Design and Evaluation of a RISC Processor with a Tomasulo Scheduler

Diplomarbeit

Lehrstuhl für Rechnerarchitektur
Prof. Wolfgang J. Paul
FB14 Informatik
Universität des Saarlandes

Daniel Kröning

Januar 1999

Hiermit erkläre ich an Eides statt, daß ich für die Anfertigung dieser Arbeit keine anderen als die angegebenen Quellen verwendet habe. Ich versichere, die Arbeit noch keinem anderen Prüfungsamt vorgelegt zu haben.

Saarbrücken, im Januar 1999

Daniel Kröning

Contents

Chapter 1

Introduction

The performance of today's microprocessors is astonishing. Beneath the progress in wafer technology, a big contribution to the improvements achieved in the past years was made by developing sophisticated scheduling algorithms. One of the major scheduling algorithms used in recent CPUs was specified long ago in 1967 by *Robert M. Tomasulo* [Tom67]. However, up to now concrete data on the impact of the algorithm on hardware cost and cycle time has been missing.

Thus, this thesis gives a detailed implementation of the Tomasulo scheduling algorithm for the DLX RISC architecture [HP96]. The design is based on a machine presented in [Lei98] and realizes full support for precise interrupts with a *reorder buffer* [SP88]. Cost and cycle time are calculated and evaluated with a formal model presented in [MP95]. The results are compared to other DLX implementations.

1.1 Results

The Tomasulo scheduling algorithm is one of the most competitive scheduling algorithms. It provides low CPI rates down to 1.1 which is shown by simulations on common benchmarks in [Ger98, Del98]. This thesis shows that adding a Tomasulo scheduler does not have any impact on the cycle time of the CPU design.

The Tomasulo scheduling algorithm with precise interrupts is known to be expensive regarding hardware cost. A complete CPU core design counts about 236,000 gate equivalents, which is about two times as much as is needed by a pipelined design with equal function units. Compared to the total costs of a CPU design (including the first level cache), this is just an increase of 26 percent at a 44 percent higher performance.

1.2 Outline

Chapter 2 describes some basic concepts, like the hardware model and gives a rough overview of the design. It also includes a terse introduction to the Tomasulo scheduling algorithm itself. Chapter 3 presents all the implementation details on gate level other than the memory system, which is presented separately in chapter 4. In chapter 5, the analysis of the cost and cycle time of the design is carried out. The results are compared to other DLX implementations in order to evaluate the overall design quality impact of different scheduling algorithms. Chapter 6 contains the correctness proof for the hardware.

Chapter 2

The Scheduling Algorithm

2.1 The DLX Architecture

The underlying CPU is an implementation of the DLX architecture citeHP96. That is a load/store architecture with support for integer and floating point instructions. It has three register files:

- The **general purpose register file** (GPR) consists of 32×32 integer registers (R_0,...,R_{31}), where R_0 is defined to be always zero. The general purpose registers are used for all integer operations and memory addressing purposes.
- The **floating point register file** (FPR) consists of 32×32 single precision floating point registers (FGR_0,...,FGR_{31}). These registers can also be accessed as 16×64 double precision floating point registers (FPR_0, FPR_2,...,FPR_{30}), well aligned accesses assumed. FPR_0 is mapped onto FGR_0 and FGR_1, and so on. The floating point registers are only used by FPU (floating point unit) instructions.
- The **special purpose register file** (SPR) consists of several registers needed for special purposes such as flags and masks. An example is the IEEE floating point flags register.

The DLX instruction set (appendix C) is a RISC instruction set and is similar to SUN's MIPS instruction set.

2.2 The Tomasulo Scheduling Algorithm

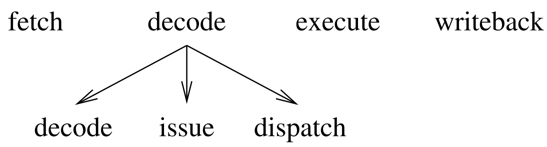
The following sections give a short summary of the Tomasulo scheduling algorithm. The algorithm was specified in 1967 by Robert M. Tomasulo for an IBM 360/91 [Tom67]. A more comprehensive description is also available in [Mül97a].

In its original form, the Tomasulo scheduling algorithm is limited to two-address-instructions (one source, one destination, e.g., R1+=R2) and multiple sequential function units for each kind of operation. However, it is easy to extend the algorithm to handle today's common instructions with three addresses (two source registers, one destination, e.g., R1:=R2+R3). The algorithm is widely used, e.g., by IBM PowerPC, Intel Pentium-Pro or AMD K5 [Mot97, CS95].

2.2.1 Pipelining vs. Out-of-Order Execution

Pipelining

There are many ways of implementing the execution of an instruction. In general, the execution of an instruction can be split into the following phases:



- Instruction fetch: The instruction is fetched from the instruction memory system into a special register.
- Instruction decode: During instruction decode the instruction is interpreted and passed to an execution unit. This phase can be split into three subparts: decode (instruction word interpretation), issue (passing the instruction and its operands to a function unit or to an instruction queue), and dispatch (passing the data for the actual execution). This terminology is not yet uniform; [HP96] states that issue and dispatch are sometimes used conversely.
- Execution: The actual calculation or data transfer is performed.
- Writeback: The result of the instruction is written into the register file.

Pipelined CPUs overlap the processing of different phases of different instructions. The first approach is to process the single phases of the instructions strictly in program order. Figure 2.1 illustrates this. Pipelining implies in-order execution, i.e., the execution of the subsequent instructions is also done strictly in program order.

However, in-order execution does not fully utilize all functional parts of a CPU. The rule of in-order execution prohibits that subsequent instructions overtake previous instructions. In figure 2.1, instruction I_2 blocks the execute stage for four cycles, since the division function unit has a long latency. Instruction I_3 has to be stalled upon the begin of its execution, since the execution stage is blocked by I_2 and since it requires the result of I_2 (*data dependence*).

Out-of-Order Execution

Data dependencies and different latencies of the function units can cause additional delays which reduce performance. In order to eliminate these delays, the rule of in-order execution of all instruction phases must be dropped. The result is an *out-of-order execution* algorithm. An out-of-order execution algorithm tries to increase performance by distributing the instructions among the available hardware components regardless their original order. There are two main requirements for such an algorithm:

- The algorithm must maintain data consistency.
- The algorithm is supposed to achieve a high utilization of the function units to reduce the delays.

Figure 2.2 depicts the execution of I_2 to I_4 on an out-of-order CPU. Instruction I_4 is now able to enter the execution stage even before I_3 does, since I_4 does not depend on any result of the preceding instructions. It even terminates before I_2, which causes a *write after write (WAW) data hazard* in R_1 [HP96].

Furthermore, I_3 tries to read R_1 before I_2 writes it. Thus, there is also a *read after write (RAW) data hazard.* Since I_4 writes R_1 before I_3 reads it, there is also a *write after read (WAR) hazard.*

There are several ways to resolve these hazards. In order to resolve RAW hazards, *result forwarding* is usually used. In the given example, the result of the division is forwarded to instruction I_3. The scheduling algorithm is supposed to stall the execution of an instruction until all operands are available.

One way to resolve WAW and WAR hazards is to skip the writeback of a result into a register if a subsequent instruction, which writes into the same register, already terminated. In the given example, the writeback of the result of instruction I_2 would have to be skipped. The result is forwarded to instruction I_3 instead. This is implemented by the Tomasulo scheduling algorithm in its original form.

Another way is to delay the result writeback until all previous instructions wrote their result into the register file, i.e., the writeback is performed in-order. This is implemented by the Tomasulo scheduling algorithm with reorder buffer used in this thesis.

2.2.2 Basics of the Tomasulo Scheduling Algorithm

The Tomasulo scheduling algorithm has several essential features:

- The Tomasulo scheduling algorithm has a distributed data structure, and requires only few global data.
- The algorithm allows data forwarding wherever possible.

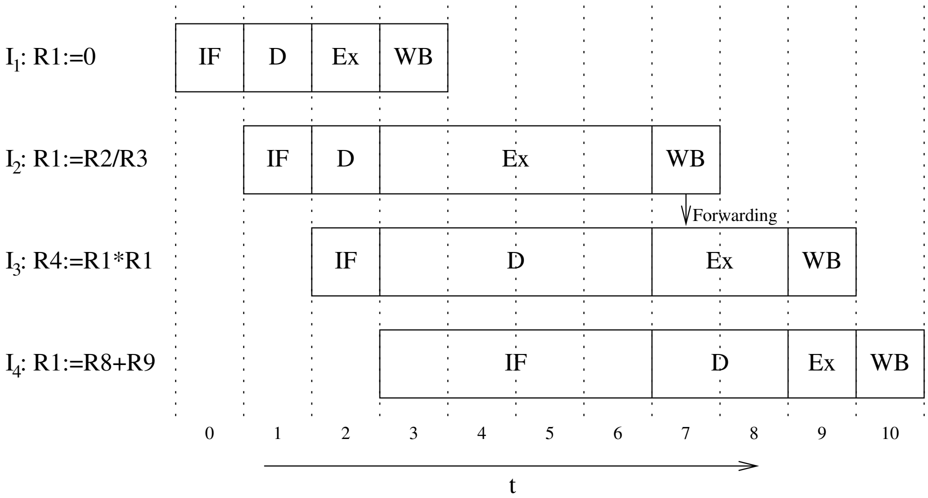


Figure 2.1: Pipelining example

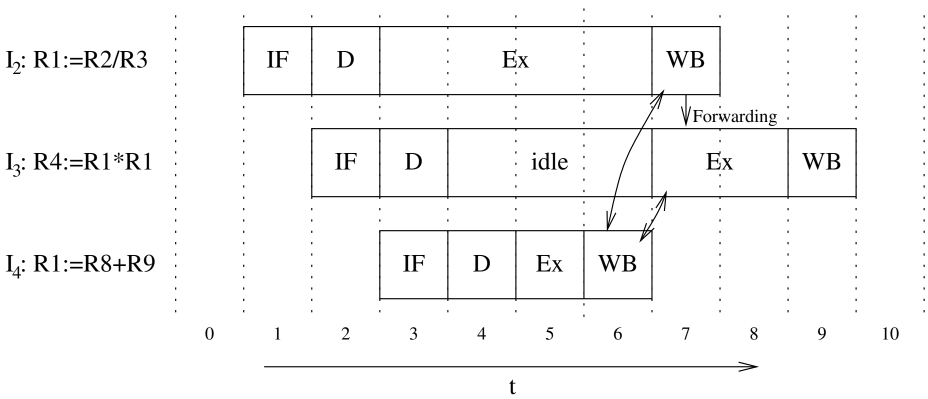
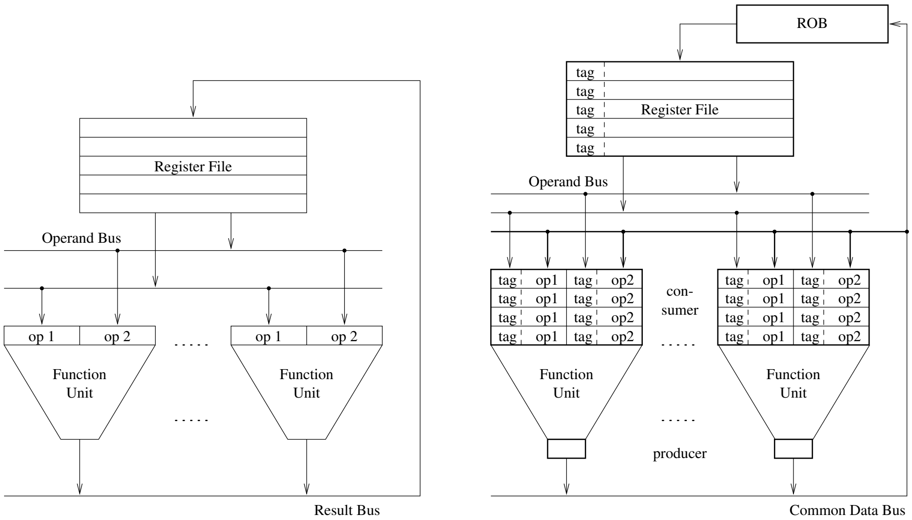


Figure 2.2: Out-of-order execution example. Instructions I_2 and I_4 cause a WAW hazard, instructions I_2 and I_3 cause a RAW hazard, and instructions I_3 and I_4 cause a WAR hazard.



Basic structure of an in-order design　　After adding Tomasulo scheduling

Figure 2.3: The basic data structures and data paths before and after adding Tomasulo scheduling

- The algorithm resolves WAW data hazards by inherent register renaming.
- The algorithm has support for function units with variable latency. This includes function units with variable latencies depending on the actual input data values.

Please note that the original Tomasulo algorithm uses out-of-order termination and thus does not support precise interrupts. In order to support precise interrupts, a *re-order buffer* (ROB) [SP88] is added to the machine described in this thesis. The re-order buffer implements in-order termination. This results in small modifications of the original scheduling algorithm. Thus, the following sections describe a modified scheduling algorithm presented in [Ger98] rather than the original Tomasulo scheduling algorithm. The complete protocol is presented in section 2.4, and its hardware implementation is presented in chapter 3.

2.2.3 Key Data Structures and Transfer Paths

Figure 2.3 gives an overview of the basic data paths of an in-order design and of the same design after adding Tomasulo scheduling with reorder buffer. The Tomasulo scheduling algorithm requires the following data structures and transfer paths:

Each register (named $\mathbf{R}_i$**.data**) is extended by a tag and a valid flag. This extension is called producer table. These additional data fields have the following purposes:

- $\mathbf{R}_i$**.valid**: The valid flag of a register is set iff the corresponding data item contains the valid value of the register.

op	tag	full	op1.valid	op1.tag	op1.data	op1.valid	op2.tag	op2.data

Figure 2.4: Reservation station data items

- **R_i.tag**: If the valid flag is not set, the tag data item of a register contains a tag for the instruction which produces the desired value.

Each function unit is extended by an instruction buffer to store instructions and operands until all operands and the function unit itself are available. These buffer entries are called **reservation stations**.

The reservation stations provide the operands for the function units. They are basically a queue for the issued instructions. Each reservation station RS_i holds exactly one instruction and its operands and has the following components (figure 2.4):

- The **RS_i.full** data item is set iff the entry is in use.
- The **RS_i.op** data item contains additional operation flags. This is, e.g., for an integer ALU, the concrete operation like addition, subtraction, shifting, etc.
- The **RS_i.tag** data item contains the ROB tag of the instruction in the reservation station. This item is an addition to the original Tomasulo algorithm.
- The **RS_i.op$_1$** and **RS_i.op$_2$** items hold the source operands of the instruction. They are a copy of the appropriate register file and producer table entries and have the same semantics.

The instructions are written into an appropriate reservation station during instruction issue. As soon as all operands of a given instruction in the queue (i.e., in a reservation station) are available, the instruction is ready to be dispatched into the actual function unit.

The result bus of the in-order design is replaced by the *common data bus* (CDB). During instruction dispatch, the instruction is passed to the function unit. On leaving the function unit, the CDB is requested for writing the result on the CDB. Functional units writing on the CDB are called *producers*. Units reading the CDB are called *consumers*. The reservation stations are the usual consumers. They watch the CDB for the operands they are missing (*bus snooping*). However, before a producer can write on the CDB, it has to request the CDB, since multiple producers might try to write on the CDB in the same cycle. These requests are handled by the CDB control, which acknowledges at most one request in the next cycle. [1]

[1] In the original Tomasulo design, even two cycles of lead time are required.

Name	Width	Purpose
valid	1	valid $= 1 \Leftrightarrow$ data field contains a valid value
data	64	result data
dest	4	address of the destination register

Table 2.1: Main components of a reorder buffer entry

2.3 The Reorder Buffer

In order to realize precise interrupts, the design in this thesis contains a **reorder buffer** (ROB). Precise interrupts are essential for today's microprocessors. An interrupt between instruction I_{i-1} and I_i is precise iff instructions I_1,...,I_{i-1} are completed before starting the ISR and later instructions (I_i,...) did not change the state of the machine [SP88, Mül97b].

On completion, the reorder buffer [SP88] gathers the results produced by the function units and sorts them by issue order, i.e., by program order. The results are written afterwards into the register file in issue order. However, before writing the result of instruction I_i, it is checked whether this instruction causes an interrupt or not. Thus, in case of an interrupt, the register file contains exactly all modifications made by instructions I_0 to I_{i-1}.

The reorder buffer is realized as circular FIFO queue with a head and a tail pointer. New instructions are put into the ROB entry pointed to by the tail pointer. This ROB address is also used as a tag to the result. This is in contrast to the original Tomasulo design, which uses tags associated with the reservation stations. Table 2.1 lists the main components of a reorder buffer entry. The ROB needs further extensions in order to support interrupts (chapter 3).

When an instruction completes, both the result and the exception flags are written into the reorder buffer entry pointed to by this reorder buffer tag. In each cycle, the entry at the head of the reorder buffer is tested. If it is valid (i.e., the instruction has completed), a check for exceptions is performed and the data is written into the register file. Depending on the type of the interrupt (abort/repeat/continue), the result of I_i is written into the register file before executing the interrupt service routine.

2.4 The Overall Scheduling Protocol

The following section presents the overall scheduling protocol, which is implemented in this thesis [Mül97a, Ger98, Del98]. The execution of an instruction I_i is split into six phases: fetch, issue, dispatch, execution, completion and writeback.

```
if (∃ free RS for Ii ∧ /ROB.full)                    ①
 {
  RS.op:=Ii;                                          ②  ⎫
  RS.full:=1;                                            ⎪
  RS.tag:=ROB.tail;                                      ⎪
  ∀ operands x of Ii                                  ③  ⎪
    if (Rx.A.valid)                                   ④  ⎪
      RS.opx.valid:=1;                                   ⎪
      RS.opx.data:=Rx.A.data;                            ⎪ initialize
    else if (CDB.tag=Rx.A.tag)                        ⑤  ⎬ reser-
      RS.opx.data:=CDB.data;                             ⎪ vation
      RS.opx.valid:=1;                                   ⎪ station
    else if (ROB[Rx.A.tag].valid)                     ⑥  ⎪
      RS.opx.data:=ROB[Rx.A.tag].data;                   ⎪
      RS.opx.valid:=1;                                   ⎪
    else                                              ⑦  ⎪
      RS.opx.tag:=Rx.A.tag;                              ⎪
      RS.opx.valid:=0;                                   ⎭

  if (Ii has a destination register y.A)              ⑧  ⎫
    Ry.A.tag:=ROB.tail;                                  ⎪ initialize
    ROB[ROB.tail].dest:=y.A;                             ⎬ ROB
  else                                                   ⎪ entry
    ROB[ROB.tail].dest:=0;                               ⎭

  ROB.tail++;                                         ⑨
 }
```

Figure 2.5: Issue protocol. The register address of operand x is denoted by $x.A$.

2.4.1 Issue

Let I_i be the instruction to be issued. For issue, it is essential that an appropriate reservation station and a ROB entry are available ① (figure 2.5). If so, the instruction is issued into this reservation station entry ②. For each operand of the instruction three sources have to be checked ③: The operand might be in the register file ④, on the CDB ⑤, or in the reorder buffer ⑥. If not, it is the destination of a preceding, incomplete instruction ⑦, and instead of the operand, the tag of this instruction is stored in the reservation station.

Simultaneously, the ROB entry is allocated and initialized for the instruction ⑧. If the instruction has a destination register, the address of this register is stored in the ROB entry and the pointer to the ROB entry is stored as tag in the producer table. After issue, the tail pointer is incremented ⑨.

```
if (∃ RS with                                    ①
    RS.opx.valid=1 ∀ operands x
    ∧ /FU.stall)
 {
  FU.op:=RS.op;                                  ② } pass in-
  FU.tag:=RS.tag;                                    } struction
  FU.opx:=RS.opx;                                    } to FU

  RS.full:=0;                                    ③
 }
```

Figure 2.6: Dispatch protocol

2.4.2 Dispatch

During instruction dispatch (figure 2.6), a valid instruction moves from a reservation station entry into the actual function unit. An instruction is valid iff all its operands are valid ①. Furthermore, the function unit must not be stalled, i.e., it must be ready to accept a new instruction. If more than one instruction for a certain function unit is valid, the scheduler has to choose one for dispatch. The correctness proof in chapter 6 relies on choosing the oldest among the valid instructions. This issue is discussed in chapter 3. If all these conditions hold, the instruction is passed to the function unit ② and the reservation station is freed ③.

In real hardware, RS.op$x$ can also be forwarded via CDB from a producer. In contrast to the forwarding during issue, this forwarding is just an optimization and not necessary for correctness. Thus, this protocol element is omitted here.

2.4.3 Completion

Before completion (figure 2.7), the reservation station requests the CDB. As soon as the reservation station gets an acknowledge ①, the result and the ROB tag are put on the CDB ②. The according reorder buffer entry is filled with the result and the valid bit is set ③.

2.4.4 Snooping on the CDB

On completion, the result of an operation is put on the CDB. Instructions in the reservation stations, which depend on this result, read the operand data from the CDB (figure 2.8). The reservation stations identify the results by the ROB tag.

```
if (FU has result ∧
    got CDB-acknowledge)                    ①
 {
  CDB.data:=FU.result;                      ②
  CDB.tag:=FU.tag;

  ROB[CDB.tag].valid:=1;                    ③
  ROB[CDB.tag].data:=CDB.data;
 }
```

Figure 2.7: Completion protocol

```
∀ operands x
  if (RS.full ∧
      /RS.opx.valid ∧
      (RS.opx.tag=CDB.tag))
   {
    RSe.opx.valid:=1;
    RSe.opx.data:=CDB.data;
   }
```

Figure 2.8: CDB snooping protocol

2.4.5 Retirement / Writeback and Interrupts

During retirement (figure 2.9), a result of an instruction in the ROB is written into the register file ③, if no interrupt of type abort or repeat is pending ②.

At the same time, the result flags are checked ④. Almost all result flags are masked with the SR registers prior this check. If an error occurred while processing the instruction, the interrupt service routine is started. Section 3 contains more details of the interrupt mechanism.

2.5 Overall Scheduling Example

Figure 2.10 contains an example of Tomasulo scheduling with reorder buffer, considering the following piece of code:

I_1: `R3:=M[R10]`
I_2: `R1:=R2+R3`

For this example, M[R10] contains the value 11 and R2 contains 9. In cycle t=0, the first instruction is already in the execution phase. It is executed by the memory unit

```
if (ROB is not empty ∧                              1
     ROB[ROB.head].valid)
 {
  if (((ROB[ROB.head].cause ∧ SR) = 0)
       ∨ interrupt is of type continue)             2
   {
     x:=ROB[ROB.head].dest;
     Rx.data:=ROB[ROB.head].data;                   3   write
     if (ROB.head=Rx.tag) Rx.valid=1;                   back
   }

  if ((ROB[ROB.head].cause ∧ SR) ≠ 0)               4
    Perform Interrupt
  else
    ROB.head++;
 }
```

Figure 2.9: Retirement / writeback protocol

and stored in reorder buffer entry 0. Furthermore, in cycle t=0 the second instruction is fetched.

In cycle t=1, this instruction is decoded and issued into a ALU reservation station. The ALU reservation is assumed to have only one reservation station. The reorder buffer entry 1 is also filled with this instruction.

In cycle t=2, the load instruction is one cycle ahead of completion. Thus, the memory reservation station requests the CDB for the next cycle.

In cycle t=3, this request is acknowledged by the CDB control. The result of the load operation (11) is put on the CDB. This makes the second operand of the ALU reservation station valid. Since both operands are now valid, the instruction is dispatched into the ALU in the same cycle. Furthermore, the ALU requests the CDB for the next cycle. In the same cycle, the result of the load instruction on the CDB is written into the reorder buffer entry 0, which becomes valid.

In cycle t=4, the result of the load instruction is written from the reorder buffer entry 0 into the register file. R3 becomes valid by this. In the same cycle, the CDB control acknowledges the CDB request by the ALU. The result of the addition is put on the CDB and reorder buffer entry 1 becomes valid.

In cycle t=5, this result is finally written into the register file.

t	ALU reservation station for I_2									register file								
	global			operand 1			operand 2			R1			R2			R3		
	op	tag	full	tag	valid	data	tag	valid	data	tag	valid	data	tag	valid	data	tag	valid	data
0	-	-	0	-	-	-	-	-	-	-	1	0	-	1	9	ROB-0	0	-
1	+	ROB-1	1	-	1	9	ROB-0	0	-	ROB-1	0	-	-	1	9	ROB-0	0	-
2	+	ROB-1	1	-	1	9	ROB-0	0	-	ROB-1	0	-	-	1	9	ROB-0	0	-
3	+	ROB-1	1	-	1	9	ROB-0	1	11	ROB-1	0	-	-	1	9	ROB-0	0	-
4	-	-	0	-	-	-	-	-	-	ROB-1	0	-	-	1	9	ROB-0	0	-
5	-	-	0	-	-	-	-	-	-	ROB-1	0	-	-	1	9	-	1	11
6	-	-	0	-	-	-	-	-	-	-	1	20	-	1	9	-	1	11

t	reorder buffer										common data bus				
	global		entry 0				entry 1								
	ROB.head	ROB.tail	valid	data	dest		valid	data	dest		req	ack	tag	valid	data
0	ROB-0	ROB-1	0	-	gpr	R3	-	-	-	-	-	-	-	0	-
1	ROB-0	ROB-2	0	-	gpr	R3	0	-	gpr	R1	-	-	-	0	-
2	ROB-0	ROB-2	0	-	gpr	R3	0	-	gpr	R1	Mem	-	-	0	-
3	ROB-0	ROB-2	0	-	gpr	R3	0	-	gpr	R1	ALU	Mem	ROB-0	1	11
4	ROB-0	ROB-2	1	11	gpr	R3	0	-	gpr	R1	-	ALU	ROB-1	1	20
5	ROB-1	ROB-2	-	-	-	-	1	20	gpr	R1	-	-	-	0	-
6	ROB-2	ROB-2	-	-	-	-	-	-	-	-	-	-	-	0	-

Figure 2.10: Scheduling example

Chapter 3

Tomasulo Hardware

3.1 Overview

3.1.1 Gates, Circuits, Cost and Delay

The hardware model used in this thesis is presented in [MP95]. The following sections just give a really short overview.

For calculation of cost and delay the methods and formulae presented in [MP95] will be used. In particular, the overall calculation is also done by transforming all the complex formulae into a C-program, which is discussed in chapter 5. Thus, cost and delay formulae are omitted in the following chapters.

Figure 3.1 lists the symbols of the basic gates used in the designs. In addition, the following basic circuits are used: n-bit adder / incrementer, n-bit multiplexer, tristate driver, n-bit register, n-bit decoder / encoder, n-bit zero tester, the generic parallel prefix circuit, RAM, shifter, and ALU. A detailed description and the cost and delay formulae can be found in [MP95].

Furthermore, the hardwired control described in this chapter requires two additional basic circuits: the n-bit find first one circuit (FFO) and the find last one circuit (FLO). They calculate the following functions:

ffo: $\{0,1\}^n \rightarrow \{0,1\}^{n+1}$, $(a_{n-1},...,a_0) \mapsto (b_{n-1},...,b_0,\text{zero})$, such that

$$b_i = \begin{cases} 1 & : \quad i = \min\{j|a_j = 1 \wedge j \in \{0,...,n-1\}\} \\ 0 & : \quad \text{otherwise} \end{cases}$$

$$\text{zero} = \begin{cases} 1 & : \quad a_i = 0 \text{ for all } i \in \{0,...,n-1\} \\ 0 & : \quad \text{otherwise} \end{cases}$$

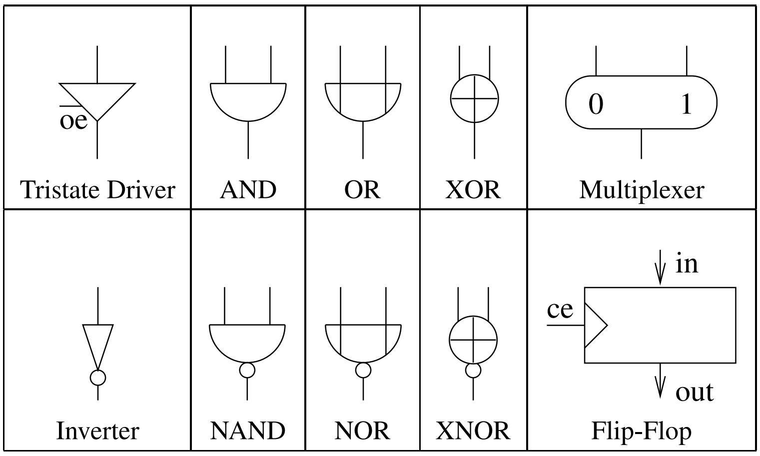


Figure 3.1: Symbols of the basic gates

flo: $\{0,1\}^n \rightarrow \{0,1\}^{n+1}$, $(a_{n-1},...,a_0) \mapsto (b_{n-1},...,b_0,\text{zero})$, such that

$$b_i = \begin{cases} 1 & : \quad i = \max\{j | a_j = 1 \wedge j \in \{0,...,n-1\}\} \\ 0 & : \quad \text{otherwise} \end{cases}$$

$$\text{zero} = \begin{cases} 1 & : \quad a_i = 0 \text{ for all } i \in \{0,...,n-1\} \\ 0 & : \quad \text{otherwise} \end{cases}$$

A recursive construction of the circuits and the cost and delay formulae are given in appendix A.1.

3.1.2 The Pipeline Stages

In this chapter, the complete hardware of a DLX RISC core with Tomasulo scheduling is presented. Chapter 4 extends the design with an interface to main memory for load/store operations.

