Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: Group 21

Matteo Fragassi (s317636), Davide Giuffrida (s310265)

July 10, 2024

Contents

List of Figures

CHAPTER 1

Introduction

In this report we are going to present the implementation of a DLX architecture based on a Tomasulo out of order engine, which has been optimised to support clock frequencies around 1 GHz. The design includes all the hardware structures that are needed to execute instructions in an out of order fashion, such as reorder buffers and renaming tables (RATs), and some hardware to perform more complex operations, such as divisions and multiplication. In particular, the OOE supports speculation, which allows the execution to go on in case of unresolved conditional branches.

In the following pages, each module will be presented in detail, together with the challenges that have been overcome in order to implement the fully-functional Tomasulo algorithm. After this overview, the simulation, the synthesis and the place and route steps will be explained, also showing the scripts and the programs that have been used to perform these tasks.

CHAPTER 2

DLX architecture: basic and advanced

2.1 Introduction

Before starting to describe in details all the features implemented in our processor, it is better to explain in broad terms what are the basic characteristics of the DLX architecture. After doing so, we will briefly highlight all the additions, detailing how they are integrated in the basic model.

2.2 DLX basic functionalities

A DLX is a pipelined processor architecture consisting of 5 stages: fetch, decode, execute, memory and write-back. It supports a RISC ISA consisting of some basic instructions to implement arithmetic and data manipulation operations. Each stage of the pipeline implements different functionalities:

- FETCH(F): the instruction to be executed is retrieved from instruction memory.
- DECODE(DEC): the instruction is decoded, producing all the control signals that will be needed to drive components in the datapath during the following phases of the instruction execution.
- EXECUTE(EXE): the arithmetic computation that is needed by the instruction is performed. For example, the result of an arithmetic instruction and the destination address of a branch or a store instruction are computed in this stage.
- MEMORY(MEM): where the memory access is performed by load and store instructions.
- WRITE-BACK(WB): to write the instruction result back in the register file.

2.3 Added functionalities

As part of our personal implementation of the DLX architecture we decided to add the following features, which are presented in order of decreasing importance in the overall design:

- An out-of-order engine based on Tomasulo's algorithm

  This engine is able to schedule instructions in an out-of-order way, considering data dependencies, and the corresponding hazards, together with the usage of hardware resources. This engine involves components like:

  – Register Alias Table (RAT): to implement register renaming.

  - ReOrder Buffer (ROB): to store instructions while they are being executed or they reside in the pipeline.
  - Reservation Stations (RS): to store instructions before they are executed, while they are waiting for their operands to become available.
  - Content Addressable Memory (CAM): it acts as a buffer to keep the values to be stored in memory until the corresponding store instructions are committed.

- Hazard detection, to ensure that instructions are executed keeping into account the data dependencies. The hazard detection logic is spread across the units that are part of the OOE.

- An execute stage that is able to host multiple instructions at the same time, both in the same execution unit, if it is pipelined, and in different ones (e.g. we may have a division executing in parallel with an addition). The instructions may terminate in an out-of-order fashion, but they will then be committed in order by the ROB.

- Some optimizations, that mainly rely on adding some degree of dynamic pipelining, in order to reduce the length of the critical path. The adopted strategy involves adding some pipeline stages that are occupied in specific occasions. For example, there is one that is used only by load instructions and another one that is used only by instructions leaving the reservation stations.

- A BPU consisting in a branch target buffer (BTB). This unit identifies branches whose previous instance resulted in a "taken" jump and predicts "taken" if they are executed again, while it predicts "untaken" otherwise (i.e. the instruction is an untaken branch or it is not a branch instruction). To perform a prediction, the BPU considers only the previous instance of the branch.

- A divider, implemented using non restoring algorithm. It is a non-pipelined unit which takes approximately 34 clock cycles to complete execution. The actual time varies depending on some factors, as we will show later.

- A pipelined version of the Booth multiplier, which consists of 17 stages (2 of them have been added to optimize over timing). It allows mul instructions to move in the pipeline independently from one another.

- Some mechanisms to dynamically define the priorities for each unit that is part of the execute stage. This may come in handy when multiple instructions from different unit terminate in the same clock cycle, since having dynamic priorities means that, in different cycles, different units will be prioritized.

- The shifter and the logicals blocks have been implemented as the ones present in the T2 architecture. In particular, the logicals have been designed in a structural way, while the shifter has been described with a behavioral approach.

- Some minor instructions that are not a part of the basic instruction set, such as "sub" (subtraction).

- Some modules from the laboratory activities have been used, such as the P4 adder and the Booth multiplier. The latter has been modified to support instructions pipelining.

CHAPTER 3

Out-of-order Engine

3.1 Introduction

The out-of-order engine is the hearth of our DLX architecture, since it handles the whole flow of execution and it is composed of multiple components that are located in different stages of the pipeline. The main purpose of this feature is to allow execution of instructions in an out-of-order fashion, enabling the concurrent execution of multiple instructions in the execution (EXE) stage and avoiding to stall the whole flow in case of hazards. The engine itself has been implemented in three consecutive steps:

1. We initially implemented a version of the architecture consisting only of ROB, RAT and memory buffer, with no reservation stations. This version was capable of executing multiple instructions of different kinds in parallel in the EXE unit. However, a simple data hazard would have caused the stall of both the instruction fetch and decode stages since there was no buffer where to store the instruction during the hazard resolution.

2. The second version of the engine is completely functional, since it also implements the reservation stations. This version allows us to temporarily store an hazard-throwing instruction in a buffer until the hazard condition is solved, thus leaving the decode stage free to host and decode other instructions during the hazard resolution.

3. The third version of the engine preserves all the functionalities introduced in the second one, adding some optimizations to allow for an increase in the clock frequency. The main modifications consist in:

   - Addition of some registers between the units
   - Implementing a pipeline inside the commit and the decode stage

   Despite being responsible for an increase in latency (in terms of number of clock cycles), the last point brings a considerable decrease in the length of critical paths.

   These kinds of optimizations have been implemented throughout the whole core, including execution and memory units.

3.2 Components

The main components of the engine are:

- Register Alias Table (RAT)

  The table that is used to implement register renaming. It stores, for each register, the location (i.e. the ROB index) of the latest instruction which marked that register as destination or an invalid entry in case there are no uncommitted instructions which were planning to overwrite that register. An entry corresponds to the following fields:

  - A valid bit: set to 1 while an instruction which plans to overwrite that register currently resides in the pipeline (the results produced have not been committed yet). When the last instruction which produces a result to be written in that register is committed, the valid bit is cleared.
  - A ROB entry index: set to the ROB index of the entry that holds the last uncommitted instruction which plans to modify the register. This field is checked every time an instruction which modifies the register is committed: in case the ROB index of that instruction matches the index stored here, the RAT entry is invalidated.

- ReOrder Buffer (ROB)

  It contains all the instructions which are currently executing or have been completed but their results have not been copied to the register file or to the memory yet (i.e. they have not been committed). The instructions residing here can be executed and terminated in an out-of-order fashion, but, when completed, they will be committed to the register file/memory in order. The commit is performed when the head instruction (i.e. the oldest instruction placed in the ROB) is flagged as terminated: when the head is committed, the head pointer is moved to the next instruction, which becomes the new head, and the previous head entry is marked as free. Each ROB entry can be in three different states:

  - Invalid ("00"): there is no instruction allocated in the ROB entry
  - Allocated ("01"): the instruction has been allocated but it has not terminated yet
  - Terminated ("10"): the instruction is waiting to be committed

  Inside each ROB entry there is the need to store:

  - The control word for the instruction stored there (bits 0 to 31 of ROB_mem);
  - The result produced by the instruction (bits 32 to 63 of ROB_mem); valid only in the "terminated" state
  - The instruction itself, since it has to be forwarded to the following stages of the pipeline (bits 64 to 95 of ROB_mem)
  - The branch prediction and branch result (bits 96 and 97 of ROB_mem); this field is meaningful only for branch instructions.
  - The value of PC+4 (bits 98 to 129 of ROB_mem)
  - The value of operand regB for the instruction, which has to be kept in ROB for store instructions (as we will show when discussing the CAM and memory buffer functioning).

  Our reorder buffer has multiple input ports to reduce the space occupation in later units: for example it would be unfeasible to store in every pipeline stage of the Booth multiplier all the information related to the instruction currently in that stage, so we simply store a ROB index and the control word in the pipeline and we fetch the remaining information from the ROB when the instruction leaves the EXE stage. Obviously this requires an additional ROB port.

- Reservation Stations (RS)

  The buffers where hazard-throwing instructions are temporarily stored until the hazard is solved. In our design we implemented multiple reservation stations, one per each unit in the EXE: the only exception for this rule is given by the adder, which is associated both to the reservation station containing arithmetic instructions involving an addition and the one where load/stores are kept (the reason why a separate RS for load/store instructions is needed will be discussed later). The main purpose of these stations is to provide temporary storage for instructions that can cause hazards, thus allowing us to free the decode stage and execute additional instructions which are located after the one causing the problem. Clearly, this is possible provided that these instructions do not show data dependencies with the one that is subject to the hazard. Let us consider the following code as an example:

```
1 mul r1,r2,r3
2 mul r4,r1,r3
3 div r5,r2,r3
```

  Without reservation stations the decode unit would stall when decoding the mul at line 2, because there would be no place where to store the hazarding mul while the mul on line 1 is executing. This means that the first mul and the div on line 3 would not be able to execute in parallel since the second mul is stalling in decode, even if the div didn't show data dependencies with other instructions. This behavior is responsible for a huge loss in performances because the execution is serialized. Each reservation station entry corresponds to an instruction that is waiting to be executed, so the corresponding entry is deleted when the instruction is sent to the EXE stage. Each entry contains the following fields:

  - The ROB index corresponding to the entry where the instruction is allocated in the ROB
  - The state of the operands. In fact, for each operand there is the need to know if it is valid in the register file (the instruction that produces it has already been committed), valid in the RB (produced by an instruction which has not been committed yet) or invalid (not yet available in the RF nor in the ROB). In the last two cases we need to know the ROB entry of the instruction which will produce that operand.
  - A valid bit for the entry, to determine if there is a valid instruction allocated or not.

- Memory buffer and CAM This module has been implemented to avoid stalling the memory stage when a load instruction arrives. The necessity to stall the stage stems from the fact that a load could perform a read over a memory address which has been overwritten by a previous store operation, but the actual write on memory is performed only when the store is committed. If we don't have a buffer where to keep the load instruction, we have to stall the MEM stage until all the instructions before the load are committed (in this way we are sure that a potential store has been committed too). The approach we decided to take consists in using a CAM to keep all the addresses that were targeted by uncommitted store operations, in parallel with a buffer keeping the ROB entry of the last store instruction which targeted that address. In this way we can avoid stalling the load in MEM, because it is enough to check in CAM if the address that the load wants to read has been accessed by a store that is still uncommitted: in case it was, it is enough to read the written value from the buffer, otherwise we can read directly from memory. This mechanism is powerful because it allows the speculation to go on also for instructions which access the memory, thus improving the performances since you don't have to stop the flow when executing load instructions.

3.3 A limitation of the presented algorithm

Using a CAM to store the address accessed by a store works only when we have a store and a load (on the same address) one after the other and they are executed in order. In case the two instructions are sent out of order, the load will end up reading wrong data from the memory because the store which comes before in the code has not been able to write the new value in CAM (since it has not been executed yet). This may happen because there is no way to recognize if there is a data dependency between the addresses accessed by a load and a store, the reason behind this being the impossibility to know the address when the instruction is sent to the execute stage. The same behavior may arise when we have a load and a store and they are executed out of order: in this case, a CAM entry will be populated with the ROB entry of the store, thus marking the address as modified and providing the load which will be executed afterward with wrong data. This means that the only solution to guarantee correct behavior in any case is the one to execute all the load and store instructions in order, so that the modifications done over the CAM will be read only by the correct load instructions. In order to do that we designed an additional reservation station which is reserved for load and store instructions and is scanned (and populated) in order: the order is guaranteed by keeping a pointer to the head of the buffer and one to the tail, moving them appropriately when instructions are outputted or sent to this station.

