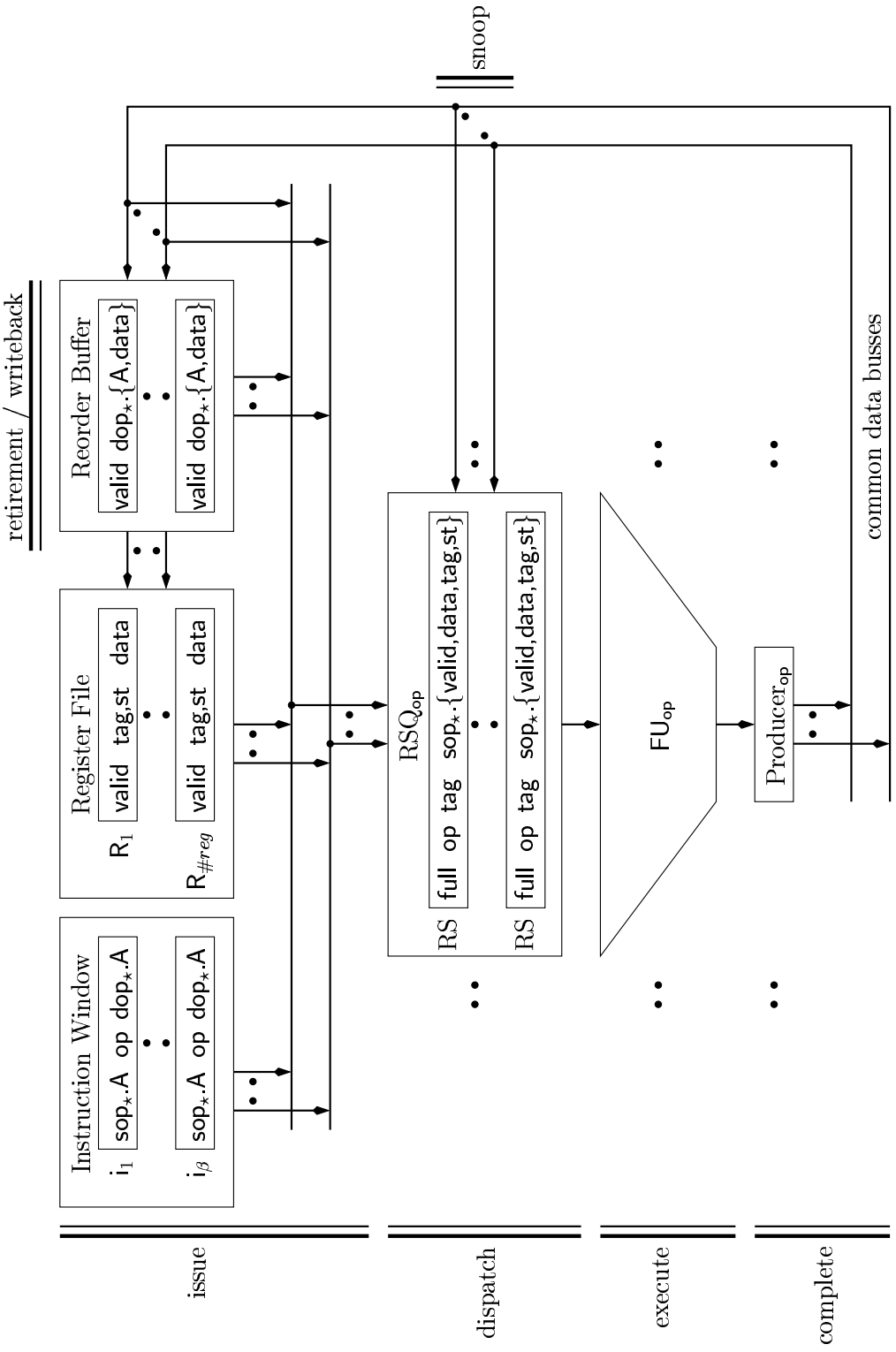

Figure 2.1: Basic data structures and paths of a Tomasulo machine

reorder buffer stores the result and will write it back to the register file in *program order*, i.e. after the results of all previous instructions have been written to the register file. This phase is called *retirement* of an instruction.

The exact procedure of the five phases issue, dispatch, snooping, completion and retirement is defined in protocols below. Basically, these protocols can be obtained by unrolling the non-superscalar Tomasulo algorithm handling only one instruction per cycle.

Issue

Superscalar issuing (algorithm 1) processes the instructions in the instruction window $\mathsf{i}_1, \ldots, \mathsf{i}_\beta$ with $\beta \in I\!N$, fixed, in a loop (l. 1) in program order. In line 2, a tag for the instruction i_i is obtained by adding index i to the tail pointer of the ROB. If the tag is not equal to the head pointer (i.e. the ROB is not full) and the appropriate reservation station queue for the instruction is not full, i_i may issue. The test of this condition is contained in line 3.

Instruction issue is divided in three phases: gathering the source operands (ll. 4–27), allocation of a free reservation station (ll. 28–31) and updates for the destination registers in the ROB and the register file (ll. 32–39).

Consider an iteration of the loop body of the source operand construction, ll. 4–27, and define S and R as in ll. 5–6. S is constructed according to five different cases:

First, S might be produced an accompanying instruction from the instruction window (ll. 7–11). This case is indicated if a destination operand's address of the previous instruction matches S.A. Naturally, the data is not available for the operand, and we only set the S.tag to the latest instruction's tag having destination S.A. The subtag denotes the index of the destination operand.

Second, the register R might be found valid in the register file (ll. 12–14). In this case we set the operand valid and take the data out of the register file.

Third, the operand might be broadcast on one of the common data busses (ll. 15–18). This is the case if the register's tag R.tag is found on one of the common data busses. The data can be taken from the result bus indicated by R.st and stored in the source operand.

Fourth, the operand's data may reside valid in the reorder buffer, waiting for its retirement (ll. 19–21). In this case, the data can be taken from $\mathsf{ROB}_{\mathsf{R.tag}}.\mathsf{data}_{\mathsf{R.st}}$.

With none of the previous cases meeting, the operand is still being computed in a functional unit and only its tag R.tag and subtag R.st is available (ll. 22–25).

The destined reservation station RS is filled with the instruction opcode, the destination tag, and the source operands (ll. 28–32). The notation of line 31 results in the assignment of all the items from all the source operands to all the reservation station source operands.

After gathering the source operands, the instruction has to update the register file and the ROB for each destination operand (ll. 32–39). Each destination register is invalidated, the tag information is set to the instruction's tag, the subtag information is set to the destination's index. The addresses of the destination register are stored in the ROB for further use.

In hardware, this protocol is parallelized to handle all the instructions in the instruction window simultaneously. Note the construction in lines 41–43 to stop issuing at the first instruction encountering a full reorder buffer or a non-available reservation station. On termination of the issue protocol, #issued contains the number of issued instructions.

For notation we define $\#\text{issued}^t$ as the value of #issued in cycle t. The same

```
1: for i ← 1 to β do
2:   i_i.tag ← (ROB.tail + i) mod ROBSIZE
3:   if i_i.valid ∧ (i_i.tag ≠ ROB.head) ∧ ¬RSQ_{i_i.op}.full then
4:     for all σ' ∈ {1, ..., σ} do                          (obtain source operands)
5:       Let S denote i_i.sop_{σ'}
6:       Let R denote R_{S.A}
7:       if ∃ i⁻ < i : ∃ δ' ∈ {1, ..., δ} : i_{i⁻}.dop_{δ'}.A = S.A then    (case: window)
8:         Let i⁻ be maximal with ∃ δ' ∈ {1, ..., δ} : i_{i⁻}.dop_{δ'}.A = S.A
9:         S.valid ← 0
10:        S.tag ← i_{i⁻}.tag
11:        S.st ← δ'
12:      else if R.valid then                               (case: register)
13:        S.valid ← 1
14:        S.data ← R.data
15:      else if ∃ k : CDB_k.tag = R.tag then               (case: snoop)
16:        Let k satisfy CDB_k.tag = R.tag
17:        S.valid ← 1
18:        S.data ← CDB_k.data_{R.st}
19:      else if ROB_{R.tag}.valid then                     (case: forwarding)
20:        S.valid ← 1
21:        S.data ← ROB_{R.tag}.data_{R.st}
22:      else                                               (case: tag)
23:        S.valid ← 0
24:        S.tag ← R.tag
25:        S.st ← R.st
26:      end if
27:    end for
28:    Let RS denote an empty reservation station in RSQ_{i_i.op}
29:    RS.full' ← 1
30:    RS.tag' ← i_i.tag
31:    RS.sop'_⋆.⋆ ← i_i.sop_⋆.⋆
32:    ROB_{i_i.valid}.valid ← 0
33:    for all δ' ∈ {1, ..., δ} do                          (update ROB, RF)
34:      Let D denote i_i.dop_{δ'}
35:      R_{D.A}.valid' ← 0
36:      R_{D.A}.tag' ← i_i.tag
37:      R_{D.A}.st' ← δ'
38:      ROB_{i_i.tag}.dop_{δ'}.A' ← D.A
39:    end for
40:    i ← i + 1
41:  else                                                   (could not issue i–th instruction)
42:    i ← i − 1
43:    Break out of loop, stop issuing
44:  end if
45: end for
46: ROB.tail' ← (ROB.tail + i) mod ROBSIZE
47: #issued ← i
```

Algorithm 1: Superscalar issue protocol

```
1:  for all reservation stations RS do
2:    for all σ' ∈ {1,...,σ} do
3:      Let S denote RS.sop_σ'
4:      if ¬S.valid ∧ ∃ k : CDB_k.tag = S.tag then          (snoop S on CDB_k)
5:        Let k satisfy CDB_k.tag = S.tag
6:        S.valid' ← 1
7:        S.data' ← CDB_k.data_S.st
8:      end if
9:    end for
10: end for
```

Algorithm 2: Superscalar common data bus snooping

```
1:  for all functional units FU_op do
2:    if ¬FU_op.stall ∧ ∃ j : RS_j.full ∧ ∀ σ' : RS_op,j.sop_σ'.valid = 1 then
3:      RS ← RS_op,j with j fairly selected
4:      FU_op.op' ← RS.op
5:      FU_op.tag' ← RS.tag
6:      for all σ' ∈ {1,...,σ} do                          (copy source operands into FU_op)
7:        FU_op.sop'_σ' ← RS.sop_σ'
8:      end for
9:      RS.full' ← 0
10:   end if
11: end for
```

Algorithm 3: Superscalar dispatch

notation defines

$$\mathcal{IW}^t \quad = \quad \left(\mathsf{i}_1{}^t, \ldots, \mathsf{i}_\beta{}^t\right)$$

as the instruction window presented to the machine in cycle t. The base b of this instruction window is the index of the first instruction in the window. Naturally the base can be computed by $b = 1 + \sum_{1 \le t' \le t} \#\text{issued}^{t'}$ and we have

$$\left(\mathsf{i}_1{}^t, \ldots, \mathsf{i}_\beta{}^t\right) \quad = \quad \left(\mathsf{I}_\mathsf{b}, \ldots, \mathsf{I}_{\mathsf{b}+\beta-1}\right)$$

Common Data Bus Snooping

Algorithm 2 defines the superscalar common data bus snooping protocol. The reservation stations snoop on the common data busses to gather their missing source operands' data. I.e., if a reservation station has a non–valid operand and the operand's tag is seen on a CDB (l. 4), then the data is taken from the result bus denoted by the subtag from this CDB and the operand is validated (ll. 5–7). While scanning multiple CDBs at once for an operand's tag, at most one matching CDB may be found because of uniqueness of tags in a single state.

Dispatch

Algorithm 3 shows the superscalar dispatch protocol. Dispatch of a full reservation station to the appropriate functional unit takes place, if all source operands are valid (l. 2). Fair candidate selection (for example by the age of the entries) is needed to ensure termination of instructions in a finite amount of time (l. 3). The reservation copies its operation code, tag and the source operands in the appropriate functional

```
 1: for all functional units FU_op do
 2:   if FU_op has result and got CDB_k-acknowledgement for next cycle then
 3:     Let T denote FU_op.tag
 4:     ROB_T.valid' ← 1
 5:     CDB_k.tag' ← T
 6:     for all δ' ∈ {1, ..., δ} do
 7:       CDB_k.data_δ' ← FU_op.result_δ'
 8:       ROB.dop_δ'.data' ← FU_op.result_δ'
 9:     end for
10:   end if
11: end for
```

Algorithm 4: Superscalar completion protocol

```
Require: register file updates have lower priority than from issue protocol
 1: for i ← 1 to ε do
 2:   r_i.tag ← (ROB.head + i) mod ROBSIZE
 3:   Let r_i denote ROB_r_i.tag
 4:   if (r_i.tag ≠ ROB.tail) ∧ r_i.valid then
 5:     for all δ' ∈ {1, ..., δ} do                          (update RF)
 6:       Let A denote r_i.dop_δ'.A
 7:       R_A.data' ← r_i.dop_δ'.data
 8:       if r_i.tag = R_A.tag then                  (validate register on tag equality)
 9:         R_A.valid' ← 1
10:       end if
11:     end for
12:   else                                  (no valid instructions left for retirement)
13:     i ← i − 1
14:     Break out of loop, stop retiring.
15:   end if
16: end for
17: ROB.head' ← (ROB.head + i) mod ROBSIZE
18: #retired ← i
```

Algorithm 5: Superscalar retirement

unit (ll. 4–8). After dispatch has taken place, the reservation station is marked as empty (l. 9).

Note, that dispatch to duplicates of functional units is only a simple extension to the protocol.

Completion

The completion protocol (algorithm 4) puts instruction results on a common data bus. If a results is avaible and a CDB has been acknowledged by the CDB bus control (l. 2) the following actions are performed: The associated ROB entry is marked valid in l. 4; the CDB tag field is set in l. 5. The results of the instruction are copied on the CDB and into the ROB (ll. 6–9).

Retirement

During retirement, completed instructions are taken in program order out of the reorder buffer and their results are stored in the register file. Without interrupts,

the reorder buffer is not a necessity. However, reordering instructions is required for the precise roll-back mechanism developed in section 2.2.

Algorithm 5 defines the retirement protocol. It exploits the queue organization of the reorder buffer, with the ROB head pointing to the oldest instruction present in the reorder buffer. The retirement protocol is defined in phases. The i–th phase attempts to retire instruction $\mathsf{r_i}$ located at position $\mathsf{ROB.head} + i$ mod ROBSIZE in the queue.

Instruction $\mathsf{r_i}$ may retire, if it still is a valid queue entry ($\mathsf{r_i.tag} \neq \mathsf{ROB.tail}$) and $\mathsf{r_i.valid} = 1$, i.e. the instruction has completed (l. 5).

All destination operands are written back into the register file (ll. 6–8). A tag comparison (l. 9) indicates, if the retiring instruction is still the producing instruction for register R. In this case, the register entry is validated (l. 10).

Note that the issue protocol and the retirement protocol concurrently write to the register file. These conflicts are resolved by giving the issue protocol a writing priority over the retirement protocol. The proof of correctness relies on this behaviour.

Like the issue protocol the retirement returns the number of processed instructions in a variable, #retired. Again, we define for notation $\#\text{retired}^t$ as the value of #retired in cycle t. The sequence

$$\mathcal{R}^t \quad = \quad \left(\mathsf{r}_1.\mathsf{tag}^t, \ldots, \mathsf{r}_{\#\text{retired}^t}.\mathsf{tag}\right)$$

denotes the tags of the retiring instructions in cycle t; $\mathsf{r_i.tag}$ is the tag of the instruction processed in the i–th retirement phase.

2.1.3 Correctness

This section proves the correctness of the superscalar Tomasulo algorithm in two steps. In the first step, data-consistency is shown: all instructions receive the correct source operands as defined by the (sequential) semantics. In the second step the termination of the protocols is examined: instructions take only a finite amount of time to process.

At first we define for notation some *partial* functions:

$$\begin{aligned}
\text{last}(A,n) &= \max\{n' < n \mid \exists\, \delta' : \mathsf{I}_{n'}.\mathsf{dop}_{\delta'}.\mathsf{A} = A\} \\
\text{st}(A,n) &= \delta' \quad \text{with } \mathsf{I}_n.\mathsf{dop}_{\delta'}.\mathsf{A} = A \\
\text{result}(A,n) &= \mathsf{I}_n.\mathsf{result}_{\delta'} \quad \text{with } \delta' = \text{st}(A,n) \\
\text{tag}(A,n) &= \mathsf{I}_n.\mathsf{tag} \quad \text{if } \exists\, \delta' : \mathsf{I}_{n'}.\mathsf{dop}_{\delta'}.\mathsf{A} = A \\
\text{idx}(tag,t) &= \max\{n' \mid \mathsf{I}_{n'}.\mathsf{tag} = tag \wedge \mathsf{I}_{n'} \text{ issued in cycle } t' \leq t\}
\end{aligned}$$

In words the notations read as follows: The index of the last instruction writing to $\mathsf{R_A}$ before instruction $\mathsf{I_n}$ is denoted by $\text{last}(A,n)$. The subtag, i.e. index of destination operand, of instruction $\mathsf{I_n}$ writing to R_A is denoted by $\text{st}(A,n)$. The result of an instruction's write to $\mathsf{R_A}$ is denoted by $\text{result}(A,n)$. The tag that has been given to instruction $\mathsf{I_n}$—if it writes $\mathsf{R_A}$—is denoted by $\text{tag}(A,n)$. Finally, $\text{idx}(tag,t)$ is the index n of the instruction lastly associated with the tag tag in cycle t.

The notation C^t for some component C denotes the value of this component at the beginning of cycle t.

Theorem 2.1 *Have a reservation station* RS *with* $\mathsf{RS.full}^t = 1$. *Let* n *be the index of the instruction that* RS *holds, i.e.* $n := \text{idx}(\mathsf{RS.tag}^t, t)$. *Then for all* $1 \leq \sigma' \leq \sigma \wedge A := \mathsf{I_n}.\mathsf{sop}_{\sigma'}.\mathsf{A}$:

$$\mathsf{RS.sop}_{\sigma'}.\mathsf{valid}^t = 0 \implies \mathsf{RS.sop}_{\sigma'}.\mathsf{tag}^t = \text{tag}(A, \text{last}(A,n))$$

$$\begin{array}{rcl} & & \mathsf{RS.sop}_{\sigma'}.\mathsf{st}^t = \mathrm{st}(A, \mathrm{last}(A,n)) \\ \mathsf{RS.sop}_{\sigma'}.\mathsf{valid}^t = 1 & \Longrightarrow & \mathsf{RS.sop}_{\sigma'}.\mathsf{data}^t = \mathrm{result}(A, \mathrm{last}(A,n)) \end{array}$$

Corollary 2.2 (Data Consistency) *If the functional units work correctly (and terminate), the superscalar Tomasulo algorithm is data consistent.*

Proof of Corollary 2.2. Once an instruction becomes dispatched, all its source operands are valid. By the theorem, all source operand data fields contain the result of the last instruction writing to them. The functional unit therefore produces the correct result.

Proof of Theorem 2.1. We simultaneously prove the claim and the following invariant:

> **Invariant 2.3** *Let A be a register address and b the base of the instruction window $\mathcal{IW}^t$ in cycle t. Then the following implications hold:*
>
> $$\begin{array}{rcl} \mathsf{R_A.valid}^t = 0 & \Longrightarrow & \mathsf{R_A.tag}^t = \mathrm{tag}(A, \mathrm{last}(A,b)) \\ & & \mathsf{R_A.st}^t = \mathrm{st}(A, \mathrm{last}(A,b)) \\ \mathsf{R_A.valid}^t = 1 & \Longrightarrow & \mathsf{R_A.data}^t = \mathrm{result}(A, \mathrm{last}(A,b)) \end{array}$$

The theorem is proved by induction over the cycles of the machine t. For the induction basis we have $t = 1$. Since an initialization condition of the machine is assumed, the reservation stations are empty and the registers are valid and zero-valued; the claims of the theorem and the invariant therefore hold trivially. Now assume that $t > 1$ and the invariant, the theorem and the corollary on data consistency hold for all $t' < t$. Let b be the base of $\mathcal{IW}^t$, $t^- := t - 1$ and b^- be the base of $\mathcal{IW}^{t^-}$.

First, the induction step for the invariant is shown. Define n^- maximal with the property $b^- \leq n^- < b \wedge \exists\, \delta' : \mathsf{I_{n^-}.dop}_{\delta'}.\mathsf{A} = A$. The following two cases must be examined:

- There was an issuing instruction with destination register R_A in cycle t^-. This is equivalent to n^- having a defined value. Then, obviously $\mathrm{last}(A,b) = n^-$. In cycle t^-, the issue protocol sets

$$\begin{array}{rcl} \mathsf{R_A.valid}^t & = & 0 \\ \mathsf{R_A.tag}^t & = & \mathrm{tag}\Big(A, b^- + \max\Big\{i \leq \#\mathrm{issued}^{t^-} \mid \exists\, \delta' : \mathsf{i_i.dop}_{\delta'}{}^{t^-} = A\Big\}\Big) \\ & = & \mathrm{tag}\big(A, n^-\big) \\ & = & \mathrm{tag}(A, \mathrm{last}(A,b)) \\ \mathsf{R_A.st}^t & = & \mathrm{st}\Big(A, b^- + \max\Big\{i \leq \#\mathrm{issued}^{t^-} \mid \exists\, \delta' : \mathsf{i_i.dop}_{\delta'}{}^{t^-} = A\Big\}\Big) \\ & = & \mathrm{st}\big(A, n^-\big) \\ & = & \mathrm{st}(A, \mathrm{last}(A,b)) \end{array}$$

  Note that in this case retiring instructions are (correctly) not taken into account, since the issue protocol has a higher write priority than the retirement protocol on the register file (cf. section 2.1.2).

- Now assume that we are not in the first case, i.e. there was no issuing instruction with destination register R_A in cycle t^-. Therefore:

$$\mathrm{last}(A,b) \quad = \quad \mathrm{last}\big(A, b^-\big)$$

Further assume, there was a retiring instruction in cycle t^- that matched tag with $\mathsf{R_A.tag}^{t^-}$; i.e.

$$\exists\, i \leq \#\text{retired}^{t^-} : \mathsf{r_i.tag} = \mathsf{R_A.tag}$$

There is at most one such i, because no two retiring instructions can have the same tag. The retirement protocol in cycle t^- correctly sets:

$$\begin{aligned} \mathsf{R_A.valid}^{t} &= 1 \\ \mathsf{R_A.data}^{t} &= \text{result}(A, \text{idx}(\mathsf{r.tag})) \\ &= \text{result}\big(A, \text{last}\big(A, b^-\big)\big) \\ &= \text{result}(A, \text{last}(A, b)) \end{aligned}$$

If there was no retiring instructions matching tag with $\mathsf{R_A.tag}$, the register, as well as $\text{last}(A, b^-)$ remaing unchanged and the induction assumption holds.

This shows the induction step for the invariant. For the induction step of the theorem two cases must be distinguished:

- $\mathsf{RS.full}^{t^-} = 0$. With the assumption of the theorem, $\mathsf{RS.full}^{t} = 1$, this means that an issue took place on the reservation station in cycle t^-.

  If $\mathsf{RS.sop}_{\sigma'}\mathsf{.valid}^{t} = 1$ the issue protocol has filled $\mathsf{RS.sop}_{\sigma'}\mathsf{.data}$ from the register file, from a common data bus CDB_k or from the reorder buffer. Because of the induction assumption on the invariant and on data consistency, each case results in:

$$\begin{aligned} \mathsf{RS.sop}_{\sigma'}\mathsf{.data} &= \begin{cases} \mathsf{CDB_k.dop}_{\delta'}\mathsf{.result} \\ \mathsf{ROB}_{\mathsf{R_A.tag}}\mathsf{.data} \\ \mathsf{R_A.data} \end{cases} \\ &= \text{result}(A, \text{last}(A, n)) \end{aligned}$$

  Otherwise, assume $\mathsf{RS.sop}_{\sigma'}\mathsf{.valid}^{t} = 0$. If there is an n^-, maximal, with the property $b^- \leq n^- < n \wedge \exists\, \delta' : \mathsf{I_{n'}.dop}_{\delta'}\mathsf{.A} = A$ the tag and the subtag were set by tag forwarding:

$$\begin{aligned} \mathsf{RS.sop}_{\delta'}\mathsf{.tag} &= \text{tag}\Big(A, b^- + \max\Big\{i \leq \#\text{issued}^{t^-} \mid \exists\, \delta' : \mathsf{i_i.dop}_{\delta'}{}^{t^-} = A\Big\}\Big) \\ &= \text{tag}\big(A, n^-\big) \\ &= \text{tag}(A, \text{last}(A, n)) \\ \mathsf{RS.sop}_{\delta'}\mathsf{.st} &= \text{st}\Big(A, b^- + \max\Big\{i \leq \#\text{issued}^{t^-} \mid \exists\, \delta' : \mathsf{i_i.dop}_{\delta'}{}^{t^-} = A\Big\}\Big) \\ &= \text{st}\big(A, n^-\big) \\ &= \text{st}(A, \text{last}(A, n)) \end{aligned}$$

  If there is no defined n^-, then then tag and subtag are taken out of the register file. By the induction assumption for the invariant:

$$\begin{aligned} \mathsf{RS.sop}_{\delta'}\mathsf{.tag} &= \mathsf{R_A.tag}^{t^-} \\ &= \text{tag}\big(\text{last}\big(A, b^-\big)\big) \\ &= \text{tag}(A, \text{last}(A, n)) \\ \mathsf{RS.sop}_{\delta'}\mathsf{.st} &= \mathsf{R_A.st}^{t^-} \\ &= \text{st}\big(\text{last}\big(A, b^-\big)\big) \\ &= \text{st}(A, \text{last}(A, n)) \end{aligned}$$

- $\mathsf{RS.full}^{t^-} = 1$. The reservation station was already full in the last cycle. If $\mathsf{RS.sop}_{\sigma'}.\mathsf{valid}^t = \mathsf{RS.sop}_{\sigma'}.\mathsf{valid}^{t^-}$, source operand $\mathsf{RS.sop}_{\sigma'}$ did not change its value in the preceding cycle and the claim holds by induction assumption.

  Otherwise, have $\mathsf{RS.sop}_{\sigma'}.\mathsf{valid}^{t^-} = 0 \wedge \mathsf{RS.sop}_{\sigma'}.\mathsf{valid}^t = 1$. This means that the reservation station snooped operand $\mathsf{RS.sop}_{\sigma'}$ in the last cycle on, say, $\mathsf{CDB_j}$. By induction assumption, have

$$\begin{aligned} \mathsf{CDB_j.tag}^{t^-} &= \mathsf{RS.sop}_{\sigma'}.\mathsf{tag}^{t^-} \\ &= \mathrm{tag}(A, \mathrm{last}(A, n)) \end{aligned}$$

  which implicates by the induction assumption on data consistency:

$$\begin{aligned} \mathsf{RS.sop}_{\sigma'}.\mathsf{data}^t &= \mathsf{CDB_j.result}^{t^-}_{\mathsf{RS.sop}_{\sigma'}.\mathsf{st}} \\ &= \mathrm{result}(A, \mathrm{last}(A, n)) \end{aligned}$$

Lemma 2.4 (Termination) *If CDB acknowledgements are fair and computations in functional units take up finite time, then the Tomasulo algorithm does not deadlock the machine.*

Proof. This lemma is also proven by induction over the instructions in the instruction stream.

We show that the first instruction terminates. For the first instruction, all register file items are valid, the reorder buffer and all the reservation stations are empty. Since the first instruction is also first in the instruction window, the instruction window case does not apply. Therefore, I_0 is issued to the appropriate reservation station with all operands valid according to the register case in the issue protocol. Dispatch takes place one cycle after issue. Since the functional unit takes only finite time to compute the result of the instruction and because of fair CDB acknowledgement, completion takes place. Retirement follows one cycle after completion, since I_0 occupies the ROB head.

Now assume that instructions $\mathsf{I}_0, \ldots, \mathsf{I}_{i-1}$ terminate. Issuing takes place if room is available in the ROB and the appropriate reservation station queue. Eventually, this is the case: according to the induction assumption $\mathsf{I}_0, \ldots, \mathsf{I}_{i-1}$ terminate and especially leave the reservation stations and the ROB. Having space, the issue protocol processes I_i. The issue protocol states that each instruction looks for its source operands at all places where results can be found. If a source operand is not valid immediately, it is snooped on the common data busses. Therefore and because of fair dispatch acknowledgements, the instruction dispatches. The functional unit executing I_i only takes finite time to compute the result of the instruction. Because of fair CDB acknowledgements, the instruction is completed in finite time. Retirement also takes finite time, since by induction assumption all the previous instructions retire.

2.2 Roll-Back

This section shows the implementation of a precise roll-back mechanism. A roll-back mechanism allows to suspend the execution of an instruction stream at a retiring instruction having tag $\mathsf{r_i.tag} \in \mathcal{R}^t$, a non-trivial operation on pipelined or superscalar machines. If $n := \mathrm{idx}(\mathsf{r_i.tag}, t)$, preciseness requires that after roll-back, the instructions $\mathsf{I}_1, \ldots, \mathsf{I}_n$ have been retired and the instructions $\mathsf{I}_{n+1}, \mathsf{I}_{n+2}, \ldots$ do not have any effect (this definition reminds of the definition of precise interrupts [Mül97, SP88]; it is formulated differently below).

A precise roll-back mechanism is an essential for support of precise interrupts and speculative execution. *Interrupts* force the machine to switch to the interrupt service routine's instruction stream on internal or external interrupt conditions. Preciseness of interrupts requires this switch to be made in well-defined and recoverable manner. *Speculative execution* needs to discard the instructions of a wrongly predicted instruction stream and to continue execution with the correct one.

This section proceeds in three steps. First, the formal definition of a precise roll-back for a superscalar machine is given. Second, precise roll-back support is incorporated in the retirement protocol. Third, we prove that the new protocol meets the formal requirements of a precise roll-back.

2.2.1 Definition of Precise Roll-Back

The semantics of a precise roll-back are defined with the help of two invariants on the semantics of a sequential machine.

Definition 2.5 (Sequential Configuration) *A configuration for the sequential machine IS_{seq} defined in section 2.1 is a tuple of the values of all its registers and the index n of the lastly executed instruction.*

Definition 2.6 (Projection on Sequential Configuration) *The projection of the Tomasulo machine TM after retirement phase i in cycle t is defined as*

$$\mathrm{pr}_i\left(C^t_{TM}\right) \quad = \quad \left(\mathrm{idx}\left(\mathsf{r_i.tag^t}, \mathsf{t}\right), \forall\, A : \mathsf{R_{i,A}.data'}\right)$$

where $\mathsf{R_{i,A}.data'}$ denotes the value of $\mathsf{R_A.data'}$ after the i–th retirement phase.

Definition 2.7 *A machine having a precise roll-back mechanism must fulfil the following two invariants:*

- *The* projection invariant *states that in each retirement phase, the register file contains the correct register data values for the instruction up to the retiring instruction:*

$$\forall\, 1 \leq i \leq \#retired^t, n{:=}\,\mathrm{idx}(\mathsf{r_i.tag}, t) \quad : \quad \mathrm{pr}_i\left(C^t_{TM}\right) = C^n_{seq}$$

- *The* halting invariant *states that on executing a roll-back after instruction $\mathsf{I_n}$ in cycle t and after retirement phase i ($n{:=}\,\mathrm{idx}(\mathsf{r_i.tag}, t)$) must not change the projected configuration on empty instruction stream:*

$$\forall\, t' > t, \mathsf{i_1.valid}^{t'} = 0 : \forall\, 1 \leq i' \leq \#retired^{t'} \quad : \quad \mathrm{pr}_{i'}\left(C^{t'}_{TM}\right) = C^n_{seq}$$

Remark 2.8 *The projection invariant ensures the projected state of the machine equal to the state of the sequential machine at* defined times. *The halting variant implicitly ensures that the machine does not do something "nasty" while it has been told to roll-back. As can be seen below, this criterion is easily verified for the Tomasulo algorithm.*

2.2.2 Roll-Back Retirement Protocol

To implement a precise roll-back mechanism, the retirement protocol, algorithm 5, has to be modified. Algorithm 6 shows the extensions, which may be inserted between line 12 and line 13 of the original algorithm 5.

The method implemented in our machine is called roll-back by *flushing*. The machine simply wipes away all the information of instructions still resident in the machine by clearing all RSQs, the ROB and the computations in functional units (ll. 2–8).

We call the machine using the extended retirement protocol TM_{rb}.

1: **if** perform roll-back? **then**
2: Update as follows with priority over the other protocols:
3: Set $\mathsf{RS.full}' \leftarrow 0$ for all reservation stations
4: Clear all computations in all functional units $\mathsf{FU_{op}}$
5: $\forall\, A : \mathsf{R_A.valid}' \leftarrow 1$
6: $\mathsf{ROB.head}' \leftarrow 0$
7: $\mathsf{ROB.tail}' \leftarrow 0$
8: Break out of loop (do not modify head)
9: **end if**

Algorithm 6: Extension to superscalar retirement to support precise roll-back

2.2.3 Proof of Preciseness

Lemma 2.9 *The machine TM_{rb} satisfies the projection invariant.*

We proof the projection invariant in the following form:

$$\forall\, t, 1 \leq i \leq \#\text{retired}^t, n{:=}\,\text{idx}(\mathsf{r_i.tag}, t)\,, A \quad : \quad \mathsf{R_A.data}' = \text{result}(A, \text{last}(A, n))$$

In words: in every retirement phase i each register's data contains the result of the last instruction writing to it.

The proof is by induction over the cycles t. Let $1 \leq i \leq \#\text{retired}^t$, $n{:=}\,\text{idx}(\mathsf{r_i.tag}, t)$. For $t = 1$ there is nothing to show, since no instructions retires and TM_{rb} and IS_{seq} are assumed to be initialized in the same way.

Let $t > 1$ and consider the claim true for any $t' < t$. Let A be a register address. If no instructions prior to and including the i–th retirements phase writes to R_A, the claim holds because of the induction assumption.

Otherwise, $\mathsf{R_A.data}'$ contains the data of the latest retirement writing to R_A. By data consistency and the queue order of the ROB, this is also the data of the last instruction writing R_A, so

$$\mathsf{R_A.data}' \quad = \quad \text{result}(A, \text{last}(A, n))$$

Lemma 2.10 *The machine TM_{rb} satisfies the halting invariant.*

Proof. This lemma is proven by taking a quick look at all the protocols. The issue protocol terminates directly on seeing $\mathsf{i_1.valid} = 0$ and does not change $\mathsf{ROB.tail}$. The snooping protocol and the dispatch protocol only operate on full reservation stations of which there are none. The functional units have been told to stop any computations, so the completion protocols does not find a result to forward to the reorder buffer. The retirement protocol terminates directly on $\mathsf{ROB.head} = \mathsf{ROB.tail}$ without changing $\mathsf{ROB.head}$.

2.3 Instruction Fetch and Speculation

2.3.1 Sequential Instruction-Memory-Based Machine

We define a sequential, instruction-memory-based machine, IM_{seq}. The definition of IM_{seq} is similar to the definition of IS_{seq} given in section 2.1.1. The machine IM_{seq} also has #reg *registers* $\mathsf{R}_1, \ldots, \mathsf{R}_{\#\text{reg}}$ over the finite domain DOM. Additionally, the machine has a special register, called the program counter register PC.

The $\mathsf{PC} \in \{1, \ldots, N\} \subseteq DOM$ for some $N \in I\!N$ determines the instructions that IM_{seq} executes from the instruction memory $\mathcal{IM} = (\mathsf{IM}_1, \ldots, \mathsf{IM}_N)$. In the n–th cycle IM_{seq} executes the instruction $I_n{:=}\,\mathsf{IM}_{\mathsf{PC}^n}$ with R^n denoting the value of

the register R in the n–th cycle. The sequence of instructions $I_1, I_2, \ldots$ is called the dynamic instruction stream, while the instructions IM_i stored in the instruction memory are called the static instructions. The instructions IM_i in the instruction memory are tuples of the following form:

$$\mathsf{IM_i} \quad = \quad (\mathrm{op}, \delta, \mathsf{dop_1.A}, \ldots, \mathsf{dop_\delta.A}, \sigma, \mathsf{sop_1.A}, \ldots, \mathsf{sop_\sigma.A})$$

If the set of register identifiers is extended to $\mathcal{R} := \{1, \ldots, \#reg\} \cup \{\mathsf{PC}\}$, the tuples are of the same type as in the instruction-stream-based machine.

The semantics of the regular registers is identical to the instruction-stream-based machine. With

$$(\mathsf{result_1}, \ldots, \mathsf{result_\delta}) \quad = \quad \mathrm{op}\,(\mathsf{R_{sop_1.A}}, \ldots, \mathsf{R_{sop_\sigma.A}})$$

the program counter is updated in the n–th cycle according to the following equations:

$$\mathsf{PC}' \quad := \quad \begin{cases} \mathsf{result_1} & \text{if } \mathsf{dop_1.A} = \mathsf{PC} \\ \mathsf{result_2} & \text{if } \mathsf{dop_2.A} = \mathsf{PC} \\ \vdots & \vdots \\ \mathsf{result_\delta} & \text{if } \mathsf{dop_\delta.A} = \mathsf{PC} \\ \mathsf{PC} + 1 & \text{otherwise} \end{cases}$$

As an initialization condition, we set $\mathsf{PC}^1 := 1$.

2.3.2 Non-Speculative Instruction Fetch

This section develops the notion of an instruction fetch mechanism. Informally speaking, an instruction fetch mechanism is a machine that constructs the dynamic instruction stream out of the static instructions. A machine simulating IM_{seq} naturally constructs the dynamic instruction stream at some time and may therefore be called an instruction fetch mechanism for IM_{seq}. What we are rather interested in are simple machines providing the dynamic instruction stream for execution on a Tomasulo machine.

Definition

We construct the Tomasulo machine TM', an extension of TM. The following three modifications are made:

- The instruction window's entries are extended by an additional item PC containing the program counter of the associated instruction. This information can be used by the machine, whenever PC is needed as a source operand. So, an instruction window entry i_i is a tuple

$$\mathsf{i_i} \quad = \quad (\mathsf{valid}, \mathsf{PC}, \quad \mathsf{op}, \delta, \mathsf{dop_1.A}, \ldots, \mathsf{dop_\delta.A}, \sigma, \mathsf{sop_1.A}, \ldots, \mathsf{sop_\sigma.A})$$

- The issue protocol allows read access to its variable #issued.
- The retirement protocol is modified to report all write accesses to the PC to an external interface. This modification allows for simple fetch mechanism, as the results of computations of the Tomasulo machine can be used.

The necessary protocol changes are trivial and not included here for the sake of brevity.

Now an instruction fetch mechanism is a machine that correctly constructs the dynamic instruction stream for execution with our Tomasulo machine TM' and without deadlock:

1: **loop**
2: Fetch a basic block starting at fetchPC
3: Wait for the CFI to execute
4: Set fetchPC to the reported result
5: **end loop**

Algorithm 7: Generic scalar instruction fetch mechanism

Definition 2.11 (Instruction Fetch Mechanism) *A machine IFM is called an instruction fetch mechanism for TM' if it constructs instruction windows $\mathcal{IW}^t$ satisfying the following property:*

> *For every n there is an index t such that the sequence of issued instruction and their PC of TM' in cooperation with IFM is a prefix of the sequence of issued instructions and their PC of IM_{seq}.*

In conclusion we find the following theorem:

Theorem 2.12 *An instruction fetch mechanism IFM and TM' in cooperation simulate IM_{seq}.*

Scalar Fetch Mechanism

We develop a simple fetch mechanism, called *scalar fetching* (cf. [Joh91]). The instructions are divided into two classes, the basic-block instructions (BBI) and the control flow instructions (CFI):

$$\begin{aligned} \mathsf{IM_i} \text{ is BBI} &:\Longleftrightarrow \quad \forall\, 1 \leq \delta' \leq \delta : \mathsf{IM_i.dop}_{\delta'} \neq \mathsf{PC} \\ \mathsf{IM_i} \text{ is CFI} &:\Longleftrightarrow \quad \exists\, 1 \leq \delta' \leq \delta : \mathsf{IM_i.dop}_{\delta'} = \mathsf{PC} \end{aligned}$$

Definition 2.13 *Let $i, l \in I\!N$. The finite instruction sequence $(\mathsf{IM_i}, \ldots, \mathsf{IM_{i+l-1}})$ with*

$$\mathsf{IM_{i+l}} \textit{ is CFI} \quad \wedge \quad \forall j \in \{i, \ldots, i+l-1\} : \mathsf{IM_j} \textit{ is BBI}$$

is called the basic block starting at i with length l. If $l = 0$ then the basic block starting at i is said to be empty.

The following lemma contains an elementary but significant observation about the semantics of our new machine:

Lemma 2.14 *Let $I_t = \mathsf{IM_{PC^t}}$. Let l be the length of the basic block starting at* PC. *Then:*

$$\forall\, l' \in \{0, \ldots, l\} \quad : \quad I_{n+l'} = \mathsf{IM_{PC^t+l'}}$$

Proof. BBIs inside a basic block do not modify the PC, so execution is sequential.