The design is based on the DLX implementations published in [MP95, MP98, Lei98]. It basically consists of a five stage pipeline. The first stage (IF) performs the instruction fetch. In the second stage (D/I), the fetched instruction word is decoded and passed into an appropriate reservation station. The third stage (EX) contains the actual function units, which execute the instruction. Fast function units (i.e., one cycle latency) combine execution and dispatch in one cycle. For slow function units, the execute phase might take several cycles. In the fourth stage (completion), the result of the instruction is stored in the reorder buffer. The fifth stage (WB) performs the writeback of the result into the register file.

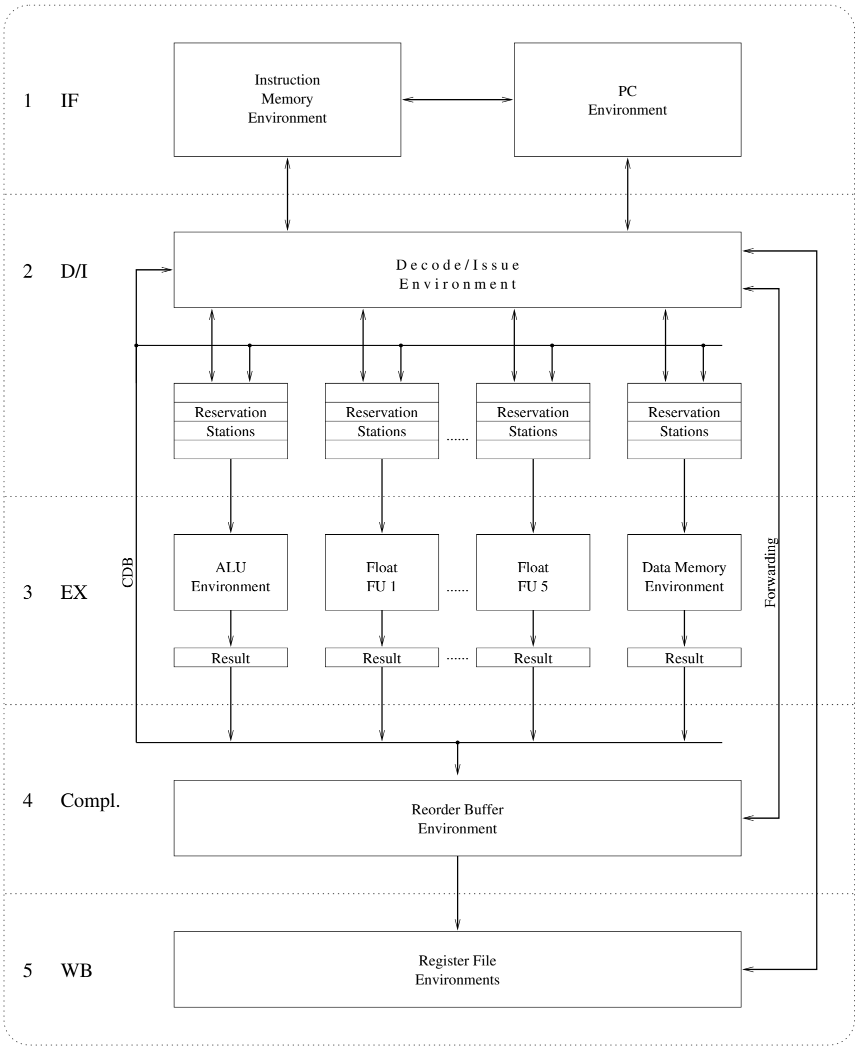


Figure 3.2: Overview of the data paths

3.1.3 Environments

The CPU consists of environments. Figure 3.2 gives an overview of the data paths and the interconnection of their envorinments.

- The **PC environment** contains the PCs of stage 0 and 1 and performs PC calculations.
- The **instruction memory environment** performs the actual instruction fetch and is the interface to the instruction memory or cache.
- The **IR environment** contains the instruction register of the decode/issue stage (IR1).

- The **decode/issue environment** decodes the fetched instructions and distributes the instructions among the function units. It also contains the main control automaton.
- Each function unit, including the data memory environment, has its own set of **reservation stations** assigned to it. Each set has an independent control circuit. The reservation station environments belong to the decode/issue stage.
- The **function unit environments** contain the function units, e.g., the ALU, the floating point units, and the data memory interface.
- The **CDB control environment** allocates the CDB to the reservation stations.
- The **reorder buffer environment** contains the reorder buffer and its control circuit. It also contains large parts of the interrupt handling circuitry and belongs to the completion stage.
- The **register file environment** holds the register files and belongs to the write-back stage.

A detailed description of the individual environments follows.

3.2 The PC Environment

The PC environment (figure 3.3) contains the program counter PC. It is almost identical with the PC environment found in [Lei98]. The PC register of stage 0 PC0 is used for the instruction fetch (section 3.3). After the instruction fetch, the value of this register is saved in oPC1. Furthermore, the PC environment calculates the new value of the PC register. This is done in dependence of several control signals, which are generated by the main control (section 3.5):

Usually, the new value of the PC0 register is the old value incremented by four. In case of a branch, rfe, jump or an interrupt, the PC register has to be clocked with another value. In these cases, the setPC signal is set. The signal is calculated as follows:

$$\text{setPC} = \text{JISRrfe} \vee \text{op.branch} \vee \text{op.jump} \vee \text{op.jumpR}$$

The op.branch, op.jump and op.jumpR are active in case of a branch/jump instruction and are calculated by the decode/issue environment. In case of a branch or jump instruction (jumpR=0), co1 (the immediate constant) is the target offset. In case of a jump register instruction (jumpR=1), the signal op1.l.data (low part of the first operand) is the new PC. The operands are provided by the decode/issue environment.

The JISRrfe signal is set iff the cause for the active setPC signal is an interrupt or a rfe instruction. Interrupts are indicated by the JISR signal, which is calculated in the

ROB environment. In case of an interrupt, the address of the interrupt service routine (SISR) is clocked into the PC0 register.

The processing of rfe instructions affects two cycles. In the first cycle after a rfe instruction, the value of the EPC special purpose register is used as address for the instruction fetch. In this cycle, the DOrfe signal is active. This signal is provided by the decode/issue environment. In the second cycle after a rfe instruction, the value of the EPCn register is used. In this cycle, the rfeEPCn signal is active. The signal is the DOrfe signal, which is delayed by one cycle with a register (figure 3.4). Thus, the JISRrfe signal is calculated as follows:

$$\text{JISRrfe} = \text{JISR} \vee \text{DOrfe} \vee \text{rfeEPCn}$$

During an issue stall (issuestall=1), all clock enable signals are disabled in order to prevent modifications of the PC registers.

3.3 Instruction Memory Environment

The instruction memory environment (figure 3.5) performs the actual instruction fetch and is the interface to the instruction memory or first level instruction cache. The instruction memory environment fetches the instruction word pointed to by the signal pc0, which is provided by the PC environment. This instruction word is returned as signal ir0. Ibusy and pff0 are signals generated by the instruction memory. Ibusy is active iff the instruction memory is temporary unable to return the requested value, e.g., because of a cache miss. The pff0 signal is used to implement virtual memory and indicates a page fault. The instruction memory system only supports word aligned memory accesses. In case of a misaligned access, the imal0 signal is active. It is calculated as follows:

$$\text{imal0} = \text{pc0}[0] \vee \text{pc0}[1]$$

In case of an interrupt (JISR=1), in case of a misaligned instruction word, or if the instruction memory system is unable to return the requested instruction word, zero is returned instead. This is the opcode for a left shift of register R_0 over 0 bits, thus, it is a NOP instruction.

3.4 Instruction Register Environment

The instruction register (IR) environment (figure 3.6) contains the instruction register IR1, the page fault register PFF1, and the instruction misaligned register IMAL1 of the decode/issue stage. The instruction register holds the instruction word fetched by the PC environment.

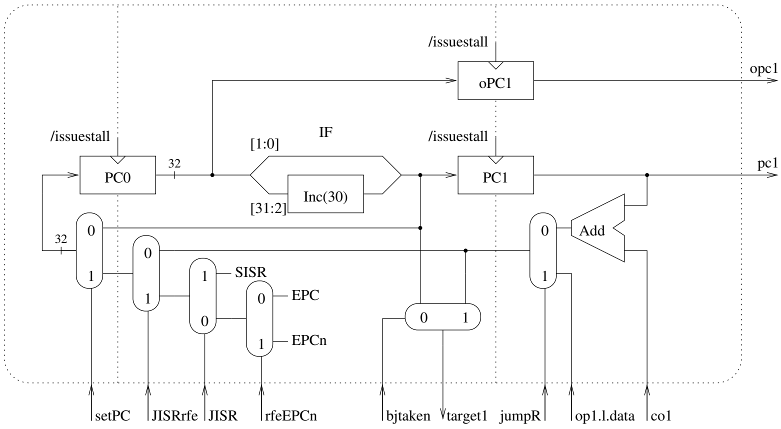


Figure 3.3: PC environment

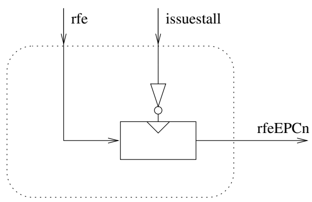


Figure 3.4: rfeEPCn register

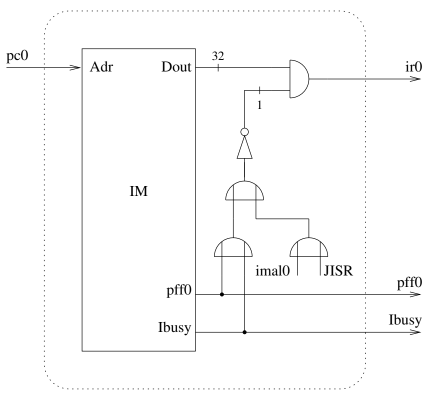


Figure 3.5: Instruction memory environment

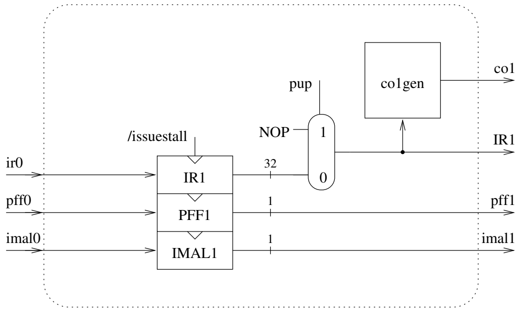


Figure 3.6: Instruction register environment

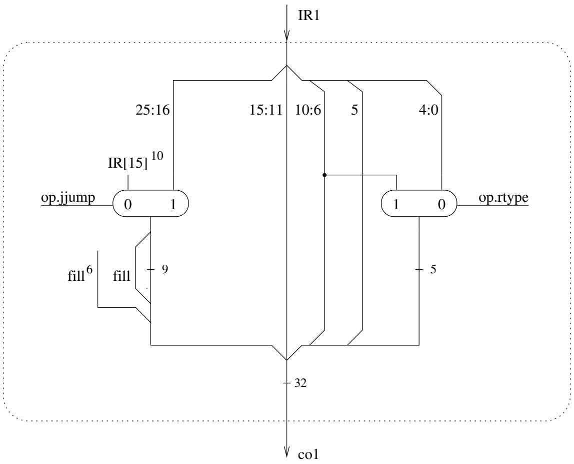


Figure 3.7: Immediate constant generation co1gen

For some instructions, the instruction word provides an immediate constant. The instruction register environment contains the co1gen circuit (figure 3.7), which extracts the immediate constant from the instruction in IR1 and performs a sign extension to 32 bits (signal co1). It is literally taken from [MP95]. The op.jjump and op.rtype signals are generated by the decode/issue environment and are used to determine the width of the constant and the position in the instruction word.

3.5 Decode/Issue Environment

The decode/issue environment serves two purposes: It decodes the instruction word in IR1 and it distributes the instructions and their operands among the reservation stations. A DLX floating point instruction can have up to four source operands (two registers, IEEEf and the interrupt mask), therefore, four operand busses (op1 to op4) originate in the decode/issue environment.

3.5.1 Decoding the Instruction Word

The decoding of the instruction word is done by the opgen (operation generation) circuit. The circuit opgen generates several control signals from the instruction word found in IR1 with a control automaton. This environment is almost literally taken from [Lei98]. A figure is omitted therefore.

The automaton has two parts, ID1 and ID2. The first part, ID1, has an automaton state assigned to each instruction. Table 3.1 contains the states and the monomials which are used to compute the new state. The state is never stored in any register, it is just used to compute the active control signals. Table 3.1 also lists the control signals which are active in a given state.

The itype, rtype, and jtype control signals correspond to instruction word formats of the same denominator (appendix C), iuFOP indicates an unimplemented floating point instruction, ill indicates an illegal instruction word. The signals fp and db specify whether floating point (fp=1) or double precision (db=1) values are involved. The signals with names beginning with "FU." indicate the function unit which is required to process the instruction. All other signals specify the action to be performed by this function unit.

Depending on its state, ID1 generates further control signals, which are used to select the correct source and destination operands (table 3.2). For example, op1.RS1 indicates that the register addressed by RS1 is expected on operand bus one. RS1, FS1, RS2, FS2, RD, FD and SA correspond to bit fields in the instruction word, which contain the desired register address (appendix C). R31, FCC, RM and MASK are constant register addresses (table 3.15 contains the coding of the special purpose register addresses). The op2.imm signal is active iff the immediate constant in the instruction word is operand two.

The second part of the automaton, ID2, is only used by branches. In case of a conditional branch, it computes the bjtaken signal, which is active iff the branch is to be taken. If the instruction is not a branch, the signal is undefined. The ID2 automaton requires two input signals: AEQZ and FCCEQZ. AEQZ is active iff the source operand of a conditional branch is zero. FCCEQZ is active iff the FCC bit is zero. Both FCC and the beqz/bnez operand are on the low part of operand bus op1 (table 3.2). Thus, op1.l.data is tested:

$$\begin{aligned} \text{AEQZ} &= \text{zero(op1.l.data[31:0])} \\ \text{FCCEQZ} &= \overline{\text{op1.l.data[0]}} \end{aligned}$$

3.5.2 Function Unit Availability Test

As mentioned above, the control automaton ID1 determines which function unit is required to process the instruction in IR1. Table 3.3 lists all function units with their

	Target state	Active control signals	Monomials IR[31:26]	IR[6]	IR[5:0]
ID1	ALU	rtype, FU.alu	000000 000000	* *	0001** 10****
	Shifti	rtype, FU.alu	000000	*	0000**
	ALUi	itype, FU.alu	0*1***	*	******
	Load	itype, load, FU.mem	100***	*	******
	Load.s	itype, load, fp, FU.mem	110001	*	******
	Load.d	itype, load, fp, db, FU.mem	110101	*	******
	Store	itype, store, FU.mem	101***	*	******
	Store.s	itype, store, fp, FU.mem	111001	*	******
	Store.d	itype, store, fp, db, FU.mem	111101	*	******
	Faddsub.s	rtype, faddsub, fp, FU.fadd	010001	0	00000*
	Faddsub.d	rtype, faddsub, fp, db, FU.fadd	010001	1	00000*
	Fmul.s	rtype, fmul, fp, FU.fmul	010001	0	000010
	Fmul.d	rtype, fmul, fp, db, FU.fmul	010001	1	000010
	Fdiv.s	rtype, fdiv, fp, FU.fdiv	010001	0	000011
	Fdiv.d	rtype, fdiv, fp, db, FU.fdiv	010001	1	000011
	Fcond.s	rtype, fcc, fp, FU.ftest	010001	0	11****
	Fcond.d	rtype, fcc, fp, FU.ftest	010001	1	11****
	Fabsneg.s	rtype, fabsneg, fp, FU.fconv	010001	0	00010*
	Fabsneg.d	rtype, fabsneg, fp, db, FU.fconv	010001	1	00010*
	Ff2i	rtype, ff2i, fp, FU.fconv	010001	*	001001
	Fi2f	rtype, fi2f, fp, FU.fconv	010001	*	001010
	FMov.s	rtype, fmov, fp, FU.fconv	010001	0	001000
	FMov.d	rtype, fmov, fp, db, FU.fconv	010001	1	001000
	FConv.s	rtype, fconv, fp, FU.fconv	010001	*	100*00
	FConv.d	rtype, fconv, fp, db, FU.fconv	010001	*	100001
	Branch	itype, bjjr, branch, noFU	00010*	*	******
	FBranch	itype, bjjr, branch, fp, noFU	00011*	*	******
	JumpReg	itype, bjjr, bjtaken, jumpR, noFU	010110	*	******
	Jump&LinkReg	itype, jalr, bjtaken, jumpR, noFU	010111	*	******
	Jump	jtype, bjjr, bjtaken, jump, noFU	000010	*	******
	Jump&Link	jtype, jalr, bjtaken, jump, noFU	000011	*	******
	Trap	jtype, trap, noFU	111110	*	000000
	RFE	jtype, rfe, noFU	111111	*	******
	Movs2i	rtype, movs2i, FU.alu	000000	*	010000
	Movi2s	rtype, movi2s, FU.alu	000000	*	010001
	FUnimp	iuFOP, noFU	010001 010001	* *	00011* 01****
	Illegal (z_0)	ill, noFU	-		
ID2	Taken	bjtaken	AEQZ · /IR1[26] /AEQZ · IR1[26] FCCEQZ · /IR1[26] /FCCEQZ · IR1[26]		
	Untaken		/taken		

Table 3.1: States, active control signals and DNFs

State	Instructions	op1.	op2.	op3.	op4.	dest.
ALU	add, sub, test/set, shift	RS1	RS2	-	-	RD
ALUi	addi, subi, test/set immediate	RS1	imm	-	-	RD
Shifti	shift with shift amount	RS1	imm	-	-	RD
Load	load GPR	RS1	-	-	-	RD
Load.s	load single precision FPR	RS1	-	-	-	FD
Load.d	load double precision FPR	RS1	-	-	-	FD
Store	store GPR	RS1	RD	-	-	-
Store.s	store single precision FPR	RS1	FD	-	-	-
Store.d	store double precision FPR	RS1	FD	-	-	-
Faddsub.s	fadd.s, fsub.s	FS1	FS2	RM	MASK	FD
Faddsub.d	fadd.s, fsub.s	FS1	FS2	RM	MASK	FD
Fmul.s	fmul.s	FS1	FS2	RM	MASK	FD
Fmul.d	fmul.d	FS1	FS2	RM	MASK	FD
Fdiv.s	fdiv.s	FS1	FS2	RM	MASK	FD
Fdiv.d	fdiv.d	FS1	FS2	RM	MASK	FD
Fcond.s	fc.cond.s	FS1	FS2	-	MASK	FCC
Fcond.d	fc.cond.d	FS1	FS2	-	MASK	FCC
Fabsneg.s	fabs.s, fneg.s	FS1	-	-	-	FD
Fabsneg.d	fabs.d, fneg.d	FS1	-	-	-	FD
Ff2i	mf2i	FS1	-	-	-	RS2
Fi2f	mi2f	RS2	-	-	-	FS1
FMov.s	mov.s	FS1	-	-	-	FD
FMov.d	mov.d	FS1	-	-	-	FD
FConv.s	cvt.s.d, cvt.s.i, cvt.i.s, cvt.i.d	FS1	-	-	-	FD
FConv.d	cvt.d.i, cvt.d.s	FS1	-	-	-	FD
Branch	beqz, bnez	RS1	-	-	-	-
FBranch	fbeqz, fbnez	FCC	-	-	-	-
JumpReg	jr	RS1	-	-	-	-
Jump&LinkReg	jalr	RS1	-	-	-	R31
Jump	j	-	-	-	-	-
Jump&Link	jal	-	-	-	-	R31
Trap	trap	-	-	-	-	-
RFE	rfe	-	-	-	-	-
Movs2i	movs2i	SA	-	-	-	RD
Movi2s	movi2s	RS1	-	-	-	SA

Table 3.2: Operands and bus use

FU	Purpose
FU[0] = FU.alu	integer instructions, movi2s, movs2i
FU[1] = FU.mem	load, store
FU[2] = FU.fadd	floating point addition and substraction
FU[3] = FU.fmul	floating point multiplication
FU[4] = FU.fdiv	floating point division
FU[5] = FU.fconv	conversion floating point / integer
FU[6] = FU.ftest	floating point condition tests

Table 3.3: Coding of the function units

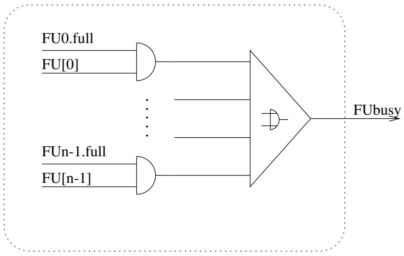


Figure 3.8: Function unit availability test FUtest

purpose and the control signals used to identify them. For each function unit, a single FU[i] is defined in order to simplify notation.

The FUtest circuit of figure 3.8 tests whether this function unit is available. This is done as follows (n denotes the number of function units):

$$\text{FUbusy} = \bigvee_{i=0}^{n-1} (\text{FU}_i.\text{full} \wedge \text{FU}[i])$$

The FU_i.full signals are generated by the reservation station controls of the function units. FU_i.full is active iff the reservation stations of the corresponding function unit are not able to accept an instruction.

The signal set D.FU_i.issue specifies the function unit which is actually used for issue. These signals are disabled in case of an issue stall, which is indicated by the issuestall signal.

$$\text{D.FU}_i.\text{issue} = \text{FU}[i] \wedge \overline{\text{issuestall}} \qquad i \in \{0, ..., n-1\}$$

3.5.3 Operand Address Generation Agen

The decode/issue environment also provides the operands of the instruction. For the source operands, the values are provided, if available. If they are not available, the

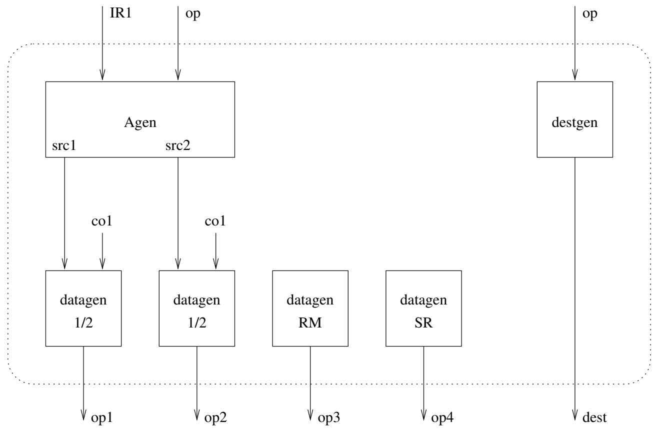


Figure 3.9: Generation of the operands

decode/issue environment provides the appropriate instruction tag to the reservation stations. This is done by the Agen and datagen circuits. For the destination operand, the type and address is determined. This is done by the destgen circuit. Figure 3.9 gives an overview of these circuits.

The operand address generation circuit Agen (figure 3.10) calculates the types and addresses of the source registers. For each operand, the operation generation environment opgen provides signals, which point to bit fields in the instruction word. In turn, these bit fields contain the register addresses of the operands. The type of an operand is represented by five signals:

- The signals op*i*.fpr, op*i*.gpr, op*i*.spr denote the register file which holds the operand, i.e., the floating point, general purpose, and special purpose register file.
- The signal op*i*.db indicates a double precision floating point register.
- The signal op*i*.imm is set iff the operand is the immediate constant.

The amount of different operand types is limited for certain operand busses (table 3.2). Operand bus op1 is used for the registers pointed to by RS1 / FS1 (they share the same bit field), RS2 (for mi2f) and SA (for movs2i). Furthermore, it is supposed to provide the value of the FCC special purpose register to process the fbeqz and fbnez instructions. The immediate constant is never on operand bus op1.

The second operand bus op2 has to provide the registers pointed to by RS2 / RD / FS2 / FD (they share the same bit field). Furthermore, it is supposed to provide the immediate constant for ALU operations.

The operand bus op3 is only used for the rounding mode RM, which is required for many floating point instructions. Operand bus op4 is only used for the interrupt

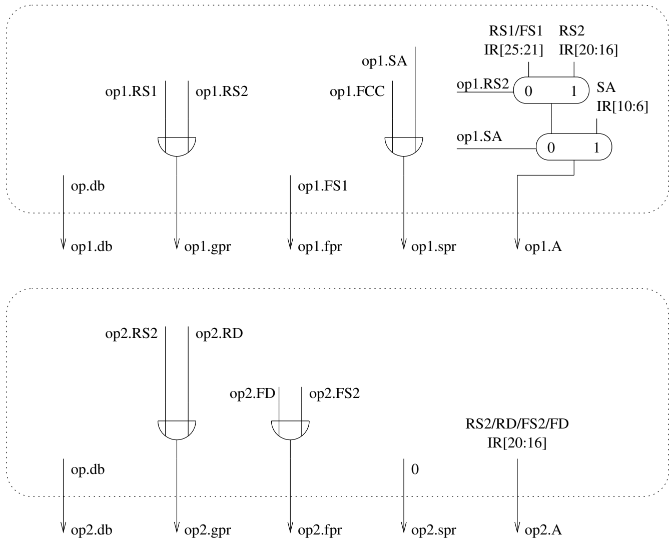


Figure 3.10: Operand address generation Agen

mask, which is also required for floating point instructions. For op3 and op4, no Agen circuit is necessary, since they are always used for the same register.

3.5.4 Operand Data Generation datagen

The operand data generation circuits generate the source operands from the addresses and types provided by the Agen circuit. These operands are distributed by four global data paths (table 3.4). The operand busses transport the operands to the reservation stations. Each operand (op1.l, op2.h, op2.l, op2.h, op3.l, op4.l) consists of three components, which are the tag, the valid bit and the operand data (table 3.5), ϑ denotes the tag width in bits (section 3.9). Each operand bus has a datagen environment of its own. The environments for op1 and op2 are identical (figure 3.11). The operands three and four do not have a high part and are only used for two fixed special purpose registers. Thus, they have a special datagen environment (figure 3.12) in order to save hardware cost.

The operand data generation environment datagen for op1 and op2 generates one operand according to the signals generated by the operand address generation environment Agen. The low and the high part of the operand are calculated separately, since each part might come from a different source. For each operand and for each part, one of the following cases applies: it is the immediate constant, it is in the register file, it is a result currently on the CDB, it is in the ROB, or it is the result of an instruction which has not yet completed (figure 2.5). Thus, four cascaded multiplexers are used to select the data from the appropriate source.

Bus	Items	Width	Purpose
op1	l	$\vartheta+32+1$	low part of the first operand
	h	$\vartheta+32+1$	high part of the first operand
	high	1	lowest bit of the register address
op2	l	$\vartheta+32+1$	low part of the first operand
	h	$\vartheta+32+1$	high part of the first operand
	high	1	lowest bit of the register address
op3	l	$\vartheta+32+1$	third operand (always integer)
op4	l	$\vartheta+32+1$	fourth operand (always integer)

Table 3.4: Components of the global data paths

Item	Width	Purpose
tag	ϑ	ROB tag of the instruction producing the operand
valid	1	valid $=1 \Leftrightarrow$ operand contains valid data
data	32	actual operand data

Table 3.5: Components of an operand

Operand is the Immediate Constant

The first step is checking whether the operand is the immediate constant (op$i$.imm=1) or not. If so, the low part of the operand is returned as follows:

$$\begin{aligned} \text{tag} &= 0^{\vartheta} \\ \text{valid} &= 1 \\ \text{data} &= \text{co1} \end{aligned}$$

In this case, the operand is valid already during issue. The data value is generated by the co1gen circuit (section 3.4). The high part of the operand can never be the immediate constant, thus, it is assumed to be zero in this case to have a defined value on the bus. The high part is also set to zero if the operand is not a double precision floating point value.

Operand is in the Register File

If the operand is not the immediate constant, it must be a register. Thus, the second step to get the operand is looking up its valid bit in the producer table. If the valid bit is set, the operand is in the register file. The operand address generation environment provides the necessary address signals op$i$.A, op$i$.fpr, op$i$.gpr, and op$i$.spr to the register files and to the producer tables, which return the requested values as op$i$.l/h.RF.Dout (register file) and op$i$.l/h.Prod.Dout (producer table), $i \in \{1,\ldots,2\}$. The registers RM and SR for operand three and four are directly provided by the SPR environment as SPR.RM and SPR.SR.

