Chapter

6

Out-of-Order Execution

6.1 Introduction

IN THE PREVIOUS SECTIONS, we presented various implementations of pipelined RISC processors. These implementations strictly processed the instructions in program order. However, the performance of these designs drops as soon as long latency instructions such as memory accesses are involved. For example, consider a load instruction with cache miss in the memory stage. Thus, the stall signal of the stage is activated and the instructions above the memory stage are stalled.

Furthermore, consider an ALU instruction that follows the load in the execute stage:

```
EX:     R3:=R1+R2
M:      R4:=Mem[R5]
```

If there is no data dependency, the result of the ALU instruction is already known in the execute stage and could be written into the register file. However, the in-order execution rule prohibits this and the ALU instruction has to wait for the load.

Thus, dropping this rule can result in better performance. This technique is called *out-of-order execution*. The most popular out-of-order execution

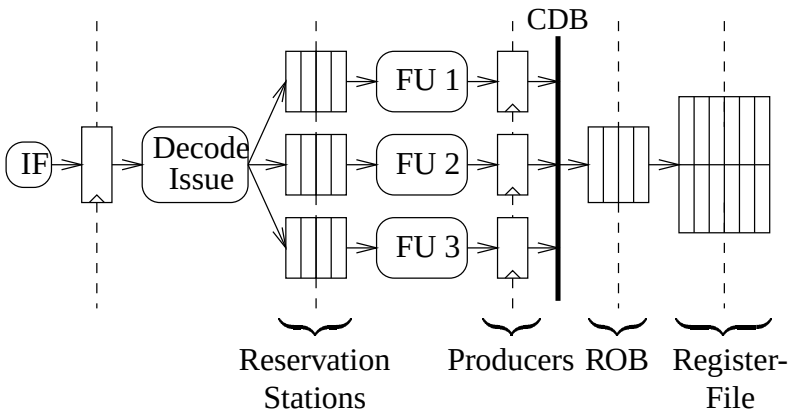

Figure 6.1 Basic structure of a microprocessor with Tomasulo Scheduler and reorder buffer

algorithms is the Tomasulo scheduling algorithm [Tom67]. It is one of the most competitive scheduling algorithms and provides CPI rates down to 1.1 on a single-instruction issue machine [Ger98, Del98, MLD+99]. The algorithm is widely used, e.g., by IBM PowerPC, Intel Pentium-Pro or AMD K5 [Mot97, CS95]. The original Tomasulo scheduler uses out-of-order termination and therefore does not support precise interrupts without extra hardware. We support precise interrupts by adding a *reorder buffer*[SP88]. The reorder buffer sorts the instructions in program order before termination.

In this chapter, we describe the results of implementing and verifying a DLX with Tomasulo scheduler, precise interrupts and floating point unit using PVS. The designs, the scheduling protocols, and most proofs are taken from [KMP99, Krö99].

6.2 The Tomasulo Algorithm with Reorder Buffer

Figure 6.1 depicts the basic structure of a microprocessor with Tomasulo scheduler and reorder buffer. The execution begins with the instruction fetch, as in the in-order machine. The Tomasulo scheduling algorithm does not cover this phase; it is assumed that the instruction fetch is done in program order. We will use the very same instruction fetch mechanism as in the pipelined in-order machines described in the previous chapters.

In the next stage, the instruction is decoded. This includes fetching the operands if available. The instruction and the operands are then passed to a *reservation station* (RS). This is called *issue*. The reservation stations are the central data structure of the Tomasulo scheduling algorithm. The reservation stations act as queue for the instructions and are between the decode/issue stage and the functional units. Note that the instruction is passed to the reservation station even if forwarding fails. This is in contrast to the in-order machine, which stalls in this case.

As soon as all operands are available, the instruction is passed from the reservation station to the functional unit. This is called *dispatch*. This is done without obeying the program order of the instructions, i.e., the instructions can overtake each other at this point. After the function unit has finished the execution, the result of the instruction is passed to a special register, called *producer*.

In case the producer holds an instruction, it requests a result bus, called *common data bus* (CDB). As soon as the request is acknowledged, the result is put on this bus. This is called *completion*. In contrast to commerical designs such as the IBM's PowerPC, we support only one CDB. The bus is used for two purposes: 1) The instruction is passed to the reservation stations that wait for the result because of a data dependency, and 2) the result is passed to the reorder buffer.

The reorder buffer re-sorts the instructions back in program order. The benefit of this is that we can write the results into the register file in program order (in-order termination). This allows precise interruptions of the instruction stream.

In the following sections, we will describe the data structures and protocols used to realize this in detail.

6.3 Tomasulo Data Structures

6.3.1 Reorder Buffer

The reorder buffer [SP88] is a ring-buffer that serves two purposes in a machine with Tomasulo scheduler. The main purpose is to re-sort the instructions such that the instructions terminate in program order. For that purpose, each reorder buffer entry provides space to store the result of an

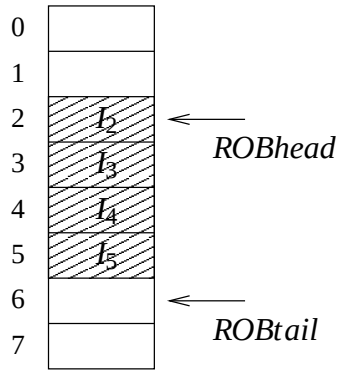

Figure 6.2 Illustration of the reorder buffer pointers

instruction. We support instructions that write multiple registers. This is useful for supporting double precision floating point instructions.

Furthermore, each reorder buffer entry has a *valid bit*. The bit indicates that the result of the instruction is in the reorder buffer entry. A reorder buffer entry with active valid bit is called valid reorder buffer entry.

The second purpose of the reorder buffer is to provide means to assign a *tag* to each instruction. The tag is assigned during instruction issue and stays unique until the instruction terminates. The tag is the address of the reorder buffer entry of the instruction. Let ϑ denote the number of tag (i.e., ROB address) bits. Thus, the reorder buffer has

$$\Theta := 2^{\vartheta}$$

entries. We denote the value of the ROB entry with address *tag* during cycle T with $ROB[tag]^T$.

The reorder buffer is accessed using to pointers, the head and tail pointers. These pointers are stored in ϑ-bit registers. We denote the value of the head pointer during cycle T by $ROBhead^T$, and the value of the tail pointer by $ROBtail^T$. Instructions are put in the ROB entry *ROBtail* points to, and removed from the entry *ROBhead* points to. After an instruction is put in the ROB, the *ROBtail* pointer is increased. After an instruction is removed from the ROB, the *ROBhead* pointer is increased. The pointers wrap-around if they reach the end of the ROB. This is illustrated in figure 6.2.

Let $issue(T)$ denote that we issue an instruction during cycle T. This allows defining the values of *ROBtail* recursively. We initialize the ROB

pointers with zero. The *ROBtail* pointer is increased iff we issue an instruction.

$$ROBtail^T := \begin{cases} 0 & : \ T = 0 \\ ROBtail^{T-1} + 1 & : \ issue(T-1) \\ ROBtail^{T-1} & : \ \text{otherwise} \end{cases}$$

Note that the incrementation for the case $issue(T-1)$ holds is a bitvector operation as described in chapter 2. Thus, the *ROBtail* pointer wraps around.

In analogy to that, let $writeback(T)$ denote that we terminate an instruction during cycle T. This allows defining the values of *ROBhead* recursively.

$$ROBhead^T := \begin{cases} 0 & : \ T = 0 \\ ROBhead^{T-1} + 1 & : \ writeback(T-1) \\ ROBhead^{T-1} & : \ \text{otherwise} \end{cases}$$

As above, the incrementation for the case $writeback(T-1)$ holds is a bitvector operation as described in chapter 2. Thus, the *ROBhead* pointer wraps around.

6.3.2 Register File Extentions

As before, the register file holds the values of the specification registers of the machine. We still denote the set of registers by $\mathcal{R}$ (in PVS, we just number the registers). We denote the value of the register $r \in \mathcal{R}$ during cycle T by $R[r]^T.data$. We assume that all registers have a common width. We denote the set of possible values of a register by $\mathcal{W}(R)$.

The register file is extended with a *producer table*. The producer table records which instruction in the machine writes a given register. For that purpose, the producer table contains two data items for each register.

The first is a valid bit. We denote the value of the valid bit of register r during cycle T with $R[r]^T.valid$. If it is set, there is no instruction currently executing with the register as destination. If it is not set, there is such an instruction. In this case, the second item, a reorder buffer tag, points to the last instruction with the register as destination. We denote the value of this tag by $R[r]^T.tag$.

6.3.3 Reservation Stations

The reservation stations act as queue for the instructions and their source operands. We give each reservation station a number. We denote the values in reservation station number *rs* during cycle T by $RS[rs]^T$. Each reservation has a full bit $RS[rs].full$. It indicates that the reservation station is in use. In addition to that, we store the tag of the instruction in the reservation station in $RS[rs].tag$.

We support instructions with an arbitrary number of source operands. Let x denote the number of a source operand. For each source operand, we store a valid bit $RS[rs].op[x].valid$. If the bit is set, the value of the operand is stored in $RS[rs].op[x].data$. If it is not set, we store the tag of the instruction producing the value in $RS[rs].op[x].tag$.

6.3.4 Producers

The producers buffer the results from the function units until the CDB is available. We have a separate producer for each function unit. Each producer consists of a full bit, a tag, and the result. We denote these items of producer fu by $P[fu].full$, $P[fu].tag$, and $P[fu].result$.

6.3.5 Initial Configuration

We make the following assumptions about the initial values of those registers.

- The valid bits of the registers must be set in the initial configuration. We do not make an assumption on the values of the registers or the tags.
- The full bits of the reservation stations must not be set. We do not make any assumptions about the other values in the reservation stations.
- The full bits of the producers must not be set. We do not make any assumptions about the other values in the producers.

It is important that we do not make to many assumptions on initial values, since realizing fixed initial values in hardware is expensive regarding hardware cost. In particular, assuming initial values of a register usually prohibits implementing the register as RAM. In particular, note that we do not make any assumption about the initial values of the ROB entries.

6.4 Tomasulo Protocols

6.4.1 Formalization

In this section, we describe the protocols of the Tomasulo Scheduling algorithm. These protocols form the transition function of a generic and abstract microprocessor with Tomasulo scheduler. The configuration set of this machine comprises of the reservation stations, the reorder buffer including the pointers, the register files, the producers, and the producer tables.

We denote the configuration of this machine during cycle T by c_{aI}^{T} (abstract implementation).

The transition function of the machine is denoted by δ_{aI}. It maps the configuration of the machine during cycle T to the next configuration of the machine during cycle $T+1$. We will compose this function using functional specifications of the Tomasulo protocols, which are issue, CDB snooping, dispatch, completion, and writeback. We name the functions for these protocols *issue*, *snoop*, *dispatch*, *completion*, and *writeback*. These functions are called protocol functions.

$$\delta_{aI} \quad := \quad issue \circ snoop \circ dispatch \circ completion \circ writeback$$

Thus, the issue protocol has priority over CDB snooping and so on. This is important if two protocols change the same register value in the same cycle. The final value in the register is the value provided by the protocol with the higher priority. We omit the transition function for the ROB pointers, since we already specified the values of those pointers above.

Notation We specify the protocols using a notation similar to the notation used in [KMP99]. The notation is also very similar to the notation

used in PVS. Consider the following example:

$$R[4].data \quad := \quad R[3].data$$

This is a shorthand for $R[4]^{T+1}.data = R[3]^{T}.data$.

As before, we consider a stream of instructions I_0, I_1, ... Each instruction has source and destination registers. By $S(i,x)$, we denote the number of register that is the source operand x. By $D(i,x)$, we denote the number of register that is the destination operand x.

By $dest(i,r)$, we denote the fact that instruction I_i has r as destination register, i.e., that there is a x with $D(i,x) = r$.

Embedding Convention In a machine with Tomasulo Scheduler and reorder buffer, there are different places where results are stored or propagated before writing the results into the register file. These are the producers, the CDB, and the ROB. We support multiple destination registers for a single instruction. By convention, each destination register is on a well-defined part of the result bus or registers. For example, consider the DLX with floating point instructions. That machine has a maximum of three results for each instruction. Thus, the result busses and registers have space for three 32-bit registers, $result[0]$, $result[1]$, and $result[2]$.

In case of the DLX, we embed the results as follows: By convention, all floating point registers with odd numbers are on $result[1]$, all other "normal" registers are on $result[0]$. In order to handle exceptions, we define a dummy register *CA*, which is on $result[2]$. This allows handling the IEEE flags register and exceptions.

For example, the result of a double precision floating point instruction with destination register FPR_0 is embedded as follows: The lower part of the result, i.e., the part that is written into FGR_0, is on $result[0]$. The higher part, i.e., the part that is written into FGR_1, is on $result[1]$. The exceptions/IEEE flags are on $result[2]$.

Formally, we define an embedding function. Let d denote the maximum number of destination operands. The embedding function e maps a register to a number in $\{0,\ldots,d-1\}$. Thus, destination register r is on $result[e(r)]$.

6.4.2 Issue

Let I_i be the instruction to be issued during cycle T (figure 6.3). The first step is to invalidate the destination registers of instruction I_i. Thus, we clear the valid bit of all registers $R[r]$ with $dest(i,r)$ and set the tag of register r to $ROBtail^T$.

In contrast to the issue protocols given in [MPK00], we cover two different ways to issue an instruction: the first way is as described in [MPK00] and as done by the original Tomasulo scheduling algorithm. During issue, the instruction is stored in a reservation station along with the source operands that are available.

The second way is to skip the reservation stations and to store the result of the instruction in the reorder buffer directly. This speeds up the execution of simple instructions. Examples for this are branches, jumps, and the *trap* instruction.

The result of these instructions is already known in the issue stage. We indicate these instructions by the predicate $issue\_with\_result(i)$. In case of such an instruction, the reservation stations are not modified by the issue protocol. However, we set the valid bit of the ROB entry *ROBtail* points to and store the result in the *result* data item. We denote this result by $issue\_result(i)$. For example, this could be the *PC* address in case of a jump-and-link instruction.

Machines that support instructions that are directly issued into the ROB are usually not covered in the open literature. The Tomasulo implementation in [Krö99] uses this feature. However, the proof does not cover it.

In case $issue\_with\_result(T)$ does not hold, we clear the valid bit of the ROB entry $ROBtail^T$. Let $issue\_rs(T,rs)$ hold iff reservation station *rs* is used for issue during cycle T. We initialize this reservation station as follows: we set the full bit of the reservation station and store the *ROBtail* pointer in the tag data item. Besides the full bit and tag, the reservation station holds the source operands.

The Tomasulo scheduling algorithm with reorder buffer supports different places to forward the source operands from. For each operand of the instruction three sources have to be checked:

1. The operand might be in the register file. In this case, the valid bit of the register is set. If it is not in the register file, the producer table

provides the tag of the last instruction writing it.

2. The operand might be on the CDB. In order to determine which instruction is on the CDB, the result on the CDB comes with a valid bit and a tag. If the valid bit is set, the tag indicates the instruction on the CDB. Thus, we check the valid bit and compare the tag on the CDB with the tag from the producer table. If they match, we take the result on the CDB as source operand according to the embedding convention.