This way, the dynamic instruction sequence $\mathsf{I} = (\mathsf{I}_0, \mathsf{I}_1, \ldots)$ can be rewritten as an alternating sequence of basic blocks and CFIs:

$$\mathsf{I} \quad = \quad (\mathsf{bb}_0, \mathsf{cfi}_0, \mathsf{bb}_1, \mathsf{cfi}_1, \ldots)$$

In this equation, bb_i is the basic block starting at $\mathsf{PC}(\mathsf{cfi}_{i-1})$ and cfi_i is the CFI following bb_{i-1}. The basic blocks are possibly empty.

Lemma 2.14 justifies the generic scalar fetch mechanism for TM' shown in Algorithm 7. The fetch mechanism fetches a blocks of BBIs followed by a CFI in the

first step (line 2). After that it waits for the machine to execute the CFI (line 3). According to the communication protocol of the fetch mechanism and the machine, the machine notifies the fetch mechanism of the newly computed value for PC. This is used by the fetch mechanism to update its fetchPC in the third step (line 3). Traditional and especially non-superscalar fetch mechanisms are based on this algorithm.

2.3.3 Speculative Instruction Fetch

Scalar instruction fetch much too slow for superscalar machines. This is due to the small average basic block length of about 5 instructions. This means, while superscalar machines intend to process many instructions in parallel, they have wait for new instructions every few cycles due to scalar instruction fetch.

Today's microprocessors have the ability to execute instructions *speculatively*. This means that for each CFI they can guess a target PC prior to its computation. The speculation must be verified in the Tomasulo machine. It verification is successful, the machine may continue execution; otherwise the machine must roll back and proceed execution on the "right" execution path.

Definition

We proceed the same way as for non-speculative instruction fetch. First, we define a Tomasulo machine TM_{spec} supporting speculative execution. Then we define speculative instruction fetch mechanisms for TM_{spec}.

The machine TM_{spec} is based on TM_{rb} and TM' with the following modifications:

- The instruction window's entries are extended by additional entries bbi, cfi and nPC. The boolean variable bbi indicates if the fetch mechanism "believes" the associated instruction to be a basic-block instruction. The boolean variable cfi indicates if the fetch mechanism "believes" the associated instruction to be a control flow instruction. The entry nPC indicates the alleged PC after the execution of the instruction, i.e. the source PC for the following instruction. So, an instruction window entry i_i is a tuple

$$\begin{aligned}\mathsf{i}_\mathsf{i} \quad = \quad & (\mathsf{valid}, \mathsf{PC}, \mathsf{nPC}, \mathsf{bbi}, \mathsf{cfi}, \\ & \mathsf{op}, \delta, \mathsf{dop}_1.\mathsf{A}, \ldots, \mathsf{dop}_\delta.\mathsf{A}, \sigma, \mathsf{sop}_1.\mathsf{A}, \ldots, \mathsf{sop}_\sigma.\mathsf{A})\end{aligned}$$

- The issue protocol is modified to stop issuing on encountering an instruction that is not properly marked as BBI or CFI instruction. The fetch mechanism is notified of that condition. Furthermore, variable #issued is passed back to the fetch mechanism.

- The retirement protocol checks for every retiring instruction, if the speculated value of the next PC, nPC, matches the computed value of the next PC. If this is not the case, the machines notifies the fetch mechanism and performs a roll-back immediately *after* the offending instruction.

- Again, the retirement protocol is modified to report all write accesses to the PC to an external interface.

Algorithms 8 and 9 list the modifications necessary to the old algorithms 5 and 6. The remaining modifications again are considered straightforward and not listed here for the sake of brevity.

As can be seen, a speculative instruction fetch involves two levels of speculation.

1: **if** $\mathsf{i}_i.\mathsf{valid}$ and i_i has missing or wrong mark in CFI / BBI field **then**
2: Notify fetch mechanism.
3: $i \leftarrow i - 1$
4: Break out of loop, stop issuing
5: **end if**

Algorithm 8: Addition to the issue protocol

1: **if** $\exists\, \delta' \in \{1, \ldots, \delta\} \wedge \mathsf{r_i}.\mathsf{dop}_{\delta'}.\mathsf{A} = \mathsf{PC}$ **then**
2: **if** $\mathsf{r_i}.\mathsf{dop}_{\delta'}.\mathsf{result} \neq \mathsf{ROB_B}.\mathsf{nPC}$ **then**
3: Initiate roll-back. Notify fetch mechanism.
4: **else**
5: Notify fetch mechanism of verification.
6: **end if**
7: **end if**

Algorithm 9: Addition to the retirement protocol

1: **loop**
2: **if** retirement protocol detects misspeculation **then**
3: Set PC to the corrected version
4: Invalidate the instruction window
5: **else**
6: Fetch a basic-block
7: Guess a target for the CFI and modify PC
8: Annotate the instructions
9: **end if**
10: **end loop**

Algorithm 10: Generic speculative instruction fetch mechanism

First, the instruction fetch mechanism is allowed to speculate about the class of an instruction. This approach will not be followed in this thesis, although it is common for high performance fetch mechanism. For example, instruction fetch mechanisms might cache the class-information of instructions basicblock-wise, to avoid the decoding of instructions (cf. [Yeh93]).

Second, it speculates control flows, i.e. the irregular change of the PC.

Both speculations are verified and may cause a roll-back. In the first case, this is detected as early as in the issue phase. As the offending instruction is kept from issuing, no "real" roll-back is necessary. In the second case, the verification takes place in the retirement protocol. A misspeculation causes a roll-back taking back the effects of the instructions still in execution.

Now we define:

Definition 2.15 (Speculative Instruction Fetch) *A machine $SIFM$ is called an instruction fetch mechanism for TM_{spec} if it constructs instruction windows $\mathcal{IW}^t$ so that the following property holds:*

> *For every n there is an index t such that the sequence of retired instruction and their PC of TM_{spec} in cooperation with $SIFM$ is a prefix of the sequence of issued instructions and their PC of IM_{seq}.*

A Generic Speculative Instruction Fetch Mechanism

```
 1: if rollback to cfcPC signalled then
 2:   Remove all instruction fetch queues
 3:   Let Q be a new queue
 4:   fetchIFQ ← Q
 5:   issueIFQ ← Q
 6:   fetchPC ← cfcPC
 7: end if
 8: i ← 0
 9: for i ← 1 to α do
10:   if IM_fetchPC is CFI then
11:     Guess a result cfcPC for the update of the fetchPC register
12:     fetchIFQ.push ("valid CFI", PC = fetchPC, nPC = cfcPC)
13:     fetchPC ← cfcPC
14:     fetchIFQ ← new queue Q
15:     Break out of loop. Stop fetching.
16:   else
17:     fetchIFQ.push ("valid BBI", PC = fetchPC, nPC = fetchPC + 1)
18:     fetchPC ← fetchPC + 1
19:   end if
20: end for
21: i_⋆.⋆ ← issueIFQ.⋆
22: issueIFQ.drain (#issued)
23: if issueIFQ.empty ∧ (fetchIFQ ≠ issueIFQ) then
24:   issueIFQ ← next IFQ
25: end if
```

Algorithm 11: Queue-based speculative instruction fetch

Algorithm 10 shows the algorithm for a generic speculative instruction fetch mechanism. If the retirement protocol detects a misspeculation, it invalidates the instruction window and sets its internal PC to the corrected version (ll. 2–4). Otherwise it fetches a basic-block. Each basic-block ends in a CFI. The outcome the CFI is guessed and the PC is modified accordingly (ll. 6–8). The process repeats.

Queue-Based speculative instruction fetch

The generic speculative instruction is not of much use in real implementations. We present here a different algorithm closer to the implementation of chapter 3.

The algorithm maintains a list of queues, each of which will store a single basic block. The instruction fetch is decoupled from the instruction issue: it addresses the memory via the fetch PC, fetchPC, and appends instructions to a designated fetching queue until a CFI has been met. The PC computation of a CFI will be guessed and fetching continues in a new queue from the guess control flow change PC, cfcPC.

Instructions will be issued by draining them from a designated issuing IFQ. If the issuing IFQ is empty, issuing switches to the the next queue.

Initialization and rollback result in the clearing of all queues and starting all over with a new queue Q, which will be fetching and issuing IFQ.

Algorithm 11 shows the algorithm just described.

Chapter 3

Hardware

This chapter develops the hardware of a superscalar DLX processor implementing the Tomasulo algorithm. The basic structure of such a processor is in analogy to a non-superscalar version; [Krö99] develops a non-superscalar processor. Figure 3.1 on the following page shows an overview of the data paths of our processor design. The processor decomposes the execution of an instruction into five pipeline stages. In the first, the instruction fetch mechanism accesses the instruction memory, predicts control flow and puts instructions in the instruction window. From the instruction window, instructions are decoded and led on to the appropriate reservation station in the second stage. During decode and in the reservation stations the instructions gather their source operands, which may come out of the register file, the reorder buffer or from a common data bus. Instructions with available source operands are led to the third stage, the execution stage. This stage contains the functional units to compute the instruction result. The functional units may be pipelined to reduce cycle time. Computed results are put on a common data bus, in the fourth stage. The common data bus is snooped on by the reservation station as was described above, and the reorder buffer, also located in the completion stage, gathers all results in a buffer. The results leave the reorder buffer in program order to be written back in the register file in the fifth stage. The fifth stage therefore is called the write back stage.

The chapter proceeds with some notes on the notation, an introduction to a half-unary number format frequently used throughout this thesis and a detailed description of the pipeline stages.

3.1 Notation

3.1.1 Control Signals

Control signals are denoted by alphanumeric names set in a special font, like $\mathsf{test} \in \{0,1\}$. Control signals are not limited to bit signals, they may form multi-dimensional arrays. In this case, indices are attached to the signals according to their dimension. If we have, for example, an n-bit signal bus called data, decalred in notation by $\mathsf{data}_\star \in \{0,1\}^n$, we might reference it such:

data_i	bit i from data
$\mathsf{data}_{\mathsf{j..i}}$	bits j to i (in that order) from data
$\mathsf{data}_\star$	all the bits from data

The following example demonstrates the reference to a $m \times n$-bit signal array / matrix called ar, declared in notation by $\mathsf{ar}_{\star,\star} \in \{0,1\}^{m \cdot n}$. This signal array is said

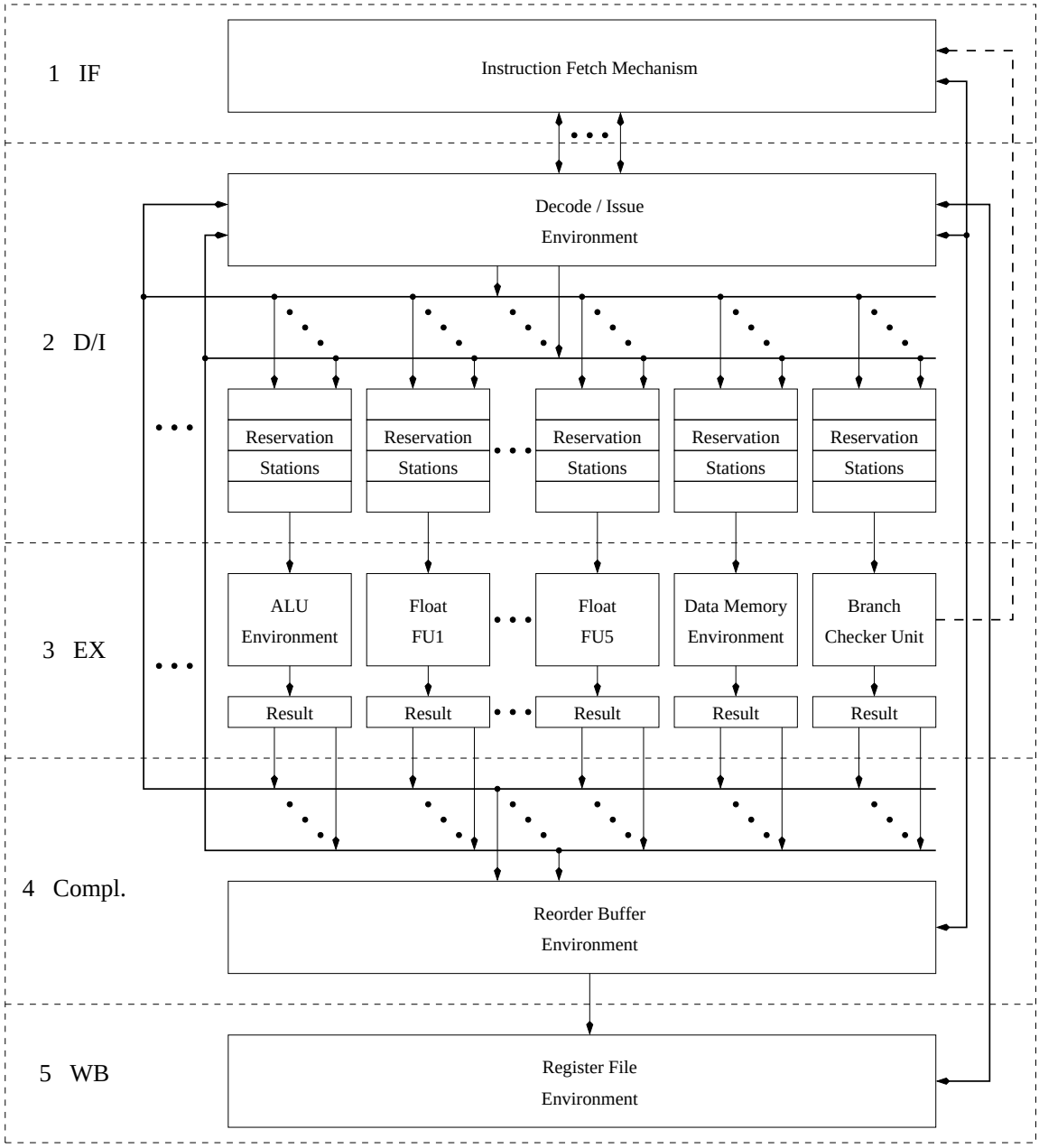

Figure 3.1: Data paths of the superscalar DLX processor

to have m rows and n columns:

$\mathsf{ar}_{\mathrm{i,j}}$	bit j from the i–th row of ar
$\mathsf{ar}_{\mathrm{i,j..k}}$	bits j to k from the i–th row of ar
$\mathsf{ar}_{\mathrm{i},\star}$	the i–th row of ar
$\mathsf{ar}_{\star,\mathrm{j}}$	the j–th column of ar
$\mathsf{ar}_{\star,\star}$	all the bits from ar

The star notation and the range notation is also frequently used in equations. It is a shortcut notation for using an all quantifier. The following sample notational equivalences hold for $\mathsf{a}_\star, \mathsf{b}_\star, \mathsf{c}_\star, \mathsf{s}_\star \in \{0,1\}^n$ and $\circ \in \{\wedge, \vee\}$:

$$\begin{aligned}
\mathsf{c}_\star = \mathsf{a}_\star &\iff \forall\, 0 \le n' < n : \mathsf{c}_{\mathrm{n}'} = \mathsf{a}_{\mathrm{n}'} \\
\mathsf{c}_\star = \overline{\mathsf{a}_\star} &\iff \forall\, 0 \le n' < n : \mathsf{c}_{\mathrm{n}'} = \overline{\mathsf{a}_{\mathrm{n}'}} \\
\mathsf{c}_\star = \mathsf{a}_\star \circ \mathsf{b}_\star &\iff \forall\, 0 \le n' < n : \mathsf{c}_{\mathrm{n}'} = \mathsf{a}_{\mathrm{n}'} \circ \mathsf{b}_{\mathrm{n}'} \\
\mathsf{c}_\star = (\mathsf{s}_\star\ ?\ \mathsf{a}_\star\ :\ \mathsf{b}_\star) &\iff \forall\, 0 \le n' < n : (\mathsf{s}_{\mathrm{n}'}\ ?\ \mathsf{a}_{\mathrm{n}'}\ :\ \mathsf{b}_{\mathrm{n}'})
\end{aligned}$$

Constructions like

$$\begin{aligned}
\mathsf{c}_{\mathrm{j..i}} &= \mathsf{a}_{(\mathrm{j+c})\cdot\cdot(\mathrm{i+c})} \\
\mathsf{c}_\star &= \mathsf{test} \wedge \mathsf{a}_\star \\
\mathsf{ar}_{\mathrm{i},\star} &= \mathsf{a}_\star
\end{aligned}$$

are also allowed if the signals are properly typed; again, inserting an all quantifier reveals their meaning.

The standard order for *writing signals* is from high to low. For example, the definition $\mathsf{b}_{3..0} := (1, \mathsf{a}_{3..2}, 0)$ results in

$$(\mathsf{b}_3, \mathsf{b}_2, \mathsf{b}_1, \mathsf{b}_0) := (1, \mathsf{a}_3, \mathsf{a}_2, 0)$$

3.1.2 Busses

Often it is convenient to identify a group of signals by a single name. In our notation such signals share the same prefix ending in a dot.

An instruction operand, for instance, might consist of a valid flag $\mathsf{op.flag} \in \{0,1\}$, a data bus $\mathsf{op.data}_\star \in \{0,1\}^{32}$ and a tag $\mathsf{op.tag}_\star \in \{0,1\}^5$. These three items form a bus, abbreviated by $\mathsf{op}.\star$.

Busses themselves might also be indexed to form vectors of busses. For example, having two instruction operands denoted by op_1 and op_2 we might reference such:

$\mathsf{op}_\star.\star$	both complete busses
$\mathsf{op}_\star.\mathsf{valid}$	the valid flags of both operands
$\mathsf{op}_\star.\mathsf{data}_\star$	the data items of both operands
$\mathsf{op}_\star.\mathsf{tag}_0$	bit 0 of the tags of both operands

If the context is clear, bus items are accessed without providing the bus prefix.

3.1.3 Figures

Figure 3.2 denotes the symbols used for gates in drawing circuits. All figures showing circuits are bounded by a dotted rectangle denoting the circuit's *interface*. Signals appearing left of or over the rectangle are *inputs* of the environment. Signals appearing right of or below the rectangle are *outputs* of the environment. Signals that

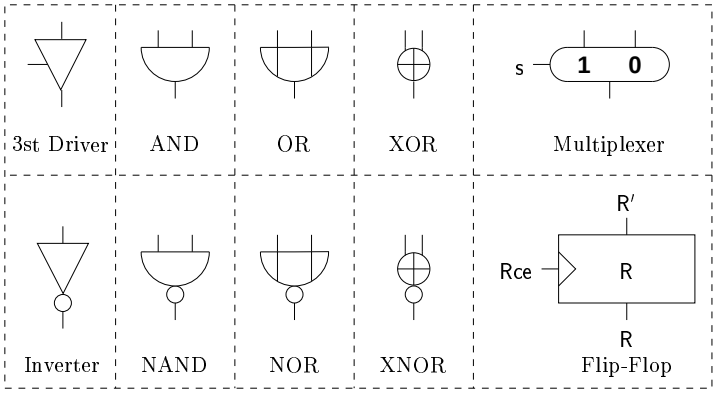

Figure 3.2: Symbols for the basic gates

have been named in a figure may be used by their name in other parts of the figure. Especially it is allowed to select single signals from an array or components of a bus by name.

The star notation and the range notation already defined will also be used in figures. For example, the signal definition $\mathsf{c}_\star := \mathsf{a}_\star \wedge \mathsf{b}_\star$ we will draw as a single AND gate having inputs $\mathsf{a}_\star$ and $\mathsf{b}_\star$ and the output $\mathsf{c}_\star$.

3.2 Half-Unary Number Format

This section defines the half-unary number format, which is frequently used in circuits throughout this thesis. The advantage of this format are threefold. First, the definition and proof of circuits, especially of such handling multidimensional signal arrays, is often greatly simplified by using the half-unary number format. Second, the half-unary format inherently encodes the information "not-equal-zero" and "greater-than-constant", which often comes in handy for control definitions. Third, a half-unary encoding can be converted in constant time to a unary encoding, an encoding requiring equivalent storage space. This does not hold for the other way round. The section proceeds with a definition of the half-unary format and a description of basic properties and circuits.

Let $n \in I\!N$. The sets

$$\begin{aligned} \mathsf{DOM}_{hu,n} &:= \{0,\ldots,n\} \\ \mathsf{BSTR}_{hu,n} &:= \left\{0^{n-i}1^i \mid 0 \le i \le n\right\} \end{aligned}$$

represent the domain and the valid bit strings of the n–bit half-unary encodings. Using this, we define the encoding function $\mathrm{enc}_{hu,n}$ and the decoding (or value) function $\langle\cdot\rangle_{hu,n}$ as follows:

$$\begin{array}{rrcl} \mathrm{enc}_{hu,n}: & \mathsf{DOM}_{hu,n} & \longrightarrow & \mathsf{BSTR}_{hu,n} \\ & c & \longmapsto & 0^{n-c}1^c \\ \langle\cdot\rangle_{hu,n}: & \mathsf{BSTR}_{hu,n} & \longrightarrow & \mathsf{DOM}_{hu,n} \\ & 0^{n-c}1^c & \longmapsto & c \end{array}$$

These functions are bijective and inversion functions of each other. Note that value 0 will be encoded as the zero bit string 0^n. If the index n is omitted from $\langle\cdot\rangle_{hu,n}$, it is determined by operand length.

Properties Have $\mathsf{a}_\star = 0^{n-c}1^c \in \mathsf{BSTR}_{hu,n}$. The following properties can be easily proven:[1]

$$\langle \mathsf{a}_\star \rangle_{hu} \geq i \iff \mathsf{a}_{i-1} = 1 \tag{3.1}$$

$$\langle \mathsf{a}_\star \rangle_{hu} \neq 0 \iff \mathsf{a}_0 = 1 \tag{3.2}$$

$$\langle \mathsf{a}_\star \rangle_{hu} \in \{i, i+1, \ldots, j\} \iff \mathsf{a}_{i-1} = 1 \wedge \mathsf{a}_j = 0 \tag{3.3}$$

$$\langle \mathsf{a}_\star, 1^i \rangle_{hu} = \langle \mathsf{a}_\star \rangle_{hu} + i \tag{3.4}$$

$$\langle 0^i, \mathsf{a}_{n-1..i} \rangle_{hu} = \max\{0, \langle \mathsf{a}_\star \rangle_{hu} - i\} \tag{3.5}$$

$$\langle \overline{\mathsf{a}_{0..n-1}} \rangle_{hu} = n - \langle \mathsf{a}_{n-1..0} \rangle_{hu} \tag{3.6}$$

$$\langle \mathrm{markLastOne}(\mathsf{a}_\star) \rangle_{un} = \langle \mathsf{a}_\star \rangle_{hu} \tag{3.7}$$

Remarks. The Properties 3.1 and 3.2 provide a simple means to compare half-unary encodings to constants and to 0 especially. Property 3.3 can be used to check an encoding against a given constant interval.

Properties 3.4, 3.5 and 3.6 are arithmetic properties. They justify the implementations of adders and subtracters given below.

Property 3.7 is frequently used for conversion of a half-unary encoding into a unary encoding. We define for notational convenience:

$$\mathrm{markLastOne}(\mathsf{a}_\star) := \text{the lower } n \text{ bits of } (0, \mathsf{a}_\star) \wedge \overline{(\mathsf{a}_\star, 0)}$$

The property then is explained by the fact that the markLastOne(·) function simply find the *edge*, i.e. the 0-1-transition, in the half-unary encoding $\mathsf{a}_\star$. The functions markLastZero(·), markFirstZero(·) and markFirstOne(·) are defined in analogy and will be also used in this thesis.

Conversion the other way round, i.e. from a unary encoding to a half-unary encoding, can be achieved by a parallel-prefix OR circuit or, preferably for delay reasons, by a find-first-one half-unary circuit $\mathrm{FF1hu}_n$ described in section C.1. Note that this conversion *inherently* requires logarithmic delay. This can be shown by a reduction of the conversion function to the 1-threshold function $(\mathrm{th1}(\mathsf{i}_\star) = 1 \iff \exists\, j : \mathsf{i}_j = 1)$.[2] A circuit for the 1-threshold function has cost $\Omega(n)$ (a proof of this classical result can found [Weg87]) and therefore, being a 1-output function, delay $\Omega(\log n)$.

Maximum and Minimum. The maximum operation on two half-unary encodings $\mathsf{a}_\star$ and $\mathsf{b}_\star$ can be computed in constant time by a slice of OR gates:

$$\mathsf{c}_\star := \mathsf{a}_\star \vee \mathsf{b}_\star \implies \mathsf{c}_\star \in \mathsf{BSTR}_{hu,n} \wedge \langle \mathsf{c}_\star \rangle_{hu} = \max\{\langle \mathsf{a}_\star \rangle_{hu}, \langle \mathsf{b}_\star \rangle_{hu}\} \tag{3.8}$$

A generalization of the maximum function on m arguments $\mathsf{a}_{0,\star}, \ldots, \mathsf{a}_{m-1,\star}$ needs logarithmic time by using trees of OR gates:

$$\mathsf{c}_\star := \bigvee_{0 \leq i < m} \mathsf{a}_{i,\star} \implies \mathsf{c}_\star \in \mathsf{BSTR}_{hu,n} \wedge \langle \mathsf{c}_\star \rangle_{hu} = \max_i \{\langle \mathsf{a}_{i,\star} \rangle_{hu}\} \tag{3.9}$$

Dually, the minimum operation on half-unary encodings can be computed using AND gates. We have:

$$\mathsf{c}_\star := \mathsf{a}_\star \wedge \mathsf{b}_\star \implies \mathsf{c}_\star \in \mathsf{BSTR}_{hu,n} \wedge \langle \mathsf{c}_\star \rangle_{hu} = \min\{\langle \mathsf{a}_\star \rangle_{hu}, \langle \mathsf{b}_\star \rangle_{hu}\} \tag{3.10}$$

$$\mathsf{c}_\star := \bigwedge_{0 \leq i < m} \mathsf{a}_{i,\star} \implies \mathsf{c}_\star \in \mathsf{BSTR}_{hu,n} \wedge \langle \mathsf{c}_\star \rangle_{hu} = \min_i \{\langle \mathsf{a}_{i,\star} \rangle_{hu}\} \tag{3.11}$$

[1] For illustration only, drawing half-unary encodings as a white box of width $(n-c)$ followed by a black box of width c has often proved useful.

[2] The lowest bit of the conversion function computes the 1-threshold function.

Relations. The equality relation "=" can be computed in logarithmic time using an equality tester, because we have a unique encoding of numbers. The less-than relation "<" can be computed in logarithmic time:

$$r := \bigvee_i \overline{\mathsf{a_i}} \wedge \mathsf{b_i} = 1 \quad \Longleftrightarrow \quad \langle \mathsf{a}_\star \rangle_{hu} < \langle \mathsf{b}_\star \rangle_{hu} \tag{3.12}$$

Addition and Subtraction. To compute the half-unary encoding representing the sum of the values of two other encodings, we use the following identity:

$$\langle \mathsf{a}_\star \rangle_{hu} + \langle \mathsf{b}_\star \rangle_{hu} \quad = \quad \max\{\langle \mathsf{a}_\star \rangle_{hu} + i \mid 0 \leq i \leq n \wedge \langle \mathsf{b}_\star \rangle_{hu} \geq i\} \tag{3.13}$$

This equation can be directly transformed in a recipe for building a half-unary adder rewriting it with equations 3.1, 3.4, 3.9 and formally setting $\mathsf{b}_{-1} := 1$:

$$\mathsf{c}_\star := \bigvee_{0 \leq i \leq n} \left((0^{2n-i}, \mathsf{a}_\star, 1^i) \wedge \mathsf{b}_{i-1} \right) \quad \Longrightarrow \quad \langle \mathsf{c}_\star \rangle_{hu} = \langle \mathsf{a}_\star \rangle_{hu} + \langle \mathsf{b}_\star \rangle_{hu} \tag{3.14}$$

For subtraction we use the negation property 3.6 and the adding equation 3.14. So, we have:

$$\mathsf{c}_\star := \bigvee_{0 \leq i \leq n} \left((0^{2n-i}, \mathsf{a}_\star, 1^i) \wedge \overline{\mathsf{b}_{n-i-1}} \right) \quad \Longrightarrow \quad \langle \mathsf{c}_\star \rangle_{hu} = n + \langle \mathsf{a}_\star \rangle_{hu} - \langle \mathsf{b}_\star \rangle_{hu} \tag{3.15}$$

Note that for $\mathsf{a}_\star$ and $\mathsf{b}_\star$ having different length it is advisable for delay reasons that $\mathsf{b}_\star$ is the shorter operand.

The Instruction Fetch Stage

3.3 Instruction Fetch Mechanism

3.3.1 Overview

This section describes the instruction fetch mechanism implemented in our processor. The fetch mechanism is a specifically tailored version of algorithm 11, p. 29, for the DLX architecture.

The fetch mechanism maintains two instruction buffers, called instruction fetch queues (IFQs). In each cycle one of these queues is designated the *fetching IFQ* and one, not necessarily different, queue is designated the *issuing IFQ*. The fetching IFQ receives newly fetched instructions and from the issuing IFQ fetched instruction are being issued.

Initially the fetching IFQ and the issuing IFQ are equal. The situation changes, when the control predicts or resolves a control flow change from a CFI in the issuing IFQ. Then, the fetching IFQ will be switched and fetched instruction will be stored in the alternative queue. The issuing IFQ follows this switch, if the instruction causing the control flow change has been issued.

On control flow speculation, the fetch mechanism goes in the outstanding speculation state. It will not predict or resolve further CFIs, until the *branch checker unit* (BCU) has positively or negatively verified the prediction. The report of the BCU causes the clearance of the outstanding speculation state. On a misprediction, the fetch mechanism has to roll back to the corrected target, involving initialization of the fetching and issuing IFQ and setting its fetch PC anew. For the special case, that a predicted taken branch is corrected to a fall-through branch, the alternative IFQ still contains the instructions following a branch; i.e. only a switch of IFQs without initialization is required.

```
1: if more than k free entries in fetchIFQ then
2:   Request k instructions from memory starting at fetchIFQ.PC
3:   Push the k instructions from the memory on fetchIFQ
4:   fetchIFQ.PC ← fetchIFQ.PC + k
5: end if
```

Algorithm 12: Implementation algorithm for instruction fetch

```
1: if rollback then
2:   if JISR then
3:     Switch fetchIFQ
4:     Initialize fetchIFQ
5:     Set issueIFQ to fetchIFQ
6:     fetchIFQ.PC ← SISR
7:   else
8:     IWdisabled ← 0
9:   end if
10: end if
```

Algorithm 13: Implementation algorithm for rollback

Exceptions may also interrupt regular instruction fetch. Similar to the treatment of mispredictions, fetching and issuing IFQ are switched and cleared and the fetch PC is set to the start of the interrupt service routine.

Buffering instructions in prefetch queues is a common approach in modern microprocessors ([Bha96, WS94]). The alternating prefetch queue design can also be found in the Pentium processor, although [AS95] does not provide an in-depth discussion.

3.3.2 Algorithms

The details of the instruction fetch mechanism are given in five algorithms dealing with the fetch of instructions, the misprediction treatment, the interrupt treatment, the prediction of control flow and the construction of the instruction window. Each round, the algorithms are executed in that sequence.

Algorithm 12 shows the implementation algorithm for instruction fetch. Instruction fetch is made in blocks of k subsequent instructions. Therefore it stalls, if the fetching IFQ has no room for k instructions left (l. 1). The instruction requested and fetched are pushed in the fetching IFQ (ll. 2–3). After this, the fetch PC advances by k steps (l. 4).

Algorithm 13 shows the implementation algorithm for the treatment of rollback. Rollback is initiated by the global signal rollback (l. 1), generated during retirement. In case that JISR is activated in addition (l. 2), the fetch IFQ is switched and initialized (ll. 3–4), followed by the issuing IFQ (l. 5). The interrupt service routine is called (l. 6). If rollback is activated but not JISR (l. 7), a mispredicted CFI was caught retiring. Since in this case the branch checker unit will have reported a misprediction before, the control flow has already been changed adequately by the misprediction implementation algorithm described below. Only the instruction window, temporarily disabled for issuing, has to be enabled again by zeroeing the status variable IWdisabled (l. 8).

Algorithm 14 on the next page shows the implementation algorithm for the treatment of reports from the branch checker unit. The branch checker unit verifies predictions made by the fetch mechanism. Its implementation is described in detail in section 3.11.4. For the moment, it is sufficient to present the branch checker unit

Component	Width	Purpose
valid	1	indicates valid bus contents
mp	1	indicates misprediction
jump	1	indicates jump instruction
btaken	1	indicates taken branch
$\mathsf{cfcPC}_\star$	32	corrected CFC address

Table 3.1: Branch checker unit bus

```
1:  if bcu.valid then
2:    oss ← 0
3:    if bcu.mp then
4:      Switch fetchIFQ
5:      Set issueIFQ to fetchIFQ
6:      if bcu.jump ∨ ¬bcu.btaken then
7:        Initialize fetchIFQ
8:        fetchIFQ.PC ← bcu.cfcPC
9:      end if
10:     IWdisabled ← 1
11:   end if
12: end if
```

Algorithm 14: Implementation algorithm for BCU treatment

interface. The BCU reports its verifications on a bus $\mathsf{bcu}.\star$. Table 3.1 lists the components. Signal bcu.valid indicates that a verification took place. On a misprediction, $\mathsf{bcu.mp} = 1$, two cases are distinguished: for a jump instruction, the corrected PC $\mathsf{cfcPC}_\star$; for a branch instruction, btaken indicates the correct outcome, $\mathsf{cfcPC}_\star$ will also contain the correct PC.

We continue the description of algorithm 14. The algorithm first checks if the BCU reports a valid bus (l. 1). Any report, be good or ill, results in clearance of the outstanding speculation state (l. 2). A misprediction of control flow results in the initiation of a rollback (ll. 3–11). The fetch IFQ is switched (l. 4), the issuing IFQ follows (l. 5). If the mispredicted CFI was a branch and has been falsely taken, fetchIFQ still holds the instruction of the fall-through target and need not be initialized. Otherwise, the (new) fetch IFQ is initialized (l. 7) and its PC is set to the corrected address, as reported by the BCU (l. 8). Note that branch checking is done out of order to speed up the machine: fetch of the correct control flow may start as soon as the misprediction is detected, though the execution of the correct control flow start after machine rollback. Mispredicted CFIs are therefore passed on to retirement and will generate a regular rollback call there if the control flow has not been interrupted by a previous interrupt condition. Instruction issue is disabled by setting IWdisabled until then (l. 10). The rollback algorithm will enable the instruction window again, as has already been seen above.

Algorithm 15 shows the implementation algorithm used for prediction and resolving of CFIs. The issuing IFQ is scanned for CFIs in case that the machine is not in an outstanding speculation situation and its instruction window is not disabled (l. 1). The first found CFI is renamed to cfi for ease of notation (l. 2). If cfi can be resolved or predicted, the algorithm will do so in ll. 4–17. For a branch, the intermediary variable cfi.cfcPC is set to the relative jump target (l. 5) and the variable cfi.cfc, indicating a control flow change, is set to the predicted or resolved outcome. Otherwise, the CFI is a jump instruction. In this case, cfi.cfcPC is set to the predicted or resolved target (l. 8) and cfi.cfc is set (l. 9) since the DLX jump instruction

```
 1: if ¬oss ∧ ¬lWdisabled ∧ ∃ j : issueIFQ_j is CFI then
 2:   Let cfi denote issueIFQ_j, j minimal s.t. issueIFQ_j is CFI
 3:   if can predict / resolve cfi then
 4:     if cfi is branch then
 5:       cfi.cfcPC ← cfi.PC + cfi.imm16
 6:       cfi.cfc ← (predicted / resolved taken ?1 : 0)
 7:     else
 8:       cfi.cfcPC ← predicted / resolved target
 9:       cfi.cfc ← 1
10:     end if
11:     On prediction only: oss ← 1
12:     if cfi.cfc then
13:       Switch fetchIFQ and initialize
14:       fetchIFQ.PC ← cfi.cfcPC
15:     end if
16:     cfi4l.valid ← 1
17:     cfi4l.cfc ← cfi.cfc
18:     cfiReady ← 1
19:   end if
20: end if
```

Algorithm 15: Implementation algorithm for prediction and resolving

```
 1: if ¬lWdisabled then
 2:   Set issueIFQ_j.valid ← 1 for all j
 3:   if ∃ j : issueIFQ_j is CFI then
 4:     Let j be minimal s.t. issueIFQ_j is CFI
 5:     Set issueIFQ_j'.bbi ← 1, issueIFQ_j'.cfi ← 0 for all j' < j
 6:     Set issueIFQ_j.bbi ← 0, issueIFQ_j.cfi ← cfi4l.valid
 7:     Set issueIFQ_j'.bbi ← 0, issueIFQ_j'.cfi ← 0 for all j' > j
 8:   else
 9:     Set issueIFQ_j.bbi ← 1 for all j
10:   end if
11: else
12:   Set issueIFQ_j.valid ← 0 for all j
13: end if
14: Wait for issue
15: Drain the issued instructions from the issuing IFQ.
16: if ∃ j : issueIFQ_j issued ∧ issuedIFQ[j].cfi then
17:   cfi4l.valid ← 0
18:   if cfi4l.cfc then
19:     Switch the issuing IFQ
20:   end if
21: end if
```

Algorithm 16: Implementation algorithm for instruction window construction

always cause a control flow change. For a prediction, the fetch mechanism sets the outstanding speculation state, preventing further predictions until the BCU signals verification (l. 11). In case of control flow change, predicted or not, the fetching IFQ is switched and its PC is set to the cfi.cfcPC variable (ll. 13–14). Finally, setting cfi4l.valid indicates that a CFI awaits issuing (l. 16). Variable cfi4l.cfc indicates that the control flow breaks at this instruction (l. 17).

Name	Width	Description
$\mathsf{IFQ_i.PC}_\star$	32	fetch PC for $\mathsf{IFQ_i}$, $i \in \{0,1\}$
$\mathsf{cfcPC}_\star$	32	control flow change PC buffer
fetchIFQ.idx	1	index of the fetching IFQ
issueIFQ.idx	1	index of the issuing IFQ
oss	1	outstanding speculation
IWdisabled	1	disabled instruction window
cfi4I.valid	1	indicate that CFI is ready to be issued
cfi4I.cfc	1	indicates that next CFI causes a CFC
$\mathsf{P}_\star$	7	pending action

Table 3.2: Registers in the instruction fetch stage

Algorithm 16 presents the procedure to construct the instruction window for the Tomasulo machine core. In case that the instruction window is not disabled (l. 1), the instructions in the issuing IFQ are set valid. First assume that a CFI is present in the issuing IFQ (l. 3–7). Basicblock instructions before the first CFI are marked as such (l. 5). The CFI will be marked as non-bbi; it is marked as CFI iff it is ready for issue, cfi4I.valid = 1 (l. 6). The following instructions are marked neither BBI or CFI, so that the Tomasulo core will not issue them (l. 7). If no CFI exists, all the instructions stored in the queue are basicblock instruction and marked appropriately (l. 9). A disabled instruction window will be all set to invalid instructions (l. 12).

After this preparation, instruction issue can take place (l. 14). The issued instructions will be drained from the issuing IFQ (l. 15). If these instructions included a CFI (l. 16), the cfi4I.valid variable is cleared (l. 17), and the issuing IFQ is switched if the cfi involved a control flow change (ll. 18–20).

3.3.3 Control

In hardware, the implementation algorithms are parallelized. The execution of the algorithms is subject to the conventions of the memory protocol. As this protocol forbids the interruption of read requests, the control flow can only be changed after completion of the previous read request. So, the control must reflect that the processor cycles are superimposed by the "memory cycles" (as indicated by the IM.busy signal).

Registers

We proceed with the description of the registers that the instruction fetch stage is maintaining. Most of these register appeared already as variables in the algorithms. Table 3.2 shows an overview; details follow:

- Registers $\mathsf{IFQ_0.PC}_\star$ and $\mathsf{IFQ_1.PC}_\star$ store a fetch PC of $\mathsf{IFQ_0}$ and $\mathsf{IFQ_1}$ respectively. Register $\mathsf{cfcPC}_\star$ is the scrapbook register for buffering the PC of control flow changes.

  Update of these three register is performed in the PC environment, described in section 3.6.

- Registers fetchIFQ.idx and issueIFQ.idx store the index of the fetching and of the issuing IFQ respectively.