If the operand part is in the register file (op$i$.l/h.Prod.Dout.valid=1), the operand bus is set to the following values:

$$\begin{array}{rcl} \text{tag} & = & 0^{\vartheta} \\ \text{valid} & = & 1 \\ \text{data} & = & \left\{ \begin{array}{lcl} \text{R[A].data[31:0]} & : & \text{low part} \\ \text{R[A].data[63:32]} & : & \text{high part} \end{array} \right. \end{array}$$

Operand is on the CDB

If not so (op$i$.l/h.Prod.Dout.valid=0), the producer table contains the tag of the instruction which produces the desired value. Since this value might be on the CDB in the current cycle, the tag retrieved from the producer table is compared with the tag on the CDB. If both tags are equal and if the valid bit of the CDB is active, the operand is forwarded from the CDB:

$$\begin{array}{rcl} \text{tag} & = & 0^{\vartheta} \\ \text{valid} & = & 1 \\ \text{data} & = & \left\{ \begin{array}{lcl} \text{CDB.data[31:0]} & : & \text{low part} \\ \text{CDB.data[63:32]} & : & \text{high part} \end{array} \right. \end{array}$$

Operand is in the Reorder Buffer

The operand might also be in the reorder buffer. The tag found in the producer table is already the proper index for the ROB to check whether the result is already in the ROB. If so, the valid bit of the ROB entry is set. For this task, ports one to six of the ROB are used. Ports one and two are for op1.l and op1.h, ports three and four are for op2.l and op2.h, and ports five and six are for op3 and op4.

$$\begin{array}{rcl} \text{tag} & = & 0^{\vartheta} \\ \text{valid} & = & 1 \\ \text{data} & = & \left\{ \begin{array}{lcl} \text{op}i\text{.l.ROB1.Dout.data[31:0]} & : & \text{low part} \\ \text{op}i\text{.h.ROB1.Dout.data[63:32]} & : & \text{high part} \end{array} \right. \end{array}$$

Operand is a Result of an Uncomplete Instruction

If none of the cases above applies, the operand must be a result of an uncomplete instruction. The tag of this instruction can be found in the producer table. In this case, the operand is not yet valid and the tag is turned over to the reservation station. The data signal is set to zero in order to have a defined value on the bus.

$$\begin{array}{rcl} \text{tag} & = & \left\{ \begin{array}{lcl} \text{op}i\text{.l.Prod.Dout.tag} & : & \text{low part} \\ \text{op}i\text{.h.Prod.Dout.tag} & : & \text{high part} \end{array} \right. \\ \text{valid} & = & 0 \\ \text{data} & = & 0^{32} \end{array}$$

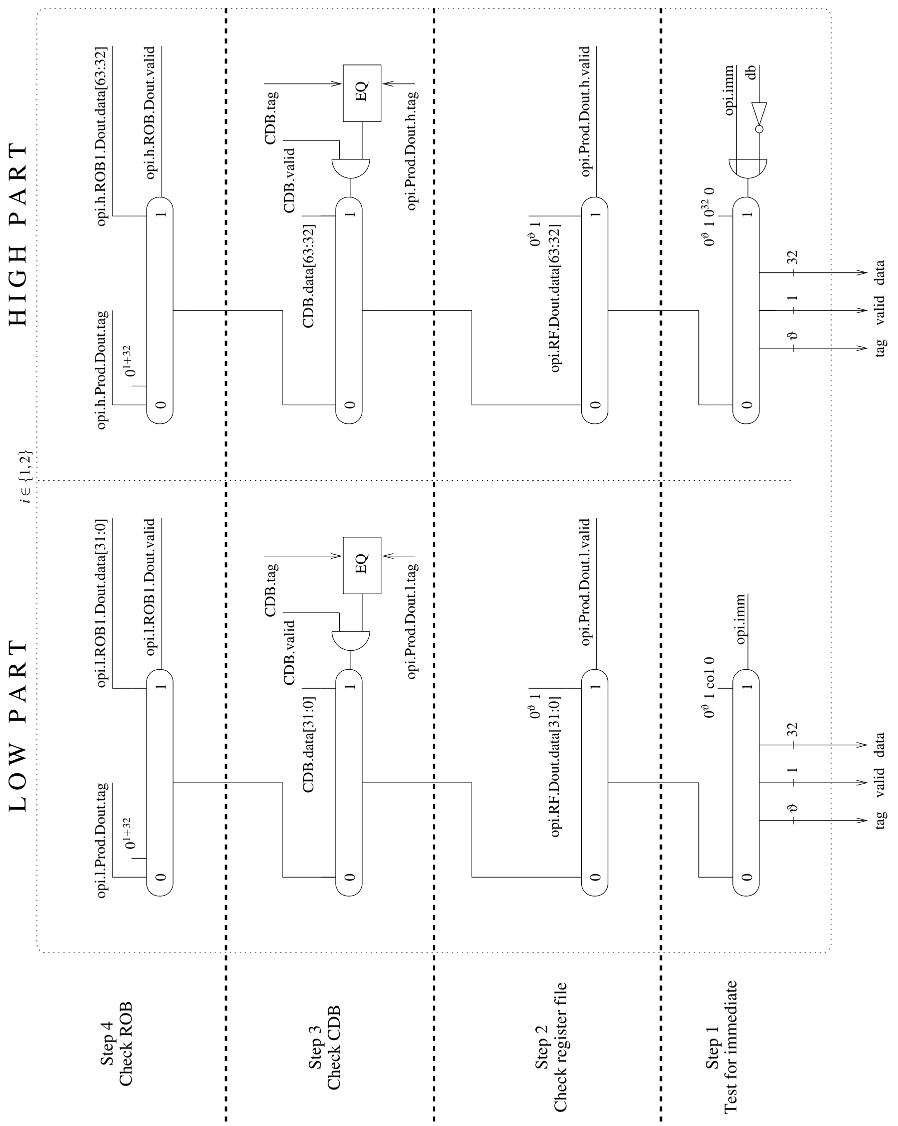


Figure 3.11: Operand data generation datagen for op1 and op2

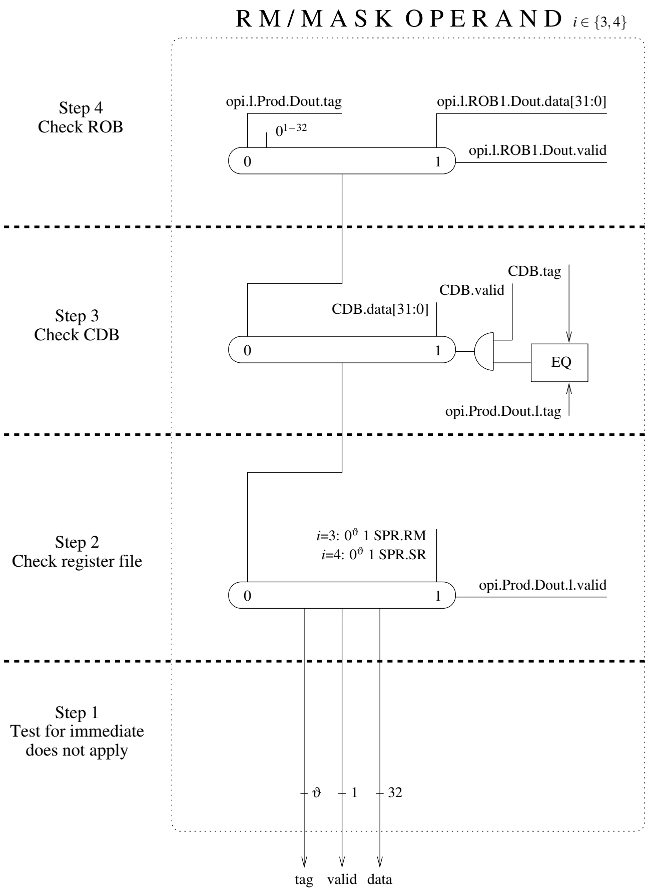


Figure 3.12: Operand data generation datagen for op3 and op4

3.5.5 Destination Operand Generation destgen

The destination operand generation environment destgen calculates the type and the address of the destination register. This circuit is similar to the address generation environment, which performs the same task for the source operands. The register type of destination is determined according to table 3.2 as:

$$\begin{aligned} \text{dest.gpr} &= \text{dest.RD} \vee \text{dest.R31} \\ \text{dest.fpr} &= \text{dest.FD} \\ \text{dest.spr} &= \text{dest.SA} \vee \text{dest.FCC} \end{aligned}$$

The destination is a double precision floating point value if the op.db signal is active or if it is a cvt.s.d or ctv.i.d instruction. These instructions can be distinguished from the other cvt instructions by IR[6] (appendix C).

$$\text{dest.db} = \text{op.db} \vee (\text{op.fconv} \wedge \text{IR}[6])$$

The destination register address is extracted from bit fields of the instruction word (appendix C). The positions of these bitfields depend on the instruction word layout, which is specified by the itype and rtype signals.

$$\text{dest.A} = \begin{cases} \text{IR}[20:16] & : \text{itype} \wedge (\text{dest.RD} \vee \text{dest.FD}) \\ \text{IR}[15:11] & : \text{rtype} \wedge (\text{dest.RD} \vee \text{dest.FD}) \\ \text{IR}[10:6] & : \text{dest.SA} \\ \text{bin}(8)\ (\text{FCC}) & : \text{dest.FCC} \\ \text{bin}(31)\ (R_{31}) & : \text{dest.R31} \\ 0 & : \text{otherwise} \end{cases}$$

3.5.6 Stall Generation stallgen

Issue stalls occur if one or more of the following conditions hold:

1. For the given instruction, all appropriate reservation stations are busy (FUbusy is active, section 3.5.2).
2. The instruction has to be stored in the reorder buffer, but the reorder buffer is full (ROB.full is active, section 3.9).
3. If the instruction is a moves2i and the source register is IEEEf, an issue stall is performed until the ROB is empty, to ensure that the register file contains the correct value. This is necessary, since floating point instructions modify the IEEEf special purpose register without any note in the producer table. The signal IEEEfstall is active under this condition:

$$\text{IEEEfstall} = \text{op.movs2i} \wedge (\underbrace{IR[10:6]}_{\text{source}} = \underbrace{00111}_{\text{IEEEf}}) \wedge \overline{\text{ROB.empty}}$$

Alternatively, a check for floating point instructions in the reorder buffer would be sufficient and could increase IPC rates at higher hardware cost.

4. The instruction is a conditional branch or a jump register instruction and the source operand op1 is not yet available. Issuing these instructions would require speculative execution, which is part of a thesis by Mark A. Hillebrand [Hil99]. The signal bstall indicates this stall condition:

$$\text{bstall}=(\text{op.branch} \vee \text{op.jumpR}) \wedge \overline{\text{op1.l.valid}}$$

5. If the instruction is a rfe instruction, an issue stall is required until the ROB is empty. This ensures that the ESR, EPC, and EPCn registers contain the correct values, since they might be modified by an instruction or interrupt prior the rfe instruction. This condition is indicated by the signal rfestall:

$$\text{rfestall}=\text{op.rfe} \wedge \overline{\text{ROB.empty}}$$

In the cycle after the stall, DOrfe is activated. This signal causes the actual register transfers, which are done in the PC environment and in the register file environments.

$$\text{DOrfe}=\text{op.rfe} \wedge \text{ROB.empty}$$

6. The instruction fetch and issue stages have to be stalled if the instruction memory system is busy (IBusy is active) in order to prevent the destruction of the PC registers.

Furthermore, interrupts overrule any issue stall condition. This is done since the instruction, which causes the interrupt, is always ahead of the instruction which causes the issue stall. Thus, the issuestall signal is generated as:

$$\text{issuestall}=(\text{FUbusy} \vee \text{ROB.full} \vee \text{IEEEfstall} \vee \text{bstall} \vee \text{rfestall} \vee \text{IBusy}) \wedge \overline{\text{JISR}}$$

In case of an issue stall, the following actions are performed:

- The instruction fetch is stalled. This is done by disabling the clock enable signals of PC0, PC1, and IR1(section 3.2).
- All D.FU$_i$.issue signals are disabled (section 3.5.2) in order to prevent that the instruction is written into a reservation station.
- The instruction is not stored in the reorder buffer (section 3.9).
- The producer table is not modified (section 3.5.5).

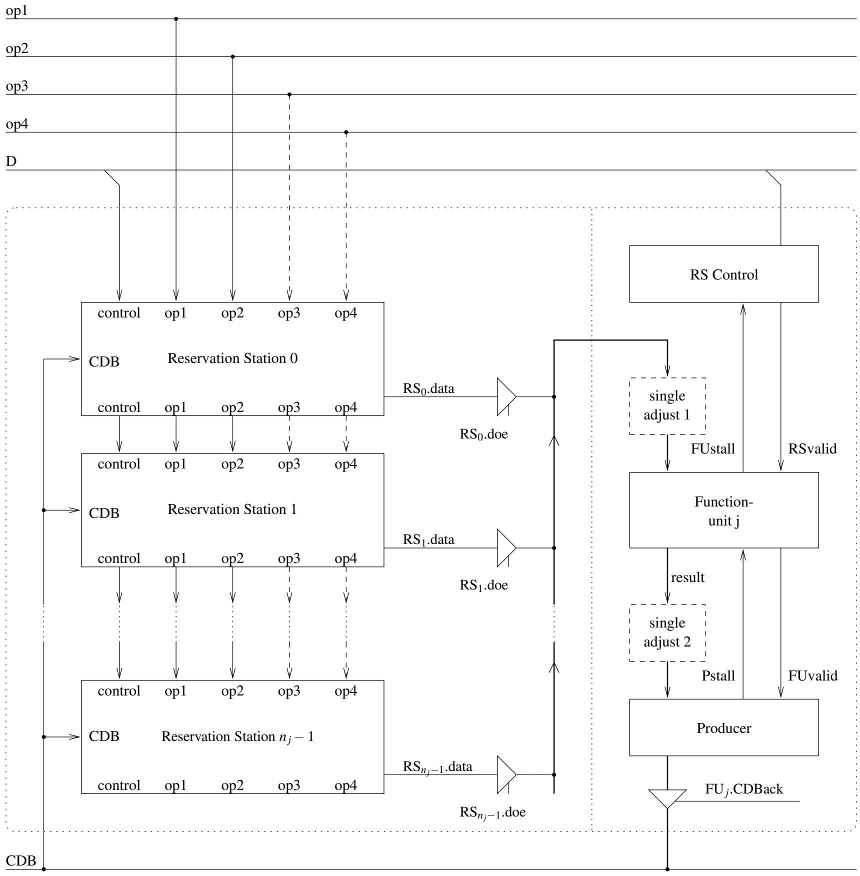


Figure 3.13: A complete function unit with reservation stations

3.6 The Reservation Station Environments

3.6.1 Overview

Each function unit has its own set of reservation stations assigned to it. Figure 3.13 gives an overview of a function unit with reservation stations and the producer. The dashed paths and circuits are extensions only needed for floating point function units.

The reservation stations form a queue for instructions and their operands which are provided on the op1 to op4 busses. These busses originate in the decode/issue environment. In each cycle, any desired instruction can move from its reservation station into the function unit. For this purpose, all reservation stations are connected to a bus with tristate drivers. The bus and the reservation stations are controlled by the reservation station control.

If the function unit is a floating point unit, the data on this bus is adjusted in the single-adjust-one circuit. This circuit makes sure that single precision values are in the lower 32 bits of the bus. After leaving the function unit, the single-adjust-two circuit makes sure that single precision values are on both lower and higher 32 bits of the bus. After that, the result is propagated on the CDB by the producer circuit.

Integer function units do not need the single adjust circuits. The instruction is passed unmodified to the function unit. After leaving the function unit, the result is passed unmodified to the producer.

3.6.2 Operation of the Reservation Stations

As mentioned above, the reservation stations of a function unit j form a queue for the instructions and their operands. Let the queue have n_j reservation stations. The design in this thesis allows any number of reservation stations. The choice of n_j only depends on cost effectiveness. Chapter 5 contains a comparison of different assignments.

New instructions are always issued in-order into the first reservation station (reservation station 0). The input values for reservation station 0 are generated by the decode/issue environment.

For each operand of an instruction, a valid bit and tag bits are stored in the reservation station. The valid bit is set iff the operand is already in the data item of the reservation station. An instruction in a reservation station is said to be valid if all its operands are available, i.e., valid. If not so, the tag bits hold the tag of the instruction which generates the operand. In this case, the operand circuits snoop on the CDB for the missing operands. The operand circuit compares the tag on the CDB to the tag stored in its register. If both are equal and if the valid bit of the CDB is active, the data item of the CDB is clocked into the data item of the operand.

As soon as one or more instructions in the queue become valid, the oldest among these instructions is dispatched into the function unit and removed from the queue. The reservation station control calculates the necessary output enable signals.

In each cycle, an instruction in reservation station i moves into reservation station $i+1$, unless reservation station $i+1$ is full and cannot be freed by moving its content into reservation station $i+2$ or by dispatching the instruction into the function unit. The reservation station control calculates the necessary clock enable signals.

3.6.3 Implementation of the Reservation Stations

Each reservation station can hold the operation code and the operands of one instruction. An implementation of an integer reservation station is given in figure 3.14. The reservation station has a register for the full bit, the tag bits and an operation code op. The full bit indicates that the reservation station is in use. The tag data item is the ROB tag of the instruction in the reservation station. The coding of the op data item depends on the interface to the function unit.

The values in reservation station i are updated if the RS_i.fill signal is active. The

RS_i.op$x$		RS_{i-1}.op$x$	new value of
fill	readCDB	readCDB	RS_i.op$x$.data
0	0	*	RS_i.op$x$.data
0	1	*	CDB.data
1	*	0	RS_{i-1}.op$x$.data
1	*	1	CDB.data

Table 3.6: Calculation of the new value of a reservation station operand

new values for the reservation station are selected in dependence of the RS_i.clear signal. If active, the reservation station is filled with an empty entry. If not active, the content of the previous reservation station RS_{i-1} is copied. The reservation station is also cleared in case of an interrupt, as indicated by the JISR signal. Thus, the content of RS_i is calculated as follows:

$$\mathrm{RS}_i = \begin{cases} \emptyset & : (\mathrm{RS}_i.\mathrm{fill} \wedge \mathrm{RS}_i.\mathrm{clear}) \vee \mathrm{JISR} \\ \mathrm{RS}_{i-1} & : \mathrm{RS}_i.\mathrm{fill} \wedge \overline{\mathrm{RS}_i.\mathrm{clear}} \wedge \overline{\mathrm{JISR}} \\ \mathrm{RS}_i & : \overline{\mathrm{RS}_i.\mathrm{fill}} \wedge \overline{\mathrm{JISR}} \end{cases}$$

The clear signal only affects the op, tag, and full bits, which are set to zero by a multiplexer. The other registers of the reservation station are not cleared in order to save hardware cost.

Integer function units require two 32 bits wide operands. Each operand has its own box (figure 3.15). Each operand has three components, which are the valid, tag, and data component. The valid bit is set iff the operand is already in the reservation station, i.e., in the data component of the operand register. If not so, the tag bits contains the ROB tag of the instruction which produces the operand. Reservation station operands are updated in two ways: The first way is to copy the content of the same operand in the previous reservation station. This is done iff the fill signal is active. The second way is to copy the content of the corresponding components of the CDB. This is done if the readCDB signal is active, which is calculated as follows:

$$\mathrm{readCDB} = (\mathrm{CDB.tag}{=}\mathrm{RS}_i.\mathrm{op}x.\mathrm{tag}) \wedge \overline{\mathrm{RS}_i.\mathrm{op}x.\mathrm{valid}} \wedge \mathrm{CDB.valid}$$

If readCDB is active, the reservation station operand provides the new value (i.e., the value on the CDB) as output to the next reservation station and to the function unit. The forwarding of the CDB data is essential for the following reasons: The operand is only one cycle on the CDB. If the data in a reservation station moves into the next reservation station, the operand on the CDB must be written into the next reservation station. Table 3.6 lists how the content of a reservation station operand is calculated.

Furthermore, the valid signal of the reservation station operand becomes active in the same cycle in which readCDB is active. This allows dispatching instructions

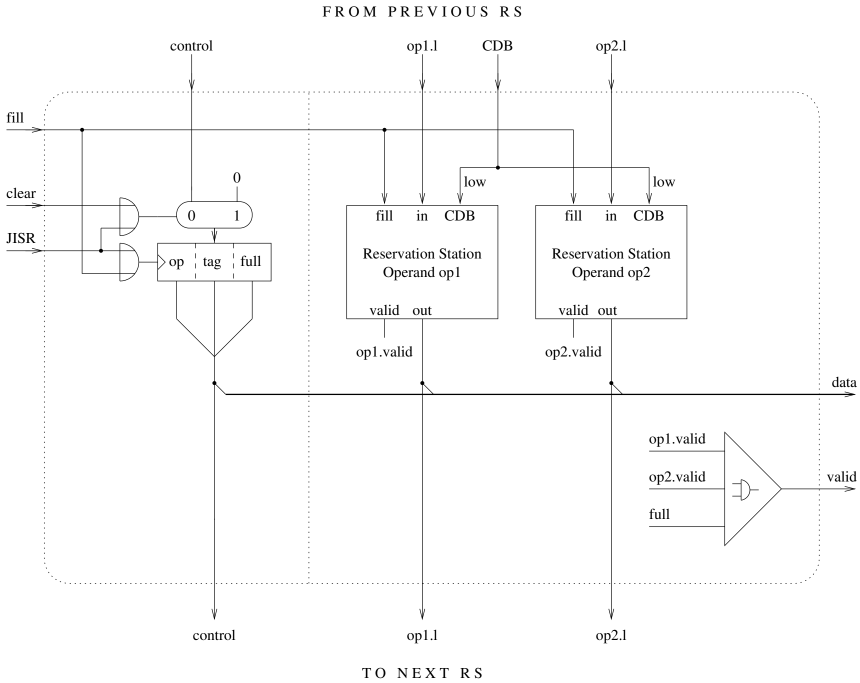


Figure 3.14: Reservation station for integer function units

in the same cycle they received their operands via the CDB. This is a performance optimization only and does not affect correctness.

Floating point function units require six operands: two 64 bits wide operands (split into low and high part, respectively) and the rounding mode **rm** and the interrupt mask **mask**. The interrupt mask is needed by the rounder, since the result of an IEEE floating point operation depends on the interrupt mask [Ins85, EP97]. The implementation of the floating point reservation station is identical to the implementation of an integer reservation station except for the additional operands. An implementation of a floating point reservation station is in figure 3.16. The implementation of the operand circuits of a floating point reservation station is identical to the implementation of the operand circuits of an integer reservation station.

3.6.4 Reservation Station Control

Dispatch Control

The reservation station control (figure 3.17) autonomously governs the dispatch of the valid instructions of the reservation stations into the function unit. Let RS_0,...,RS_{n_j-1} be the reservation stations of function unit FU_j.

The RS_i.doe signal is set iff the instruction in reservation station i is dispatched into the function unit. This transfer is done by a special bus. Each reservation station can

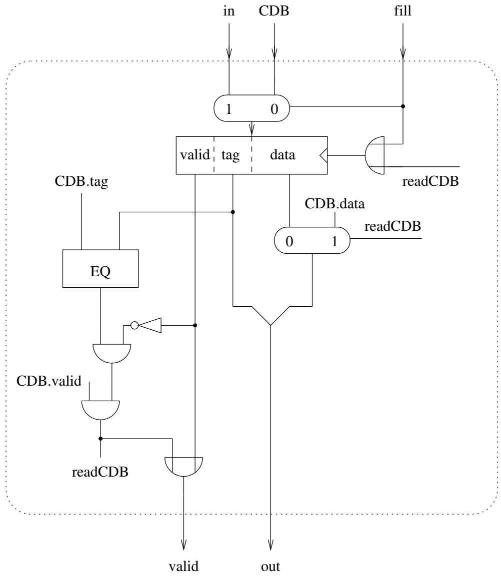


Figure 3.15: Reservation station operand

write on this bus. RS_i.doe is the output enable signal of the bus driver of reservation station i.

The correctness proof (chapter 6) relies on choosing the oldest among the valid instructions. Since new instructions are always placed in reservation station 0, the oldest valid instruction is obviously in the reservation station with the highest index among the valid reservation stations. Let RS_α be the reservation station which is to become dispatched.

$$\alpha = \max\{i \in \{0, ..., n_j - 1\} \mid \mathrm{RS}_i.\mathrm{valid{=}1}\}$$

The RS_i.valid signals are provided by the reservation stations. The max is calculated by a find last one (FLO) circuit (appendix A.1). The circuit returns α in unary representation. Let $\mathcal{A}_i$ denote this output. The dispatch has to be stalled if the function unit is stalled as indicated by the FUstall signal:

$$\mathrm{RS}_i.\mathrm{doe} = \mathcal{A}_i \wedge \overline{\mathrm{FUstall}}$$

At most one of the RS_i.doe signals has to be set in order to prevent bus contention. This is ensured by the find last one circuit.

Flow Control Signals

The reservation station control generates two flow control signals: The RSvalid and FU_j.full signal.

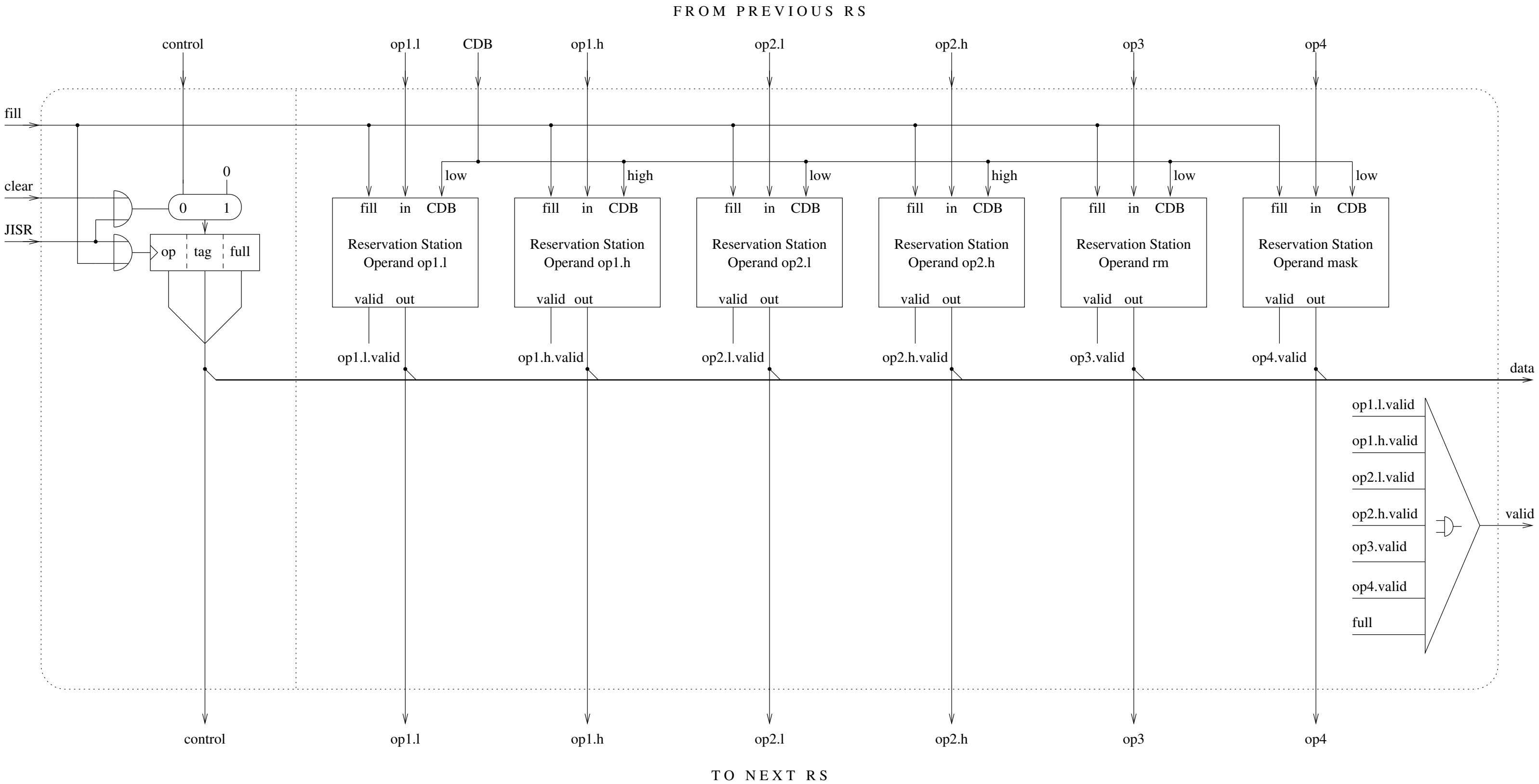


Figure 3.16: Reservation station for floating point function units

The RSvalid signal is active iff data is dispatched into the function unit. This is true iff there is at least one valid reservation station and the function unit is not stalled. Thus:

$$
\begin{aligned}
\text{RSvalid} &= \left(\bigvee_{i=0}^{n_j-1} \text{RS}_i.\text{valid}\right) \wedge \overline{\text{FUstall}} \\
&= \text{Zero}(\text{RS}_0.\text{valid},...,\text{RS}_{n_j-1}.\text{valid}) \quad \text{NOR} \quad \text{FUstall}
\end{aligned}
$$

The find last one ciruit (appendix A.1) has a built-in zero tester, so that the signal can be generated with a single NOR gate.

The FU_j.full signal is active iff the reservation stations of function unit j are not able to accept an instruction. However, even if all reservation stations of a function unit are full, an instruction can be issued into a reservation station by dispatching one instruction into the function unit if the function unit itself is not stalled.

$$
\text{FU}_j.\text{full} = \bigwedge_{i=0}^{n_j-1} \text{RS}_i.\text{full} \wedge \overline{\text{RSvalid}}
$$

In case of an active FU_j.full signal, the decode/issue environment does not generate a FU_j.issue signal for the function unit.

Queue Control Signals

The reservation station control also computes the RS_i.fill and RS_i.clear signals. As described above, RS_i.fill is active iff entry i is to be filled with new values. The RS_i.clear signal controls whether to clear the reservation station or to copy the data of its predecessor RS_{i-1}. In case of the first reservation station RS_0, the data of the predecessor is the instruction provided by the decode/issue environment. The clear signal of a reservation station is only used in the following cases:

- It is used if the entry of the previous reservation station is dispatched and therefore leaves the reservation station queue.
- It is used in case of the first reservation station, if no instruction is issued into the first reservation station.