3.4 General Algorithm

Let us show the complete execution flow for each instruction, including the steps that involve the out-of-order engine. The flow presented here doesn't include the intermediate pipeline stages that have been added to optimize the clock frequency. Some of these improvements will be showcased later, while commenting all the modules in detail. The main steps are the following ones:

- When the instruction reaches the decode step it is sent to the RAT, which overwrites the entry corresponding to the destination register with the index of the ROB entry where the instruction will be allocated while marking the RAT entry as valid.
- In the decode stage, the instruction is also allocated in the ROB, in the position corresponding to the first free entry (the ROB is scanned as a circular queue and the new instruction is allocated in correspondence of the tail pointer, which is then increased by one).
- An evaluation is done to assess if the parameters of the instruction are ready. In this case, the instruction is not allocated in the reservation stations and it is scheduled to be executed in the next cycle. Otherwise, it must be saved in the reservation station corresponding to the functional unit where the instruction will be executed. Originally, there were bypasses from the WB stage to allow an instruction to proceed if its operands were coming in that cycle from the instruction in WB. However, they have been removed to optimize over the clock frequency since they created a very long path in decode. There is also the corner case where an instruction from the reservation stations becomes ready in the same cycle when an instruction, whose operands are available, is sent to decode. In this situation, the instruction from the reservation station has the precedence, so the instruction in decode has to be stored in the reservation station until at least the next clock cycle, while the one from the RS is promoted in execute.
- Reservation stations are used to store instructions while their operands are not ready. Each entry is updated when an instruction which produces an operand for that entry reaches write-back or when it is committed. When both the operand are flagged as ready the instruction is flagged as available to be sent to EXE. If multiple instructions coming from different reservation stations become ready at the same time, a static priority favours additions and subtractions over

other operations. The policy could be adjusted by simulating the design over a benchmark and checking which static priority assignment is the best one, or we could even resort to a dynamic approach.

- After the operands become ready, the instruction is promoted to the execute stage. The execution will last a different number of clock cycles depending on the functional unit involved.
- When the instruction reaches the MEM stage, we need to check whether the instruction requires to access the memory or not. In the former case, we need to handle it depending on if it is a load or a store while in the latter case we can simply promote it to the WB stage. If the instruction is a store, we need to allocate a new entry in the CAM for the target address and write in the corresponding buffer entry the ROB entry of the store instruction itself. If the instruction is a load, we check if there is a CAM entry reporting the memory address that it is trying to access; if so, we read the corresponding ROB entry from the buffer and we access the ROB to look for the value of regB (the one to be written in memory) corresponding to the store instruction, otherwise we read it directly from memory.
- As a part of the WB stage we need to mark the instruction as completed in the ROB. If the earlier instructions have not been committed yet, then the instruction cannot be committed since we could trigger WAW or WAR hazards. While the instruction resides in the ROB after completion, the value produced can be read from the ROB when scheduling instructions that use that value as an operand.
- When the instruction is committed, the RAT entry of the destination register is checked. If the ROB entry reported there is the one that holds the instruction that we are committing, then we need to invalidate that RAT entry, otherwise we do nothing. The latter case means that there is an instruction, located later in the code, which has been issued and targets the same register. After this operation, if the instruction to commit is a store and the ROB entry in the buffer is the same of the instruction we are committing, we need to clear the CAM entry and the memory buffer containing the ROB entry for the produced address. After doing so, we update all the reservation stations where the ROB entry appeared as an operand, marking the operand as available in the register file. In the end we can invalidate the ROB entry, since we have removed all the references in other modules, and, in parallel, we can write the result in the destination register or in the correct memory address. Some of these operations can be parallelized for efficiency, while some of them have been serialized in different clock cycles to reduce the critical path length.

3.5 Hazard handling

The biggest strength of this algorithm is that it helps us to handle all the possible hazards that could be raised in an out-of-order architecture. Depending on the kind of hazards considered the handling is managed in a different way:

- Structural hazards

  There are various kinds of structural hazards that can arise during the execution, most of them are related to the fact that there is no superscalarity between the DEC and the EXE stage. As an example, if an instruction from the RS becomes available while the one from decode is found out to be ready, only the first one has to be sent to EXE while the second one gets stored in the RS. Another example is the structural hazard that arises when the ROB or the RS are full, thus making it impossible to allocate new instructions. This is solved by stalling the instruction to be allocated in decode, blocking the whole decode stage until at least one entry is freed from the buffer that is full.

- Data hazards

  These hazards are handled differently depending on whether they are RAW, WAR or WAW hazards:

  - Read-After-Write hazards

```
1         mul r1,r2,r3
2         div r4,r1,r3
```

    In this case the mul produces a value that is used as operand by the div. This means that we have to make sure that the mul completes before executing the div, otherwise this last instruction will execute with wrong values (e.g. the ones in the RF before the mul commit, in the wrong ROB entry or garbage values from the ROB entry of the mul). The handling mechanism for this kind of hazard involves the hazard detection circuitry, which checks the availability of operands in the ROB before scheduling the instruction for execution; in particular, for each instruction the circuitry performs reads from the RAT to determine if the operands are available in the RF or produced by instructions in the ROB. In this last case we have to check the ROB too, to assess if the instruction has already terminated or not.

  - Write-After-Read hazards

```
1         mul r1,r2,r3
2         div r3,r2,r4
```

    The second instruction can execute before the first one, but it produces a value to be written in a register that is used as operand by the first instruction. In this case, the correctness of the flow is guaranteed by the in order commit plus the behavior of the RAT and the RS combined. In particular, when we check the availability of the operand we read the RAT, which reports the ROB entry of the last instruction that used that register as destination, then we allocate the instruction in the RS marking the operands with the ROB entries that we read. This makes it impossible for the mul to use the r3 produced by the div, because the div will correspond to a different ROB entry with respect to the one read when the mul has been decoded. The key point is that the RAT read is performed once, when the mul is decoded and the div has not yet been sent to the RAT. This mechanism is known as "register renaming", because the register operands are actually remapped to a ROB entry which acts as a virtual name for that register.

  - Write-After-Write hazards

```
1         mul r4,r2,r3
2         div r4,r1,r3
```

    The hazardous scenario is the one in which the div writes over r4 before the mul. This is trivially addressed by the commit mechanism, since it is performed in order. However, a critical case that we have to address is the following one:

```
1         mul r4,r2,r3
2         div r4,r1,r3
3         add r2,r1,r4
```

    In this scenario, the r4 value is also used by another instruction which follows in the code. In this case the correctness of the operand is again ensured by the RAT, because the RAT entry for register r4 will be overwritten with the ROB index corresponding to the div when this instruction is decoded. This means that, when decoding the add, r4 will be marked as produced by the ROB entry of the div and not by the ROB entry of the mul.

- Control hazards

They consist in speculative execution of some parts of the code like the ones that follow conditional branch instructions. In this case, it is necessary to resume from a correct state if we find out that the prediction we did at the beginning was wrong. This kind of hazard is handled by the commit mechanism, because all the results will actually be echoed to the RF or to the memory only when the speculative instruction is committed. This will necessarily happen after the commit of the branch.

3.6 Interactions between the modules in decode

The schema in figure 3.1 has been drawn to clarify the behavior of the components when they are considered together. Let's explain one by one all the numbered connections between different modules:

1. Connection through which the RAT receives the index of the operands for the instruction residing in the F2D pipeline register. The RAT reads the entries corresponding to these values and outputs them on channel 2. The RAT also overwrites the RAT entry associated to the destination register of the instruction with the index of the ROB entry where the instruction will be allocated, so that the register can be marked as being produced by that instruction. The instruction will be allocated in the ROB at the end of the clock cycle when this read is performed (the connection for the allocation is not shown in the diagram, which only represents the operands flow).

2. Connection which delivers the RAT output data. These are the ROB entries of the last uncommitted instructions that have the operand registers of the instruction currently in decode as destination register or an invalid entry otherwise. In this case the operands values can later be retrieved from the RF. All the data provided by the RAT are sent to the register which divides the first two decode substages (the details will be explained when discussing the modules).

3. Connection which feeds the ROB with the ROB entries that have been read from the RAT. The ROB is in charge of controlling if the corresponding instructions have already terminated their execution; in this case, the operands are ready and the instruction can proceed in EXE (except if there is another instruction leaving the RS), otherwise the instruction must be kept in the RS.

4. Connection which allows the read of the operand to be performed from the RF if the RAT entry (corresponding to the register where the operand is stored) is invalid, which means that the operand is available directly in the RF since there are no scheduled instructions which are planning to overwrite that register.

5. Connection which allows an instruction whose operands are not ready to be stored in the RS while waiting for their availability.

6. Connection which brings to the mux the operands read from the ROB.

7. Connection which brings to the mux the operands read from the RF. The mux will select values on this connection or the previous one depending on whether the operands were produced by an instruction in the ROB or if they were already available in the RF.

8. Output of the mux for the operands of the instruction currently residing in decode.

9. Output of the mux which selects between the instruction currently residing in decode or the one coming from the RS, whose operands have just become available.

10. Connection that delivers an instruction residing in the RS whose operands are ready. This instruction is sent to one of the pipeline registers that divides the decode substages.

11. The operands of the instruction that is leaving the RS can be read from the ROB if they are not available in the RF yet. This works exactly as we explained for instructions coming from connection 3.

12. The operands that are already available in the RF have to be read directly from there. This works exactly as connection 4, but in this case the instruction involved is the one coming from the RS.

13. The operands for the instruction from the RS that have been read from the ROB are sent to a mux, exactly as it happens for the instruction residing in decode.

14. Exactly as the previous connection, but for the operands coming from the RF.

15. Output of the mux which receives the the two connections above as inputs.

16. Connection that represents the ROB index and the data related to the instruction that is currently undergoing write-back. These data are read by the ROB and used to update the corresponding entry.

17. Connection that handles the communication between ROB and RAT during the commit of an instruction. The RAT entry corresponding to the destination register of the committing instruction should be invalidated if the ROB entry stored in that RAT entry holds the instruction that is being committed.

18. Communication channel between ROB and RS during commit. The availability of the operands of the instructions in the RS has to be updated because an instruction has been committed.

19. Connection to handle the data transfer in the register file when performing a commit.

3.7 Interactions between the modules in the memory stage

In this section, all the connections between modules that are part of the memory stage will be discussed, highlighting the roles they play during different phases in the instruction execution. The organization of the memory stage will be shown in details later (in section 5.5, it may be necessary to read this section for a better understanding), but we can hint about how the stage is divided in two substages; the second one is only for load instructions. This allows us to implement a form of dynamic pipelining, where non-load instructions are allowed to pass directly to M2WB, while load instructions have to go through the intermediate pipeline stage. In particular, the first cycle is needed to check the CAM while the second one is necessary to read the operands from the ROB. In case there is already a load instruction in the intermediate register when a new non-load instruction arrives, the non-load should take the long path too, in order to avoid to have two instructions competing to leave the memory stage. In case the non-load instruction were to take the short path too, hazard detection and stalling would be needed, making the flow more difficult to handle and potentially increasing the critical path length. All the connections we will discuss are numbered in figure 3.2:

1. This connection represents the flow of data for an instruction which is not a load.

2. In case there is already a load in the intermediate stage when a non-load arrives, the non-load is sent to the intermediate stage too.

3. The ALU_res is sent to the compare circuitry (CMP) to allow condition resolution for branches.

4. The output of the CMP is sent to the pipeline register.

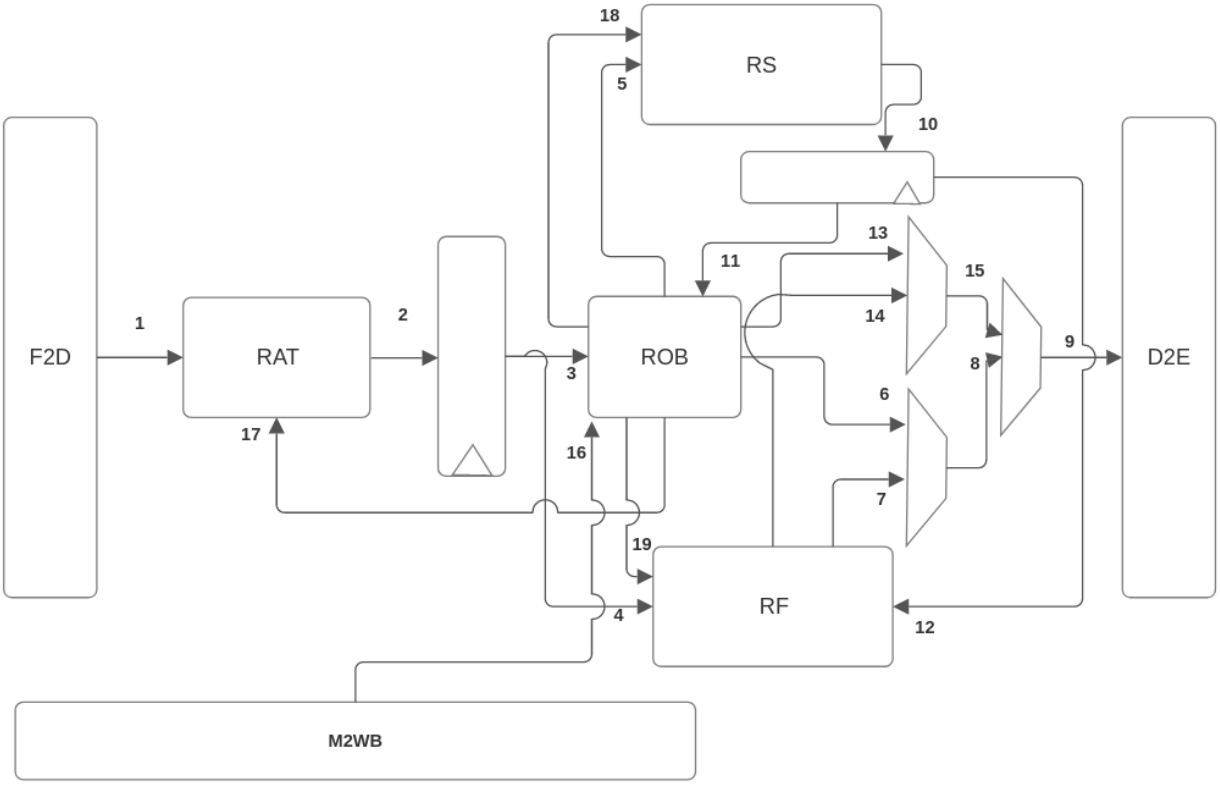

Figure 3.1: A schema of the interactions between the different modules involved in defining OOE functionalities

5. For load instructions, the address to be read from memory is sent to CAM to check if there are uncommitted store instructions which accessed the corresponding location. For store instructions, the address where the write operation has to be performed is allocated in a new CAM entry.
6. The entry read from the CAM is sent to the intermediate pipeline register.
7. The ROB index read from the CAM entry is used to identify the executed, but uncommitted, store instruction which accesses that memory location.
8. The value that the store instruction is supposed to write in memory is read from the ROB.
9. In case there is no CAM entry associated to the address that the load is trying to read, it means the value to be read is already available in memory. This value is sent to a mux together with the one coming from the ROB, and the one to be selected depends on the valid bit set by the CAM.
10. The output of the mux mentioned in the description of the previous interconnection.
11. The path for the non-load instruction which arrived in MEM when there was already a load instruction in the intermediate stage. In the clock cycle after the arrival, the instruction will be put on the output of the intermediate stage.
12. The output of the mux which selects data related to the non-load instruction.