3. The operand might be in the reorder buffer. This is indicated by the valid bit of the reorder buffer entry that the tag in the producer table points to. If the bit is set, we take the result from the ROB according to the embedding convention.

If none of the three cases above applies, the source register is the destination of a preceding, incomplete instruction. The tag of this instruction is in the producer table, and instead of the operand, the tag of this instruction is stored in the reservation station.

6.4.3 CDB Snooping

During issue, the operands in the reservation station that are not available are marked as not valid. 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 6.4). The reservation stations identify the results by comparing the tag on the CDB with the tag in the reservation station.

6.4.4 Dispatch

During instruction dispatch (figure 6.5), an instruction moves from a reservation station entry into the actual function unit. We denote this fact by the predicate $dispatch(T, rs)$. If the predicate holds, the instruction in reservation station *rs* is dispatched during cycle T.

The reservation stations that are dispatched are determined by the hardware using a fair arbiter, which selects only full reservations with valid

```
if issue(T) then
{
  RS[rs].full := 1;
  RS[rs].tag := ROBtail;

  For all source operands x of I_i, let r be S(i,x):
     if R[r].valid then
        RS[rs].op[x] := R[r];
     elsif CDB.tag = R[r].tag ∧ CDB.valid then
        RS.op[x].valid := 1;
        RS.op[x].data := CDB.result[e(r)];
     elsif ROB[R[r].tag].valid then
        RS.op[x].valid := 1;
        RS.op[x].data := ROB[R[r].tag].result[e(r)];
     else
        RS.op[x].valid := 0;
        RS.op[x].tag := R[r].tag;
     endif

  For all registers r with dest(i,r):
     R[r].tag := ROBtail;
     R[r].valid := 0;
}
```

Figure 6.3 Issue protocol for issuing instruction I_i during cycle T.

```
  ∀ operands x of instruction I_i
     if RS[rs].full ∧ /RS[rs].op[x].valid∧
          (RS[rs].op[x].tag = CDB.tag)
       {
         RS[rs].op[x].valid := 1;
         RS[rs].op[x].data := CDB.result[e(S(i,x))];
       }
```

Figure 6.4 CDB snooping protocol for instruction I_i in reservation station *rs*

```
if dispatch(T,rs) then
 {
  Pass instruction, operands,
  and tag to FU

  RS.full := 0;
 }
```

Figure 6.5 Dispatch protocol

operands. Thus, we can assume that the reservation stations *rs* that are dispatched are full and have valid operands:

$$dispatch(T,rs) \implies RS[rs]^T.full \wedge \forall x : RS[rs]^T.op[x].valid$$

In addition to passing the instruction to the function unit, the reservation station is freed during dispatch. Note that clearing the full bit may conflict with setting the full bit as done by the issue protocol. Since the issue protocol has priority, the full bit is set in this case.

6.4.5 Completion

During completion (figure 6.6), the result and the ROB tag in a producer $P[fu]$ are put on the CDB. Let the predicate $completion(T)$ hold iff the machine completes an instruction. Let $fu = compl\_p(T)$ denote the number of the producer that holds that instruction. That number is determined by the hardware among the full producers using a fair arbiter. Thus, we can assume that the producer is full:

$$completion(T) \implies P[compl\_p(T)]^T.full$$

During completion, the according reorder buffer entry is filled with the result and the valid bit is set. Let $FU[fu]^T.valid$ denote that the function unit provides a result. Let $FU[fu]^T.result$ denote that result. Let $FU[fu]^T.tag$ denote the tag that accompanies the result.

If the function unit provides a new result, this result is stored in the producer. If not so, the full bit of the producer is cleared.

```
if completion(T) then
 {
  CDB^T.valid = 1;
  CDB^T.result = P[compl_p(T)].result;
  CDB^T.tag = P[compl_p(T)].tag;

  ROB[CDB^T.tag].valid := 1;
  ROB[CDB^T.tag].result := CDB^T.result;
 }

∀ function units fu:
  if FU[fu]^T.valid then
   {
    P[fu].full := 1;
    P[fu].result := FU[fu]^T.result;
    P[fu].tag := FU[fu]^T.tag;
   }
  elsif completion(T) ∧ compl_p(T) = fu then
    P[fu].full := 0;
  endif
```

Figure 6.6 Completion protocol

```
if writeback(T)
  for all registers r with dest(i,r):
   {
    R[r].data := ROB[ROBhead].result[e(r)];
    if ROBhead = R[r].tag then
       R[r].valid := 1;
   }
```

Figure 6.7 Retirement / writeback protocol for instruction I_i.

6.4.6 Writeback

During writeback (figure 6.7), a result of the instruction in the ROB entry that *ROBhead* points to is written into the register file. As introduced above, we denote this fact by the predicate $writeback(T)$. We assume that writeback is done iff the ROB entry is valid and the ROB is not empty. Let $ROBempty(T)$ denote that the ROB is empty during cycle T. We will later on define it.

$$writeback(T) \iff \overline{ROBempty(T)} \wedge ROB[ROBhead(T)]^T.valid$$

During writeback, we store the result in the ROB in the registers. Furthermore, we set the valid bit of the register if the tag of the instruction matches the tag in the producer table.

Note that setting the valid bit may conflict with clearing the valid bit during issue. As described above, the issue protocol has priority over the writeback protocol, i.e., the setting of the valid bit is suppressed.

6.5 Data Consistency

6.5.1 Scheduling Functions

We need a formal way to state that "instruction I_i is being issued during cycle T" or "instruction I_i is being dispatched during cycle T". We do this in analogy to the previous chapters using a *scheduling function*. While this concept was introduced for in-order machines by [MP00], we extend it to out-of-order machines in the obvious way.

Issue We recursively define a function *sIissue* that maps a cycle T to the number of the instruction that is in the issue stage. Since we issue in program order, that number increases by one in case that $issue(T)$ holds and stays unmodified otherwise. We start with instruction I_0.

$$sIissue(T) := \begin{cases} 0 & : \ T = 0 \\ sIissue(T-1)+1 & : \ issue(T-1) \\ sIissue(T-1) & : \ \text{otherwise} \end{cases}$$

Reservation Stations We also desire a way to define the instruction in a given reservation station *rs* during a given cycle T. We do this by defining a schedule function $sIRS(rs,T)$ for reservation stations. Instructions are put in a reservation station during issue. In case an instruction is issued into reservation station *rs*, we take the value of $sIissue(T-1)$. Otherwise, the value of $sIRS(rs,T)$ remains unchanged.

$$sIRS(rs,T) := \begin{cases} 0 & : \ T = 0 \\ sIissue(T-1) & : \ issue(T-1) \\ sIRS(rs,T-1) & : \ \text{otherwise} \end{cases}$$

Note that the only point we put an instruction into a reservation station is during issue. This is in contrast to the implementation given [Krö99], which moves the instructions from one reservation station into the next.

Reorder Buffer In analogy to the schedule of the reservation stations, we can provide a schedule for the ROB. The function $sIROB(tag,T)$ denotes the instruction that is in the ROB entry with tag *tag* during cycle T. We start with -1, which denotes that no instruction is in the ROB entry. We need this special value because the ROB entries have no such thing like a full bit.

$$sIROB(tag,T) := \begin{cases} -1 & : \ T = 0 \\ sIissue(T-1) & : \ issue(T-1) \wedge \\ & \quad tag = ROBtail^{T-1} \\ sIROB(tag,T-1) & : \ \text{otherwise} \end{cases}$$

Function Units Let $dispatch\_fu(T,fu)$ denote the number of the reservation station that is used for dispatching an instruction to function unit fu during cycle T. In hardware, this number is represented unary using $dispatch(T,rs)$.

Let $sIdispatch(fu,T)$ denote the number of the instruction passed to function unit fu during cycle T. This is defined using the schedule of the reservation station.

$$sIdispatch(fu,T) \quad := \quad sIRS(dispatch\_fu(T,fu),T)$$

We also define schedules for the functional units. Let $sIfu(fu,T)$ denote the number of the instruction that **leaves** function unit fu during cycle T. The most simple functional unit is a combinatorial functional unit that calculates its result within the same cycle the arguments are passed. The 32-bit ALU presented in chapter 2 is an example. For such a function unit, $sIfu(fu,T)$ just is:

$$sIfu(fu,T) \quad := \quad sIdispatch(fu,T)$$

In case of more complex function units such as floating point dividers, one has to construct a scheduling function. There are two ways to do so: 1) one constructs the function such that it matches the pipeline structure of the functional unit, and 2) one defines the schedule using the tags the function unit provides.

As an example for the first method, consider a function unit with four stages and a cycle that allows iterating the instruction in stage 2 (figure 6.8). We denote the instruction in stage k of the function unit fu during cycle T by $sI_{fu}(k,T)$. The instruction in stage 0 of the function unit is the instruction that is dispatched:

$$sI_{fu}(0,T) \quad := \quad sIdispatch(fu,T)$$

This instruction proceeds into stage 1 iff the update enable signal $ue_{fu,0}$ is active. This update enable signal is local to the function unit fu.

$$sI_{fu}(1,T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sI_{fu}(0,T-1) & : \quad ue_{fu,0}^{T-1} = 1 \\ sI_{fu}(1,T-1) & : \quad \text{otherwise} \end{cases}$$

This must be changed for stage 2, the stage with the back-cycle.

$$sI_{fu}(2,T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sI_{fu}(1,T-1) & : \quad ue_{fu,1}^{T-1} = 1 \wedge sel_1^{T-1} = 0 \\ sI_{fu}(2,T-1) & : \quad ue_{fu,1}^{T-1} = 1 \wedge sel_1^{T-1} = 1 \\ sI_{fu}(2,T-1) & : \quad \text{otherwise} \end{cases}$$

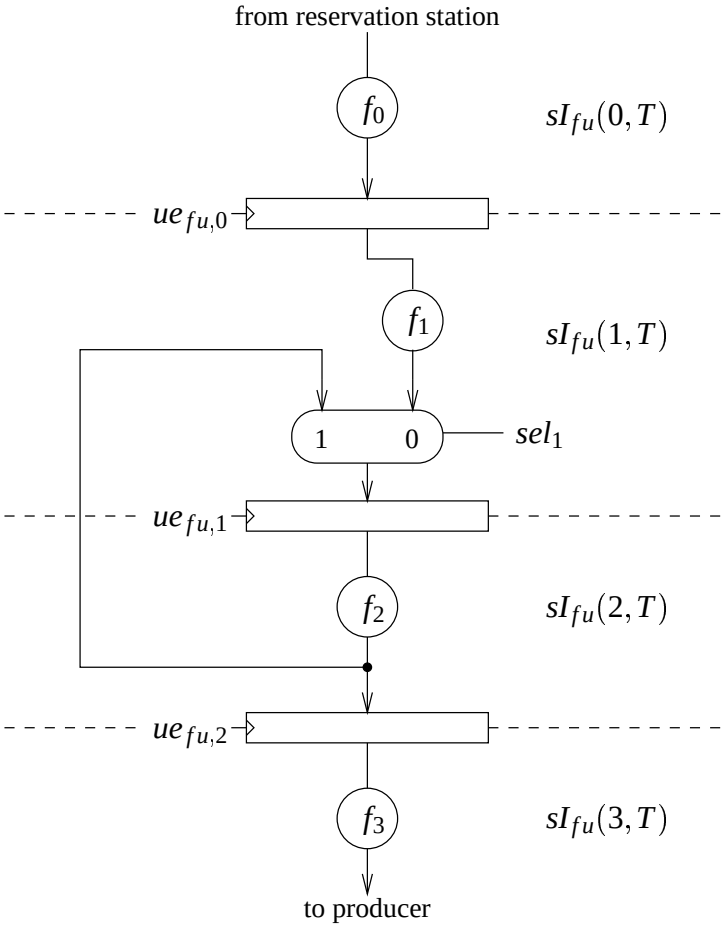

Figure 6.8 Construction of the scheduling function for a function unit with cycles

For stage 3 of the function unit, the scheduling function is defined in analogy to the scheduling function of stage 1:

$$sI_{fu}(3,T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sI_{fu}(2,T-1) & : \quad ue_{fu,2}^{T-1} = 1 \\ sI_{fu}(3,T-1) & : \quad \text{otherwise} \end{cases}$$

Since this is also the last stage of the function unit fu, we have

$$sIfu(fu,T) \quad := \quad sI_{fu}(3,T)$$

Producers In analogy to the scheduling function of the reservation stations, we define the scheduling function of the producer registers. We denote the number of the instruction in producer number fu during cycle T by $sIP(fu,T)$. In case the function unit provides a result, we take the value from the schedule of the function unit as defined above. If not so, the value of $sIP(fu,T)$ does not change.

$$sIP(fu,T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sIfu(fu,T-1) & : \quad FU[fu]^{T-1}.valid = 1 \\ sIP(fu,T-1) & : \quad \text{otherwise} \end{cases}$$

As described above, the instruction in producer with the number given by $compl\_p(T)$ is put on the CDB during completion. We therefore define the following shorthand for the instruction on the CDB during cycle T:

$$sICDB(T) \quad := \quad sIP(compl\_p(T),T)$$

Writeback In analogy to *sIissue*, we recursively define a scheduling function *sIwriteback* that maps a cycle T to the number of the instruction that is in the writeback stage. Since we writeback in program order, that number increases by one in case that $writeback(T)$ holds and stays unmodified otherwise. We start with instruction I_0.

$$sIwriteback(T) \quad := \quad \begin{cases} 0 & : \quad T = 0 \\ sIwriteback(T-1)+1 & : \quad writeback(T-1) \\ sIwriteback(T-1) & : \quad \text{otherwise} \end{cases}$$

6.5.2 Function Unit Axioms

In this section, we describe the assumptions we make regarding data consistency properties of the functional units. We consider the functional units as a "black box". In particular, we do not provide implementations for data memory or floating point function units. The design and verification of a data memory function unit including virtual memory is subject of the thesis of Sven Beyer [Bey01]. The design and verification of an IEEE compliant floating unit including a divider is subject of the thesis of Christian Jacobi [Jac01].

Inputs and Outputs As described above, $FU[fu]^T.valid$ indicates that function unit fu provides a result during cycle T. $FU[fu]^T.tag$ denotes the tag the function unit provides, and $FU[fu]^T.result$ denotes the result the function unit provides.

Let $fuins(fu,T)$ denote the inputs of function unit fu during cycle T. This is a defined as follows: Let *rs* be a shorthand for $dispatch\_rs(T, fu)$. This is the reservation station that is used for dispatching to function unit fu.

$$\begin{aligned} fuins(fu,T).valid &:= dispatch\_rs(T,rs) \\ fuins(fu,T).tag &:= RS[rs]^T.tag \\ fuins(fu,T).source[x] &:= RS[rs]^T.op[x].data \end{aligned}$$

Tag Consistency Given that the function unit gets correct tags as inputs upto cycle T, we assume that the function unit provides the correct tag of the instruction as output during cycle T.