  Updates of these registers are controlled by three signals. Signal fIFQswitch and iIFQswitch indicates the toggling of fetchIFQ.idx and issueIFQ.idx. Signal

#	Name	Detection	Active Control Signals	
			on Enter	on Leave
0	CFC	cfi.ready ∧ cfi.cfc	cfcPC.ce	fIFQswitch
1	MP0	bcu.valid ∧ bcu.mp	cfcPC.ce disableIW	fIFQswitch, iIFQ2fIFQ
2	MP0sw	bcu.valid ∧ bcu.mp ∧ (bcu.jump ∨ $\overline{\text{bcu.btaken}}$)	cfcPC.ce disableIW	fIFQswitch, iIFQ2fIFQ dontInitIFQ
3	MP1	rollback	enableIW	
4	MP2	MP0 ∧ rollback	—	fIFQswitch, iIFQ2fIFQ enableIW
5	MP2sw	MP0sw ∧ rollback	—	fIFQswitch, iIFQ2fIFQ dontInitIFQ, enableIW
6	JISR	JISR	cfcPC.ce disableIW	fIFQswitch, iIFQ2fIFQ enableIW

Table 3.3: Pending actions in the instruction fetch mechanism

iIFQ2fIFQ indicates that the issuing IFQ index will be set to the fetching IFQ index. We have:

$$\begin{aligned} \text{fetchIFQ.idx}' &:= \text{fetchIFQ.idx} \oplus \text{fIFQswitch} \\ \text{issueIFQ.idx}' &:= (\text{iIFQ2fIFQ} \ ? \ \text{fetchIFQ.idx}' \ : \ \text{issueIFQ.idx} \oplus \text{iIFQswitch}) \end{aligned}$$

- Register oss indicates if the fetch mechanism is in an outstanding speculation situation.

  As the algorithms state, the oss register is set on a prediction of a CFI and cleared on rollback, misprediction and successful verification. We have:

$$\text{oss}' := (\text{oss} \wedge \overline{\text{bcu.valid}}) \vee (\text{cfi.ready} \wedge \text{cfi.doPred})$$

- Register IWdisabled indicates a disabled instruction window. It is updated by the control signals disableIW and enableIW according to the following equation:

$$\text{IWdisabled}' := (\text{IWdisabled} \vee \text{disableIW}) \wedge \overline{\text{enableIW}}$$

- Register cfi4I.valid is active, if a CFI in the issuing IFQ awaits issue. It will be set, if a CFI is ready and it will be reset, if a CFI is issued or the instruction window has been disabled:

$$\text{cfi4I.valid}' := (\text{cfi4I.valid} \vee \text{cfi.ready}) \wedge \overline{(\text{cfiIssued} \vee \text{disableIW})}$$

  If cfi4I.valid = 1 register cfi4I.cfc denotes, if the CFI awaiting issue generates a change control flow. The register is updated with cfi.cfc, whenever the prediction environment got a prediction ready, signalled by cfi.cfc:

$$\text{cfi4I.cfc}' := (\text{cfi.ready} \ ? \ \text{cfi.cfc} \ : \ \text{cfi4I.cfc})$$

- The pending action register $\mathsf{P}_\star \in \{0,1\}^7$ will be described in detail below.

Pending Actions

Changes to the fetching IFQ may be only allowed on the completion of an instruction memory read access. There are three actions of the algorithms resulting in a change of the fetching IFQ (and the fetch PC):

- The prediction or resolving of a CFI causing a control flow change (cf. algorithm 15, ll. 12–15).
- A misprediction report by the branch checker unit (cf. algorithm 14, ll. 4–9). The instruction fetch mechanism may *initiate* rollback but might not complete until a rollback is signalled from the retirement stage as well.
- A JISR condition is reported by the retirement stage (cf. algorithm 13, ll. 3–6). A rollback to the interrupt service must be initiated.

Since all the three actions might possibly be detected in a single memory cycle, they must be prioritized to ensure the same order of execution as in the algorithms. Regular control flow changes are overruled by mispredictions and JISR conditions, misprediction is overruled by JISR conditions. Furthermore, the misprediction case has to be broken down in three subcases, to treat the waiting for the rollback adequately.

Table 3.3 shows an overview of the events we catch in each memory round. They are ordered by *increasing* priority. Each event has a detection signal associated, which evaluates to 1 if the preconditions for the event are met. An event will "enter" its pending execution, if it is the event of highest priority detected in the current memory round. On entering control signals as listed will be activated. If the instruction memory is not busy, as indicated by $\overline{\mathsf{IM.busy}}$, the event of highest priority so far detected may actually be executed; it "leaves" the state of pending action and activates the signals listed under column "on leave". We continue with a detailed description of the cases and will then develop a circuit to control the pending actions.

Following are the descriptions of the cases listed in table 3.3 from low to high priority:

- The prediction environment (cf. section 3.7), scanning for CFIs, signals $\mathsf{cfi.ready} \wedge \mathsf{cfi.cfc}$ if a control flow instruction causing a control flow change has been resolved or predicted.

  On enter, the determined $\mathsf{cfcPC}_\star$ is clocked by setting cfcPC.ce. On leave, the fetching IFQ is switched by setting fIFQswitch.

- Misprediction treatment must be broken down into a number subcases according to the two phasses detection and retirement of a mispredicted CFI.

  The cases MP0 and MP0sw account for the detection of the misprediction by the report $\mathsf{bcu.valid} \wedge \mathsf{bcu.mp} = 1$ of the branch checker unit. If additionally $\mathsf{bcu.jump} \vee \overline{\mathsf{bcu.btaken}} = 1$, the rollback will be by switching the IFQ only. On enter, MP0 and MP0sw both clock the $\mathsf{cfcPC}_\star$ register. On leave, both will switch the fetching IFQ and set the issuing IFQ to the fetching IFQ. MP0sw will prevent the initialization of the fetching IFQ by setting dontInitIFQ. Additionally, both will disable the instruction window by the signal disableIW.

  Cases MP1, MP2 and MP2sw deal with the treatment of the rollback, if the mispredicted CFI is retired in the reorder buffer. At earliest, this is one cycle after the detection of cases MP0 or MP0sw.

  Case MP2 and MP2sw deal with the rollback of a mispredicted CFI in the memory cylce *of the detection of MP0 or MP0sw*. Their detection therefore depends on the detection of MP0 or MP0sw in the same cycle. They will perform the same updates as MP0 and MP0sw and additionally enable the instruction window (enableIW) on leave.

  Case MP1 deals with the mispredicted CFI retiring in a memory cycle *after detection of MP0 or MP0sw*. The queue states therefore has already been

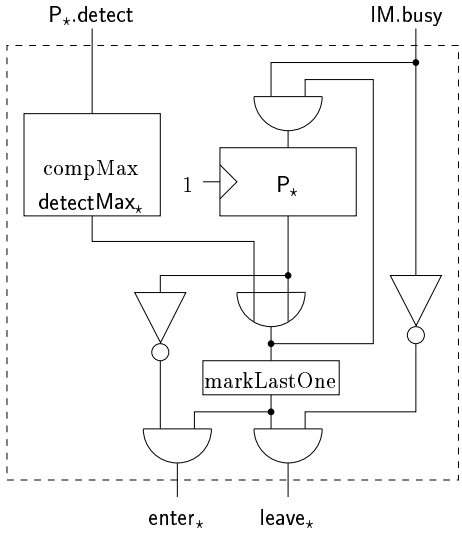

Figure 3.3: Pending actions of the instruction fetch

updated appropriately and only the instruction window needs to be enabled again.

- Detection of an interrupt requests by $\mathsf{JISR} = 1$ has the highest priority. It disables the instruction window on entering by activating disableIW and clocks the start of the interrupt service routine in the $\mathsf{cfcPC}_\star$ register by setting cfcPC.ce. On leave, the fetching IFQ is switched by setting fIFQswitch and the issuing IFQ follows (iIFQ2fIFQ). The instruction window is enabled again.

Figure 3.3 shows the circuit used to implement this control. The register $\mathsf{P}_\star$ holds the current action level detected in half-unary encoding.

Arriving at the control are the detection signals $\mathsf{P}_\star$.detect from table 3.3.[3] With a half-unary maximum computation we filter out the detected action:[4]

$$\mathsf{detectMax}_\star \quad := \quad \bigvee_{0 \le i < 7} (0^{7-i}, \mathsf{P_i.detect}^i)$$

The new (auxiliary) pending level can likewise be computed by a half-unary maximum computation:

$$\mathsf{auxP}_\star \quad := \quad \mathsf{P}_\star \vee \mathsf{detectMax}_\star$$

Now, the $\mathsf{auxP}_\star$ is also the new value of the $\mathsf{P}_\star$ register if the memory is still busy fetching. Otherwise the pending register will be cleared:

$$\mathsf{P'}_\star \quad := \quad \mathsf{auxP}_\star \wedge \mathsf{IM.busy}$$

The unary signals $\mathsf{enter}_\star$ and $\mathsf{leave}_\star$ signal the entering and leaving of a certain pending level. We have $\mathsf{enter_i} = 1$ iff level i was entered in the round and $\mathsf{leave_i} = 1$ iff level i was left in the round. We define as follows:

$$\begin{aligned} \mathsf{enter}_\star &:= \mathrm{markLastOne}\,(\mathsf{auxP}_\star) \wedge \overline{\mathsf{P}_\star} \\ \mathsf{leave}_\star &:= \mathrm{markLastOne}\,(\mathsf{auxP}_\star) \wedge \overline{\mathsf{IM.busy}} \end{aligned}$$

[3] Define $\mathsf{MP0} := \mathsf{P}_1 \wedge \overline{\mathsf{P}_2}$ and $\mathsf{MP0sw} := \mathsf{P}_2 \wedge \overline{\mathsf{P}_3}$

[4] Actually this computation simplifies greatly on filling in the detection expressions. This strategy will not be followed here for ease of reading.

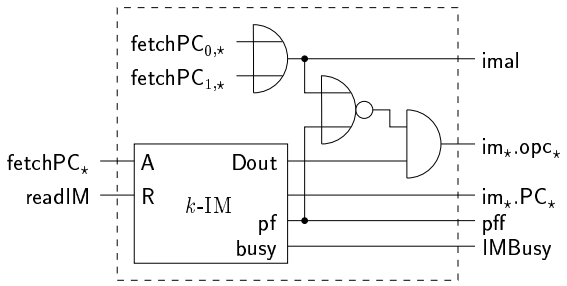

Figure 3.4: Instruction memory environment

These equations are explained as follows: the highest pending level, given in half-unary encoding by markLastOne ($\mathsf{auxP}_\star$) is entered, if it is greater than the previous pending level. It is left, if the instruction memory is not busy.

The control signals given in table 3.3 can be computed via the $\mathsf{enter}_\star$ and $\mathsf{leave}_\star$ busses. If S is such a signal, we set

$$\begin{aligned} \mathsf{S} \;:=\; & \bigvee_{\text{s activated on enter } i} \mathsf{enter}_i \\ \vee\; & \bigvee_{\text{s activated on leave } i} \mathsf{leave}_i \end{aligned}$$

3.4 Instruction Memory Environment

The design of the instruction memory environment IMenv follows the design presented in [Krö99]. However, the underlying instruction memory k-IM always returns k instruction opcodes and their PCs starting at the base address $\langle \mathsf{A}_\star \rangle_2$, not necessarily a multiple of k. The design of such instruction memory is not treated here; such extensions are closely related to cache design. An approach to cache design can be found in [MP00].

Figure 3.4 shows the implementation of the IMenv. Read requests are controlled by the IM read signal readIM. This signal requests reading whenever there is enough room in the fetching IFQ:

$$\mathsf{readIM} \;:=\; \overline{(\mathsf{fetchIFQ.idx}\;?\;\mathsf{ifq}_1\mathsf{.full}\;:\;\mathsf{ifq}_0\mathsf{.full})}$$

The fetch PC $\mathsf{fetchPC}_\star$, forwarded from the PC environment, specifies the read address. On $\overline{\mathsf{IM.busy}}$ Instruction PCs are returned on the busses $\mathsf{im}_\star\mathsf{.PC}_\star$. Instruction opcodes are returned on the busses $\mathsf{im}_\star\mathsf{.opc}_\star$. In two cases the opcodes are cleared, resulting in a nop instruction having no effect: First, the instruction fetch may be misaligned, signalled by $\mathsf{im}_\star\mathsf{.imal} := \mathsf{fetchPC}_1 \vee \mathsf{fetchPC}_0$. Second, one of the instructions may reside on a memory page swapped out of physical memory. In that case, the memory returns a page fault signal. This convention requires an additional software requirement for the interrupt service routine: on return from page fault recovery it must be guaranteed that all k instructions starting at address $\langle \mathsf{fetchPC}_\star \rangle_2$ reside in physical memory. The page fault signal is returned on $\mathsf{im}_\star\mathsf{.pff}$.

3.5 Instruction Fetch Queue Environment

Figure 3.5 shows the implementation of the instruction fetch queue environments using a multiported queue. Multiported queues are data structures that allow storing and retrieving of multiple elements in a round on a first-in-first-out basis. See

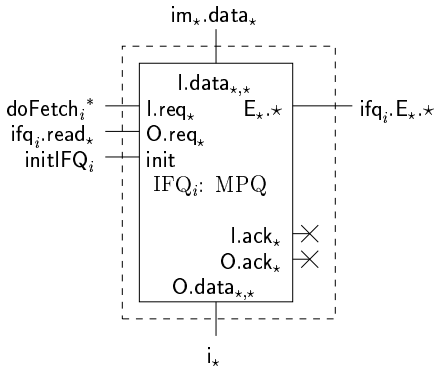

Figure 3.5: Instruction fetch queue environment

Component	Width	Purpose
valid	1	indicates valid entry (maintained by queue control)
$PC_\star$	32	instruction's program counter
$opc_\star$	32	instruction's opcode
pff	1	indicates page fault on fetch
imal	1	indicates misalignment

Table 3.4: Components of the instruction fetch queue entries

section 5.3 for the abstract definition, implementation and correctness proof of a multiported queue.

Table 3.4 shows the composition of an entry in the IFQ. The $PC_\star$ and $opc_\star$ component are self-explanatory. The page fault exception bit pff and the misalignment exception bit imal are passed by the instruction memory environment.

Each IFQ is controlled by three signals. The signal $initIFQ_i$ requests an initialization (i.e. a clearing of the valid bits) of the queue. The signal doFetch is activated in the same cycle that instructions from the memory arrive at this queue.

$$\begin{aligned} doFetch_i &:= \overline{IM.busy} \wedge readIM \wedge (fetchIFQ.idx = i) \\ initIFQ_i &:= fIFQswitch \wedge \overline{dontInitIFQ} \wedge (fetchIFQ.idx' = i) \end{aligned}$$

Data is requested for read out by the read signals $ifq_i.r_\star$ generated by the issue selector environment. It leaves the queue on the output busses $O_\star$. For prediction purposes, the queue entries are exported on the entry bus $E_\star.\star$ and received back on $E'_\star.\star$.

3.6 PC environment

Figure 3.6 shows the PC environment. It contains two registers, $IFQ_0.PC_\star$ and $IFQ_1.PC_\star$, holding the fetch PC for IFQ_0 and IFQ_1. Only one of these PCs is active at a time—in the sense of addressing the instruction memory. This PC is selected by the index of the fetching IFQ, fetchIFQ.idx, generated by the control. The resulting bus $fetchPC_\star$ was already used to address the instruction memory in the IMenv.

The update of these registers is as follows. For $i =$ fetchIFQ.idx register $IFQ_i.PC_\star$ receives the incremented-by-k-version of $fetchPC_\star$. It will only store it, if a fetch takes place, as indicated by the signal $doFetch_i$, which has been defined already. For $i = \overline{fetchIFQ.idx}$ register $IFQ_i.PC_\star$ is updated if it becomes the fetching PC in

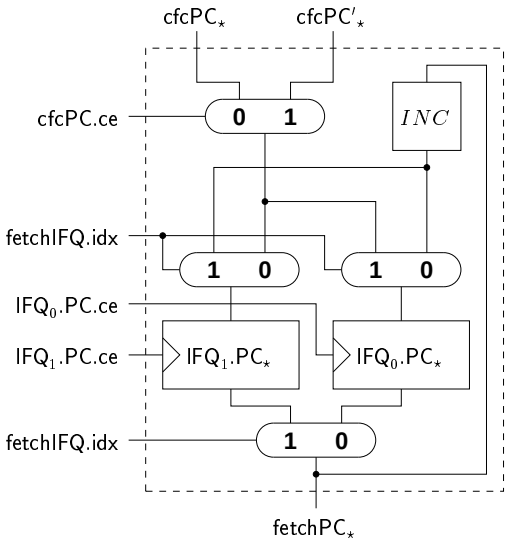

Figure 3.6: PC environment

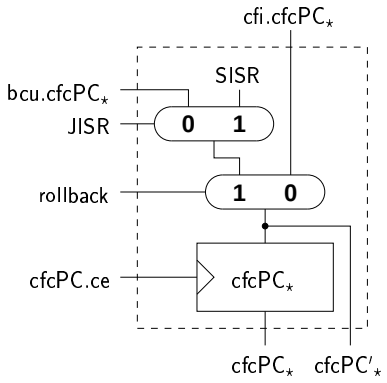

Figure 3.7: Control flow change PC generation

Signal	$\mathsf{opc}_{31..26}$	Associated instructions
t.epc	111111	`rfe`
t.gpr	01011*	`jr`, `jalr`
t.imm16	0001**	`beqz`, `bnez`, `fbeqz`, `fbnez`
t.imm26	00001*	`j`, `jal`
cfi	t.epc ∨ t.gpr ∨ t.imm16 ∨ t.imm26	
bbi	$\overline{\text{cfi}}$	
relJump	t.imm16 ∨ t.imm26	
absJump	$\overline{\text{relJump}}$	

Table 3.5: Control signals in the control flow environment

the next round and is initalized, i.e. if $\mathsf{initIFQ}_i = 1$ In this case it is fed either the stored value $\mathsf{cfcPC}_\star$ or the update value $\mathsf{cfcPC}'_\star$ of the CFC PC register, chosen by cfcPC.ce.

In summary, the clock enable signals of these register is defined as follows:

$$\mathsf{IFQ}_i.\mathsf{PC.ce} \quad := \quad \mathsf{doFetch}_i \vee \mathsf{initIFQ}_i$$

3.7 Prediction Environment

The prediction environment PredEnv is part of the hardware implementation of the prediction algorithm 15. The prediction environment scans for CFI in the issuing IFQ and attemps to predict or resolve them.

The following sections review the scan procedure for the CFI, the prediction of CFIs, the resolving of CFIs and the construction of the following bus:

cfi.ready	indicate that a CFI is ready
cfi.doPred	the CFI was predicted
cfi.btaken	the CFI is a taken branch
cfi.cfc	the CFI causes a control flow change
$\mathsf{cfi.cfcPC}_\star$	the control flow change PC of the CFI

3.7.1 Finding the first CFI

We start with the selection of the issuing IFQ from IFQ_0 and IFQ_1:

$$\mathsf{issueIFQ}_\star.\star \quad := \quad (\mathsf{issueIFQ.idx}\ ?\ \mathsf{IFQ}_1.\mathsf{E}_\star.\star\ :\ \mathsf{IFQ}_0.\mathsf{E}_\star.\star)$$

In hardware this is done in three steps.

First, the instruction opcodes $\mathsf{issueIFQ}_\star.\mathsf{opc}_\star$ are predecoded to determine their type. Table 3.5 shows the signals generated for each entry in the fetching IFQ. Refer to appendix A for an overview of the DLX instruction set architecture. The signals t.epc, t.gpr, t.imm16 and t.imm26 characterize the target a CFI can take; relJump indicates PC-relative jumps as opposed to absolute jumps. The signals cfi and bbi classify instructions into basic-block and control flow instructions.

Second, a find-first-one half-unary circuit $\mathrm{FF1hu}_n$ receiving $\mathsf{issueIFQ}_\star.\mathsf{cfi}$ returns the position of the first control flow instruction in the issuing IFQ in half-unary encoding. If $\mathsf{cfiNegHU}_\star \in \{0,1\}^n$ is the output of the circuit, we have

$$\langle \overline{\mathsf{cfiNegHU}_\star} \rangle_{hu} = l \quad \Longleftrightarrow \quad \begin{array}{ll} & \mathsf{issueIFQ}_{l'}.\mathsf{cfi} = 0 \quad l' < l \\ \wedge & \mathsf{issueIFQ}_l.\mathsf{cfi} = 1 \end{array}$$

Third, the found instruction is driven on an auxiliary bus $\mathsf{cfi}.\star$ by using

$$\mathsf{firstCFI}_\star \quad := \quad \mathrm{markFirstOne}\,(\mathsf{cfiNegHU}_\star)$$

as an output enable signal. The contents of $\mathsf{cfi}.\star$ are valid, if a candidate was found, the fetch mechanism has no outstanding speculation and no action is pending in the $\mathsf{P}_\star$ register:

$$\mathsf{cfi.valid} \quad := \quad \mathsf{cfiNegHU}_{n-1} \wedge (\overline{\mathsf{oss}} \vee \mathsf{bcu.valid} \wedge \overline{\mathsf{bcu.mp}}) \wedge \overline{\left(\bigvee \mathsf{P}_\star\right)}$$

3.7.2 Prediction

The $\mathsf{cfi}.\star$ bus is passed on to the branch predictor unit (BPU) for prediction. The BPU acknowledges the prediction requests with the signal bpu.pred. On acknowledgement, the BPU guarantess a valid bpu.btaken signal indicating the outcome of a branch or a valid $\mathsf{bpu.cfcPC}_\star$ bus indicating an absolute jump target.

Appendix B introduces a sample branch predictor for branch instructions only (so bpu.pred := cfi.t.imm16).

3.7.3 Resolving

To resolve branches, we access the source operand data generation of the decode / issue environment. The bus $\mathsf{firstCFI}_\star$ is used to select the first source operand of the CFI by drivers from all the first operands $\mathsf{i}_\star.\mathsf{op}_1.\mathsf{data}_\star$ in the decode / issue environment:

$$\mathsf{cfi.op}_1.\mathsf{data}_\star \quad := \quad \mathsf{i}_\mathsf{i}.\mathsf{op}_1.\mathsf{data}_\star \quad \text{with } \mathsf{firstCFI}_\mathsf{i} = 1, i \in \{0, \ldots, \beta - 1\}$$

The contents of this bus are only valid, if the decode / issue environment actually "sees" the CFI, i.e. if the CFI has an index which is less than the issue width. Additionally, the data on $\mathsf{i}_\mathsf{i}.\mathsf{op}_1.\mathsf{data}_\star$ is only valid if $\mathsf{i}_\mathsf{i}.\mathsf{op}_1.\mathsf{valid} = 1$. We obtain with selecting on $\mathsf{i}_\star.\mathsf{op}_1.\mathsf{valid}$ by drivers:

$$\mathsf{cfi.op}_1.\mathsf{valid} \quad := \quad \overline{\mathsf{cfiNegHU}_\beta} \wedge \mathsf{i}_\mathsf{i}.\mathsf{op}_1.\mathsf{valid} \quad \text{with } \mathsf{firstCFI}_\mathsf{i} = 1, i \in \{0, \ldots, \beta - 1\}$$

We resolve a CFI in three situations:

- A `rfe` instruction is resolved, when the ROB is emptied. This ensures that the exceptional registers $\mathsf{ESR}_\star$ and $\mathsf{EPC}_\star$ have the values of the last instruction writing to it.
- The unconditional relative jump instructions `j` and `jal` are resolved rightaway, since their target address can be immediately computed and they are known to jump unconditionally.
- The register jump instructions `jr` and `jalr` are resolved when their first operand becomes valid. The operand contains the target address.

This conditions are summarized in the following equation:

$$\mathsf{cfi.resolve} \quad := \quad (\mathsf{cfi.t.epc}\ ?\ \mathsf{ROB.empty}\ :\ \mathsf{cfi.op}_1.\mathsf{valid} \vee \mathsf{cfi.t.imm26})$$

If $\mathsf{cfi.resolve} = 1$, a taken branch is signalled by

$$\mathsf{cfi.resolve.btaken} \quad := \quad \mathsf{cfi.op}_1.\mathsf{eqz} \oplus \mathsf{cfi.opc}_{26}$$

where $\mathsf{cfi.op}_1.\mathsf{eqz} := (\bigvee \mathsf{cfi.op}_1.\mathsf{data}_\star)$. The target of a direct jump can be directly taken from $\mathsf{cfi.op}_1.\mathsf{data}_\star$.

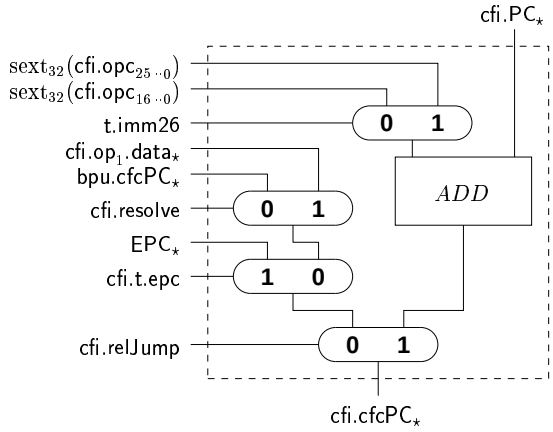

Figure 3.8: Generation of the CFI CFC PC

3.7.4 Construction of the CFI bus

A CFI is ready for execution, if it is valid can be either predicted or resolved:

$$\mathsf{cfi.ready} \quad := \quad \mathsf{cfi.valid} \wedge (\mathsf{cfi.resolve} \vee \mathsf{bpu.pred})$$

Resolving a CFI has priority over prediction. This is reflected in the definitions of the signals doPred, telling the BPU to predict actually, cfi.btaken indicating a taken branch and cfi.cfc indicating a control flow change:

$$\begin{aligned} \mathsf{cfi.doPred} &:= \overline{\mathsf{cfi.resolve}} \wedge \mathsf{bpu.pred} \\ \mathsf{cfi.btaken} &:= (\mathsf{cfi.resolve}\ ?\ \mathsf{cfi.resolve.btaken}\ :\ \mathsf{bpu.btaken}) \\ \mathsf{cfi.cfc} &:= \overline{(\mathsf{cfi.t.imm16} \wedge \overline{\mathsf{cfi.btaken}})} \end{aligned}$$

Finally, prediction or resolving results in the computation of a control flow change PC for the CFI, $\mathsf{cfi.cfcPC}_\star$. Figure 3.8 shows how to compute this bus. For relative jumps, indicated by cfi.relJump, the $\mathsf{cfcPC}_\star$ is computed as the PC of the CFI plus the immediate constant, either 16 or 26 bits. Otherwise, the target is the $\mathsf{EPC}_\star$ register for `rfe` instructions, the resolved or the predicted address for jumps to register locations.

3.8 Instruction Window Environment

3.8.1 Construction of the Instruction Window

The construction of the instruction window takes place in the instruction window environment IWenv. Here, the instructions that the fetch mechanism choses to present to the decode / issue environment are selected and the entries are padded with the necessary interface information (classification of instructions by flags bbi, cfi and providing the next PC $\mathsf{nPC}_\star$) to the decode / issue environment.

Some work has already been done in section 3.7.1: recall the definition of the issuing IFQ as

$$\mathsf{issueIFQ}_\star.\star \quad := \quad (\mathsf{issueIFQ.idx}\ ?\ \mathsf{IFQ}_1.\mathsf{E}_\star.\star\ :\ \mathsf{IFQ}_0.\mathsf{E}_\star.\star)$$

In accordance to the nomenclature used in the algorithms we rename the first entries of the issuing IFQ. For clarity, we explicitly list all the renamed bus items, $i \in$

$\{0, \ldots, \beta - 1\}$:

$$\begin{aligned} \mathsf{i_i.opc_\star} &:= \mathsf{issueIFQ_i.opc_\star} \\ \mathsf{i_i.PC_\star} &:= \mathsf{issueIFQ_i.PC_\star} \\ \mathsf{i_i.imal} &:= \mathsf{issueIFQ_i.imal} \\ \mathsf{i_i.pff} &:= \mathsf{issueIFQ_i.pff} \end{aligned}$$

The valid, bbi, cfi signals are computed as follows:

$$\begin{aligned} \mathsf{i_\star.valid} &:= \mathsf{issueIFQ_\star.valid} \wedge \overline{\mathsf{IWdisabled}} \\ \mathsf{i_\star.bbi} &:= \overline{\mathsf{cfiNegHU_\star}} \\ \mathsf{i_\star.cfi} &:= \mathsf{firstCFI_\star} \wedge (\mathsf{cfi.ready} \vee \mathsf{cfi4I.valid}) \\ \mathsf{i_\star.cfc} &:= \mathsf{firstCFI_\star} \wedge (\mathsf{cfi.ready}\ ?\ \mathsf{cfi.cfc} : \mathsf{cfi4I.cfc}) \end{aligned}$$

The abstract interface of a speculative instruction fetch mechanism requires to compute the next PC for every instruction. As the PC is not a general-purpose source register in the DLX instruction set, we need to supply its alleged new value only for the verification of control flow instructions. Since at most one control flow instruction is presented to the decode / issue stage in the issuing IFQ, we supply the next PC via

$$\mathsf{cfi4I.cfcPC_\star} := (\mathsf{cfcPC.ce}\ ?\ \mathsf{cfcPC'_\star} : \mathsf{cfcPC_\star})$$

already generated in the CFC PC generation. Additionally, we also have to supply the sequential PC for every CFI, as we will see in the description of the branch checker unit, section 3.11.4. For this we compute

$$\mathsf{cfi4I.inlinePC_\star} := \mathrm{inc4}\,(\mathsf{cfi.PC_\star})$$

with inc4(·) denoting an increment-by-4 function.

3.8.2 Draining and Switching the Issuing IFQ

The second task of the issue selector environment is to drain the issuing IFQ and initiate a switch, in case that a CFI causing a control flow change has been issued.

The decode / issue environment returns for this purpose the $\mathsf{i_\star.issue}$ signals; giving a half-unary encoding of the number of issued instructions. The following signals detects, if a CFI has been issued:

$$\mathsf{cfiIssued} := \left(\bigvee \mathsf{i_\star.issue} \wedge \mathsf{i_\star.cfi}\right)$$

If a CFI causing a control flow change has been issued, the issuing IFQ must be switched:

$$\mathsf{iIFQswitch} := \mathsf{cfiIssued} \wedge (\mathsf{cfi.ready}\ ?\ \mathsf{cfi.cfc} : \mathsf{cfi4I.cfc})$$

The issuing IFQ is drained by the number of issuing instructions, supplied in half-unary encoding. We have:

$$\mathsf{ifq_i.read_\star} := \mathsf{i_\star.issue} \wedge (\mathsf{issueIFQ.idx} = i)$$

Decode / Issue Stage

3.9 Decode / Issue Environment

The purpose of the decode / issue environment (DIenv) is to take instructions from the IFQ in a sequential order, compute control signals for them, gather their operands if possible and direct them to the appropriate reservation stations to initiate their execution. These operations have to be executed for an arbitrary but fixed number β of instructions simultaneously.

In accordance to the nomenclature of issue protocol, algorithm 1 (p. 16), the instructions in the instruction window are delivered by the instruction fetch stage on busses $\mathsf{i}_\mathsf{i}.\star$. The following two subsections describe the exact operation of the DIenv.

3.9.1 Decoding the Instruction

The decoding component for instruction i_i takes the instruction opcode $\mathsf{i}_\mathsf{i}.\mathsf{opc}$ and computes the following control signals:

- Signal isCFI indicates that the decoded instruction is a CFI. The opposite condition, i.e. that the instruction is a BBI, is expressed by the negation, $\overline{\mathsf{isCFI}}$.

- The signals itype, jtype and rtype determine the *instruction type* according to the instruction set architecture of the DLX, appendix A. The instruction type specifies the location of operand addresses or immediate constants in the instruction's opcode.

  Naturally, exactly one of the signals itype, jtype and rtype should be activated.

- Floating point instructions are marked by the signal fp; in case that they operate on double IEEE encodings, signal db is activated additionally.

- The functional unit identifiers FU.$\star$, explicitly listed in table 3.6, specify which functional unit will be used for the execution of instruction i_i. The special signal noFU indicates, that the instruction does not need a functional unit for execution, i.e. it completes on issue. Only illegal instructions meet this requirement.

  Naturally, exactly one of the signals FU.$\star$ should be activated.

- The *operand signals* are defined for the source operands op_1 and op_2 and for the destination operand d of an instruction i_i. Their signals can be divided into two subgroups.

  The first group, consisting of the signals IMM, RS1, RS2, RD, FS1, FS2, SA, R31, describe the possible sources and destination in a DLX instruction. These are, in order of appearance, a 16-bit or 26-bit immediate constant, the first and second general purpose register operand, the integer destination register, the first and second floating point source operand, the shift amount and register R_{31}. Again, the nomenclature and semantics is shown in appendix A.

  The second group locates register operands in the machine: gpr, fpr and spr determine if the register is general-purpose, floating-point or special; db specifies a double operand in case of a floating-point reference. The register address is specified by the address bus $\mathsf{a}_\star \in \{0,1\}^5$.

Functional Unit	Purpose
$FU_0 \equiv$ FU.alu	int computation
$FU_1 \equiv$ FU.mem	load / store
$FU_2 \equiv$ FU.add	fp addition / subtraction
$FU_3 \equiv$ FU.fmul	fp multiplication
$FU_4 \equiv$ FU.fdiv	fp division
$FU_5 \equiv$ FU.fconv	conversion int / fp
$FU_6 \equiv$ FU.ftest	fp condition test
$FU_7 \equiv$ FU.bcu	branch checker unit

Table 3.6: Coding of the functional units

- The remaining signals, load, fabsneg, ff2i, fi2f, fmov, link, jump, noChk, trap, rfe, movs2i, movi2s and ill, specify behaviour of the functional unit executing the instructions. They are just passed on to the functional unit and will not be described in detail here.

The decoding of the instruction, i.e. the computation of the signals just described, is divided into two parts. The control automaton CSig decodes the instruction word and activates the control signals necessary for the execution of the instruction. This is done by specifying states for sets of instructions, as listed in table 3.7. Table 3.8 defines the automaton CSig. For each state, monomials are listed matching to the corresponding instructions (cf. appendix A). The automaton computes the monomials to determine its state at the beginning of each cycle. Then, it activates the control signals for each state, in correspondance to the signals given in the table.

A discussion of such automata, including analysis for cost and delay, is found in [MP95].

Additionally, for each operand op_1, op_2 and dest the operand address is generated. The operand address is gathered from opcode bits specified in appendix A according to the operand type. So, we have:

$$op_1.a_\star := \begin{cases} opc_{10..6} & \text{for } op_1.SA = 1 \\ opc_{20..16} & \text{for } op_1.RS2 = 1 \\ opc_{25..21} & \text{otherwise} \end{cases}$$

$$op_2.a_\star := opc_{20..16}$$

$$d.a_\star := \begin{cases} (11111) & \text{for } d.R31 = 1 \\ (01000) & \text{for } d.FCC = 1 \\ opc_{10..6} & \text{for } d.SA = 1 \\ opc_{15..11} & \text{for } \overline{itype} \wedge (d.RD \vee d.FD) = 1 \\ opc_{20..16} & \text{for } itype \wedge (d.RD \vee d.FD) = 1 \end{cases}$$

The computation of the signals db, fpr, gpr and spr for the operands is straightforward. Refer to figure 3.9 showing the operand address computation.

3.9.2 Issuing the Instruction

Testing for Reservation Station Availability

The issue protocol, algorithm 1 (p. 16), states that an instruction i_i cannot be issued, if there is no appropriate reservation station available. This section derives the computation of the signal i_i.noRS indicating this condition.

Consider a functional unit FU_j, fixed. This functional unit is associated with a set of reservation stations which we will later organize in a queue RSQ_j (section 3.14)

State	Instructions
ALU	sll, sla, srl, sra add, addu, sub, subu, and, or, xor, lhg clr, sgr, seq, sge, sls, sne, sle, set
Shifti	slli, slai, srli, srai
ALUi	addi, addiu, subi, subiu andi, ori, xori, lhgi clri, sgri, seqi, sgei, slsi, snei, slei, seti
Load	lb, lh, lw, lbu, lhu
Load.s	load.s
Load.d	load.d
Store	sb, sh, sw
Store.s	store.s
Store.d	store.d
Faddsub.s	fadd.s, fsub.s
Faddsub.d	fadd.d, fsub.d
Fmul.s	fmul.s
Fmul.d	fmul.d
Fdiv.s	fdiv.s
Fdiv.d	fdiv.d
Fcond.s	fc.cond.s
Fcond.d	fc.cond.d
Fabsneg.s	fabs.s, fneg.s
Fabsneg.d	fabs.d, fneg.d
Ff2i	mf2i
Fi2f	mi2f
FMov.s	mov.s
FMov.d	mov.d
FConv.s	cvt.s.d, cvt.s.i, cvt.i.s, cvt.i.d
FConv.d	cvt.d.i, cvt.d.s
Branch	beqz, bnez
FBranch	fbeqz, fbnez
JumpReg	jr
JLinkReg	jalr
Jump	j
JLink	jal
Trap	trap
RFE	rfe
Movs2i	movs2i
Movi2s	movi.s
uFOP	fsqt.s, fsqt.d, frem.s, frem.d

Table 3.7: Correspondance of states and instructions

State	Monomials			Active Control Signals				
	$opc_{31..26}$	opc_6	$opc_{5..0}$		FU.	op_1.	op_2.	d.
ALU	000000	*	0001**	rtype	alu	RS1	RS2	RD
	000000	*	10****					
Shifti	000000	*	0000**	rtype	alu	RS1	imm	RD
ALUi	0*1***	*	******	itype	alu	RS1	imm	RD
Load	100***	*	******	itype, load	mem	RS1	–	RD
Load.s	110001	*	******	itype, load, fp	mem	RS1	–	FD
Load.d	110101	*	******	itype, load, fp, db	mem	RS1	–	FD
Store	101***	*	******	itype	mem	RS1	RD	–
Store.s	111001	*	******	itype, fp	mem	RS1	FD	–
Store.d	111101	*	******	itype, fp, db	mem	RS1	FD	–
Faddsub.s	010001	0	00000*	rtype, fp	fadd	FS1	FS2	FD
Faddsub.d	010001	1	00000*	rtype, fp, db	fadd	FS1	FS2	FD
Fmul.s	010001	0	000010	rtype, fp	fmul	FS1	FS2	FD
Fmul.d	010001	1	000010	rtype, fp, db	fmul	FS1	FS2	FD
Fdiv.s	010001	0	000011	rtype, fp	fdiv	FS1	FS2	FD
Fdiv.d	010001	1	000011	rtype, fp, db	fdiv	FS1	FS2	FD
Fcond.s	010001	0	11****	rtype, fp	ftest	FS1	FS2	FCC
Fcond.d	010001	1	11****	rtype, fp, db	ftest	FS1	FS2	FCC
Fabsneg.s	010001	0	00010*	rtype, fabsneg, fp	fconv	FS1	–	FD
Fabsneg.d	010001	1	00010*	rtype, fabsneg, fp, db	fconv	FS1	–	FD
Ff2i	010001	*	001001	rtype, ff2i, fp	fconv	FS1	–	RS2
Fi2f	010001	*	001010	rtype, fi2f, fp	fconv	RS2	–	FS1
FMov.s	010001	0	001000	rtype, fmov, fp	fconv	FS1	–	FD
FMov.d	010001	1	001000	rtype, fmov, fp, db	fconv	FS1	–	FD
FConv.s	010001	*	100*00	rtype, fp	fconv	FS1	–	FD
FConv.d	010001	*	100001	rtype, fp, db	fconv	FS1	–	FD
Branch	00010*	*	******	itype, isCFI	bcu	RS1	–	–
FBranch	00011*	*	******	itype, isCFI	bcu	FCC	–	–
JumpReg	010110	*	******	itype, jump, isCFI	bcu	RS1	–	–
JLinkReg	010111	*	******	itype, jump, isCFI	bcu	RS1	–	R31
Jump	000010	*	******	jtype, jump, noChk, isCFI	bcu	imm	–	–
JLink	000011	*	******	jtype, jump, noChk, isCFI	bcu	imm	–	R31
Trap	111110	*	000000	jtype, trap, noChk	bcu	imm	–	–
RFE	111111	*	******	jtype, rfe, noChk, isCFI	bcu	imm	–	–
Movs2i	000000	*	010000	rtype, movs2i	alu	SA	–	RD
Movi2s	000000	*	010001	rtype, movi2s	alu	RS1	–	SA
uFOP	010001	*	00011*	uFOP	noFU	–	–	–
	010001	*	01****					
Illegal (z_0)	(alternative case)			ill	noFU	–	–	–

Table 3.8: Decode control

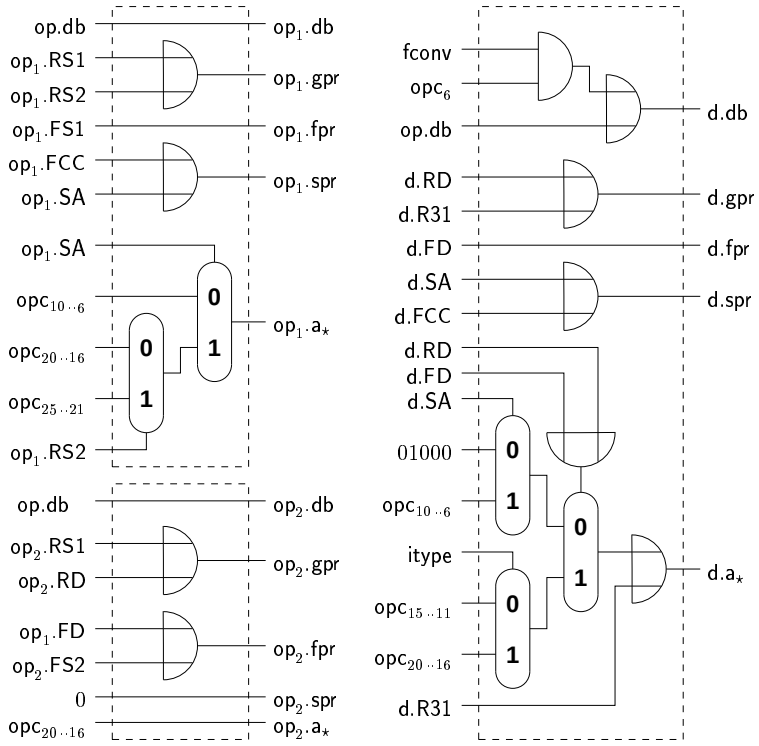

Figure 3.9: Operand address generation

of size RSQ.SZ$_j$. The fill state of the queue is indicated by the signals $\mathsf{RSQ_j.empty_\star}$; we have $\langle \mathsf{RSQ_j.empty_\star} \rangle_{hu} = l$ iff there no more than $(l+1)$ free reservation stations left. Furthermore, the number of instructions issuing on the queue each cycle is bounded by its "indegree", the parameter $k := \mathrm{RSQ.IN}_j$.