13. When the commit of a store instruction is performed, the CAM entry corresponding to the address where the write operation has been performed is invalidated.

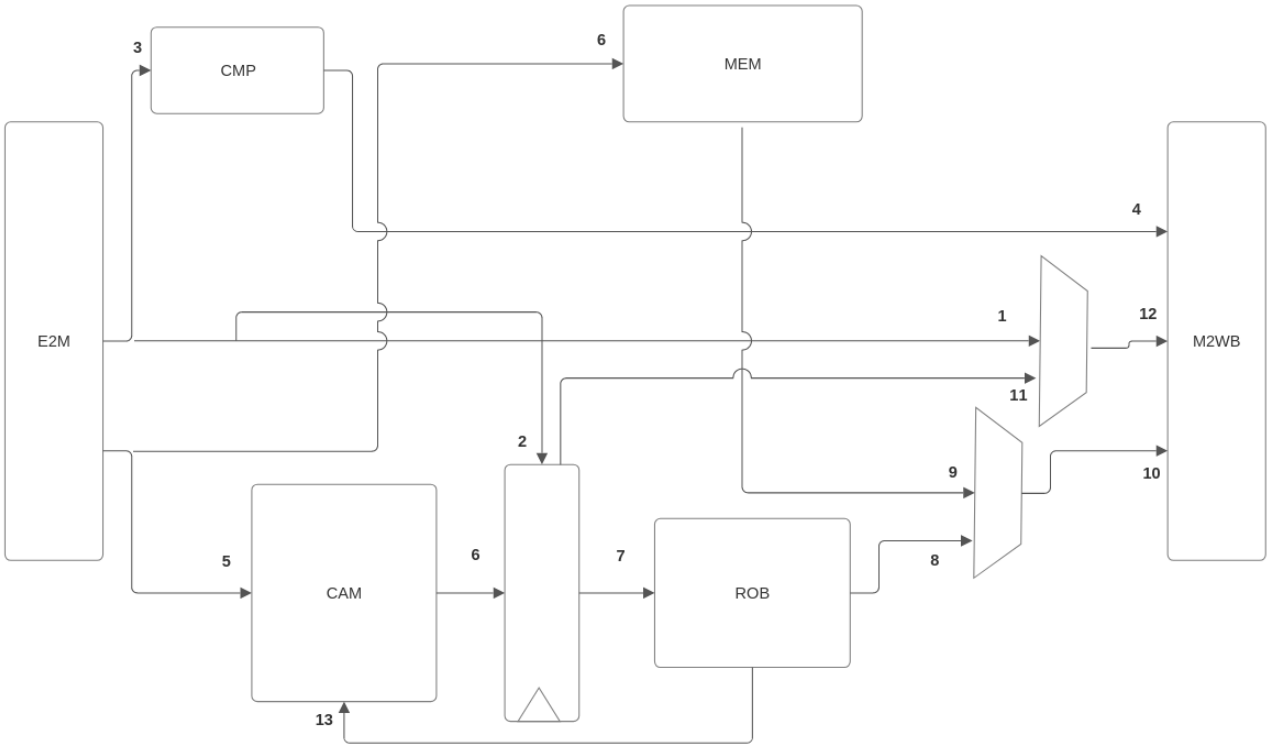

Figure 3.2: A schema of the interactions between the different modules in memory stage

CHAPTER 4

Control Unit

4.1 Introduction

The implemented control unit is a simple hardwired CU, which produces all the control signals inside a case-when statement. In order for the execution to proceed correctly, the CU must check if hazards of any kind are raised during the execution of an instruction and, in case they are, it should prevent that instruction from being promoted to the EXE stage. In our design the CU acts only as a checker, which means that the actual hazard signals are produced by circuitry which is part of the datapath.

4.2 Covered hazards

As anticipated, the CU simply checks if the hazard signals are raised and it stalls the instruction in decode if deemed necessary. The hazards that are explicitly covered are the following ones:

- Hazards over the operands

  They are verified inside an hazard detection unit in the datapath, which resets a signal haz_neg in case there is a data dependency over at least one operand.

- Structural hazards due to unavailability of functional units

  These hazards may be raised because some units can process only one instruction at a time (e.g. the divisor), but also because the paths between adjacent stages is not superscalar, so only one instruction per clock cycle can be promoted to the latest stage. This means that there could be the need to stall one instruction in a functional unit if multiple instructions become available at the same time. Both cases are covered by setting a busy signal, checked by the CU, for each functional unit.

- Structural hazards due to a full buffer

  This could happen when ROB or RS are full. The first case is explicitly tackled by the CU, thanks to the ROB_full signal, while the second one is handled by the datapath, which checks the RS_full_int when generating the enable signal for the intermediate decode pipeline stage. In case the ROB is full, the F2D_en signal is reset to avoid promoting new instructions in decode while the current instruction is stalling there.

- Control flow hazards that may rise when a branch is found out to be mispredicted

  In this case we need to clean the whole pipeline, by setting all the clear signals to 1, all the RAT entries, the memory buffer entries and all the ROB entries which follow the branch that

has been mispredicted. The cleanup is performed only when the branch is committed, because if it was performed during write-back there would be the risk to eliminate instructions executed after the branch despite being located before the branch in the code.

CHAPTER 5

Datapath

5.1 Introduction

The datapath is a collection of functional units responsible for data processing operations within a computer system. It manages the execution of instructions defined in the Instruction Set Architecture (ISA) like load, store, and ALU (Arithmetic Logic Unit) operations.

To enhance the performance, the processor is divided into five stages that work in parallel. This approach significantly improves the throughput (i.e. the number of instructions executed per unit of time) since approximately one instruction can be executed per clock cycle. In non pipelined scenarios, each instruction would take 5 clock cycles to be completed.

5.2 Fetch

The Fetch stage is the first one in a pipelined processor, and it is in charge of retrieving from memory the instruction to be executed. Our fetch stages can be summarized in three parts:

- The Branch Target Buffer (BTB)
- The assignments which handle the definition of the next PC value
- The interface with the instruction memory

In particular, the next PC value can assume three possible values:

- PC+4 when the instruction that we are committing is not a mispredicted branch.
- The result of a committed branch instruction when there is a misprediction and the prediction was "untaken".
- The address of the branch plus 4 if there is a misprediction and the original prediction was "taken".

5.2.1 BTB

The BTB is the unit which performs branch prediction. A BTB entry consists in:

- A valid bit: to tell if the entry corresponds to a taken branch. Untaken branches are not logged inside the BTB, because the default prediction if the instruction is not found is already "untaken".

- The address of the branch: the complete address of the branch instruction which occupies that entry.
- The branch target: the target address of the branch instruction located at the address stored in the entry.

The BTB works on instructions when they are still undecoded, so it provides a prediction for each instruction without knowing if this instruction will be identified as a branch or not. This mechanism works because an instruction is predicted to be a "taken branch" only when the whole instruction address is found inside the BTB, and because only branches are written inside the BTB. In fact, the write is performed only when committing a predicted branch. This means that even if a non branch instruction corresponds to the same BTB entry of a taken branch (i.e. they share the same 8 LSbs, so the same addr value), then the instruction will not be considered as a "taken branch" because the value of jmp_addr (the complete address of the instruction) will surely be different from the one found in the BTB entry (the complete address of the taken branch). The interface of the BTB consists in the following signals:

- "addr" represents the 8 least significant bits of the address of the instruction we are considering. When execution is going on normally (i.e. no mispredicted branches are being committed) the instruction considered is the one being fetched, otherwise we consider the address of the branch that we are committing. The 8 least significant bits are used as an address inside the buffer, so the buffer itself is 256 entries long.
- "target" is the target address of the branch. This value is meaningful only when the branch is taken, otherwise the whole BTB entry is marked as invalid since the default prediction is "untaken". This value is written in the BTB entry when there is a taken branch that is being committed; the entry is selected looking at addr.
- "jmp_addr" is the complete version of addr. It has to be written in the BTB entry because when making the prediction we have to check if the instruction that is present in the BTB is the same branch that we are trying to predict (we need to consider the whole address for the comparison).
- "res" signals if the committed branch is taken or untaken.
- "rw" is a read-write signal. It is reset when we are reading the BTB (i.e. when fetching a new instruction, to make a prediction) and it is set when a committing branch is found out to have been mispredicted.
- "taken" holds the prediction for the instruction being fetched.
- "predicted_target" is the predicted target for the instruction. It is meaningful only when the instruction is found inside the BTB, because this is the case in which the instruction is a "predicted taken" branch.

5.3 Decode

The decode stage is divided in three sub stages, as a part of an optimization that has been devised to reduce the critical path length. The first stage is in charge of performing ROB and RAT allocation, the second one checks the availability of the operands and performs the allocation in the RS if needed and the last one is in charge of reading the operands for the instruction which is leaving the RS. Let's discuss the three stages and all the components that are part of these stages:

5.3.1 RAT

As anticipated in section 3, the RAT is the unit in charge of implementing register renaming. It is accessed during the first decode substage, to mark the destination register of the instruction currently in this substage with the ROB index of the entry where the instruction will be allocated. Moreover, it is also accessed during the commit, when the destination register is unmarked if altered last by the committing instruction. In fact, from that point on, the value will be accessible directly from the register file. in the following, the meaning of each port of this component is discussed:

- RD1/RD2: the register where the first/second operand for the instruction is stored. The corresponding entry has to be checked to assess if the operand is available in the RF or it is produced by an instruction currently in the ROB.
- WR1: register where the result of the committing instruction has to be written.
- WR2: register where the result of the instruction being decoded has to be written.
- inp: ROB entry associated to the instruction that will modify the WR2 register.
- inv: the signal that tells the RAT if the entry associated to WR1 should be invalidated or not during commit. An entry is invalidated only when the ROB entry stored in that RAT entry is the same of the instruction being committed.
- WR2_en: it tells the RAT if the entry associated to the WR2 register has to be overwritten with inp. This should not be done for instructions that do not write a value in a register.
- rob_entry_to_be_deleted: the ROB entry corresponding to the instruction being committed.
- out1/out2: the ROB entry stored in the entry corresponding to register RD1/RD2.
- out1_valid/out2_valid: the valid bits corresponding to the out1 and out2. If the bits are set to zero, then the entries are not valid. This means that the operand can be found directly in the register file.