We formalize "gets correct tags as inputs upto cycle T" as follows:

$$\begin{aligned} & \forall T' \leq T : fuins(fu,T').valid \\ \Longrightarrow \quad & fuins(fu,T').tag = I\_tag(sIdispatch(fu,T')) \end{aligned}$$

We formalize "provides the correct tag of the instruction" as follows:

$$FU[fu]^T.valid \quad \Longrightarrow \quad FU[fu]^T.tag = I\_tag(sIfu(fu,T))$$

Operand Consistency Given that the function unit gets correct source operands as inputs upto cycle T, we assume that the function unit provides the correct results of the instruction as output during cycle T.

We formalize "gets correct source operand as inputs upto cycle T" as follows:

$$\begin{aligned} & \forall T' \leq T : fuins(fu, T').valid \\ \Longrightarrow \quad & fuins(fu, T').source = source(sIdispatch(fu, T')) \end{aligned}$$

We formalize "provides the correct results of the instruction" as follows:

$$FU[fu]^T.valid \quad \Longrightarrow \quad FU[fu]^T.result = result(sIfu(fu, T))$$

Phase Consistency In order to show data consistency, we have to argue that the function units does not generate "garbage output". We assume two things: 1) If an instruction leaves the function unit, it entered it before, and 2) if instructions upto cycle T enter the function unit at most one, the instructions leave the function unit at most once.

We formalize this as follows: Let $in(i, T, fu)$ denote that instruction I_i enters the function unit fu during cycle T.

$$in(i, T, fu) \quad :\Longleftrightarrow \quad fuins(fu, T).valid \wedge sIdispatch(fu, T) = i$$

In analogy to that, let $out(i, T, fu)$ denote that instruction I_i leaves the function unit fu during cycle T.

$$out(i, T, fu) \quad :\Longleftrightarrow \quad FU[fu]^T.valid \wedge sIfu(fu, T) = i$$

If instruction I_i leaves function unit fu during cycle T, there must be a cycle $T' \leq T$ such that it entered the function unit:

$$out(i, T, fu) \quad \Longrightarrow \quad \exists T' \leq T : in(i, T', fu)$$

If the cycle $T' \leq T$ such that instruction I_i enters the function unit during cycle T' is unique, then the cycle $T'' \leq T$ such that instruction I_i leaves the function unit during cycle T'' is unique.

$$\left|\{T' \leq T \mid in(i, T', fu)\}\right| = 1 \quad \Longrightarrow \quad \left|\{T'' \leq T \mid out(i, T'', fu)\}\right| = 1$$

We do not make further assumptions regarding data consistency. In particular, this allows that the latency of the function unit is variable and that the instructions leave the dispatch order within the function unit.

We make further assumptions on the function units in order to show liveness. We will later on describe these assumptions.

6.5.3 ROB Flags

We need means to determine wether the reorder buffer is full or not. For this purpose, we take the circuit from [Lei99]. It uses a $\vartheta+1$ bit counter register. The counter is incremented if we issue and instruction and do not writeback one simulataneously. This is indicated by $ROBinc(T)$.

$$ROBinc(T) \quad = \quad issue(T) \wedge \overline{writeback(T)}$$

In analogy to that, $ROBdec(T)$ indicates that we decrement the counter. This is done if we writeback an instruction but do not issue one simultaneously.

$$ROBdec(T) \quad = \quad \overline{issue(T)} \wedge writeback(T)$$

Thus, the value of the counter register during cycle T is defined as follows:

$$ROBcount(T) \quad := \quad \begin{cases} 0^{\vartheta} & : \quad T = 0 \\ ROBcount(T-1)+1 & : \quad ROBinc(T-1) \\ ROBcount(T-1)-1 & : \quad ROBdec(T-1) \\ ROBcount(T-1) & : \quad \text{otherwise} \end{cases}$$

The ROB is empty iff the counter is zero:

$$ROBempty(T) \quad = \quad (ROBcount(T) = 0^{\vartheta+1})$$

The ROB is full iff the counter is the number of ROB entries Θ. We use the binary encoding of Θ.

$$ROBempty(T) \quad = \quad (ROBcount(T) = 10^{\vartheta})$$

We make the following assumptions:

- If we issue an instruction without simultaneous writeback, the ROB must not be full.

$$ROBinc(T) \implies \overline{ROBfull(T)}$$

- If we writeback an instruction, the ROB must not be empty.

$$writeback(T) \implies \overline{ROBempty(T)}$$

6.5.4 ROB Properties

Definition 6.1 ▶ $tag \oplus i$

Let $tag \oplus i$ be a shorthand for a tag that is incremented i times. Formally, this is defined using a recursion and the bit-vector incrementation as defined in chapter 2:

$$tag \oplus i := \begin{cases} tag & : \ i = 0 \\ (tag \oplus (i-1)) + 1 & : \ \text{otherwise} \end{cases}$$

Note that we increment a bit vector with limited range. Thus, it will wrap-around. One easily verifies the following properties of the ROB pointers:

Lemma 6.1 ▶ Let i be the number of the instruction in the issue stage. The ROB tail pointer has been increased i times.

$$ROBtail^T = 0^\vartheta \oplus sIissue(T)$$

Lemma 6.2 ▶ Let i be the number of the instruction in the writeback stage. The ROB head pointer has been increased i times.

$$ROBhead^T = 0^\vartheta \oplus sIwriteback(T)$$

The proof for both lemmas is easily done using induction on T.

Lemma 6.3 ▶ The value in the *ROBcount* register is smaller or equal than the number of ROB entries.

$$\langle ROBcount(T) \rangle \leq \Theta$$

PROOF One verifies this claim by induction on T. For $T = 0$, we have

$$\langle \mathit{ROBcount}(T) \rangle \quad = \quad 0.$$

For $T+1$, we show the claim by a full case split on the values of $\mathit{ROBinc}(T)$ and $\mathit{ROBdec}(T)$.

- If neither $\mathit{ROBinc}(T)$ or $\mathit{ROBdec}(T)$ holds, the value of *ROBcount* does not change and the claim is concluded using the induction premise.
- If $\mathit{ROBinc}(T)$ holds, we assert the claim as follows: in case

  $$\langle \mathit{ROBcount}(T) \rangle \quad < \quad \Theta$$

  holds, the claim is easily concluded. Assume

  $$\langle \mathit{ROBcount}(T) \rangle \quad = \quad \Theta$$

  holds. In this case, we have a contradiction to the assumption above since $\mathit{ROBinc}(T)$ holds and the ROB is full.
- If $\mathit{ROBdec}(T)$ holds, we assert the claim as follows: in case

  $$\langle \mathit{ROBcount}(T) \rangle \quad \neq \quad 0$$

  holds, the claim is easily concluded. Assume

  $$\langle \mathit{ROBcount}(T) \rangle \quad = \quad 0$$

  holds. In this case, we have a contradiction to the assumption above since $\mathit{ROBdec}(T)$ holds and the ROB is empty. QED

Let ◀ Lemma 6.4

$$\mathit{instr\_in\_rob}(T) = \mathit{sIissue}(T) - \mathit{sIwriteback}(T)$$

denote the difference between the number of issued and terminated instructions, i.e., the number of instructions in the reorder buffer. We claim that this number is equal to the binary number interpretation of the value of $\mathit{ROBcount}(T)$:

$$\mathit{instr\_in\_rob}(T) \quad = \quad \langle \mathit{ROBcount}(T) \rangle$$

PROOF This claim is asserted by induction on T. For $T = 0$ we have

$$\begin{aligned} \mathit{instr\_in\_rob}(T) &= \langle \mathit{ROBcount}(T) \rangle \\ \mathit{sIissue}(T) - \mathit{sIwriteback}(T) &= \langle 0^{\vartheta+1} \rangle \\ 0 - 0 &= \langle 0^{\vartheta+1} \rangle. \end{aligned}$$

For $T+1$, we do a full case split on the values of the signals $\mathit{issue}(T)$ and $\mathit{writeback}(T)$.

- If neither $\mathit{issue}(T)$ nor $\mathit{writeback}(T)$ holds, both the values of the scheduling functions and the ROB counter do not change from cycle T to $T+1$. Thus, the claim is concluded by the induction premise.
- If both $\mathit{issue}(T)$ and $\mathit{writeback}(T)$ hold, both scheduling functions are incremented by one. Thus, the difference stays the same. The ROB counter does not change from cycle T to $T+1$. Thus, the claim is concluded by the induction premise.
- In case $\mathit{issue}(T)$ holds and $\mathit{writeback}(T)$ does not hold, the difference is increased by one. The ROB couter is also increased by one. One asserts that the ROB counter does not wrap around by lemma 6.3.
- In case $\mathit{issue}(T)$ doe not hold and $\mathit{writeback}(T)$ holds, the difference is decreased by one. The ROB couter is also decreased by one. One asserts that the ROB counter does not wrap around using the assumption that we do not writeback in case of an empty ROB. 6.3.

QED

Lemma 6.5 ▶ The number of instructions in the ROB is greater or equal than zero.

$$\mathit{instr\_in\_rob}(T) \geq 0$$

One easily asserts this using lemma 6.4.

Lemma 6.6 ▶ The number of instructions in the ROB is smaller or equal than the number of ROB entries.

$$\mathit{instr\_in\_rob}(T) \leq \Theta$$

This is easily shown using lemma 6.4 and lemma 6.3.

The following lemma is easily concluded using lemma 6.5:

The number of issued instructions is greater or equal than the number of terminated instructions. ◀ Lemma 6.7

$$sIissue(T) \geq sIwriteback(T)$$

If we terminate an instruction using cycle T, the number of issued instructions is greater than the number of terminated instructions. ◀ Lemma 6.8

$$writeback(T) \implies sIissue(T) > sIwriteback(T)$$

One easily shows this using lemma 6.7, and lemma 6.4, and the fact that we only writeback if the ROB is not empty.

The number of issued instructions upto cycle T is greater or equal than the number of terminated instructions upto cycle $T+1$. ◀ Lemma 6.9

$$sIissue(T) \geq sIwriteback(T+1)$$

One easily verifies this claim using lemma 6.8 for the case $writeback(T)$ and using lemma 6.7 otherwise.

As described above, we assign a tag to each instruction during issue. This is the value of the ROB tail pointer. This pointer is increased by one each time we issue an instruction. Thus, we define a function $I\_tag(i)$, which denotes the tag of instruction I_i, as follows: ◀ Definition 6.2 $I\_tag(i)$

$$I\_tag(i) := 0^{\vartheta} \oplus i$$

$I\_tag(i)$ is the value of the ROB tail pointer during issue of instruction I_i. ◀ Lemma 6.10

$$ROBtail^T = I\_tag(sIissue(T))$$

This claim is easily concluded using lemma 6.1 and the definition of I_tag.

Lemma 6.11 ▶ If an instruction is in ROB entry tag, then the tag of that instruction is tag.

$$sIROB(tag, T) = i \implies tag = I\_tag(i)$$

PROOF One shows this claim by induction on T. For $T = 0$, there is nothing to show since there is no instruction in the ROB (formally, $sIROB(tag, 0)$ is -1, and there is no instruction I_{-1}).

For $T + 1$, the claim is concluded by expanding the definition of *sIROB*. If

$$issue(T) \wedge tag = ROBtail^T$$

holds, we have $sIROB(tag, T + 1) = sIissue(T)$. The claim is then concluded using lemma 6.10.

If not so, we have $sIROB(tag, T + 1) = sIROB(tag, T)$. The claim is then concluded using the induction premise. QED

We will now show that this tag is unique beginning with the cycle the instruction is issued until the instruction terminates. Formally, this means that we can assign a single, unique instruction to each such tag.

Let $issued(i, T)$ hold iff instruction I_i is already issued during cycle T. We define this predicate using the scheduling function *sIissue*:

$$issued(i, T) \quad :\Longleftrightarrow \quad sIissue(T) > i$$

However, it is not obvious that instruction I_i was issued before cycle T if $sIissue(T) > i$ and vice-versa. It is an implication of in-order issue. The following lemma asserts one direction.

Lemma 6.12 ▶ If $issued(i, T)$ holds, there is a cycle $T' < T$ such that I_i is issued during cycle T'.

$$issued(i, T) \implies \exists T' < T : sIissue(T') = i \wedge issue(T')$$

PROOF The claim is shown by induction on T. For $T = 0$, we have $sIissue(0) = 0$. Thus, $sIissue(0) > i$ cannot hold and there is nothing to show.

For $T+1$, we show the claim using a case split on $issue(T)$.

- If $issue(T)$ holds, we have

$$sIissue(T+1) = sIissue(T) + 1$$

  and therefore $sIissue(T)+1 > i$. Let $sIissue(T) > i$ hold. In this case, we can apply the induction premise and the claim holds. Thus, let $sIissue(T) = i$ hold. In this case, cycle T satisfies the claim.

- If $issue(T)$ does not hold, we have $sIissue(T+1) = sIissue(T)$ and we can apply the induction premise to show the claim. QED

In analogy to $issued(i,T)$, we define a predicate $terminated(i,T)$ that holds iff instruction I_i already terminated before cycle T.

$$terminated(i,T) \quad :\Longleftrightarrow \quad sIwriteback(T) > i$$

Let the predicate $\tau(i,T)$ be a shorthand for the fact that instruction I_i is already issued during cycle T but has not yet terminated.

$$\tau(i,T) \quad :\Longleftrightarrow \quad issued(i,T) \wedge \overline{terminated(i,T)}$$

The following lemma will be used in order to show that issue is done in program order.

Consider the instruction in the issue stage during cycle T. During cycle $T+1$, there is the same or a later instruction in the issue stage. ◀ Lemma 6.13

$$sIissue(T+1) \quad \geq \quad sIissue(T)$$

The proof of lemma 6.13 is easily done by expanding the definition of the scheduling function $sIissue(T+1)$.

The instructions are issued in order, i.e., during cycle $T' \leq T$ there is the same or an earlier instruction in the issue stage. ◀ Lemma 6.14

$$\forall T' \leq T : sIissue(T') \leq sIissue(T)$$

This lemma is easily shown using induction on T and lemma 6.13 as induction step.

Lemma 6.15 ▶ Let $i \geq 0$ and $j \geq 0$ hold. If one increments a tag i times and after that j times, this is equivalent to incrementing the tag $i+j$ times.

$$(tag \oplus i) \oplus j \quad = \quad tag \oplus (i+j)$$

This is easily shown by induction on j.

Lemma 6.16 ▶ Let T and $T' \geq T$ be cycles. $ROBtail^{T'}$ is equal to $ROBtail^{T}$ incremented $sIissue(T') - sIissue(T)$ times.

$$\forall T' \geq T: \qquad ROBtail^{T'} = ROBtail^{T} \oplus (sIissue(T') - sIissue(T))$$

PROOF By applying lemma 6.1 twice, the claim is transformed into:

$$0^{\vartheta} \oplus sIissue(T') \quad \stackrel{!}{=} \quad (0^{\vartheta} \oplus sIissue(T)) \oplus (sIissue(T') - sIissue(T))$$

One shows $sIissue(T') - sIissue(T) \geq 0$ using lemma 6.14. This allows concluding the claim using lemma 6.15. **QED**

One easily verifies the following property of tag arithmetic (i.e., bit-vector arithmetic). It applies for incrementing tags as done for *ROBhead* and *ROBtail*.