The instruction $\mathsf{i_i}$ cannot issue on $\mathsf{RSQ_j}$ if it is the l–th candidate requesting $\mathsf{FU_j}$ and $\mathsf{RSQ_j}$ has no l free reservation stations left. In circuitry, we compute the candidate number $\mathsf{i2RSQ_{i,j}.cand_\star} \in \{0,1\}^{k+1}$ for instruction $\mathsf{i_i}$ issuing on $\mathsf{RSQ_j}$:

$$\langle \mathsf{i2RSQ_{i,j}.cand_\star} \rangle_{hu} \quad = \quad \min\{k+1, \mathrm{ones}(\mathsf{i_{i..0}.FU_j})\}$$

A find-first-$(k+1)$-ones-half-unary circuit, described in section C.2, computes this information. Now we check $\langle \mathsf{i2RSQ_{i,j}.cand_\star} \rangle_{hu} > \langle \mathsf{RSQ_j.empty_\star} \rangle_{hu}$ by half-unary comparison (equation 3.12, p. 36) and obtain

$$\mathsf{RSQbusy4l_{j,i}} \quad := \quad \left(\bigvee \overline{\mathsf{i2RSQ_{i,j}.cand_\star}} \wedge \mathsf{RSQ_j.empty_\star} \right)$$

The following equation computes $\mathsf{i_i.noRS}$:

$$\mathsf{i_i.noRS} \quad := \quad \overline{\mathsf{i_i.noFU}} \wedge \left(\bigvee \mathsf{i_i.FU_\star} \wedge \mathsf{RSQbusy4l_{\star,i}} \right)$$

This reflects the fact that an instruction needing no functional unit never fails in finding one and for $\mathsf{i_i.FU_j} = 1$ there must be a free reservation station in $\mathsf{RSQ_j}$, i.e. $\mathsf{RSQbusy4l_{j,i}} = 1$.

Source Operand Data Generation

The source operand data generation computes for each source operand a tuple $(\mathsf{tag_\star}, \mathsf{valid}, \mathsf{data_\star}) \in \{0,1\}^\nu \times \{0,1\} \times \{0,1\}^{32}$, where ν is the tagwidth (cf. section

3.13). Double source operands are composed of a low part and a high part, so four operands $\mathsf{op}_{k,b}$, $k \in \{1,2\}$ and $b \in \{lo, hi\}$ have to be generated.

The source operand data generation is a direct implementation from the appropriate portion of the issue protocol, algorithm 1, p. 16. Recall that $\mathsf{valid} = 1$ signals a valid $\mathsf{data}_\star$ component while $\mathsf{valid} = 0$ ensures that $\mathsf{tag}_\star$ contains the tag of the producing instruction of the operand.

We proceed with a description of the various cases. They are ordered according to their priority, so the latter cases only apply if none of the previous cases applies.

- Let $b = lo$. If the operand is the immediate constant, signalled by $\mathsf{i_i.op_k.imm}$, we set

$$(\mathsf{tag}_\star\,,\mathsf{valid}\,,\mathsf{data}_\star) \;:=\; (0^\nu, 1, \mathsf{i_i.co}_\star)$$

  To compute the immediate constant $\mathsf{i_i.co}_\star \in \{0,1\}^{32}$, three cases must be distinguished: for register type instructions, the immediate constant is taken from the shift amount field, $\mathsf{i_i.opc}_{10..6}$. Otherwise, it is the sign-extended 16-bit constant for immediate type instructions or the sign-extended 26-bit constant for jump type instructions. The distinction of the three cases results in the following equations:

$$\begin{aligned}
\mathsf{i_i.co}_{5..0} &:= (\mathsf{i_i.rtype}\ ?\ \mathsf{i_i.opc}_{10..6} : \mathsf{i_i.opc}_{5..0}) \\
\mathsf{i_i.co}_{15..6} &:= \mathsf{i_i.opc}_{15..6} \\
\mathsf{i_i.co}_{25..16} &:= (\mathsf{i_i.jtype}\ ?\ \mathsf{i_i.opc}_{25..16} : \mathsf{i_i.opc}_{15}) \\
\mathsf{i_i.co}_{32..26} &:= \mathsf{i_i.co}_{25}
\end{aligned}$$

  For $b = hi$, the procedure is simplified. If an immediate constant operand is indicated by $\mathsf{i_i.op_k.imm}$ *or* if the operand is no double operand ($\overline{\mathsf{i_i.op_k.db}} = 1$) we define:

$$(\mathsf{tag}_\star\,,\mathsf{valid}\,,\mathsf{data}_\star) \;:=\; \left(0^\nu, 1, 0^{32}\right)$$

- We now check, if an accompanying instruction $\mathsf{i_j}$ with $j < i$ produces the source operand. This is done by comparing the source operand's location information db, gpr, fpr, spr and $\mathsf{a}_\star$ to the destination location for $\mathsf{i_j}$. Let the predicate eqA (i , k , b , j) indicate that operand $\mathsf{i_i.op_{k,b}}$ is produced by $\mathsf{i_j}$. For $b = lo$ this is the case, if the operand has the same address for the same location. If the destination of $\mathsf{i_j}$ is a double floating point register, only the upper four bits of the address must match. So, with S denoting $\mathsf{i_i.op_{k,b}}$ and D denoting $\mathsf{i_j.d}$, we define:

$$\begin{aligned}
\mathrm{eqA}\,(\mathsf{i}\,,\mathsf{k}\,,\mathsf{lo}\,,\mathsf{j}) \;:=\; & ((\mathsf{S.gpr}\,,\mathsf{S.fpr}\,,\mathsf{S.spr}\,,\mathsf{S.a}_{4..1}\,,\mathsf{S.a}_0 \vee \mathsf{D.db}) \\
& = (\mathsf{D.gpr}\,,\mathsf{D.fpr}\,,\mathsf{D.spr}\,,\mathsf{D.a}_{4..1}\,,\mathsf{D.a}_0 \vee \mathsf{D.db}))
\end{aligned}$$

  For $b = hi$, the computation is similar. This case is only relevant for $\mathsf{i_i.op_k}$ being a double floating point register, since the other operands are all 32 bits. Instruction $\mathsf{i_j}$ produces the source operand, if its destination is a single floating point register with $\mathsf{a}_0 = 1$ or double floating point register. So, with S denoting $\mathsf{i_i.op_{k,b}}$ and D denoting $\mathsf{i_j.d}$, we define:

$$\mathrm{eqA}\,(\mathsf{i}\,,\mathsf{k}\,,\mathsf{hi}\,,\mathsf{j}) \;:=\; (\mathsf{i_i.fpr}\,,\mathsf{a}_{4..1}\,,1) = (\mathsf{i_j.fpr}\,,\mathsf{a}_{4..1}\,,\mathsf{a}_0 \vee \mathsf{i_j.db})$$

  In case that equal (or as in the double case rather overlapping) location is signalled, we obtain the instruction tag $\mathsf{i_j.tag}_\star := \mathsf{ROB.tail}_{\mathsf{j},\star}$ for the maximum j with eqA (i , k , b , j) = 1

$$(\mathsf{tag}_\star\,,\mathsf{valid}\,,\mathsf{data}_\star) \;:=\; \left(\mathsf{i_j.tag}_\star, 1, 0^{32}\right)$$

- Now we have to access the register file to obtain the tag and the valid bit the source operand. In our implementation, tag and valid bit are stored in the so-called producer tables, while the actual data of registers is stored in the register file.

  The producer table is accessed via the addresses of the source operands and returns the signal $\mathsf{PTvalid} := \mathsf{pt.i_i.op_{k,b}.valid}$ and bus $\mathsf{PTtag}_\star := \mathsf{pt.i_i.op_{k,b}.tag}_\star$. If $\mathsf{PTvalid} = 1$, then the data stored in the register file, $\mathsf{rf.i_i.op_b.data_{b,\star}}$ is valid and can be used to compose the source operand:

$$(\mathsf{tag}_\star\,,\mathsf{valid}\,,\mathsf{data}_\star) \quad := \quad (0^\nu, 1, \mathsf{rf.i_i.op_i.data_{b,\star}})$$

- Having $\mathsf{PTvalid} = 0$, the instruction producing the source operand has not yet retired. To detect the result being broadcast on a common data bus in the same cycle, we define

$$\mathsf{snoopCDB_l} \quad := \quad ((\mathsf{CDB_l.valid}\,,\mathsf{CDB_l.tag}_\star) = (1\,,\mathsf{PTtag}_\star))$$

  We have $\mathsf{snoopCDB_l} = 1$, if the result has been detect on $\mathsf{CDB_l}$. Note, that as tags are unique, at most one CDB can qualify in having the tag being looked for, so the order of checks is not important. A succesful snooping has taken place, if one of the signals $\mathsf{snoopCDB_l}$ is active:

$$\mathsf{snoop} \quad := \quad \left(\bigvee \mathsf{snoopCDB_l}\right)$$

  In case of snooping, data from the snooped CDB can be driven by $\mathsf{snoopCDB_l}$ on the source operand bus.

- If the reorder buffer, accessed by $\mathsf{PTtag}_\star$, signals valid data with $\mathsf{rob.i_i.op_i.valid}$, the instruction having tag $\mathsf{PTtag}_\star$ already completed its computation and the result is stored in the reorder buffer as $\mathsf{rob.i_i.op_{k,b}.data}_\star$. So, under the assumption of $\mathsf{rob.i_i.op_{k,b}.valid} = 1$, we set:

$$(\mathsf{tag}_\star\,,\mathsf{valid}\,,\mathsf{data}_\star) \quad := \quad \left(0^\nu, 1, \mathsf{rob.i_i.op_{k,b}.data}_\star\right)$$

- Finally, only the tag is available in this case. Therefore we set

$$(\mathsf{tag}_\star\,,\mathsf{valid}\,,\mathsf{data}_\star) \quad := \quad \left(\mathsf{PTtag}_\star\,, 0, 0^{32}\right)$$

The cases just described lead to the implementation of the source operand data generation as shown in figure 3.10.

Figure 3.11 shows the source operand data generation for the special registers RM (rounding mode) and MSK (interrupt mask) needed by floating point operations. The procedure is similar to the general operands. Since these source operand generation refers to fixed-address registers, most part of its hardware may be shared for different instructions. The predicates eqRM (j) and eqMSK (j) are used to detect instructions writing to RM and MSK. With the addresses of the RM and the MSK register being $\langle 0110\rangle_2$ and $\langle 0000\rangle_2$, according to table 3.13, these predicates are defined as follows:

$$\begin{aligned} \mathrm{eqRM}\,(\mathsf{j}) \quad &:= \quad (\mathsf{i_j.d.spr}\,,\mathsf{i_j.a}_{3..0}) = (1, 0110) \\ \mathrm{eqMSK}\,(\mathsf{j}) \quad &:= \quad (\mathsf{i_j.d.spr}\,,\mathsf{i_j.a}_{3..0}) = (1, 0000) \end{aligned}$$

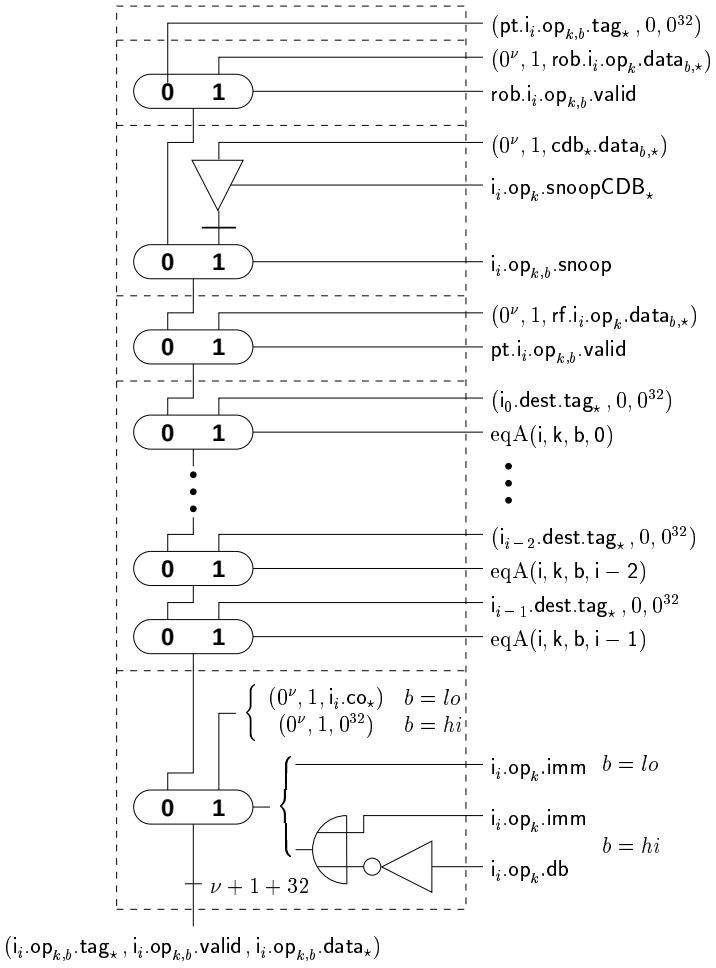

Figure 3.10: Generating the source operand data, $b \in \{lo, hi\}$

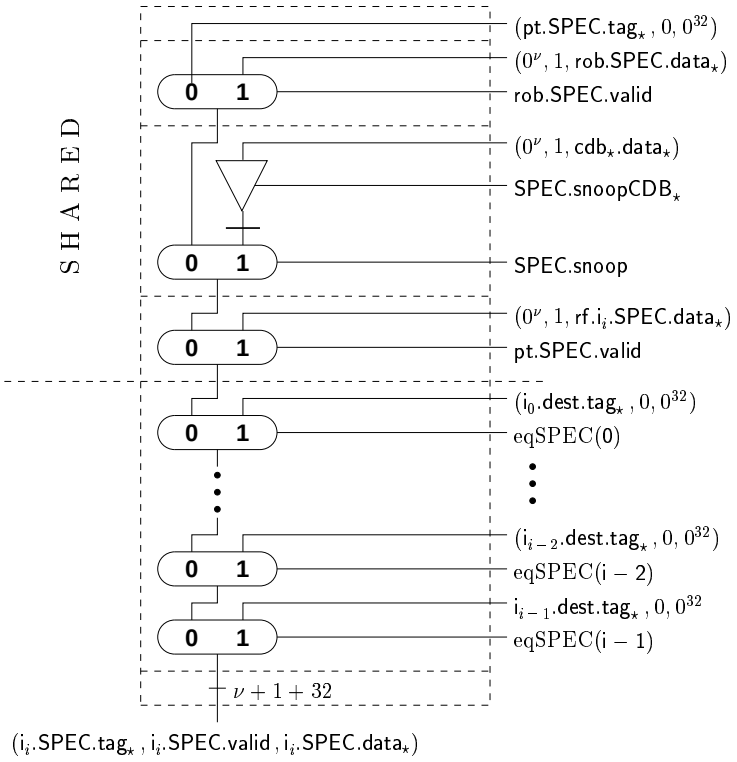

Figure 3.11: Generating the source operand data, SPEC $\in \{RM, MSK\}$

Stall Generation

The purpose of the stall generation is to model the execution of the issue loop present in the issue protocol, algorithm 1 (p. 16). The goal is the computation of the signal i_i.stall (and its complement i_i.issue$:= \overline{i_i.\text{stall}}$); i_i.stall $= 1$ iff instruction i_i cannot be issued. We proceed in two steps. First, we define the signal i_i.stallAux, indicating a "local" stall condition. This signal is used to compute the actual stall signals in the second step.

The following cases form the local stall condition for an instruction i_i:

- On rollback condition, signalled by rollback, no instruction shall issue.
- The cycle after an `rfe` instruction has been issued, the machine performs the actual register transfers for returing from exception (i.e. SR will be initialized to 0). The register doRFE is used to catch this event and stall issuing in this case:

$$\text{doRFE}' \quad := \quad \left(\bigvee i_\star.\text{rfe} \wedge i_\star.\text{issue}\right)$$

- The instruction may not be ready for issue, which happens in two cases: the instruction is not fetched or the instruction was falsely classified by the instruction fetch mechanism. This results in

$$i_i.\text{invalid} \quad := \quad \overline{i_i.\text{valid}} \vee (i_i.\text{cfi} \oplus i_i.\text{isCFI}) \vee \left(i_i.\text{bbi} \oplus \overline{i_i.\text{isCFI}}\right)$$

- There is no reservation station left to issue instruction i_i, i.e. i_i.noRS $= 1$.
- There is no room left to store i_i in the reorder buffer. The reorder buffer signals this condition by ROB.i_i.full $= 1$.

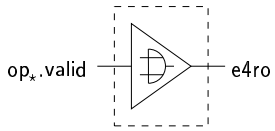

Figure 3.12: Eligibility for read-out for reservation station

- The instruction is a `movs2i` with source register $\mathsf{IEEEf}_\star$ and the ROB is not empty. This is necessary since no forwarding mechanism for the $\mathsf{IEEEf}_\star$ register is implemented, so preceding floating point instructions may modify $\mathsf{IEEEf}_\star$ without notice. This condition is written as follows:

$$\mathsf{i_i.IEEEfstall} \quad := \quad \mathsf{i_i.movs2i} \wedge (\mathsf{i_i.opc}_{10..6} = 00111) \wedge \overline{\mathsf{ROB.empty}}$$

The following equation summarizes all the above cases:

$$\begin{aligned} \mathsf{i_i.stallAux} \quad &:= \quad \mathsf{rollback} \vee \mathsf{doRFE} \\ &\vee \quad \mathsf{i_i.invalid} \vee \mathsf{i_i.noRS} \vee \mathsf{ROB.i_i.full} \vee \mathsf{i_i.IEEEfstall} \end{aligned}$$

The (global) stall condition is written as follows:

$$\mathsf{i_i.stall} \quad := \quad \left(\bigvee \mathsf{i_{i..0}.stallAux}\right)$$

This information can be computed using a parallel-prefix OR or a find-first-1 half-unary circuit.

3.10 Reservation Station Environment

The Reservation Station Environment is built up using a RS-Queue. An RS-Queue is a queue that provides an additional signal e4ro for each entry which is active if the entry is eligible for read-out. Among the candidates for read-out a RS-queue selects the oldest. The formal specification of an RS-queue and its implementation can be found in section 5.4.

The data elements of the RS queue consist of the control signals for the FU, tuples of (tag, valid, data) for each source operand and the tag for the destination operand. An entry is eligible for read-out if it is not empty (that is accounted for automatically in the RS queue) and if all its source operands are valid. The resulting circuit is drawn in figure 3.12.

The reservation station queues are initialized on the activation of the global signal rollback.

3.10.1 Scheduling of the reservation station input busses

Let $\mathsf{RSQ_j}$ have indegree RSQ.IN$_j$. The instruction destined for $\mathsf{FU_j}$ must be mapped on to the reservation stations write busses $\mathsf{RSQ_j.W_\star.\star}$. We control this mapping by the signal $\mathsf{i2RSQ2W_{i,j,k}}$; $\mathsf{i2RSQ2W_{i,j,k}}$ indicates that instruction $\mathsf{i_i}$ forwards its operand to the k–th write bus of the $\mathsf{RSQ_j}$.

Instruction $\mathsf{i_i}$ is mapped on the k–th write bus of $\mathsf{RSQ_j}$, iff it requests for $\mathsf{FU_j}$ and is the $(k+1)$–th candidate in doing so. Since $\mathsf{i2RSQ_{i,j}.cand_\star}$, defined in section 3.9.2 on page 52, is a half-unary encoding of the candidate number, we have

$$\mathsf{i2RSQ2W_{i,j,\star}} \quad := \quad \mathsf{i_i.FU_j} \wedge \mathrm{markLastOne}\left(\mathsf{i2RSQ_{i,j}.cand_\star}\right)$$

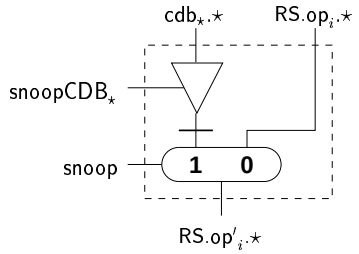

Figure 3.13: Common data bus snooping

The interface of the reservation station queues requires us to indicate the write requests. The k–th write bus of $\mathsf{RSQ_j}$ is requested, if there exists an instruction actually *issuing* on the k–th write bus of $\mathsf{RSQ_j}$. So:

$$\mathsf{RSQ_j.W.req_k} \quad := \quad \left(\bigvee \mathsf{i2RSQ2W_{\star,j,k}} \wedge \mathsf{i_\star.issue}\right)$$

3.10.2 Snooping for source operands

This section implements the common data bus snooping protocol, algorithm 2, p. 17.

Have a reservation station RS and one of its operands $\mathsf{RS.op_i}$. To scan for the presence of the operand's tag on $\mathsf{CDB_j}$ we define the signal $\mathsf{RS.op_i.snoopCDB_j}$ as follows:

$$\begin{aligned}\mathsf{RS.op_1.snoopCDB_j} \quad := \quad & ((\mathsf{RS_i.op_1.valid}\,, 1, \mathsf{RS_i.op_1.tag_\star}) \\ & = (0, \mathsf{CDB_j.valid}\,, \mathsf{CDB_j.tag_\star}))\end{aligned}$$

The reservation snoops an operand, if one of the signals $\mathsf{RS.op_i.snoopCDB_j}$ is active:

$$\mathsf{RS.op_i.snoop} \quad := \quad \left(\bigvee \mathsf{RS.op_i.snoopCDB_\star}\right)$$

The following equation is used to update the contents of the $\mathsf{RS.op_i.valid}$ register:

$$\mathsf{RS.op_i.valid'} \quad := \quad \mathsf{RS.op_i.valid} \vee \mathsf{RS.op_i.snoop}$$

The definition of these signals justifies the implementation the update circuits for the data fields given in figure 3.13. The signal $\mathsf{snoopCDB_j}$ is used as output enable signal to drive $\mathsf{CDB_j.\star}$ on an internal update bus. The mux selects between the internal update bus and the old data $\mathsf{RS.op_i.\star}$ with the signal $\mathsf{RS.op_i.snoop}$.

Execution Stage

3.11 Function Unit Environments

3.11.1 Arithmetical and Logical Functional Unit

The arithmetical and logical functional unit (ALU) executes the integer compute instructions. For the distinction of these operations, the following items need to be stored in its reservation stations:

$\mathsf{opc}_{4..0}$	Operation
00000	left shift
00010	right shift
00011	arithmetic right shift
10000	add without overflow
10001	add with overflow
10010	subtract without overflow
10011	subtract with overflow
10100	bitwise AND
10101	bitwise OR
10110	bitwise XOR
10111	load high
11001	test "$>$"
11010	test "$=$"
11011	test "$\geq$"
11100	test "$<$"
11101	test "$\neq$"
11110	test "$\leq$"

Table 3.9: Encoding of the ALU operations

- The operation code $\mathsf{op}_\star \in \{0,1\}^5$. The operation code is gathered from the instruction's opcode. Let instruction i_i issue on the ALU. Then:

$$\begin{aligned} \mathsf{op}_4 &:= (\mathsf{i}_\mathsf{i}.\mathsf{itype}\ ?\ 1 : \mathsf{opc}_5) \\ \mathsf{op}_3 &:= (\mathsf{i}_\mathsf{i}.\mathsf{itype}\ ?\ \mathsf{opc}_{30} : \mathsf{opc}_3) \\ \mathsf{op}_2 &:= (\mathsf{i}_\mathsf{i}.\mathsf{itype}\ ?\ \mathsf{opc}_{28} : \mathsf{opc}_2 \wedge \mathsf{opc}_5) \\ \mathsf{op}_{1..0} &:= (\mathsf{i}_\mathsf{i}.\mathsf{itype}\ ?\ \mathsf{opc}_{27..26} : \mathsf{opc}_{1..0}) \end{aligned}$$

  Table 3.9 shows an the encoding of the ALU operations.

- Signals movi2s and movi2s indicate special move operations.

The design of the ALU is copied from [MP95, MP00] and is therefore not presented here.

3.11.2 Floating Point Functional Units

The machines has five floating point units for IEEE-compliant addition, multiplication, subtraction, relation checking and conversion. We use the same configuration as presented in [Krö99].

The design of the floating point functional units lies beyond the scope of this thesis. The design of the units is treated in [Lei99, MP00] from which they are copied. [Krö99] provides more detail on the integration of the FPUs in Tomasulo design.

3.11.3 Data Memory Functional Unit

The data memory functional unit can be copied from [Krö99]. We only describe here the structure of its reservation station queue, containing the modified superscalar dispatch protocol.

Reservation Station Entry

A reservation station $\mathsf{RS_i}$ for the data memory functional unit contains the following items:

- The first operand, contained in $\mathsf{op_1.data_\star}$, is used as address operand for a memory access. On issue, $\mathsf{op_1.data_\star}$ is initialized with the sum of the immediate constant and the first operand:

$$\mathsf{op_1.data_\star} \quad := \quad \mathrm{add}\,(\mathsf{i_i.op_1.data_\star}\,,\mathsf{i_i.co_\star})$$

  This sum equals the immediate constant, if the first operand was not valid on issue. Eventually, this operand will be snooped and then added to $\mathsf{op_1.data_\star}$. This process is defined below.

- Items $\mathsf{op_2.valid}$ and $\mathsf{op_2.data_\star}$ are the standard fields for the second source operand.

- Item load is active for a load instruction, inactive for a store instruction.

- Item fp signals a floating point instruction.

- Item db signals a double floating point operand.

- The least significant bit of the second operand's address, $\mathsf{op_2.a_0}$, is used for the single adjustment of floating point operands: single operands with $\mathsf{op_2.a_0} = 1$ arriving on $\mathsf{op_2.data_{63..32}}$ will be shifted to the lower portion of the bus on entering the functional unit.

- The least significant bit of the destination operand's address, $\mathsf{d.a_0}$, is likewise used for single adjustment on leaving the functional unit.

- Instruction opcode bits $\mathsf{opc_{28..26}}$ encode the operand width and whether the operation is signed or unsigned.

Operand Snooping

We only describe snooping for the address operand. Let $\mathsf{RS_i}$ be the i–th reservation station in the data memory reservation station queue. The signal $\mathsf{RS_i.snoopCDB_j}$ is defined to detect the operand's tag on $\mathsf{CDB_j}$:

$$\begin{aligned}\mathsf{RS_i.op_1.snoopCDB_j} \quad := \quad & ((\mathsf{RS_i.op_1.valid}\,, 1, \mathsf{RS_i.op_1.tag_\star}) \\ & = (0, \mathsf{CDB_j.valid}\,, \mathsf{CDB_j.tag_\star}))\end{aligned}$$

Operand $\mathsf{RS_i.op_1}$ can be snooped, if one of these signals is active:

$$\mathsf{RS_i.op_1.snoop} \quad := \quad \left(\bigvee \mathsf{RS_i.op_1.snoopCDB_\star}\right)$$

The new valid register is computed as:

$$\mathsf{RS_i.op_1.valid} \quad := \quad \mathsf{RS_i.op_1.valid} \vee \mathsf{RS_i.op_1.snoop}$$

The update circuit for the $\mathsf{RS_i.op_1.data_\star}$ field is given in figure 3.14. The signal $\mathsf{snoopCDB_j}$ is used as output enable signal to drive $\mathsf{CDB_j.data_\star}$ on the input of a 32-bit adder $\mathsf{ADD_{32}}$. The other operand is $\mathsf{RS_i.op_1.data_\star}$. The adder's result is the memory access address, in case that snoop is valid.

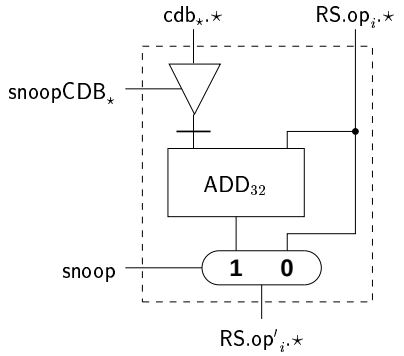

Figure 3.14: Common data bus snooping for the data memory FU

Dispatch Protocol

The interface of the reservation station queue requires the definition of an eligible for read-out signals $\mathsf{e4ro_i}$ for each reservation station $\mathsf{RS_i}$. These signals constitute the dispatch protocol.

The dispatch protocol used herein is based on [Mül97]. It is defined separately for load and store instructions:

- A load instruction may dispatch if the following two conditions are met:
  - The operands are valid, i.e. $\mathsf{RS_i.op_1.valid} \wedge \mathsf{RS_i.op_2.valid}$.
  - Each previous reservation station contains either also a load or its address does not overlap with $\mathsf{RS_i.op_1.data_\star}$:

    $$\forall\, j < i : \mathsf{RS_j.load} \vee \left(\overline{\mathsf{overlap\,(i\,,j)}} \wedge \mathsf{RS_j.op_1.valid}\right)$$

    The notion that two addresses overlap is defined by the memory access width. The memory is accessed in 64-bit portions for double operands and in 32-bit portions for all other memory operations. Therefore, if one instruction accesses a 64-bit portion, the accesses overlap, if their upper 29 address bits are the equal. If both instructions are not double, i.e. address 32-bit portions of the memory, their region overlaps, if their upper 30 address bits are equal. These two conditions can be comprised in the following equation of the overlap (i , j) macro:

    $$\begin{aligned}\mathsf{overlap\,(i\,,j)} \;&:=\; (\mathsf{RS_i.op_1.data_{31..3}} = \mathsf{RS_j.op_1.data_{31..3}}) \\ &\wedge\; ((\mathsf{RS_i.op_1.data_2} = \mathsf{RS_j.op_1.data_2}) \vee (\mathsf{RS_i.db} \vee \mathsf{RS_j.db}))\end{aligned}$$

- Stores may only dispatch, if all preceding instructions have completed. This is the case, if the tag matches the (foremost) head of the ROB queue, $\mathsf{RS_i.tag_\star} = \mathsf{ROB.head_{0,\star}}$. This condition implies the following two, noteworthy observations:
  - Due to the organization of reservation station queues, the store instructions is the first in the reservation station queue, since all previous instructions must have completed. Especially, the condition $\mathsf{RS_i.tag_\star} = \mathsf{ROB.head_{0,\star}}$ can only be true for $i = 0$ and automatically evaluates to false for $i > 0$.
  - The operands of the store are valid since the preceding instructions have completed and therefore broadcast their results on the common data busses.

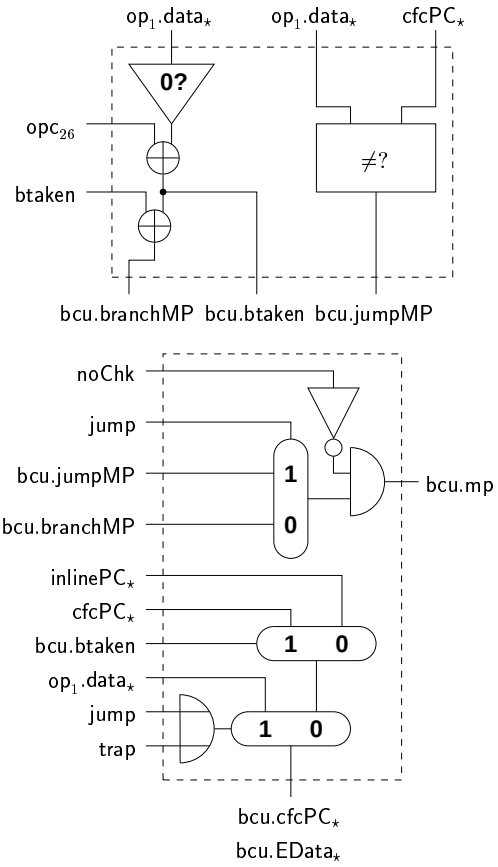

Figure 3.15: Branch checker unit

We summarize the load and store case for the first reservation station:

$$\begin{aligned} \mathsf{e4ro}_0 \quad &:= \quad (\mathsf{ROB.head}_{0,\star} = \mathsf{RS}_0.\mathsf{tag}_\star) \\ &\vee \quad (\mathsf{RS}_0.\mathsf{load} \wedge \mathsf{RS}_0.\mathsf{op}_1.\mathsf{valid} \wedge \mathsf{RS}_0.\mathsf{op}_2.\mathsf{valid}) \end{aligned}$$

The succeeding reservation stations $\mathsf{RS_i}$ for $i > 0$ may dispatch on the following condition:

$$\begin{aligned} \mathsf{e4ro_i} \quad &:= \quad \mathsf{RS_i.load} \wedge \mathsf{RS_i.op_1.valid} \wedge \mathsf{RS_i.op_2.valid} \\ &\wedge \quad \bigwedge_{j<i} \mathsf{RS_j.load} \vee \left(\mathsf{RS_j.op_1.valid} \wedge \overline{\mathrm{overlap}\,(\mathsf{i}\,,\mathsf{j})}\right) \end{aligned}$$

3.11.4 Branch Checker Unit

The definition of the rollback protocol (9, p. 28) states, that the predicted next PC has to be compared with the computed next PC to verify a prediction. On a failing verification, the computed next PC has to be reported back to the instruction fetch unit. As has already been outlined in the overview of the implemented instruction fetch mechanism, section 3.3, the verification of CFI predictions is the task of a special functional unit called the branch checker unit, BCU.

A reservation station RS of the branch checker unit contains the standard fields valid, $\mathsf{op_1.valid}$, $\mathsf{op_1.tag_\star}$ and $\mathsf{op_1.data_\star}$. In addition, we have the following items to identify control flow instructions and their predictions:

- Signal jump indicates the presence of a jump instruction on value 1 and of a branch instruction on value 0.
- Signal cfc equals 1, if a CFI was predicted to cause a control flow change. In connection with jump = 0, this indicates a taken branch.
- Opcode bit opc_{26} is stored to differentiate between the branch types `eqz` and `nez`.
- The signal noChk indicates that no verification has to be made for the instruction. Currently, there are three types of instructions just being passed through the BCU without check (cf. 3.8): jump instructions with immediate relative offset, the `trap` instruction and the `rfe` instruction. No speculation is allowed for these instructions, this behaviour may change for branch prediction.
- Signal trap indicates the presence of a `trap` instruction in the reservation station. A `trap` instruction passes its immediate constant via $\mathsf{op}_1.\mathsf{data}_\star$ and the ROB to the EData register. This implementation has been chosen to save an additional ROB write port for the issue.
- The $\mathsf{inlinePC}_\star$ register contains the inline sequential PC of the instruction.
- The $\mathsf{cfcPC}_\star$ register contains the (predicted) PC of a control flow change. In case of a branch instruction, it is the branch target; in case of a jump instruction, it is the predicted PC of a control flow change.

The branch checker unit operates, according to the definition of the DLX ISA, in two different situations:

- On a jump instruction, jump = 1, a valid operand $\mathsf{RS.op}_1.\mathsf{data}_\star$ contains the computed next PC. The predicted next PC is stored in $\mathsf{RS.cfcPC}_\star$. Testing these two values for equality defines the signal

$$\mathsf{bcu.jumpMP} \quad := \quad \left(\bigvee \mathsf{RS.op}_1.\mathsf{data}_\star \, , \mathsf{RS.cfcPC}_\star\right)$$

- On a branch instruction, jump = 0, we first test the valid operand $\mathsf{op}_1.\mathsf{data}_\star$ for zero:

$$\mathsf{bcu.op}_1.\mathsf{eqz} \quad := \quad \overline{\left(\bigvee \mathsf{RS.op}_1.\mathsf{data}_\star\right)}$$

  Since $\mathsf{RS.opc}_{26} = 1$ for a `nez` branch and $\mathsf{RS.opc}_{26} = 0$ for a `eqz` branch, the correct branch result is computed by

$$\mathsf{bcu.btaken} \quad := \quad \mathsf{bcu.op}_1.\mathsf{eqz} \oplus \mathsf{RS.opc}_{26}$$

  The misprediction of a branch is therefore indicated by

$$\mathsf{bcu.branchMP} \quad := \quad \mathsf{RS.cfc} \oplus \mathsf{bcu.btaken}$$

The interface of the branch checker unit is completed with the following definitions:

$$\begin{aligned} \mathsf{bcu.cfcPC}_\star \quad := \quad & (\mathsf{RS.jump} \vee \mathsf{RS.trap?}\, \mathsf{RS.op}_1.\mathsf{data}_\star \\ & : (\mathsf{bcu.btaken?}\, \mathsf{RS.cfcPC}_\star : \mathsf{RS.inlinePC}_\star)) \end{aligned}$$

$$\begin{aligned} \mathsf{bcu.mp} \quad &:= \quad \overline{\mathsf{RS.noChk}} \wedge (\mathsf{RS.jump}\ ?\ \mathsf{bcu.jumpMP}\ :\ \mathsf{bcu.branchMP}) \\ \mathsf{bcu.data}_\star \quad &:= \quad \mathsf{RS.inlinePC}_\star \end{aligned}$$

Refer to figure 3.15 for the actual implementation of the branch checker unit. The computed signals are forwarded to the producer unit and to the branch checker unit bus $\mathsf{bcu}.\star$.

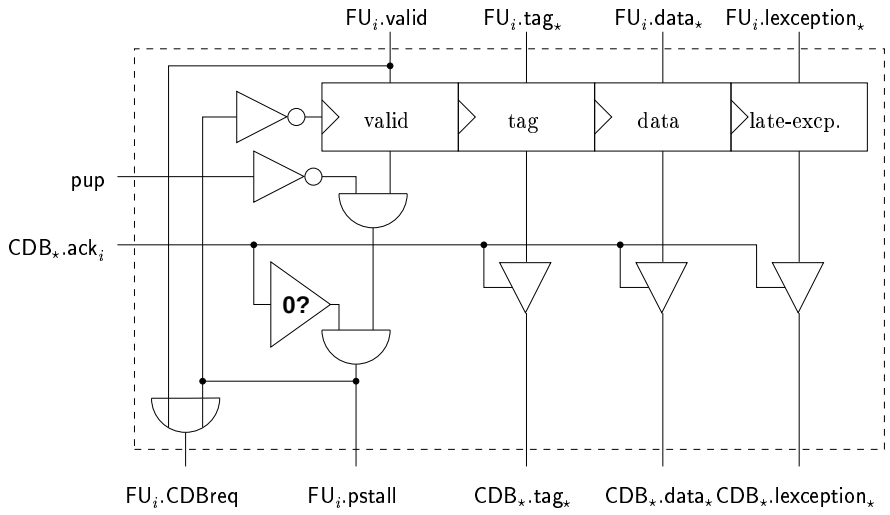

Figure 3.16: Result producer environment

3.11.5 Producer Environments

Implementation

Associated with each functional unit $\mathsf{FU_i}$ is a producer environment. The producer buffers results from the functional unit, requests for a common data bus and forwards the result to the CDB on acknowledgement. Figure 3.16 shows the result producer environment for $\mathsf{FU_i}$. The implementation features a central control register valid, which operation will now be described.

The functional unit signals a computed result by setting the bit $\mathsf{FU_i.valid}$. The producer answer with $\mathsf{FU_i.pstall} = 1$ in case that it cannot take the result, because it is already full and did not receive a CDB acknowledgment. With $\mathsf{CDB_j.ack_i}$ signalling an acknowledgment from the j–th CDB for producer i, we define:

$$\mathsf{FU_i.pstall} \quad := \quad (\mathsf{PROD_i.valid} \wedge \overline{\mathsf{pup}}) \wedge \overline{\left(\bigvee \mathsf{CDB_\star.ack_i}\right)}$$

The valid register is clocked, whenever no producer stall is generated:

$$\mathsf{PROD_i.valid.ce} \quad := \quad \overline{\mathsf{FU_i.pstall}}$$

On clocking the valid register, it receives the functional unit's valid bit $\mathsf{FU_i.valid}$.