5.3.2 ROB

The reorder buffer is used to store instructions that currently reside in the pipeline. The instructions are allocated during the first half of the decode stage and they are removed in the cycle they are committed. The ROB is in charge of keeping all the signals that are needed during the execution of each instruction, like the control word, PC+4 and the branch prediction outcome, together with the values that are produced as a result of the instruction execution, such as the ALU result or the actual branch outcome. In theory, five read ports would be enough for the ROB to work correctly: two should be used when assessing the state of the operands for the decoded instruction (this information comes from the RAT); the third port is used to retrieve the control word when the instruction execution is resumed (i.e. when it is sent to the EXE stage) after keeping it stored in the RS, waiting for the operands to be available; the last two ports are used to read the operands for an instruction which is leaving the RS. However, some arithmetic units can execute many instructions at the same time, mainly because of pipelining, making it unfeasible to store every piece of information for each instruction in different memory elements. In order to tackle this problem, we decided to send to the EXE stage only the information that are strictly needed by the stage, keeping the remaining ones inside the ROB. The remaining data are read only when the instruction is promoted to MEM stage, so we need an additional ROB port to perform this read operation which should be accessible from the EXE stage. Another port has been added to retrieve the value written in memory (regB) by an uncommitted store instruction according to the mechanism we discussed in chapter 3, because the

only thing we actually store in the memory buffer is the ROB entry of the instruction which altered a certain memory address last. All the most important ports are discussed below (the remaining ones can be easily inferred by checking the comments in the code):

- WR1: ROB entry of the instruction to be committed.
- inval: set to 1 if the instruction addressed by WR1 has to be deleted.
- WR2: ROB entry of the instruction that is terminating in write-back stage.
- ins_terminated: set to 1 if there is a new instruction in write-back.
- ins_res: result produced by the instruction that is terminating in write-back.
- branch_res: result of the branch that is terminating.
- regb: value for regb of the instruction that is leaving decode. It is read when a load instruction that comes after tries the read the memory location that the store was supposed to access.

All the read ports that follow are associated to the corresponding output port from which the result can be read:

- RD1/2: ROB entry associated to the first/second operand for the instruction being decoded.
- RD3: ROB entry associated to the instruction which is leaving the EXE stage, in order to read the additional information that have not been read in the passage between DEC and EXE.
- RD4: ROB entry associated to the instruction which is leaving the RS because its operands became available.
- RD5/6: ROB entry for the instruction that produces the first/second operand for the instruction that is leaving the RS.
- RD7: ROB entry associated to the store instruction that performed an uncommitted write operation in a certain memory location. It is accessed when a load reads a memory address on which there is an uncommitted store instruction pending: the ROB entry of the store is read from the memory buffer, and it is used to retrieve the value from the ROB.
- WR_line: the instruction to be allocated in the ROB.
- en_line: set to 1 if the allocation of WR_line in a new ROB entry must be performed in the current clock cycle.
- WR_ctrl_word: the control word for the instruction to be allocated.
- branch_prediction: branch prediction for the instruction to be allocated. To be kept here in order to determine if the prediction in the BTB should be updated when the branch is committed.
- PCN: the address of the instruction to be allocated plus 4.
- allocation_done: raised by the ROB when the allocation of WR2 is completed.
- reg_modified: the register which has to be modified by the committing instruction.
- res: the result to be written in reg_modified.
- to_be_written: an enable signal for the writing operation to be performed on reg_modified. It is 1 only if the instruction result is actually written in a register.

- rob_full: raised when the ROB is full, so no other allocations can be performed until at least one ROB entry is freed (i.e. one instruction is committed).
- is_mispredicted: raised if during commit the committing instruction is recognized as a mispredicted branch.
- WB_instruction_is_head: raised if the instruction which has just performed the write-back phase is recognized as the one stored in the head position of the buffer. This is used by the datapath to instruct the ROB to start a new commit strike with the instruction that is terminating as the one committed first.
- rob_entry_leaving_dec: the ROB entry for the instruction which is leaving DEC stage, either coming directly from decode susbstage 2 or from the reservation stations. It is an additional write port which is used to write in the ROB the value of regb, which could be used in case the instruction is a store as we discussed before (in the "regb" entry of the list).

5.3.3 Register File

The register file is a simple module that is mainly accessed when there is the need to retrieve register values that have not been modified by uncommitted instructions. A write operation over the register file is needed when an instruction that modifies a register is committed, together with an update of the ROB, to eliminate the committed instruction, and an update of the RS, to mark the register as available in the register file. This last information is needed by all the instructions which use the destination register of the committed instruction as an operand. The ports of the RF are:

- enable: to enable the register file globally.
- rd1/rd2: enable for the first or the second read port.
- wr: enable for the write port. This has to be driven by the ROB, which knows the control word for the instruction that is currently undergoing commit.
- add_wr: the address/index of the register whose value has to be updated.
- add_rd1/rd2: the register that has to be read as an operand. It can be driven either by the decode logic, when we have to read an operand to send the instruction directly to EXE from DEC, or by the RS logic, if an instruction in the RS uses an operand stored in the RF.
- datain: the value to be written in register add_wr.
- out1/out2: the value read from register add_rd1/rd2.

5.3.4 Reservation stations

The reservation stations are buffers where the instructions whose operands are not ready yet are temporarily stored. Each entry tracks the availability of the operands for the instruction that is stored there, including the location where the operand is stored; this could be either the ROB, if the instruction which produces the operand has not been committed yet, or the RF, if the operand was already available or it was produced by an already committed instruction. When both the operands are ready, the instruction is sent to the third decode stage, which is in charge of reading the operands from ROB/RF in the following clock cycle. This third stage cannot be stalled because it would block the RS output, so we should pay attention to how we select the instruction which can leave the RS. A critical case could be when we output a division from the RS because the divider is free (the busy signal coming from the EXE stage is low), then, in the next clock cycle, another division becomes

available in the RS while the previous one is still in the third DEC substage (i.e. it has not reached the divider yet). In this case the busy signal from the divider would still be off, thus allowing the second instruction to be selected to leave the RS. In the next cycle, however, the first instruction will reach the divider, keeping it busy for at least the next 34 cycles, preventing the second instruction from proceeding and thus stalling the third decode stage. This would effectively block the reservation stations output until the first division does not leave the EXE stage. The same can happen with any other operation, for example an addition that is stalled in ALU because multiple instructions from different units are competing to leave the ALU at the same time (the EXE-MEM path is single-issue): in this case the second addition would be stalled in the third substage until the first one is allowed to leave the EXE. In order to avoid that, we can avoid outputting two instructions associated to the same ALU units in two consecutive cycles: in this way the first instruction will have time to propagate to the relative unit before another one to be processed by the same unit is selected as output from the RS. An optimization has been performed for instructions which make use of the adder, since these are relatively common: we decided to avoid stalling these instructions in the ALU by always assigning them the highest possible priority when leaving the EXE, in this way it is guaranteed that an instruction using the adder will keep it busy for one clock cycle maximum. This allows us to output two additions consecutively from the RS, thus implementing a more efficient pipeline with less NOP bubbles in the third decode substage. Some important ports and the way in which they are used are presented below:

- allocation_done: the signal which notifies the datapath about whether the allocation succeeded or not.
- instruction_type: an input which tells the RS circuitry in which reservation station the instruction must be allocated, depending on the functional unit where it will be executed.
- add/logicals/shifter/mul/div_busy: the busy signals to inform the RS if some ALU units are currently busy. In case there is an addition available but the adder is busy, the addition doesn't leave the RS until the adder is freed. As we discussed before, we should avoid outputting two instructions processed by the same unit consecutively from the RS, because the one which resides in the third decode stage is invisible to the busy checks. The only exception for this rule consists in operations involving the adder, because their permanence in EXE is guaranteed to be one cycle long.
- is_terminating: set to 1 if an instruction is terminating (i.e. the destination register will become available in the ROB). In case there is an instruction which was waiting for that operand, the availability state will be changed at the end of the current clock cycle. This means that there is no forwarding mechanism to allow the instruction to rely on the value while the generating instruction is in write-back. The forwarding mechanism was present once, but we removed it to cut a possible critical path.
- rob_entry_terminated: the ROB entry of the instruction that is terminating.
- commit: set to 1 if there is a commit undergoing.
- new_ins: set to 1 if a new instruction has to be allocated. If an instruction is allocated in the RS but it has not yet been allocated in the ROB, for example because the ROB was full, the instruction is stalled in the second DEC stage until at least a ROB entry is freed. However, the allocation in the RS is done only once so the new_ins is lowered after the first cycle. The same happens if the allocation in the ROB is immediately available but the one in the RS is not.
- new_rob_entry: rob entry of the new instruction to be allocated.
- new_op1/op2: the register which is used as first operand for the instruction to be allocated.

- new_valid_op1/op2: the valid bits for the corresponding operand. These two bits can assume three meaningful values: "00" in case the operand is not yet available, "01" when it is available in the ROB, "10" when it is available in the registers. The instruction is flagged as available to be forwarded to EXE when the states of all the operands used by the instruction are "01" or "10".
- rob_entry_committed: rob entry of the instruction that is being committed.
- reg_written: register which becomes available in the RF as part of the commit.
- ins_is_leaving: it signals if there is an instruction currently leaving the RS. Needed in order to force the instruction in DEC substage 2 to be stored in the RS, since in the datapath logic we decided to assign an higher priority to instructions coming from the RS. The rationale behind this decision is that the instructions that were stored in the RS are much probably related to other instruction which are stored in the RS as well (e.g. in case of a piece of code with many dependencies). This means that preferring the incoming instructions would cause to starve the instructions in the RS, possibly reaching a point where all the fetched instructions have a dependency on the instructions in the RS.
- rob_entry_leaving: the rob entry of the instruction leaving the RS.
- rob_valid_op1/op2: the valid bits for the operands of the instruction that is leaving the reservation stations, so that the datapath is able to determine where to fetch the operands.
- rob_rf_entry_op1/op2: the ROB entry or the register where to look for the operand. The place where to retrieve the operand depend on rob_valid.
- RS_full: if the RS where the instruction has to be allocated is full. In this case it is necessary to stall the DEC and the FETCH stage.
- rs_full_tc: the full flags for each reservation station.
- type_of_ins_leaving: a one hot representation of a string which tells which kind of instruction is leaving the RS.

The remaining parts of the decode stage implement the logic that is needed to connect the components that we previously discussed about (RAT,ROB and RS) and all the circuitry that is needed to handle the three substages.

5.3.5 A side note about hazards that could stem from dividing the DEC in multiple stages

Adding an intermediate decode substage which separates the RAT read from the ROB read complicates the hazard handling. In particular, we should take into account the case in which the instruction which produces one of the operands for the incoming instruction is committed while the new one is located in the first substage. In fact, in this case the ROB entry read from the RAT must be ignored, because in the next cycle the associated instruction will not be in the ROB anymore. In order to keep into account this scenario, we introduced a "bypass" which intercepts the instruction triggering this behavior (i.e. the one being committed) and marks the operands as available in the RF.
There also is the possibility to add another stage to separate the ROB update from the RAT update as a part of the commit phase. However, this tweak would not cut the current critical path. For this reason, this solution has not been implemented in the current version of the DLX, but it should be considered for further reductions of the critical path. While introducing new stages allows to decrease the length of critical paths, it can cause to complicate the hazard handling. In fact, in this case, it

would be necessary to consider that during the second commit subcycle the instruction would not be in the ROB anymore (the invalidation would be performed in the first subcycle), so we should pay attention to incoming instructions which are potentially reading a ROB entry from RAT that is related to an instruction that doesn't exist anymore (in the current cycle too, not only in the next as we described before).

5.4 Execute

The EXE stage is the one which performs the possible computation required by an instruction starting from the operands. It can execute multiple instructions in parallel in different instruction units and the amount of instructions per unit depends on the specific unit considered since some of them can be pipelined. Exactly as the other stages, the EXE is not superscalar in the current implementation, so it can output only one instruction per clock cycle. This could lead to problems in case multiple instructions from different units terminate at the same time, so it is necessary to implement a logic to associate different priorities to different instruction units. With this mechanism the instruction from the unit with the highest priority will be promoted to memory, while the other ones will stall. The priorities change dynamically thanks to a FSM, which changes state when an instruction is allowed to leave the EXE stage. This FSM, named FSM_DECODE, prioritizes different kinds of instructions in different states, defining a priority sorting for each state. This is a small caveat regarding this FSM: in each state the priority of the adder is still the highest one. This is needed because it is the only way to pipeline the third decode substage for additions (allowing two additions to be outputted from the RS in two consecutive cycles, as explained in section 5.3.4), thing that speeds up the performances considerably since the operations involving the adder are very common in generic programs. The highest amount of complexity for this stage is located inside the ALU , the components which actually performs the computation. Ideally, there would be the need to implement inside the EXE stage also a unit which performs the computation of the branch outcome. As we will discuss, this kind of computation relies on performing a subtraction and checking the result and the carry. We tried to implement the needed circuitry directly inside the EXE stage, but it created a very long path which, in the end, resulted to be critical. In order to cut it down, we decided to place the comparison circuitry in memory, since there is no need to compute the branch outcome in execute. This would only be a problem in the original pipelined version, because the branch resolution is performed in memory. This solution allows to use the memory stage also for instructions which do access the memory, thus "rebalancing" the pipeline.

5.4.1 ALU

The ALU is in charge of preparing the operands for the execution units and performing the actual computation. It must also select which instruction has to be promoted to memory when there are multiple ones available in different execution units, so there is the need to implement the previously discussed priority assignment. The output of each unit can be stored in a register in case the result is ready but the unit is not connected to the ALU output in that clock cycle. When the unit will be selected, the value from the register will be read. If the unit takes only one cycle to complete the computation, it will be marked as busy whenever the output register is full, because computing a new instruction could potentially lead to a stall in the whole pipeline if the one currently in the output register is not sent to MEM. An example of this critical scenario is when we have two shift instructions one after the other, but the first one is stalled since there is another unit terminating with an higher priority. If we send the second instruction to the shifter while the previous instruction is still not selected, we would be forced to stall the whole EXE stage waiting for the first shift to be selected. This happens because the shifter is sourced directly by the D2E pipeline register.