Lemma 6.17 ▶ If one increments a tag i times, the value of this tag is the value of the old tag plus i modulo Θ (number of ROB entries).

$$\langle tag \oplus i \rangle \quad = \quad \langle tag \rangle + i \bmod \Theta$$

The following lemma will be used in order to argue that certain entries in the ROB are not overwritten.

Lemma 6.18 ▶ If one increments a tag at least once and less than Θ times, the incremented tag is different from the old tag.

$$0 < j < \Theta \quad \Longrightarrow \quad (tag \oplus j) \neq tag$$

PROOF According to lemma 6.17, we have

$$\langle tag \oplus j \rangle \quad = \quad \langle tag \rangle + j \bmod \Theta$$

Assume $(tag \oplus j) = tag$ holds. In this case, the equation above transforms into:

$$\langle tag \rangle \quad = \quad \langle tag \rangle + j \bmod \Theta$$

This only holds if j is a multiple of Θ (this property of mod is shown in the PVS libraries). This is a contradiction to the premise of the lemma and we therefore have $(tag \oplus j) \neq tag$. QED

Entries in the ROB are overwritten if the ROB tail pointer wraps around. This happens each Θ (number of ROB entries) instructions. The following lemma asserts the fact that instruction I_i in the ROB is overwritten only in this case.

◀ Lemma 6.19 Let instruction I_i be issued during cycle T'. Consider cycles $T > T'$. As long as no more than Θ instructions are issued from cycle T' to T, the instruction in the ROB entry during cycle T that $ROBtail^{T'}$ points to is instruction i.

$$\begin{array}{ll} & issue(T') \wedge sIissue(T') = i \wedge sIissue(T) \leq (i + \Theta) \\ \Longrightarrow & sIROB(ROBtail^{T'}, T) = i \end{array}$$

PROOF The proof proceeds by induction on T. For $T = 0$, there is nothing to show since there is no cycle $T' \geq 0$ with $T > T'$.

For $T + 1$, let us consider the case $T = T'$. In this case, the claim holds by definition of *sIROB*.

The claim for the case $T > T'$ is (we swap left hand side and right and side):

$$\begin{array}{lll} i & \stackrel{!}{=} & sIROB(ROBtail^{T'}, T) \\ & \stackrel{!}{=} & \begin{cases} sIissue(T) & : \; issue(T) \wedge \\ & \quad ROBtail^{T'} = ROBtail^{T} \\ sIROB(ROBtail^{T'}, T) & : \; \text{otherwise} \end{cases} \end{array}$$

We argue the two cases above separately. Assume

$$issue(T) \; \wedge \; ROBtail^{T'} = ROBtail^{T}$$

holds. This implies that $sIissue(T+1) = sIissue(T)+1$ holds because $issue(T)$ holds. This allows concluding that

$$sIissue(T)+1 \leq i+\Theta$$

holds. This allows applying lemma 6.18 with $j = sIissue(T)-i$, which states:

$$ROBtail^{T'} \neq ROBtail(T') \oplus (sIissue(T)-i)$$

According to lemma 6.16 for cycles T' and T, we have

$$ROBtail^{T} = ROBtail^{T'} \oplus (sIissue(T)-i).$$

Thus, this is a contradiction to $ROBtail^{T} = ROBtail^{T'}$. Thus,

$$issue(T) \wedge ROBtail^{T'} = ROBtail^{T}$$

cannot hold. We therefore only have to show $sIROB(ROBtail^{T'}, T) = i$. This is done using the induction premise.

QED

Lemma 6.20 ▶ If instruction I_i has been issued but has not not yet terminated, less than Θ (number of ROB entries) instructions have been issued since I_i was issued.

$$\tau(i,T) \implies sIissue(T) \leq i+\Theta$$

This claim is easily concluded using lemma 6.6.

The following theorem provides the unique mapping from tags to instructions: we just use the ROB schedule. The tag of an instruction is unique, if the instruction in the ROB.

Theorem 6.21 ▶ If instruction I_i has been issued but has not not yet terminated, the instruction in ROB entry $I\_tag(i)$ is instruction i.

$$\tau(i,T) \implies sIROB(I\_tag(i),T) = i$$

PROOF According to lemma 6.12, there is a cycle $T' < T$ such that instruction I_i is issued during cycle T'. According to lemma 6.19 for cycle T' and T and instruction i, we have:

$$sIissue(T) \leq i+\Theta \implies sIROB(ROBtail^{T'}, T) = i$$

We assert the left hand side of the implication using lemma 6.20. Thus, we have:

$$sIROB(ROBtail^{T'}, T) \quad = \quad i$$

It is therefore left to show that $ROBtail^{T'}$ is equal to $I\_tag(i)$. This is done using lemma 6.10. QED

From lemma 6.21, one easily concludes the following claim:

◀ Lemma 6.22 Let I_i and I_j be instructions. If the tags of the instructions are equal and both unique, instruction i is instruction j.

$$I\_tag(i) = I\_tag(j) \wedge \tau(i,T) \wedge \tau(j,T) \quad \Longrightarrow \quad i = j$$

In analogy to lemma 6.10, we show:

◀ Lemma 6.23 The *ROBhead* pointer during cycle T is the tag of the instruction in write-back stage.

$$ROBhead(T) \quad = \quad I\_tag(sIwriteback(T))$$

One easily concludes this claim using lemma 6.2

◀ Lemma 6.24 If we writeback an instruction during cycle T, that instruction is in the ROB entry that *ROBhead* points to.

$$writeback(T) \quad \Longrightarrow \quad sIwriteback(T) = sIROB(ROBhead(T), T)$$

PROOF Using lemma 6.23, we transform the claim into:

$$writeback(T) \quad \Longrightarrow \quad sIwriteback(T) = sIROB(I\_tag(sIwriteback(T)), T)$$

The claim is concluded using lemma 6.21. It is left to show that the premise of lemma 6.21 holds, i.e., we have to show that

$$\tau(sIwriteback(T), T)$$

holds. We show that the instruction is already issued using lemma 6.8. Furthermore, the instruction is obviously not terminated yet. QED

6.5.5 Instruction Phases

We distinguish the following phases of executing instruction I_i:

- **Not issued:** Before an instruction is issued, the instruction is in the "not issued" phase. Formally, this holds if $\overline{issued(i,T)}$ holds.
- **In RS:** During issue, the instruction is stored in a reservation station unless $issue\_with\_result(i)$ holds. Formally, instruction I_i is in a reservation station during cycle T iff

$$\exists rs:\ RS[rs]^T.full \wedge sIRS(rs,T) = i$$

holds.

- **In FU:** During dispatch, the instruction is passed from the reservation station to a function unit. Formally, we say an instruction is dispatched during cycle T iff there is a cycle $T' < T$ and a reservation station *rs* such that instruction I_i is in reservation station *rs* and the instruction in that reservation station is dispatched.

$$\begin{aligned} & dispatched(i,T) \\ :\Longleftrightarrow\quad & \exists T' < T, rs:\ dispatch\_rs(T', rs) \wedge sIRS(rs,T') = i \end{aligned}$$

The instruction leaves the function unit if it is passed to a producer. Formally, an instruction is executed iff there is a cycle $T' \leq T$ and a producer fu such that instruction I_i is in the producer fu and that producer is full.

$$\begin{aligned} & executed(i,T) \\ :\Longleftrightarrow\quad & \exists T' < T, fu:\ FU[fu]^{T'}.valid \wedge sIfu(fu,T') = i \end{aligned}$$

Formally, instruction I_i is in a function unit during cycle T iff

$$dispatched(i,T) \wedge \overline{executed(i,T)}$$

holds. Note that there are function units (ALU, for example), that return the result in the same cycle they get it. In this case, the condition above never holds, although the function unit is not bypassed.

- **In producer:** After leaving the function unit, the result of the instruction is stored in a producer. Formally, an instruction is in a producer iff there is a producer fu such that instruction I_i is in the producer fu and the producer is full.

$$\exists fu:\ P[fu]^T.full \wedge sIP(fu,T) = i$$

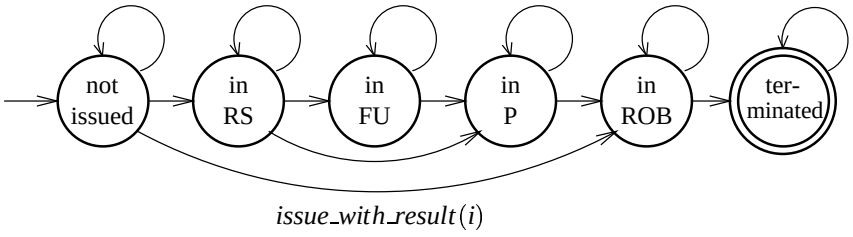

Figure 6.9 Instruction phase state diagram

- **In ROB:** As soon as the producer gets the CDB, the result in the producer is stored in the ROB. Formally, an instruction is in the ROB during cycle T iff there is a ROB entry *tag* such that the instruction in that entry is I_i and the entry is valid and the instruction has not terminated yet.

$$\exists\, tag:\ ROB[tag]^T.valid \wedge sIROB(tag,T) = i \wedge \overline{terminated(i,T)}$$

The phases of "normal" instructions, i.e., instructions I_i that are not issued with result, are processed in the order above. Instructions with $issue\_with\_result(i)$ skip the phases "in RS", "in FU", and "in producer". This is illustrated in figure 6.9. The figure shows the different phases and the transitions between the phases. However, one has to assert this property of the machine. This is done by the following lemmas.

Let $p(i,T)$ denote that instruction I_i is in phase p during cycle T.

Let $pred(p)$ denote the set of predecessor phases of phase p according to figure 6.9. For example, the "not issued" phase only has itself as predecessor. The "in ROB" phase has three predecessor phases: "in ROB", "not issued", and "in producer".

In analogy to $pred(p)$, let $succ(p)$ denote the set of successor phases of phase p according to figure 6.9. For example, the "not issued" phase has two successor phases: "in RS" and "in ROB".

◀ Lemma 6.25

If instruction I_i is in a given phase during cycle T, and not in any other phase, we show that the instruction is in at most one successor phase during cycle $T+1$, i.e., the sucessor phases mutually exclude each other.

PROOF

For most phases, the claim is trivial, because they only thave themselves

and another state as successors. The only exception is the "not issued" phase, which has three successors. We therefore show the claim exemplary for the "not issued" phase.

- If $issue(T)$ and $sIissue(T) = i$ does not hold, one easily concludes that instruction I_i stays in "not issued" phase during cycle $T+1$. Thus, we have to show that it is not in a reservation station or in the ROB. According to the premise of the lemma, the phases of I_i are unique during cycle T. Thus, I_i is not in the ROB or in a reservation station during cycle T. Since I_i is also not issued, one easily verifies that it does not move into the ROB or into a reservation station.
- If $issue(T)$ and $sIissue(T) = i$ holds, one easily concludes that instruction I_i either enteres the ROB or a reservation station, depending on $issue\_with\_result(i)$. If $issue\_with\_result(i)$ holds, one verifies that the instruction cannot be in a reservation station. If not so, one verifies that the instruction cannot be in the ROB.

QED

Lemma 6.26 ▶ If instruction I_i is in a given phase during cycle $T+1$, we show that it must have been in one of the predecessor phases as given in figure 6.9 during cycle T:

$$p(i, T+1) \implies \bigvee_{p' \in pred(p)} p'(i, T)$$

For example, if instruction I_i is in phase "not issued" during cycle $T+1$, this implies that it must be in phase "not issued" during cycle T.

PROOF In PVS, we split this claim into 6 lemmas, one for each phase. We show the claim for the "not issued" phase and the "in RS" phase here exemplary.

- The claim for the "not issued" phase is easily asserted by expanding the definition of "not issued" and by applying lemma 6.13.
- The claim for the "in RS" phase is asserted as follows: according to the premise, there is a reservation station *rs* such that

  $$RS[rs]^{T+1}.full \wedge sIRS(rs, T+1) = i$$

  holds. Let $issue\_rs(T, rs)$ hold. In this case, we have

  $$sIRS(rs, T+1) = sIissue(T)$$

Thus, the instruction I_i is in issue stage during cycle T. Thus, it is in "not issued" phase during cycle T, which concludes the claim.

Let $issue\_rs(T, rs)$ not hold. In this case, one easily asserts that the full bit $RS[rs]^T.full$ is active and $sIRS(rs, T) = i$ holds. Thus, the instruction is in "in RS" phase during cycle T, which concludes the claim.

QED

◀ Lemma 6.27

The phase of instruction I_i during cycle T is unique, i.e., the phases above exclude each other mutually.

PROOF

One easily shows this claim by induction on T. For $T = 0$, one asserts that all instructions are in "not issued" phase only.

For $T + 1$, one shows the claim as follows: according to the induction premise, instruction I_i is in at most one phase during cycle T. One applies lemma 6.25, which shows that the successor states mutually exclude each other.

Furthermore, the instruction I_i cannot be in a phase that is not a successor phase during cycle $T + 1$, which is asserted by lemma 6.26.

QED

6.5.6 Tag Consistency

We will now show that the tags transported in the machine are consistent with the scheduling functions, i.e., we will show that the tag stored together with instruction I_i is $I\_tag(i)$.

◀ Lemma 6.28

If a reservation station is full, the tag in that reservation station is the tag of the instruction in the reservation station.

$$RS[rs]^T.full \implies RS[rs]^T.tag = I\_tag(sIRS(rs, T))$$

PROOF

The claim is shown using induction on T. For $T = 0$ there is nothing to show because the reservation stations are not full in the initial configuration.

For $T + 1$, we show the claim as follows: If an instruction I_i is issued into reservation station *rs* during cycle T, the value of the tag in reservation

station is defined by the issue protocol:

$$RS[rs]^{T+1}.tag \quad = \quad ROBtail^T$$

According to lemma 6.1, this is equivalent to $0^\vartheta \oplus sIissue(T)$. This is the definition of $I\_tag(i)$.

If no instruction is issued into reservation station *rs* during cycle T, we apply the induction premise. QED

Lemma 6.29 ▶ If there is an instruction in a producer, the tag in the producer matches the tag of the instruction.

$$P[fu]^T.full \implies P[fu]^T.tag = I\_tag(sIP(fu,T))$$

PROOF We show this claim by induction on T. For $T = 0$, there is nothing to show because the producer is not full in the initial configuration.

For $T+1$, we show the claim as follows: For the case that the instruction in the producer did not change from cycle T to $T+1$, we apply the induction premise.

If a new instruction moved into the producer, we conclude the claim by making the following assumption: if the function unit gets correct tags as inputs for cycles T' with $T' \leq T$, this implies that the function unit passes the correct tag during cycle T. We will later on describe how to verify that property of the function units. We show that the function units get correct tags for T' with $T' \leq T$ using lemma 6.28. QED

Lemma 6.30 ▶ The tag on the CDB matches the tag of the instruction on the CDB.

$$CDB^T.valid \implies CDB^T.tag = I\_tag(sICDB(T))$$

PROOF We assume that we only complete instructions from producers that are full. Thus, we can apply lemma 6.29. The tag on the CDB matches the tag from the producer. Furthermore, the instruction on the CDB matches the instruction in the producer, by definition of *sICDB*. QED

6.5.7 Data Consistency Criterion

In this section we describe our data consistency criterion for the Tomasulo protocols. We define a formal notion for the correct input and output values of an instruction. We do this by defining an abstract machine that processes an instruction with each transition. We call this machine abstract specification machine (aS). The configuration set of this machine consists of the registers.