A CDB is requested via the signal $\mathsf{FU_i.CDBreq}$ in two situation. First, the producer may contain valid data for which it did not receive an acknowledgment. This is the case, if $\mathsf{FU_i.pstall} = 1$. Second, a valid FU result requests the CDB. So $\mathsf{FU_i.CDBreq}$ is defined as:

$$\mathsf{FU_i.CDBreq} \quad := \quad \mathsf{FU_i.valid} \vee \mathsf{FU_i.pstall}$$

Producer Data Convention

The implementation of our reorder buffer given below requires, that each producer *always* produces 64-bit data. With double IEEE values, this is automatically the case. Single IEEE values, and general purpose or special purpose register result data is 32-bit and needs to be duplicated on the higher 32 bits of the operand bus. So:

$$\mathsf{PROD.data}_{63\cdot\cdot 0} = \begin{cases} \mathsf{FU.data}_{63\cdot\cdot 0} & \text{for 64-bit data} \\ (\mathsf{FU.data}_{31\cdot\cdot 0}, \mathsf{FU.data}_{31\cdot\cdot 0}) & \text{for 32-bit data} \end{cases}$$

3.12 CDB Control Environment

Let k equal the number of common data busses and n equal the number of producers. Already introduced in the producer environment, section 3.11.5, the producer requests the CDB via the signal $\mathsf{FU}_\star.\mathsf{CDBreq}$. The task of the CDB control is to fairly select each round up to k producers for acknowledgments. To satisfy the fairness of the selection, we use a k-from-n multiported round-robin selector $k{:}n$–MPRRS (cf. section 5.1). Taking $\mathsf{FU}_\star.\mathsf{CDBreq}$ as input, this circuit computes the acknowledgment array $\mathsf{ack}_{\star,\star} \in \{0,1\}^{k \cdot n}$ such that the i–th row $\mathsf{ack}_{\mathsf{i},\star}$ of this array contains a half-unary encoding of the i–th acknowledgment.

So, the signal $\mathsf{CDB_i}.\mathsf{ack_j}$, acknowledging $\mathsf{CDB_i}$ for producer $\mathsf{FU_j}$ can be defined via the following equation:

$$\mathsf{CDB_i}.\mathsf{ack}_\star \quad := \quad \mathrm{markLastOne}\,(\mathsf{ack}_{\mathsf{i},\star})$$

Additionally, the $k{:}n$–MPRRS returns the signal $\mathsf{numAck}_\star$, a half-unary encoding of the number of acknowledged candidates. This signal defines the validness of the CDB for the computed acknowledgments:

$$\mathsf{CDB_i}.\mathsf{valid} \quad := \quad \mathsf{numAck}_\star$$

According to the semantics of the producer bus requests, requesting the CDB for the next round, these signals, $\mathsf{CDB}_\star.\mathsf{ack}_\star$ and $\mathsf{CDB}_\star.\mathsf{valid}$, are buffered in flip-flops.

Completion Stage

3.13 Reorder Buffer Environment

The reorder buffer (ROB) guarantees in-order retirement and allows the speculative execution of instruction streams. These properties are necessary to realize precise interrupts and branch prediction. The idea of a the reorder buffer was introduced in [SP88].

The ROB presented in this design is based on a multi-ported queue implemented with multiported RAM (section 5.3 describes the implementation of such a queue). In addition to writing entries at the queue tail and reading them at the queue head, the queue must have update ports to read out and write to entries in the "middle" of the queue.

The ROB stores $\Theta = 2^\nu$ entries. An entry is a dynamically updated data structure allocated for each issued (but not retired) instruction. The entry stores information about the position, the result and the exception conditions of an instruction. It is referenced by a state-unique id, the instruction tag; a number in the set $\{0, \ldots, 2^\nu - 1\}$.

Table 3.10 contains a complete list of the ROB entry components. Putting the components into groups provides a way to specify, which item is accessed in which contexts. Of these contexts there are four: During *issue* entries are written in the queue (at the tail). Also during issue, operands are *forwarded* from the ROB to the source operand data generation. Addressing is done via the instruction's tag, which can be obtained by the producer tables. Third, a *completing* instruction on a CDB must store its results and its exception status in the reorder buffer. Here, the "write address" is given by the instruction's tag available on the CDB. The last context in which the ROB is accessed is the *retirement* of an instruction. The ROB queue is read out (at its head) for this purpose.

Looking at the components of a ROB entry reveals that they are not accessed in every context. The implementation of the ROB can exploit this information to

Group	Name	Width	Purpose
valid	valid	1	valid data item
dataLow	$\mathsf{data}_{31..0}$	32	low 32 bit of result
dataHigh	$\mathsf{data}_{63..32}$	32	high 32 bit of result
onCompletion	dmal	1	misaligned DMem access
	Dpf	1	data memory page fault
	ovf	1	overflow in ALU instruction
	$\mathsf{IEEEf}_\star$	5	IEEE flags
	mp	1	mispredicted CFI
	$\mathsf{EData}_\star$	32	exception data
onIssue	ill	1	illegal instructions
	imal	1	misaligned IMem access
	Ipf	1	IMem page fault
	trap	1	trap instruction
	uFOP	1	unimplemented fp instruction
	$\mathsf{d.a}_\star$	4	destination register address
	d.db	1	double precision
	d.fpr	1	FPR destination
	d.spr	1	SPR destination
	d.gpr	1	GPR destination
	$\mathsf{PC}_\star$	32	instruction's PC

Table 3.10: Components of a ROB entry

Group	Width	Ports	Description
valid	1	$1I\cdot$ W:	issue
		$(4I+2)\cdot$ RM:	forw. $\mathsf{op}_{1..2,\mathrm{lo}}$, $\mathsf{op}_{1..2,\mathrm{hi}}$, RM, MSK
		$1C\cdot$ WM:	completion
		$1R\cdot$ R:	retirement
dataLow	32	$(2I+1)\cdot$ RM:	forw. $\mathsf{op}_{1,\mathrm{lo}}$, $\mathsf{op}_{2,\mathrm{lo}}$, RM
		$1C\cdot$ WM:	completion
		$1R\cdot$ R:	retirement
dataHigh	32	$(2I+1)\cdot$ RM:	forw. $\mathsf{op}_{1,\mathrm{hi}}$, $\mathsf{op}_{2,\mathrm{hi}}$, MSK
		$1C\cdot$ WM:	completion
		$1R\cdot$ R:	retirement
onCompletion	41	$1C\cdot$ WM:	completion
		$1R\cdot$ R:	retirement
onIssue	45	$1I\cdot$ W:	issue
		$1R\cdot$ R:	retirement

Table 3.11: ROB ports

reduce cost and delay. Critical path analysis in [Krö99] suggests that it may be worthwhile to reduce the delay of the ROB. Table 3.11 contains the of the analysis of the access structure. The table lists for each group the accessing contexts as well as the modes of access (write at tail, read from middle, write in middle, read from head). The constants I, C and R specify the maximum number of issuing, completing and retiring instructions respectively. This information is used for the cost and delay calculation, chapter 4.

3.13.1 ROB Queue Control

The queue is controlled by a number of signals. For queue write operations, the bus $i_\star$.issue supplies the number of instruction to be written in half-unary encoding. The queue returns the address of the i–th entry written on the bus $\mathsf{ROB.tail}_{i,\star}$. This address is the *tag* of the i–th instruction and is stored in the producer table.

For read operations, the retirement signals $r_\star$.retire generated from the retirement control below are used as read request signals. As shown below, they provide the number of retiring instructions in half-unary encoding; r_i.retire $= 1$ iff the instruction on the i–th retire bus can retire. The tag of the i–th retiring instructions is returned on the bus $\mathsf{ROB.head}_{i,\star}$.

The read-in-the-middle and the write-in-the-middle ports are directly controlled by appropriate address, request / enable and data-in signals, as specified below.

In addition to this, the queue returns its fill status on bus $\mathsf{ROB.full}_\star$ that contains a half-unary encoding of the number of occupied entries:

$$i = \#\text{ of occupied entries} \iff \mathsf{ROB.full}_\star = 0^{\Theta-i}1^i$$

The queue is initialized on a roll-back condition:

$$\mathsf{ROB.init} := \mathsf{rollback}$$

3.13.2 Issue

An issuing instruction i_i sets the valid and the onIssue group of the ROB entry located at the position $\mathsf{ROB.tail}_{i,\star}$. The components ill, imal, IPf, trap, uFOP, d.$\star$ and $\mathsf{PC}_\star$ are set to their corresponding signals out of the instruction environment. The component bj is set to the term i_i.jumpR $\vee$ i_i.jump $\vee$ i_i.jumpR. The component valid is set to the signal i_i.noFU.

3.13.3 Forwarding

In section 3.9.2 the source operand data generation was described. Every instruction i_i requests up to six 32–bit operands to be forwarded from the ROB. These are $\mathsf{op}_{1,\mathrm{lo}}$, $\mathsf{op}_{1,\mathrm{hi}}$, $\mathsf{op}_{2,\mathrm{lo}}$, $\mathsf{op}_{2,\mathrm{hi}}$, the rounding mode RM and the mask register MSK. For each operand the data and the valid signals have to be requested.

The producer data convention (section 3.11.5) states, that 32-bit results can be found in both the dataHigh and the dataLow component. This justifies forwarding $\mathsf{op}_{1,\mathrm{lo}}$, $\mathsf{op}_{2,\mathrm{lo}}$ and RM from the dataLow group and forwarding $\mathsf{op}_{1,\mathrm{hi}}$, $\mathsf{op}_{2,\mathrm{hi}}$ and MSK from the dataHigh group.

Note that without this simple convention, the number of ports is increased for the dataLow group resulting in an imbalanced port distribution for dataLow and dataHigh. There are two reasons for this:

- *Without the convention*, the rounding mode RM and the mask register MSK would both have to be forwarded from the dataLow group. Therefore a read port would have to be removed from dataHigh and moved to the dataLow group.

- Double floating point source operands may have two producing instructions: one producing the lower 32-bit half of the operand and another one producing the upper 32-bit half of the operand. If the producing instructions are both 32-bit instructions, their result will be stored in the dataLow group only *without the convention*. Again, the dataLow group would have to provide additional read ports.

3.13.4 Completion

For each valid CDB, the data must be updated in the valid, data and late-exception groups of the corresponding ROB entry. The $\mathsf{CDB_i.valid}$ signal provides the write enable signal and the $\mathsf{CDB_i.tag_\star}$ bus provides the address. The valid component is set to 1. The dataLow and dataHigh components are filled with $\mathsf{CDB_i.data_{31..0}}$ and $\mathsf{CDB_i.data_{63..32}}$ respectively. The onCompletion group is filled with $\mathsf{CDB_i}$'s late exception signals, Dmal, Dpf, ovf, $\mathsf{IEEEf_\star}$, the misprediction signal mp and the exception data $\mathsf{EData_\star}$.

3.13.5 Retirement

During retirement exception conditions and branch predictions are checked. Define $\mathsf{r_i}$ to be the i–th read-out bus from the ROB. For each $\mathsf{r_i}$ the following important interface signals must be generated:

- $\mathsf{r_i.retire}$ signals that the instruction retires
- $\mathsf{r_i.wb}$ indicates that the result of the instruction actually has to be written back
- $\mathsf{r_i.ilRQ}$ signals the detection of an internal interrupt at instruction $\mathsf{r_i}$

Table 3.12 contains a listing of all the possible interrupts. These fall into three categories. The reset exception has the highest priority and is the only unmaskable external interrupt. Exceptions 1 to 12 are the internal exceptions. They are stored in the corresponding ROB entry and are exceptions that occur during the execution of an instruction. The exceptions 13 to 31 are the external I/O exceptions. They are triggered by the external event lines $\mathsf{ex_{18..0}}$. The processor detects external interrupt conditions for the *last retiring instruction* (LRI) in a cycle, i.e. the instruction on $\mathsf{r_i}$ with $\mathsf{r_i.retire} = 1$ and i maximal.

Mispredicted CFIs are handled in much the same way. So, instruction retiring also stops if $\mathsf{r_i.mp} = 1$. A mispredicted retiring instruction may only be accompanied by external interrupts or a reset interrupt. All the other interrupts either do not occur in connection with DLX CFIs or their occurrence would have caused the instruction to be transformed in a machine nop (for instruction memory misalignment or instruction memory page fault).

In the following two sections, the computations of control signals for external interrupts and internal interrupts / branch mispredictions is described. Then, in the last section, these results are used to complete the retirement of instructions.

External Interrupts Control Signals

The external interrupt processing requires computing the register $\mathsf{MCA'_{31..13}}$ and the signal elRQ signalling an external interrupt request. As external interrupts will be associated with the last retiring instruction $\mathsf{lri.\star}$, we define:

$$\begin{aligned} \mathsf{lri.MCA_{31..13}} &:= \mathsf{ex_\star} \wedge \mathsf{SR_{31..13}} \\ \mathsf{lri.elRQ} &:= \left(\bigvee \mathsf{lri.MCA_{31..13}}\right) \end{aligned}$$

Internal Interrupt Control Signals

For each retirement bus $\mathsf{r_i}$ a number of signals have to be computed. First, the masked cause bus, $\mathsf{r_i.MCA_\star}$, is generated, by taking the exception signals and masking them out with the status register $\mathsf{SR_\star}$:

Interrupt	Symbol	Priority	Resume	Maskable	External
reset	reset	0	abort	no	yes
illegal instruction	ill	1	abort	no	no
misaligned access	mal	2			
page fault IM	Ipf	3	repeat		
page fault DM	Dpf	4			
trap	trap	5	continue		
FXU overflow	ovf	6	continue	yes	
FPU overflow	fOVF	7	abort / continue		
FPU underflow	fUNF	8			
FPU inexact result	fINX	9			
FPU divide by zero	fDBZ	10			
FPU invalid operation	fINV	11			
FPU unimplemented	uFOP	12	continue	no	
external I/O	ex_i	$12+i$	continue	yes	yes

Table 3.12: Coding of the interrupts

$$
\begin{aligned}
r_i.MCA_0 &:= pup \\
r_i.MCA_1 &:= r_i.ill \\
r_i.MCA_2 &:= r_i.dmal \vee r_i.imal \\
r_i.MCA_3 &:= r_i.Ipf \\
r_i.MCA_4 &:= r_i.Dpf \\
r_i.MCA_5 &:= r_i.trap \\
r_i.MCA_6 &:= SR_6 \wedge r_i.ovf \\
r_i.MCA_{11..7} &:= r_i.IEEEf_\star \wedge SR_{11..7} \\
r_i.MCA_{12} &:= r_i.uFOP
\end{aligned}
$$

Second, the following signals are additionally computed for each retiring instruction r_i:

$$
\begin{aligned}
r_i.eligible &:= ROB.full_i \wedge r_i.valid \\
r_i.SRdest &:= r_i.d.spr \wedge (r_i.d.a_\star = *0000) \\
r_i.ilRQ &:= \left(\bigvee r_i.MCA_{12..0}\right) \\
r_i.wbAux &:= \overline{\left(\bigvee r_i.MCA_{4..0}\right)} \vee r_i.mp
\end{aligned}
$$

The meaning of these signals is as follows: $r_i.eligible$ indicates that there is an instruction at the i–th head of the ROB and it has already completed and may therefore retire; $r_i.SRdest$ indicates that the instruction writes to the status register $SR_\star$. Only the last retiring instruction in a cycle may write to the status register $SR_\star$, since otherwise interrupts may incorrectly be masked or unmasked. This signal stops instruction retirement beyond r_i. Last, $r_i.ilRQ$ indicates an internal interrupt at the instruction (an interrupt with priority between 0 and 12). Such an interrupt requires write-back iff $r_i.wbAux$ is active; mispredicted CFIs will always be written back. If $r_i.ilRQ = 1$ instruction retirement will also stop.

Completing the Retirement Interface

With the definition

$$r_i.\mathsf{retireVeto} \quad := \quad \overline{r_i.\mathsf{eligible}} \vee r_{i-1}.\mathsf{SRdest} \vee r_{i-1}.\mathsf{ilRQ} \vee r_{i-1}.\mathsf{mp}$$

the $r_i.\mathsf{retire}$ signals can be computed by a parallel-prefix OR:

$$r_i.\mathsf{retire} \quad := \quad \overline{\left(\bigvee r_{j-1..0}.\mathsf{retireVeto}\right)}$$

The $r_\star.\mathsf{retire}$ bus contains the index of the last retiring instruction in half-unary encoding. We convert this information into a unary encoding:

$$r_\star.\mathsf{lri} \quad := \quad \mathrm{markLastOne}\,(r_\star.\mathsf{retire})$$

Thereby, $r_i.\mathsf{lri} = 1$ iff the i–th instruction is the last retiring instructions. This signal is used as an output enable signal to construct the bus of the last retiring instruction, $\mathsf{lri}.\star$, from the busses $r_\star.\star$.

Furthermore, we compute the write-back signals $r_\star.\mathsf{wb}$; instruction r_i writes back iff it not the last retiring instructions or it is the last retiring instructions and requires write back. This is expressed by:

$$r_i.\mathsf{wb} \quad := \quad r_{i+1}.\mathsf{retire} \vee (r_i.\mathsf{retire} \wedge r_i.\mathsf{wbAux})$$

The global signal JISR is set if an external or internal interrupt was found:

$$\mathsf{JISR} \quad := \quad \mathsf{lri.eIRQ} \vee \mathsf{lri.iIRQ}$$

The machine performs a rollback on interrupt or misprediction situation, i.e.:

$$\mathsf{rollback} \quad := \quad \mathsf{JISR} \vee \mathsf{lri.mp}$$

The Write Back Stage

3.14 Register File Environment

The register file environment contains the general purpose register file, the floating point register file and the special purpose register file. The register files are accessed in three different main contexts:

- During issue the source operands are read from the register files if the producer tables indicates their validness. This requires $2I$ read ports since each instruction has two variable-address operands.

- During retirement, the register file environment takes each retirement bus r_j and writes it back to the appropriate register files on activation of $r_j.\mathsf{wb}$. This requires R write ports.

- The special purpose register file has extended modes of access during exception conditions specified in detail in section 3.14.3.

The section proceeds with a description of each register file.

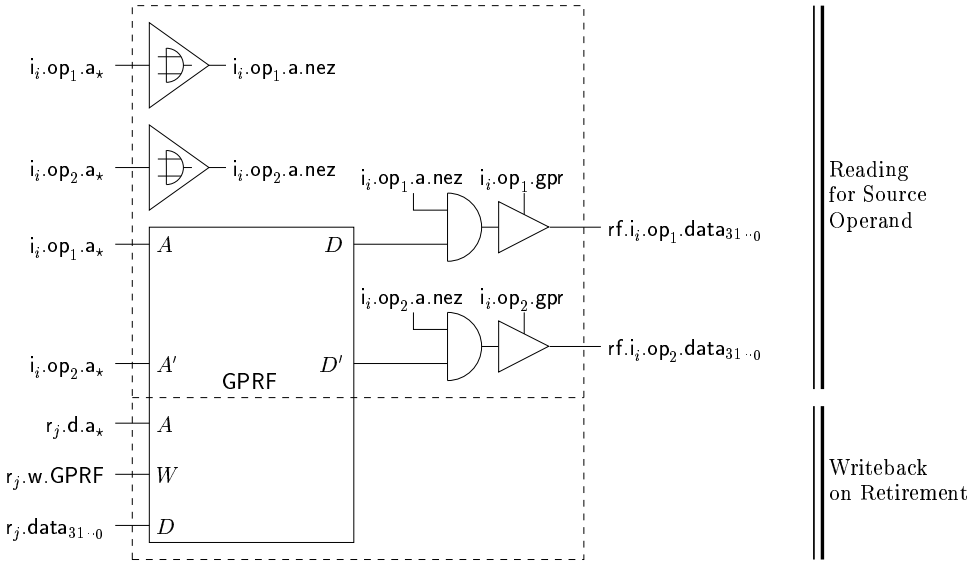

Figure 3.17: Operand reading and retirement ports for the GPRF

3.14.1 General Purpose Register File

The general purpose register file GPRF consists of 32 integer registers having a width of 32 bits. A 32×32–RAM is used for the implementation.

This RAM has to provide two read ports per instruction for the operands $i_i.op_1$ and $i_i.op_2$. These ports are addressed by $i_i.op_1.a_\star$ and $i_i.op_2$ respectively. Since accesses to GPR R_0 must always return 0^{32}, we check if

$$i_i.op_j.a.nez \quad := \quad \left(\bigvee i_i.op_j.a_\star\right)$$

and force the output of the RAMs to zero in this case. The data is driven on the operand bus, if GPR data was requested, as indicated by $i_i.op_1.gpr$ and $i_i.op_2.gpr$. The arrangement is shown in figure 3.17.

Additionally, the GPR must also have one write port per retirement bus r_j. This port takes the address $r_j.d.a_\star$ and the input data $r_j.data_{31..0}$. The write enable signal $r_j.w.GPRF$ are computed as follows:

$$r_j.w.GPRF \quad := \quad r_j.d.gpr \wedge r_j.wb$$

Again, figure 3.17 shows the arrangement.

3.14.2 Floating Point Register File

The floating point register file consists of 32 single precision floating point registers, alternatively accessed as 16 double precision floating point register. It is constructed of two 16×32–RAMs, named FPRF0 and FPRF1. The FPRF0 RAM stores the single precision registers with even addresses, the FPRF1 RAM stores the single precision registers with odd addresses. Double precision registers are decomposed in two 32–bit parts; the low part is stored in FPRF0, the high part is stored in FPRF1.

As with the GPRF, two read ports per instruction for the operands $i_i.op_1$ and $i_i.op_2$ are provided. These ports are addressed by $i_i.op_1.a_{4..1}$ and $i_i.op_2.a_{4..1}$ respectively. The data is driven on the operand bus, if a floating point register was requested, as indicated by $i_i.op_1.fpr$ and $i_i.op_2.fpr$. The arrangement is shown in figure 3.18.

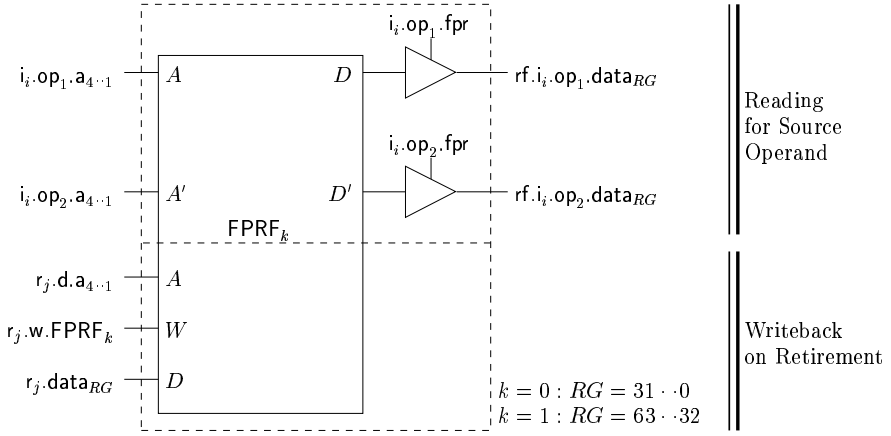

Figure 3.18: Operand reading and retirement ports for the FPRF

Special Purpose Register	Width	Purpose
$\mathsf{SPR}_{0,\star} \equiv \mathsf{SR}_\star$	32	Status register (interrupt mask)
$\mathsf{SPR}_{1,\star} \equiv \mathsf{ESR}_\star$	32	Exception status register
$\mathsf{SPR}_{2,\star} \equiv \mathsf{EPC}_\star$	32	Exception program counter
$\mathsf{SPR}_{3,\star} \equiv \mathsf{EPCn}_\star$	32	Exception program counter 2 *obsolete!*
$\mathsf{SPR}_{4,\star} \equiv \mathsf{ECA}_\star$	32	Exception cause register
$\mathsf{SPR}_{5,\star} \equiv \mathsf{EData}_\star$	32	Exception data register
$\mathsf{SPR}_{6,\star} \equiv \mathsf{RM}_\star$	2	Rounding mode
$\mathsf{SPR}_{7,\star} \equiv \mathsf{IEEEf}_\star$	5	IEEE interrupt flags
$\mathsf{SPR}_{8,\star} \equiv \mathsf{FCC}$	1	Floating point comparison flags

Table 3.13: Special purpose registers

One write port is needed for each retirement bus $\mathsf{r_j}$. The address is provided by $\mathsf{r_j.d.a}_{4\cdot\cdot1}$. According to the number format indicated by $\mathsf{r_j.db}$ and the lowest address bit $\mathsf{r_j.d.a}_0$ we define the write enable signals for the FPRi as:

$$\mathsf{r_j.w.FPRF_i} \quad := \quad \mathsf{r_j.d.fpr} \wedge (\mathsf{r_j.d.db} \vee (\mathsf{r_j.d.a}_0 = i)) \wedge \mathsf{r_j.wb}$$

Again, figure 3.18 shows the arrangement.

3.14.3 Special Purpose Register File

The special purpose register file SPRF consists of nine variable-width registers, as listed in table 3.13. Note that the $\mathsf{EPCn}_\star$ register is specific for the implementation of a delayed branch. Since our processor does not have a delayed branch mechanism it is not used. Numbering of the special purpose registers has been preserved, though, to ensure address register compatibility to the DLX processors of [MP95, MP00, Krö99, Lei99]. The SPRF is build up using flip-flops for the register storage, since the registers must have extended access structures.

Two read ports per instruction for the operands $\mathsf{i_i.op_1}$ and $\mathsf{i_i.op_2}$ are provided.[5] These ports are addressed by $\mathsf{i_i.op_1.a}_\star$ and $\mathsf{i_i.op_2.a}_\star$ respectively. The data is driven on the operand bus, if SPRF data was requested, as indiciated by $\mathsf{i_i.op_1.spr}$ and $\mathsf{i_i.op_2.spr}$. If $\mathsf{i_i.op_1.aun}_{8\cdot\cdot0}$ is a unary encoding of the adress $\mathsf{i_i.op_1.a}_\star$, then the signal

$$\mathsf{i_i.op_1.SPRread_k} \quad := \quad \mathsf{i_i.op_1.spr} \wedge \mathsf{i_i.op_1.aun_k}$$

[5] Instruction access the SPRF only via the first operand, though.

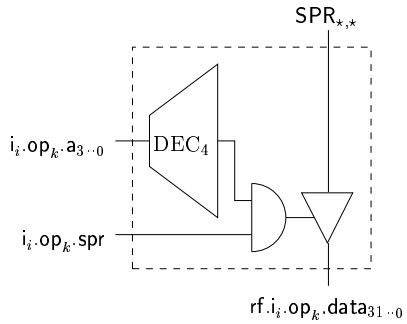

Figure 3.19: Reading from the special purpose register file, $k \in \{0, 1\}$

can be used as an output enable signal to drive special register $\mathsf{SPR}_{k,\star}$ on the operand bus $\mathsf{rf.i_i.op_1.data}_{31..0}$. The same can be done for $\mathsf{i_i.op_2}$. The arrangement is shown in figure 3.19.

Additionally, each instruction might request RM and MSK. These requests can be hard-wired though, since they are fixed address and the underlying implementation is based on flip-flops.

For write purposes we always clock each register and consider three modes of access:

- The first mode is the *update* access. This write access is default for all registers. It feeds the data $\mathsf{SPR_j.upd}_\star$ in register $\mathsf{SPR_j}$. Usually we just have

$$\mathsf{SPR_j.upd}_\star \quad = \quad \mathsf{SPR_j}$$

  which results in $\mathsf{SPR'}_{j,\star} = \mathsf{SPR}_{j,\star}$ in case of update access. The exception is the IEEE mask register $\mathsf{IEEEf}_\star$, which accumulates the mask of all retiring instructions on normal update:

$$\mathsf{IEEEf.upd}_\star \quad = \quad \bigvee_j (\mathsf{r_j.wb} \wedge \mathsf{r_j.IEEEf}_\star)$$

- The second mode of access is the regular write access. Let $\mathsf{r_j.aun}_{8..0}$ be a unary encoding of the retirement address for $\mathsf{r_j}$. Then

$$\mathsf{r_j.SPR.wreq_k} \quad := \quad \mathsf{r_j.d.spr} \wedge \mathsf{r_j.aun_k} \wedge \mathsf{r_j.wb}$$

  indicates a write request for $\mathsf{SPR}_{k,\star}$ by retirement bus $\mathsf{r_j}$. The signal

$$\mathsf{SPR_k.write} \quad := \quad \left(\bigvee \mathsf{r}_\star\mathsf{.SPR.wreq_k}\right)$$

  detects if $\mathsf{SPR}_{k,\star}$ is written to. In this case, the data written must be

$$\mathsf{SPR_k.writeData}_\star \quad = \quad \mathsf{r_j.data} \quad \text{with } j = \max\left\{j' \mid \mathsf{r_j.SPR.wreq_j} = 1\right\}$$

  Layers of muxes are sufficient to solve this equation in circuitry.

- The *exceptional* write access refers to the exception status registers $\mathsf{SPR}_{1,\star}$ to $\mathsf{SPR}_{5,\star}$ only. It is activated on encountering an interrupt situation, JISR = 1.

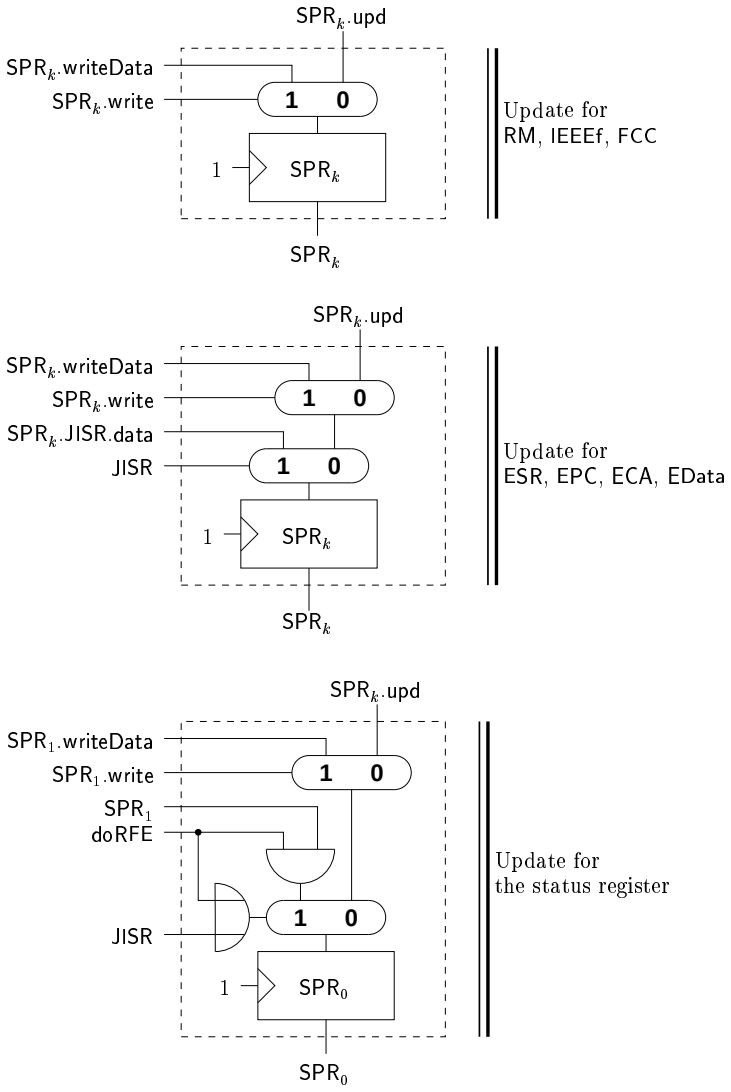

Figure 3.20: Writing to the special purpose register file

The following data is fed into the various special registers:

$$
\begin{aligned}
\mathsf{SR.JISR.data}_\star &:= 0 \\
\mathsf{ESR.JISR.data}_\star &:= \mathsf{SR}_\star \\
\mathsf{Iri.nPC}_\star &:= \mathrm{inc4}\,(\mathsf{Iri.PC}_\star) \\
\mathsf{EPC.JISR.data}_\star &:= (\mathsf{Iri.mp}\ ?\ \mathsf{Iri.EData}_\star\ :\ (\mathsf{Iri.wb}\ ?\ \mathsf{Iri.nPC}_\star\ :\ \mathsf{Iri.PC}_\star)) \\
\mathsf{ECA.JISR.data}_\star &:= \mathsf{Iri.MCA}_\star \\
\mathsf{EData.JISR.data}_\star &:= \mathsf{Iri.EData}_\star
\end{aligned}
$$

- Finally, the status registers SR has to be cleared if doRFE is signalled (cf. 3.9, p. 3.9.2 for the definition of doRFE).

Figure 3.20 shows the update paths for the special purpose registers.

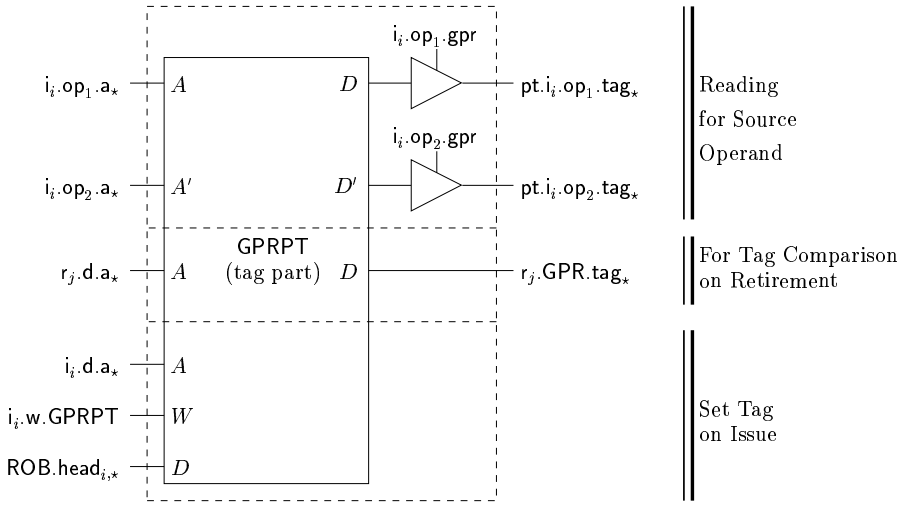

Figure 3.21: Tag information of the GPR producer table

3.15 Producer Table Environment

Producer tables indicate, whether data stored in the register files is valid or being computed in the processor. If the data is still being computed, the producer tables return an identifying tag for the instruction producing the result.

The producer tables are accessed in four contexts:

- First, during issue, information on the valid bit and the tag are taken out of the producer tables for the generation of the source operands i_i.op$_i$.
- Second, the producer tables must be read out during retirement, to determine if a retiring instruction is the producing instruction for its destination register.
- Third, if tags matched in the above context, a retiring instruction stores its result in the register file and may therefore validate the register in the producer table.
- Fourth, destination registers of instructions are marked invalid during issue and a new tag is stored for them. This operation has priority over the third operation (cf. section 2.1.2).

Our processor features three producer tables, one for each register type, which are implemented with register-based RAM to allow write concurrency. The different accessing contexts above suggest—similar to the ROB—a division of each producer table into two parts: one storing the valid information and the other storing the tag information. We follow this approach; however, it will turn out that the producer tables do not lie on a critical path in the processor.

3.15.1 General Purpose Register Producer Table

The general purpose register producer table GPRPT producer table consists of a 32×1 valid RAM and of a $32 \times \nu$ tag RAM; ν is the tag width, cf. section 3.13.

Reading the Source Operands

The tag and the valid part provide two read ports per instruction for the operands i_i.op$_1$ and i_i.op$_2$. These ports are addressed by i_i.op$_1$.a$_\star$ and i_i.op$_2$ respectively. Ac-

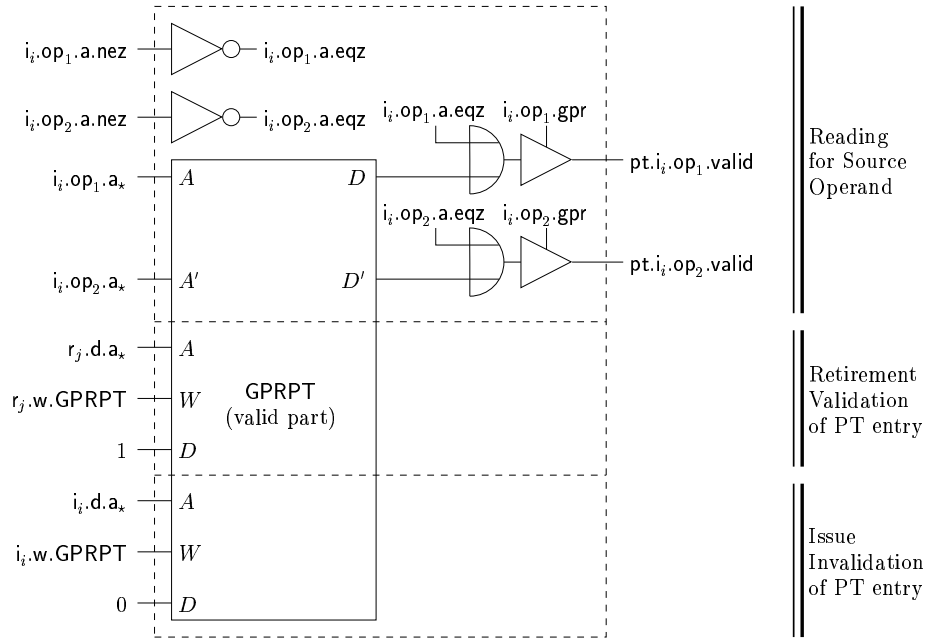

Figure 3.22: Valid information of the GPR producer table

cesses to the GPR R_0 always return a valig flag. We check

$$\mathsf{i_i.op_j.a.eqz} \quad := \quad \overline{\mathsf{i_i.op_j.a.nez}}$$

and force the output of the valid bit to 1 with an OR gate. The data is only driven on the operand bus, if the instructions are integer operands as indicated by $\mathsf{i_i.op_1.gpr}$ and $\mathsf{i_i.op_2.gpr}$. Figures 3.21 and 3.22 include this arrangement for the instruction $\mathsf{i_i}$.

Update During Retirement

During retirement, the producer tables tags are read out to decide, if the entry must be validated.

First, the tag of register $\mathsf{r_j.d.a_\star}$ as stored in the tag part is read out and returned on the bus $\mathsf{r_j.GPR.tag_\star}$. Define:

$$\mathsf{r_j.w.GPRPT} \quad := \quad \mathsf{r_j.wb} \wedge \mathsf{r_j.d.gpr} \wedge (\mathsf{ROB.head_{j,\star}} = \mathsf{r_j.GPR.tag_\star})$$

This signal equals 1 iff $\mathsf{r_j}$ must validate GPR register entry $\mathsf{r_j.d.a_\star}$ (cf. algorithm 5, page 18, ll. 9–11).

These equations suggest that the full time of a read and a write access to the producer table is needed. However, a closer look the implementation of the register-based RAM reveals that the RAM address decoders for both accesses can work in parallel. This allows for a large reduction in path length.

Figures 3.21 and 3.22 include this arrangement for the retirement bus $\mathsf{r_j}$.

Update during Issue

If instruction $\mathsf{i_i}$ issues, it invalidates its destination register in the producer table and sets the tag information accordingly. The signal

$$\mathsf{i_i.w.GPRPT} \quad := \quad \mathsf{i_i.issue} \wedge \mathsf{i_i.d.gpr}$$

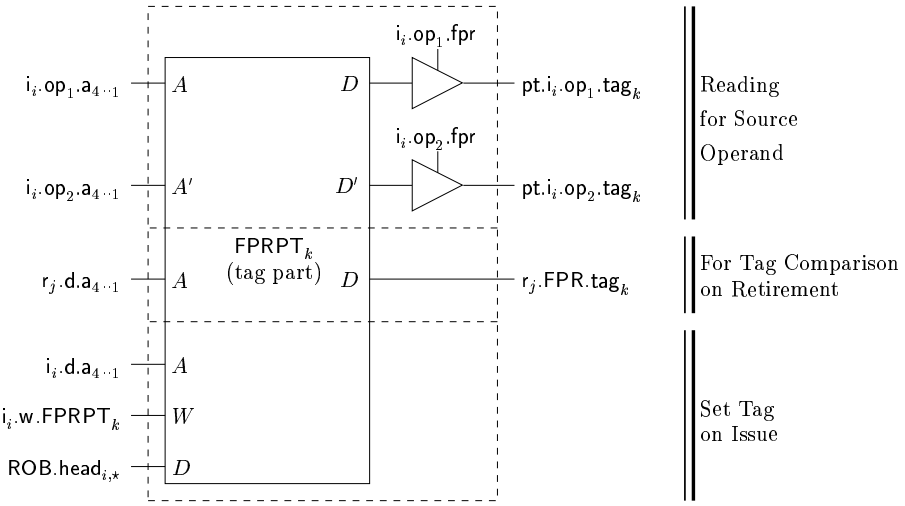

Figure 3.23: Tag information of the FPR producer table

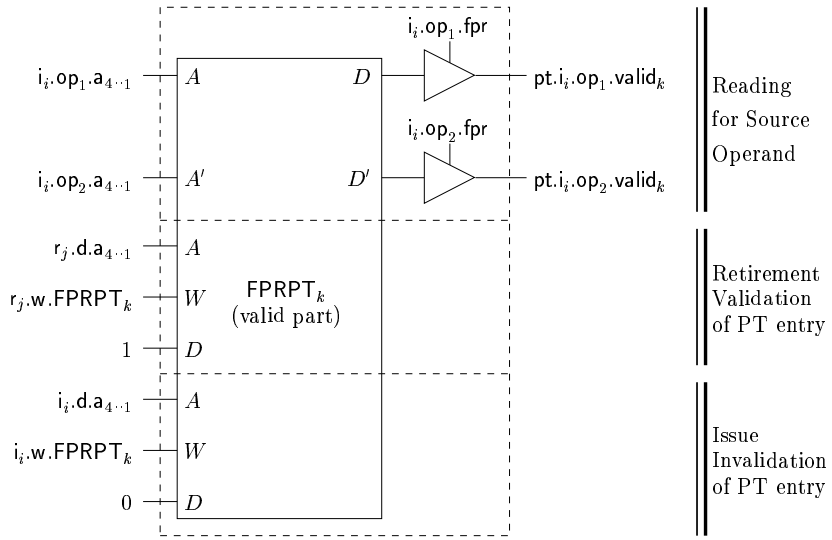

Figure 3.24: Valid information of the FPR producer table

indicates this condition for the GPR producer table. If i_i.w.GPRPT equals one the following two actions are taken:

- The valid part of the GPR producer table stores a 0 at i_i.d.a$_\star$.
- The tag part of the GPR producer table stores the instruction's tag, ROB.tail$_{i,\star}$, at the address i_i.d.a$_\star$.