The calculation of the qeue control signals is non-trivial and recursively defined as follows:

$i = n_j - 1$: The last reservation station does not have a successor. It is filled with the data of its predecessor iff its content is dispatched into the function unit (RS_{n_j-1}.doe=1) or if it is empty (RS_{n_j-1}.full=0):

$$\mathrm{RS}_{n_j-1}.\mathrm{fill} = \mathrm{RS}_{n_j-1}.\mathrm{doe} \vee \overline{\mathrm{RS}_{n_j-1}.\mathrm{full}}$$

If the content of the predecessor (i.e., RS_{n_j-2}) is dispatched into the function unit, it must not become copied. Thus, the clear signal of the last reservation station is active in this case:

$$\mathrm{RS}_{n_j-1}.\mathrm{clear} = \mathrm{RS}_{n_j-2}.\mathrm{doe}$$

Table 3.7 contains a list of the possible values.

$i \in \{1, ..., n_j - 2\}$: For RS_1 to RS_{n_j-2}, the calculation of the RS_i.fill signal is slightly modified, since these reservation stations have a successor. The signal RS_i.fill is also active if RS_{i+1} takes over the content of RS_i.

$$\mathrm{RS}_i.\mathrm{fill} = \mathrm{RS}_i.\mathrm{doe} \vee \overline{\mathrm{RS}_i.\mathrm{full}} \vee \mathrm{RS}_{i+1}.\mathrm{fill}$$

The calculation of the clear signal is identical to the calculation in the previous case.

$$\mathrm{RS}_i.\mathrm{clear} = \mathrm{RS}_{i-1}.\mathrm{doe}$$

$i = 0$: The first reservation station does not have a predecessor. The input values for the first reservation station are provided by the decode/issue environment. These values are only valid if an instruction is issued into the first reservation station of function unit j. Thus, the reservation station is filled with an empty entry except on issue:

$$\mathrm{RS}_0.\mathrm{clear} = \overline{\mathrm{D.FU}_j.\mathrm{issue}}$$

The calculation of the fill signal of reservation station zero is identical to the calculation in the general case.

In order to resolve the recurrency in the formulae of the RS_i.fill signals, define a set of signals $\mathcal{F}_j(i)$ as:

$$\mathcal{F}_j(i) = \mathrm{RS}_i.\mathrm{doe} \vee \overline{\mathrm{RS}_i.\mathrm{full}}$$

Now, a closed formula for RS_i.fill can be specified for $i \in \{0, ..., n_j - 1\}$:

$$\mathrm{RS}_i.\mathrm{fill} = \bigvee_{k=i}^{n_j-1} \mathcal{F}_j(k)$$

Since OR is associative, a parallel prefix circuit can be used in order to compute the RS_i.fill signals (figure 3.17).

RS_{n_j-2}.full	RS_{n_j-1}.full	RS_{n_j-2}.doe	RS_{n_j-1}.doe	RS_{n_j-1}.clear	RS_{n_j-1}.fill	action in RS_{n_j-1}
0	0	0	0	0	1	copy previous RS, which is empty
		0	1	not possible		
		1	0			
		1	1			
0	1	0	0	0	0	no action
		0	1	0	1	copy previous RS, which is empty
		1	0	not possible		
		1	1	not possible		
1	0	0	0	0	1	copy instruction in previous RS
		0	1	not possible		
		1	0	1	1	clear RS, although already empty
		1	1	not possible		
1	1	0	0	0	0	no action
		0	1	0	1	replace the current instruction with instruction in previous RS
		1	0	1	0	no action
		1	1	not possible		

Table 3.7: Deduction of the RS_{n_j-1}.clear and RS_{n_j-1}.fill signals

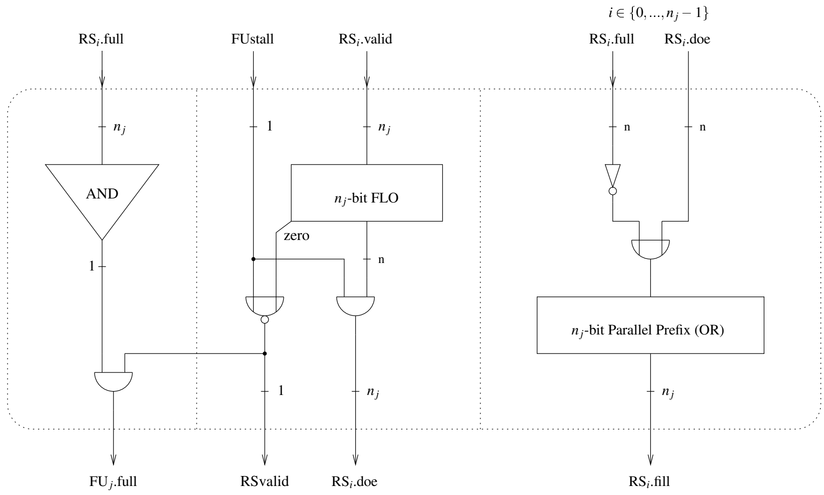


Figure 3.17: Reservation station control

Correctness

The correctness of the calculation of the queue control signals RS_i.fill and RS_i.clear is an implication of the following three claims:

Claim 1: Issued instructions are stored in RS_0.

Proof of Claim 1: During issue, D.FU_j.issue is active (page 25), and therefore FU_j.full is inactive. This implies that there is either an empty reservation station or that there is a reservation station which is being dispatched. In either case, there is at least one reservation station RS_i with RS_i.fill=1. Thus, RS_0.fill is active, and RS_0.clear is inactive. The instruction is therefore stored in RS_0.

Claim 2: No instruction in a reservation station gets lost, i.e., it is either dispatched to the function unit or remains in a reservation station.

Claim 3: Reservation stations, which are copied or dispatched, are cleared or overwritten afterwards. Reservation stations, which are dispatched, are not copied.

Proof of claim 2 and 3: Let instruction I be in RS_i. If instruction I is dispatched (i.e., RS_i.doe is active), claim 2 is obvious. Claim 3 follows from RS_i.fill=1 and RS_{i+1}.clear=1.

Let instruction I not be dispatched (i.e., RS_i.doe=0). We will now show that I then either moves to the next reservation station RS_{i+1} or stays in RS_i. For that purpose, we distinguish the cases that the signal RS_{i+1}.fill is active or inactive.

Let RS_{i+1}.fill=1. The RS_{i+1}.clear signal is inactive because of RS_i.doe=0, and thus the entry is copied into RS_{i+1} and claim 2 follows. Claim 3 holds because of RS_i.fill=1, which is true because of RS_{i+1}.fill=1.

Let RS_{i+1}.fill=0. The claim 3 does not apply because the entry is neither dispatched nor copied. Claim 2 only applies for RS_i if it contains an instruction, i.e., RS_i.full=1. We distinguish two cases:

- If i is $n_j - 1$, i.e., if the reservation station is the last reservation station in the queue, RS_i.fill is calculated as:

$$\text{RS}_{n_j-1}.\text{fill} = \text{RS}_{n_j-1}.\text{doe} \vee \overline{\text{RS}_{n_j-1}.\text{full}}$$

  Since RS_{n_j-1}.doe=0 and RS_{n_j-1}.full=1, the fill bit RS_{n_j-1}.fill is inactive, and the entry therefore remains in the queue.

- If i is not $n_j - 1$, RS_i.fill is calculated as:

$$\text{RS}_i.\text{fill} = \underbrace{\text{RS}_i.\text{doe}}_{0} \vee \underbrace{\overline{\text{RS}_i.\text{full}}}_{0} \vee \underbrace{\text{RS}_{i+1}.\text{fill}}_{0} = 0$$

3.6.5 Single Adjust

The single adjust circuits are only used in floating point function units. They are controlled by three signals, which are stored in the op data item of each reservation station. The op1.high, op2.high, and dest.high signals are the least significant bits of the register address of operand one, two, and the destination, respectively. The db signal is active iff the operation has double precision source registers.

Before the function unit, the operands one and two from the reservation station pass the single-adjust-one circuit. Double precision operands are passed unmodified (op$x$.high is false in this case, since double precision operands always have even register addresses). If single precision operands are used (the db signal is not active), it ensures that the operand is always in the lower 32 bits of the data path. The upper 32 bits are set to zero, which is a requirement of the floating point function units used in this design. Table 3.8 lists the results of the circuit in dependence of the input signals. An implementation of this function is given in figure 3.18.

After leaving the function unit, the single-adjust-two circuit between the function unit and the producer part of the reservation station reverts this procedure. It ensures that a single precision result is both on the low and on the high part of the CDB to avoid any possible alignment problems. The implementation is given in figure 3.19.

3.6.6 The Producer

The producer (figure 3.20) propagates the results of an associated function unit on the CDB. Table 3.9 lists all components of the CDB. The producer has to generate a value for all components. The function unit provides a signal FUvalid. If FUvalid is set, the

Inputs		Result	
high	db	low part	high part
0	0	data[31:0]	0^{32}
1	0	data[63:32]	0^{32}
0	1	data[31:0]	data[63:32]
1	1	not possible	

Table 3.8: Single adjust before function unit for one operand. The input from the reservation is data[63:0].

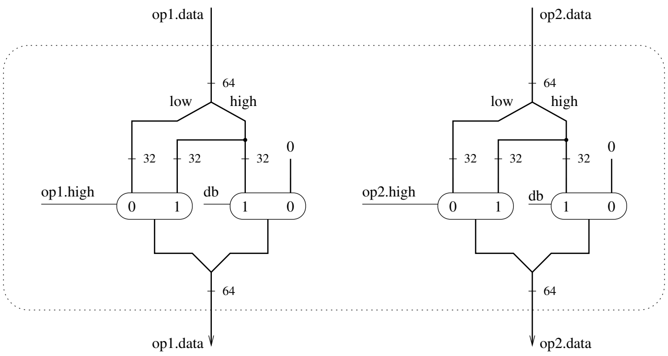


Figure 3.18: Single adjust before function unit

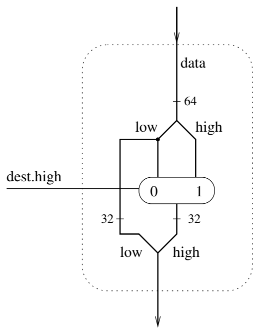


Figure 3.19: Single adjust after function unit

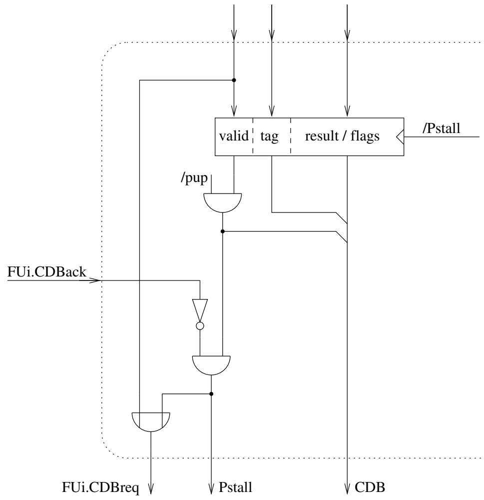


Figure 3.20: Producer

function unit delivers a result, result flags (ovf, IEEE flags, etc), and the tag. These values are stored in registers. In the same cycle, the producer requests the CDB for the next cycle at the CDB control by raising FU_j.CDBreq. Let t be the cycle of the request.

In case of an acknowledgement (FU_j.CDBack=1) by the CDB control in cycle $t+1$, the values in these registers are put on the CDB, and the register is filled with the next result. If the CDB control does not acknowledge the request within cycle $t+1$, the values stay in the registers, the function unit is stalled with the signal Pstall, and the producer requests the CDB again for cycle $t+2$.

Let FU_j.P be the register of the producer part of function unit j. The Pstall signal stalls the whole function unit. Alternatively, the function unit might contain a stall engine. The Pstall signal is active, if there is an instruction in the producer register stage (FU_j.P.valid=1) and if there is no acknowledge from the CDB control (FU_j.CDBack=0). The valid bit from the register is forced to be zero in the power-up cycle (pup=1). Thus, Pstall is calculated as follows:

$$\mathrm{Pstall} = \overline{\mathrm{FU}_j.\mathrm{CDBack}} \wedge \overline{\mathrm{pup}} \wedge \mathrm{FU}_j.\mathrm{P.valid}$$

The CDB is requested iff the function unit provides a result (FUvalid=1) or iff there is a result in the register and no acknowledge from the CDB control (Pstall=1).

$$\mathrm{FU}_j.\mathrm{CDBreq} = \mathrm{FUvalid} \vee \mathrm{Pstall}$$

Bus	Items	Width	Purpose
CDB	tag	ϑ	ROB tag of the instruction producing the result
	valid	1	CDB.valid= 1 $\Leftrightarrow$ CDB contains valid data
	data	64	actual result
	mal	1	misaligned memory access
	Dpf	1	page fault during data memory access
	ovf	1	overflow in ALU instruction
	IEEEf	5	IEEE conforming floating point flags
	EData	32	exception data

Table 3.9: Components of the CDB

3.7 Function Unit Environments

The function unit environments contain the function units, which are the ALU, floating point units, and the data memory interface (table 3.3). These environments belong to the execute stage.

3.7.1 Integer Function Unit

The integer function unit performs traditional ALU functions and shifting. Table 3.10 defines the interface to this function unit. It contains the coding of the operation control signals op[4:0]. The unit can generate one exception (FXU overflow). The exception can be suppressed by a bit in the opcode. This test is done in the ALU itself.

Figure A.4 (appendix A, page 100) gives the implementation of the integer function unit, which is taken almost literally from [MP95]. During issue, the op[] signals are calculated from corresponding bits in the instruction word as follows: The ALU function is defined by bits in the opcode. The position of these bits depends on the instruction format. Instructions with itype format (op.itype=1) use IR[30] and IR[28:26] for this task. Instructions with rtype format use IR[5:0]. The circuit in figure 3.21 selects the correct signals.

3.7.2 Floating Point Function Units

With respect to cost and delay, the floating point units are taken from [Lei98]. Nevertheless, the design supports any function units which comply with the interface. Since each function unit can generate an independent stall signal, even function units with variable latency can be used.

In contrast to [Lei98], each function unit is assumed to have a rounder of its own. However, this is only relevant for cost, delay, and CPI calculation. The scheduling algorithm itself does support sharing of floating point unit parts between function units,

op[4]	op[3]	op[2]	op[1]	op[0]	Function
0	0	0	0	0	$a << b$
0	0	0	1	0	$a >> b$
0	0	0	1	1	$a >> b$ (arithmetic)
1	0	0	0	0	$a + b$ with test of overflow
1	0	0	0	1	$a + b$ without test of overflow
1	0	0	1	0	$a - b$ with test of overflow
1	0	0	1	1	$a - b$ without test of overflow
1	0	1	0	0	$a \wedge b$
1	0	1	0	1	$a \vee b$
1	0	1	1	0	$a \oplus b$
1	0	1	1	1	$b[0:15]0^{16}$
1	1	0	0	1	$a > b\ ?\ 1 : 0$
1	1	0	1	0	$a = b\ ?\ 1 : 0$
1	1	0	1	1	$a \geq b\ ?\ 1 : 0$
1	1	1	0	0	$a < b\ ?\ 1 : 0$
1	1	1	0	1	$a \neq b\ ?\ 1 : 0$
1	1	1	1	0	$a \leq b\ ?\ 1 : 0$

Table 3.10: Coding of integer operations

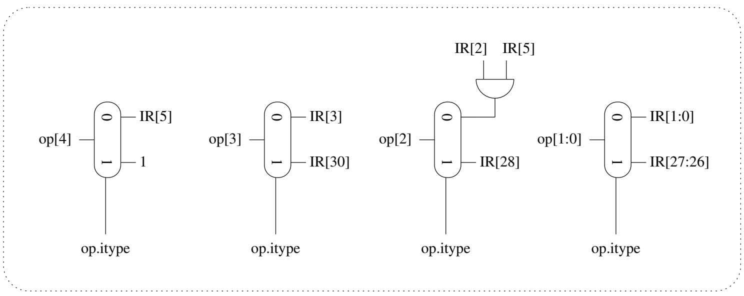


Figure 3.21: Calculation of op[4:0] for the ALU

Purpose	Latency	# RS
floating point addition and substraction	5	2
floating point multiplication	5	2
floating point division	15	1
conversion floating point / integer	4	1
floating point condition tests	1	1

Table 3.11: Floating point function units

which results in big cost savings. However, since no CPI simulations are available for floating point units with shared rounder, separate rounder are used to keep the machine comparable. For the same reason, the list of floating point function units (table 3.11) is taken from [Ger98]. The table also lists the number of reservation stations which belong to each function unit. Floating point reservation stations are very expensive regarding hardware cost. Thus, it is advisable to combine the multiplication/division FU and the conversion/test FU to save two sets of reservation stations. Again, simulations for this configuration are missing.

One third of the cost of a floating point reservation station is caused by the operand entries for the rounding mode and the interrupt mask. In order to save this cost, it is possible to encode the rounding mode RM in the instruction opcode (there is still room left, appendix C). Furthermore, forwarding of the interrupt mask is not cost efficient, since it changes rarely. It is only required in the rounder which is in the last stages of each floating point function unit. Due of that, it is more cost efficient to design a floating point function unit which directly reads the interrupt mask from the register file as soon as an instruction arrives at the rounder stage. If the interrupt mask is not valid, the function unit could generate a stall signal.

The implementation of floating point units is beyond this thesis. The actual operation performed by the FU is determined by IR[8:0]. These bits are forwarded to the reservation station during issue as part of the op bits.

3.8 CDB Control Environment

The CDB control environment allocates the CDB to the function units. The CDB is requested by the producer of the function unit i by raising FU_i.CDBreq. The CDB control environment generates exactly one FU_j.CDBack in the next cycle. Figure 3.22 gives an implementation.

3.8.1 Deduction

Let n be the total number of producers and let $\mathrm{FU}_i(t)$.CDBreq and $\mathrm{FU}_i(t)$.CDBack be the request and acknowledge signals of function unit $i \in \{0,\ldots,n-1\}$ in cycle t. Now,

$\mathcal{R}(t)$ and $\mathcal{A}(t)$ are defined as follows: $\mathcal{R}(t)$ contains the producers which request the CDB in cycle t. $\mathcal{A}(t)$ contains the active acknowledge signals in cycle t.

$$\begin{aligned}\mathcal{R}(t) &= \{i \in \{0,...,n-1\} \mid \mathrm{FU}_i(t).\mathrm{CDBreq} = 1\}\\ \mathcal{A}(t) &= \{i \in \{0,...,n-1\} \mid \mathrm{FU}_i(t).\mathrm{CDBack} = 1\}\end{aligned}$$

In each cycle t, multiple producers might be requesting the CDB. The CDB control has to choose exactly one because only one unit can use the CDB. The correctness proof of the Tomasulo scheduling algorithm with reorder buffer requires a guaranty that any unit requesting the CDB will get an acknowledgement within a finite limit of time (chapter 6). This is done by allocating the CDB **round robin**. This leads to the following algorithm for the calculation of the acknowledge signals:

1. Since only one unit can get the CDB, $\alpha(t)$ can be defined as:

   $$\alpha(t) = i \iff \mathcal{A}(t) = \{i\}$$

   Thus, it is required that there is exactly one function unit which gets the CDB for each cycle. The producer has to put defined values (with CDB.valid=0) on the CDB if it does not have real data. If there is only one request for the CDB for a given cycle, the calculation of $\alpha(t+1)$ is obvious. In case of more requests, round robin scheduling requires that the CDB is assigned to the unit which comes next after the unit which got the CDB in the previous cycle. In case of the last unit, the next unit is the first one.

2. $\mathcal{M}(t)$ contains the producers which have higher indices than $\alpha(t)$. It is defined as:

   $$\mathcal{M}(t) = \{i \in \{0,...,n-1\} \mid i > \alpha(t)\}$$

3. The requests from these producers are in $\mathcal{R}_{high}(t)$:

   $$\mathcal{R}_{high}(t) = \mathcal{R}(t) \cap \mathcal{M}(t)$$

4. Now, $\alpha(t)$ can be defined inductively. In the first (powerup) cycle, α is forced to be zero, i.e., the first function unit gets the CDB. The processing of $\alpha(t+1)$ is done as follows: If there are no requests, α remains the same. If there is one or more request, the first step is to check the requests of units in $\mathcal{R}_{high}(t)$. If there are no such requests, the request with the lowest index is acknowledged.

   $$\begin{aligned}\alpha(0) &= 0\\ \alpha(t+1) &= \begin{cases}\alpha(t) & : \mathcal{R}(t) = \emptyset\\ \min(\mathcal{R}_{high}(t)) & : \mathcal{R}_{high}(t) \neq \emptyset\\ \min(\mathcal{R}(t)) & : \text{otherwise}\end{cases}\end{aligned}$$

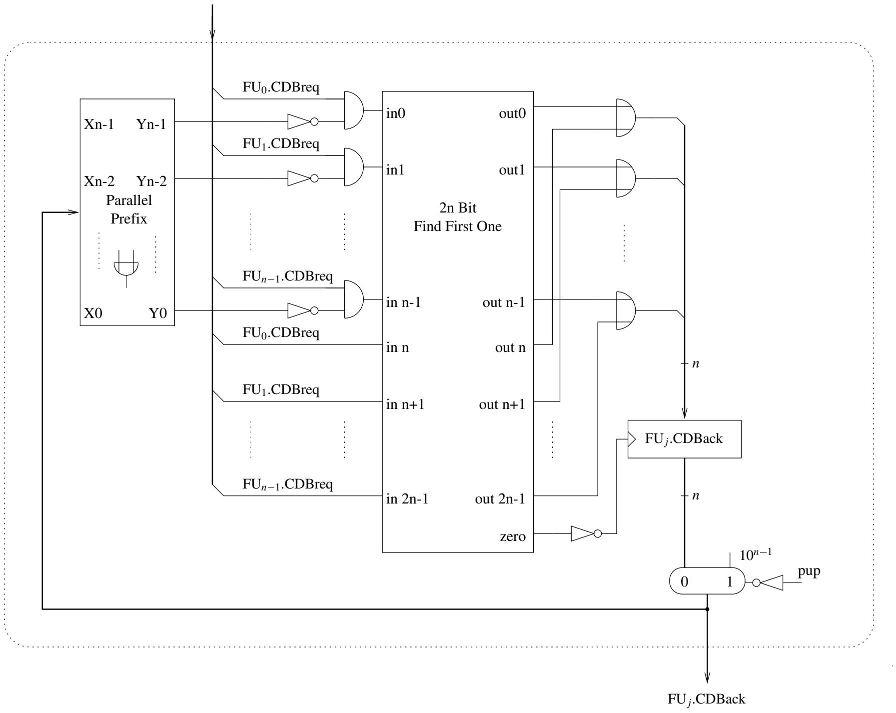


Figure 3.22: Common data bus control

3.8.2 Implementation

- The $\mathcal{R}(t)$ set, which contains the requests for the CDB, is provided by the producers of the function units as FU$_j$(t).CDBreq.
- The $\mathcal{A}(t)$ set is taken from a register, i.e., this register contains the FU$_j$(t).CDBack signals.
- $\mathcal{M}(t)$ is the set of producers with higher indices than $\alpha(t)$:

$$\begin{aligned} i \in \mathcal{M}(t) &\Leftrightarrow i > \alpha(t) \\ &\Leftrightarrow \forall j \in \{i, \ldots, n-1\} : j \notin \mathcal{A}(t) \\ &\Leftrightarrow \bigwedge_{j=i}^{n-1} \overline{\mathrm{FU}_j(t).\mathrm{CDBack}} \end{aligned}$$

Let $\mathcal{M}_i(t)$ denote that i lies in $\mathcal{M}(t)$, thus:

$$\mathcal{M}_i(t) = 1 \Leftrightarrow i \in \mathcal{M}(t)$$

Now, $\mathcal{M}_i(t)$ is calculated as follows:

$$\overline{\mathcal{M}_i(t)} = \bigvee_{j=i}^{n-1} \mathrm{FU}_j(t).\mathrm{CDBack}$$

This calculation is done by a parallel prefix circuit, since OR is associative. The outputs of the parallel prefix circuit are inverted to get the final signals of $\mathcal{M}$.

- The implementation of $\mathcal{R}(t) \cap \mathcal{M}(t)$ is obvious:

$$\begin{aligned} j \in (\mathcal{R}(t) \cap \mathcal{M}(t)) &\Leftrightarrow (j \in \mathcal{R}(t)) \wedge (j \in \mathcal{M}(t)) \\ (\mathcal{R}(t) \cap \mathcal{M}(t))_j &= \mathcal{R}_j(t) \wedge \mathcal{M}_j(t) \end{aligned}$$

- The two minimum operations for the calculation of $\alpha(t+1)$ are combined in one $2n$-bit find first one circuit to save cost and delay. The lower n-bit input signals of the circuit are connected to the $\mathcal{R}_{high}(t)$ bits calculated in the previous step. The input signals n to $2n-1$ are connected to $\mathcal{R}(t)$

  If $\mathcal{R}(t)$ is empty, the zero output of the find first one circuit is active. This disables the clock enable signal of the register which holds the acknowledge signals.

  If $\mathcal{R}_{high}(t)$ is not empty, the find first one circuit returns $\min(\mathcal{R}_{high}(t))$ in output bits 0 to $n-1$. The output bits n to $2n-1$ are zero in this case.

  If $\mathcal{R}_{high}(t)$ is empty, the find first one circuits returns zero in output bits 0 to $n-1$ and $\min(\mathcal{R}(t))$ in output bits n to $2n-1$. The bits n to $2n-1$ are mapped onto bits 0 to $n-1$ with n OR gates. This bit set is $\mathcal{A}(t+1)$ and is stored in the register.

3.9 Reorder Buffer Environment

The reorder buffer realizes in-order termination which is essential for precise interrupts. An introduction on reorder buffers is given in chapter 2.

The size of the reorder buffer has a significant impact on the CPI rate [Ger98]. For this thesis, a reorder buffer with Θ=16 entries is assumed, which is a cost efficient size, as shown by simulations. The size is assumed to be a power of two. The buffer requires $\vartheta = \log_2 \Theta = 4$ address bits (i.e., tag bits).

The reorder buffer itself is realized with two RAMs: ROB1 and ROB2. Table 3.12 lists all components of the ROB, their purpose and size, and the RAM they belong to. This separation saves cost and delay since the values in ROB2 are only used during retire and for the destination operand during issue. For these values, the forwarding read ports are saved. Table 3.13 shows the use of the ports.

ROB1 is a nine port $\Theta \times 105$ RAM (figure 3.23). ROB2 is a two port $\Theta \times 78$ RAM. The reorder buffer is organized as circular FIFO queue. It is addressed by two pointers: ROB.tail is the tail pointer and points to the target entry for new instructions. ROB.head is the head pointer and points to the next instruction for retire.

The head and tail pointers are maintained by two circuits which are identical to the circuits in [Lei98]. They provide ROB.head and ROB.tail and are controlled by two clock enable signals, ROB.headce and ROB.tailce, respectively. If the clock enable signal of a circuit is active, the corresponding pointer is incremented by one (with warp-around) in each cycle. In case of an interrupt, both pointers are set to zero. An implementation of both circuits is in figure A.5 (appendix A, page 101).

Another auxiliary circuit, which is also taken from [Lei98], calculates the ROB.full signal. If set, the signal indicates that the ROB is full. Furthermore, the circuit provides ROB.empty, which is active iff the ROB is empty. The circuit is controlled by the same control signals used for ROB.head and ROB.tail (figure A.6, appendix A, page 101).

3.9.1 Issue

During issue, the ROB entry pointed to by ROB.tail is allocated and initialized for the new instruction. This is done via port eight of the ROB. The write enable signal of this port is active iff issuestall is inactive, i.e., when an issue is performed. During issue, the tail pointer is incremented. The tail pointer is cleared during JISR. Thus, the ROB.tailce signal is calculated as:

$$\text{ROB.tailce} = \text{JISR} \vee \overline{\text{issuestall}}$$

The valid data item is initialized with one, iff the instruction is not passed to a function unit. This is indicated by the noFU signal, which is generated by the control automaton. The data and IEEEf items are filled with dummy data to have defined values in the ROB.

$$\begin{aligned}
\text{dest.ROB1.Din.valid} &= \text{noFU} \\
\text{dest.ROB1.Din.data} &= 0^{64} \\
\text{dest.ROB1.Din.IEEEf} &= 0^{5}
\end{aligned}$$

The dmal, Dpf, ovf, and IEEEf data items are used for interrupt processing. They indicate exceptions which can occur during the execution phase of an instruction and they are initialized with zero. In case of a trap instruction, EData contains the immediate constant co1 from the instruction register environment, which allows passing of an argument to the interrupt service routine.

$$\text{dest.ROB1.Din.EData} = \text{co1}$$

The ill, imal, Ipf, trap, and uFOP items of the ROB2 RAM are used for exceptions which occur during fetch or decode/issue. They are initialized with the corresponding signals provided by the decode/issue environment.

$$\begin{aligned}
\text{dest.ROB1.Din.ill} &= \text{op.ill} \\
\text{dest.ROB1.Din.imaö} &= \text{imal1} \\
\text{dest.ROB1.Din.Ipf} &= \text{pff1} \\
\text{dest.ROB1.Din.trap} &= \text{op.trap} \\
\text{dest.ROB1.Din.uFOP} &= \text{op.iuFOP}
\end{aligned}$$

The ROB2 RAM contains data items dest, db, gpr, fpr, and spr. These items specify the register file, the register file address and the operand width (double or single) for writeback. They are initialized by the corresponding values generated by the decode/issue environment (section 3.5).

dest.ROB2.Din.dest	=	dest.A
dest.ROB2.Din.db	=	dest.db
dest.ROB2.Din.gpr	=	dest.gpr
dest.ROB2.Din.fpr	=	dest.fpr
dest.ROB2.Din.spr	=	dest.spr

For the calculation of EPC and EPCn after the interrupt, the following additional information is required: the PC of the instruction and the branch/jump target from the PC environment. The bj data item is active iff the instruction in the ROB entry is a branch or jump instruction.

dest.ROB2.Din.PC	=	opc1
dest.ROB2.Din.target	=	target1
dest.ROB2.Din.bj	=	branch $\vee$ jump $\vee$ jumpR

3.9.2 Retire

On retire, a result is fetched from the head of the reorder buffer and written into the register file. This is done with ROB port seven. The conditions for retire are that the ROB must not be empty and that the entry at the head is valid.

$$\text{retire} = /\text{ROB.empty} \wedge \text{ROB.p7.Dout.valid}$$

During retire, the ROB head pointer is incremented. The head pointer is cleared during JISR. Thus, the ROB.headce signal is calculated as:

$$\text{ROB.headce} = \text{JISR} \vee \text{ROB.retire}$$

Before the actual writeback, interrupts are checked (almost identical to [MP95][1]). The first step is to collect the occurred interrupts in in CA[i]. CA[i] is active iff an interrupt of priority *i* occurred. Table 3.14 lists all interrupts and their priority. Lower numbers denote higher priority. CA[0] is the reset interrupt. It is triggered by pup, the power-up signal, in any case, even if there is no instruction to interrupt. CA[1] to CA[12] are internal interrupts. Their event signals are stored in the ROB. These event signals are only valid during retire therefore.