Given an instruction (configuration of this machine), we define the correct value of a source register r to be the value of the register r if $r \neq 0$ and to be zero if $r = 0$:

$$source(i,r) \quad := \quad \begin{cases} 0 & : \quad r = 0 \\ c^i_{aS}.R & : \quad \text{otherwise} \end{cases}$$

The function $source(i)$ maps an instruction to the values of all source operands. Remember that $S(i,x)$ denotes the number of the register of source operand x. Let s denote the number of source registers.

$$source : \mathbb{N} \longrightarrow \mathcal{W}(R)^s$$

$$source(i)(x) \quad := \quad source(i, S(i,x))$$

Let f_i be the function that maps the values of the source operands of instruction I_i to the values of the destination operands unless we have $issue\_with\_result(i)$. Let d denote the number of destination registers.

$$f_i : \mathcal{W}(R)^s \longrightarrow \mathcal{W}(R)^d$$

Thus, the result of instruction I_i is:

$$result(i,r) \quad := \quad \begin{cases} issue\_result(i) & : \quad issue\_with\_result(i) \\ f_i(source(i)) & : \quad \text{otherwise} \end{cases}$$

This allows defining the configurations of the abstract specification machine. We start with an initial configuration c^0_{aS} and proceed using f. If instruction $i-1$ has register r as destination register, then we take the the new value of $R[r]$ from the result of I_{i-1}. If not so, we take the value from the old configuration.

$$c^i_{aS}.R[r] \quad := \quad \begin{cases} c^0_{aS}.R[r] & : \quad i = 0 \\ result(i-1)[e(r)] & : \quad i \neq 0 \wedge dest(i-1,r) \\ c^{i-1}_{aS} & : \quad \text{otherwise} \end{cases}$$

Proof Strategy We will show the correctness of a DLX implementation with Tomasulo scheduler as follows:

- We will show that a machine implementing the Tomasulo protocols given in the previous sections simulates the abstract machine *aS*. This is the hardest part of the proof.
- We will show that the DLX implementation with Tomasulo scheduler implements the Tomasulo protocols.

We will now conclude several trivial properties of the abstract specification machine *aS*.

Lemma 6.31 ▶ If instruction I_i has no destination register $R[r]$, then $R[r]$ is not changed by instruction I_i.

$$\overline{dest(i,r)} \implies R[r]^{i+1}_{aS} = R[r]^{i}_{aS}$$

The proof is done by expanding the definition of $R[r]^{i+1}_{aS}$.

Definition 6.3 ▶ $\mathcal{L}(i,r)$ Let the predicate $\mathcal{L}(i,r)$ hold iff there is an instruction $j < i$ such that instruction I_j has destination register r.

$$\mathcal{L}(i,r) \quad :\Longleftrightarrow \quad \exists j < i : dest(j,r)$$

Lemma 6.32 ▶ Let i and $j \leq i$ be instructions. If $\mathcal{L}(j,r)$ holds, so does $\mathcal{L}(i,r)$.

$$j \leq i \wedge \mathcal{L}(j,r)) \implies \mathcal{L}(i,r)$$

This holds by definition of the predicates.

Definition 6.4 ▶ $last(i,r)$ Let $\mathcal{L}(i,r)$ hold. Let $last(i,r)$ denote the number of the last instruction with destination register r prior to instruction I_i. Formally, this is the maximum of the set of instructions I_j with $j < i$ and $dest(j,r)$.

$$last(i,r) \quad := \quad max\{j \mid j < i \wedge dest(j,r)\}$$

This set is always non-empty because of $\mathcal{L}(i,r)$. Furthermore, the set is finite and has an upper bound. Thus, the maximum is defined if $\mathcal{L}(i,r)$ holds.

The following property is easily shown using the definition of *last* and the definition of max.

If $\mathcal{L}(i,r)$ holds, the instruction $I_{last(i,r)}$ has destination register r. ◀ Lemma 6.33

$$\mathcal{L}(i,r) \implies dest(last(i,r),r)$$

Let $\mathcal{L}(i,r)$ and $i \geq 1$ hold. If instruction I_{i-1} does not have a destination register r, $\mathcal{L}(i-1,r)$ holds. ◀ Lemma 6.34

$$i \geq 1 \wedge \mathcal{L}(i,r) \wedge \overline{dest(i-1,r)} \implies \mathcal{L}(i-1,r)$$

PROOF

Because $\mathcal{L}(i,r)$ holds, there must be an instruction I_j with $j < i$ and $dest(j,r)$. Since this is not instruction $i-1$, it must be an instruction with $j < i-1$. Thus, $\mathcal{L}(i-1,r)$ holds.

Let $i \geq 1$ and $\mathcal{L}(i,r)$ hold. If instruction I_{i-1} does not have a destination register r, then $last(i,r)$ is equal to $last(i-1,r)$. ◀ Lemma 6.35

$$i \geq 1 \wedge \mathcal{L}(i,r) \wedge \overline{dest(i-1,r)} \implies last(i,r) = last(i-1,r)$$

PROOF

Because of $\mathcal{L}(i,r)$, $last(i,r)$ is defined. According to lemma 6.34, $\mathcal{L}(i-1,r)$ holds. Thus, $last(i-1,r)$ is defined.

Let j be $last(i,r)$. By definition of max, this number is element of $\{0,\ldots,i-1\}$. Because of $\overline{dest(i-1,r)}$, j cannot be $i-1$. Thus, j is equal to $last(i-1,r)$.

QED

Let $i \geq 1$ hold. If instruction I_{i-1} has destination register r, $last(i,r)$ is equal to $i-1$. ◀ Lemma 6.36

$$i \geq 1 \wedge dest(i-1,r) \implies last(i,r) = i-1$$

This is easily shown by using the definition of max.

Let I_i and I_j with $j \leq i$ be instructions. If all instructions $I_{j'}$ with $j \leq j' < i$ do not have a destination register r, the value of $R[r]$ does not change from configuration c^i_{aS} to c^j_{aS}. ◀ Lemma 6.37

$$j \leq i \wedge (\forall\, j \leq j' < i : \overline{dest(j',r)} \implies R[r]^i_{aS} = R[r]^j_{aS}$$

One easily concludes this using induction on i and the transition function of $R[r]$.

Lemma 6.38 ▶ Let $R[r]$ with $r \neq 0$ be a register and let $\mathcal{L}(i,r)$ hold. In this case, the correct source register of I_i is the result of the last instruction writing $R[r]$.

$$r \neq 0 \wedge \mathcal{L}(i,r) \implies source(i,r) = result(last(i,r))[e(r)]$$

PROOF By definition of $last(i,r)$, the instructions I_j with $last(i,r) < j < i$ do not have destination register r. According to lemma 6.37, we have

$$R[r]^{i}_{aS} = R[r]^{last(i,r)+1}_{aS}$$

QED The left hand side is $source(i,r)$ by definition, and the right hand side is $result(last(i,r))[e(r)]$ by definition of $R[r]^{last(i,r)+1}_{aS}$.

Lemma 6.39 ▶ Let there not be an instruction that is issued during cycle T with destination $R[r]$. This implies that the value of source register r of instruction $I_{issue(T)}$ matches the value of source register r of instruction $I_{issue(T+1)}$.

$$\begin{array}{ll} & \overline{issue(T) \wedge dest(sIissue(T),r)} \\ \implies & source(sIissue(T),r) = source(sIissue(T+1),r) \end{array}$$

PROOF If $issue(T)$ does not hold, we have $sIissue(T) = sIissue(T+1)$ and the claim obviously holds.

QED If $issue(T)$ holds, we apply lemma 6.37 and expand the definition of *source*.

6.5.8 Forwarding Tags Consistency

The Tomasulo scheduling algorithm does forwarding at two places: 1) during issue, we forward from the CDB and from the ROB, 2) while in a reservation station, we forward from the CDB.

Both forwarding from the ROB and from the CDB is done using the tag. We will now show that the tags used for forwarding are correct.

◀ Lemma 6.40

Let I_i be the instruction in issue stage during cycle T. If a register $R[r]$ is marked as "not valid" during cycle T in the producer table, there is an instruction prior to instruction I_i that writes $R[r]$ and the tag of the register in the producer table is the tag of the last instruction prior instruction $I_{sIissue(T)}$ writing $R[r]$.

$$\begin{aligned} & sIissue(T) = i \wedge \overline{R[r]^T.valid} \\ \Longrightarrow \quad & \mathcal{L}(i,r) \wedge R[r]^T.tag = I\_tag(last(i,r)) \end{aligned}$$

PROOF

We verify that claim by induction on T. For $T = 0$, there is nothing to show because we make the valid bits of the registers active in the initial configuration.

For $T+1$, we conclude the claim as follows: In case $R[r]^{T+1}.valid$ holds, there is nothing to show. Thus, let $R[r]^{T+1}.valid$ not hold. We distinguish three cases:

- If an instruction with destination register $R[r]$ is issued during cycle T, we easily assert $\mathcal{L}(i,r)$, since instruction $sIissue(T)$ satisfies the claim.

  We assert $R[r]^T.tag = I\_tag(last(i,r))$ as follows: we apply lemma 6.36, which states:

  $$last(i,r) \quad = \quad i-1$$

  Thus, we have to show:

  $$R[r]^{T+1}.tag \quad \stackrel{!}{=} \quad I\_tag(i-1)$$

  During issue, the ROB tail pointer is stored in $R[r].tag$. Thus, the claim is equivalent to:

  $$ROBtail^T \quad \stackrel{!}{=} \quad I\_tag(i-1)$$

  According to the definition of I_tag and lemma 6.1, this is equivalent to:

  $$\begin{aligned} 0^\theta \oplus sIissue(T) \quad & \stackrel{!}{=} \quad 0^\theta \oplus (i-1) \\ sIissue(T) \quad & \stackrel{!}{=} \quad i-1 \end{aligned}$$

This is concluded using the fact that $i = issue(T+1)$ holds, and by expanding the definition of $issue(T+1)$, and the fact that $issue(T)$ holds.

- If an instruction with no destination register $R[r]$ is issued during cycle T, consider $R[r]^T.valid$. If $R[r]^T.valid$ holds, this implies that $R[r]^{T+1}.valid$, which is a contradiction.

  Thus, $R[r]^T.valid$ does not hold. This allows applying the induction premise for instruction I_{i-1} and we get:

  $$\mathcal{L}(i-1,r) \wedge R[r]^T.tag = I\_tag(last(i-1,r))$$

  We conclude $\mathcal{L}(i,r)$ from $\mathcal{L}(i-1,r)$ using lemma 6.32.

  As the instruction that is issued during cycle T does not have a destination register $R[r]$, we have $R[r]^{T+1}.tag = R[r]^T.tag$, which transforms the claim into:

  $$R[r]^T.tag \stackrel{!}{=} I\_tag(last(i,r))$$

  Thus, it is left to show that $last(i-1,r) = last(i,r)$ holds. This is concluded using lemma 6.35.

- If no instruction is issued during cycle T, we assert that $R[r]^T.valid$ does not hold as in the case above. This allows applying the induction premise, which concludes the claim.

QED

The following lemma will be used for the induction step for the proof of lemma 6.42.

Lemma 6.41 ▶ Let reservation station *rs* be full during cycle $T+1$ and let the operand x be not valid. There are two possible reasons for this: 1) this was already true during cycle T, and 2) an instruction was issued into the reservation station during cycle T.

$$\begin{array}{ll} & RS[rs]^{T+1}.full \wedge \overline{RS[rs]^{T+1}.op[x].valid} \\ \Longrightarrow & (RS[rs]^T.full \wedge \overline{RS[rs]^T.op[x].valid}) \vee \\ & (issue(T) \wedge issue\_rs(T,rs)) \end{array}$$

One easily asserts this claim by applying the definition of the issue protocol. Full bits of reservation stations are only set by the issue protocol, the valid bit of the operand is only cleared by the issue protocol.

The following lemma will be used to argue the correctness of data that is forwarded into a reservation station.

◀ Lemma 6.42

Let reservation station rs be full and let instruction I_i be in this reservation station. Let operand x be not valid, and let r be $S(i,x)$. This implies that r is not zero, and that there is an instruction prior to instruction I_i with destination $R[r]$ and the tag of operand x is the tag of the last instruction prior to I_i with destination $R[r]$.

$$\begin{aligned} & RS[rs]^T.full \wedge sIRS(rs,T) = i \wedge \overline{RS[rs]^T.op[x].valid} \\ \Longrightarrow \quad & r \neq 0 \wedge \mathcal{L}(i,r) \wedge RS[rs]^T.op[x].tag = I\_tag(last(i,r))) \end{aligned}$$

PROOF

One asserts this claim by induction on T. For $T = 0$, there is nothing to show since the full bits of the reservation stations are not set in the initial configuration.

For $T+1$, we show the claim by applying lemma 6.41. Consider the case that an instruction is issued into the reservation station during cycle T. In this case, the claim is easily concluded using lemma 6.40 (correctness of the tags in the producer tables).

If no instruction is issued into the reservation station during cycle T, the tag in the reservation station does not change and we have

$$RS[rs]^T.full \wedge \overline{RS[rs]^T.op[x].valid}$$

according to lemma 6.41. This allows applying the induction premise, which concludes the claim.

QED

6.5.9 Tag Uniqueness

We will now show the tag uniqueness properties for the different places tags are used in the Tomasulo machine.

Recall that this property was shown in lemma 6.21. This lemma uses $\tau(i,T)$ as premise. Thus, we use "tag is unique" and $\tau(i,T)$ synonymously.

◀ Lemma 6.43

Let I_i be the instruction in issue stage and let the valid bit of register $R[r]$ be not set. This implies that there is an instruction prior to I_i writing $R[r]$ and the tag of the last such instruction is unique.

$$sIissue(T) = i \wedge \overline{R[r]^T.valid} \quad \Longrightarrow \quad \mathcal{L}(i,r) \wedge \tau(last(i,r),T)$$

PROOF This claim is concluded by induction on T. For $T = 0$, there is nothing to show since we make the valid bits of all registers set in the initial configuration.

For $T+1$, we apply lemma 6.40, which states that there is an instruction prior to I_i writing $R[r]$ and that the tag in the producer table is the tag of instruction $j := last(i,r)$. In order to show the uniqueness of the tag, we have to assert that instruction I_j is already issued but not yet terminated.

One easily asserts that instruction I_j is already issued by definition of $last(i,r)$.

We show that instruction I_j is not yet terminated by distinguishing two cases:

1. If an instruction with destination $R[r]$ is issued during cycle T, we show that $j = i-1$ holds using lemma 6.36. This instruction cannot be terminated in cycle $T+1$, because this is a contradiction to lemma 6.9.