Each instruction is associated to three pieces of information while it resides in the ALU: the result produced, the ROB entry where it is stored and its control word. This is the minimum amount of information that is needed to correctly perform a computation without losing track of the remaining information stored inside the ROB. Minimizing the information to be stored inside ALU units is vital especially for the multiplier, which is pipelined and it has been implemented relying on the Booth algorithm; this means that there are 16 stages that ideally can host up to 16 instructions, thus making the amount of information per pipeline stage a critical parameter to avoid using too many sequential elements.
Let's discuss all the ALU ports to clarify their meaning:

- inp1: input 1 for the ALU
- inp2: input 2 for the ALU
- op: ALU_op, provided by the CU
- unit: one hot representation of the unit where the operation will be performed
- res: result on 32 bits
- carry_out: carry out from the adder
- terminal_cnt: terminal count, used as a validity bit to tell if a new instruction has to be written in the E2M register
- divType: for the type of division (1 signed, 0 unsigned)
- ins_in: instruction entering the ALU, needed because load must be stalled in the ALU when there is already a load in memory waiting for its operand
- ROB_entry_in: ROB index of the instruction
- is_branch: 1 if the incoming instruction is a branch
- beqz_or_bnez: 1 if the incoming instruction is a bnez, otherwise it is 0
- add/logicals/shifter/mul/div_busy:
- ROB_entry_out: ROB entry of the instruction that is leaving, additional data will be retrieved from the ROB when the instruction leaves the execute stage
- branchres_out:
- inp_branch:

5.4.1.1 Adder

The adder we included in this design is the Pentium 4 one, which has been replicated as a part of laboratory assignments. This kind of adder is made of two parts: a network to implement the sum starting from carries and couple of bits and a "look-ahead network" to compute the carries. The first network is built with carry select adders. This kind of adder uses the carry only as a way to select between two results which are computed in parallel to the carry computation. The carry computation algorithm relies on combining general propagate ($P_{i,j}$) and general generate ($G_{i,j}$) values to compute propagate and generate values for each carry. In particular, in this algorithm, carry for stage i depends also on carries computed for previous stages. The main advantage of this approach with respect to the one based on a pure carry look-ahead adder is a reduced fan-in and fan-out of the gates which are included inside the tree; aspect which is critical in a fully unrolled look-ahead adder.

5.4.1.2 Shifter

Shifters are digital circuits used to perform shifting operations on data moving the bits of a binary number to the left or right by a certain number of positions.
In particular, a shifter is characterized by three inputs and one output. The three inputs are: the number to be shifted, the direction of the shift, the number of positions by which we have to perform the shift. The output is the result of the shift operation.

There are two main types of shifting operations that shifters can perform: logical shifts and arithmetic shifts. Logical shifts are bitwise operations that move the bits of a bit string to the left or to the right with a zero padding (i.e. 0's are added in the shifted positions). There are two subtypes of logical shifts:

- Shift Left Logical (SLL): In a left logical shift, the bits of the binary number are moved to the left by a specified number of positions. The vacated positions on the right are filled with zeroes. This operation effectively multiplies the number by a power of 2.

  Example: ‘‘‘ Original Number: 0010111 After LSL by 2: 10101100 ‘‘‘

- Shift Right Logical (SRL): In a right logical shift, the bits of the binary number are moved to the right by a specified number of positions. The vacated positions on the left are filled with zeroes. Example: ‘‘‘ Original Number: 11011011 After LSR by 2: 00110110 ‘‘‘

Arithmetic shifts, unlike logical shifts, take into account the sign bit when shifting. They are used when working with signed numbers to ensure that the sign bit is preserved. In this case we have only the Shift Right Arithmetic:

- Shift Right Arithmetic (SRA): In a right arithmetic shift, the bits of the binary number are moved to the right by a specified number of positions. The vacated positions on the left are filled with bits that have the same value of the MSB. This operation effectively performs a division by a power of 2 for positive numbers.

Original Number	10110100 (negative)
After SRA by 2	11101101 (still negative)

In the implemented shifter, one bit is used to choose between a left or right shift and, after this step, the operation is organized in two/three phases:

1. In the first phase we prepare 8 possible "masks", each shifted on 0,8,16 or 24 bit.

2. In the second phase, we choose the mask that is the nearest to the shift amount using the bit 4 and 3 of R2. In this way we compose the shift in two consecutive phases, by interpreting it as a shift by a multiple of 8 + a shift by a number between 0 and 7. This last one will be done in step 3 over the mask selected in the current step.

3. In the last phase, the shift between 0 and 7 is done over the mask according to the bits 2, 1 and 0 of operand R2. This shift simply consists in selecting a certain window of bits from the mask, since each mask is 40 bits long.

5.4.1.3 Logicals

We implemented the circuit for computation of logic functions according to the structure we studied for the T2 architecture. The peculiarity of this model resides in the fact that every logic function is implemented by combining together the results of different NAND3 gates which are sourced by the operands (asserted or complemented) and by a bit which depends on the operation. There is a total

of 4 NAND3 gates, one for each combination of the inputs asserted or negated. The combination is done through another NAND gates with 4 inputs which receives the 4 outputs of the NAND3 gates. This kind of operation is equivalent to selecting minterms from a truth table. This behavior can be showed with a simple example which consists in the implementation an OR gate. In order to do the OR function we need to define the proper values for the third bit of each NAND3 gate. Let's call S this string of bits, where S_0 is sent in NAND with A' and B', while S_3 with A and B. Let's show that S="0111" is the correct assignment for S if we want to implement an OR function (O_0 to O_3 are the outputs of the intermediate gates):

$$O_0 = \overline{S_0 A' B'} = \overline{0 A' B'} = 1$$

$$O_1 = \overline{S_1 A B'} = \overline{1 A B'} = \overline{A B'}$$

$$O_2 = \overline{S_2 A' B} = \overline{1 A' B} = \overline{A' B}$$

$$O_3 = \overline{S_3 A B} = \overline{1 A B} = \overline{A B}$$

$$O = \overline{O_0 O_1 O_2 O_3} = \overline{1 (\overline{A B'})(\overline{A' B})(\overline{A B})}$$

As we can see from the last expression, it is enough that one of the intermediate results is zero to make the whole expression evaluate as one. A zero from an intermediate result is possible only if it selected for the operation we are trying to compute (S_X is 1) and both the inputs for the corresponding NAND3 are 1 (it's like observing the corresponding minterm in the truth table). If we put everything together, the expression will evaluate to 1 if A=1 and B=0, A=0 and B=1 or A=1 and B=1. Exactly as we did for the OR function, we can infer the values for S to implement whichever logic operation we desire.

5.4.2 Divider

The divider implements a non-restoring division algorithm that works only on unsigned values 5.1. Despite this limitation the division can be performed over both signed and unsigned values. This is achieved through additional logic that prepares the operands before starting the division. This setup lasts for 1 clock cycle when there is at most 1 signed operand, 2 clock cycles otherwise. The division algorithm lasts 33 clock cycles because it has to perform a number of iterations equal to the number of bits of the operands and a last clock cycle to eventually restore the remainder. Moreover, at the end of the division, additional clock cycles may be used to convert the quotient and the remainder. In particular, 2 additional clock cycles are used if both values need to be converted, 1 additional clock cycle is used if only one value needs to be converted and no additional clock cycle is used if both values are positive (i.e. they do not need to be converted). This means that a complete division operation lasts 34 clock cycles in the best case and 36 clock cycles in the worst case. The case with 2 setup cycles before starting the actual division and 2 conversion cycles at the end is not possible. This can be demonstrated by looking at the boolean functions, reported in the following, that compute the sign of quotient and remainder using the operation type (i.e. signed='1' and unsigned='0') and the MSB of divider and dividend:

```
1     nextSignQ(0) <= (D(NBIT-1) xor Z(NBIT-1)) and opType; -- quotient sign
2     nextSignR(0) <= Z(NBIT-1) and opType; -- remainder sign
```

The divider is implemented resorting to a mixed approach that uses both structural and behavioural descriptions. In fact, the main blocks (i.e. adder, operands registers, counters and state registers) are implemented with specific modules while the logic that drives the control signals of these blocks and manages the divider in its different states is described behaviourally.

5.4.3 FSM_DECODE

As anticipated in section 5.4, there is the need to define a priority scale for each operation. In the proposed implementation, the priority scale changes dynamically. This mechanism is managed by a finite state machine which evolves through different stages, each one with a different priority sorting. The FSM basically acts as the producer for the selection input of the multiplexer which selects between the results coming from different functional units. As anticipated in section 5.3.4, the adder will always have the maximum priority in each FSM state. This guarantees that all the operations involving the adder will complete in exactly one clock cycle.

5.4.4 Booth Multiplier

The multiplier implemented in this architecture is a pipelined version of the classic multiplier based on Booth algorithm. The base upon which we built this implementation is the multiplier we implemented as a part of laboratory assignments. Inside the multiplier we used P4 adders to implement the addition needed in each stage, but it is possible to use less complex adders since the paths inside the multiplier are not critical in the final version of the processor. Implementing a pipeline means adding registers for each stage of the multiplier, preserving in these registers all the operands and the information related to the instruction which currently resides there. Handling the pipeline means handling the discontinuous flow of instructions that can propagate in the multiplier, allowing the instructions to proceed until the last free stage and waiting for the ones afterward to be done with the computation before proceeding further. This requires some intelligent circuitry to produce the enable signals that are needed to handle the propagation inside the pipeline, which has been implemented as a part of the following process:

```
1  -- enable generation process and propagation of pipeline signals
2       process(A, B, enables, new_mul, mul_to_mem, valid_curr, A_curr, B_curr, ROB_entry_curr)
       begin
3          -- map the last dummy enable to mul_to_mem
4          enables(18) <= mul_to_mem;
5          for i in 17 downto 0 loop
6            if(i=0) then
7              valid_next(i) <= new_mul;
8              A_next(i) <= A;
9              B_next(i) <= B;
10             ROB_entry_next(i) <= ROB_entry_in;
11             enables(i) <= enables(i+1) or (not valid_curr(i));
12           else
13             -- pipeline signal propagation
14             valid_next(i) <= valid_curr(i-1);
15             A_next(i) <= A_curr(i-1);
16             B_next(i) <= B_curr(i-1);
17             ROB_entry_next(i) <= ROB_entry_curr(i-1);
18             -- enable generation
19             enables(i) <= enables(i+1) or (not valid_curr(i));
20           end if;
21         end loop;
22       end process;
```

This process generates all the enable signals for each stage in the pipeline, considering the first hardware stage as a corner case (since it is fed directly by the D2E register). As we can see in lines 11 and 19, the enable of the generic hardware stage-i is set to 1 if: the next stage is enabled, so that the instruction currently in stage-i will move in stage-(i+1) in the next clock cycle, leaving space for the one currently in stage -(i-1); the stage i contains a void, because this one can be filled by a valid instruction currently in the stage-(i-1). This definition is recursive, because the condition under which an instruction is able to proceed depends on the availability of all the stages that are located afterward. Since a stage that is empty at cycle N can be filled with an instruction at cycle N+1 because of the condition which checks not valid_curr, a bubble in the multiplier pipeline can be saturated by instructions coming from the stages located before, even if there is a stall at the end of the pipeline.

An additional optimization has been implemented to avoid producing critical paths inside the multiplier stages. In fact, in each stage, the partial result coming from the previous stage is added to an operand whose value is chosen using the corresponding bits of the multiplicand. The algorithm is radix-4 so, supposing A is the multiplier, the possible values of the second operand are 0, A,-A,-2A or +2A, all multiplied by $radix^{stage}$. In stage-0 we would need to compute two operands, because the first adder in the multiplier structure operates on the values that are dictated by bits 3,2,1 and 1,0,-1 of the second multiplicand. It should be clear that generating these operands requires to perform multiple operations on A, in particular we may need an additional adder to compute the two's complement. To avoid having multiple adders cascaded (the two's complement one and the one used for the computation), we decided to precompute the second operand of a generic adder in the stage before the one in which it is actually used. The precomputation consists in shifting the second operand and computing its one's complement, since the two's complement is computed by passing a 1 as carry-in to the LSbs adder of the next stage (we compute the actual value of the second operand in two steps on different stages). The problem in doing so resides in the first stage, because we need to compute two operands in there, and we could potentially need to pass two carries to the next stage. This means that one of the two carries must be propagated through the pipeline and added later. In particular, we decided to do so with the carry-in that should be used to perform the two's complement in stage-1 first operand, which is passed through the pipeline and is added to the result in the second to last stage, another dummy stage added because of the optimizations. The same design choice described for the carry-ins has been done over the carry-outs that come from the LSbs adders. In fact, the computation of the 64-bit partial result of each stage is performed with two 32-bit adders instead of a single 64-bit adder: one 32-bit adder computes the lower 32 bits of the partial result (i.e. the LSbs adder) while another computes the higher 32 bits (i.e. the MSbs adder). As already seen with the previous optimization, the hardware structure needs to be changed w.r.t. to the logic structure of the algorithm. In this case, it is necessary to place the LSbs adder of stage-i in a certain hardware stage while the MSbs adder of stage-i is placed in the next hardware stage, together with the LSbs adder of stage-(i+1). With this configuration, the carry-out of an LSbs adder is propagated to the carry-in of the associated MSbs adder in the next hardware stage, allowing to have the delay of a 32-bit addition, instead of a 64-bit one, in each hardware stage. This configuration requires two additional stages at the end: in the first one we add the possible carry corresponding to a two's complement on the operands of the first hardware stage, thus producing the first half of the final result (i.e. the LSbs) and the intermediate carry-out; in the second one we combine the intermediate carry-out just produced with the MSbs, producing the other half of the final result. These two optimizations make the multiplier pipeline longer (i.e. they increase the latency), but they eliminate some paths which could become critical when trying to reach sub nanosecond clock periods. The hardware overhead associated to these modifications is very low, since they mainly rely on recycling some components or anticipating the computation of a certain value.