[1] MCA[6] must be masked in contrast to [MP95].

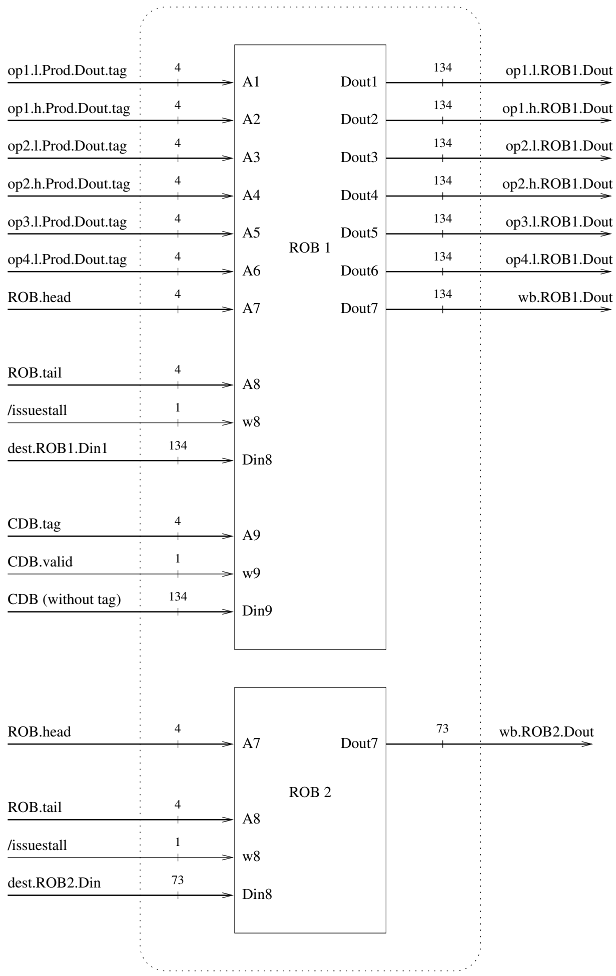


Figure 3.23: Reorder buffer

Name	Width	ROB	Purpose
valid	1	ROB1	valid = 1 $\Leftrightarrow$ data contains a valid value
data	64	ROB1	result data
dmal	1	ROB1	misaligned data memory access
Dpf	1	ROB1	data memory page fault
ovf	1	ROB1	overflow in ALU instruction
IEEEf	5	ROB1	IEEE flags (only used by floating point instr.)
EData	32	ROB1	exception data
	$\sum 105$		
ill	1	ROB2	illegal instruction
imal	1	ROB2	misaligned instruction memory access
Ipf	1	ROB2	instruction memory page fault
trap	1	ROB2	trap = 1 $\Leftrightarrow$ instruction is a trap instruction
uFOP	1	ROB2	unimplemented floating point instruction
dest	4	ROB2	destination register address
db	1	ROB2	db = 1 $\Leftrightarrow$ result has double precision
fpr	1	ROB2	fpr = 1 $\Leftrightarrow$ dest is a floating point register
spr	1	ROB2	spr = 1 $\Leftrightarrow$ dest is a special purpose register
gpr	1	ROB2	gpr = 1 $\Leftrightarrow$ dest is a general purpose register
PC	32	ROB2	PC of the instruction
target	32	ROB2	target / fallthrough address
bj	1	ROB2	bj = 1 $\Leftrightarrow$ instruction is a branch/jump
	$\sum 78$		

Table 3.12: Components of a reorder buffer entry

Port	Use	Purpose
1	read only ROB1	Forwarding of low part of operand 1
2	read only ROB1	Forwarding of high part of operand 1
3	read only ROB1	Forwarding of low part of operand 2
4	read only ROB1	Forwarding of high part of operand 2
5	read only ROB1	Forwarding of operand 3
6	read only ROB1	Forwarding of operand 4
7	read only ROB1, ROB2	Retire
8	write only ROB1, ROB2	Issue (destination)
9	write only ROB1	Completion

Table 3.13: Use of the reorder buffer ports

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

Table 3.14: Interrupts and coding of SR/CA

The misaligned access (mal) interrupt indicates both instruction memory and data memory misaligned accesses, which have separate event signals in the ROB. The calculation of CA[2] is different therefore. CA[13] to CA[31] are left over for external interrupts with event signals ex_1 to ex_{19}.

$$\text{CA[i]} = \begin{cases} \text{pup} & : \; i = 0 \\ \text{retire} \wedge \text{ROB.p7.ill} & : \; i = 1 \\ \text{retire} \wedge (\text{ROB.p7.dmal} \vee \text{ROB.p7.imal}) & : \; i = 2 \\ \text{retire} \wedge \text{ROB.p7.Ipf} & : \; i = 3 \\ \text{retire} \wedge \text{ROB.p7.Dpf} & : \; i = 4 \\ \text{retire} \wedge \text{ROB.p7.trap} & : \; i = 5 \\ \text{retire} \wedge \text{ROB.p7.ovf} & : \; i = 6 \\ \text{retire} \wedge \text{ROB.p7.IEEEf[i-7]} & : \; 7 \leq i \leq 11 \\ \text{retire} \wedge \text{ROB.p7.uFOP} & : \; i = 12 \\ \text{retire} \wedge \text{ex}_{i-12} & : \; i \geq 13 \end{cases}$$

Most interrupts are *maskable* (table 3.14). The service of interrupt i can be suppressed by setting SR[i] to zero. MCA contains the occurred interrupts which are not masked.

$$\text{MCA[i]} = \begin{cases} \text{CA}[i] & : \; i \leq 5 \vee i = 12 \\ \text{CA}[i] \wedge \text{SR}[i] & : \; \text{otherwise} \end{cases}$$

If an interrupt is serviced (i.e., at least one MCA[i] signal is active), the JISR signal is activated.

$$\text{JISR} = \bigvee_{i=0}^{31} \text{MCA}[i]$$

If the interrupt is of type continue, which is indicated by the IRQcontinue signal, the writeback must take place in spite of the interrupt. Interrupts 5 to 31 are of this type. The writeback is controlled by the writeback signal.

$$\text{IRQcontinue} = \overline{\left(\bigvee_{i=0}^{4} \text{MCA}[i]\right)}$$

$$\text{writeback} = (\overline{\text{JISR}} \vee \text{IRQcontinue}) \wedge \text{retire}$$

The writeback of the result of the instruction into the register file is performed via register file / producer table port three with the wb (writeback) signals. The address (wb.A) and the register file and the result data is taken from the ROB. These values are also used to address the producer table.

$$\begin{array}{lcl}
\text{wb.A} & = & \text{wb.ROB2.Dout.dest} \\
\text{wb.gpr} & = & \text{wb.ROB2.Dout.gpr} \\
\text{wb.fpr} & = & \text{wb.ROB2.Dout.fpr} \\
\text{wb.spr} & = & \text{wb.ROB2.Dout.spr} \\
\text{wb.db} & = & \text{wb.ROB2.Dout.db} \\
\\
\text{wb.l.RF.Din} & = & \text{wb.ROB1.Dout.data}[31:0] \\
\text{wb.h.RF.Din} & = & \text{wb.ROB1.Dout.data}[63:32]
\end{array}$$

The interface to the register file specifies two write enable signals: The first one, wb.wl, is used for GPR and SPR register files and for the low part of the floating point registers. The second, the wb.wh signal, is only used for the high part of the floating point registers. A single precision floating point register is in the low part, iff its address is even (i.e., wb.A[0] is zero). For double precision values, both write enable signals have to be active.

$$\begin{array}{lcl}
\text{wb.l.RF.w} & = & \text{writeback} \wedge \\
 & & ((\text{wb.db} \vee \overline{\text{wb.A}[0]}) \wedge \text{wb.fpr}) \vee \text{wb.gpr} \vee \text{wb.spr}) \\
\\
\text{wb.h.RF.w} & = & \text{writeback} \wedge \\
 & & (\text{wb.db} \vee \text{wb.A}[0]) \wedge \text{wb.fpr}
\end{array}$$

For the producer table, port three is also used as read port to compare the tag with the address of the reorder buffer entry. If they are equal, the valid bit of the register is set.

$$\begin{array}{lcl} \text{wb.l.Prod.w} & = & \text{wb.l.RF.w} \wedge (\text{wb.l.Prod.Dout.tag=ROB.head}) \\ \text{wb.h.Prod.w} & = & \text{wb.h.RF.w} \wedge (\text{wb.h.Prod.Dout.tag=ROB.head}) \end{array}$$

$$\begin{array}{lcl} \text{wb.l.Prod.Din.valid} & = & 1 \\ \text{wb.l.Prod.Din.tag} & = & 0^{\vartheta} \\ \text{wb.h.Prod.Din.valid} & = & 1 \\ \text{wb.h.Prod.Din.tag} & = & 0^{\vartheta} \end{array}$$

Furthermore, during writeback, the IEEEf special purpose register is updated with the value of the IEEEf data item of the ROB entry. This is done by the register file environment (section 3.10).

3.9.3 Completion

During completion, the producer parts of the reservation stations put a result on the CDB. This result has to be written into the ROB. This is done with port nine of the ROB1 RAM. The ROB2 RAM does not contain any values which are to be modified during completion.

The valid flag of the CDB is used as the write enable signal of the write port. The write addres to the ROB RAM is the tag of the instruction on the CDB. The CDB is used as data input to the RAM, since all data items of the ROB1 RAM have corresponding items on the CDB.

3.10 Register File Environment

3.10.1 Register Values

The register file environment contains the different register files, which are the general purpose register file (GPR), the floating point register file (FPR) and the special purpose register file (SPR). All register files have three ports. The ports one and two are read only; they are used by the issue / decode environment for the source operands. Port three is used for writeback by the reorder buffer. This port is a write only port. The Tomasulo scheduling algorithm prevents concurrent read/write accesses of register values on the same address.

The GPR (figure 3.24) consists of 32×32 integer registers (R_0,...,R_{31}). It is implemented as three port 32×32 standard RAM. R_0 is defined to be always zero, which is realized by testing the register address and by pulling the output down in case of R_0.

The FPR (figure 3.25) consists of 32×32 single precision floating point registers (FGR_0,...,FGR_{31}). These registers can also be accessed as 16×64 double precision floating point registers (FPR_0, FPR_2,...,FPR_{30}). The FPR is split into two parts, one for the lower 32 bits and one for the higher 32 bits. It is implemented as two three port 16×32 standard RAMs.

The SPR consists of several registers needed for special purposes such as flags and masks. The SPR registers are listed in table 3.15. The SPR is designed in analogy to the SPR in [MP95], i.e., real registers are used instead of a RAM. The interface to the decode/issue environment is identical to the interface of the GPR. Thus, the SPR environment has three address decoders (two for read, one for write).

Of these three decoders, a maximum of two is used simultaneously, because the DLX instruction set has no instruction with two explicit SPR registers as source. The decoders and the ouput busses for the read ports are in figure 3.26. Based on the values generated by the decoders, signals P1[31:0] to P3[31:0] for the ports one to three are calculated. These signals are used as output enable signals for the drivers of the read ports and as clock enable signals for the write port.

3.10.2 Special Circuits for the SPR

For the SPR register file, several registers have extended access modes. The IEEE standard requires a status flags register for floating point instructions [Ins85, EP97]. It contains a bit for each IEEE exception. Whenever a floating point exception occurs, the flag bit in the IEEEf register is set. Since the IEEE standard requires the IEEE flags to be sticky, the new value is ORed with the old value and re-written into the register (figure 3.27). This only applies to new values from the ROB IEEEf data item, which is only used by floating point instructions. For integer instructions, the IEEEf data item of the ROB is zero. Values written into IEEEf with movi2s are written without modification.

During rfe, only one SPR register has to be modified: The content of the ESR register is copied into the SR register. This is controlled by the DOrfe signal (chapter 3.5.6).

Both JISR and rfe are realized by direct access to the single registers. Figure 3.27 gives the implementation for the SR, ESR, and IEEEf special purpose registers. The implementation of EPC, EPC, ECA, and EDATA is identical to the implementation of ESR. RM and FCC do not require any special circuits.

The registers ECA, SR and EDATA require special circuits. During JISR, the following SPR actions have to be performed:

$$\begin{aligned} \text{ESR} &= \text{SR} \\ \text{ECA} &= \text{MCA} \\ \text{SR} &= 0 \\ \text{EDATA} &= \text{ROB[ROB.head].Edata} \end{aligned}$$

Furthermore, EPC and EPCn are updated by a calculation based on values of the ROB. This calculation is identical to the calculation found in [Lei98]. It ensures that the EPC/EPCn registers hold the PCs of the next two instructions. The calculation is done in dependence of several cases (branch taken/not taken, in delay slot or not). An implementation of this calculation is in figure A.7 (appendix A, page 102).

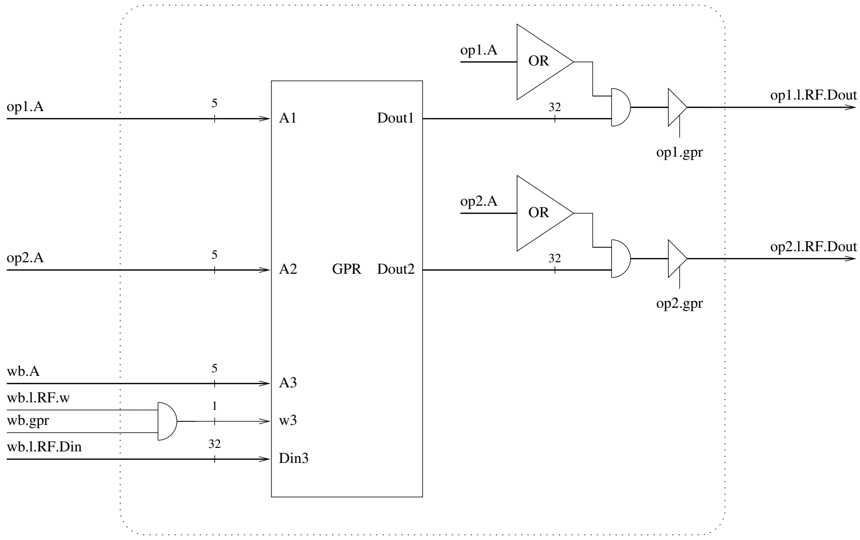


Figure 3.24: The general purpose registers

Nr.	Name	Purpose
0	SR	Status register (interrupt mask)
1	ESR	Exception status register
2	EPC	Exception program counter
3	EPCn	Exception program counter 2
4	ECA	Exception cause register
5	EData	Exception data register
6	RM	Floating point rounding mode
7	IEEEf	IEEE interrupt flags
8	FCC	Floating point comparison flag

Table 3.15: Special purpose registers

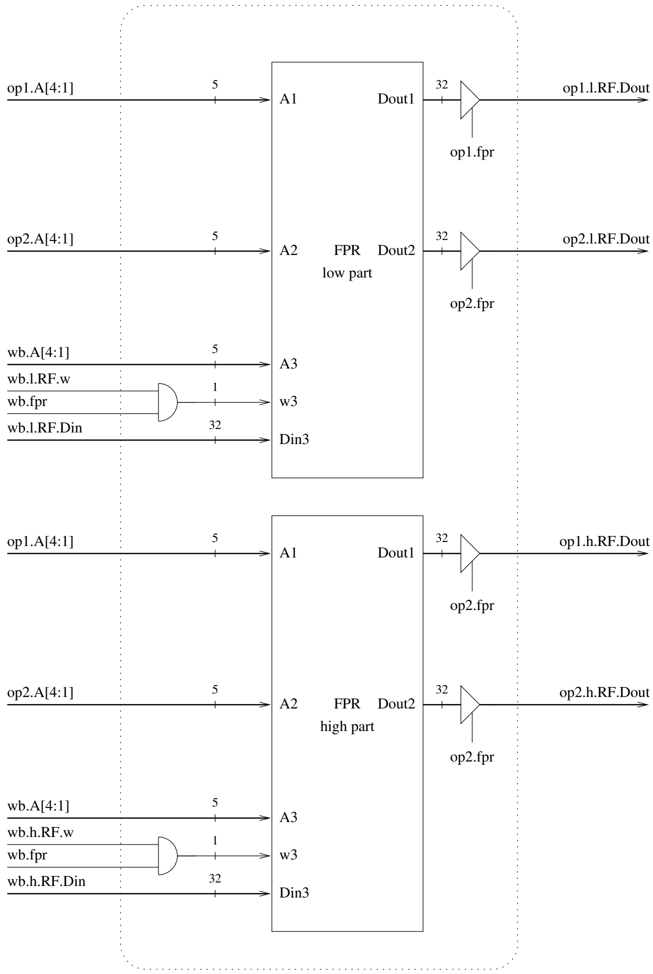


Figure 3.25: The floating point registers

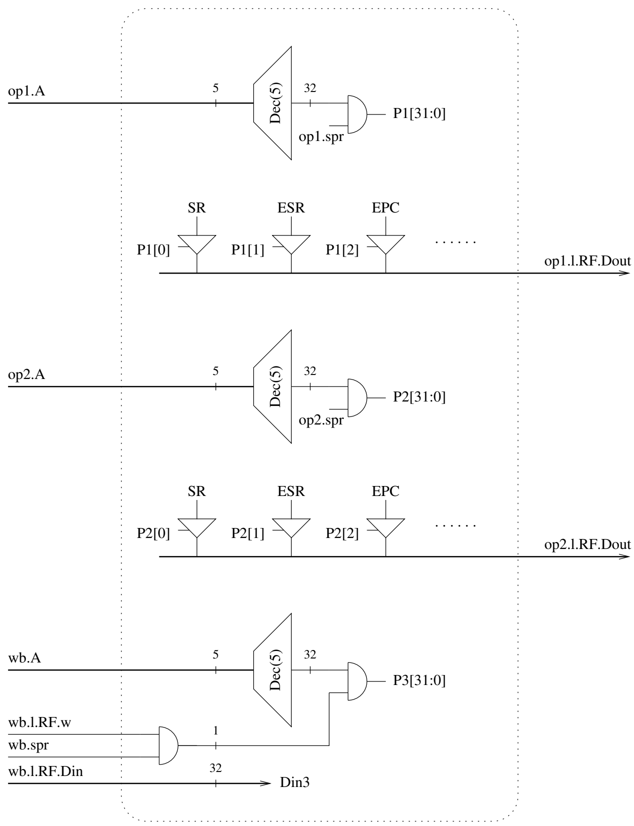


Figure 3.26: The decoders of the special purpose register file

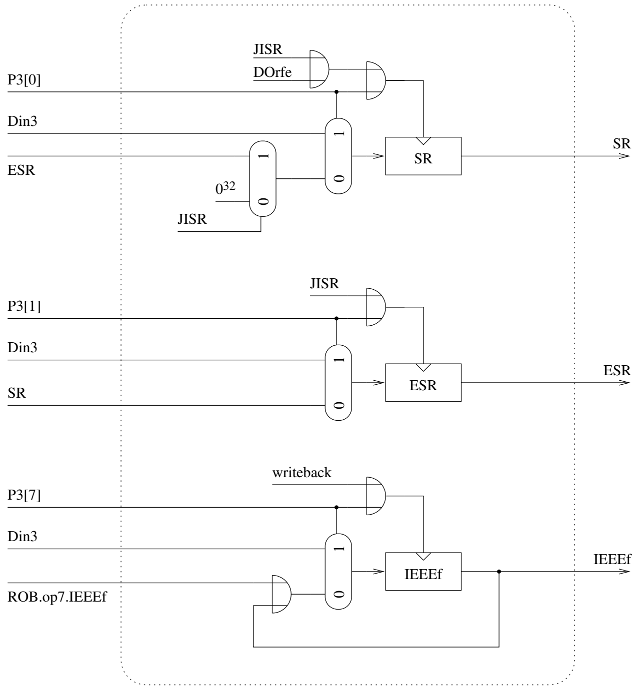


Figure 3.27: Three special purpose registers

3.10.3 Producer Tables

The register file environment also contains the producer tables (storage for valid bits and tags). There is one table for each of the three register files. All producer tables have four ports. The ports one and two are read only; they are used by the decode/issue environment for the two source operands. Port three is used for writeback by the reorder buffer; it is a write only port. Port four is used by decode/issue in order to set the flags for the destination register. In contrast to the register files, all producer tables are made of register based RAM to allow concurrent read/write access to the same address. Furthermore, all producer tables have an input signal init, which sets all valid bits in one cycle if active.

The GPR producer table (figure 3.28) is implemented as four port $32 \times (\vartheta + 1)$ register based RAM. R_0 is defined to be always zero, thus it is required to keep R_0.valid always true to prevent result forwarding from instructions writing into R_0.

The FPR producer table (figure 3.29) is split into two parts, just as the FPR register file. It is implemented as two four port $16 \times (\vartheta + 1)$ register based RAMs.

The SPR producer table (figure 3.30) is similar to the GPR producer table with two exceptions. It does not contain the test for address zero and it has two additional multiplexers in order to realize accesses to the RM and MASK registers during issue of floating point instructions. This saves two extra read ports for operand bus three and four. In case of a floating point instruction, the ports one and two are used for operand bus three and four. Since floating point instructions never read any other special purpose registers, no conflict arises.

Updating of the Producer Tables during Issue

During issue, the valid bit of the destination register has to be cleared and the tag of the instruction has to be stored in the producer table. This is done with RAM port four.

The dest.l.w and dest.h.w signals are the write enable signals of port four of the low and the high memory bank, respectively. The GPR and the SPR only have the low bank. The FPR has a high bank, which is in use while accessing double precision registers or while accessing single precision registers with odd addresses (dest.A[0]=1). The low bank is in use while accessing double precision registers or while accessing single precision registers with even addresses (dest.A[0]=0). Thus:

$$\begin{aligned} \text{dest.l.w} &= \text{/issuestall} \wedge (\text{db} \vee \overline{\text{dest.A[0]}} \vee \overline{\text{dest.fpr}}) \\ \text{dest.h.w} &= \text{/issuestall} \wedge (\text{db} \vee \text{dest.A[0]}) \wedge \text{dest.fpr} \end{aligned}$$

The valid bit written is always zero. The tag bits written are the ROB tail pointer bits provided by the reorder buffer environment.

$$\begin{aligned} \text{Dest.l.Prod.Din.valid} &= 0 \\ \text{Dest.l.Prod.Din.tag} &= \text{ROB.tail} \\ \text{Dest.h.Prod.Din.valid} &= 0 \\ \text{Dest.h.Prod.Din.tag} &= \text{ROB.tail} \end{aligned}$$

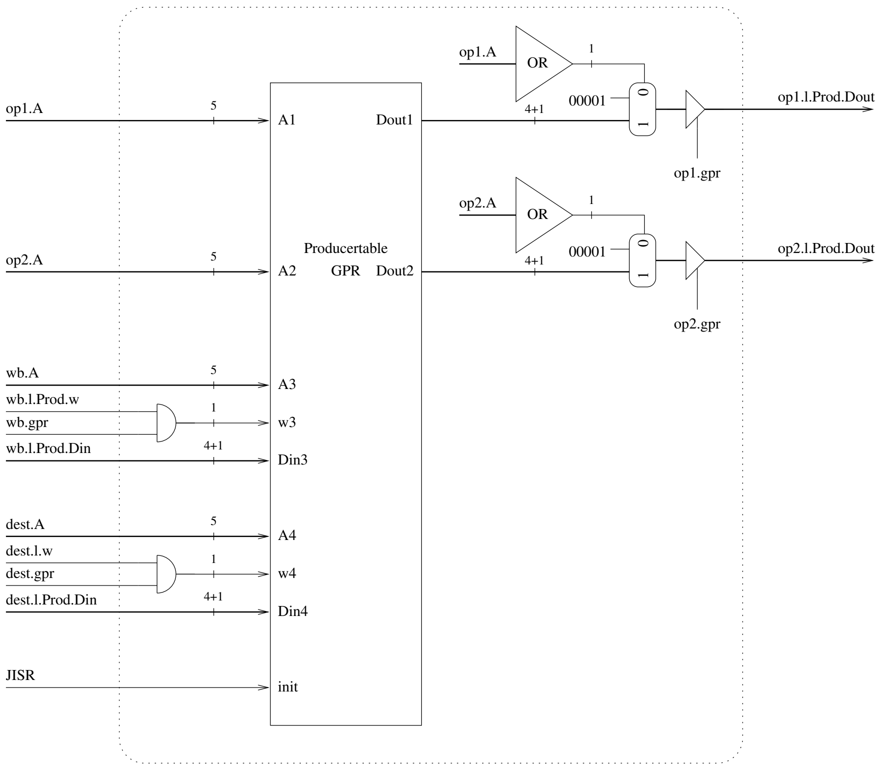


Figure 3.28: The general purpose registers producer table

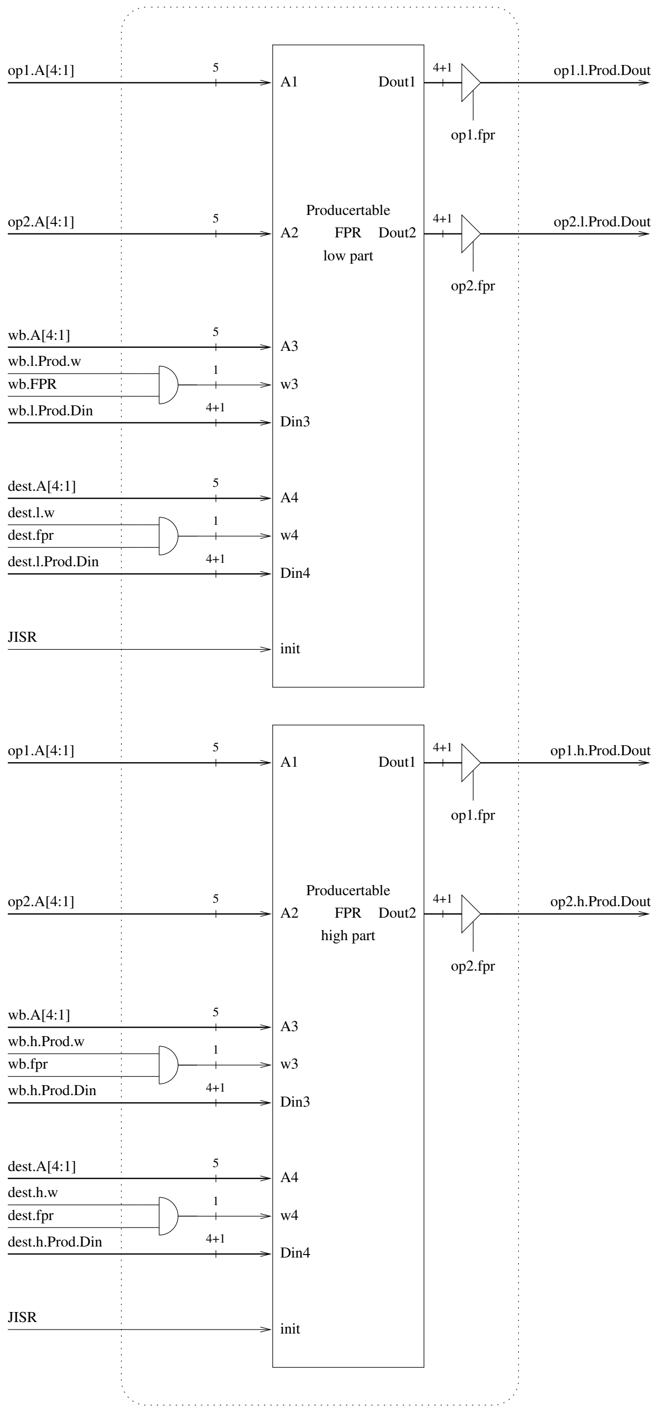


Figure 3.29: The floating point registers producer table

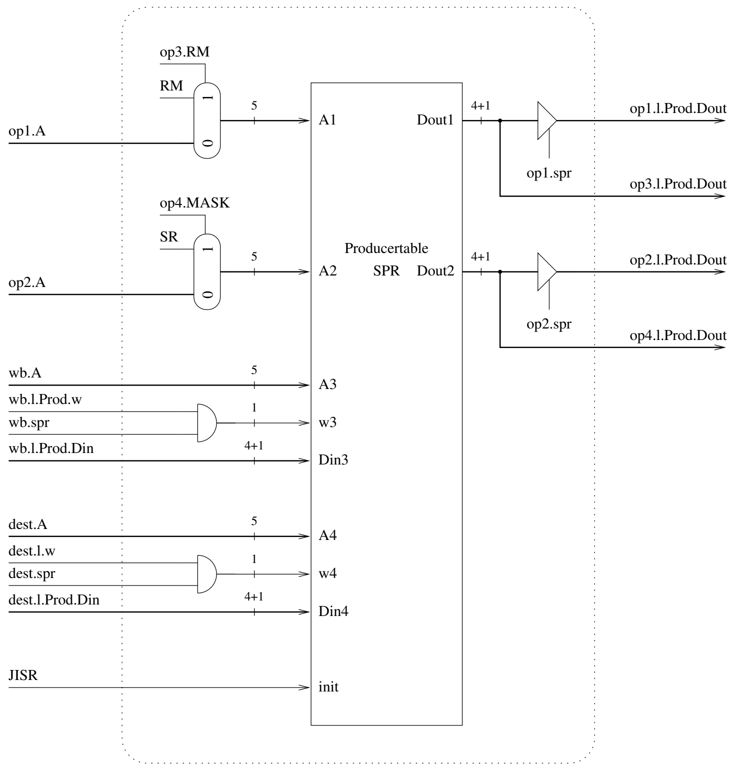


Figure 3.30: The special purpose registers producer table

Chapter 4

Memory System

4.1 Overview of the Data Memory System

The data memory interface is embedded just as an ordinary floating point unit and handles both loads and stores. There are only few exceptions from this rule. Figure 4.1 depicts the complete data memory function unit including the reservation stations.