2. If no instruction with destination $R[r]$ is issued during cycle T, we assert that the valid bit of register $R[r]$ is not set during cycle T:

$$\overline{R[r]^T.valid}$$

This allows applying the induction premise for the instruction issued during cycle T (instruction $sIissue(T)$). Thus, we have:

$$\tau(last(sIissue(T),r),T)$$

If $issue(T)$ does not hold, we have $sIissue(T) = i$ and the claim is concluded. Thus, let $issue(T)$ hold. We already showed the claim for the case that instruction $sIissue(T)$ has destination register $R[r]$. For the case it does not have such a destination register, we apply lemma 6.35, which states that

$$last(i,r) \quad = \quad last(sIissue(T),r)$$

holds. Thus, we have:

$$\tau(j,T)$$

We therefore know that instruction I_j did not terminate before cycle T. It is left show show that it does not terminate during cycle T. Assume it does terminate during cycle T. One easily asserts that the

tag in the producer table of register $R[r]$ is the tag of instruction I_j since it is unique according to the induction premise.

Thus, according to the writeback protocol, the valid bit of $R[r]$ is set during cycle T. This is a contradiction to the fact that $R[r]^{T+1}.valid$ does not hold. QED

◀ Lemma 6.44

Let I_i be in reservation station rs and let that reservation station be full. This implies that the tag of instruction I_i is unique.

$$RS[rs]^T.full \implies \tau(sIRS(rs,T),T)$$

PROOF

One easily concludes that instruction I_i is in phase "in RS", as formally defined above. According to lemma 6.27, the instruction cannot be in two different phases during cycle T. Thus, it cannot be in "not issued" phase, which allows concluding that it is already issued.

Furthermore, it cannot be in "terminated" phase. Thus, $\tau(i,T)$ holds. QED

◀ Lemma 6.45

Let I_i be an instruction in a full reservation station. Let x be a source operand that is not valid, and $r := S(i,x)$ be the source register. There is an instruction prior to I_i writing $R[r]$. Let I_j be the last instruction prior to instruction I_i that writes $R[r]$.

We claim that instruction I_j is in one of the following phases: 1) it is in a reservation station, 2) it is in a function unit, or 3) it is in a producer.

PROOF

This claim is shown by induction on T. For $T = 0$, there is nothing to show since the reservation stations are not full in the initial configuration.

For $T+1$, we conclude the clain as follows: According to lemma 6.41, there are two cases: an instruction is issued into reservation station rs during cycle T or the instruction already was in the reservation station during cycle T.

- If an instruction is issued into the reservation station during cycle T, one easily asserts that the valid bit of the source register cannot be active (otherwise, the valid bit of the reservation station source operand is set and we have nothing to show). This allows applying lemma 6.40, which states that the tag of the last instruction writing the register is in the producer table. According to lemma 6.43,

the tag is unique, i.e., instruction I_j is already issued and has not yet terminated. Futhermore, the instruction is not in the "in ROB" phase during cycle T and not on the CDB (otherwise, the valid bit of the reservation station source operand is set and we have nothing to show). Thus, it must be in a reservation station, function unit or producer during cycle T.

We conclude the claim as follows: if the instruction is in a reservation station, we use lemma 6.59 in order to conclude that it either stays in that phase or enters a function unit. This concludes the claim.

If the instruction is in a function unit, we use lemma 6.59 in order to conclude that it either stays in that phase or enters a producer. This concludes the claim.

If the instruction is in a producer, we use 6.59 in order to conclude that it either stays in that phase or moves into the ROB. The last case cannot happen, since this is a contradiction to the fact that the valid bit of the operand is not active. This is easily concluded since the tag of I_j is valid because the instruction is in the "in producer" phase.

- If no instruction is issued into the reservation station during cycle T, one applies the induction premise. The induction premise states that instruction I_j is in a reservation station, a function unit, or in a producer. After that, the claim is concluded as in the case above.

QED

Lemma 6.46 ▶ Let I_i be an instruction in a full reservation station. Let x be a source operand that is not valid, and $r := S(i,x)$ be the source register. There is an instruction prior to I_i writing $R[r]$. Let I_j be the last instruction prior to instruction I_i that writes $R[r]$. The tag of that instruction is unique.

$$\begin{aligned} & RS[rs]^T.full \wedge sIRS(rs,T) = i \wedge \overline{RS[rs]^T.op[x].valid} \\ \Longrightarrow \quad & \mathcal{L}(i,r) \wedge \tau(j,T) \end{aligned}$$

PROOF One easily asserts this lemma by applying lemma 6.45. According to lemma 6.27, the phases exclude each other. Thus, I_j cannot be in "not issued" or "terminated" phase, which concludes the claim.

QED

Lemma 6.47 ▶ Let I_i be in producer fu and let that producer be full. This implies that the tag of instruction I_i is unique.

$$P[fu]^T.full \quad \Longrightarrow \quad \tau(sIP(fu,T),T)$$

PROOF The instruction in the producer is in the "in producer" phase. According to lemma 6.27, the phases exclude each other. Thus, the instruction cannot be in "not issued" or "terminated" phase, which concludes the claim.

The tag of the instruction on the CDB is unique. ◀ Lemma 6.48

$$CDB^T.valid \implies \tau(sICDB(T), T)$$

One easily asserts this lemma by expanding the definition of $sICDB(T)$ and by applying lemma 6.47.

6.5.10 Data Consistency Invariants

In order to show data consistency, we claim a set of invariants. As done in the previous chapters, we will show that all these invariants hold by induction on T. The invariants are taken from [MPK00].

Let instruction I_i be in the issue stage. Let $r \neq 0$ be a register. Let the valid bit of register $R[r]$ be set. In this case, the register data is correct. ◀ Invariant 6.1

$$sIissue(T) = i \wedge r \neq 0 \wedge R[r]^T.valid \implies R[r]^T.data = source(i, r)$$

Let reservation station rs be full and let instruction I_i be in reservation station rs. If an input operand of the reservation station is valid, the value in the operand registers is the correct source operand of instruction I_i. ◀ Invariant 6.2

$$\begin{aligned} & sIRS(rs, T) = i \wedge RS[rs]^T.full \wedge RS[rs]^T.op[x].valid \\ \implies\ & RS[rs]^T.op[x].data = source(i)(x) \end{aligned}$$

After all operands are valid, the instruction is passed to the function unit. Once the instruction leaves the function unit, the result is stored in a producer. The following invariant asserts that the producer holds the correct result.

Let producer p be full and let instruction I_i be in producer fu. The result in this producer is the result of instruction I_i. ◀ Invariant 6.3

$$sIP(fu, T) = i \wedge P[fu]^T.full \implies P[fu]^T.result = result(i)$$

Once there is an instruction in a producer, the producer requests the CDB. After the request is acknowledged, the result is put on the CDB.

Invariant 6.4 ▶ Let I_i be on the CDB. The result on the CDB is the result of I_i.

$$sICDB(T) = i \wedge CDB^T.valid \implies CDB^T.result = result(i)$$

While on the CDB, the results are written into the ROB. The following invariant asserts that the results in the ROB are correct.

Invariant 6.5 ▶ Let I_i be in ROB entry tag and let that entry be valid. This implies that the result in the ROB entry is the result of instruction I_i.

$$\begin{aligned} & sIROB(tag, T) = i \wedge ROB[tag]^T.valid \\ \implies\ & ROB[tag]^T.result = result(i) \end{aligned}$$

We now show lemmas that form the induction step of the invariant proof.

Lemma 6.49 ▶ Let invariant 6.3 (producer data consistency) hold during cycle T. This implies that invariant 6.4 (CDB data consistency) holds during cycle T.

PROOF By definition, $CDB^T.valid$ only holds iff we complete an instruction, i.e., iff $completion(T)$ holds. The producer the instruction we complete is in, is denoted by $compl\_p(T)$. We assume that we only complete an instruction in a producer, if that producer is full. Thus,

$$P[compl\_p(T)]^T.full$$

holds. This allows applying invariant 6.3, which states that the result in the producer is correct:

$$P[compl\_p(T)]^T).result = result(sIP(compl\_p(T), T))$$

The term on the left hand side is the result on the CDB by definition.

$$CDB^T.result = result(sIP(T))$$

By definition of $sICDB(T)$, we have $sICDB(T) = sIP(compl\_p(T), T)$. This concludes the claim.

Let invariant 6.5 (ROB data consistency) and invariant 6.4 (CDB data consistency) hold during cycle T. This implies that invariant 6.5 (ROB data consistency) holds during cycle $T+1$. ◀ Lemma 6.50

In order to show the claim, we distinguish three cases: PROOF

1. Consider the case that an instruction is issued into ROB entry *tag* during cycle T, i.e., we have:

   $$issue(T) \wedge ROBtail^T = tag$$

   In this case, the ROB entry *tag* is valid iff we have the result of the instruction available during issue, i.e., if $issue\_with\_result(T)$ holds. Thus, there is nothing to show unless $issue\_with\_result(T)$ holds. We easily conclude that $sIROB(tag, T+1)$ is equal to $sIissue(T)$. Thus, the result in the ROB is correct by definition.

2. Consider the case that we do not issue an instruction into ROB entry *tag* during cycle T and that we receive a result from the CDB during cycle T, i.e.:

   $$CDB^T.valid \wedge CDB^T.tag = tag$$

   In this case, the result on the CDB is stored in the ROB and we have to argue its correctness:

   $$\begin{aligned} result(sIROB(tag, T+1)) &\stackrel{!}{=} ROB[tag]^{T+1}.result \\ &\stackrel{!}{=} CDB^T.result \end{aligned}$$

   According to invariant 6.4 (CDB data consistency), we have:

   $$CDB^T.result = result(sICDB(T))$$

   Thus, the claim holds if we show $sIROB(tag, T+1) = sICDB(T)$, i.e., it is left to show that the tag maps to the correct instruction. These arguments are weak in [MPK00].

   We show this formally using lemma 6.48. Lemma 6.48 states that

   $$\tau(sICDB(T), T)$$

holds. This allows applying theorem 6.21, which states:

$$sIROB(I\_tag(sICDB(T)), T) \quad = \quad sICDB(T)$$

Thus, it is left to show:

$$sIROB(tag, T+1) \quad \stackrel{!}{=} \quad sIROB(I\_tag(sICDB(T)), T)$$

According to lemma 6.30, we have $tag = I\_tag(sICDB(T))$. This transforms the claim into:

$$sIROB(tag, T+1) \quad \stackrel{!}{=} \quad sIROB(tag, T)$$

This is concluded by expanding the definition of $sIROB(tag, T+1)$.

3. Consider the case that no instruction is issued in ROB entry *tag* and that no result for ROB entry *tag* is on the CDB. We assert this case using invariant 6.5 for cycle T.

QED

Lemma 6.51 ▶ Let invariant 6.5 (ROB data consistency) and invariant 6.1 (register file data consistency) hold during cycle T. This implies that invariant 6.1 (register file data consistency) holds during cycle $T+1$.

PROOF We distinguish three cases:

1. Consider the case that we issue an instruction with destination r during cycle T. In this case, the valid bit $R[r]^{T+1}.valid$ cannot hold and there is nothing to show.

2. Consider the case that we writeback an instruction with destination r during cycle T and let the valid bit of $R[r]$ be not active during cycle T. We only do this writeback if the ROB entry that the ROB head pointer points to is valid. According to invariant 6.5, this implies that the result in the rob entry is the result of the instruction. This transforms the claim into:

$$result(sIROB(ROBhead^T, T))[e(r)] \quad \stackrel{!}{=} \quad source(i, r)$$

The tag of $R[r]$ matches the the ROB head pointer, since otherwise $R[r]^{T+1}.valid$ cannot hold and there is nothing to show.

According to lemma 6.40, that tag is equal to the tag of the last instruction prior to instruction $issue(T)$ that writes $R[r]$. This transforms the claim into:

$$result(sIROB(I\_tag(last(sIissue(T),r),T))[e(r)] \stackrel{!}{=} source(i,r)$$

According to lemma 6.43, that tag is unique. This allows applying lemma 6.21, which transforms the claim into:

$$result(last(sIissue(T),r))[e(r)] \stackrel{!}{=} source(i,r)$$

According to lemma 6.39, we have:

$$source(sIissue(T),r) = source(sIissue(T+1),r)$$

This transforms the claim into:

$$result(last(sIissue(T),r))[e(r)] \stackrel{!}{=} source(sIissue(T),r)$$

This is concluded using lemma 6.38.

3. If we neither issue an instruction with destination $R[r]$ nor write-back an instruction with destination $R[r]$ with $\overline{R[r]^T.valid}$, assume $R[r]^T.valid$ does not hold. In this case, valid bit $R[r]^{T+1}.valid$ cannot hold and there is nothing to show.

   Thus, $R[r]^{T+1}.valid$ holds. The claim is:

$$R[r]^T.data \stackrel{!}{=} source(i,r)$$

   After applying the induction premise, this is transformed into:

$$source(sIissue(T),r) \stackrel{!}{=} source(i,r)$$

   We assert this using lemma 6.39. QED

Let invariant 6.3 (producer data consistency) hold during cycle T and invariant 6.2 (reservation station data consistency) hold during cycles T' with $T' \leq T$. This implies that invariant 6.3 (producer data consistency) holds during cycle $T+1$. ◀ Lemma 6.52

PROOF One concludes this claim as follows: if an instruction moves into the producer during cycle T, we make the assumption that the function unit delivers a correct result given that it got correct inputs during all cycles $T' \leq T$. This is easily asserted using invariant 6.2 (reservation station data consistency). For this, we have to assume that we only dispatch instructions with valid operands.

If no instruction moves into the producer during cycle T, we conclude

$$sIP(fu, T) \quad = \quad sIP(fu, T+1).$$

Furthermore, we conclude that $P[fu]^T.full$ holds and that the value in $P[fu].result$ does not change from cycle T to cycle $T+1$. This allows concluding the claim from invariant 6.3 (producer data consistency) for cycle T.

QED

Lemma 6.53 ▶ If the tag on the CDB matches the tag of an instruction I_i and the tag of that instruction is unique, then the instruction on the CDB is instruction I_i.

$$CDB^T.valid \wedge CDB^T.tag = I\_tag(i) \wedge \tau(i,T) \implies sICDB(T) = i$$

This is easily shown using lemma 6.30 (uniqueness of CDB tag) and 6.22.

The following two lemmas are used to argue the data consistency of the reservation stations (invariant 6.2). Since this is where all forwarding is done, this is the most complicated part of the proof. We therefore split the proof of invariant 6.2 into two lemmas.

The first lemma shows the claim for the case the operand reading is done in the issue stage. The second lemma shows the claim for the case the operand reading is done in the reservation station. The same case split is also done in [MPK00].

Lemma 6.54 ▶ Let invariant 6.2 (reservation station data consistency) and invariant 6.1 (register file data consistency) and invariant 6.4 (CDB data consistency) and invariant 6.5 (ROB data consistency) hold during cycle T.

If an instruction is issued into reservation station *rs*, invariant 6.2 for reservation station *rs* holds during cycle $T+1$.

PROOF We show this claim by a case split on the location the operand x is read from. Let I_i be the instruction in the issue stage and let $r = S(i,x)$ be a shorthand for the number of the register we read.