5.5 Memory access

The MEM stage is the one where there is the actual communication with the data memory. The stage itself is divided in multiple substages, in particular there is the possibility for the instruction to take two different paths depending on its type: if the instruction is not a load it can be sent directly to the M2WB pipeline register, if it is a store, then the update of the CAM and the memory buffer is done during a single clock cycle; if it is a load, then we need to keep the instruction in MEM for an additional cycle, so we send it to an intermediate pipeline register. This flow is actually possible for non load instructions only if there is NO load in the intermediate register when they reach the MEM stage, otherwise they have to take the long path too. This is because if they took the short path then they would compete with the load for the promotion to WB. This situation should be avoided, because

it would make necessary to use a signal to be sent to EXE to stall other incoming instructions. This stalling would propagate backward, thus making a very long path traversing multiple pipeline stages. The reason behind the need for two stages is that processing a load instruction requires accessing the CAM to read if an uncommitted store accessed the same address and then accessing the ROB to read the last store result, in case there is a CAM entry associated to the address; since these two accesses are serialized, we needed to introduce a pipeline substage to cut in half what could have been a critical path. If there is no CAM entry associated to the address accessed by the load, the memory is accessed in the second cycle instead. Actually both the ROB and the memory are accessed in parallel, but then we select only one of the two values through the multiplexer below:

```
1     M2WB_MEMres_next <= data_from_DM when committed_in_first_cycle_curr='1' or CAM_match_curr='0'
      else ROB_regb_store;
```

The multiplexer selects the data coming from the memory if there is no CAM entry (CAM_match_curr='0') or when the last store instruction modifying that address has been committed while the load address was residing in the first MEM substage. This last check is very important, because if the store is committed during the first MEM cycle then the ROB entry which is read from the CAM doesn't make sense anymore during the second one. This means that the value has to be read directly from memory because it has become available there at the end of the clock cycle before. As a further complication, the memory write when a store is committed lasts two cycles: in the first cycle the value to be written in RAM is read from the ROB (it corresponds to the regB operands of the store instruction), then, in the second cycle, the update of the CAM is performed in parallel with the actual write operation on the data memory. This is needed again to reduce the length of a path that is among the critical ones, which is the one traversing ROB and CAM.
The circuitry which handles the propagation through the intermediate stage is described by a process, which uses the following line as a condition to move the incoming instruction to the intermediate stage:

```
1     if(is_load='1' or (valid_load_curr='1' and E2M_valid_bit_curr='1')) then
```

The same process handles the two cycles of a store commit.
Let's now discuss the CAM, the only component that is actually present in this stage.

5.5.1 CAM

As discussed thoroughly before, the CAM is needed to keep trace of all the memory addresses that have been targeted by uncommitted store instructions. In order to track the instructions we need to store two values: the address accessed by the instruction and the ROB entry where it has been allocated. The remaining information (i.e. the value to be stored in memory) can be retrieved from the ROB when committing an instruction, so there is no need to preserve it in a buffer. However, this comes at the expense of a further ROB read port which ultimately increases the ROB fan-in. Since the functionalities of this module are well known, we can proceed discussing all the ports and the way in which they are related to the functionalities which the module should implement:

- R: is the input ports where we provide the address that we are performing the check over, so it corresponds to the address produced during a load.
- W_inval: the address to be invalidated, driven by the ROB output, which corresponds to the address overwritten by the store that is being committed.
- ROB_inval: the ROB entry of the store instruction that is being committed. It has to be compared to the ROB entry stored in correspondence of the address to be invalidated, because the invalidation has to be performed only if the store that is being committed was the last one to perform a write over that address, otherwise we would invalidate the entry corresponding to another store.

- W_regb: the memory address accessed by the store. It is the value that is written in a new entry, together with the ROB entry, whenever a store passes through the memory stage.
- ROB_write: the ROB entry of the store instruction which is producing a new CAM entry.
- W_en_inval: set to 1 if the invalidation of an entry has to be performed (only when a store is being committed).
- W_en_regb: set to 1 if a new allocation has to be performed.
- M: set to 1 if there has been a match in the CAM, which means that a valid entry associated to the address accessed by the store has been found, so the result has to be read from the ROB and not directly from the memory.
- rob_entry: the rob entry corresponding to the CAM entry which produced the match.

5.6 Write-Back

The write-back, in this architecture, is a very simple stage, since it contains only the muxes that are needed to select the data to be written in the RF and the register where to write, other than the circuitry needed to update the instruction state in the ROB. The two muxes are reported below, but they are exactly as what we would have found in a simple pipelined architecture:

```
1     data_to_write <= M2WB_MEMres_curr when WB_res_sel="00" else M2WB_ALUres_curr;
2
3     reg_to_write <= M2WB_IR_curr(15 downto 11) when WB_dest_sel="00" else M2WB_IR_curr(20 downto
       16) when WB_dest_sel="01" else "11111";
```

5.7 Enable and reset signals

The enable and reset signals for pipeline stages are produced as follows:

```
1     F2D_en_int <= (F2D_en and dec1_to_dec2_en);
2     D2E_en <= haz_neg_with_RS;
3     D2E_rst <= (D2E_valid_bit_curr and not D2E_en) or D2E_clr;
4     E2M_en <= terminal_cnt;
5     E2M_rst <= (E2M_valid_bit_curr and not E2M_en) or E2M_clr;
6     M2WB_en <= (E2M_valid_bit_curr and not is_load) or valid_load_curr;
7     M2WB_rst <= (M2WB_valid_bit_curr and not M2WB_en) or M2WB_clr;
```

The enable signals that are not a function of other signals are taken directly from the CU. Let's discuss what are the enabling and reset conditions one by one:

- F2D_en_int is raised when F2D_en from the CU is raised and the dec1_to_dec2 pipeline register is active. This is needed because if an instruction stalls in the intermediate stage we need to prevent further instructions from being promoted to decode until the condition which is responsible for the stall is solved.
- D2E_en is raised when there are no hazards involving the instruction currently being decoded or there is an instruction ready from the reservation stations, considering also the structural hazards that may arise when the ALU units are full.
- D2E_rst is raised when the D2E register is forced to be cleared, because of a misprediction or because of the global reset, or when there is an instruction in the D2E but there are no new instructions that can be promoted to EXE. In fact, if we left the instruction currently in EXE in D2E for another clock cycle we would duplicate it when it is promoted to MEM. Instruction duplication in the pipeline is something we should absolutely avoid, because it could affect ROB

logic. In fact, we would ask for a double write-back over the same instruction. This is dangerous in case the original instruction has been committed before its copy performs write-back.

- E2M_en is raised when an instruction from the functional units is ready, which corresponds to terminal_cnt raised.
- E2M_rst follows the same logic behind the D2E reset signal.
- M2WB_en is enabled when either there is an instruction in E2M which is not a load (i.e. it takes the short path) or there is a load ready in the intermediate stage.
- M2WB_rst follows the same logic of E2M_rst and D2E_rst.

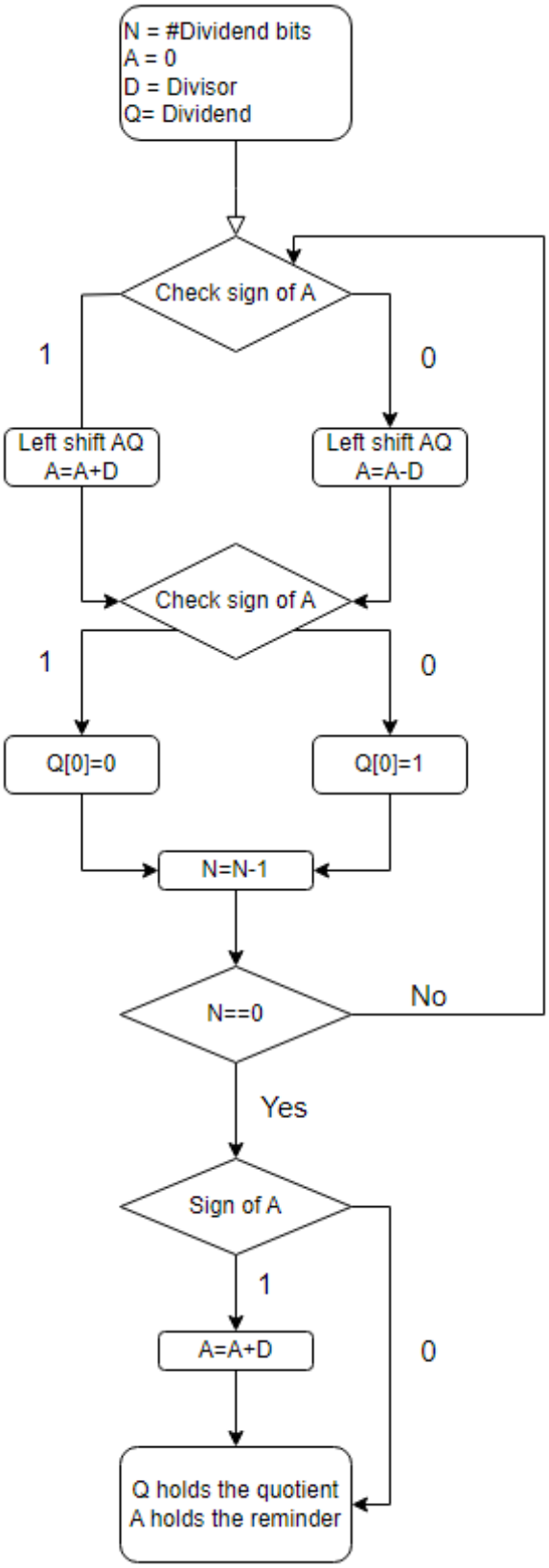

Figure 5.1: Non-restoring divison algorithm flow chart

CHAPTER 6

External components and Testbenches

6.1 Introduction

In order to run meaningful programs to test the processor functionalities it is needed to interface the core itself with some external components, mainly memories and testbenches. The memories we used in our design are two: a read-only memory used to store instructions to be executed (IRAM) and a DRAM for the data. After discussing the peculiarities of these two memories we will go on presenting the testbench, showing a program that has been written to test all the major mechanisms of the design (hazards, out-of-order execution, in order commit, speculation, etc) and a generic program implementing a bubble sort.

6.2 IRAM

As anticipated, this memory is a read only one whose contents are read from file. The actual memory behavior is given by these 4 concurrent assignments:

```
1     Dout(7 downto 0) <= IRAM_mem(to_integer(unsigned(Addr)));
2     Dout(15 downto 8) <= IRAM_mem(to_integer(unsigned(Addr))+1);
3     Dout(23 downto 16) <= IRAM_mem(to_integer(unsigned(Addr))+2);
4     Dout(31 downto 24) <= IRAM_mem(to_integer(unsigned(Addr))+3);
```

As we would expect the ROM behaves exactly as a combinational circuit, because the contents cannot be overwritten and assume the values that were assigned at the beginning.

6.3 DRAM

The DRAM has been implemented directly inside the testbench. As we would expect from a read-write memory we need two processes, one per each operation; the write process is synchronous. The read could ideally be done by simply resorting to concurrent assignments as we saw for the IRAM, but we decided to group them inside a process.

6.4 Testbench and test programs

The testbench has to initialize the IRAM with the content (i.e. the instructions encoded in base-16) from the compiled assembly program and run the program itself. As a part of the architecture simulation, the testbench rises the global reset signal to 1 for a certain amount of time to reset the pipeline, then it waits indefinitely.