Figures 3.21 and 3.22 include this arrangement for the instruction i_i.

3.15.2 Floating Point Register Producer Table

The floating point register producer table FPRPT producer table consist of two 16×1 valid RAMs, and of two $16 \times \nu$ tag RAMs. This split is in accordance with the structure of the floating point register file.

Reading the Source Operands

Reading out the tag and valid information for an instruction operand $i_i.op_1$ and $i_i.op_2$ at the operand's addresses $i_i.op_1.a_{4..1}$ and $i_i.op_2.a_{4..1}$ returns two valid flags and two tags for each operand. This data are driven on the operand busses according to the signals $i_i.op_1.fpr$ and $i_i.op_2.fpr$.

Figures 3.23 and 3.24 include this arrangement for the instruction i_i.

Update During Retirement

During retirement, the producer tables tag are read out to determine, if the entry must be validated.

First, we read out the low and high tags of the registers $r_j.d.a_{4..1}$ and return them on the busses $r_j.FPR_{lo}.tag_\star$ and $r_j.FPR_{hi}.tag_\star$. Then define:

$$\begin{aligned} r_j.w.FPRPT_k \quad := \quad & r_j.wb \wedge r_j.d.fpr \wedge (r_j.d.db \vee (r_j.d.a_0 = k)) \\ & \wedge (ROB.head_{j,\star} = r_j.FPR_k.tag_\star) \end{aligned}$$

These two signals indicate, whether the floating point register entry must be validated.

Again, no two real RAM lookups are required for the same reason as stated in 3.15.1, p. 79.

Figures 3.23 and 3.24 include this arrangement for the retirement bus r_j.

Update during Issue

If instruction i_i issues, it invalidates its destination register in the producer table and sets the tag information accordingly. The signals

$$i_i.w.FPRPT_k \quad := \quad i_i.issue \wedge i_i.d.fpr \wedge (i_i.d.db \vee (r_i.d.a_0 = k))$$

indicates this condition for the low and the high part of the FPR producer tables. If $i_i.w.PTGPR_{lo}$ (resp. $i_i.w.PTGPR_{hi}$) equals one the following two actions are taken:

- The valid part of the low FPR producer table (resp. the high FPR producer table) stores a 0 at $i_i.d.a_{4..0}$.
- The tag part of the low FPR producer table (resp. the high FPR producer table) stores the instruction's tag, $ROB.tail_{i,\star}$, at the address $i_i.d.a_{4..0}$.

Figures 3.23 and 3.24 include this arrangement for the instruction i_i.

3.15.3 Special Purpose Register Producer Table

The SPR producer table consists of a 32×1 valid RAM and of a $32 \times \nu$ tag RAM.

Reading the Source Operands

The tag and the valid part provide two read ports per instruction for the operands $i_i.op_1$ and $i_i.op_2$. These ports are addressed by $i_i.op_1.a_\star$ and $i_i.op_2$ respectively.

Figures 3.25 and 3.26 include this arrangement for the instruction i_i.

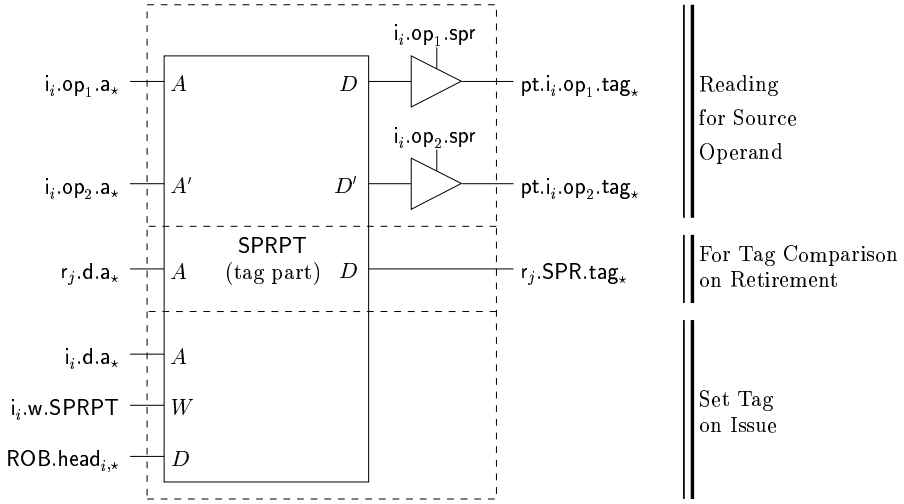

Figure 3.25: Tag information of the SPR producer table

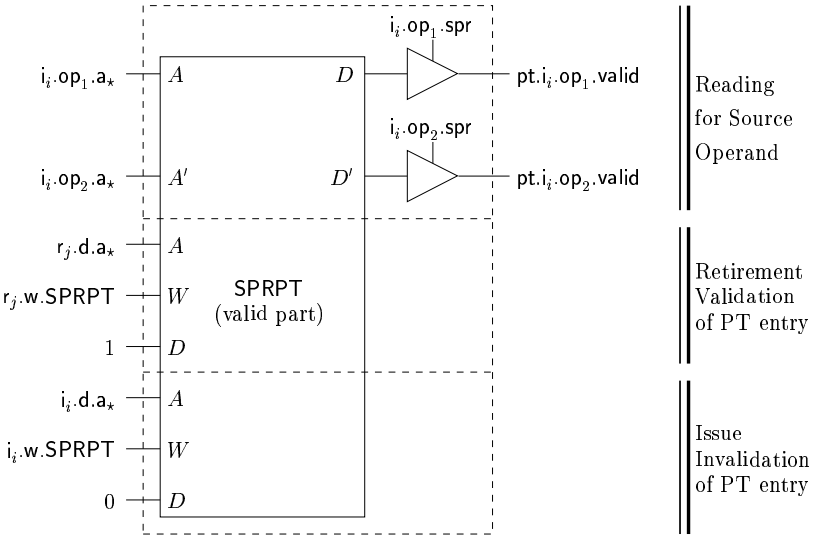

Figure 3.26: Valid information of the SPR producer table

Update During Retirement

During retirement, the producer tables tag are read out to determine, if the entry must be validated.

First, the tag of register $r_j.d.a_\star$ as stored in the tag part are read out and returned on the bus $r_j.SPR.tag_\star$. Define:

$$r_j.w.SPRPT \quad := \quad r_j.wb \wedge r_j.d.spr \wedge (ROB.head_{j,\star} = r_j.SPR.tag_\star)$$

This signal equals 1 iff r_j must validate the SPRF entry stored at $r_j.d.a_\star$.

Figures 3.23 and 3.24 include this arrangement for the retirement bus r_j.

Update during Issue

If instruction i_i issues, it invalidates its destination register in the producer table and sets the tag information accordingly. The signal

$$i_i.w.SPRPT \quad := \quad i_i.issue \wedge i_i.d.spr$$

indicates this condition for the SPR producer table. If $i_i.w.PTSPR$ equals one the following two actions are taken:

- The valid part of the SPR producer table stores a 0 at $i_i.d.a_\star$.
- The tag part of the SPR producer table stores the instruction's tag, $ROB.tail_{i,\star}$, at the address $i_i.d.a_\star$.

Figures 3.25 and 3.26 include this arrangement for the instruction i_i.

Chapter 4

Evaluation

4.1 Hardware Model

The model used herein to determine hardware cost and delay has been introduced in [MP95, KP95]. In this model, basic gates have a fixed cost and delay. The cost of a circuit is defined as the accumulated cost of its gates. The cycle time of a circuit is defined as the longest delay on a path between two registers. Table 4.1 lists the normalized cost and delay for the *MOTOROLA* technology as taken from [MP95].

Note that this model does not take into account fanout restrictions and wire delay. As the Tomasulo algorithm especially in its superscalar variant employs several large bus structures, this topic leaves room for further research.

The cost and the delay of our processor is computed by C++ programs available via WWW[1]. They use a circuit library from [MP95] extended by the additional circuits of this thesis. Cost is simply computed by adding up the individual circuits' costs that compose the processor. Delay is computed by modelling the paths of the processor with a C++ class supplied by [Krö99]. This class allows constructing paths by taking the maximum of several paths and by adding delay to a single path. The paths have symbolical names. Two additional program, both written in `perl`, were used to simplify the task of constructing the paths. The first program extracts the equations for signal definitions directly from the text of this thesis. This procedure simplifies the error-prone task to maintain the coherence between the delay program and the description of the machine. The second program allows for the modular definition of paths and the instantiation of modules. It takes an arbitrarily ordered input of path equations and sorts these to resolve forward references.

4.2 Parameter Space

The design presented in this thesis leaves a big parameter space to explore. Whenever possible explicit constants have been avoided in the design and have been replaced by a parameter.

The parameters of the processor fall into two classes:

- The *scale parameters* are concerned with the widths of major data paths in the design. They put a theoretical limit on the processor's performance, i.e. the cycles per instruction (CPI) rate.

  The parameters we consider belonging to this class, are the number of fetched instructions F, the number of issued instructions I, the number of issues $I(RS)$ on a reservation station RS, the number of dispatches $D(RS)$ of a reservation

[1] `http://www-wjp.cs.uni-sb.de/~mah/thesis/costdelay.tgz`

station RS to its functional unit(s), the number of completions C (i.e. the number of CDBs) and the number of retirements R. As a trivial lower bound on the CPI rate we have:

$$CPI \geq \frac{1}{\min\{F, I, C, R\}}$$

- The *size parameters* are concerned with the sizes of the major data structure in our design. Size parameters influence the CPI rate only implicitly: the processor is stalled, if the resources are low. The thought is tempting that by choosing size parameters large enough, stalling can be overcome. However, as we also see below, size parameters have an impact on the processor's cycle time.

  The size parameters of our processor that we are concerned with are the size of its instruction fetch queues IFQSIZE, the size of the reservation station queues RSQSIZE(RS) for reservation station RS and the size of the reorder buffer ROBSIZE. Parameters we leave out of consideration are the BPU table size and the instruction cache size; these parameters do not influence the processor itself.

Because of the lack of simulations, we cannot study the parameter space adequately. We provide a short glimpse on the effects of the scale and the size parameters. For this consider an overall scale parameters sc and a size parameter sz defining the following point in the parameter space:

$$\begin{aligned} F = I = I(ALU) = D(ALU) &= sz \\ I(RS) = D(RS) &= 1 \\ C = R &= sz \\ \\ IFQSIZE &= 8 \cdot sc \\ RSQSIZE(RS) &= 4 \cdot sc \\ RSQSIZE(ALU) &= 4 \cdot sz \cdot sc \\ ROBSIZE &= 2^{4+2(sz-1)} \end{aligned}$$

The choice of parameters reflects the fact that for $sz = sc = 1$, the processor is similar to [Krö99]. Duplication of the ALU with increasing scale sc is suggested by [Joh91] since 40% to 50% of the dynamic instructions of a MIPS instruction set architecture are ALU instructions.

Keeping F, I, C and R to the same value seems to be the natural solution, although it is unclear that this is an optimal choice. This question may be of interest in simulations of the processor.

Tables 4.2 and 4.3 show nine choices for sz and sc. Cost and cycle time grow monotonically with the rows and the columns. These tables must be read with care, since they only concern the machine core. Scaling of the machine has also an impact on components that lie outside the processor core, for example the instruction cache. These must also be adapted, when more power is wished for.

4.3 A 2-Superscalar Processor

We now restrict our choice of parameters further:

- Set $F = 2$, because the underlying instruction cache (cf. [MP00]) has a 64-bit bus to the processor. We assume the modifications necessary to support our instruction fetch protocol are of negligible cost and have no performance impact.

Gate	Cost	Delay
Inverter	$C_{inv} = 1$	$D_{inv} = 1$
NAND	$C_{nand} = 2$	$D_{nand} = 1$
NOR	$C_{nor} = 2$	$D_{nor} = 1$
AND	$C_{and} = 2$	$D_{and} = 2$
OR	$C_{or} = 2$	$D_{or} = 2$
XOR	$C_{xor} = 4$	$D_{xor} = 2$
XNOR	$C_{xnor} = 4$	$D_{xnor} = 2$
Multiplexer	$C_{mux} = 3$	$D_{mux} = 2$
Tristate Driver	$C_{driv} = 5$	$D_{driv} = 2$
Flip-Flop	$C_{ff} = 8$	$D_{ff} = 4$

Table 4.1: Cost and delay of the basic gates

Delay	$sz = 1.0$		$sz = 1.5$		$sz = 2.0$	
$sc = 1$	107	100.00%	113	105.61%	139	129.91%
$sc = 2$	122	114.02%	143	133.64%	185	172.90%
$sc = 4$	164	153.27%	201	187.85%	275	257.01%

Table 4.2: Effect on Delay due to Scaling and Sizing

Cost	$sz = 1.0$		$sz = 1.5$		$sz = 2.0$	
$sc = 1$	317279	100.00%	424346	133.75%	569332	179.44%
$sc = 2$	410673	129.44%	551572	173.84%	738716	232.83%
$sc = 4$	637495	200.93%	863926	272.29%	1153707	363.63%

Table 4.3: Effects on Cost due to Scaling and Sizing

- Consequently, we set $I = C = R = 2$, since there is no hint for a better allocation.

- The ALU is duplicated. Its reservation station are configured for indegree 2 and outdegree 2. So, two instructions may be issued on RS(ALU) and two instructions may also be dispatched from RS(ALU). Studies in [Joh91] suggest that providing an additional ALU in a register-renaming-implementation is worthwhile.

- We have 6 ALU reservations stations and 4 reservation stations for each other functional unit. For a non-superscalar processor 4 and 2 were a good choice according to [Krö99].

- The reorder buffer size is set to 32 to accommodate the additional instructions executed in the superscalar design. Again, 16 seemed to be a wise choice for a non-superscalar processor.

- The IFQ size is set to 8.

4.3.1 Cost and Delay Optimization

Table 4.4 shows a detailed overview of the processor cost.

The execute and the decode / issue stage make up for about 70% of the whole machine cost. In the execute stage, the floating point units are particularly expensive. In the decode / issue stage the reservation station queues contribute most of the cost; note, that there also reservation stations in the DMemFu. So, all the reservation stations cost about half of the machine.

As has been indicated in section 3.13, we consider various variants of the ROB design. First, the ROB may be considered as "one RAM". This RAM requires $4 * I + 2 + R$ read and $I + C$ write ports. Second, the ROB may be broken down into functional groups, and providing only the ports needed for each group. An overview of the groups is shown in table 3.11. "Trimming" the ROB results in cost savings *and* delay reduction and is therefore the better option in all scenarios.

As we will see, the ROB lies on the critical path of the design. Further delay optimization are therefore advisable. One approach is to replace the regular RAM by register-based RAM. Having the "one RAM" ROB implementation, this results in a fabulous cost increase. With a "trimmed ROB", however, one is able to optimize the time-critical parts of the ROB only.

Table 4.5 shows a number options. Trimmed ROB is cheapest and has only 50% cost of a regular ROB implementation. Furthermore, the trimmed ROB implementation has only delay 160 compare to gate delay 248 of a regular ROB implementation. Four optimization options are shown for the trimmed ROB. The first one implements the valid group with register-based RAM, the second implements the valid group and the onIssue group with register-based RAM. As these components lie on the critical path, delay decreases from 160 to 143 to 135 with relatively little cost increase. With gate delay 135, the data group lies on the critical path. The data group, divided in low and high data, has a width of 64 bits, and therefore takes up a great portion of the ROB storage. Implementing it with register-based RAM is therefore quite expensive, although delay also reduces considerably.

For comparison to other architectures, we will use the "Trimmed (opt: valid, onIssue)" design with delay 135. Since there the floating point unit has a path of gate delay 137 ([Lei99]), further optimization are not necessary without optimizing the floating point units as well.

Component	Cost		Cf.
	Gates	%	
IF stage (2 fetches, 2 issues)	**21349**	**5.5**	
IF Control	385	0.1	3.3.3
IM environment	136	0.0	3.4
IFQ (2 of size 8)	3822	1.0	3.5
PC environment	1563	0.4	3.6
Prediction environment	2682	0.7	3.7
BPU (2-bit sat. cntr, 2048 entries)	8582	2.2	app. B
Instruction Window	357	0.1	3.8
DI stage (2 issues)	**111513**	**28.8**	
decode / issue	7678	2.0	3.9
instruction control	2504	0.6	
instruction data	2118	0.5	
global control	410	0.1	
global data	1528	0.4	
reservation stations (snooping from 2 CDBs)	103835	26.9	3.10
EX stage	**135658**	**35.1**	
Float Adder	23735	6.1	[Lei99]
Float Mul/Div	47557	12.3	[Lei99]
Float Conv.	15926	4.1	[Lei99]
Float Tranfer	2209	0.6	[Lei99]
Integer ALU (2)	7386	1.9	[MP95]
BCU	461	0.1	3.11.4
DMemFU (including 4 reservation stations)	25664	6.6	[Krö99]
Producers (to 2 CDBs)	12720	3.3	3.11.5
Completion stage (2 compl., 2 ret.)	**62940**	**16.3**	
CDB Control	794	0.2	3.12
ROB (32 entries)	61448	15.9	3.13
Retirement Computation / Rollback Checking	698	0.2	3.13.5
WB stage (2 retirements)	**55146**	**14.3**	
Register Files	24452	6.3	3.14
Producer Tables	30694	7.9	3.15
Total	**386606**	**100.0**	

Table 4.4: Overview of the 2-Superscalar Machine Cost

Type	ROB Cost		Total Cost		Delay
Trimmed	33956	100.00%	359114	100.00%	160
Trimmed (opt: valid)	38132	112.30%	363290	101.16%	143
Trimmed (opt: valid,onIssue)	61448	180.96%	386606	107.66%	135
Trimmed (opt: valid,onIssue,data)	139472	410.74%	464630	129.38%	112
Trimmed (all reg-based)	160716	473.31%	485874	135.30%	112
One RAM	70202	206.74%	395360	110.09%	248
One RAM (reg-based)	428234	1261.14%	753392	209.79%	112

Table 4.5: Comparison of ROB designs

	Pipelined		Tomasulo		Superscalar	
CPU core only	108949	100%	235989	216%	386606	354%
with 16 KB cache	483928	100%	610968	126%	761585	157%
CPI/speedup	2.12	0%	1.47	44%	*1.41*	*50%*

Table 4.6: Cost of Complete CPU and CPI rates

```
  0 p_issueIFQ_idx
  2 issueIFQ[*].*
  0 i_opc
 13 i_CSIG
  4 i_op_a
  9 pt_i_op_tag_GPR
  2 pt_i_op_tag
 43 rob_i_op_data
  2 i_op_4ROB
  2 i_op_3CDB
  2 i_op_2RF
  2 i_op_1DI
  2 cfi_op_validAux
  2 cfi_op_valid
  4 cfi.resolve
  4 cfi.ready
  4 i[*].cfi
  6 i[i].invalid
  4 i[i].stallAux
  2 i[*].stall
  1 i[i].issue
  4 RSQ[j].W.req[k]
  0 floatRSQ_r_req
  2 floatRSQ_m
  0 floatRSQ_r_ack
  6 floatRSQ_validAfterRead
  1 floatRSQ_w_ack
  5 floatRSQ_validAfterWrite
  2 floatRSQ_validP
  0 floatRSQ_data_ce
  5 p_floatRSQ_data register in
135 TOTAL (31 circuits)
```

Trimmed

```
  0 p_issueIFQ_idx
  2 issueIFQ[*].*
  0 i_opc
 13 i_CSIG
  4 i_op_a
  9 pt_i_op_tag_GPR
  2 pt_i_op_tag
 44 rob_i_op_valid
  2 i_op_4ROB
  2 i_op_3CDB
  2 i_op_2RF
  2 i_op_1DI
  2 cfi_op_validAux
  2 cfi_op_valid
  4 cfi.resolve
  4 cfi.ready
  4 i[*].cfi
  6 i[i].invalid
  4 i[i].stallAux
  2 i[*].stall
  1 i[i].issue
 44 rob_issue_VALIDp
  5 p_rob_issue_VALID register in
160 TOTAL (23 circuits)
```

Trimmed (opt:valid,onIssue)

Table 4.7: Path of maximum delay

4.3.2 Longest Path

Table 4.7 shows a path of maximum delay for two variants of the processor. The processor producing the path on the left-hand side is implemented with a trimmed ROB RAM. The processor producing the path on the right-hand side is the "Trimmed (opt:valid,onIssue)" we have chosen for comparison with other architectures.

The slower, right-hand path starts in the instruction window and continues with instruction decoding. The ROB valid RAM is requested during source operand generation to decide, if the source operand is already stored in the RAM. The constructed source operand is passed on to the fetch mechansism (cfi.$\star$) for resolving branches. Only after the resolving of branches, the instruction fetch mechanism may annotate the instruction window with the bbi and cfi tags and the decode / issue environment may generate the stall signals i_i.stall.

The path resembles the critical paths for control flow resolving of classical pipelined designs. It can be avoided with a branch predictor unit predicting *all* types of CFIs, so that CFI resolving will not be necessary.

The faster, left-hand path has the slowest component replaced. However, it takes the same route. The paths ends in the reservation station queue control, which can only update its state, if read and write requests have been computed.

4.3.3 Quality Comparison

Quality Measure

This section presents a quality comparison of three DLX designs. Quality is defined, according to [Grü94, MP95], as the q–weighted geometric mean of the performance of an architecture and of the reciprocal of its cost. The weight $q \in [0, 1]$ can be used to put emphasis either on cost or on performance: Only performance counts with $q = 0$, only cost counts with $q = 1$. For $q = 0.5$ cost and performance are equally weighted. The choice of q between 0.2 and 0.5 seems to be realistic.

Justified below, we measure the performance of an architecture by the reciprocal of its CPI rate. Cost is measured by gate counting. Therefore, the quality of a design can be written as follows:

$$Q_q \quad = \quad \frac{1}{CPI^{1-q}C^q}$$

With q fixed, two designs A and B can be compared by their quality Q_q^A and Q_q^B. Better design with respect to q have a greater quality.

Comparison

We compare our design with two others: a pipelined DLX design with precise interrupts and floating point units from [MP95, MP00] and a non-superscalar Tomasulo design presented in [Krö99].

All designs use similar floating point units and therefore share the same longest path with 137 gate delays. Performance therefore depends only on the CPI ratio. The CPI values for the pipelined DLX and the non-superscalar Tomasulo design are taken from [Ger98, Del98]. Due to lack of simulations we have to estimate the CPI rate of the 2-superscalar processor. [Joh91] suggests that a superscalar out-of-order design using 2 ALUs and a *scalar* fetcher has a 1.5-fold speed-up over a non-superscalar processor. We apply this (conservative) estimate to the CPI rate of the pipelined design and obtain a CPI ratio of $2.12/1.5 = 1.41$.

Figure 4.1 plots the quality functions of these designs in a logarithmic scale. The two vertical lines indicate the points of equal quality of the non-superscalar to the

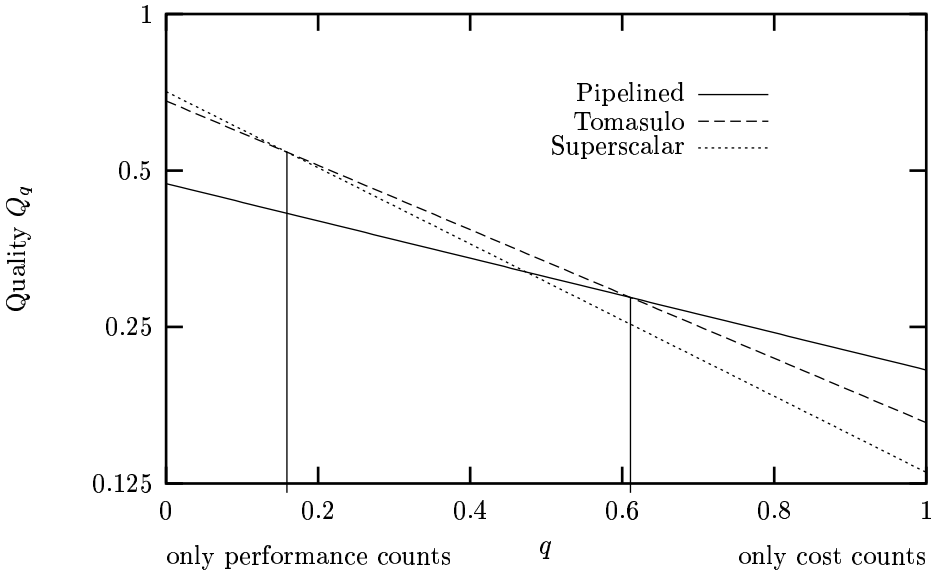

Figure 4.1: Quality for Pipeline DLX, Non-Superscalar and Superscalar Tomasulo

super-scalar and of the non-superscalar to the pipelined processor. With an unrealistic emphasis $q \approx 0.6$ on cost, the pipelined design has the highest quality. For $q \approx 0.18$, the superscalar Tomasulo design wins over the non-superscalar Tomasulo design. This parameter is, however, out of the "realistic" interval. Two reasons can be given for the bad quality: First, our generic approach to superscalar processors most likely produces overhead in cost and delay for instantiation of the machine. This can be mended by optimizing the machine for fixed parameters. Second, simulations miss to replace the CPI ratio 1.41 by a more realistic and hopefully smaller value.

Chapter 5

Circuits

This chapter introduces four circuits forming integral parts of our design.

The multiported round-robin selector presented first, is used to generate bus acknowledgement signals for the common data bus control. Second, a multicounter is developed. This component is used for the implementation of multiported queues in section 5.3. Multiported queues are used in the instruction fetch stage to build the IFQs and in the completion stage to build the reorder buffer. Finally, a (multiported) reservation station queue is developed, providing a multiple-issue and multiple-dispatch framework for reservation stations.

5.1 Multiported Round-Robin Selector

5.1.1 Abstract View on Multiported Round-Robin Selectors

Definition 5.1 *Let $n \in \mathbb{N}$ and $N := \{0, \ldots, n-1\}$. The round-robin selection function for width n is defined as follows:*[1]

$$\begin{array}{rccc} \mathrm{rrs}_n: & \mathcal{P}(N) \times N & \longrightarrow & N \\ & (\mathcal{R}, p) & \longmapsto & a \end{array}$$

such that

$$a = \begin{cases} \min \mathcal{R}^{>p} & \text{if } \mathcal{R}^{>p} \neq \emptyset \\ \min \mathcal{R}^{<p} \cup \{p\} & \text{otherwise} \end{cases} .$$

The function $\mathrm{rrs}(\cdot)$ is used to describe round-robin selecting. $\mathcal{R}$ represents the set of requests received from n busses, p is the previously selected bus and the function result a is the bus that receives acknowledgement.

Next, we define a similar function that handles up to k requests in a round.

Definition 5.2 *Let $n \in \mathbb{N}$ and $N := \{0, \ldots, n-1\}$. Let $1 \leq k \leq n$ and $K := \{0, \ldots, k-1\}$. The k–round-robin selection function for width n is defined as follows:* [2]

$$\begin{array}{rccc} k\text{–}\mathrm{rrs}_\mathrm{n}: & \mathcal{P}(N) \times N & \longrightarrow & K \times N^{\leq k} \\ & (\mathcal{R}, p) & \longmapsto & (numAck, a_1, \ldots, a_{numAck}) \end{array}$$

[1] For a set M, the power set is denoted by $\mathcal{P}(M)$. For a relation R on M and $a \in M$ (respectively $b \in M$) define

$$\begin{array}{rcl} M^{a\ R} & := & \{b \in M \mid a\ R\ b\} \\ M^{R\ b} & := & \{a \in M \mid a\ R\ b\} \end{array}$$

[2] For a set M define $M^{\leq l}$ as the set of tuples of M with maximum length l. This overloads the notation $M^{R\ b}$ defined above; the context, however, always makes the meaning clear.

such that

$$\begin{aligned} numAck &= \min\{k, \#\mathcal{R}\} \\ a_1 &= \mathrm{rrs_n}(\mathcal{R}, p) \\ a_i &= \mathrm{rrs_n}(\mathcal{R} \setminus \{a_1, \ldots, a_{i-1}\}, a_{i-1}) \quad \textit{for } i > 1 \end{aligned}$$

The following lemma justifies the implementation of the k-from-n round-robin selector as given below:

Lemma 5.3 *Let* $k{-}\mathrm{rrs_n}(\mathcal{R}, p) = (numAck, a_1, \ldots, a_{numAck})$ *and let* $(p_1, \ldots, p_l)$ *be the sorted sequence of elements in* $\mathcal{R}^{>p}$ *and* $(p_{l+1}, \ldots, q_m)$ *be the sorted sequences of elements in* $\mathcal{R}^{\leq p}$*. Then:*

$$(a_1, \ldots, a_{numAck}) \quad \textit{is a prefix of} \quad (p_1, \ldots, p_m)$$

Proof. We make a finite induction of the number of acknowledgements, $numAck$. For the induction basis, it is clear that a_1 equals p_1 by the definition of $\mathrm{rrs_n}(\cdot)$. Assume, that we have showed that $(a_1, \ldots, a_i)$ is prefix of $(p_1, \ldots, p_m)$. Then by definition of $k{-}\mathrm{rrs_n}$ and the induction hypothesis:

$$\begin{aligned} a_{i+1} &= \mathrm{rrs_n}(\mathcal{R} \setminus \{a_1, \ldots, a_i\}, a_i) \\ &= \mathrm{rrs_n}(\{p_{i+1}, \ldots, p_m\}, p_i) \\ &= p_{i+1} \end{aligned}$$

5.1.2 Implementation of a Multiported Round-Robin Selector

This section deduces the implementation of a k-from-n round-robin selector. Such a circuit receives n requests signals $\mathsf{req}_\star \in \{0,1\}^n$; given the encoding of definition 5.2, we have

$$\mathsf{req}_\mathrm{i} = 1 \quad \Longleftrightarrow \quad i \in \mathcal{R}$$

Furthermore, the circuit stores the index of the last acknowledgement in a register $\mathsf{h}_\star \in \{0,1\}^n$ in (negated) half-unary encoding. Again, with the notation of the definition, we have

$$\langle \overline{\mathsf{h}_\star} \rangle_{hu} = p$$

The circuit computes the 2-dimensional signal array $\mathsf{ack}_{\star,\star} \in \{0,1\}^{k \cdot n}$. The k'–th row $\mathsf{ack}_{\mathrm{k}',\star}$ of this array contains a negated half-unary encoding of the k'-th acknowledgement, so

$$\langle \overline{\mathsf{ack}_{\mathrm{k}',\star}} \rangle_{hu} = \begin{cases} a_{k'} & \text{if } k' < numAck \\ 0 & \text{otherwise} \end{cases}$$

The register $\mathsf{h}_\star$ must be set each cycle to the last acknowledgement i.e.

$$\langle \overline{\mathsf{h}'_\star} \rangle_{hu} = \begin{cases} a_{numAck} & \text{if } numAck > 0 \\ \langle \overline{\mathsf{h}_\star} \rangle_{hu} & \text{otherwise} \end{cases}$$

This completes the definition of the behaviour of the circuit. We proceed with the description of an implementation.

As lemma 5.3 showed, the candidates can be found by looking first in the sorted elements of $\mathcal{R}^{>p}$ and then in the sorted elements of $\mathcal{R}^{\leq p}$. The implementation computes these sets by the equations

$$\begin{aligned} \mathsf{rh}_\star &:= \mathsf{req}_\star \wedge \mathsf{h}_\star \\ \mathsf{rl}_\star &:= \mathsf{req}_\star \wedge \overline{\mathsf{h}_\star} \end{aligned}$$

Since $\mathsf{h}_\star = 1^{n-p}0^p$ we have:

$$\begin{aligned} \mathsf{rh}_\mathrm{i} = 1 &\iff i \in \mathcal{R}^{>p} \\ \mathsf{rl}_\mathrm{i} = 1 &\iff i \in \mathcal{R}^{\leq p} \end{aligned}$$

The search for the candidates for acknowledgement can now be reduced on finding the first k ones in the bus $(\mathsf{rh}_\star, \mathsf{rl}_\star)$. We use a FF$k$1hu$_{2n}$ circuit, defined in section C.2. The FFk1hu$_{2n}$ circuit returns a signal array $\mathsf{o}_{\star,\star} \in \{0,1\}^{2n \cdot k}$. By corollary C.3 (p. 125) the columns $\mathsf{o}_{\star,\mathrm{j}}$ of $\mathsf{m}_{\star,\star}$ being different from 0^{2n} deliver a prefix of the indices of the input bits equal to 1. This sequence is given in negated half-unary encoding: if $\langle \overline{\mathsf{o}_{\star,\mathrm{j}}} \rangle_{hu} = a$ the $(j+1)$–th one is located at the input bit a. Due to the construction, positions indicated being greater or equal to n refer to indices in $\mathcal{R}^{>p}$ and have to shifted back. Such a shift is indicated by $\mathsf{o}_{\mathrm{n}-1,\mathrm{k}'} = 0$ since

$$\mathsf{o}_{\mathrm{n}-1,\mathrm{k}'} = 0 \iff \langle \overline{\mathsf{o}_{\star,\mathrm{k}'}} \rangle_{hu} \geq n$$

The acknowledgement signals can therefore be computed the following way:[3]

$$\mathsf{ack}_{\mathrm{k}',\star} := (\mathsf{o}_{\mathrm{n}-1,\mathrm{k}'} \ ? \ \mathsf{o}_{\mathrm{n}-1\cdots 0,\mathrm{k}'} \ : \ \mathsf{o}_{2\mathrm{n}-1\cdots \mathrm{n},\mathrm{k}'})$$

Finally, the register $\mathsf{h}_\star$ must be updated, if a candidate was acknowledged. Again, $\mathsf{o}_{\star,\star}$ provides sufficient information to distinguish the different cases: the signal

$$\mathsf{numAck}_\star := \mathsf{o}_{2\mathrm{n}-1,\star}$$

contains a half-unary encoding of the number of candidates found. We clock register $\mathsf{h}_\star$ in case that at least one candidate was found,

$$\mathsf{hce} := \mathsf{numAck}_0$$

Since the signal $\mathsf{numAck}_\star$ is a half-unary encoding of the number of candidates acknowledges, a pointer to the last acknowledged candidate can be found using an edge detector:

$$\mathsf{lastAck}_\star := \mathrm{markLastOne}(\mathsf{numAck}_\star)$$

The update formula for $\mathsf{h}_\star$ can therefore be written the following way:

$$\mathsf{h}'_\star = \bigvee \mathsf{ack}_{\mathrm{k}',\star} \wedge \mathsf{lastAck}_\mathrm{i}$$

Figure 5.1 shows an overview of the implementation just described. Much of the computation just outlined is hidden in the "update" box. For the initialization, $\mathsf{h}_\star$ is forced to 1^n.

Proof of Correctness. The correctness of the circuit follows from the construction of the circuit with help of lemma 5.3 and corollary C.3 as pointed out above.

[3] By a notational twist we also conver the columns $\mathsf{o}_{\star,\mathrm{k}'}$ to the rows $\mathsf{ack}_{\mathrm{k}',\star}$. The definition of $\mathsf{ack}_{\star,\star}$ by rows is the more natural one.

5.2 Multicounter

A k-MCNT$_n$ is a counter that makes up to k steps in a single cycle. The step width is determined by an input $\mathsf{inc}_\star \in \{0,1\}^k$ containing a half-unary encoding of the number of steps. The counting register $\mathsf{CNT}_\star \in \{0,1\}^n$ satisfies the following equation:

$$\langle \mathsf{CNT'}_\star \rangle_2 \quad \equiv_{2^n} \quad \begin{cases} 0 & \mathsf{init} = 1 \\ \langle \mathsf{CNT}_\star \rangle_2 + k' & \text{if } \langle \mathsf{inc}_\star \rangle_{hu} = k' \end{cases}$$

Additionally, the multicounter returns the counters $\mathsf{cnt}_{\mathrm{j},\star} \in \{0,1\}^n$ for $0 \leq j \leq k$ with the property

$$\langle \mathsf{cnt}_{\mathrm{j},\star} \rangle_2 \quad = \quad \langle \mathsf{CNT}_\star \rangle_2 + j$$

An implementation of such a multicounter is shown in figure 5.2. An multiple incrementer (cf. section C.3) is used to compute the output counters $\mathsf{cnt}_{\mathrm{j},\star}$ for $j \in \{0,\ldots,k\}$. From these counters the correct one has to be selected for the update of $\mathsf{CNT'}_\star$. Selection signals can be obtained by converting $\mathsf{inc}_\star$ into unary encoding using an edge detector:

$$\mathsf{incU} \quad := \quad \mathrm{markLastOne}\,((\mathsf{inc}_\star, 1))$$

The bus $\mathsf{incU}_\star \in \{0,1\}^{k+1}$ satisfies $\mathsf{incU}_\mathrm{j} = 1 \iff j = \langle \mathsf{inc}_\star \rangle_{hu}$. Therefore, $\mathsf{CNT'}_\star$ can be computed with

$$\mathsf{CNT'}_\star \quad = \quad \bigvee_{0 \leq j \leq k-1} \mathsf{incU}_\mathrm{j} \wedge \mathsf{cnt}_{\mathrm{j},\star}$$

5.3 Multiported Queue

This section develops a multiported queue. A queue is a storage structure supporting read and write operations on a first-in-first-out (FIFO) basis. Multiporting allows multiple read / write accesses per cycle.

This section proceeds in two steps. First, an abstract model of a multiported queue is developed on the basis of finite state transducers. Second, an interface for the implementation is described which we use in two different implementations. The correctness of the implementations is verified by showing the equivalence to the abstract definition.

5.3.1 Definition of an Abstract Multiported Queue

Definition 5.4 *A finite state transducer A is an 6-tuple $A = (I, Z, \delta, z_0, O, \epsilon)$; I is the input alphabet, Z is a set of states, $\delta : I \times Z \to Z$ is the state transition function, $z_0 \in Z$ is the initial state, O is the output alphabet, $\epsilon : I \times Z \to O$ is the output function.*

Note that finite state transducers, as needed in this thesis, do not require a set of end states. Finite state transducers are only used to represent computations from one step to another.