The address operand of a memory instruction is always a GPR register and it is transported in the low part of operand bus one. The immediate constant (provided by the decode/issue environment), which is used as address offset, is added to the value on this bus before it is stored in reservation station R_0. If the operand is already valid during issue, the sum is the correct memory address. If not so, the decode/issue environment puts zero on the operand bus. In this case, the sum is the immediate constant.

The second operand bus is only used by store instructions and provides the actual value to be stored. The operand busses op3 and op4 are not used by the data memory system.

The instructions and operands provided on these busses are stored in the data memory reservation stations. The data memory reservation stations are described in the next section. During dispatch, one instruction is passed from a reservation station to the single-adjust-one circuit (figure 4.2). This circuit is identical to the single-adjust-one ciruit presented in chapter 3 except that it only modifies operand two. Operand one is always integer. After leaving the single adjust circuit, the instruction is passed to the data memory interface, which contains the actual interface to the data memory or data memory cache. After the memory access, the data memory interface passes the result of the instruction to the single-adjust-two circuit, which is identical to the single-adjust-two circuit in chapter 3. The single-adjust-two circuit passes the result to the producer circuit, which propagates it on the CDB.

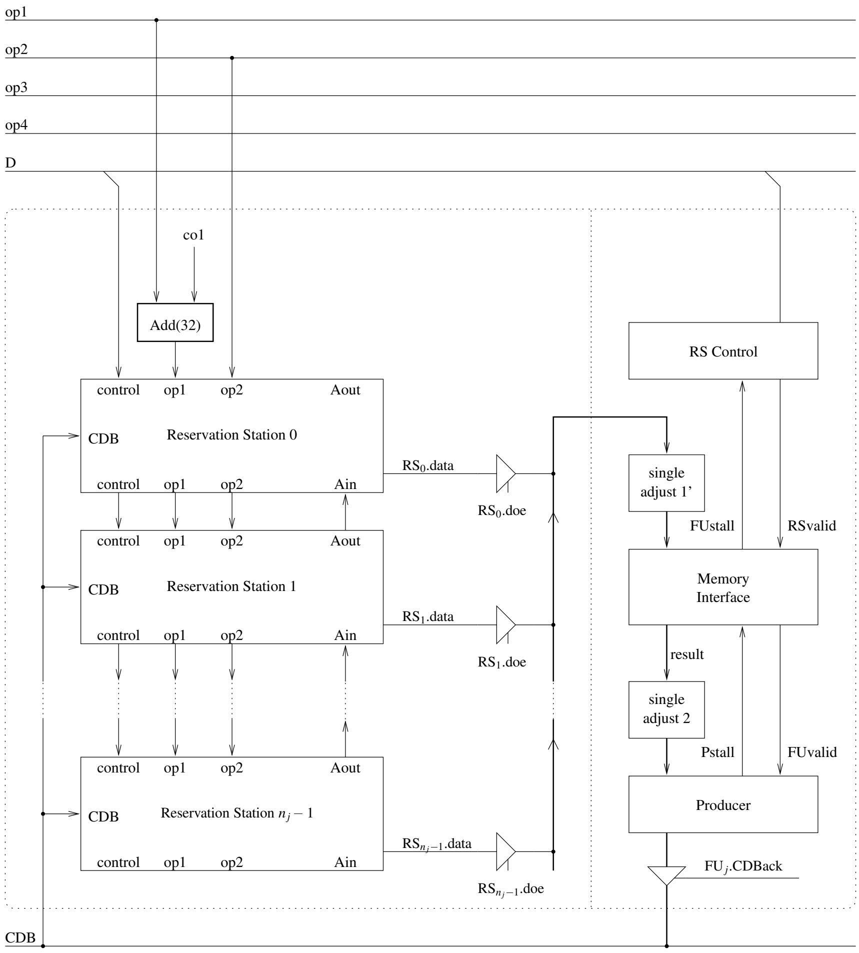


Figure 4.1: The data memory reservation stations

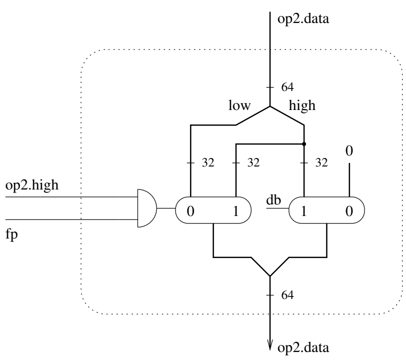


Figure 4.2: Single adjust for the data memory reservation stations

4.2 The Data Memory Reservation Station

Each reservation station (figure 4.3) can hold one load/store instruction and its operands. The reservation station has a register for the full bit, the tag bits and an operation code op. The full bit indicates that the reservation station is in use. The tag data item is the ROB tag of the instruction in the reservation station. The op data item has the following components:

- The op.load data item is one if the instruction is a load, and it is zero otherwise.
- The op.fp data item is active iff the instruction is a floating point load or store.
- The op.db data item is active iff the instruction is a double precision operation.
- The op.op2.high data item is the least significant bit of the address of the data source register on stores, op.dest.high is the least significant bit of the address of the destination register on loads.
- The op.IR[28:26] data items are bits 26 to 28 of the instruction word and are used to determine the width of the memory operand (byte, halfword, word, double).

The first operand in the data memory reservation station is the address operand. This operand is always 32 bits wide. Thus, one reservation station operand is sufficient to store this data. Since the immediate constant has to be added to the address register, this operand requires special circuits (figure 4.4). It is identical to the usual reservation station operand circuit presented in chapter 3 (figure 3.15, page 38) except for the additional adder. This adder calculates the sum of the data on the CDB and the data in the operand register, which is the immediate constant.

The second operand is only used for store instructions. It holds the value to be stored. Since this can be a double precision floating point value, two reservation station operands are required, one for the low and one for the high part. The operand circuit used for operand two is identical to the operand circuit presented in chapter 3 (figure 3.15).

The operation of the reservation stations of the memory system is identical to the operation of the reservation stations presented in chapter 3. However, the dispatch protocol is modified to ensure data integrity.

4.3 Dispatch Protocol

The dispatch protocol used in this design is taken from [Mül97a]. There are four conditions whether to dispatch a store in entry *i*: The first condition ensures that all operands of the entry are valid. The second condition is a test whether the address operands of all preceding instructions are valid. These instructions are in the reservationstations

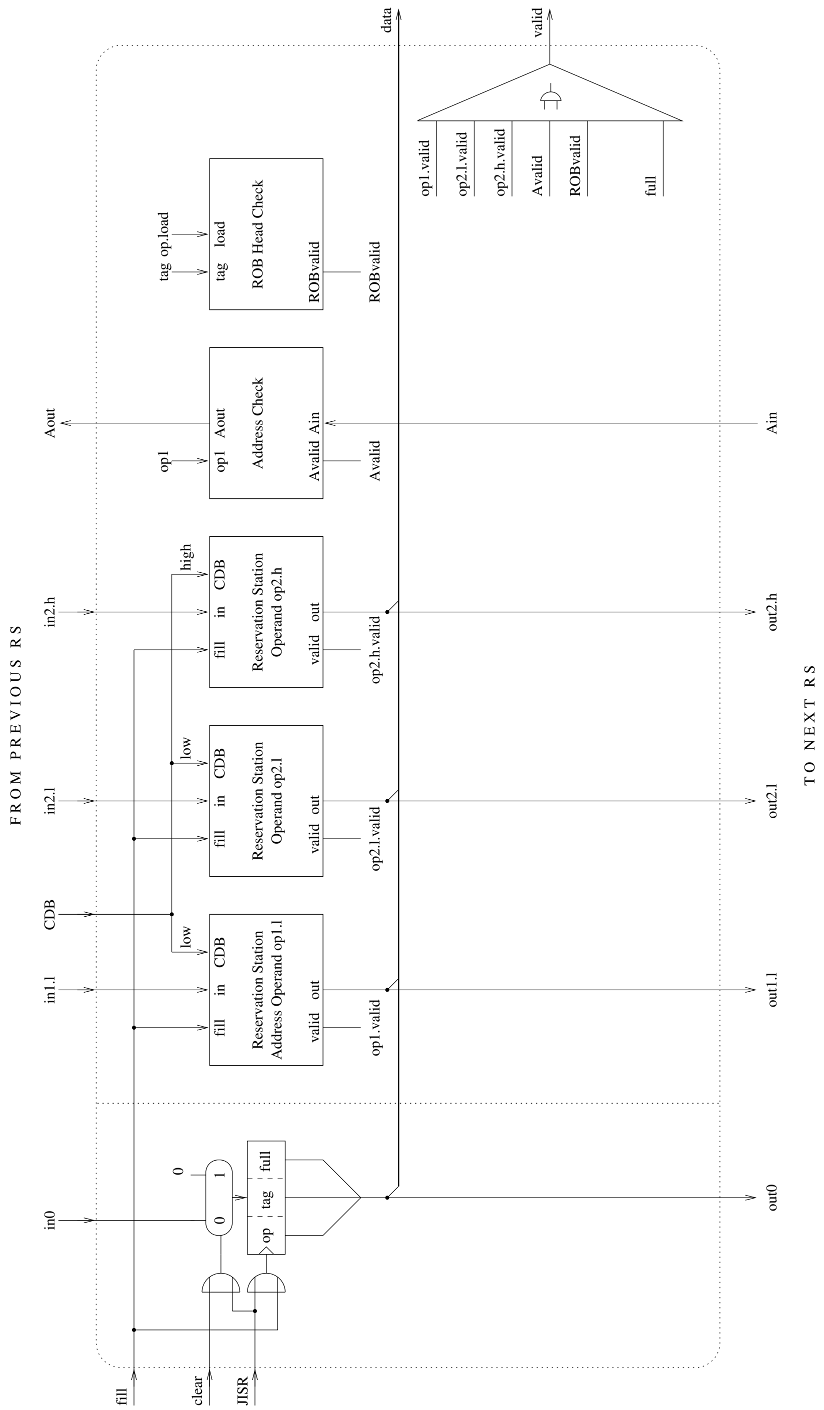


Figure 4.3: A single data memory reservation station

with indices higher that i. The third condition makes sure that the memory operands of the preceding instructions do not overlap with the memory operand of the instruction in the entry to be dispatched. This condition is tested by the overlap(i, j) macro. The value of overlap(i, j) is true if the memory operands in RS_i and RS_j overlap.

Condition four is an extension of the dispatch protocol presented in [Mül97a] and is necessary to realize precise interrupts. Stores must not be executed before all previous instructions have terminated, because any previous instruction might cause an interrupt. This is realized by comparing the ROB.head pointer with the tag stored in the reservation station.

(S1) $\text{RS}_i.\text{op1.valid} = \text{RS}_i.\text{op2.valid} = 1$

(S2) $\forall j > i : \text{RS}_j.\text{op1.valid} = 1$

(S3) $\forall j > i : \overline{\text{overlap}(i, j)}$

(S4) $\text{ROB.head} = \text{RS}_i.\text{tag}$

The conditions for load dispatch are different, because loads do not require in-order execution, therefore, the test for overlapping memory operands is omitted if both instructions are a load. Condition four is omitted, too, because loads do not modify the memory.

(L1) $\text{RS}_i.\text{op1.valid} = \text{RS}_i.\text{op2.valid} = 1$

(L2) $\forall j > i : \text{RS}_j.\text{op1.valid} = 1 \lor \text{RS}_j.\text{op.load}$

(L3) $\forall j > i : \overline{\text{overlap}(i, j)} \lor \text{RS}_j.\text{op.load}$

4.4 Implementation of the Dispatch Protocol

The additional conditions for dispatching a reservation station are implemented in the reservation station itself. The reservation station generates the RS_i.valid signal only iff all operands and all dispatch conditions are valid. The reservation station control of the data memory system is therefore identical to the reservation station control presented in chapter 3.

The condition (1) is tested by the AND-tree in the reservation station (figure 4.3). Conditions (2) and (3) are tested by the address check circuit (figure 4.5). This circuit takes the address operands of all previous reservation stations as input and generates a valid signal named Avalid.

The first step in order to calculate this signal is to define the overlap(i, j) macro (figure 4.6). In the given implementation, only two different memory operand widths are considered, which are 64-bit and 32-bit. Halfword and byte wide operands are

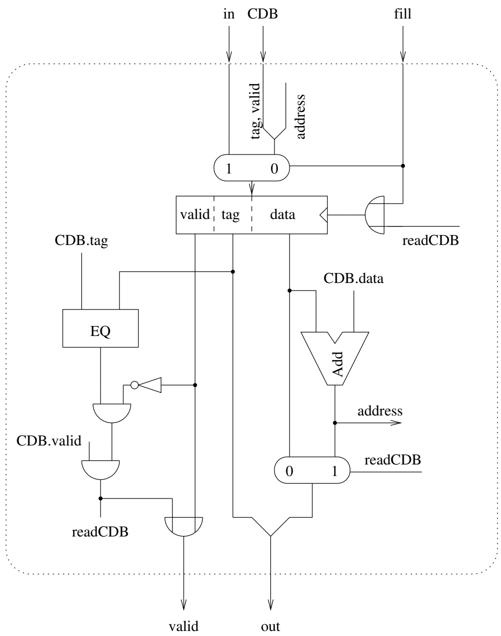


Figure 4.4: The data memory reservation station address operand

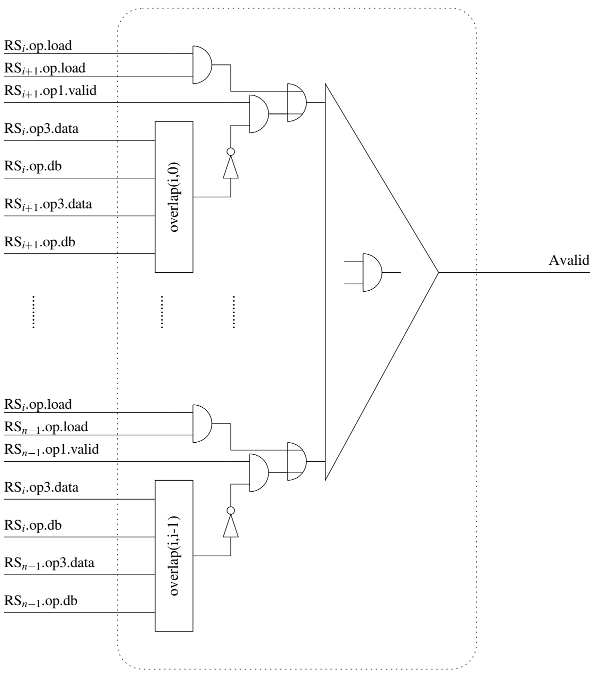


Figure 4.5: The data memory reservation station address comparator operand for reservation station RS$_i$

handled as 32-bit operands. In order to determine this operand with, the macro $\mathrm{DB}(i,j)$ is used. It is true iff at least one of the operands in RS_i or RS_j is a double precision value. The test for overlapping operands is done as follows: In case of single precision values, address bits 2 to 31 are compared. If double precision values are involved, address bits 3 to 31 are compared.

$$\mathrm{DB}(i,j) = \mathrm{RS}_i.\mathrm{op.db} \vee \mathrm{RS}_j.\mathrm{op.db}$$

$$\begin{aligned} \mathrm{overlap}(i,j) \quad &= \quad (\mathrm{RS}_i.\mathrm{op1.data}[31{:}3] = \mathrm{RS}_j.\mathrm{op1.data}[31{:}3]) \wedge \\ & \quad\quad ((\mathrm{RS}_i.\mathrm{op1.data}[2] = \mathrm{RS}_j.\mathrm{op1.data}[2]) \vee /\mathrm{DB}(i,j)) \end{aligned}$$

The $\mathrm{overlap}(i,j)$ macro compares a pair i,j of reservation stations. In order to calculate the Avalid signal, the second step is to apply the overlap macro to all preceding instructions. This step includes a test for loads, which do not require this condition.

$$\mathrm{RS}_i.\mathrm{Avalid} = \bigwedge_{j=i+1}^{n-1} \left((\overline{\mathrm{overlap}(i,j)} \wedge \mathrm{RS}_j.\mathrm{op1.valid}) \vee (\mathrm{RS}_j.\mathrm{load} \wedge \mathrm{RS}_i.\mathrm{load}) \right)$$

Condition (4) is tested as follows: The ROBvalid signal is active, iff condition (4) holds.

$$\mathrm{RS}_i.\mathrm{ROBvalid} = (\mathrm{ROB.head} = \mathrm{RS}_i.\mathrm{tag}) \vee \mathrm{RS}_i.\mathrm{op.load}$$

Futher CPI optimization is possible by implementing load forwarding or write combining on stores. In order to save hardware cost at lower performance, it is possible to perform all memory instructions in program order. The exact performance quantification of both implementations is left for simulations.

4.5 Memory Interface

4.5.1 Control

The memory interface (figure 4.7) is a generic function unit with the usual interface to the reservation stations and the producer as used in chapter 3. It has an additional pipeline stage to save cycle time. The pipeline registers of this stage are placed before the input signals of the data memory. M denotes this register. The output signals of the data memory are almost directly connected to the registers of the producer, thus, there are no critical paths through the data memory.

The memory is accessed by a 64-bit wide data path. In order to store single bytes, halfwords and words, the memory interface uses eight bank write signals mw[7:0]. These bank write signals are calculated by the Mwgen circuit, which is taken from

RS_i:

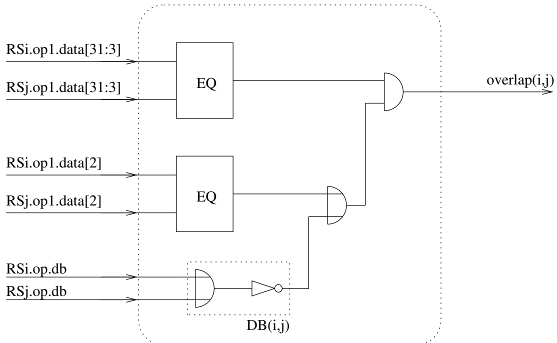


Figure 4.6: The overlap(i, j) macro

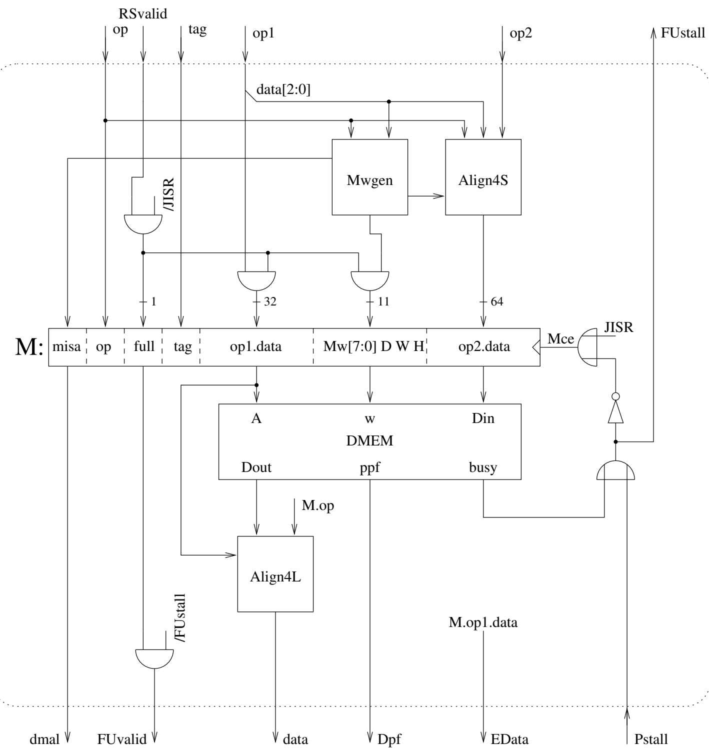


Figure 4.7: Memory interface

[Lei98]. The first step is to determine the exact width of the operand. For this purpose, the bits B (byte), H (halfword), W (word), and D (doubleword) are calculated from bits of the instruction word.

$$
\begin{array}{rcl}
B & = & \overline{IR[27]} \wedge \overline{IR[26]} \\
H & = & \overline{fp} \wedge \overline{IR[27]} \wedge IR[26] \\
W & = & IR[27] \wedge IR[26] \vee fp \wedge \overline{IR[28]} \\
D & = & fp \wedge IR[28]
\end{array}
$$

The bits B[7:0] are derived from the address bits op1.data[2:0] by a decoder. It specifies the offset of the memory operand in an aligned double word. The bank write signals mw[7:0] and the misalignment signal misa are computed as follows:

$$
\begin{array}{rcl}
misa & = & (store \vee load) \wedge \\
 & & (D \wedge (B[0] \vee B[1] \vee B[2]) \vee W \wedge (B[0] \vee B[1]) \vee H \wedge B[0]) \\
mw[0] & = & \overline{misa} \wedge store \wedge B[0] \\
mw[1] & = & \overline{misa} \wedge store \wedge ((B \wedge B[1] \vee H \wedge B[0]) \vee (W \wedge B[0] \vee D \wedge B[0])) \\
mw[2] & = & \overline{misa} \wedge store \wedge ((B \wedge B[2] \vee H \wedge B[2]) \vee (W \wedge B[0] \vee D \wedge B[0])) \\
mw[3] & = & \overline{misa} \wedge store \wedge ((B \wedge B[3] \vee H \wedge B[2]) \vee (W \wedge B[0] \vee D \wedge B[0])) \\
mw[4] & = & \overline{misa} \wedge store \wedge ((B \wedge B[4] \vee H \wedge B[4]) \vee (W \wedge B[4] \vee D \wedge B[0])) \\
mw[5] & = & \overline{misa} \wedge store \wedge ((B \wedge B[5] \vee H \wedge B[4]) \vee (W \wedge B[4] \vee D \wedge B[0])) \\
mw[6] & = & \overline{misa} \wedge store \wedge ((B \wedge B[6] \vee H \wedge B[6]) \vee (W \wedge B[4] \vee D \wedge B[0])) \\
mw[7] & = & \overline{misa} \wedge store \wedge ((B \wedge B[7] \vee H \wedge B[6]) \vee (W \wedge B[4] \vee D \wedge B[0]))
\end{array}
$$

The bank write signals mw[7:0] and the D, W, and H signals are stored in the pipeline register M. The mw[7:0] are fed into the data memory and the D, W, and H signals are used in the alignment circuit for loads.

As mentioned above, the data memory interface is built like any other function unit. This implies that the data memory interface has to generate and respect the flow control signals, which are RSvalid, FUvalid, Pstall, and FUstall. The FUstall signal is active iff the data memory interface is not able to accept further instructions. This is the case if the data memory itself is busy (DMEM.busy=1) or if the producer stalls the function unit (Pstall=1).

$$FUstall = DMEM.busy \vee Pstall$$

The FUvalid signal indicates that the function unit provides a valid result. This is true iff there is an instruction in the register (M.full=1) and if the function unit is not stalled (FUstall=0).

$$FUvalid = M.full \wedge \overline{FUstall}$$

The clock enable signal of the pipeline register of the function unit (Mce) is active if the function unit is not stalled (FUstall=0) or if there is an interrupt (JISR=1). In case of an interrupt, the register is cleared.

$$\text{Mce} = \overline{\text{FUstall}} \vee \text{JISR}$$

4.5.2 Memory Exceptions

The memory system can generate two types of exceptions: page faults and misalignment exceptions. Page faults are used to implement virtual memory. The data memory indicates page faults by raising the pff signal. The data memory system propagates this event on the CDB by enabling the Dpf bit of the CDB. The current memory address in M.op1.data is passed in the EData component of the CDB.

Misaligned memory accesses are indicated by the CDB.dmal signal, which is the misa signal stored in the pipeline register.

4.5.3 Alignment Shifts

The align-for-store and align-for-load boxes take care of correct alignment before a store and after a load, respectively. These circuits are specialized shifters. Figure 4.8 depicts the valid alignment of the memory operands and the corresponding values of the lower address bits A[2:0].

The align-for-load (Align4L) circuit (figure 4.9) performs the alignment shift after a load instruction. The first step is to select the bits of the memory operand from the 64-bit memory bus. This is done by three cascaded multiplexers, which are controlled by the address bits A[2:0]. The first multiplexer selects the correct 32-bit word from the 64-bit bus. The second multiplexer selects the correct 16-bit halfword from the 32-bit word generated by the first multiplexer. The third multiplexer selects the correct byte from this halfword.

The DLX instruction set supports two different types of integer load instructions. Loads of 8 or 16 bits memory operands can be performed with or without sign extension. Loads of 32 or 64 bits values are always done without sign extension. The align-for-load circuit uses a macro $Sext_{n,m}(a,s)$, which is a conditional sign extension of a n-bit value a to m bits if the condition bit s is active. If s is not active, the a is extended to m bits with leading zeros. The circuit is defined in appendix A.2. The condition bit is provided as bit IR[28] in the instruction word. The Align4L circuit returns zero in case of a store instruction in order to have defined values on the CDB.

The align for store (Align4S) circuit (figure 4.10) is much simpler. The operand provided by the single-adjust-one circuit is copied on all valid locations on the 64-bit memory bus. Three multiplexers select the operand with the correct width.

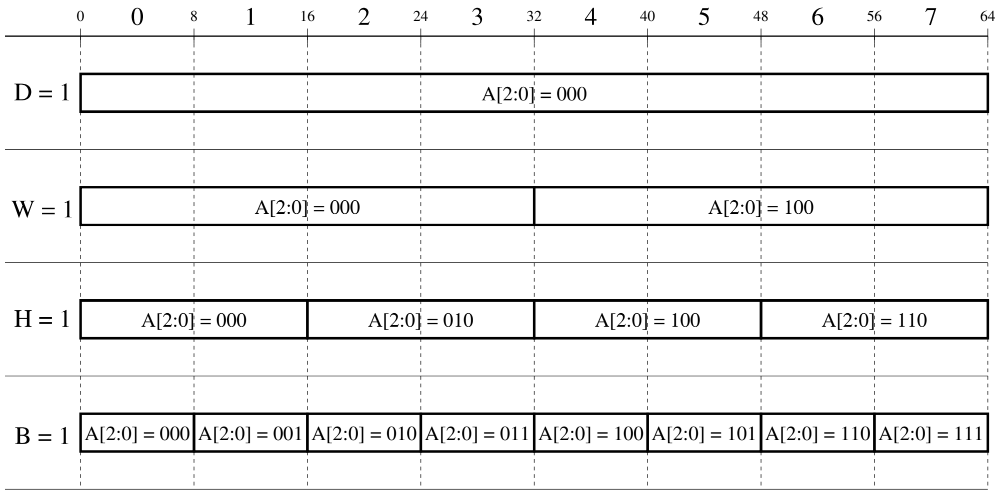


Figure 4.8: Valid alignments

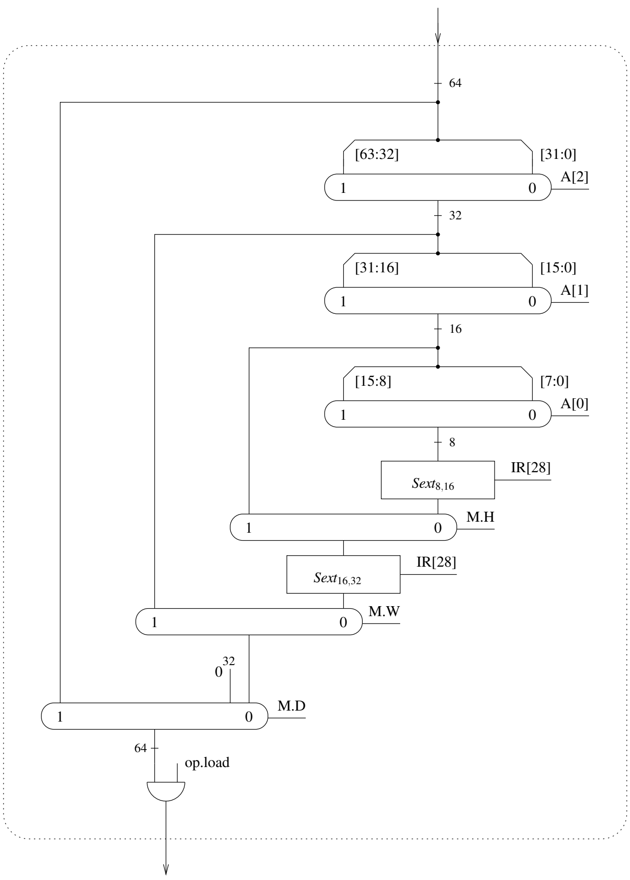


Figure 4.9: Align for load

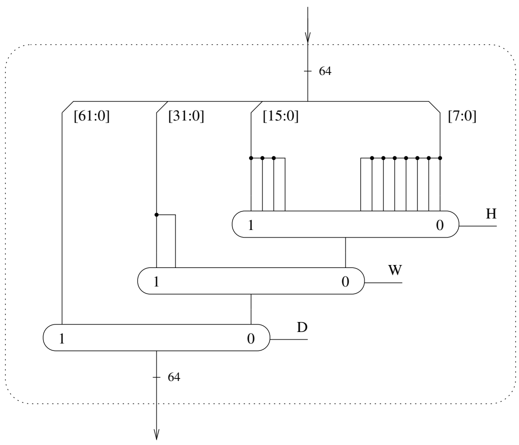


Figure 4.10: Align for store

Chapter 5

Cost and Cycle Time

5.1 Hardware Cost

In spite of progress in miniaturization, hardware cost, e.g., transistor count, is still a crucial matter in CPU design. The hardware cost model used in the following sections is presented in [MP95, KP95] at length. The following sections give just a short summary.