As we hinted before, in order to test the architecture we had to develop an assembly program which exhibits all the instruction patterns that could become critical to handle for the out-of-order engine. Let's show the program we have written, highlighting some of those patterns:

```
1     addi r1,r1,4
2     addi r2,r2,5
3     mul r3,r2,r1
4     add r4,r2,r1 ; mul and add are run out of order; they write their results on different regs
5     div r5,r3,r4
6     add r5,r3,r4
7     mul r5,r1,r5 ; check if the mul uses the result of the add as r5
8     mul r6,r2,r1 ; executed in parallel with the previous one
9     nop
10    nop
11    nop
12    nop
13    nop
14    nop
15    nop
16    nop
17    nop
18    nop
19    add r8,r1,r2
20    add r8,r1,r2
21    add r8,r1,r2
22    add r11,r3,r4
23    add r10,r3,r2
24    add r9,r1,r4
25    div r7,r3,r4 ; executed in parallel with the two previous mul
26    mul r4,r3,r0 ; should reset r4
27    beqz r4,label ; make sure the div terminates execution
28    sw 0(r0),r2 ; should not be executed
29    nop
30 label:
31    mul r5,r1,r3
32    sw 0(r0),r2
33    lw r4, 0(r0)
34 infinite_loop:
35    j infinite_loop
36    exit
```

Here are some critical points that should be verified in simulation:

- The add at line 4 and the mul before are performed in an out-of-order fashion. We should check if they are committed in the correct order and if the operands are correct, since add and mul should wait for the previous addi to complete before being executed. Figure 6.1 shows the mul receiving the correct operands. Figure 6.2 shows the issuing of add and mul, which is done in order, as expected. In figure 6.3 it is shown how the commit for the two instructions is done in order too, even if it is clear how the add was the first one to conclude its execution (it reached state 2 first).

- The div and the add at line 6 are executed out-of-order and operate on the same register. We should verify that only the add result is written in r5. We choose the first instruction (i.e. the div) so that it is much longer the second one, in order to force the second one to complete first. The aim is to check that the result produced by the second instruction is not overwritten by the first one. It is possible to rely on the execution of the mul at line 7 to check if r5 assumes the correct value. This behavior is shown in figure 6.4, where it is evident how the mul receives the correct operands after the WAW.

- We tested a possible hazards rising on the beqz at line 27. Its execution depends on the r4 value produced by the mul which comes immediately before in the code. Since the mul requires many cycles to complete, the following instructions will be executed in a speculative way. This means that their results should not be permanent, so they should not be written into the RF or to the data memory. In our case the speculative instruction is a store (line 28) whose result should not

be written in memory. Speculative execution is shown in figure 6.5, through the actions taken when the branch commit is performed: all the instructions that are located afterward in the ROB are deleted and it is clear how some of them had already concluded execution. This means that those instructions will not be committed, so they will not alter the memory or the register file content. In figure 6.6 it is shown how the memory is not overwritten by the speculative store, which would have stored 5 at location 0 and 0s at locations 1,2,3.

- The load at line 33 reads the same memory location accessed by the store immediately before, so it should wait for the result to be available. As we saw before, the address marked by the store is written in CAM, so the corresponding value should be immediately available for the load. In figure 6.7, the two cycles when the load is residing in memory are shown: it is evident how the load reads the correct value (5) thanks to the tracking mechanism implemented by the CAM.
- We had to place a custom defined exit instruction at the end, in order to stop speculating over data which are not instructions.

Another program we wrote is the one implementing the bubble sort which is reported below:

```
1     addi r1,r0,12
2     sw 0(r0),r1
3     addi r1,r0,3
4     sw 4(r0),r1
5     addi r1,r0,5
6     sw 8(r0),r1
7     addi r1,r0,4
8     sw 12(r0),r1
9     addi r1,r0,10
10    sw 16(r0),r1
11    addi r1,r0,6
12    sw 20(r0),r1
13    addi r1,r0,7
14    sw 24(r0),r1
15    addi r1,r0,0
16    sw 28(r0),r1
17    addi r1,r0,24
18    sw 32(r0),r1
19    addi r1,r0,16
20    sw 36(r0),r1
21    addi r1,r0,0 ; external counter of iterations
22    addi r3,r0,36 ; limit for external iterations
23    loop_ext:
24        addi r2,r0,0 ; internal counter of iterations
25        sub r4,r3,r1 ; number of internal iterations
26    loop_int:
27            lw r5,0(r2) ; first value
28            lw r6,4(r2) ; second value
29            sle r7,r5,r6 ; r7 is true if there is no need to perform the swap
30            bnez r7,already_sorted
31                sw 0(r2),r6 ; write the second value in the first position
32                sw 4(r2),r5 ; write the first value in the second position
33        already_sorted:
34            addi r2,r2,4
35            sub r7,r4,r2
36            bnez r7,loop_int
37        addi r1,r1,4
38        sub r7,r3,r1
39        bnez r7,loop_ext
40    forever:
41        j forever
42    exit
```

For this program we simply verified that the memory contents at the end are the expected ones.

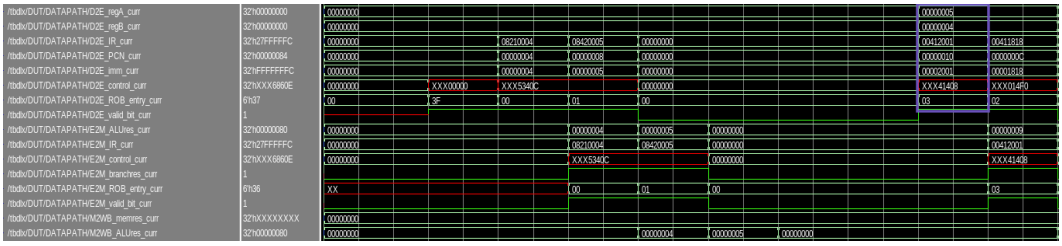

Figure 6.1: The operands 5 and 4 received correctly by the multiplication 0x412001

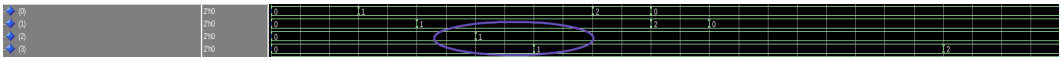

Figure 6.2: mul and add are issued in order as they appear in the code

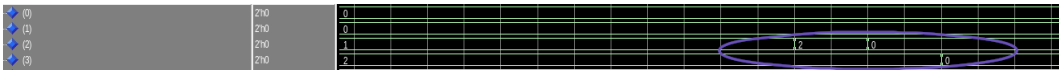

Figure 6.3: mul and add are committed in order, even if add concludes first

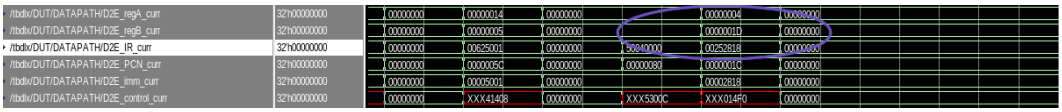

Figure 6.4: mul receives the correct parameters after the WAW involving div and add

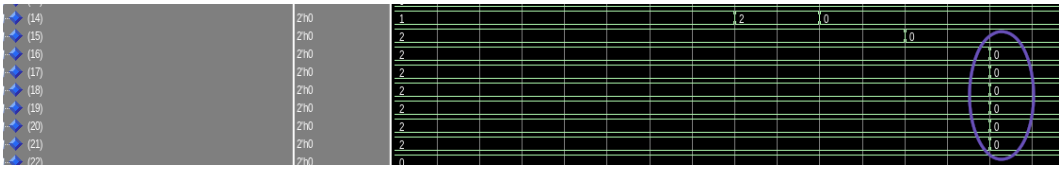

Figure 6.5: When a branch is committed all the speculative instructions are deleted

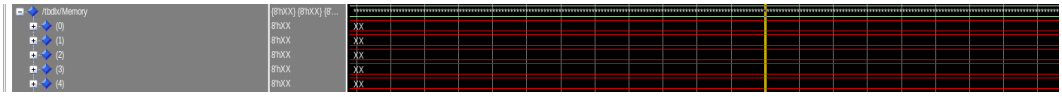

Figure 6.6: The memory is not overwritten by the speculative store

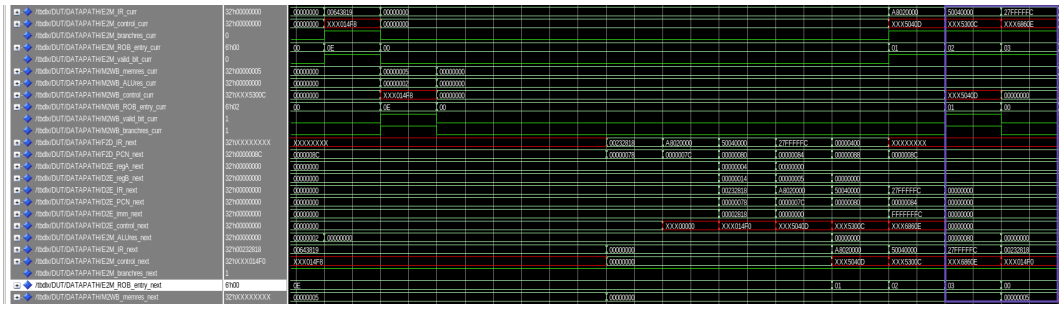

Figure 6.7: The two cycles that the load spends in memory

CHAPTER 7

Synthesis and Place and Route

7.1 Synthesis

In order to synthesize the architecture the following script needs to be run in Design Compiler:

```
analyze -library WORK -format vhdl {constants.vhd}
analyze -library WORK -format vhdl {iv.vhd}
analyze -library WORK -format vhdl {nd2.vhd}
analyze -library WORK -format vhdl {nd3.vhd}
analyze -library WORK -format vhdl {nd4.vhd}
analyze -library WORK -format vhdl {fa.vhd}
analyze -library WORK -format vhdl {rca.vhd}
analyze -library WORK -format vhdl {mux21.vhd}
analyze -library WORK -format vhdl {mux51.vhd}
analyze -library WORK -format vhdl {carry_select_block.vhd}
analyze -library WORK -format vhdl {sum_generator.vhd}
analyze -library WORK -format vhdl {g.vhd}
analyze -library WORK -format vhdl {pg.vhd}
analyze -library WORK -format vhdl {pg_elem_net.vhd}
analyze -library WORK -format vhdl {carry_generator.vhd}
analyze -library WORK -format vhdl {P4adder.vhd}
analyze -library WORK -format vhdl {encoder.vhd}
analyze -library WORK -format vhdl {boothmul_pipelined.vhd}
analyze -library WORK -format vhdl {counter.vhd}
analyze -library WORK -format vhdl {Left_shifter.vhd}
analyze -library WORK -format vhdl {PIPO.vhd}
analyze -library WORK -format vhdl {SIPO.vhd}
analyze -library WORK -format vhdl {divider.vhd}
analyze -library WORK -format vhdl {shifter.vhd}
analyze -library WORK -format vhdl {logicals.vhd}
analyze -library WORK -format vhdl {FSM_DECODE.vhd}
analyze -library WORK -format vhdl {ALU.vhd}
analyze -library WORK -format vhdl {BTB.vhd}
analyze -library WORK -format vhdl {CU_with_RS.vhd}
analyze -library WORK -format vhdl {registerfile.vhd}
analyze -library WORK -format vhdl {RAT.vhd}
analyze -library WORK -format vhdl {ROB_with_RS.vhd}
analyze -library WORK -format vhdl {CAM.vhd}
analyze -library WORK -format vhdl {reservation_stations_processes_separated.vhd}
analyze -library WORK -format vhdl {dlx_datapath_with_RS.vhd}
analyze -library WORK -format vhdl {DLX_with_RS.vhd}

# elaborating lower entities
elaborate LOGICALS -architecture BEHAVIORAL2

# elaborating the top entity
elaborate DLX -architecture STRUCTURAL -library WORK

set_wire_load_model -name 5K_hvratio_1_4

# Create clock and set constraints
create_clock -name "CLK" -period 1 clk
```

```
48 set_max_delay 1 -from [all_inputs] -to [all_outputs]
49
50 # optimize + mapping
51 compile_ultra -timing_high_effort_script
52 # registers (edge and level triggered) retiming + incremental compilation
53 optimize_registers -sync_trans multiclass -async_trans multiclass
54
55 # produce reports and backup the design ddc
56 report_timing > dlx_t.rpt
57 report_area > dlx_a.rpt
58 write -hierarchy -format ddc -output DLX_newstage_ultra.ddc
```

As first step, the script performs the analysis of all the VHDL files that are part of the design, thanks to the `analyze` command. The structure of the core is built in the work library, preparing the field for the `elaborate` (line 39 and 42), which performs a first translation of the design components to generic cells. Right after, we run the command `set_wire_load_model`, which sets the model for wires used during the place and route. The actual technology mapping is performed through the `compile_ultra` command (line 51), in charge of translating the formal cells to actual cells whose characteristics are described in the liberty file (i.e. the file that defines the cell library). The compile_ultra has been run with an option which prioritizes optimizations over the critical path delay, because we wanted to reach a sub nanosecond clock period. In order to implement a constrained synthesis, it is necessary to specify the constraints before performing the compile_ultra; this has been done in lines 47 and 48, where the maximum accepted delay has been specified. After performing the tech mapping, we run a command which performs retiming, an operation which consists in inserting additional registers where possible to cut down the critical path length. When the whole flow is complete, results are written in report files, which collect timings and area metrics.
The design has been changed multiple times to parallelize the computation as much as possible, to introduce additional substages or to move functional units to a different stage while keeping intact all the functionalities. This led to repeat the synthesis process many times. Some of the optimizations that have been performed as part of this iterative process have been summed up here:

- Decomposition of the decode stage in three substages, in order to avoid serial accesses to ROB/RAT/RS in the same clock cycle.
- Relocation of the comparison circuitry in memory to cut down the path passing through this circuitry and the adder.
- Division of the reservation stations in multiple processes, one per RS, generated with a FOR GENERATE statement. At the beginning, there was a single process which handled all the RS, however, this resulted in a serialization of all the computations. In fact, the different stations where handled as a part of a single FOR LOOP inside the process. Dividing the stations in different processes allows to perform the computation inside each RS in parallel, thus improving performances considerably. Some fine grained optimizations have been implemented too, like manipulating the position, in terms of nesting, of some conditional checks.
- Addition of a substage as a part of the CAM update.
- Division of the memory stage in two stages for load instructions, allowing the creation of two different paths inside the stage itself. This allows to promote immediately to WB all the instructions which do not require to access the memory.