Definition 5.5 (Abstract Multiported Queue) *Let $k, l, n \in \mathbb{N}$, $k, l \leq n$, DOM a finite set. Define $K := \{0,\ldots,k-1\}$, $L := \{0,\ldots,l-1\}$, $N := \{0,\ldots,n-1\}$. An*

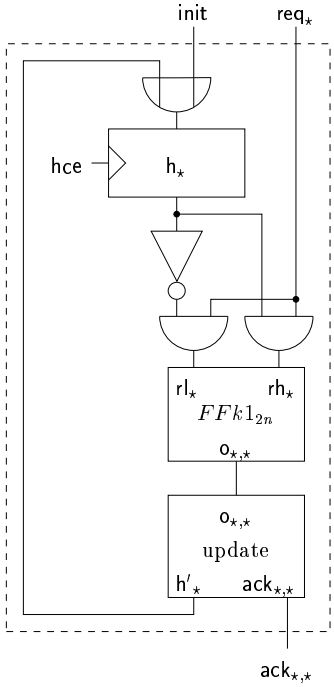

Figure 5.1: Implementation of a multiported round-robin selector

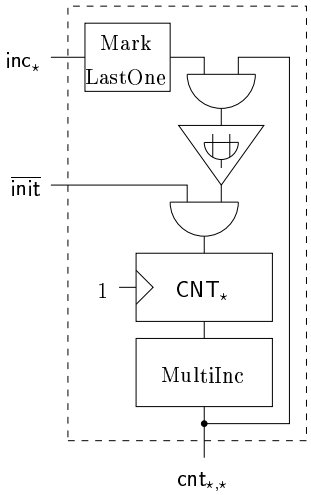

Figure 5.2: Definition of a multicounter

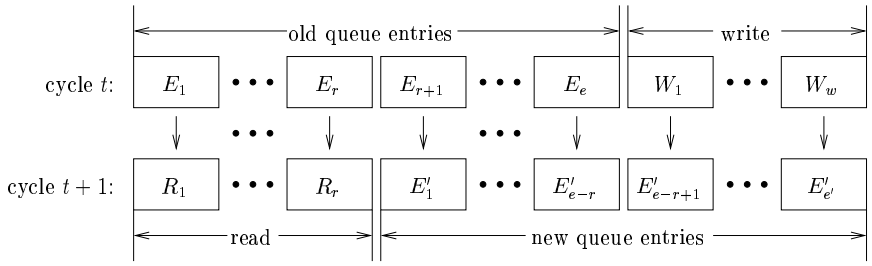

Figure 5.3: Read-write-operation on an abstract MPQ with $wack = w, rack = r$

abstract k:l–multiported queue of size n over the domain DOM is a finite state transducer

$$\begin{aligned} A_{k\,l-MPQ_n} &= (I, Z, \delta, z_0, O, \epsilon) \\ \textit{input alphabet } I &= K \times DOM^{\leq k} \times L \\ \textit{set of states } Z &= N \times DOM^{\leq n} \\ \textit{initial state } z_0 &= 0 \\ \textit{output alphabet } O &= K \times L \times DOM^{\leq l} \end{aligned}$$

The set of states Z models the storage space of the queue. The input alphabet I encodes the arrival of new objects and the output alphabet O encodes the reading-out of objects. The transition function $\delta : I \times Z \longrightarrow Z$ and the output function $\epsilon : I \times Z \longrightarrow O$ are defined as follows:

$$\begin{aligned} \delta\,(w, W_1, \ldots, W_w, r, \quad e, E_1, \ldots, E_e) &= (e', E'_1, \ldots, E'_{e'}) \\ \epsilon\,(w, W_1, \ldots, W_w, r, \quad e, E_1, \ldots, E_e) &= (wack, rack, R_1, \ldots, R_{rack}) \\ rack &:= \min\{r, e\} \\ wack &:= \min\{n - e + rack, w\} \\ e' &:= e + wack - rack \\ (R_1, \ldots, R_{rack}) &:= (E_1, \ldots, E_{rack}) \\ (E'_1, \ldots, E'_{e'}) &:= (E_{rack+1}, \ldots, E_e, W_1, \ldots, W_{wack}) \end{aligned}$$

It is easy to see, that this definition is a formalization of the concept of a multiported queue: the order of the queue entries (according to the insertion time) $E_1, \ldots, E_e$ is preserved. Entries are read head-first and written tail-first. Multiplicity is introduced by the read and write parameters. Figure 5.3 shows a graphical interpretation of these equations for the case that the queue is "nicely" filled (i.e. we have $wack = w, rack = r$). The first row shows the entries $E_1, \ldots, E_e$ being stored in the queue in cycle t. New data $W_1, \ldots, W_w$ arrives also in cycle t and r elements are requested for read-out. The second row shows the read-out entries $R_1, \ldots, R_r$ and the newly formed queue $E'_1, \ldots, E'_{e'}$.

5.3.2 Interface

For the implementation consider the domain $DOM = \{0,1\}^m$ for some $m \in \mathbb{N}$. Table 5.1 shows the definition of the implementation interface of a multiported queue. In addition to the interface for the read / write operations, we consider the information on how the data is stored in the queue also part of the interface. The queue data is stored in *slots* each of which is tagged by a valid bit. The entries $E_1, \ldots, E_e$ are to be found in the valid slots, by convention $\mathsf{S}_1, \ldots, \mathsf{S}_e$. Note the

Name	Type	Meaning
$\mathsf{W}_\star.\mathsf{data}_\star$	In	the write data, busses for $W_1, \ldots, W_k$
$\mathsf{W}_\star.\mathsf{req}$	In	the write requests, half–unary encoding of r
$\mathsf{W}_\star.\mathsf{ack}$	Out	the write acknowledgments, half–unary encoding of $rack$
$\mathsf{S}_\star.\mathsf{valid}$	Out	half–unary encoding of e
$\mathsf{S}_\star.\mathsf{data}_\star$	Out	storage slots for the entries
$\mathsf{R}_\star.\mathsf{data}_\star$	Out	the read data, busses for $R_1, \ldots, R_l$
$\mathsf{R}_\star.\mathsf{req}$	In	the read requests, half–unary encoding of w
$\mathsf{R}_\star.\mathsf{ack}$	Out	the read acknowledgments, half–unary encoding of $wack$

Table 5.1: Multi-ported queue interface

negation and reversion of the $\mathsf{E}_\star.\mathsf{valid}$ bus can also be interpreted as a full bus, $\mathsf{full}_\star := \overline{\mathsf{E}_{0..n-1}.\mathsf{valid}}$. In this case, $\mathsf{full}_\mathsf{i} = 1$ iff i entries cannot be stored in queue (before read-out).

5.3.3 Register-Based Implementation

In the register-based implementation, the slots are directly modelled with flip-flops $\mathsf{S}_\mathsf{i}.\mathsf{data}_\mathsf{j}$ for $0 \leq i \leq n-1$ and $0 \leq j \leq m-1$. A valid register $\mathsf{valid}_\star \in \{0,1\}^n$ indicates which slots hold valid entries. It encodes the fill state of the queue in half-unary encoding, i.e. $\langle \mathsf{valid}_\star \rangle_{hu}$ is the number of allocated queue slots.

Queue Control

This section develops the queue control signals in three different steps. The "Read Phase" describes how the read operations are acknowledged and performed. The "Slot Propagation" describes the adjustment of the queue slots after the read-out. The "Write Phase" describes how the write operations are acknowledged and how the data reaches the appropriate slots.

Read Phase. The reading of slots is simple. For the computation of the read acknowledgements signals we use the fact, that the minimum function used to compute $rack$ (cf. definition 5.5), is easily implemented in half-unary encoding with a slice of AND-gates:

$$\mathsf{R}_\star.\mathsf{ack} \quad := \quad \mathsf{S}_{\mathsf{l}-1..0}.\mathsf{valid} \wedge \mathsf{R}_\star.\mathsf{req}$$

In preparation for the write phase, we assume that the read operations have been separately performed before the write operations. The fill state of the queue after read-out, $\langle \mathsf{validAfterRead}_\star \rangle_{hu}$ can be obtained by a half-unary subtraction operation:

$$\langle \mathsf{validAfterRead}_\star \rangle_{hu} \quad = \quad \langle \mathsf{valid}_\star \rangle_{hu} - \langle \mathsf{R}_\star.\mathsf{ack} \rangle_{hu}$$

Slot Propagation. After reading out, the entries stored in the slots move down the queue to fill the read-out empty slots again. This movement is called slot propagation. Let $\mathsf{prop}_{\mathsf{l}'} = 1$ indicate that the propagation distance equals l' steps ($l' \in \{0, \ldots, l\}$). Since $\mathsf{R}_\star.\mathsf{ack}$ provides a half-unary encoding of the number of read-out entries, we can compute $\mathsf{prop}_\star$ with an edge detector:

$$\mathsf{prop}_\star \quad := \quad \mathrm{markLastOne}\,(\mathsf{R}_\star.\mathsf{ack})$$

Write Phase. In the write phase, we first want to compute the write acknowledgement signals. According to property 3.6 (p. 35), of half-unary numbers, the reversed and negated version of $\mathsf{validAfterRead}_\star$ encodes the number of free entries after read-out:

$$\langle \overline{\mathsf{validAfterRead}_{0\cdots n-1}} \rangle_{hu} \quad = \quad n - e + rack$$

Because $wack = \min\{n - e + rack, w\}$ by definition 5.5 we can compute the write acknowledgement signals by a slice of AND-gates:

$$\begin{aligned} \mathsf{W}_\star.\mathsf{ack} &= \mathsf{W}_\star.\mathsf{req} \wedge \overline{\mathsf{validAfterRead}_{n-k\cdots n-1}} \\ &= \overline{\overline{\mathsf{W}_\star.\mathsf{req}} \wedge \mathsf{validAfterRead}_{n-k\cdots n-1}} \end{aligned}$$

The acknowledged write data must be appended to the valid entries in the queue after read-out. To control the distribution of the write busses, we define the signal array $\mathsf{W2S}_{\star,\star}$. The condition $\mathsf{W2S}_{k',n'} = 1$ indicates that write bus $\mathsf{W}_{k'}$ shall be written to slot $\mathsf{S}_{n'}$.

For $k' = 0$, the bus

$$\mathsf{insPos}_\star \quad := \quad \mathrm{markFirstZero}\,(\mathsf{validAfterRead}_\star)$$

marks the insertion position, i.e. $\mathsf{W2S}_{0,\star} = \mathsf{W}_0.\mathsf{req} \wedge \mathsf{insPos}_\star$. For $k' > 0$ we obtain $\mathsf{W2S}_{k',\star}$ by shifting $\mathsf{insPos}_\star$:

$$\mathsf{W2S}_{k',n'} \quad = \quad \mathsf{W}_{k'}.\mathsf{req} \wedge \mathsf{insPos}_{n'-k'} \quad \text{for} \quad 1 \le n' - k' \le n$$

This ensure that the written entries are appended "en bloc" after the valid queue entries.

Update of the Valid Register. The new valid register $\mathsf{valid}'_\star$ can be computed by a half-unary addition that satisfies:

$$\langle \mathsf{valid}'_\star \rangle_{hu} \quad = \quad \langle \mathsf{validAfterRead}_\star \rangle_{hu} + \langle \mathsf{W}_\star.\mathsf{ack} \rangle_{hu}$$

Figure 5.4 contains an overview of the calculations for the valid register. Additionally it initializes the $\mathsf{valid}_\star$ register if the signal $\overline{\mathsf{init}_\star}$ equals 0.

Structure of the Slots

Figure 5.5 shows the definition of the slot $\mathsf{S}_{n'}$. The register $\mathsf{S}_{n'}.\mathsf{data}_\star$ stores the data. Sources for this register are the k write busses $\mathsf{W}_\star.\mathsf{data}$, the own data $\mathsf{S}_{n'}$ (which may be looped back for update purposes) and the data of the l above slots $\mathsf{S}_{n'+1}, \ldots, \mathsf{S}_{n'+1}$ for slot propagation purposes. Drivers are used to select the data from the different sources; $\mathsf{W2S}_{k',n'} = 1$ indicates, that write bus $\mathsf{W}_{k'}$ shall write into slot $\mathsf{S}_{n'}$. At most one bit of $\mathsf{W2S}_{\star,n'}$ can be active by definition of $\mathsf{W2S}_{\star,\star}$. The signal

$$\mathsf{S2S}_{n'+l',n'} \quad = \quad \mathsf{valid}_{n'+l'} \wedge \mathsf{prop}_{l'}$$

for $l' \in \{0, \ldots, l\}$ is used to select $\mathsf{S}_{n'+l'}$ as a source. Again, at most one of $\mathsf{S2S}_{n'+l\cdots n',n'}$ can be active, since at most one of $\mathsf{prop}_\star$ is active by definition. Furthermore, the two different output enables busses are mutually exclusive:

$$\begin{aligned} \exists\, l' : \mathsf{S2S}_{n'+l',n'} = 1 \quad &\Longleftrightarrow \quad \mathsf{prop}_{l'} = 1 \wedge \mathsf{valid}_{n'+l'} = 1 \\ &\Longleftrightarrow \quad \mathsf{R}_\star.\mathsf{ack} = 0^{l-l'}1^{l'} \wedge \mathsf{valid}_{n'+l'} = 1 \\ &\Longleftrightarrow \quad \mathsf{validAfterRead}_{n'} = 1 \\ &\Longleftrightarrow \quad \mathsf{W2S}_{\star,n'} = 0 \end{aligned}$$

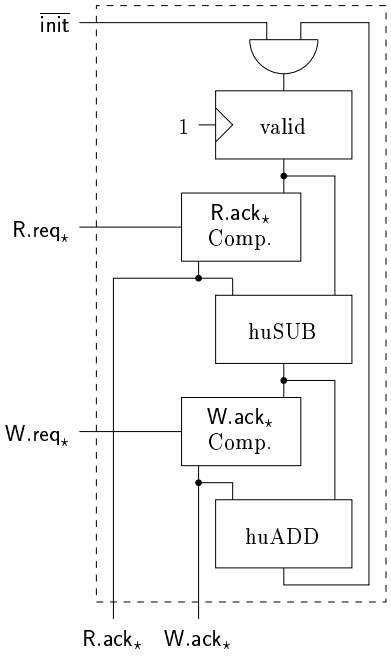

Figure 5.4: Definition of the valid register

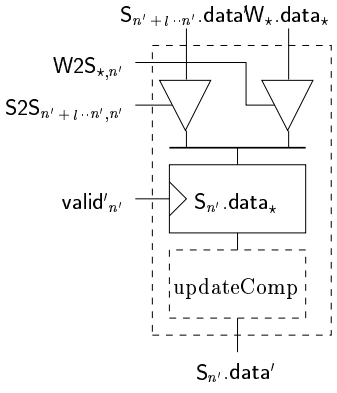

Figure 5.5: Definition of the n'–th slot

The data register is only clocked, if it is valid in the next cycle, i.e. $\mathsf{S}_{\mathrm{n}'}.\mathsf{data.ce}' = \mathsf{valid}'_{\mathrm{n}'}$.

Note that the construction of the slots allows the integration of an update circuit for the queue data, as shown in figure 5.5. Such a circuit can be used to modify entries after they have been written. However, for simplifying notation, we just consider the update circuit computing the identity function for the remaining part of this section.

Correctness

Our implementation behaves the same way as an abstract queue A_{kl-MPQ_n} on domain $DOM = \{0,1\}^m$. In an inductive proof, we show that the initialization configuration and the state transitions are equivalent subject to the interface defined in section 5.3.2.

For initialization we have $e = 0$ and no entries in the abstract machine. The implementation machine is in an equivalent state since

$$\begin{aligned} \langle \mathsf{valid}_\star \rangle_{hu} &= \langle 0^n \rangle_{hu} \\ &= 0 \\ &= e \end{aligned}$$

and the slots must not match any entries.

Now we show that both machines make equivalent transitions. Under the assumptions

$$\begin{aligned} \mathsf{W}_\star.\mathsf{req} &= 0^{k-w}1^w \\ \mathsf{W}_{\mathrm{w}-1\cdot\cdot 0}.\mathsf{data} &= (W_w, \ldots, W_1) \\ \mathsf{R}_\star.\mathsf{req} &= 0^{l-r}1^r \end{aligned} \tag{5.1}$$

the following equations hold:

$$\langle \mathsf{R}_\star.\mathsf{ack} \rangle_{hu} = rack \tag{5.2}$$

$$\mathsf{R}_{\mathrm{rack}-1\cdot\cdot 0}.\mathsf{data} = (E_{rack}, \ldots, E_1) \tag{5.3}$$

$$\langle \mathsf{W}_\star.\mathsf{ack} \rangle_{hu} = wack \tag{5.4}$$

$$\langle \mathsf{valid}'_\star \rangle_{hu} = e' \tag{5.5}$$

$$\mathsf{S}_{\mathrm{e}'-1\cdot\cdot 0}.\mathsf{data} = (E'_{e'}, \ldots, E'_1) \tag{5.6}$$

So, assume 5.1. First, we consider the read phase, for which we can directly derive by properties of half-unary encodings:

$$\begin{aligned} \langle \mathsf{R}_\star.\mathsf{ack} \rangle_{hu} &= \langle \mathsf{R}_\star.\mathsf{req} \wedge \mathsf{S}_{\mathrm{l}-1\cdot\cdot 0} \rangle_{hu} \\ &= \min\{r, e\} \\ &= rack \\ \mathsf{R}_{\mathrm{rack}-1\cdot\cdot 0}.\mathsf{data}_\star &= \mathsf{S}_{\mathrm{rack}-1\cdot\cdot 0}.\mathsf{data}_\star \\ &= (E_{rack}, \ldots, E_1) \end{aligned}$$

Next, the equivalence of the write acknowledgement signals are shown. The following equations use properties of half-unary numbers and the correctness of the read acknowledgements already shown:

$$\begin{aligned} \langle \mathsf{W}_\star.\mathsf{ack} \rangle_{hu} &= \langle \mathsf{W}_\star.\mathsf{req} \wedge \overline{\mathsf{validAfterRead}_{\mathrm{n}-\mathrm{k}\cdot\cdot\mathrm{n}-1}} \rangle_{hu} \\ &= \langle (0^{n-k}, \mathsf{W}_\star.\mathsf{req}) \wedge \overline{\mathsf{validAfterRead}_{0\cdot\cdot\mathrm{n}-\mathrm{k}}} \rangle_{hu} \end{aligned}$$

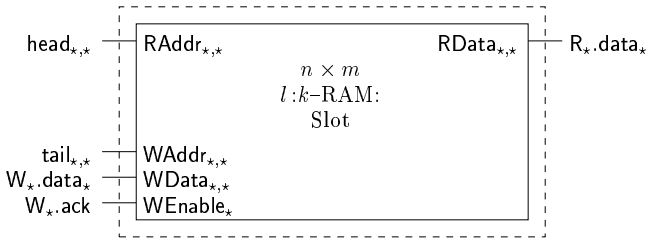

Figure 5.6: Slot storage

$$\begin{aligned} &= \max\{w, n - \langle \mathsf{validAfterRead}_{n-1\cdot\cdot 0}\rangle_{hu}\} \\ &= \max\{w, n - \max\{0, e - rack\}\} \\ &= \max\{w, n - e + rack\} \\ &= wack \end{aligned}$$

The preservation of the queue slot structure remains to be shown. The updated valid register is computed by a half unary addition, so we have

$$\begin{aligned} \langle \mathsf{valid'}_\star\rangle_{hu} &= \langle \mathsf{validAfterRead}_\star\rangle_{hu} + \langle \mathsf{W}_\star.\mathsf{ack}\rangle_{hu} \\ &= e - rack + wack \\ &= e' \end{aligned}$$

Each propagated slot moves down by a distance of $rack$. The written entries are stored in positions $e - rack + 1$ to $e - rack + wack$. So:

$$\begin{aligned} \mathsf{S}_{e'-1\cdot\cdot 0}.\mathsf{data} &= (W_{wack}, \ldots, W_1, E_e, \ldots, E_1) \\ &= (E'_{e'}, \ldots, E'_1) \end{aligned}$$

5.3.4 RAM-Based Implementation

The RAM-based implementation uses a regular multiported RAM accessed by head and tail pointers in a wrap-around fashion. This implementation is cheaper although slower than a register-based implementation. The implementation restricts for n being a power of 2. Sections on the structure of a RAM-based implementation follow.

Slot Storage

The slot storage is a multiported RAM with l write and k read ports. The read ports are addressed by the head pointers $\mathsf{head}_{\star,\star}$ returning their data to the read busses $\mathsf{R}_\star.\mathsf{data}_\star$. The write ports are addressed by the tail pointers $\mathsf{tail}_{\star,\star}$ and enabled by the write acknowledgement signals $\mathsf{W}_\star.\mathsf{ack}$. The data is on the incoming write busses $\mathsf{W}_\star.\mathsf{data}_\star$. Figure 5.6 shows the slot storage environment.

Head and Tail Pointers

The queue is controlled a head and a tail pointer. The (master) head pointer $\mathsf{HEAD}_\star$ always points to the first (oldest) element in the slot storage. The (master) tail pointer $\mathsf{TAIL}_\star$ always points to the first free slot in the slot storage. The queue entries consist of the entries stored between head and tail. Addresses wrap around if they get larger than n.

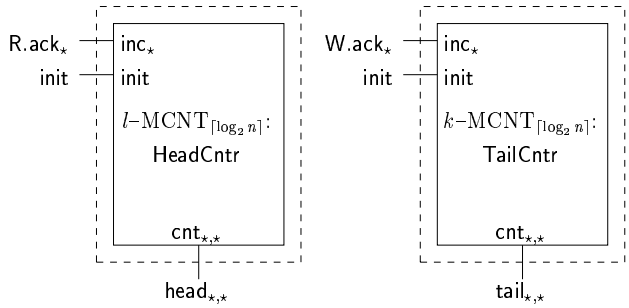

Figure 5.7: Definition of the head and tail pointer

However, one head pointer and one tail pointer is not enough for multiple read and write operations. To support k read operations, we need k head pointers $\mathsf{head}_{\star,\star}$ with

$$\langle \mathsf{head}_{\mathrm{k}',\star} \rangle_2 \quad = \quad \langle \mathsf{HEAD}_\star \rangle_2 + k'$$

The k'–th head pointer can be used to read out the k'–th element. After reading out, the master head is incremented by $\langle \mathsf{R}_\star.\mathsf{ack} \rangle_{hu}$.

The same holds for the tail pointers. To support l write operations, we need l tail pointers $\mathsf{tail}_{\star,\star}$ with

$$\langle \mathsf{tail}_{\mathrm{l}',\star} \rangle_2 \quad = \quad \langle \mathsf{TAIL}_\star \rangle_2 + l'$$

The l'–th tail pointer can be used to write the l'–th element. After writing, the master tail pointer is incremented by $\langle \mathsf{W}_\star.\mathsf{ack} \rangle_{hu}$.

The multicounter, already implemented in section 5.2, provides this functionality for both the head and the tail pointer as shown in figure 5.7.

Valid Register

The valid register is implemented exactly the same way as for the register-based implementation. Refer to section 5.3.3 and figure 5.4 for details.

Correctness

We show correctness the same way as for the register–based implementation. First, we show that the initial configurations are equivalent and then we show inductively that equivalent transitions are performed. Equivalence is defined along with the interface of section 5.3.2. The definition of the slots of the RAM-based implementation is a slightly more complicated; the slots are located between head and tail of the queue. With $e = \langle \mathsf{S}_\star.\mathsf{valid} \rangle_{hu}$ we satisfy

$$E_{e'} \quad = \quad \mathsf{S}_{\mathrm{head+e}'}$$

For the initialization we correctly have $e = 0$ and $\mathsf{S}_\star.\mathsf{valid} = 0$. Then we show under the assumptions:

$$\begin{aligned} \mathsf{W}_\star.\mathsf{req} &= 0^{k-w}1^w \\ \mathsf{W}_{\mathrm{w}-1\cdots 0}.\mathsf{data} &= (W_w, \ldots, W_1) \\ \mathsf{R}_\star.\mathsf{req} &= 0^{l-r}1^r \end{aligned} \tag{5.7}$$

that the following equations hold:

$$\langle \mathsf{R}_\star.\mathsf{ack} \rangle_{hu} = rack \quad (5.8)$$
$$\mathsf{R}_{\mathrm{rack}-1\cdot\cdot 0}.\mathsf{data} = (E_{rack}, \ldots, E_1) \quad (5.9)$$
$$\langle \mathsf{W}_\star.\mathsf{ack} \rangle_{hu} = wack \quad (5.10)$$
$$\langle \mathsf{S}_\star.\mathsf{valid}' \rangle_{hu} = e' \quad (5.11)$$
$$\mathsf{S}_{\mathrm{e}'-1\cdot\cdot 0}.\mathsf{data} = (E'_{e'}, \ldots, E'_1) \quad (5.12)$$

We only show here that the data is correctly read and written to the queue. The other equations have already been proved in 5.3.3.

For read requests $r' \in \{0, \ldots, rack - 1\}$, the r'–th head pointer is used. This pointer points to the r' element starting with and including the master head pointer $\mathsf{HEAD}_\star$. Therefore by induction assumption:

$$\mathsf{R}_{\mathrm{r}'}.\mathsf{data} = E_{r'+1}$$

Data from the write busses is written at the queue tail; $\mathsf{W}_{\mathrm{w}'}$ is written to the position indicated by the w'–th head pointer. With $T := \langle \mathsf{TAIL}_\star \rangle_2$ we have

$$\mathsf{S}_{\mathrm{T}+\mathrm{w}'}.\mathsf{data}' = W_{w'+1}$$

Otherwise, the slot is not modified, so for $n' \notin \{T, \ldots, T + wack\}$ we have

$$\mathsf{S}'_{\mathrm{n}'} = \mathsf{S}_{\mathrm{n}'}$$

Update Ports

The design of the RAM-based MPQ allows for the integration of *update ports*: these ports allow reading and writing data in the middle of the queue. To access an entry, its position is used as an address.

The update ports can be implemented by adding the required read and write ports to the slot storage RAM.

5.4 Reservation Station Queue

A reservation station queue is a specialized multiported queue maintaining an additional status bit e4ro ("eligible for read-out") for each entry. If the bit is 1, an entry may be read; if it is 0 an entry cannot be read. This sort of behaviour is needed in collection of reservation stations: each reservation station becomes eligible for read-out, if it has gathered all its source operands.

We proceed the same way as for multiported queues: first we develop an abstract definition of a reservation station queue based on finite state transducers. A description of an implementation interface follows. Then, a register-based implementation is presented only.

5.4.1 Definition of an Abstract Multiported Queue

Definition 5.6 (Abstract Reservation Station Queue) *Let $k, l, n \in \mathbb{N}$, $k, l \leq n$, DOM a finite set. Define $K := \{0, \ldots, n\}$, $L := \{0, \ldots, l\}$, $N := \{0, \ldots, n\}$, An abstract k:l–reservation station queue of size n over the domain DOM is a finite state transducer*

$$A_{k\,l-RSQ_n} = (I, Z, \delta, z_0, O, \epsilon)$$
$$\textit{input alphabet } I = K \times DOM^{\leq k} \times L$$

Name	Type	Meaning
$\mathsf{W}_\star.\mathsf{data}_\star$	In	the write data, busses for $W_1, \ldots, W_k$
$\mathsf{W}_\star.\mathsf{req}$	In	the write requests, half–unary encoding of r
$\mathsf{W}_\star.\mathsf{ack}$	Out	the write acknowledgments, half–unary encoding of $rack$
$\mathsf{S}_\star.\mathsf{valid}$	Out	half–unary encoding of e
$\mathsf{S}_\star.\mathsf{e4ro}$	Out	the eligibility for read–out
$\mathsf{S}_\star.\mathsf{data}_\star$	Out	storage slots for the entries
$\mathsf{R}_\star.\mathsf{data}_\star$	Out	the read data, busses for $R_1, \ldots, R_l$
$\mathsf{R}_\star.\mathsf{req}$	In	the read requests, half–unary encoding of w
$\mathsf{R}_\star.\mathsf{ack}$	Out	the read acknowledgments, half–unary encoding of $wack$

Table 5.2: Reservation station queue interface

$$\begin{aligned} \textit{set of states } Z &= N \times DOM^{\leq n} \\ \textit{transition function } \delta &: I \times Z \longrightarrow Z \\ \textit{initial state } z_0 &= 0 \\ \textit{output alphabet } O &= K \times L \times DOM^{\leq l} \\ \textit{output function } \epsilon &= I \times Z \longrightarrow O \end{aligned}$$

Let E4RO ("eligible for read-out") be a finite predicate on DOM, $E4RO \subseteq DOM$. The transition function δ and the output function ϵ are defined as follows:

$$\begin{aligned} \delta\,(w, W_1, \ldots, W_w, r,\ e, E_1, \ldots, E_e) &= (e', E'_1, \ldots, E'_{e'}) \\ \epsilon\,(w, W_1, \ldots, W_w, r,\ e, E_1, \ldots, E_e) &= (wack, rack, R_1, \ldots, R_{rack}) \end{aligned}$$

Define $e4ro := |\{i \in \{1, \ldots, e\} \mid E_i \in E4RO\}|$. *Then*

$$\begin{aligned} rack &:= \min\{rack, e4ro\} \\ wack &:= \min\{n - e + rack, w\} \\ e' &:= e + wack - rack \end{aligned}$$

Let $(e4ro_j)$ *be the prefix of length rack of the sorted sequence of the indices of the elements eligible for read out. Let* $(\overline{e4ro}_k)$ *be the sorted sequence of the remaining indices. Then, with an update function upd for the entries, the queue operation looks as follows:*

$$\begin{aligned} (R_1, \ldots, R_{rack}) &:= (E_{e4ro_1}, \ldots, E_{e4ro_{rack}}) \\ (E'_1, \ldots, E'_{e'}) &:= \left(upd(E_{\overline{e4ro}_1}), \ldots, upd(E_{\overline{e4ro}_{e-rack}}), W_1, \ldots, W_{wack}\right) \end{aligned}$$

5.4.2 Description of the Interface and Equivalence Criterion

For the implementation of the abstract multiported queue we consider a domain $DOM = \{0,1\}^m$ for some $m \in I\!N$. Table 5.2 shows the definition of the implementation interface of a multiported queue.

5.4.3 Register-Based Implementation

In the register-based implementation, the slots are directly modelled with flip-flops $\mathsf{S}_\star.\mathsf{data}_\star$. A valid register $\mathsf{valid}_\star \in \{0,1\}^n$ indicates which slots hold valid entries. It encodes the fill state of the queue in half-unary encoding, i.e. $\langle \mathsf{valid}_\star \rangle_{hu}$ is the number of allocated queue slots.

Queue Control

This section develops the queue control signals in three steps. The procedure is according to the multiported queue case. The "Read Phase" describes how queue read operation are acknowledged and performed. The "Entry Propagation" describes the adjustment of the queue entries after the read-out. The "Write Phase" describes how queue write operations are acknowledged and how the data reaches the appropriate slots.

Read. Candidates for read-out must be searched in all valid and eligible entries:

$$\mathsf{S}_\star.\mathsf{cand} \quad = \quad \mathsf{S}_\star.\mathsf{valid} \wedge \mathsf{S}_\star.\mathsf{e4ro}$$

The definition of the abstract reservation station queue requires us to find the first l ones in this bit string. A find-first-l-ones half-unary circuit executes this operation. Let $\mathsf{S}_\star.\mathsf{cand} \in \{0,1\}^n$ be the input of this circuit, $\mathsf{mAux}_{\star,\star} \in \{0,1\}^{n \cdot l}$ the output. We have

$$\langle \mathsf{mAux}_{\mathrm{i},\star} \rangle_{hu} \quad = \quad \min\{l, \mathrm{ones}(\mathsf{S}_{\mathrm{i}..0}.\mathsf{cand})\}$$

by definition. As we are interested to determine the number of read-out entries, we compute a half-unary minimum operation

$$\mathsf{m}_{\mathrm{i},\star} \quad := \quad \mathsf{R}_\star.\mathsf{req} \wedge \mathsf{mAux}_{\mathrm{i},\star}$$

and obtain

$$\begin{aligned} \langle \mathsf{m}_{\mathrm{i},\star} \rangle_{hu} \quad &= \quad \min\{\langle \mathsf{R}_\star.\mathsf{req} \rangle_{hu}, l, \mathrm{ones}(\mathsf{S}_{\mathrm{i}..0}.\mathsf{cand})\} \\ &= \quad \min\{\langle \mathsf{R}_\star.\mathsf{req} \rangle_{hu}, \mathrm{ones}(\mathsf{S}_{\mathrm{i}..0}.\mathsf{cand})\} \end{aligned}$$

From this signal array, the read acknowledgement signals can be directly read out from the last line:

$$\mathsf{R}_\star.\mathsf{ack} \quad := \quad \mathsf{m}_{\mathrm{n}-1,\star}$$

To compose the read busses we further need to compute output enable signals $\mathsf{S2R}_{\mathrm{i,j}}$ indicating that slot i shall be mapped to read bus j. Again, the signal array $\mathsf{m}_{\star,\star}$ provides this information. By corollary C.3 (p. 125) the columns $\mathsf{m}_{\star,\mathrm{j}}$ of $\mathsf{m}_{\star,\star}$ being different from 0^n deliver a $\langle \mathsf{R}_\star.\mathsf{ack} \rangle_{hu}$–prefix of the sorted sequence of the indices of the input bits equal to 1. This sequence is given in negated half-unary encoding; to obtain the output signals, we merely compute the edge in them:

$$\mathsf{S2R}_{\star,\mathrm{j}} \quad := \quad \mathrm{markLastOne}\,(\mathsf{m}_{\star,\mathrm{j}})$$

The rest of the read phase works the same as in the multiported queue. In preparation of the write phase, we assume that the read operations have been separately performed before the write operations. The signal bus $\mathsf{validAfterRead}_\star$ indicating the valid slots after the completion of the read phase is obtained by a half-unary subtraction operation:

$$\langle \mathsf{validAfterRead}_\star \rangle_{hu} \quad = \quad \langle \mathsf{valid}_\star \rangle_{hu} - \langle \mathsf{R}_\star.\mathsf{ack} \rangle_{hu}$$

Slot Propagation. The entry propagation in a reservation station queue is more complicated than in a multiported queue. Each entry may be propagated by a different distance since reading of queue entries can occur at different places of the queue and is not restricted to the head.

Let $\mathsf{S_i.prop_j}$ indicate that the propagation distance of entry i is j. The slot propagation is characterized by two rules. A read-out slot is not propagated at all. Any other slot is propagated by the number of entries that have been read out before it.

Define

$$\mathsf{S_i.prop_\star} \quad := \quad \mathsf{m_{i-1,\star}} \wedge \mathrm{markLastOne}\,(\mathsf{m_{i,\star}})$$

Then, for any read-out slot $\mathsf{S_i}$, we have $\langle \mathsf{m_{i-1,\star}} \rangle_{hu} < \langle \mathsf{m_{i,\star}} \rangle_{hu}$ and therefore in $\mathsf{S_i.prop_\star} = 0^l$. Otherwise,

$$\begin{aligned} \langle \mathsf{S_i.prop_\star} \rangle_{hu} &= \mathrm{ones}(\mathsf{S_{i..0}.cand}) \\ &= \mathrm{ones}(\mathsf{S_{i-1..0}.cand}) \end{aligned}$$

Write. The write phase works the same as in the multiported queue (cf. section 5.3.3). In the write phase, we first want to compute the write acknowledgement signals. According to property 3.6 (p. 35) of half-unary encodings, the reversed and negated version of $\mathsf{validAfterRead_\star}$ encodes the number of free entries after read-out:

$$\langle \overline{\mathsf{validAfterRead_{0..n-1}}} \rangle_{hu} \quad = \quad n - e + rack$$

Because $wack = \min\{n - e + rack, w\}$ by definition 5.5, we can compute the write acknowledgement signals by a slice of AND-gates:

$$\begin{aligned} \mathsf{W_\star.ack} &= \mathsf{W_\star.req} \wedge \overline{\mathsf{validAfterRead_{n-k..n-1}}} \\ &= \overline{\overline{\mathsf{W_\star.req}} \wedge \mathsf{validAfterRead_{n-k..n-1}}} \end{aligned}$$

The acknowledged write data must be appended to the valid entries in the queue after read-out. To control the distribution of the write busses, we define the signal array $\mathsf{W2S_{\star,\star}}$. The condition $\mathsf{W2S_{k',n'}} = 1$ indicates that write bus $\mathsf{W_{k'}}$ shall be written to slot $\mathsf{S_{n'}}$.

For $k' = 0$, the bus

$$\mathsf{insPos_\star} \quad := \quad \mathrm{markLastOne}\,(\mathsf{validAfterRead_\star})$$

marks the insertion position, i.e. $\mathsf{W2S_{0,\star}} = \mathsf{W_0.req} \wedge \mathsf{insPos_\star}$. For $k' > 0$ we obtain $\mathsf{W2S_{k',\star}}$ by shifting $\mathsf{insPos_\star}$:

$$\mathsf{W2S_{k',n'}} \quad = \quad \mathsf{W_i.req} \wedge \mathsf{insPos_{n'-k'}} \text{ for } 1 \le n' - k' \le n$$

This ensures that the written entries are appended "en bloc" after the valid queue entries.

Update of the Valid Register. The new valid register $\mathsf{valid'_\star}$ can be computed by a half-unary addition that satisfies:

$$\langle \mathsf{valid'_\star} \rangle_{hu} \quad = \quad \langle \mathsf{validAfterRead_\star} \rangle_{hu} + \langle \mathsf{W_\star.ack} \rangle_{hu}$$

Entry

Figure 5.8 shows the definition of the slot $\mathsf{S_{n'}}$. The register $\mathsf{S_{n'}.data_\star}$ stores the data. Sources for this register are the k write busses $\mathsf{W_\star.data}$, the own data $\mathsf{S_{n'}}$ looped-back for update purposes and the data of the l above slots $\mathsf{S_{n'+l}}, \ldots, \mathsf{S_{n'+1}}$ for slot propagation purposes. Drivers are used to select the data from the different

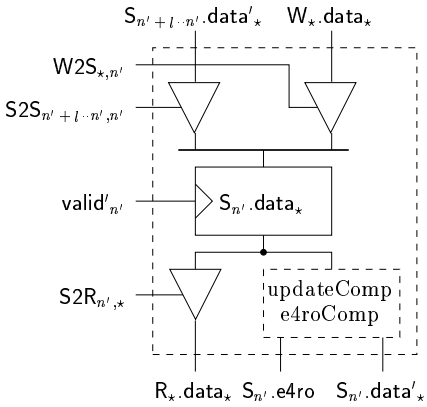

Figure 5.8: Definition of a reservation station queue slot

sources; $W2S_{k',n'} = 1$ indicates that write bus $W_{k'}$ shall write into slot $S_{n'}$. At most one bit of $W2S_{\star,n'}$ can be active by definition of $W2S_{\star,\star}$. The signal

$$S2S_{n'+l',n'} \quad := \quad valid_{n'+l'} \wedge S_{n'+l'}.prop_{l'}$$

for $l' \in \{0,\ldots,l\}$ is used to select $S_{n'+l'}$ as a source. Again, at most one of $S2S_{n'+l..n',n'}$ can be active. The data register is only clocked, if it is valid in the next cycle, i.e. $S_{n'}.data.ce := valid'_{n'}$.

Chapter 6

Perspective

Several aspect of superscalar processor design have not been examined in-depth in this thesis. They are left for further research.

- Modern microprocessors execute multiple control flows with several outstanding branches in parallel. Such execution can be implemented by introduction of a *control flow tag*, which associates each instruction in execution with a specific control flow followed. The reorder buffer will only retire the instructions of the control flow that eventually is know to execute, results for other control flows will be dropped. Duplication of the producer tables is necessary for an individual register renaming and forwarding in each control flow.

  Multiple control flow execution therefore depends on major changes of the machine design; an analysis would be of interest.

- More elaborate branch predictors and fetch mechanism still remain to be examined. [Yeh93] states the importance of such mechanism with the growing instruction-level parallelism of the processor since mispredictions incur a great penalty in performance. Therefore, current processor designs spend more cost for sophisticated control flow predictors.

- Branch problem "solved", the memory operations are likely to limit the machine's performance. In our design, stores are executed in order and loads execute on completed address computation. More elaborate schemes, as developed for example in [AMS97], involve the speculation of memory access addresses by maintaining appropriate history information. Schemes replacing in-order store by something more efficient are also thinkable, although preciseness for such schemes regarding interrupt and roll-back is a most delicate matter.

- Our design provided a superscalar framework for the whole instruction set. In contrast to this, today's microprocessor often provide different levels of superscalarity for fixed-point and for floating point instructions. Additionally, the grouping of reservation stations in a centralized reservation station pool / dispatch stack results in a better utilization of reservation stations on a cost overhead ([Joh91, WS84, AKT86]).

  The effect of such choices on cost and performance remains to be examined.

- Simulations are missing to measure the exact impact of the parameters of our design. The conservative CPI rate estimate made in chapter 4 could be could be corrected, improving the quality of this design.

- The hardware model used neglects the effect of wiring and fanout. A study of this design in a model dealing with layout, like [PS98], could adequately cope with the large wiring overhead present in our design.

Appendix A

DLX Instruction Set Architecture

This instruction set is taken from [MP95, MP00] with minimal modifications.