- If $r = 0$ holds, we read zero and the claim holds by definition of $source(i,0)$.
- **Reading from the register file:** This is done only iff $R[r]^T.valid$ holds. This allows applying invariant 6.1. This concludes the claim.
- **Reading from the CDB:** This is done only iff $R[r]^T.valid$ does not hold. This allows applying lemma 6.40, which states that the tag in the producer table is the tag of the last instruction writing $R[r]$. According to lemma 6.43, that tag is unique. This allows applying lemma 6.53, which states that the last instruction writing $R[r]$ is on the CDB. According to lemma 6.4, the result on the ROB is the result of that instruction.

  Thus, it is left to show:

  $$result(last(i,r))[e(r)] \stackrel{!}{=} source(i)(x)$$

  We assert this using lemma 6.38.
- **Reading from the ROB:** We repeat the arguments from the case above in order to show that the tag in the producer table is the tag of the last instruction writing $R[r]$. Let tag denote the tag. This tag is unique, and we therefore know that the instruction in ROB entry tag is the last instruction writing $R[r]$ (lemma 6.21). According to invariant 6.5, the result in the ROB is the result of this instruction. As before, we conclude the claim using lemma 6.38. QED

◀ Lemma 6.55 Let invariant 6.2 (reservation station data consistency) and invariant 6.4 (CDB data consistency) hold during cycle T.

If no instruction is issued into reservation station rs, invariant 6.2 for reservation station rs holds during cycle $T+1$.

PROOF Let x be a source operand number. If the valid bit of operand x holds during cycle T, one just applies invariant 6.2 for cycle T.

If not so, we snoop an operand from the CDB or we have nothing to show. The argue the correctness of CDB snooping as follows: Let i be the

number of the instruction in reservation station *rs* during cycle $T+1$. The claim of invariant 6.2 is:

$$RS[rs]^{T+1}.op[x].data \quad \stackrel{!}{=} \quad source(i)(x)$$

By expanding the definition of $RS[rs]^{T+1}.op[x].data$ on the left hand side, this is transformed into:

$$CDB^{T}.result[e(S(i,x))] \quad \stackrel{!}{=} \quad source(i)(x)$$

Invariant 6.4 states:

$$CDB^{T}.result \quad = \quad result(sICDB(T))$$

Thus, the claim is transformed into:

$$result(sICDB(T))[e(S(i,x))] \quad \stackrel{!}{=} \quad source(i)(x)$$

Thus, it is left to show that the result of the instruction on the CDB is the source operand of the instruction in the reservation station. This is argued as follows: According to lemma 6.38 with instructions I_i and $I_{sICDB(T)}$, the claim above holds if we show the premises of the lemma. These premises are:

$$S(i,x) \neq 0 \wedge \mathcal{L}(i,S(i,x)) \wedge last(i,S(i,x)) = sICDB(T))$$

Thus, we have to show that the source register is not register 0 and that there is an instruction before I_i that writes the register. One easily argues this using invariant 6.42.

Furthermore, one has to show that the last instruction before I_i writing the register is the instruction on the CDB. We argue this using the fact that the tag on the CDB matches the tag stored in the reservation station for the operand. According to invariant 6.42, that tag is the tag of the last instruction writing the register.

Lemma 6.44 states that the tags in the reservation stations are unique. This allows applying lemma 6.53, which concludes the claim.

QED

The following lemma combines the claims of lemma 6.54 and lemma 6.55.

◀ Lemma 6.56

Let invariant 6.2 (reservation station data consistency) and invariant 6.1 (register file data consistency) and invariant 6.4 (CDB data consistency) and invariant 6.5 (ROB data consistency) hold during cycle T. This implies that invariant 6.2 for reservation station *rs* holds during cycle $T+1$.

This claim is shown using lemma 6.54 and lemma 6.55.

◀ Theorem 6.57

The invariants 6.1 to 6.5 hold.

PROOF

We show this claim by induction on T. We omit the simple arguments for cycle $T=0$.

The claim for $T+1$ is shown by applying lemma 6.50, 6.51, 6.52, and 6.56 for cycle T and lemma 6.49 for cycle $T+1$.

QED

◀ Theorem 6.58

A machine implementing the Tomasulo protocols above, satisfies the following data consistency criterion:

$$R[r]^{T}_{aI}.data \quad = \quad R[r]^{sIwriteback(T)}_{aS}$$

Since all speculation registers are output of the writeback stage, this criterion exactly matches the data consistency criterion as proposed for the in-order pipelined machine.

PROOF

Given the data consistency invariants above, one easily shows this claim by induction on T. For $T=0$, we have $sIwriteback(T)=0$ and we therefore have the claim that the registers are in the initial configuration. We assume this.

For $T+1$, we show the claim as follows: In case $writeback(T)$ does not hold, one easily asserts that

$$sIwriteback(T) \quad = \quad sIwriteback(T+1)$$

holds and that the registers do not change from cycle T to $T+1$. Thus, the claim is concluded using the induction premise. Let i be a shorthand for $sIwriteback(T)$.

In case $writeback(T)$ holds, we do a case split on $dest(i,r)$. If $dest(i,r)$ does not hold, we easily assert the claim using the induction premise.

If $dest(i,r)$ and $writeback(T)$ hold, we have the following claim:

$$ROB^T[ROBhead(T)].result[e(r)] \quad \stackrel{!}{=} \quad R[r]^{i+1}_{aS}$$

The register on the right hand side expands to the result of instruction I_i:

$$ROB^T[ROBhead(T)].result[e(r)] \quad \stackrel{!}{=} \quad result(i)[e(r)]$$

We assert this using invariant 6.5 for instruction I_i and tag $ROBhead(T)$, which holds according to theorem 6.57.

The claim of invariant 6.5 concludes the claim above. It is left to show the premises of invariant 6.5, which are:

$$sIROB(ROBhead(T),T) = i \wedge ROB[ROBhead(T)]^T.valid$$

We assert the first part of this claim using lemma 6.24. The valid bit of the ROB entry holds since we assume that we only writeback if the valid bit holds.

QED

6.6 Liveness

We propose the following liveness criterion for the Tomasulo machine with reorder buffer: we will show that all instructions will eventually be in the terminated phase.

We use a similar liveness proof strategy as employed in chapter 4. We show our claim by induction on T. Thus, the induction step is: given all instructions up to instruction I_{i-1} terminated, instruction I_i eventually terminates.

Informally, we show this as follows: We will show that instruction I_i must be in a phase. According to lemma 6.27, that phase is unique. We do a case split on the phase of instruction I_i. If instruction I_i is in "in ROB" phase, we easily assert that it eventually terminates. If instruction I_i is in a producer, we assert that it will move into "in ROB" phase. We

then conclude the claim as before. These arguments are continued until all phases are covered.

We will now formalize this proof.

If instruction I_i is in phase p during cycle T, this implies that it is in one of the successor phases of phase p during cycle $T+1$. ◀ Lemma 6.59

$$p(i,T) \implies \bigvee_{p' \in succ(p)} p'(i,T+1)$$

We show this claim exemplary for phase "not issued". Thus, we have to show that instruction I_i is still not issued, in a reservation station, or in the ROB during cycle $T+1$. PROOF

- If $issue(T)$ and $sIissue(T) = i$ does not hold, one easily concludes that instruction I_i stays in "not issued" phase.
- If $issue(T)$ and $sIissue(T) = i$ holds and $issue\_with\_result(i)$ holds, one easily shows that instruction I_i is in the reorder buffer during cycle $T+1$.
- Otherwise, we assume that there is a reservation station rs such that $issue\_rs(T, rs)$ holds. One easily verifies that instruction I_i is in that reservation station during cycle $T+1$. QED

Instruction I_i is in at least one phase during cycle T. ◀ Lemma 6.60

The claim is concluded by induction on T. For cycle $T = 0$, we conclude the claim easily since all instructions are in the "not issued" phase. PROOF

For $T+1$, we conclude as follows: According to the induction premise, instruction I_i is in at least one phase during cycle T. This allows applying lemma 6.59, which states that instruction I_i is in one of the successor phases of that phase. This concludes the claim. QED

The following lemmas form the induction step for the liveness proof.

If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in "in ROB" phase, instruction I_i will eventually terminate. ◀ Lemma 6.61

PROOF Let T be the cycle given by the premise. According to the premise, instruction I_i is in "in ROB" phase during cycle T. This implies that it is not terminated yet. Since we either have $i = 0$ or the previous instruction is terminated, we have

$$i \quad = \quad sIwriteback(T)$$

We show that instruction I_i terminates during cycle T, i.e., it is left to show that $writeback(T)$ holds. As described above, we assume that we always terminate if the ROB is not empty and the ROB entry that *ROBhead* points to is valid. One easily asserts that the ROB is not empty during cycle T using that instruction I_i is in "in ROB" phase during cycle T.

According to the premise, there is a ROB entry *tag* that is valid and such that

$$sIROB(tag, T) \quad = \quad i$$

holds. Using lemma 6.11, we assert that *tag* is the tag of instruction I_i. Using lemma 6.23, we assert that entry *tag* is the entry $ROBhead(T)$ points to. Thus, the ROB entry $ROBhead(T)$ points to is valid and we writeback.

QED

Lemma 6.62 ▶ If producer fu is full during cycle T, then there is a cycle $T' \geq T$ such that the instruction is put on the CDB.

$$\begin{aligned} P[fu]^T.full \quad \Longrightarrow \quad & \exists T' \geq T : completion(T') \wedge \\ & compl\_p(T') = fu \wedge \\ & sIP(fu, T') = sIP(fu, T) \end{aligned}$$

PROOF In order to show this claim, we make the assumption that the CDB requests are served using a fair arbiter. One has to show that instruction I_i stays in the producer fu until the request is served using induction. For this purpose, we have to assume that the function unit does not overwrite an instruction in its producer. This is illustrated in figure 6.10. Formally, the function unit fu provides a result during cycle T iff $FU[fu]^T.valid$ holds.

The producer generates a stall signal if it is full and does not get the CDB. Let $fuins(fu, T).stall$ denote the value of this signal during cycle T.

$$\begin{aligned} fuins(fu, T).stall \quad := \quad & P[fu]^T.full \wedge \\ & \overline{(completion(T) \wedge compl\_p(T) = fu)} \end{aligned}$$

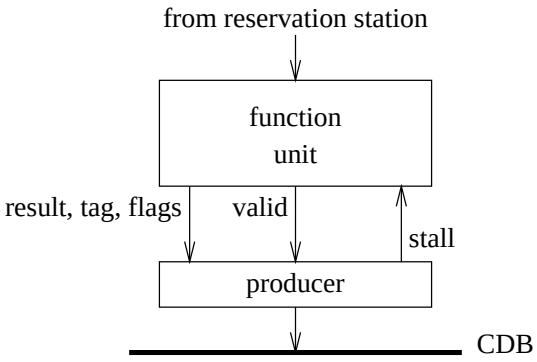

Figure 6.10 Interface between function unit and producer

We assume that the function unit does not provide a result if it gets a stall signal.

$$fuins(fu,T).stall \implies \overline{FU[fu]^T.valid}$$

Since the CDB is assigned using a fair arbiter, there is a cycle T' such that the request is acknownledged. Using the assumption on the function unit above, one easily shows by induction that the instruction stays in the producer until this happens and is not overwritten. QED

◀ Lemma 6.63

If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in "in producer" phase, instruction I_i will eventually terminate.

PROOF

Let T be the cycle from the premise of the lemma. Thus, instruction I_i is in a producer during cycle T. Let this be producer fu. We will show that this instruction eventually moves into the reorder buffer. Although we assume that all instructions prior to instruction I_i already terminated, this is not obvious. In particular, there might be instructions *later* than instruction I_i that block the CDB.

According to lemma 6.62, there is a cycle $T' \geq T$ such that the request is served and the instruction is still in the producer. Formally, we have:

$$completion(T') \wedge compl\_p(T') = fu \wedge sIP(fu,T') = sIP(fu,T)$$

One easily concludes that instruction I_i is in ROB entry $I\_tag(i)$ during cycle $T+1$. This allows applying lemma 6.61, which shows that the instruction eventually terminates. QED

Note that assuming that the CDB is allocated using a fair arbiter is not necessary for liveness, we do it for sake of simplicity only. If the CDB is not allocated using a fair arbiter, we can argue as follows: Informally, assume instruction I_i is blocked in a producer by instructions later than I_i. Since we terminate in-order, there is an upper bound for the number of these instructions, which is the number of ROB entries. Thus, instruction I_i will eventually get the CDB.

Lemma 6.64 ▶ If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in "in FU" phase, instruction I_i will eventually terminate.

PROOF Let T be the cycle from the premise of the lemma. Thus, instruction I_i is in a function unit during cycle T. Let this be function unit fu. We will show that this instruction eventually moves into the producer P. Although we assume that all instructions prior to instruction I_i already terminated, this is not obvious. In particular, there might be instructions *later* than instruction I_i that block the function unit or the producer.

In order to show this claim, we have to make the following assumption on the functional units: Given that the signal $fuins(fu,T).stall$ is finite true and that instruction I_i entered the function, there is a later cycle such that the instruction leaves the unit.

$$\begin{aligned} & \forall T' \exists T'' \geq T' : \overline{fuins(fu,T'').stall} \wedge in(i,T,fu) \\ \Longrightarrow \quad & \exists T''' \geq Tout(i,T''',fu) \end{aligned}$$

One easily asserts that the signal $fuins(fu,T).stall$ is finite true using the fact that the CDB is allocated using a fair arbiter. Thus, we have a cycle T''' such that the instruction leaves the function unit. One easily asserts that this instruction moves into the producer during that cycle. We then apply lemma 6.63 in order to conclude the claim. QED

In analogy to lemma 6.62, we show:

Lemma 6.65 ▶ If a reservation station is full during cycle T, there is a cycle $T' \leq T$ such that this reservation station is dispatched during cycle T'. Furthermore, the instruction in the RS during cycle T' is the same as during cycle T.

$$\begin{aligned} RS[rs]^T.full \quad \Longrightarrow \quad & \exists T' \geq T : dispatch\_rs(T',rs) \wedge \\ & sIRS(rs,T') = sIRS(rs,T) \end{aligned}$$

PROOF As described above, dispatching is done using a fair arbiter. The arbiter selects among the reservation stations that are full and valid. The first thing to assert is that the reservation station is valid. Assume it is not. In this case, one can apply lemma 6.45, which states that there is an instruction I_j with $j = last(i,r)$ that is in a reservation station, in a function unit, or in a producer. This is a contradition to the premise that all instructions I_j with $j < i$ are already terminated.

The function unit provides a stall singal. We denote this stall signal by $FU[fu]^T.stall$. Dispatching is only done if the function unit is not stalled. We assert this using the following assumption on function units: If the stall singal that is input of the function unit is finite true, then the stall signal that is output of the function unit is finite true.

$$\begin{aligned} & \left(\forall T' \exists T'' \geq T' : \overline{fuins(fu,T'').stall}\right) \\ \Longrightarrow \quad & \left(\forall T' \exists T'' \geq T' : \overline{fuins(fu,T'').stall}\right) \end{aligned}$$

One shows that the stall singal that is input of the function unit is finite true using that the CDB is assigned using a fair arbiter, as above. This concludes the claim. QED

◀ Lemma 6.66 If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in "in RS" phase, instruction I_i will eventually terminate.