The model does not take wiring into account; only gates are relevant for the hardware cost calculation. Since the Tomasulo algorithm requires several large bus structures, further research on this topic might be of interest. Table 5.1 lists the cost and delay of the basic gates. The values are normalized to the cost and the delay of an inverter.

The overall cost calculation is done by a program, since resolving recurrences is beyond the interest of this thesis. See appendix B for detailed instructions. The detailed cost values in table 5.2 are calculated for a DLX core with Tomasulo scheduler without

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 5.1: Cost and delay of the basic gates

data or instruction cache, and 16 reorder buffer entries. The following sections expose the hardware cost of main circuits in this thesis.

5.1.1 Cost Calculation of the Control Automaton

The cost of the control boolean equations in the decode/issue environment (page 21) are estimated with a model found in [MP95]. The automaton in this thesis does not store the state. Thus, the cirtuits for the next state calculation and the output signals calculation are sufficient.

The calculation of cost and delay depends on a set of parameters. This set is given in table 5.3. The cost of the automata is calculated in three steps: C_M denotes the cost of the calculation of the monomials. With the result of this calculation, the next state is calculated at cost C_N. The state is not stored but depending on it, the output signals are calculated at cost C_O.

$$\begin{aligned} C_M &= \sigma \cdot C_{inv} + (l_{sum} - \#M) \cdot C_{and} \\ C_N &= \text{fanin}_{sum} \cdot (C_{and} + C_{or}) - (k-1) \cdot C_{or} \\ C_O &= (\nu_{sum} - \gamma) \cdot C_{or} \end{aligned}$$

The cost calculation program uses these formulae in order to calculate the cost of the ID1 and ID2 automata in the `opgen()` function.

5.1.2 Reservation Stations and Function Units

With more than 30% of the total cost of the core, the reservation stations are the most expensive data structure in the design, mainly because of the large number of registers and equality testers for each of the six operands. In the data memory reservation stations, the address adders have the biggest cost share. The adders calculate the sum of the memory address operand and the immediate constant co1. The immediate constant is 16 bits wide with sign extension to 32 bits. Thus, the total cost of the data memory reservation station can benefit from a specialized adder, which assumes that the upper 16 input bits of one operand are equal.

Since [Ger98, Del98] state that many reservation stations have a low use in commom benchmarks, the total cost of the reservation stations may be reduced by removing rarely used reservation stations without or with slight impact on the performance. Four reservation station entires are therefore used for the ALU and the data memory and only two for each floating point function unit. Table 5.4 lists cost values for the different assignments. The last entry is the hardware cost with variable number of reservation stations for each function unit as proposed in chapter 3.

In order to save hardware cost within the data memory environment, processing the load/store instructions strictly in-order could save all address adders up to one and

Circuit	Cost	#	Total	%	Figures
two float RS, without FU	9839	2	19678	8.3	3.13 p.34
one float RS, without FU	5600	3	16800	7.1	3.13 p.34
Floating point adder	23735	1	23735	10.1	
Floating point mul/div unit	47557	1	47557	20.2	
Floating point converter	15926	1	15926	6.7	
Floating point transfer	2209	1	2209	0.9	
four integer RS, without FU	7201	1	7201	3.1	3.13 p.34
Integer ALU	3693	1	3693	1.6	A.4 p.100
Data memory environment (four RS)	37846	1	37846	16.0	4.1 p.70
Instruction memory environment	70	1	70	0.0	3.5 p.20
Instruction register environment	158	1	158	0.1	3.6 p.21
PC environment	2252	1	2252	1.0	3.3 p.20
CDB control environment	196	1	196	0.1	3.22 p.51
Decode / issue environment	3742	1	3742	1.6	3.9 p.26
Reorder buffer environment	19807	1	19807	8.4	3.23 p.55
Register files	19545	1	19545	8.3	3.24 p.61
Producer tables	15574	1	15574	6.6	3.28 p.66
Total			235989	100.0	

Table 5.2: Cost of the core components

Symbol	Meaning	ID1	ID2
σ	# input signals	13	3
γ	# output signals	45	1
k	# states	37	2
ζ	$\zeta = \lceil \log k \rceil$	6	1
ν_{max}	maximal frequency of a control signal	21	1
ν_{sum}	accumulated frequency of all control signals	196	1
$\#M$	# monomials, nontrivial	39	4
l_{max}	length of longest monomial	13	2
l_{sum}	accumulated length of all monomials	340	8
fanin_{max}	maximal fanin of $n \neq z_0$	2	8
fanin_{sum}	accumulated fanin	38	8

Table 5.3: Paramenters for the control automata ID1 and ID2

# RS / FU	CPI / speedup		cost without cache		cost with cache	
1	1.6602	0.0%	198076	100.0%	573055	100.0%
2	1.5644	6.1%	229080	115.7%	604059	105.4%
4	1.5161	9.5%	291116	147.0%	666095	116.2%
8	1.4720	12.7%	415216	209.6%	790195	137.9%
var.	∼1.5	10.7%	235989	119.1%	610968	106.6%

Table 5.4: Variations of the number of reservation stations. The last line lists the values for the configuration with a variable number of reservation stations depending on the function unit.

save all the address comparators, since it would be no longer necessary to compare the addresses in the reservation stations.

Another approach to save hardware cost of the reservation stations is to remove operands not worth forwarding, i.e., RM and MASK. This could save up to one third of the total cost of the reservation stations. Section 3.7.2 (page 49) discusses the consequences.

The cost values for the floating point units are taken from [Lei98]. The cost of the floating point converter ist estimated. The cost of the ALU environment is estimated from values found in [MP95] since this environment is almost literally copied. In order to save cycle time, the expensive variant with conditional sum adder is used, since the ALU with carry look-ahead adder would be on the critical path.

5.1.3 Hardware Cost of the ROB and the Register Files

One disadvantage of the reorder buffer is that it requires many RAM ports for forwarding. Thus, the cost impact of adding a large amount of ports to a RAM is significant for the total hardware cost. The cost and delay of a n-bit on chip RAM with A addresses and r read and w write ports is estimated as follows in analogy to [MP95]:

$$\begin{aligned} C_{ram}(A,n,r,w) &= C_{ram}(A,n)\cdot(0.4+0.6\cdot(2w+r)/2) \\ D_{ram}(A,n,r,w) &= D_{ram}(A,n)\cdot(0.5+0.5\cdot(2w+r)/2) \end{aligned}$$

The ROB has many components and reorder buffer with more entries require more tag bits, which increases the cost of the reservation stations. It is therefore advisable to keep the number of entries small. Table 5.5 shows the cost and CPI values for four different ROB sizes. The simulations in [Ger98] did not show that the CPI decreases at bigger ROB sizes. A ROB size of 16 entries therefore seems to be most cost efficient for the given design.

The GPR and FPR is implemented as RAM to save hardware cost. In order to allow cycles from Din to Dout, the SPR and all producer tables are made of register based RAM, which is much more expensive. However, there are only nine special purpose registers, which keeps the cost of the SPR low.

ROB entires	tag bits	CPI / speedup		cost without cache		cost with cache	
16	4	1.4720	0.0%	235989	100.0%	610968	100.0%
32	5	1.5186	-3.1%	254008	107.6%	628987	102.9%
64	6	1.4365	2.4%	288177	122.1%	663156	108.5%
128	7	1.4639	0.6%	354667	150.3%	729646	119.4%

Table 5.5: Effect of ROB size on hardware cost and CPI

	Pipelined		RSR+ROB		Tomasulo	
CPU core only	108949	100%	169701	155%	235989	216%
with 16 kb cache	483928	100%	544680	112%	610968	126%
CPI/speedup	2.12	0%	1.73	22%	1.47	44%

Table 5.6: Cost of core and complete CPU and CPI rates

According to the results in table 5.2, reorder buffer, register files, and producer tables together have about 23 percent of the total core cost.

5.1.4 Caches

The overall cost of the Tomasulo core of about 236k gate equivalents seems to be a huge augmentation compared to the pipelined core (table 5.6). However, any modern CPU design provides data and instruction caches to speed up access to the memory system. These caches are rather large and expensive compared to the actual core. The values in table 5.6 are based on a 16 KB direct mapped cache; they are taken from [MP98]. Comparing both pipelined and Tomasulo architectures with caches shows the cost efficiency of Tomasulo scheduling. At 26 percent higher cost, the design performs 44 percent better. In order to get realistic values, all CPI rates have been simulated with caches.

5.2 Cycle Time

In the given hardware model, the maximum clock frequency of a CPU is determined by the accumulated gate delay of the longest path in the design. This path is determined by a special program. In this program, all data paths are modelled by C++ data structures. Details on this program are contained in appendix B.

For the given design **without floating point units**, the critical path is determined by the program as follows: Figure 5.1 shows the path. It starts in the IR1 register of the decode/issue environment (figure 3.9, page 26). The instruction word is processed by the control automaton ID1 (opgen). After that, the generated values are used to calculate op1.A, the register address of the first operand (figure 3.10, page 27). With

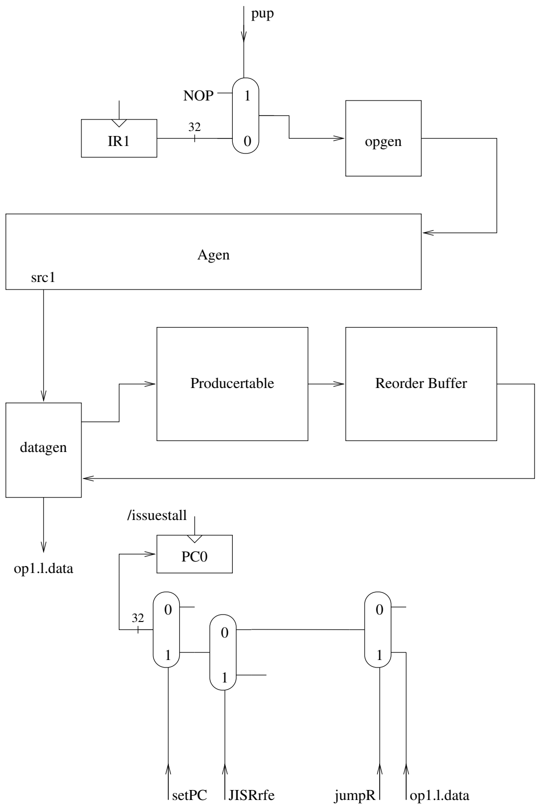


Figure 5.1: The longest path

this address as index, the tag bit of the register is looked up in the producer table. This tag is used as address for the ROB RAM. The output of the ROB is used as input for the top MIX of the datagen circuit (figure 3.11, page 30). The operand determined by datagen is used as input for address computation on branches in the PC environment. The path ends in the PC0 register and has an accumulated delay of 106, which is comparable to other designs, e.g., the pipelined DLX presented in [MP95].

For this calculation, it is assumed that the memory interfaces (instruction and data memory) do not increase the delay. In case of slow memory, it is assumed that appropriate caches are added.

The delay can be easily reduced down to 66 by replacing the ROB RAMs with a register based RAM. However, using a register based RAM of this size increases the hardware cost of the core by 34 percent. Adding the FPU introduces a much longer data path with a length of 137 gate delays within the floating point function units [Lei98]. The overall clock rate of the design is therefore determined by the FPU and not by the scheduler used. However, the design of the FPUs leaves much room for optimization.

```
Longest Path:
  0 DECODE.IR1
 25 opgen
  4 Agen/sourcereg
  0 Prod.opx
  9 Prod.opx.Dout
  0 ROB.Ax
 49 ROB1 RAM
  0 opx.ROB.Dout
  8 operands
  6 new_pc0
  5 pc0
106 TOTAL (11 circuits)
```

Table 5.7: Longest path of the design

5.3 Quality Survey and Comparison

5.3.1 Introduction on Quality Metrics

The quality of a given design depends on the cost and on the performance of the design [Grü94, MP95]. These two values have different weights for certain tasks. For a given task, the hardware cost might be more important than the performance or vice versa. In order to quantify this weight, a quality parameter $q \in [0,1]$ is introduced. With $q = 1$, only cost is relevant for the quality. With $q = 0$, only performance counts and with $q = 0.5$, both parameters have equal weight. Realistic quality weights for cost and performance are values between 0.2 and 0.5.

In order to compare two designs given such a weight, the CPI and the cost C in gate equivalents is used for the quality function. It is assumed that both designs have equal or similar cycle times.

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

For this definition of quality and regarding a fixed quality parameter q, a design A is better than a design B iff $Q_q^A > Q_q^B$, i.e., higher values of Q_q denote better designs.

5.3.2 Comparison

Two other DLX designs are used as reference: The pipelined DLX with precise interrupts and floating point units presented in [MP95, MP98] and a DLX with result shift

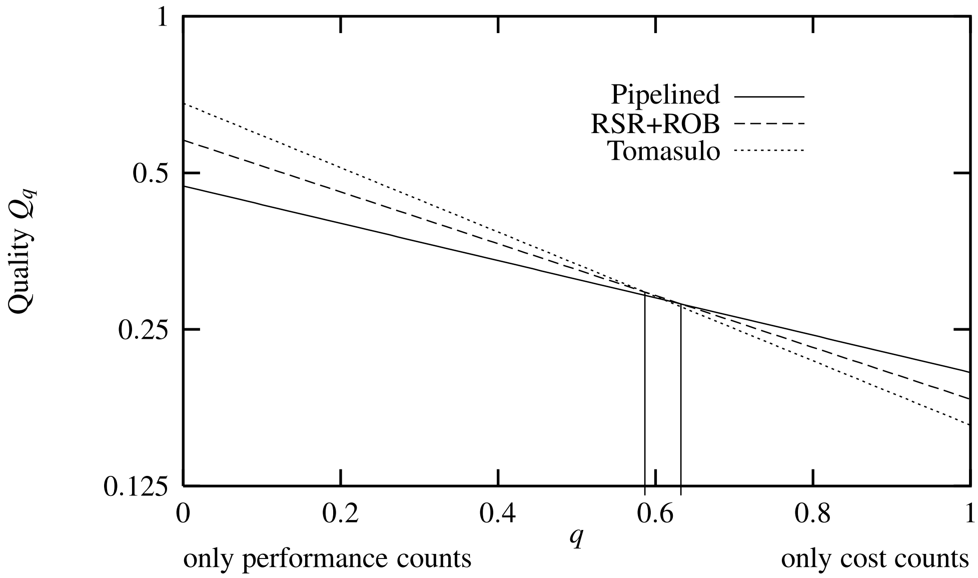


Figure 5.2: Quality for Pipelined DLX, RSR+ROB, and Tomasulo+ROB

register (RSR) and reorder buffer presented in [Lei98]. Table 5.6 contains the cost and CPI values for all three designs. The CPI values are taken from [Ger98, Del98].

Since all designs use similar floating point units, all share the same path in the rounding unit of 137 gate delays. The RSR+ROB DLX in [Lei98] has a critical path with 164 gate delays. Since this path can be easily optimized, the cycle time of the three designs can be assumed to be equal.

Figure 5.2 is a plot of the quality function of these designs in a logarithmic scale. As expected, the pipelined design is always the best design if hardware cost is crucial. The Tomasulo scheduler wins in spite of the high cost if performance counts. The cost and performance of the RSR+ROB design is exactly between both other designs. All three designs have an interval of q, in which their quality Q_q is optimal. The **points of equal quality** (i.e., the value of q with $Q_q^A = Q_q^B$), are marked with two lines in figure 5.2, to show the ranges of q, in which the single designs are optimal.

Chapter 6

Correctness

The correctness proof of the Tomasulo scheduling algorithm with reorder buffer itself is already in [Ger98]. It is based on a proof for the Tomasulo scheduler without reorder buffer in [Mül97a]. The proof presented here uses a slightly different notation. The correctness proof has two parts: The first part shows that data consistency is maintained. The second part proves that the algorithm terminates, i.e., the hardware is deadlock-free.

6.1 Data Consistency

6.1.1 Definitions

Let I_1 to I_n be a sequence of n instructions. Data consistency intuitively means that the results generated by any given instruction I_i in the program conform to the semantics of an abstract DLX machine. These semantics are defined as transition rules on an abstract configuration set $C(i)$ of the machine. This configuration includes all DLX registers, which are $R_0(i)$ to $R_{31}(i)$, SPR, FPR, and $PC(i)$, and the rest of the program to be executed. A shorthand for all register files is $RF[](i)$. The start configuration $C(0)$ is the configuration after the reset.

This abstract DLX machine processes one instruction with each transition. Let the source registers of instruction I_i be $S_1(i)$ to $S_{s(i)}(i)$ with values $\sigma_1(i)$ to $\sigma_{s(i)}(i)$ and the destination registers $D_1(i)$ to $D_{d(i)}(i)$ with values $\delta_1(i)$ to $\delta_{d(i)}(i)$. We use $s(i)$ source and $d(i)$ destination registers to handle double precision floating point instructions without disturbing distinction of cases. For example, the instruction

$$FPR_0{:=}FPR_2{+}FPR_4$$

has six source operands (FGR_2 to FGR_5 and RM and MASK) and three destination operands (FGR_0, FGR_1, and IEEEf). Let op_i be the operation of instruction I_i, this is a double precision floating point addition in the example.

Now, I_i can be specified as

$$\mathrm{I}_i\text{: } (D_1(i),..,D_{d(i)}(i)) = \mathrm{op}_i(S_1(i),..,S_{s(i)}(i))$$

with values

$$\mathrm{I}_i\text{: } (\delta_1(i),..,\delta_{d(i)}(i)) = \mathrm{op}_i(\sigma_1(i),..,\sigma_{s(i)}(i))$$

As mentioned above, the abstract DLX machine processes instruction I_i in transition i, which results in configuration $C(i)$. Registers not written by instruction I_i are passed unmodified from configuration $C(i-1)$. Let $\mathrm{last}(r,i)$ be index of the last instruction prior I_i which modified register r:

$$\mathrm{last}(r,i) = \max\{j < i \mid \mathrm{I}_j \text{ has destination register } r\}$$

This allows an easy criterion for data consistency for any given (non-abstract) machine.

Criterion 1 For any given source register, the value of this register must be the result of the last instruction writing it:

$$\sigma_x(i) = \delta_y(\mathrm{last}(S_x(i),i)) \qquad \forall x \in \{1,..,s(i)\},\, D_y(i) = S_x(i)$$

There are some exceptions from this rule, e.g., the register R_0, which are left out for sake of simplicity. Furthermore, if there is no such instruction, this value is undefined, therefore, it is assumed that all registers, which are used as source register, are initialized with proper values by the program.

The rest of the section will prove that this condition holds for the machine presented in chapter 3. The denominators of the abstract DLX machine for the registers and the values are now used for the Tomasulo DLX configuration. This is done in analogy to the proof in [Mül97a] with the help of four invariants.

Invariant 1 During the issue of instruction I_i, source operand $S_x(i)$ is in the register file, as indicated by RF[r].valid=1. In this case, the data item of the register holds the desired value, i.e., it holds the result of the last instruction writing RF[r].

$$\text{RF}[r]\text{.valid=1} \quad \text{during issue of } \mathrm{I}_i \quad \Longrightarrow \quad \text{RF[r].data} = \delta_y(\mathrm{last}(r,i))$$

Invariant 2 During the issue of instruction I_i, source operand $r := S_x(i)$ is, as indicated by RF[r].valid=0, the result of an previous instruction I_j, which has not yet retired. In this case, the tag item of the register holds I_j.tag, i.e., the tag of the instruction I_j. This instruction is the last instruction writing RF[r].

$$\mathrm{RF}[r].\mathrm{valid{=}0} \quad \text{during issue of } \mathrm{I}_i \quad \Longrightarrow \quad \mathrm{RF}[r].\mathrm{tag} = \mathrm{I}_{\mathrm{last}(r,i)}.\mathrm{tag}$$

Invariant 3 Let instruction I_i be an instruction in reservation station RS_l. If there is an operand x in this reservation station which is not yet valid (RS_l.op$x$.valid=0), there must be a previous instruction I_j, which produces the value for the operand and has not yet completed. The tag of this instruction is the tag in RS_l.op$x$.tag.

$$\mathrm{RS}_l.\mathrm{op}x.\mathrm{valid{=}0} \quad \text{and} \quad \mathrm{I}_i \text{ in } \mathrm{RS}_l \quad \Longrightarrow \quad \mathrm{RS}_l.\mathrm{op}x.\mathrm{tag} = \mathrm{I}_{\mathrm{last}(S_x(i),\,i)}.\mathrm{tag}$$

Invariant 4 Let invariant 2 or 3 apply for I_i and source operand $r := S_x(i)$ with tag $S_x(i)$.tag. If the CDB is valid and if the CDB tag is equal to the tag of the operand, the result of the last instruction writing the operand is on the CDB. If the ROB entry pointed to by the tag of the operand is valid, the result of the last instruction writing the operand is in this ROB entry.

$$\begin{aligned}&\text{Invariant 2 or 3 applies} \quad \wedge \quad (\mathrm{CDB.tag} = S_x(i).\mathrm{tag})\\ &\qquad \Longrightarrow \quad \delta_y(\mathrm{last}(r,i)) \text{ is on the CDB}\end{aligned}$$

$$\begin{aligned}&\text{Invariant 2 or 3 applies} \quad \wedge \quad (\mathrm{ROB}[S_x(i).\mathrm{tag}].\mathrm{valid} = 1)\\ &\qquad \Longrightarrow \quad \delta_y(\mathrm{last}(r,i)) \text{ is in ROB entry } \mathrm{ROB}[S_x(i).\mathrm{tag}]\end{aligned}$$

With the help of these invariants, the proof of the data consistency is done by induction over n. For $n = 0$, the claim is obvious. The induction for $n > 0$ requires a distinction of two cases. Let instruction I_i read source operand $S_x(i)$. This is possible in two different phases.

- Let instruction I_i read register $r{:=}S_x(i)$ **during issue**. In dependence of the value of RF[r].valid, either invariant 1 or invariant 2 applies. If RF[$S_x(i)$].valid is set, the claim is an implication of invariant 1, since the operand is in the register file.

  If not so, invariant 2 states that RF[r] contains the tag of the instruction I_j which produces the result. As the operand of the instruction is already available during issue, it must be either in the ROB or on the CDB. In both cases, invariant 4 applies. Thus, the result in the ROB or on the CDB is $\delta(\mathrm{last}(r,i))$. The instruction I_j had correct operands because of $j < i$.

- Let instruction I_i read register r:=$S_x(i)$ **while in reservation station** RS_l. This only happens if RS_l.op$x$.valid is not set. In this case, invariant 3 states that the tag in RS_l.op$x$.tag is the tag of the instruction I_j which produces $\delta(\mathrm{last}(r,i))$. This tag must be equal to the tag on the CDB, thus invariant four applies. I_i therefore reads the result of I_j. This is the correct result because of $j < i$.

6.2 Termination

6.2.1 Definitions

The termination proof will show that an instruction sequence I_1 to I_n is processed in a finite amount of clock cycles. The following shorthands are used: fetch(i), issue(i), disp(i), compl(i) and term(i) denote the cycles in with instruction I_i is fetched, issued, dispatched, completed and terminated, respectively.

6.2.2 Termination

Lemma 1 All instructions are issued in program order and terminate in program order, i.e., issue(i) < issue(j) and term(i) < term(j) for any $i < j$.

In-order termination is a conclusion of the fact that an instruction terminates when leaving the ROB. The ROB is a fifo queue and is filled in program order. Consequently, to prove termination, it is sufficient to prove that a finite α exists with $\mathrm{term}(i) + \alpha \geq \mathrm{term}(j)$ for any $i < j$, i.e., to find an upper bound for the number of cycles of an instruction. A weak upper bound for α can be determined by summing up the maximum time which an instruction spends in each stage after all previous instructions terminated:

$$\alpha \leq \alpha_0 + \ldots + \alpha_4$$

This bound is prooved by induction over n. The induction hypothesis is that instruction I_i obeys to this bound.

Fetch After all previous instructions terminated, instruction I_i can be fetched if there is no stall from the instruction memory. If $lmem < \infty$ is the maximum latency of the memory, $\alpha_0 = lmem$.

Issue Instruction I_i can be issued iff there is no issue stall (the conditions for an issue stall are in section 3.5.6, page 32). This is obviously true one cycle after all previous instructions terminated and if there is no stall from the instruction memory. Since $lmem$ is the maximum latency of the memory, $\alpha_1 = lmem + 1$.

Dispatch Instruction I_i can be dispatched iff all its operands are valid and if the function unit is able to accept data. Obviously, the operands are valid after all previous instructions terminated. However, the function unit still might be blocked by instructions *after* I_i, i.e, instructions I_j with $j > i$. Let *imax* be the maximum number of instructions the function unit can hold (including the producer pipeline stage), and l the maximum latency of the function unit. For iterative function units, $imax \cdot l$ is a weak bound for the number of cycles until the function unit is empty. This bound is valid for piplined function units, too. However, before an instruction can leave the function unit, it is necessary to wait for the CDB. Since the CDB is allocated round robin, this takes n cycles at most, with n being the number of function units. Thus, it takes $\alpha_2 = imax \cdot l \cdot n$ cycles at most until the function unit is empty. As I_i is the oldest valid instruction in the reservation stations (all previous already terminated), I_i is dispatched afterwards.

Completion As shown above, an instruction I_i in a function unit leaves it (completes) after at most $\alpha_3 = imax \cdot l \cdot n$ cycles.

Termination A valid (completed) instruction in the ROB terminates one cycle after all previous instructions terminated. Thus, α_4 is one cycle.

Chapter 7

Perspective

Several aspects of the Tomasulo scheduler have not been examined in this thesis. They are left for further research.

- Modern CPUs with Tomasulo scheduler issue up to eight instructions in one cycle. Extending the hardware to handle **multiple instruction issue** is subject of a thesis by Mark A. Hillebrand [Zitat].
- The design presented in this thesis performs a stall on each conditional branch until the operand is available. In order to make effective use of a scheduling algorithm, **branch prediction** is required to eliminate these stalls. This is also part of the thesis of Mark A. Hillebrand.
- The Tomasulo scheduling algorithm has built-in support for function units with variable latency. This allows for floating point units sharing expensive components such as the rounder. This results in a slightly lower IPC but great cost savings.

  Further cost savings are possible by removing the forwarding of the rounding mode by encoding it in the instruction word. Another cost-saving opportunity is dropping the forwarding of the interrupt mask. The floating point rounder itself could access the SR (status register) special purpose register directly, performing a stall if not available.
- The present design already supports 64 bit wide floating point operands. The extension to 64 bit wide integer operands is therefore available at very low extra cost. Simulations should show how much improvement in IPC is possible by wider integer operands.
- The hardware model used in this thesis does not take wiring in account. An improved model presented in [PS98] includes the significant impact of wiring on cost and delay.

Appendix A

Auxiliary circuits

A.1 The Find First One Circuit

A.1.1 Purpose

The (unary) n-bit find first one circuit (FFO) and the (unary) n-bit find last one circuit (FLO) calculate the following function:

ffo: $\{0,1\}^n \rightarrow \{0,1\}^{n+1}$, $(a_{n-1},...,a_0) \mapsto (b_{n-1},...,b_0,\text{zero})$, such that

$$b_i = \begin{cases} 1 & : \quad i = \min\{j | a_j = 1 \wedge j \in \{0,...,n-1\}\} \\ 0 & : \quad \text{otherwise} \end{cases}$$

$$\text{zero} = \begin{cases} 1 & : \quad a_i = 0 \text{ for all } i \in \{0,...,n-1\} \\ 0 & : \quad \text{otherwise} \end{cases}$$

flo: $\{0,1\}^n \rightarrow \{0,1\}^{n+1}$, $(a_{n-1},...,a_0) \mapsto (b_{n-1},...,b_0,\text{zero})$, such that

$$b_i = \begin{cases} 1 & : \quad i = \max\{j | a_j = 1 \wedge j \in \{0,...,n-1\}\} \\ 0 & : \quad \text{otherwise} \end{cases}$$

$$\text{zero} = \begin{cases} 1 & : \quad a_i = 0 \text{ for all } i \in \{0,...,n-1\} \\ 0 & : \quad \text{otherwise} \end{cases}$$

This means that the circuit finds the first (last) active signal within a set of signals. If the signals are ascending numbered, the circuit calculates the minimum (maximum) of the set of active signals. This value is returned unary, i.e., by an set of signals as big as the input set. Furthermore, a signal zero is returned, which is set iff the minimum (maximum) is undefined.