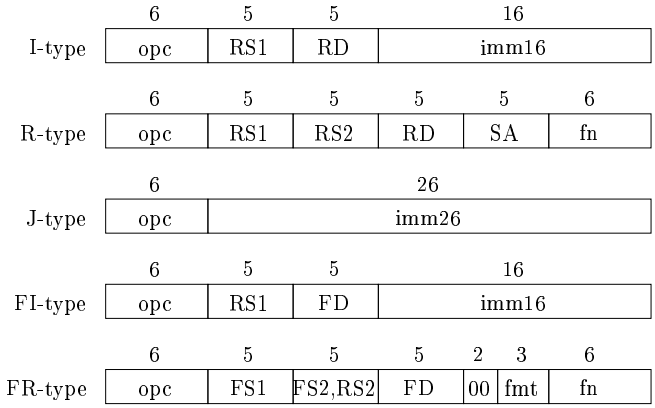

Figure A.1: Instruction formats of the DLX

$I_{31..26}$	Mnemonic	Effect
Control Flow Operation		
000010	`j`	$\langle PC_\star \rangle_2 =_{2^{32}} \langle PC_\star \rangle_2 + 4 + [imm26_\star]$
000011	`jal`	$\langle R31_\star \rangle_2 =_{2^{32}} \langle PC_\star \rangle_2 + 4, \quad PC_\star = imm26_\star$
Exception Control		
111110	`trap`	$EPC_\star = PC_\star, PC_\star = SISR_\star$ $ESR_\star = SR_\star, ECA_\star = MCA_\star$ $SR_\star = 0^{32}, EDATA_\star = sext_{32}(imm26_\star)$ clear CA but catch new interrupt events
111111	`rfe`	$SR_\star = ESR_\star; PC_\star = EPC_\star$

Table A.1: J-type instructions

$I_{31..26}$	Mnemonic	Effecto
Load / Store, $\mathsf{mem}_\star = \mathsf{M}[\langle \mathsf{RS1}_\star \rangle_2 + [\mathsf{imm16}_\star]]$		
1000000	`lb`	$\mathsf{RD}_\star = \mathrm{sext}_{32}(\mathsf{mem}_{7..0})$
100001	`lh`	$\mathsf{RD}_\star = \mathrm{sext}_{32}(\mathsf{mem}_{15..0})$
100011	`lw`	$\mathsf{RD}_\star = \mathsf{mem}_{31..0}$
100100	`lbu`	$\mathsf{RD}_\star = (0^{24}, \mathsf{mem}_{7..0})$
100101	`lhu`	$\mathsf{RD}_\star = (0^{16}, \mathsf{mem}_{15..0})$
101000	`sb`	$\mathsf{mem}_{7..0} = \mathsf{RD}_{7..0}$
101001	`sh`	$\mathsf{mem}_{15..0} = \mathsf{RD}_{15..0}$
101011	`sw`	$\mathsf{mem}_{31..0} = \mathsf{RD}_\star$
Arithmetic / Logical Operation		
001000	`addi`	$\langle \mathsf{RD}_\star \rangle_2 =_{2^{32}} [\mathsf{RS1}_\star] + [\mathsf{imm16}_\star]$
001001	`addiu`	$\langle \mathsf{RD}_\star \rangle_2 =_{2^{32}} [\mathsf{RS1}_\star] + [\mathsf{imm16}_\star]$ (no overflow)
001010	`subi`	$\langle \mathsf{RD}_\star \rangle_2 =_{2^{32}} [\mathsf{RS1}_\star] + [\mathsf{imm16}_\star]$
001011	`subiu`	$\langle \mathsf{RD}_\star \rangle_2 =_{2^{32}} [\mathsf{RS1}_\star] + [\mathsf{imm16}_\star]$ (no overflow)
001100	`andi`	$\mathsf{RD}_\star = \mathsf{RS1}_\star \wedge \mathrm{sext}_{32}(\mathsf{imm16}_\star)$
001101	`ori`	$\mathsf{RD}_\star = \mathsf{RS1}_\star \vee \mathrm{sext}_{32}(\mathsf{imm16}_\star)$
001110	`xori`	$\mathsf{RD}_\star = \mathsf{RS1}_\star \oplus \mathrm{sext}_{32}(\mathsf{imm16}_\star)$
001110	`lhgi`	$\mathsf{RD}_\star = (\mathsf{imm16}_\star, 0^{16})$
Test Set Operation [a]		
011000	`clri`	$\mathsf{RD}_\star = 0^{32}$
011001	`sgri`	$\mathsf{RD}_\star = ([\mathsf{RS1}_\star] > [\mathsf{imm16}_\star] \; ? \; 0^{31}1 : 0^{32})$
011010	`seqi`	$\mathsf{RD}_\star = ([\mathsf{RS1}_\star] = [\mathsf{imm16}_\star] \; ? \; 0^{31}1 : 0^{32})$
011011	`sgei`	$\mathsf{RD}_\star = ([\mathsf{RS1}_\star] \geq [\mathsf{imm16}_\star] \; ? \; 0^{31}1 : 0^{32})$
011100	`slsi`	$\mathsf{RD}_\star = ([\mathsf{RS1}_\star] < [\mathsf{imm16}_\star] \; ? \; 0^{31}1 : 0^{32})$
011101	`snei`	$\mathsf{RD}_\star = ([\mathsf{RS1}_\star] \neq [\mathsf{imm16}_\star] \; ? \; 0^{31}1 : 0^{32})$
011110	`slei`	$\mathsf{RD}_\star = ([\mathsf{RS1}_\star] \leq [\mathsf{imm16}_\star] \; ? \; 0^{31}1 : 0^{32})$
011111	`seti`	$\mathsf{RD}_\star = 0^{31}1$
Control Flow Operation		
000100	`beqz`	$\langle \mathsf{PC}_\star \rangle_2 =_{2^{32}} \langle \mathsf{PC}_\star \rangle_2 + 4 + (\mathsf{RS}_\star = 0^{32} \; ? \; [\mathsf{imm16}_\star] : 0)$
000101	`bnez`	$\langle \mathsf{PC}_\star \rangle_2 =_{2^{32}} \langle \mathsf{PC}_\star \rangle_2 + 4 + (\mathsf{RS}_\star \neq 0^{32} \; ? \; [\mathsf{imm16}_\star] : 0)$
010110	`jr`	$\mathsf{PC}_\star = \mathsf{RS1}_\star$
010111	`jalr`	$\langle \mathsf{R31}_\star \rangle_2 =_{2^{32}} \langle \mathsf{PC}_\star \rangle_2 + 4, \quad \mathsf{PC}_\star = \mathsf{RS1}_\star$

[a] The six relations defined are redundant for compiler-generated code in most situations: Compilers can replace relations "$<$", "$\neq$" and "$\leq$" by their inverted operations in arithmetic expressions. The redundant operations could be replaced by unsigned comparisons which miss in the ISA presented. We will not follow this approach, however, to maintain binary compatibility to [MP95, MP00] and because the underlying ISA is not of primary interest of the thesis.

Table A.2: I-type instructions

$I_{31..26}$	$I_{5..0}$	Mnemonic	Effect
Shift Operation			
000000	000000	`slli`	$RD_\star = RS1_\star \ll SA_\star$
000000	000001	`slai`	$RD_\star = RS1_\star \ll SA_\star$ (arithmetic shift)
000000	000010	`srli`	$RD_\star = RS1_\star \gg SA_\star$
000000	000011	`srai`	$RD_\star = RS1_\star \gg SA_\star$ (arithmetic shift)
000000	000100	`slli`	$RD_\star = RS1_\star \ll RS_{4..0}$
000000	000101	`slai`	$RD_\star = RS1_\star \ll RS_{4..0}$ (arithmetic shift)
000000	000110	`srli`	$RD_\star = RS1_\star \gg RS_{4..0}$
000000	000111	`srai`	$RD_\star = RS1_\star \gg RS_{4..0}$ (arithmetic shift)
Data Transfer Operation			
000000	010000	`movs2i`	$RD_\star = SA_\star$
000000	010001	`movi2s`	$SA_\star = RD_\star$
Arithmetic / Logical Operation			
000000	100000	`add`	$\langle RD_\star \rangle_2 =_{2^{32}} [RS1_\star] + [RS2_\star]$
000000	100001	`addu`	$\langle RD_\star \rangle_2 =_{2^{32}} [RS1_\star] + [RS2_\star]$ (no overflow)
000000	100010	`sub`	$\langle RD_\star \rangle_2 =_{2^{32}} [RS1_\star] + [RS2_\star]$
000000	100011	`subu`	$\langle RD_\star \rangle_2 =_{2^{32}} [RS1_\star] + [RS2_\star]$ (no overflow)
000000	001100	`and`	$RD_\star = RS1_\star \wedge RS2_\star$
000000	001101	`or`	$RD_\star = RS1_\star \vee RS2_\star$
000000	001110	`xor`	$RD_\star = RS1_\star \oplus RS2_\star$
000000	001110	`lhg`	$RD_\star = (RS2_\star, 0^{16})$
Test Set Operation			
000000	101000	`clri`	$RD_\star = 0^{32}$
000000	101001	`sgri`	$RD_\star = ([RS1_\star] > [RS2_\star] \; ? \; 0^{31}1 : 0^{32})$
000000	101010	`seqi`	$RD_\star = ([RS1_\star] = [RS2_\star] \; ? \; 0^{31}1 : 0^{32})$
000000	101011	`sgei`	$RD_\star = ([RS1_\star] \geq [RS2_\star] \; ? \; 0^{31}1 : 0^{32})$
000000	101100	`slsi`	$RD_\star = ([RS1_\star] < [RS2_\star] \; ? \; 0^{31}1 : 0^{32})$
000000	101101	`snei`	$RD_\star = ([RS1_\star] \neq [RS2_\star] \; ? \; 0^{31}1 : 0^{32})$
000000	101110	`slei`	$RD_\star = ([RS1_\star] \leq [RS2_\star] \; ? \; 0^{31}1 : 0^{32})$
000000	101111	`seti`	$RD_\star = 0^{31}1$

Table A.3: R-type instructions

$I_{31..26}$	Mnemonic	Effect
Load / Store, $mem_{63..0} = M[\langle RS1_\star \rangle_2 + [imm16_\star]]$		
110001	`load.s`	$FD_{31..0} = mem_{31..0}$
110101	`load.d`	$FD_{63..0} = mem_{63..0}$
111001	`store.s`	$mem_{31..0} = FD_{31..0}$
111101	`store.d`	$mem_{31..0} = FD_{63..0}$
Control Flow Operation		
000110	`fbeqz`	$\langle PC_\star \rangle_2 =_{2^{32}} \langle PC_\star \rangle_2 + 4 + (FCC = 0 \; ? \; [imm16_\star] : 0)$
000111	`fbnez`	$\langle PC_\star \rangle_2 \neq_{2^{32}} \langle PC_\star \rangle_2 + 4 + (FCC = 1 \; ? \; [imm16_\star] : 0)$

Table A.4: FI-type instructions

$I_{31..26}$	$I_{5..0}$	$I_{8..6}$	Mnemonic	Effect
Arithmetic / Compare Operation, $\mathsf{fmt} := I_{8..6}$				
010001	000000		`fadd.s/d`	$\mathsf{FD}_\star = \text{ieeeAdd}(\mathsf{FS1}_\star, \mathsf{FS2}_\star, \mathsf{fmt}_\star)$
010001	000001		`fsub.s/d`	$\mathsf{FD}_\star = \text{ieeeSub}(\mathsf{FS1}_\star, \mathsf{FS2}_\star, \mathsf{fmt}_\star)$
010001	000010		`fmul.s/d`	$\mathsf{FD}_\star = \text{ieeeMul}(\mathsf{FS1}_\star, \mathsf{FS2}_\star, \mathsf{fmt}_\star)$
010001	000011		`fdiv.s/d`	$\mathsf{FD}_\star = \text{ieeeDiv}(\mathsf{FS1}_\star, \mathsf{FS2}_\star, \mathsf{fmt}_\star)$
010001	000100		`fneg.s/d`	$\mathsf{FD}_\star = \text{ieeeNeg}(\mathsf{FS1}_\star, \mathsf{fmt}_\star)$
010001	000100		`fabs.s/d`	$\mathsf{FD}_\star = \text{ieeeAbs}(\mathsf{FS1}_\star, \mathsf{fmt}_\star)$
010001	000100		`fsqt.s/d`	$\mathsf{FD}_\star = \text{ieeeSqt}(\mathsf{FS1}_\star, \mathsf{fmt}_\star)$
010001	000100		`frem.s/d`	$\mathsf{FD}_\star = \text{ieeeRem}(\mathsf{FS1}_\star, \mathsf{FS2}_\star, \mathsf{fmt}_\star)$
010001	$11\mathsf{c}_{3..0}$		`fc.cond.s/d`	$\mathsf{FCC} = \text{ieeeCond}(\mathsf{FS1}_\star, \mathsf{FS2}_\star, \mathsf{c}_\star, \mathsf{fmt}_\star)$
Data Transfer Operation				
010001	001000	000	`fmov.s`	$\mathsf{FD}_{31..0} = \mathsf{FS1}_{31..0}$
010001	001000	001	`fmov.d`	$\mathsf{FD}_{63..0} = \mathsf{FS1}_{63..0}$
010001	001001		`mf2i`	$\mathsf{RS}_\star = \mathsf{FS1}_{31..0}$
010001	001010		`mi2f`	$\mathsf{FD}_{31..0} = \mathsf{RS}_\star$
Conversion				
010001	100000	001	`cvt.s.d`	$\mathsf{FD}_\star = \text{ieeeConv}(\mathsf{FS1}_\star, s, d)$
010001	100000	100	`cvt.s.i`	$\mathsf{FD}_\star = \text{ieeeConv}(\mathsf{FS1}_\star, s, i)$
010001	100001	000	`cvt.d.s`	$\mathsf{FD}_\star = \text{ieeeConv}(\mathsf{FS1}_\star, d, s)$
010001	100001	100	`cvt.d.d`	$\mathsf{FD}_\star = \text{ieeeConv}(\mathsf{FS1}_\star, d, i)$
010001	100100	000	`cvt.i.s`	$\mathsf{FD}_\star = \text{ieeeConv}(\mathsf{FS1}_\star, i, s)$
010001	100100	001	`cvt.i.d`	$\mathsf{FD}_\star = \text{ieeeConv}(\mathsf{FS1}_\star, i, d)$

Table A.5: FR-type instructions

Appendix B

Sample Branch Predictor Unit

B.1 Branch Predictor

We only implement a simple branch predictor in our design. The design of more a complex branch predictor lies beyond the scope of this thesis. Our branch predictor is a branch target buffer design using a 2-bit saturating counter scheme. The description and analysis such schemes can be found in [LS84]. [1]

B.1.1 Implementation

Each branch, identified by its address, is associated with a status information $(c_1, c_0) \in \{0,1\}^2$. This information is the state of an automaton called 2-bit saturating counter. Figure B.1 shows this automaton. The automaton is used in two ways. First, branches are predicted based on the state (c_1, c_0). A branch is predicted taken (predT) if $c_1 = 1$; it is predicted fall-through (predFT) if $c_1 = 0$. After the prediction of a branch, the state is updated according to the real outcome of that branch. A taken result increments the state number $\langle c_\star \rangle_2$ up to its maximum while a fall-through result decrements the state number $\langle c_\star \rangle_2$, but not below 0. This way, the counter "saturates" for many taken or for many fall-through results. A simple proof shows that the next state (c'_1, c'_0) can be computed out of (c_1, c_0) and T (for taken result) with the following equations:

$$\begin{aligned} c'_0 &:= c_1 \wedge \overline{c_0} \vee \overline{(c_1 \oplus c_0)} \wedge T \\ c'_1 &:= c_1 \wedge \overline{T} \vee c_0 \wedge T \end{aligned}$$

Figure B.2 shows the data paths for a 2-bit saturating counter branch predictor. The predictor is interfaced by 6 signals. Requests for predictions are made by setting $R = 1$ and ID to the ID of the branch. We identify branches in a direct-mapped approach by part of its PC The predictor outputs the prediction result with the signal PRED; $PRED = 1$ indicates a taken branch. If a prediction is actually used, the signal DO must be activated.

The remaining part of the interface deals with the verification of predictions. Signal WB indicates that a branch has been verified. In this case MP is set to 1 if a misprediction has been detected.

[1] Note that the DLX only implements branches with *static targets*, i.e. the target is encoded as a constant offset to the PC of the branch instructions. The original branch target buffer design, presented by Lee and Smith, also treated *dynamic targets* by storing an additional target item, giving the buffer its name. Dropping this target item simplifies the description of the scheme.

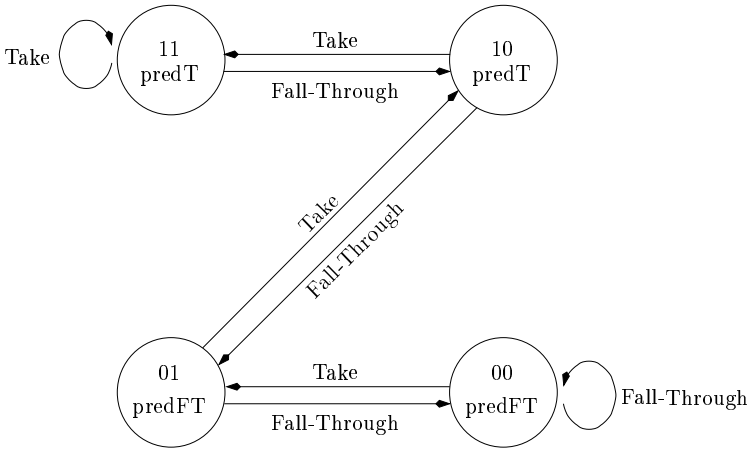

Figure B.1: State automaton for the 2–bit saturating counter

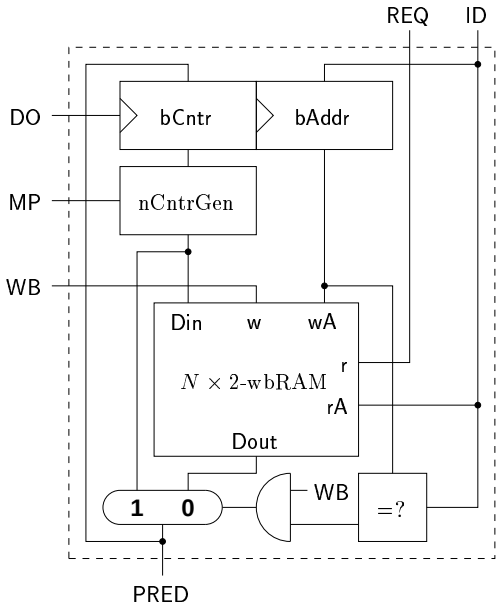

Figure B.2: Data paths for a 2–bit saturating counter branch predictor

The predictor features a central $N \times 2$-wbRAM (see below) storing N 2-bit saturating counters. If a prediction is requested, this RAM is read out at the address $\mathsf{ID}_\star$, returning the counter $(\mathsf{c}_1\,, \mathsf{c}_0)$ of the requested branch. From this counter, we can directly take the upper bit c_1 to obtain the prediction PRED. In case this prediction is actually used, i.e. $\mathsf{DO} = 1$, the address $\mathsf{ID}_\star$ and the counter $(\mathsf{c}_1\,, \mathsf{c}_0)$ are clocked into two buffer registers $\mathsf{bAddr}_\star$ and $\mathsf{bCntr}_\star$. There it waits for the prediction verification.

If $\mathsf{WB} = 1$, the verification result MP is available. The new counter $(\mathsf{c'}_1\,, \mathsf{c'}_0)$ can be computed by the equations above. Note that $\mathsf{T} = \mathsf{c}_1 \oplus \mathsf{MP}$. A write access to address $\mathsf{bAddr}_\star$ writes back the counter. The writing counter is forwarded for reading, if it is requested in the same round: so if $\mathsf{ID}_\star = \mathsf{bAddr}_\star$ and $\mathsf{WB} = 1$ then $(\mathsf{c'}_1\,, \mathsf{c'}_0)$ is returned.

A close observation of our processor reveals that the underlying RAM of the branch predictor need not have two "real" ports for reading and writing. A cycle after prediction, our processor will not requests for another prediction, because it first has to resolve the outstanding speculation. This means, that between two possible write-backs due to prediction verification there is always a cycle where no prediction is requested, i.e. no read access occurs. So-called write-buffered RAM, implemented in section B.2, exploits this property and is used to build a fast counter RAM for the predictor.

B.1.2 Integration in the processor

The interface of the BPU developed in the previous section are defined as follows:

$$\begin{aligned}
\mathsf{BPU.ID}_\star &:= \mathsf{cfi.opc}_{31..2}\\
\mathsf{BPU.REQ} &:= \mathsf{cfi.valid}\\
\mathsf{BPU.DO} &:= \mathsf{cfi.doPred}\\
\mathsf{BPU.MP} &:= \mathsf{bcu.mp}\\
\mathsf{BPU.WB} &:= \mathsf{bcu.valid} \wedge \overline{\mathsf{bcu.jump}}\\
\mathsf{bpu.pred} &:= \mathsf{cfi.t.imm16}
\end{aligned}$$

Note that allowCFI will be zero the cycle after a prediction has been made.

B.2 Write-Buffered RAM

This section describes the implementation of a special 2-ported RAM that requires a cycle of no read requests between two write requests. Such RAM can be built with conventional single-ported RAM and an additional buffer for write operations.

Figures B.3 and B.4 shows the implementation of a $N \times m$-write-buffered RAM featuring the central $N \times m$-RAM. We continue describing the write and the read operation.

B.2.1 Write

Let $n := \lceil \log_2 N \rceil$. Written data is always stored in a buffer register first; $\mathsf{bAddr}_\star \in \{0,1\}^n$ buffers the write address, $\mathsf{bDin}_\star \in \{0,1\}^m$ buffers the incoming data. An additional register $\mathsf{valid} \in \{0,1\}$ equals one, if the buffer contains data to be written.

A full buffer is written to the RAM, if no read request has to be fulfilled:

$$\mathsf{wb} := \mathsf{valid} \wedge \overline{\mathsf{r}}$$

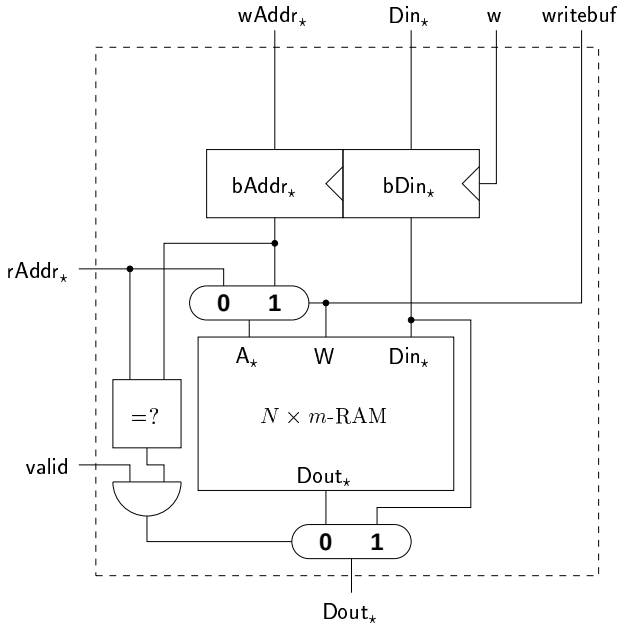

Figure B.3: Implementation of write-buffered RAM

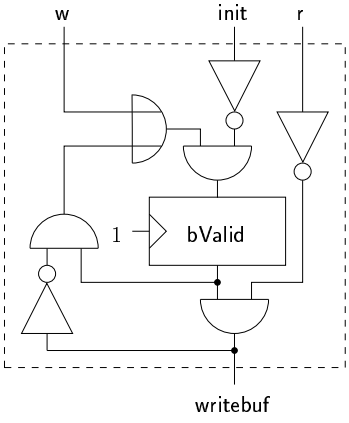

Figure B.4: Implementation of the valid register

In this case the write signal is set and the buffered write address is selected as the RAM access address. The buffer valid register bValid is set to 1 iff it was valid and has not yet been written or if new write data has arrived in the same cycle:

$$\mathsf{valid}' \quad := \quad (\mathsf{bValid} \wedge \overline{\mathsf{wb}}) \vee \mathsf{w}$$

On initialization valid is reset.

B.2.2 Read

As seen above, no write operation will be executed, if a read is requested by $\mathsf{r} = 1$. The address multiplexor will forward the read address to the address input of the RAM. Additionally, a forwarding logic is implemented for the data stored in the write buffer. A compare circuit "=?" is used to determine matching read and buffer address. If this circuit signals matching addresses *and* the buffer is valid, read data is not taken from the RAM but forwarded from the buffer data register bDin.

Appendix C

Auxiliary Circuits

This appendix describes three auxiliary circuits used in the implementation of our machine.

C.1 Find-First-One Half-Unary

C.1.1 Definition and Construction

This section implements a find-first-one half-unary circuit $\mathrm{FF1hu}_n$ frequently used in this thesis. The $\mathrm{FF1hu}_n$ receives an input bus $\mathsf{a}_\star \in \{0,1\}^n$ and produces the output bus $\mathsf{o}_\star \in \{0,1\}^n$ with the following property:

$$\mathsf{o}_\mathsf{i} = 1 \quad \Longleftrightarrow \quad \exists\, j \in \{0,\ldots,i\} : \mathsf{a}_\mathsf{j} = 1$$

Equivalently, the negated output is the number of leading zeroes at the beginning of $\mathsf{a}_\star$ in half-unary encoding:

$$\langle \overline{\mathsf{o}_\star} \rangle_2 = k \quad \Longleftrightarrow \quad \mathsf{a}_{\mathsf{k}..0} = 10^k$$

For $n = 1$, $\mathrm{FF1hu}_n$ is defined by $\mathsf{o}_0 := \mathsf{a}_0$. For $n > 1$, figure C.1 shows the recursive definition. It has delay of $\mathcal{O}(\log n)$ and cost of $\mathcal{O}(n \log n)$. Alternatively, a find-first-one half-unary circuit can be implemented with a parallel-prefix OR circuit (cf. [Lei99]).

C.1.2 Correctness.

The OR gate used in the construction can also be seen as the first bit of a half-unary adder for two 1-bit encodings. This way, the correctness proof of section C.2 for a generalized version of the circuit applies.

C.2 Find-First-k-Ones

C.2.1 Definition

Let the function $\mathrm{ones}(\cdot)$ sum up the ones of a binary argument, i.e.

$$\mathrm{ones}(\mathsf{x}_{\mathsf{m}-1..0}) \quad := \quad \sum_{i<m} \mathsf{x}_\mathsf{i}$$

Let $k, n \in I\!N$, $\mathsf{a}_\star \in \{0,1\}^n$. Define

$$O_j \quad := \quad \min\{k, \mathrm{ones}(\mathsf{a}_{\mathsf{j}..0})\}$$

On input $\mathsf{a}_\star$ a n-bit find-first-k-ones half-unary circuit FFk1hu$_n$ computes the output array $\mathsf{o}_{\star,\star} \in \{0,1\}^{n \cdot k}$ such that

$$\langle \mathsf{o}_{\mathrm{j},\star} \rangle_{hu} \quad = \quad O_j$$

C.2.2 Construction

For $k \geq n = 1$ the FFk1hu$_n$ is defined by

$$\begin{aligned} \mathsf{o}_{1,1} &:= \mathsf{a}_1 \\ \mathsf{o}_{1,\mathrm{i}} &:= 0 \end{aligned}$$

Now assume the construction of FFk1hu$_n$. We construct FFk1hu$_{2n}$ using two FFk1hu$_n$ and n half-unary adders (cf. equation 3.14) in the following way. One FFk1hu$_n$ receives

$$\mathsf{al}_\star \quad := \quad \mathsf{a}_{\mathrm{n}-1\ldots 0}$$

and computes $\mathsf{ol}_{\star,\star} \in \{0,1\}^{n \cdot k}$. The other receives

$$\mathsf{ah}_\star \quad := \quad \mathsf{a}_{2\mathrm{n}\ldots \mathrm{n}}$$

and computes $\mathsf{oh}_{\star,\star} \in \{0,1\}^{n \cdot k}$. The outputs of the FF$k$1hu$_{2n}$ are computed such:

$$\mathsf{o}_{\mathrm{j},\star} \quad := \quad \begin{cases} \mathsf{ol}_{\mathrm{j},\star} & \text{for } j \in \{0,\ldots,n-1\} \\ \mathrm{huAdd}(\mathsf{oh}_{\mathrm{j}-\mathrm{n},\star}, \mathsf{ol}_{\mathrm{n}-1,\star}) & \text{for } j \in \{n,\ldots,2n-1\} \end{cases}$$

Only the k lower bits of the add operations are used; this puts an upper bound to the values $\langle \mathsf{o}_{\mathrm{j},\star} \rangle_{hu}$ computed as in the definition of O_j.

C.2.3 Correctness

In an inductive proof we assume the correctness of FFk1hu$_n$ to show the correctness of FFk1hu$_{2n}$. According to the construction of FFk1hu$_{2n}$ we distinguish two cases.

- $j \in \{0,\ldots,n-1\}$. We have:

$$\begin{aligned} \langle \mathsf{o}_{\mathrm{j},\star} \rangle_{hu} &= \langle \mathsf{ol}_{\mathrm{j},\star} \rangle_{hu} \\ &= O_j \end{aligned}$$

- $j \in \{n,\ldots,2n-1\}$. We have:

$$\begin{aligned} \langle \mathsf{o}_{\mathrm{j},\star} \rangle_{hu} &= \min\left\{k, \langle \mathrm{huAdd}(\mathsf{oh}_{\mathrm{j}-\mathrm{n},\star}, \mathsf{ol}_{\mathrm{n}-1,\star}) \rangle_{hu}\right\} \\ &= \min\left\{k, \langle \mathsf{oh}_{\mathrm{j}-\mathrm{n},\star} \rangle_{hu} + \langle \mathsf{ol}_{\mathrm{n}-1,\star} \rangle_{hu}\right\} \\ &= \min\left\{k, \mathrm{ones}(\mathsf{a}_{\mathrm{j}\ldots\mathrm{n}}) + \mathrm{ones}(\mathsf{a}_{\mathrm{n}-1\ldots 0})\right\} \\ &= \min\left\{k, \mathrm{ones}(\mathsf{a}_{\mathrm{j}\ldots 0})\right\} \\ &= O_j \end{aligned}$$

C.2.4 Different Interpretation

The following two lemmas satisfy a column-wise interpretation of the output array $\mathsf{o}_{\star,\star}$. This way, a FFk1hu$_n$ can be used to actually *find* the first k ones, i.e. mark up their positions.

Lemma C.1 *Let* $\mathsf{o}_{\star,\star} \in \{0,1\}^{k \cdot n}$ *satisfy* $\langle \mathsf{o}_{\mathrm{j},\star} \rangle_{hu} = O_j$. *Then:*

$$O_{j-1} = O_j - 1 = l \quad \Longrightarrow \quad \langle \overline{\mathsf{o}_{\star,\mathrm{l}}} \rangle_{hu} = j$$

Proof. Assume $\mathrm{ones}(\mathsf{a}_{(\mathrm{j}-1)\cdot\cdot 0}) = \mathrm{ones}(\mathsf{a}_{\mathrm{j}\cdot\cdot 0}) - 1 = l$. Because of half-unary number encoding, we have $\mathsf{o}_{\mathrm{j}-1,\mathrm{l}} = 0 \wedge \mathsf{o}_{\mathrm{j},\mathrm{l}} = 1$. Since the ones$(\cdot)$ function is monotonous in growing inputs, i.e. $\mathrm{ones}(\mathsf{x}_{\mathrm{m}\cdot\cdot 0}) \geq \mathrm{ones}(\mathsf{x}_{\mathrm{m}-1\cdot\cdot 0})$, the l–th column of $\mathsf{o}_{\star,\star}$ satisfies:

$$\mathsf{o}_{\mathrm{j}',\mathrm{l}} \quad = \quad \begin{cases} 0 & \text{for } j' \in \{0,\ldots,j-1\} \\ 1 & \text{for } j' \in \{j,\ldots,n-1\} \end{cases}$$

Therefore $\langle \overline{\mathsf{o}_{\star,\mathrm{l}}} \rangle_{hu} = j$.

Lemma C.2 *Let* $\mathsf{o}_{\star,\star} \in \{0,1\}^{k\cdot n}$ *satisfy* $\langle \mathsf{o}_{\mathrm{j},\star} \rangle_{hu} = O_j$. *Then:*

$$\mathrm{ones}(\mathsf{a}_{\mathrm{n}-1\cdot\cdot 0}) = l \quad \Longrightarrow \quad \forall\, l' \geq l : \langle \overline{\mathsf{o}_{\star,\mathrm{l}'}} \rangle_{hu} = n$$

Proof. With $\langle \mathsf{o}_{\mathrm{n}-1,\star} \rangle_{hu} = \mathrm{ones}(\mathsf{a}_{\mathrm{n}-1\cdot\cdot 0}) = l$ we obtain $\mathsf{o}_{\mathrm{n}-1,\mathrm{k}-1\cdot\cdot\mathrm{l}} = 0^{k-l}$. Since $\mathrm{ones}(\mathsf{x}_{\mathrm{m}\cdot\cdot 0}) \geq \mathrm{ones}(\mathsf{x}_{\mathrm{m}-1\cdot\cdot 0})$, this results in the claim $\mathsf{o}_{\star,\mathrm{k}-1\cdot\cdot\mathrm{l}} = 0^{(k-l)n}$.

Corollary C.3 (Sorting) *Let* $K := O_{n-1}$. *Define*

$$s_l \quad := \quad \langle \mathsf{o}_{\star,\mathrm{l}} \rangle_{hu} \quad \textit{for } l \in \{0,\ldots,K-1\}$$

Then $(s_1,\ldots,s_{K-1})$ *is the prefix of length* K *of the sorted sequences of indices of the inputs bits being* 1. *Formally:*

$$s_l < s_{l+1} \quad \wedge \quad \mathsf{i}_{\mathrm{s_{l+1}}\cdot\cdot\mathrm{s_l}} = \left(1, 0^{s_{l+1}-s_l-1}, 1\right)$$

Proof. The corollary follows from lemmas C.1 and C.2.

C.3 Multiple Incrementer

The circuit K-MInc$_n$, $K = 2^k$ and $k, n \in I\!N$, computes the following function:

$$\begin{array}{rccc} K\text{–minc}_n : & \{0,1\}^n & \longrightarrow & \{0,1\}^{(K+1)\cdot n} \\ & \mathsf{a}_\star & \longmapsto & \mathsf{o}_{\star,\star} \end{array}$$

with

$$\langle \mathsf{o}_{\mathrm{K}',\star} \rangle_2 \quad \equiv_{2^n} \quad \langle \mathsf{a}_\star \rangle_2 + K'$$

We develop an implementation suitable for small k. The computation of the outputs proceeds in two steps. The input string $\mathsf{a}_\star$ is split into a high and a low part:

$$\begin{array}{rcl} \mathsf{al}_\star & := & \mathsf{a}_{\mathrm{k}-1\cdot\cdot 0} \\ \mathsf{ah}_\star & := & \mathsf{a}_{\mathrm{n}-1\cdot\cdot\mathrm{k}} \end{array}$$

To generate the low parts of the results $\mathsf{ol}_{\star,\star}$ we use a $2K(k+1)$–bit barrel right-shifter SH. We let the shifter operate on $2K$ groups of $(k+1)$ bits length. It receives $\mathsf{al}_\star$ as shift distance and its inputs are the bit string of length $(k+1)$ in lexicographical order:

$$\begin{array}{rcl} \mathsf{SH.input}_{\mathrm{i},\star} & := & \mathrm{bin}\,(\mathrm{i}) \\ \mathsf{SH.dist}_\star & := & \mathsf{al}_\star \\ \mathsf{ol}_{\mathrm{i},\star} & := & \mathsf{SH.output}_{\mathrm{i},\star} \end{array}$$

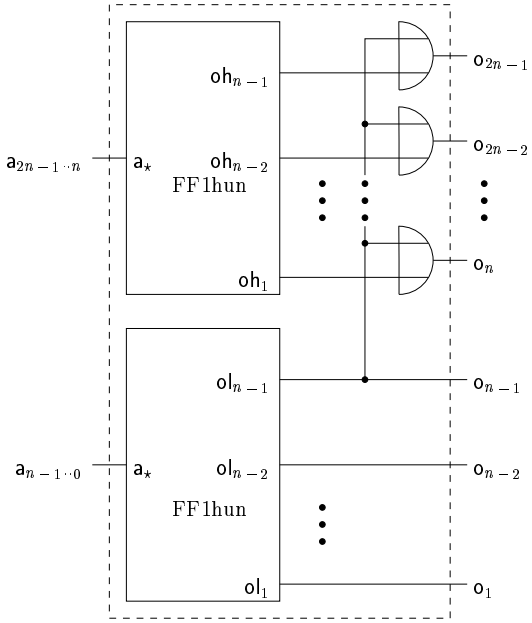

Figure C.1: Definition of a find-first-one half-unary circuit

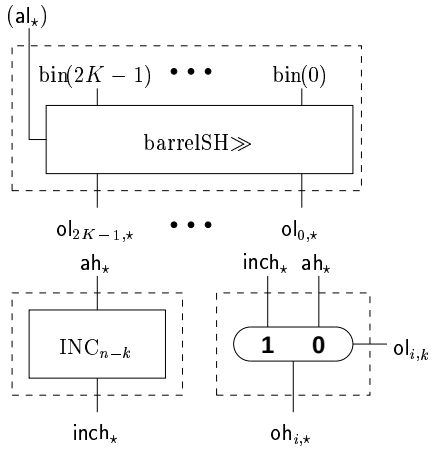

Figure C.2: The multiple incrementer

Thereby the shifter computes

$$\begin{array}{rcl} \langle \mathsf{ol}_{\mathrm{i},\star} \rangle_2 & = & \langle \mathrm{bin}(i) + \langle \mathsf{al}_\star \rangle_2 \quad mod\, 2K \rangle_2 \\ & \equiv_{2K} & i + \langle \mathsf{al}_\star \rangle_2 \end{array}$$

The uppermost bit $\mathsf{ol}_{\mathrm{i,k}}$ can be used to select between the original and an incremented version $\mathsf{inch}_\star$, $\langle \mathsf{inch}_\star \rangle_2 \equiv_{2^{n-k}} \langle \mathsf{ah}_\star \rangle_2 + 1$, of the input bus in conditional-sum-adder fashion:

$$\mathsf{oh}_{\mathrm{i},\star} \quad := \quad (\mathsf{ol}_{\mathrm{i,k}} \ ?\ \mathsf{ah}_\star \ :\ \mathsf{inch}_\star)$$

The output busses are composed as

$$\mathsf{o}_{\mathrm{i},\star} \quad := \quad (\mathsf{oh}_{\mathrm{i},\star}, \mathsf{ol}_{\mathrm{i,k}-1\cdot\cdot 0})$$

Bibliography

[AKT86] R. D. Acosta, J. Kjelstrup, and H. C. Torng. An instruction issuing approach to enhancing performance in multiple function unit processors. In *IEEE Transactions on Computers*, volume C-35, pages 815–828. 1986.

[AMS97] T.N. Vijaykumar A. Moshovos, S. E. Breach and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Annual International Symposium on Computer A rchitecture Conference*, 1997.

[AS95] Don Anderson and Tom Shanley. *Pentium Processor System Architecture, second edition.* Addison-Wesley, Amsterdam;Sydney;Singapore, 1995.

[Bha96] Dileep P. Bhandarkar. *Alpha Implementations and Architecture, Complete Reference and Guide.* Digital Press, Boston;Oxford;Tokyo, 1996.

[Del98] Peter Dell. Die Auswirkung von Mechanismen zur Out-of-Order Ausführung auf den Cyclecount von RISC-Architekturen. Master's thesis, Universität des Saarlandes, FB. Informatik, 7/ 1998.

[Ger98] Nikolaus Gerteis. Die Auswirkung von Mechanismen für die präzise Interruptbehandlung auf den Cyclecount von RISC-Prozessoren. Master's thesis, Universität des Saarlandes, FB. Informatik, 4/ 1998.

[Grü94] Thomas Grün. *Quantitative Analyse von I/O-Architekturen.* PhD thesis, Universität des Saarlandes, FB. Informatik, 1994.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[KP95] Jörg Keller and Wolfgang J. Paul. *Hardwaredesign – Formaler Entwurf digitaler Schaltungen.* Stuttgart, Leipzig, 1995.

[Krö99] Daniel Kröning. Design and evaluation of a RISC processor with a Tomasulo scheduler. Master's thesis, University of Saarland, Computer Science Department, Germany, 1999.

[Lei99] Holger Leister. *Quantitative Analysis of Precise Interrupt Mechanisms for Processors with Out-Of-Order Execution.* PhD thesis, University of Saarland, Computer Science Department, Germany, 1999.

[LS84] Johnny K. F. Lee and Alan J. Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(1):6–22, January 1984.

[MP95] Silvia M. Müller and Wolfgang J. Paul. *The complexity of simple computer architectures*, volume 995 of *Lecture Notes in Computer Science.* Springer-Verlag Inc., New York, NY, USA, 1995.

[MP00] Silvia M. Müller and Wolfgang J. Paul. Computer Architectures: Complexity and Correctness. Draft. EMail: {smueller,wjp}@cs.uni-sb.de, 2000.

[Mül97] Silvia M. Müller. Vorlesung Rechnerarchitektur II, WS 96/96, Universität des Saarlandes, FB. Informatik. 1997.

[PS98] Wolfgang J. Paul and Peter-Michael Seidel. On the complexity of Booth recoding. In *Proc. 3rd Conference on Real Numbers and Computers (RNC3)*, pages 199–218, 1998.

[SP88] James E. Smith and Andrew R. Pleszkun. Implementing precise interrupts in pipelined processors. In *IEEE Transactions on Computers*, volume C-37, pages 562–573. 1988.

[Tom67] Robert M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Developement*, volume 11 (1), pages 25–33. IBM, 1967.

[Weg87] Ingo Wegener. *The Complexity of Boolean Functions*. Stuttgart, 1987.

[WS84] Shlomo Weiss and James E. Smith. Instruction issue logic for pipelined supercomputers. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 110–118, Ann Arbor, Michigan, June 5–7, 1984. IEEE Computer Society and ACM SIGARCH.

[WS94] Shlomo Weiss and James E. Smith. *Power and PowerPC*. Morgan Kaufmann, San Francisco, 1994.

[Yeh93] Tse-Yu Yeh. *Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanism for High Performance Superscalar Processors*. PhD thesis, University of Michigan, USA, 1993.