PROOF Let T be the cycle from the premise of the lemma. We conclude this claim easily using lemma 6.65. According to this lemma, there is a cycle $T' \geq T$ such that the instruction is dispatched. There are two cases:

- The funcition unit returns the result of instruction I_i in the same cycle. In this case, one shows that the instruction moves into the "in producer" phase and uses lemma 6.63 in order to conclude the claim.
- The funcition unit does not return the result of instruction I_i in the same cycle. In this case, one shows that the instruction is in "in FU" phase during cycle $T+1$ and uses lemma 6.64 in order to conclude the claim.

QED

◀ Lemma 6.67 If there is a cycle such that instruction I_{i-1} either not exists or terminated and instruction I_i is in "not issued" phase, instruction I_i will eventually terminate.

PROOF We will show that the instruction eventually either moves into the ROB or into a reservation station, depending on $issue\_with\_result(i)$. This happens if the instruction is issued. We then conclude the claim using lemma 6.61 or 6.66, respectively.

Thus, it is left to show that the instruction is eventually issued. The issue stage belongs to the in-order part of the machine. As done in the previous chapters, one easily concludes that this happens if the stall signal of the stage is finite true. The issue stage is stalled if one of the following conditions hold [Krö99]:

- The ROB is full. One argues that this cannot be the case since all instructions I_j prior to I_i terminated. Thus, we have

  $$sIissue(T) = sIwriteback(T),$$

  which implies that the ROB is empty (lemma 6.4).

- There is no reservation station available. One easily concludes that all reservation stations are empty because all instructions are either in "not issued" or "terminated" phase during cycle T. Thus, they cannot be in "in RS" phase according to lemma 6.27.

- In case of the DLX, there are some instructions that require stalling issue because they depend on registers that the Tomasulo scheduler cannot forward. In case of a conditional branches or jump register instruction, one has to wait until the source register is valid. Assume it is not. In this case, we can apply lemma 6.43, which states that there is an instruction I_j with $j = last(i,r)$ that is already issued but not yet terminated. This is a contradiction.

- In case the instruction is a *movs2i* and the source register is $IEEEf$, we have to stall issue until the ROB is empty. This arises from the fact that the Tomasulo scheduling algorithm is not able to forward this register. As above, one easily concludes that the ROB is empty.

- The desings we verify are based on the designs presented in [Krö99]. The machine stalls issue until the ROB is empty in case the instruction is an rfe instruction. This arises from the hardware cost constraints. We do not have enough read ports for the SPR producer table to forward ESR, EPC, and $EDPC$. As above, one easily concludes that the ROB is empty.

QED Thus, the instruction is issued eventually, which concludes the claim.

Note that in contrast to the machine given in [Krö99], we do not have to stall issue because of busy instruction memory. This arises from the fact that our stall engine allows stalling stages indepandantly.

The following lemma forms both the induction step and induction base for the main liveness claim.

If there is a cycle such that instruction I_{i-1} either not exists or terminated, instruction I_i will eventually terminate. ◀ Lemma 6.68

Let T be the cycle from the premise. According to lemma 6.60, instruction I_i is in a phase. If this is "not issued", we conclude the claim using lemma 6.67. If it is "in RS", we conclude the claim using lemma 6.66. If it is "in FU", we conclude the claim using lemma 6.64. If it is "in producer", we conclude the claim using lemma 6.63. If it is "in ROB", , we conclude the claim using lemma 6.61. If it is "terminated", the claim obviously holds. PROOF QED

Instruction I_i eventually terminates. ◀ Lemma 6.69

We show this claim by induction on i. For $i = 0$, we apply lemma 6.68. This is also done for the induction step. PROOF

6.7 Verifying the DLX Implementation

In this section, we show that the implementation machine I with configurations $c_I^0, \ldots$ complies with the specification.

6.7.1 Implementation Differences

We do not describe the implementation of the DLX with Tomasulo scheduler and reorder buffer, since this design is already presented in [Krö99] in detail including cost and cycle time analysis.

In this section, we describe the differences between the implementation given in [Krö99] and the implementation used for this thesis. Figure 6.11 shows an overview of the hardware.

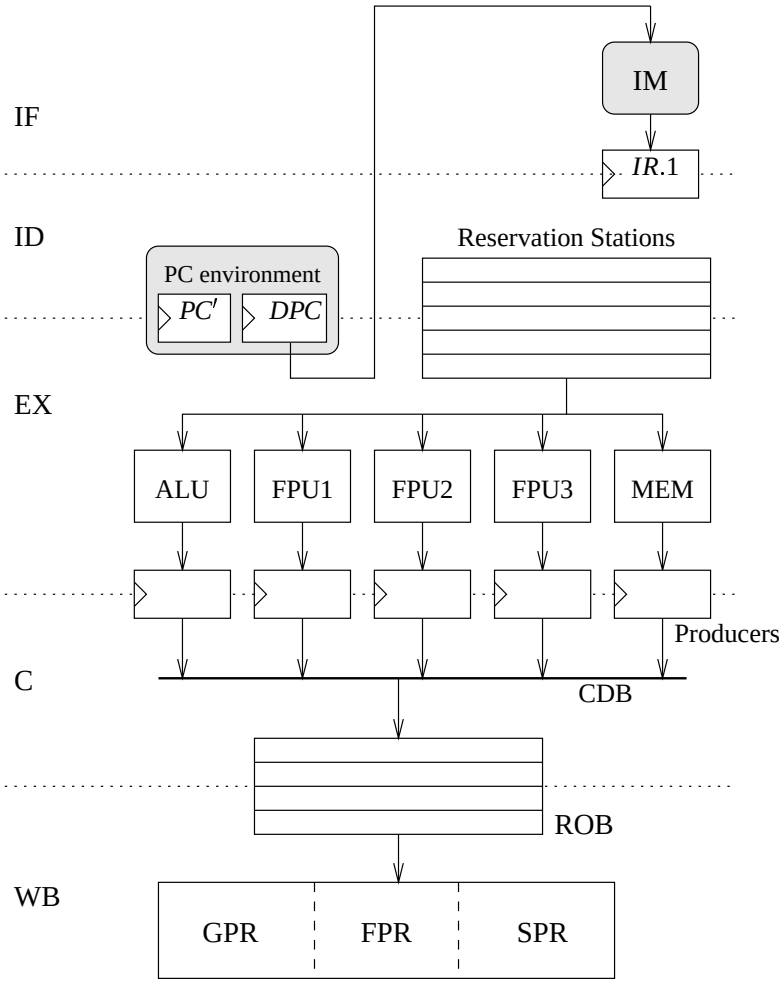

Figure 6.11 Overview of the Tomasulo Hardware

Instruction Fetch In [Krö99], the PC environment from [Lei99] is used. In order to prevent the destruction of the PC registers, stage 0 and 1 are always clocked simultaneously. We remove this limitation by using the PC environment and the stall engine described in chapter 5 (in-order machine with Delayed PC and speculation) instead.

Issue As described above, we no longer need an issue stall because of instruction memory stalls. This is a feature of the new stall engine.

Dispatch In contrast to [Krö99], the instructions do not move from one RS into another. This implementation in [Krö99] is motivated by the liveness proof, which uses the fact that one selects the oldest instruction for dispatch. We use a fair arbiter instead.

Function Units In contrast to [Krö99], we do not implement out-of-order dispatch for the memory unit. This simplifies implementing paging. As an example, consider two store instructions. The first one modifies the page table and the second one modifies a memory cell in a page that is affected. Passing the instructions in program order to the memory function unit significantly simplifies the task of building such a functional unit.

CDB In [Krö99], we allocated the CDB round-robin. We use a fair arbiter instead (this is weaker than round-robin).

6.7.2 Verifying the Instruction Fetch

In the proofs above, we assumed that the instruction fetch is correctly done. The instruction fetch mechanism in the stages 0 and 1 operates like the in-order pipelined machine as described in section 5. The verification of the forwarding of *DPC* for the instruction fetch uses the very same arguments as before.

One combines the two machines as follows: we define that we issue an instruction if the output registers of the decode/issue stages are clocked. This happens iff ue_1^T is active, as described in the previous chapters.

$$issue(T) \quad := \quad ue_1^T$$

For the correctness proof, we argue on the schedules of both parts of the machine. We argue that the schedule of the issue stage of the Tomasulo part matches the schedule of the issue stage of the in-order pipeline.

$$issue(T) \quad \stackrel{!}{=} \quad sI(1,T)$$

We show this claim by inducition on T. For $T = 0$, we have $issue(T) = 0$ and $sI(1,T) = 0$.

For $T+1$, we show the claim by a case-split on ue_1^T. If ue_1^T does not hold, the value of both scheduling functions does not change from cycle T to $T+1$ by definition. Thus, the claim is concluded using the induction premise.

If ue_1^T holds, we have

$$sI(1,T+1) \quad = \quad sI(1,T)+1$$

according to invariant 5.1.

By definition, $issue(T)$ holds if ue_1^T holds. Thus, we have

$$issue(T+1) \quad = \quad issue(T)+1$$

by definition of $issue(T+1)$. This allows concluding the claim using the induction premise.

6.7.3 Verifying IEEEf

The $IEEEf$ (IEEE flags) register is a special case for the correctness proof of the machine, since the IEEE standard [IEE85] requires that the bits in this register are sticky. Thus, if a floating point instruction generates a masked IEEE exception, the bit of this exception is set in the $IEEEf$ register. The bits that were set previously are maintained. However, in case of a *movi2s* instruction with destination $IEEEf$, all bits are overwritten.

One argues the data consistency of the register by induction. As induction claim we show the data consistency of the complete machine. For $T = 0$, we show the correctness of the initialization. For $T+1$, we have the data consistency upto cycle T as premise. The first thing is to argue the correctness of the interrupt mask in SR_I^T. This holds according to the induction premise. Let i be a shorthand for $sIwriteback(T)$. We distinguish three cases:

- If we do not writeback an instruction, we have

$$sIwriteback(T) \quad = \quad sIwriteback(T+1).$$

The registers also do not change. Thus, the claim holds.

- If we writeback an instruction that is *movi2s* with destination register $IEEE\,f$, the correctness is shown as above.

- If we writeback an instruction which sets IEEE flags, we have:

$$sIwriteback(T+1) \quad = \quad i+1$$

We assert the correctness of the flags as above using invariant 6.5. Let $ieeeflags(i)$ denote the IEEE flags generated by instruction I_i:

$$ROB[ROBhead]^T.result[2] \quad = \quad ieeeflags(i)$$

We assert the correctness of the old value in the IEEE flags register using the induction premise:

$$IEEE\,f_I^T \quad = \quad IEEE\,f_S^i$$

The new value written into the IEEE flags register is the old value *OR* the masked new one.

$$IEEE\,f_I^{T+1} \quad = \quad IEEE\,f_I^T \vee (ROB[ROBhead]^T.result[2] \wedge SR_I^T)$$

The claim is that this the correct value:

$$IEEE\,f_I^{T+1} \quad \stackrel{!}{=} \quad IEEE\,f_I^{i+1}$$

One expands the transition function of the specification machine on the right hand side:

$$IEEE\,f_I^{T+1} \quad \stackrel{!}{=} \quad IEEE\,f_I^i \vee (ieeeflags(i+1) \wedge CA_S^i)$$

This is easily concluded using the the equations above.

One cannot forward the $IEEE\,f$ register using the mechanisms described above. We therefore stall the issue stage if we read this register until the ROB is empty. As soon as the ROB is empty, we have

$$sIissue(T) \quad = \quad sIwriteback(T).$$

In this case, one easily concludes the correctness of the value in the register using the data consistency criterion above.

6.7.4 Verifying Interrupts

In this section, we describe how to verify a machine that generates interrupts. The proof method is taken from [MP00]. We show the data consistency by induction on T. For $T = 0$, we have the correctness of the initialization of the machine. Note that we do not process an interrupt during cycle T. We realize the *reset* interrupt by adjusing the initial configuration accordingly, as done in chapter 5.

Let $lastint(T)$ denote the number of the last cycle before cycle T in which we processed an interrupt **plus one** (i.e., the maximum value of $lastint(T)$ is T). In case no such cycle exists, we define $lastint(T)$ to be zero.

In order to show the claim for $T+1$, we distinguish two cases:

- If we have an interrupt during cycle T, we argue as follows: according to the induction premise, the data consistency for cycle T holds. The modifications made by an interrupt on the configuration are easy to verify using this fact.

- If we do not have an interrupt during cycle T, we argue as follows: We claim that the machine works as the abstract implementation machine without interrupts above from cycle $lastint(T)$ to cycle $T+1$. We initialize the abstract machine without interrupts using the configuration $c_I^{lastint(T)}$:

$$c_{aI}^0 \quad := \quad c_I^{lastint(T)}$$

  We then show that the transitions made by both machines are equal from cycle $lastint(T)$ to cycle $T+1$ using induction on the cycle number. For this one uses the fact that there are no interrupts from cycle $lastint(T)$ to cycle $T+1$ by definition of $lastint(T)$.

Liveness Note that the liveness of the machine with interrupts does not require extra arguments as required in chapter 5. This arises from the fact that the instruction that generates the interrupt retires as usual and is not executed a second time. This is in contrast to the implementation of interrupts given in chapter 5.

6.8 Literature

In this chapter, we formally verify the Tomasulo scheduling algorithm with reorder buffer as presented in [MPK00]. In contrast to [MPK00], we verify the correctness using PVS and argue the uniqueness of the tags.

The parts of the hardware are based on machines described in [Lei99]. The correctness of the designs presented in [Lei99] is not verified by means of machine.

Hosabettu et.al. verify implementations using a Tomasulo scheduler both with and without reorder buffer [HGS99, HGS00, Hos00] using the completion functions approach. The verification is done using PVS at a very high level of abstraction. Gate-level designs are not verified. The functional units are very simple and do not contain cycles. Despite that, the size of the PVS proofs in [Hos00] is four times the size of the proofs for this chapter of this thesis. However, [Hos00] makes extensive use of proof strategies, which enlarges the PVS proofs significantly.

In [BBCZ98], Clarke et.al. verify out-of-order processors by combining symbolic model-checking with uninterpreded functions. In [BCRZ99], Clarke et.al. verify safety properties of a PowerPC, which implements out-of-order execution and precise interrupts.

Sawada and Hunt [SH99] verify the FM9801, which also features a reorder buffer, using the theorem proving system ACL2. The number of lemmas is enormous (nearly 4000).

Henzinger et al. [HQR98] verify a simple out-of-order processor using a model checker. McMillan [McM98] partly automates the proof by refinement of Tomasulo's algorithm presented in [DP97] with the help of compositional model checking. This technique is improved in [McM99b] by theorem proving methods to support an arbitrary register size and number of function units. In [McM99a], McMillan verifies the liveness of a machine with Tomasulo scheduler using SMV.

Arvind and Shen [AS99] describe how to apply term rewriting systems in order to model microprocessors. The authors give a simple out-of-order RISC machine with reorder buffer as an example. The authors suggest the use of tools such as PVS for verifying large, realistic machines.