It is used by the reservation station control (section 3.6.4, page 37) and by the CDB control (section 3.8, page 49).

A.1.2 Construction

Figure A.1 and A.2 depict a recursive definition of the unary find first one circuit. The cost and delay run at:

$$
\begin{array}{lcl}
C_{ffo}(1) & = & C_{inv} \\
C_{ffo}(n) & = & C_{ffo}(\lceil n/2 \rceil) + C_{ffo}(\lfloor n/2 \rfloor) + C_{inv} + \\
 & & (\lfloor n/2 \rfloor) \cdot C_{and} + C_{and}
\end{array}
$$

$$
\begin{array}{lcl}
D_{ffo}(1) & = & D_{inv} \\
D_{ffo}(n) & = & D_{ffo}(\lceil n/2 \rceil) + D_{inv} + D_{and}
\end{array}
$$

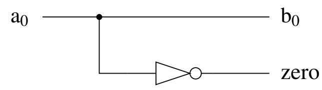


Figure A.1: 1-bit find first one

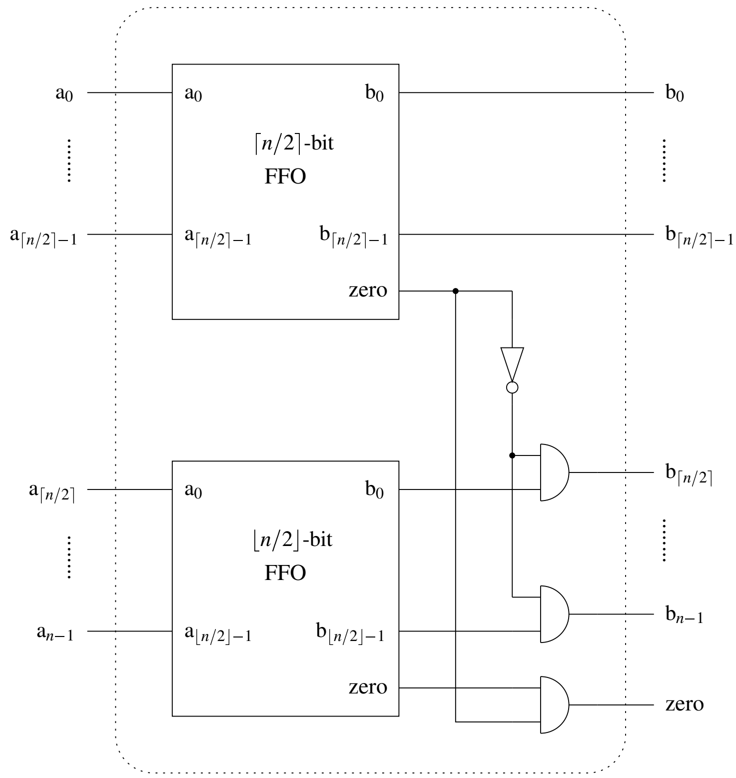


Figure A.2: n-bit find first one

A.2 Conditional Sign Extension

The conditional sign extension circuit is used for the data memory interface and calculates the following function:

$$sext_{n,m}\colon \quad \{0,1\}^{n+1} \to \{0,1\}^m,\ (a_{n-1},...,a_0,s) \mapsto (b_{m-1},...,b_0), \text{ such that}$$

$$b_i = \begin{cases} a_i & : \ i < n \\ a_{n-1} & : \ i \geq n \wedge (s=1) \\ 0 & : \ \text{otherwise} \end{cases}$$

This means that $sext_{n,m}(a,0)$ extends an unsigned n-bit integer value a to m bits, whereas $sext_{n,m}(a,1)$ extends a signed n-bit integer value a to m bits.

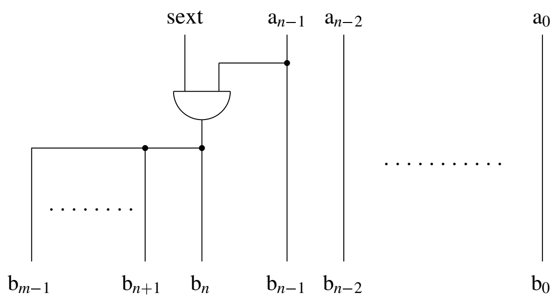


Figure A.3: $Sext(n,m)$

A.3 The Integer Function Unit

The integer function unit (figure A.4) handles all integer operations, which are the ALU functions and shifting instructions. Table 3.10. page 48 specifies the interface. The function unit consists of the ALU and shifter environment presented in [MP95] with one small modification: The overflow bit ovf, which is generated by the ALU, is masked, since it is only used for two instructions. The mask bit is calculated by the ovfmask circuit, which implements the following function:

$$\text{ovfmask} = \text{op}[4] \wedge \overline{\text{op}[3]} \wedge \overline{\text{op}[2]} \wedge \overline{\text{op}[0]}$$

A.4 ROB Auxiliary Circuits

The ROB requires three auxiliary circuits, which calculate the ROB.full and ROB.empty signals and provide the ROB pointers ROB.head and ROB.tail. The circuits are literally taken from [Lei98].

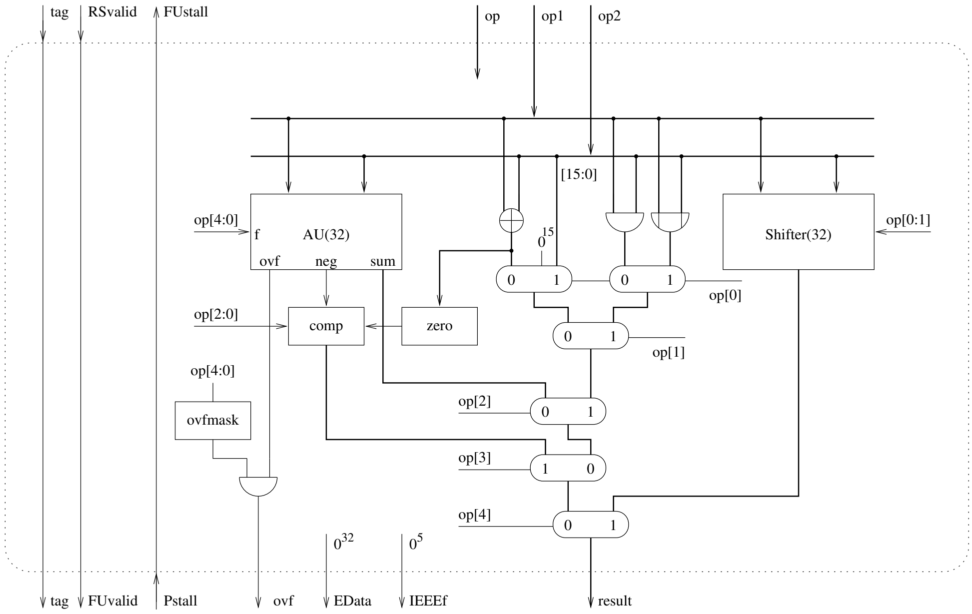


Figure A.4: Integer function unit environment

The ROB pointers ROB.head and ROB.tail are stored in registers (figure A.5). A pointer register is incremented by one (modulo 16, the ROB size), if ROB.tailce or ROB.headce is active, respectively. In case of an interrupt, the JISR signal causes that a zero is clocked into the counter register.

The ROB flags circuit (figure A.6) provides the ROB.full and ROB.empty signals. The ROB.full signal is active, iff the reorder buffer is full, the ROB.empty signal is active iff the reorder buffer is empty. In both cases, the values of the pointer registers are equal, a comparator therefore is not suited to calculate the signals. A counter register, which holds the current number of instructions in the ROB, is used instead. The counter is updated if ROB.tailce or ROB.headce is active:

$$\text{ROB.countce} = \text{ROB.tailce} \vee \text{ROB.headce}$$

A.5 Calculation of EPC/EPCn

During retire, the instruction in the ROB is checked for interrupts. In case of an interupt, a jump to the interrupt service routine (JISR) is performed. During JISR, the EPC and EPCn special purpose registers are filled with values calculated from values from the ROB. The EPC and EPCn registers hold the addresses of the two PC at which the program is to be resumed after processing the interrupt. The circuit for this calculation (figure A.7) is identical to the circuit presented in [Lei98].

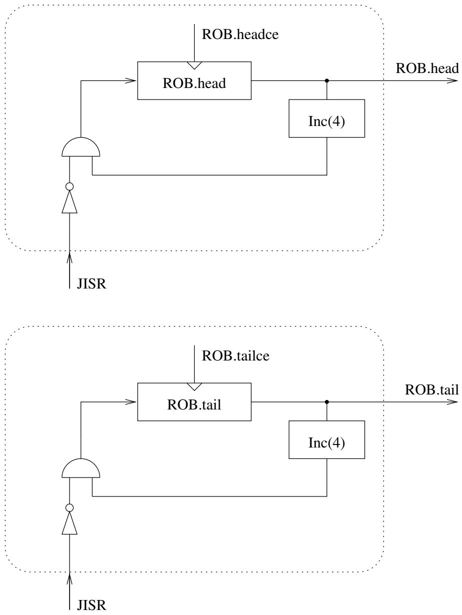


Figure A.5: Reorder buffer pointer

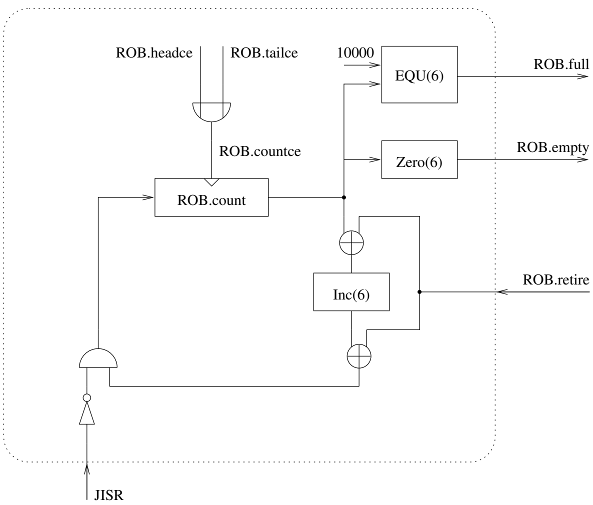


Figure A.6: Reorder buffer flags

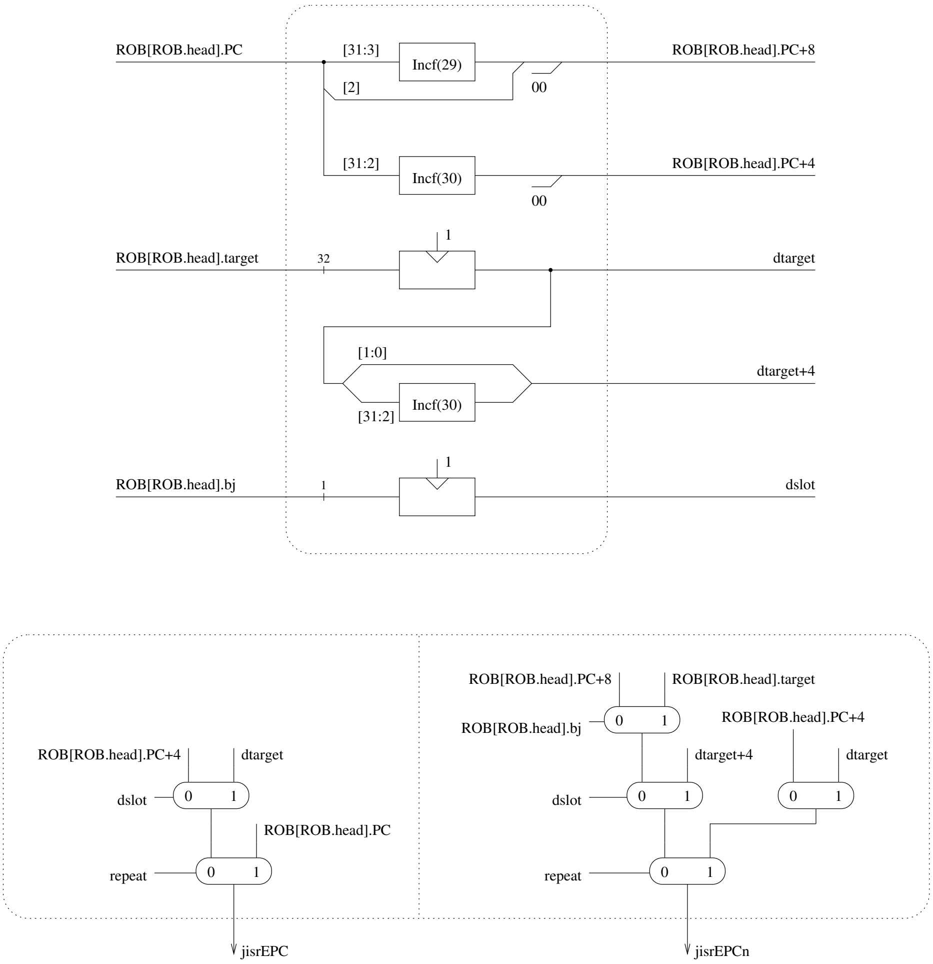


Figure A.7: EPC/EPNc computation

Appendix B

The Cost and Delay Calculation Programs

In order to calculate the hardware cost and the longest path of the design, two C++ programs are used. The following sections describe the structure of the programs and give usage notes. The programs themselves are not printed here because of their length. The programs should run in any C++/32-bit environment. Both programs use a library presented in [MP95], which provides functions for the cost and delay of the basic gates and simple ciruits such as adders, RAMs, and binary trees of gates. A GNU makefile is supplied to make compilation simple. The programs are available via WWW at the address `http://www-wjp.cs.uni-sb.de/~kroening/tomasulo/`.

B.1 The Hardware Cost Calculation Program

B.1.1 Usage

The hardware cost in gate equivalents is calculated by the cost program. The program takes several command line arguments, which are given in table B.1. By default, the program prints table 5.2 in LaTeX syntax. Other options allow printing of the ROB and reservation station comparison tables.

In order to reduce the delay, the design offers two optimization tricks, which also affect the hardware cost. The ROB is the critical component on the longest path, both tricks therefore modify the path through the ROB. Option `--trick1` enables the use of two ROB1 RAMs, one RAM with four read and two write ports, and another with three read and two write ports. Thus, seven read ports are available altogether. Option `--trick2` controls the type of the ROB RAM. The ROB RAM is replaced by register based RAM if this option is set. Both options can be combined, but this does not result in further improvement of the delay.

Parameter	Purpose
`--addcache`	Add a 16 kb direct mapped cache
`--onlysum`	Only print cost total
`--robtable`	Print the ROB size comparison table
`--rstable`	Print the reservation station comparison table
`--comptable`	Print the comparison table for the three designs
`--tag=n`	Set the number of tag bits to n
`--trick1`	Enable optimization trick1
`--trick2`	Enable optimization trick2

Table B.1: Command line arguments of the cost calculation program

Parameter	Purpose
`--tag=n`	Set the number of tag bits to n
`--trick1`	Enable optimization trick1
`--trick2`	Enable optimization trick2

Table B.2: Command line arguments of the delay calculation program

B.1.2 Implementation

The implementation of the cost calculation program is similar to the implementation of the cost calculation programs in [MP95]. For each environment or circuit, a function is defined, which returns its cost in gate equivalents in dependence of certain parameters, e.g., tag size. The main function just calculates the sum of these figures.

B.2 The Delay Calculation Program

B.2.1 Usage

The delay calculation program finds the longest path of the design. The program takes several command line arguments, which are given in table B.2. By default, the program prints all components on the longest path, their delay and the accumulated delay of all components. In order to reduce the delay, the design offers two optimization tricks. The tricks and the corresponding command line parameters have been described above.

B.2.2 Implementation

The implementation of the delay program is slightly different from the programs in [MP95]. The program determines the delay of the longest path and also prints all components on this path. In order to realize this, the program uses a C++ class library which implements an abstract data type `patht` which is used as datapath. Data paths start in a register or are provided as external signal. For example, the power up signal pup is defined as follows:

```
patht pup("pup");
```

The string argument is the name of the source of the data path, as used for the printout. A data path can be extended by further components just by adding them:

```
patht inv_pup=pup+cinv;
```

This defines a new data path called inv_pup (inverted power up). The new signal is calculated from pup with the delay of an inverter. In order to allow correlation of gate delays to components of the design, it is possible to add a string to a data path. This string is the name of all gates, which have been added since the last component. The example above is now:

```
patht inv_pup=pup+cinv+"inverted power up";
```

Most gates and circuits use more than one data path as input. The accumulated delay of the new data path is the maximum of the accumulated delays of all input data paths. This can be expressed as follows:

```
patht new_path=max(path1, path2, ....)+circuit_delay;
```

The `max` function is defined for up to four arguments. However, the source code of the data path library can be extended to any given number of arguments with ease. As soon as a data path ends in a register, it is stored in a list of data paths. After all data paths in the design have been added to that list, the `max` function of the pathlist is called. It returns the data path with the maximum accumulated delay.

Appendix C

The DLX Instruction Set

This instruction set is taken from [MP95, MP98] with minimal modifications.

C.1 Instruction Formats

	6	5	5	16
I-type	Opcode	RS1	RD	Immediate

	6	5	5	5	5	6
R-type	Opcode	RS1	RS2	RD	SA	Function

	6	26
J-type	Opcode	PC Offset

	6	5	5	16
FI-type	Opcode	RS1	FD	Immediate

	6	5	5	5		3	6
FR-type	Opcode	FS1	FS2/RS2	FD	00	Fmt	Function

Figure C.1: Instruction formats of the DLX

C.2 Instruction Set Coding

IR[31 : 26]		Mnemonic	d	Effect
Data Transfer, mem = M[RS1 + Sext(imm)]				
100000	0x20	lb	1	RD = Sext(mem)
100001	0x21	lh	2	RD = Sext(mem)
100011	0x23	lw	4	RD = mem
100100	0x24	lbu	1	RD = 0^{24}mem
100101	0x25	lhu	2	RD = 0^{16}mem
101000	0x28	sb	1	mem = RD[7 : 0]
101001	0x29	sh	2	mem = RD[15 : 0]
101011	0x2b	sw	4	mem = RD
Arithmetic, Logical Operation				
001000	0x08	addi		RD = RS1 + Sext(imm)
001001	0x09	addiu		RD = RS1 + Sext(imm) (no overflow)
001010	0x10	subi		RD = RS1 - Sext(imm)
001011	0x11	subiu		RD = RS1 - Sext(imm) (no overflow)
001100	0x12	andi		RD = RS1 $\wedge$ Sext(imm)
001101	0x13	ori		RD = RS1 $\vee$ Sext(imm)
001110	0x14	xori		RD = RS1 $\oplus$ Sext(imm)
001111	0x15	lhgi		RD = imm0^{16}
Test Set Operation				
011000	0x18	clri		RD = (false ? 1 : 0)
011001	0x19	sgri		RD = (RS1 $>$ Sext(imm) ? 1 : 0)
011010	0x1a	seqi		RD = (RS1 $=$ Sext(imm) ? 1 : 0)
011011	0x1b	sgei		RD = (RS1 $\geq$ Sext(imm) ? 1 : 0)
011100	0x1c	slsi		RD = (RS1 $<$ Sext(imm) ? 1 : 0)
011101	0x1d	snei		RD = (RS1 $\neq$ Sext(imm) ? 1 : 0)
011110	0x1e	slei		RD = (RS1 $\leq$ Sext(imm) ? 1 : 0)
011111	0x1f	seti		RD = (true ? 1 : 0)
Control Operation				
000100	0x04	beqz		PC = PC + 4 + (RS1 $=$ 0 ? Sext(imm): 0)
000101	0x05	bnez		PC = PC + 4 + (RS1 $\neq$ 0 ? Sext(imm): 0)
000110	0x16	jr		PC = RS1
000111	0x17	jalr		R31 = PC + 4; PC = RS1

Table C.1: I-type instruction layout

IR[31 : 26]		IR[5 : 0]		Mnemonic	Effect
Shift Operation					
000000	0x00	000000	0x00	slli	RD = RS1 << SA
000000	0x00	000001	0x01	slai	RD = RS1 << SA (arith.)
000000	0x00	000010	0x02	srli	RD = RS1 >> SA
000000	0x00	000011	0x03	srai	RD = RS1 >> SA (arith.)
000000	0x00	000100	0x04	sll	RD = RS1 << RS2[4 : 0]
000000	0x00	000101	0x05	sla	RD = RS1 << RS2[4 : 0] (arith.)
000000	0x00	000110	0x06	srl	RD = RS1 >> RS2[4 : 0]
000000	0x00	000111	0x07	sra	RD = RS1 >> RS2[4 : 0] (arith.)
Data Transfer					
000000	0x00	010000	0x10	movs2i	RD = SA
000000	0x00	010001	0x11	movi2s	SA = RS1
Arithmetic, Logical Operation					
000000	0x00	100000	0x20	add	RD = RS1 + RS2
000000	0x00	100001	0x21	addu	RD = RS1 + RS2 (no overflow)
000000	0x00	100010	0x22	sub	RD = RS1 - RS2
000000	0x00	100011	0x23	subu	RD = RS1 - RS2 (no overflow)
000000	0x00	100100	0x24	and	RD = RS1 $\wedge$ RS2
000000	0x00	100101	0x25	or	RD = RS1 $\vee$ RS2
000000	0x00	100110	0x26	xor	RD = RS1 $\oplus$ RS2
000000	0x00	100111	0x27	lhg	RD = RS2[15:0] 0^{16}
Test Set Operation					
000000	0x00	101000	0x28	clr	RD = (false ? 1 : 0)
000000	0x00	101001	0x29	sgr	RD = (RS1 $>$ RS2 ? 1 : 0)
000000	0x00	101010	0x2a	seq	RD = (RS1 $=$ RS2 ? 1 : 0)
000000	0x00	101011	0x2b	sge	RD = (RS1 $\geq$ RS2 ? 1 : 0)
000000	0x00	101100	0x2c	sls	RD = (RS1 $<$ RS2 ? 1 : 0)
000000	0x00	101101	0x2d	sne	RD = (RS1 $\neq$ RS2 ? 1 : 0)
000000	0x00	101110	0x2e	sle	RD = (RS1 $\leq$ RS2 ? 1 : 0)
000000	0x00	101111	0x2f	set	RD = (true ? 1 : 0)

Table C.2: R-type instruction layout

IR[31 : 26]		Mnemonic	Effect
Control Operation			
000010	0x02	j	PC = PC + 4 + Sext(imm)
000011	0x03	jal	R31 = PC + 4; PC = PC + 4 + Sext(imm)
111110	0x3e	trap	trap = 1; EPC = PC; PC = SISR; ESR = SR; ECA = masked CA; SR = 0; EDATA = Sext(imm); clear CA but catch new interrupt events
111111	0x3f	rfe	SR = ESR; PC = EPC

Table C.3: J-type instruction layout

IR[31 : 26]		Mnemonic	d	Effect
Load, Store				
110001	0x31	load.s	4	FD[31 : 0] = mem
110101	0x35	load.d	8	FD[63 : 0] = mem
111001	0x39	store.s	4	m = FD[31 : 0]
111101	0x3d	store.d	8	m = FD[63 : 0]
Control Operation				
000110	0x06	fbeqz		PC = PC + 4 + (FCC = 0 ? Sext(imm) : 0)
000111	0x07	fbnez		PC = PC + 4 + (FCC $\neq$ 0 ? Sext(imm) : 0)

Table C.4: FI-type instruction layout

IR[31 : 26]		IR[5 : 0]		Fmt	Mnemonic	Effect
Arithmetic and Compare Operations						
010001	0x11	000000	0x00		fadd [.s, .d]	FD = FS1 + FS2
010001	0x11	000001	0x01		fsub [.s, .d]	FD = FS1 - FS2
010001	0x11	000010	0x02		fmul [.s, .d]	FD = FS1 * FS2
010001	0x11	000011	0x03		fdiv [.s, .d]	FD = FS1 / FS2
010001	0x11	000100	0x04		fneg [.s, .d]	FD = - FS1
010001	0x11	000101	0x05		fabs [.s, .d]	FD = abs(FS1)
010001	0x11	000110	0x06		fsqt [.s, .d]	FD = sqrt(FS1)
010001	0x11	000111	0x07		frem [.s, .d]	FD = rem(FS1, FS2)
010001	0x11	$11c_3c_2c_1c_0$			fc.cond [.s, .d]	FCC = (FS1 *cond* FS2)
Data Transfer						
010001	0x11	001000	0x08	000	fmov.s	FD[31 : 0] = FS1[31 : 0]
010001	0x11	001000	0x08	001	fmov.d	FD[63 : 0] = FS1[63 : 0]
010001	0x11	001001	0x09		mf2i	RS = FS1[31 : 0]
010001	0x11	001010	0x0a		mi2f	FD[31 : 0] = RS
Conversion						
010001	0x11	100000	0x20	001	cvt.s.d	FD = cvt(FS1, s, d)
010001	0x11	100000	0x20	100	cvt.s.i	FD = cvt(FS1, s, i)
010001	0x11	100001	0x21	000	cvt.d.s	FD = cvt(FS1, d, s)
010001	0x11	100001	0x21	100	cvt.d.i	FD = cvt(FS1, d, i)
010001	0x11	100100	0x24	000	cvt.i.s	FD = cvt(FS1, i, s)
010001	0x11	100100	0x24	001	cvt.i.d	FD = cvt(FS1, i, d)

Table C.5: FR-type instruction layout. Fmt=IR[8:6]

RM	Symbol	Rounding Mode
00	RZ	toward zero
01	RNE	to next even
10	RPI	toward $+\infty$
11	RMI	toward $-\infty$

Bit	Symbol	Purpose
0	OVF	overflow
1	UNF	underflow
2	INX	inexact result
3	DBZ	divide by zero
4	INV	invalid operation

Table C.6: Coding of the rounding mode RM and the interrupt flags IEEEf

Bibliography

[CS95] Robert P. Colwell and Randy L. Steck. A 0.6um bicmos processor employing dynamic execution. International Solid State Circuits Conference (ISSCC), 1995.

[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, 1998.

[EP97] G. Even and W.J. Paul. On the design of IEEE compliant floating point units. In *Proc. 13th IEEE Symposium on Computer Arithmetic*, pages 54–63. IEEE Computer Society, 1997.

[Ger98] Nikolaus D. Gerteis. Die Auswirkung von Mechanismen für die präzise Interruptbehandlung auf den Cyclecount von RISC-Prozessoren. Master's thesis, Universität des Saarlandes, Fachbereich 14 Informatik, 1998.

[Grü94] Thomas Grün. *Quantitative Analyse von I/O-Architekturen*. PhD thesis, Universität des Saarlandes, FB. Informatik, 1994.

[Hil99] Mark Hillebrand. Design and Evaluation of a Superscalar RISC Processor. Master's thesis, Universität des Saarlandes, FB. Informatik, 1999. Started in May 1998.

[HP96] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, INC., San Mateo, CA, 2nd edition, 1996.

[Ins85] Institute of Electrical and Electronics Engineers. *ANSI/IEEE standard 754–1985, IEEE Standard for Binary Floating-Point Arithmetic*, 1985. For a readable account see the article by W.J. Cody et al. in the IEEE MICRO Magazine, Aug. 1984, 84–100.

[KP95] Jörg Keller and Wolfgang J. Paul. *Hardware Design — Formaler Entwurf Digitaler Schaltungen*. TEUBNER, Stuttgart, Leipzig, 1995.

[Krö97] Daniel Kröning. http://www-wjp.cs.uni-sb.de/~kroening/tomasulo/, 1997.

[Lei98] Holger Leister. *Quantitative Analysis of Precise Interrupt Mechanism for Processors with Out-Of-Order Execution*. PhD thesis, Universität des Saarlandes, Fachbereich 14 Informatik, 1998. Preliminary Version, 03/1998.

[Mot97] PowerPC 750 RISC Microprocessor Technical Summary, 1997.

[MP95] S.M. Müller and W.J. Paul. *The Complexity of Simple Computer Architectures*. Lecture Notes in Computer Science 995. Springer, 1995.

[MP98] S.M. Müller and W.J. Paul. *The Complexity of Simple Computer Architectures II*, 1998. Monograph (Draft). Email: {smueller, wjp}@cs.uni-sb.de.

[Mül97a] S.M. Müller. Vorlesung Rechnerarchitektur II WS 96/97, Universität des Saarlandes, Fachbereich 14 Informatik, 1997.

[Mül97b] S.M. Müller. Complexity and correctness of computer architectures. In *Proc. 4th Workshop on Parallel Systems and Algorithms (PASA'96)*, pages 125–146. World Scientific Publishing, 1997.

[PS98] W.J. Paul and P.-M. Seidel. On the complexity of Booth recoding. In *Proc. 3rd Conference on Real Numbers and Computers (RNC3)*, 1998.

[SP88] J.E. Smith and A.R. Pleszkun. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers*, 37(5):562–573, 1988.

[Tom67] R.M. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. In *IBM Journal of Research and Development*, volume 11 (1), pages 25–33. IBM, 1967.