The results obtained after the final synthesis process are the following ones:

- Data arrival time (i.e critical path length) : 1.38ns. This path is the one taken by a committing branch instruction whose entry in the BPU has to be updated. As already mentioned, the main responsible for this critical path could be the cells mapped to the memory elements. It would

be possible to break this path, by adding an intermediate pipeline stage, but this would delay the BPU update by one cycle; this should not be a problem since it is very rare to have two instances of the same jump executed close in time.

- Area: the report shows the following values for area metrics

```
Number of ports:                        1678
Number of nets:                       212599
Number of cells:                      197589
Number of combinational cells:        163158
Number of sequential cells:            34382
Number of macros/black boxes:              0
Number of buf/inv:                     10824
Number of references:                     25

Combinational area:            176685.180712
Buf/Inv area:                    8077.356009
Noncombinational area:         167318.255600
Macro/Black Box area:               0.000000
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:               344003.436312

```

The sequential logic area is very high, as expected, because we introduced a lot of memory buffers when implementing the circuitry for the out of order engine.

- Power: the power analysis produced the following results

```
                    Internal         Switching           Leakage             Total
Power Group      Power             Power               Power               Power   (   %    )
     Attrs
--------------------------------------------------------------------------------------------
io_pad              0.0000            0.0000            0.0000            0.0000  (   0.00%)
memory              0.0000            0.0000            0.0000            0.0000  (   0.00%)
black_box           0.0000            0.0000            0.0000            0.0000  (   0.00%)
clock_network       0.0000            0.0000            0.0000            0.0000  (   0.00%)
register        1.6779e+05          182.6642        2.7074e+06        1.7067e+05  (  93.15%)
sequential         33.4451            3.3204        1.0865e+04           47.6300  (   0.03%)
combinational   1.8750e+03        6.5980e+03        4.0391e+06        1.2511e+04  (   6.83%)
--------------------------------------------------------------------------------------------
Total           1.6970e+05 uW     6.7840e+03 uW     6.7574e+06 nW     1.8323e+05 uW

```

It is clear that most of the power consumption comes from memory elements such as registers. This kind of analysis is however very imprecise, because it assumes a switching activity of 50% for each node in the circuit. In order to perform a more accurate analysis we would need to perform post-synthesis simulation and collect the static probabilities of having 1 on each internal node, then using these values to compute the switching activities. This would lead to a more accurate estimation of the power consumption in a typical/common scenario; we should find a program that is similar to the ones that will be commonly run on the machine.

7.2 Place and Route

The complete script extracted from Innovus log has been provided in the folder we delivered. The time analysis we performed at the end of the place and route flow returned the following results:

```
------------------------------------------------------------
          timeDesign Summary
------------------------------------------------------------

+--------------------+---------+---------+---------+
|     Setup mode     |   all   | reg2reg | default |
+--------------------+---------+---------+---------+
|           WNS (ns):| -0.164  | -0.164  |  0.194  |
```

```
|           TNS (ns):| -86.132 | -86.132 |  0.000  |
|    Violating Paths:|  3375   |  3375   |    0    |
|          All Paths:|  49471  |  34110  |  33846  |
+--------------------+---------+---------+---------+

+----------------+------------------------------+------------------+
|                |              Real            |       Total      |
|    DRVs        +------------------+-----------+------------------|
|                |  Nr nets(terms)  | Worst Vio |  Nr nets(terms)  |
+----------------+------------------+-----------+------------------+
|   max_cap      |      0 (0)       |   0.000   |      0 (0)       |
|   max_tran     |     7 (254)      |   -0.073  |     7 (254)      |
|   max_fanout   |      0 (0)       |     0     |      0 (0)       |
|   max_length   |      0 (0)       |     0     |      0 (0)       |
+----------------+------------------+-----------+------------------+

Density: 61.566%
       (99.998% with Fillers)
------------------------------------------------------------
```

We haven't been able to solve some of the design rules violations on the transition times for signals that are transmitted on the core pins. The main responsible could be the clock, which among the signals on the pins is the one with the highest fan-out since it has to be redistributed to all sequential elements. A possible way to solve this problem could be to decrease the clock frequency (to allow higher transition times) or to increase the number of pins for clock distribution (to have lower resistances associated to the paths which distribute the clock). For what concerns the timings, we observed that there are no paths through which signal propagation takes more than 1.164 ns. The discrepancy between this result and the one we found out after synthesis may reside in the fact that a more accurate model is being used. It could be possible to repeat the flow with a more relaxed constraint in terms of clock period, in order to obtain no violations.
All the other meaningful reports have been included in the delivered folder.

CHAPTER 8

Future work

8.1 Introduction

It is still possible to optimize the design over various metrics. In the following, we will discuss alternative and additional design choices with respect to the ones adopted in our architecture. Some of the following proposals have already been mentioned in previous parts of the report, so they will not be explained thoroughly. Their classification is done depending on the metric they aim to optimize.

8.2 OBJECTIVE: reducing the critical path to reach sub 1ns length in post-syn design

In this case multiple solutions are viable, since it is possible to operate at different levels of the design flow. The key problem of all the modifications that target the critical path length is that they inevitably increase the latency. Let's present at least two of this modifications which can be implemented without a big degree of effort, one which targets the synthesis and the other one which involves the RTL description.

8.2.1 Performing synthesis with a library with some CAM cells

The CAM is a very important part of the memory infrastructure of our out-of-order engine. It is crucial for it to be fast since it may be accessed in series with other slow components, such as the ROB when retrieving the result of an uncommitted store instruction whose entry is stored in the CAM. The real problem with the CAM implementation resides in the synthesis, because of the absence of cells to properly implement CAM bits; this results in CAM lines being implemented with cells that performs comparisons, which increases considerably the access times. If an optimized library was available it could be possible to avoid dividing the MEM stage in two substages, thus reducing the latency (in clock cycles) for load instructions. A more refined technology library would probably be helpful also for the other big memories, such as RAT, ROB and RS.

8.2.2 Adding an intermediate stage between commit and RAT update

This choice has already been discussed in section 5.3.5 so, in this section, only its possible implications will be presented.

Adding this stage would obviously complicate the circuitry needed to handle the fact that the RAT update is performed later and this could be responsible for the creation of an unwanted critical

paths. Another important point is that in order to deal with newborn hazards it could be needed to stop incoming instructions in the first decode substage (to wait for RAT updates), so this could potentially increase the latency.

8.2.3 Adding an intermediate stage to divide ROB update and BPU update in commit

This could be done to break the current critical path. As suggested in section 7.1, adding a new stage in this position would not compromise much the performances, because branch predictions are evaluated very rarely. This means that delaying the BPU update of a single clock cycle would not slow down the program flow in a significant way, since the distance between two executions of the same branch is probably much bigger than one clock cycle.

8.3 OBJECTIVE: reducing the latency

In order to reduce the clock cycles latency of instructions we should reduce the number of stages traversed; it is possible to do so by eliminating some of the intermediate substages introduced in DEC or MEM. The main issue which arises from doing so is that the length of the critical path is inevitably going to increase. A possible way to compensate for this increase could be to use a tech library with very optimized memory cells. In fact, these elements are a major problem since most of the critical paths traverse a big memory (i.e. ROB, CAM, etc) at some point. In the following, we are going to present some ways to improve the latency, explaining why some of them cannot be applied at the same time.

8.3.1 Introducing a bypass in write-back

In the current implementation there is no bypass in write-back to make the delivered results immediately available to the instruction in decode or to the ones in the RS. This means that they will have to wait for an additional clock cycle before being able to use that value. This slows down the execution of the instructions which show data dependencies, so it is an addition that could bring an advantage in most programs. As we hinted before, the main problem in implementing a bypass would stem from a possible increase in the critical path. In particular, the critical case could be the one traversing the RS. However, this is not a problem if the third decode substage is implemented because the length of this path should be reduced by a lot.

8.3.2 Removing the third decode substage

Removing the third decode substage would potentially decrease the latency for all instructions with dependencies, similarly to the previous solution. Again, this could result in an increase in the critical path, because we would have to access the RS and the ROB in series. A possible way to reduce the access time could be once again the usage of a cell library optimized for memory elements.

8.3.3 Unifying the memory stage

Removing the division in two substages for the MEM stage could result in a limited improvement, since it would mainly affect load instructions, but it could be considered anyway. In this case too, the problem would reside in the potential increase in the critical path length, due to the serialized CAM and ROB/memory accesses.

8.3.4 Implementing forwarding from EXE and MEM

Implementing a forwarding mechanism would result in a considerable improvement in performances, especially if it is coupled with a write-back bypass. Forwarding could take the values produced during execution by a certain instruction and bring them to the ROB before the actual write-back, thus making it available sooner for other instructions. This would require creating an additional ROB port and handling the possible increase in critical paths.

8.4 OBJECTIVE: optimizing over area

Area optimization mainly relies on substituting hardware components with simpler versions of them, which are usually slower. However, decreasing the area makes the die more compact, thus making the interconnection delays smaller. This also brings a decrease in the consumed power.

8.4.1 Reducing memory sizes

It is possible to reduce the sizes of the main memories which are part of the core, such as ROB and RS. Acting on the RS is more difficult, because the single stations are already small enough and decreasing them further may be responsible for the introduction of many stalls. Decreasing the ROB size is tricky too but it can be done. The current size is 64 entries and takes into account the case in which there is a division in the code. In fact, this operation lasts more than 32 cycles so a ROB of 32 entries would be get full and stall fetch and decode. However, this scenario could not be considered a problem, since divisions may be rare in the programs of interest. Another solution could be to size the memory with a number of entries which is not a power of two. The memories size could also be reduced by optimizing the layout of the word, even though this may be critical for the ROB. In fact, its data words are very long because everything that will be involved in the instruction execution and the produced result need to be stored. A possible improvement in this sense could be done by recycling some fields for multiple uses depending on the instruction. For example, the field that normally stores the instruction result could be used to hold the content to be written in memory for store instructions. However, this solution would complicate the addressing logic.

8.5 OBJECTIVE: optimizing over power

Power optimization relies on reducing static and dynamic power consumption. The reduction of static power could be achieved by introducing different clock domains; this is done using some Design Compiler directives during the synthesis. Dynamic power reduction is more challenging and it could be performed by acting directly on the desig; we will discuss some techniques to implement it effectively.

8.5.1 Introducing guarded evaluation and clock gating

Guarded evaluation is a technique which makes use of latches to hold the previous input values fed to a unit in case the unit is not used in the current clock cycle. In this way, useless transitions are avoided, thus reducing the overall switching activity of the circuit. The technique can be implemented very easily by controlling the latch enable of the input registers of a unit with the enable signals of that unit. These latches can be described behaviorally as in the following assignment:

```
1 signal_out <= signal_in when enable = '1';
```

Clock gating consists in using as clock signal the AND function between the clock signal and an enable signal, so that the whole sequential element is disabled in case the enable is low. This kind of optimization can be enabled very easily during synthesis, so it could be an interesting addition. It could also result in an increase in area due to the added AND gates and latches.

8.5.2 Reducing memory sizes

This solution is the only one which could potentially be beneficial for every kind of metric, but we should take into account how it could affect the latency when some long latency instructions are executed, as we discussed before.

8.5.3 Reducing the degree of parallelism in the multiplier

Another possibility to save power and possibly latency could be the one to merge adjacent pipeline stages in the Booth multiplier, thus reducing the latency of the overall circuit and the degree of pipelining. We should also pay attention not to increase too much the length of the critical path.