Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering
Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group 9
Nicole Dai Prà, Leonardo Izzi

October 19, 2020

Contents

CHAPTER 1

Introduction

1.1 General architecture

The proposed DLX is a 5-stage, MIPS-based, scalar processor. Its implementation respects the MIPS ABI and O32 calling convention, it supports all the basic instructions, most of the ones proposed as PRO and a few 32-bits ones coming from the MIPS64 architecture [3]. The DLX also features a 32 entries *BTB*, a 64 entries *four-way associative data cache* and a *forwarding unit*. To support the ABI and the extended instruction set we had to modify the file `dlxasm.pl`. By going a little bit more in details, here there is a summary of what each stage does:

1. **IF stage**: here is where the *program counter*, also referred to as PC, is stored. This stage contains the logic to update the value of the next PC and to choose the right PC at each clock cycle.
2. **ID stage**: in this stage the instruction fetched with the PC calculated the cycle before is decoded and the register file is accessed for reading. In case an instruction has an immediate field it is extracted and properly extended.
3. **EXE stage**: based on the decoding happened the cycle before, one of the ALU's functional units is activated to perform the requested calculation. It is worth noting that the ALU is capable of executing an instruction like `jalr` in a single clock cycle instead of the 2 specified in the MIPS architecture manual [3]. Another exception is the `mult` instruction which works with the *integer register file* (also referred to as RF) and it is a multi-cycle operation.
4. **MEM stage**: if the operation in this stage is a load or a store the cache is accessed, and in case of a load shorter than 32 bits sign extension is performed (if required). On the other hand, if an instruction does not fall in the two categories outlined before, its result is simply propagated to the next stage.
5. **WB stage**: in this stage all the operations that produce a result to be stored in the RF can do so. The result to be written is chosen among the one coming from the cache and the one coming from the ALU.

The datapath is managed by the *control unit*, which has been implemented as a mix of two approaches: the hardwired and the FSM. The reasoning behind this choice is explained in chapter 8. Within the control unit there is also a stall unit, whose job is to detect any possible hazard, and to either enable data forwarding or to force a bubble in the pipeline for stalling the processor.

Outside the control unit and the datapath there is the *memory controller*, which handles the communication between the RAM and the cache as well as cache misses.

1.2 Supported ISA

In table 1.1 are shown all the supported instructions:

Name	Opcode	Func	Type	Name	Opcode	Func	Type
nop	0x00	0x00	R	sll	0x00	0x00	R
srl	0x00	0x06	R	sra	0x00	0x07	R
jr	0x00	0x08	R	jalr	0x00	0x09	R
mult	0x00	0x0E	R	mfhi	0x00	0x10	R
mflo	0x00	0x12	R	add	0x00	0x20	R
addu	0x00	0x21	R	sub	0x00	0x22	R
subu	0x00	0x23	R	and	0x00	0x24	R
or	0x00	0x25	R	xor	0x00	0x26	R
seq	0x00	0x28	R	sne	0x00	0x29	R
slt	0x00	0x2A	R	sgt	0x00	0x2B	R
sle	0x00	0x2C	R	sge	0x00	0x2D	R
sltu	0x00	0x3A	R	sgtu	0x00	0x3B	R
sleu	0x00	0x3C	R	sgeu	0x00	0x3D	R
bgez	0x01	0x00	I	bltz	0x01	0x00	I
j	0x02	0x00	J	jal	0x03	0x00	J
beq	0x04	0x00	I	bne	0x05	0x00	I
blez	0x06	0x00	I	bgtz	0x07	0x00	I
addi	0x08	0x00	I	addui	0x09	0x00	I
subi	0x0A	0x00	I	subui	0x0B	0x00	I
andi	0x0C	0x00	I	ori	0x0D	0x00	I
xori	0x0E	0x00	I	beqz	0x10	0x00	I
bnez	0x11	0x00	I	slli	0x14	0x00	I
srli	0x16	0x00	I	srai	0x17	0x00	I
seqi	0x18	0x00	I	snei	0x19	0x00	I
slti	0x1A	0x00	I	sgti	0x1B	0x00	I
slei	0x1C	0x00	I	sgei	0x1D	0x00	I
lb	0x20	0x00	I	lh	0x21	0x00	I
lw	0x23	0x00	I	lbu	0x24	0x00	I
lhu	0x25	0x00	I	sb	0x28	0x00	I
sh	0x29	0x00	I	sw	0x2B	0x00	I
sltui	0x3A	0x00	I	sgtui	0x3B	0x00	I
sleui	0x3C	0x00	I	sgeui	0x3D	0x00	I

Table 1.1: Supported instructions

It is worth pointing out that `nop` and `sll` have exactly the same opcode and func fields, because the `nop` actually corresponds to `sll $zero, $zero, $zero` ([3]), which is an instruction that achieves nothing. Also `bltz` and `bgez` share the same opcode (as stated in [3] for release 1 up to 5), as their difference lies in the instruction's bits 20-16: they are set to `0b00000` for the former one and to `0b00001` for the latter one. Finally, `jr` and `jalr` are considered R instructions as specified in the manual [3].

1.3 Design objectives

Our goal during the design phase was to deliver a fast processor and to cut latencies of the most used general purpose instructions. Our frequency target was to reach 1 GHz, and to be able to execute `jal` and `jalr` in a single clock cycle instead of the 2 specified in [3]. We also felt that was important to provide to a potential user a complete instruction set, therefore we have introduced also `mult`, which is normally executed by the FPU. As discussed in chapter 5, achieving high clock frequencies with a multiplication unit is not trivial and some compromises had to be done in term of latencies for this particular operation. We decided to trade the presence of the operation with latency, because as stated in [2] multiplications accounts only for the 0.02% of the utilized instructions.

1.4 Folder organization

In the root folder it is possible to find 3 folders and this file. The folders are:

1. **dlx_sim**: all the file needed for the ModelSim simulation, as well as the files we used to test the DLX, can be found here.
2. **dlx_syn**: the synthetizable files are stored here along with the synthesis report.
3. **dlx_impl**: it contains all the files and the outputs of the physical design.

The VHDL files in dlx_syn are a subset of the ones present in dlx_sim. Moreover, in dlx_sim there are two additional folders:

1. **tb**, that contains the VHDL testbenches needed to perform the actual simulation on ModelSim.
2. **test_assembly**, where the `.asm` and `.mem` files are saved. These files are read by their related testbench and are used to fed the *IROM*.

CHAPTER 2

Assembler

To support the expanded instruction set and to support the MIPS ABI, we had to modify the provided assembler, `dlxasm.pl`.

2.1 Instruction classes

First, we had to add new class of instructions, because there were not existing suitable classes for our needs. These classes are:

1. `r3`: it includes operations like `mflo` and `mfhi`, which are R instructions with both `rs` and `rt` always set to 0.
2. `r4`: it allows to the `mult` instruction to be executed as an integer, R, instruction. Notably, since it writes its result inside the register `lo` and `hi`, its `rd` register is set to 0.
3. `b1`: it is used for `bgez` and `bltz`, because they have the same opcode but a different value is stored in the bits 16-20. For more informations refer to section 8.1
4. `b2`: it is used to properly encode branches that uses registers instead of $PC + immediate$ as next PC value.

2.2 Instructions

The full list of added and/or modified instructions is provided in table 2.1.

2.3 Register names

To provide full ABI support, we have added support for the actual registers' name. The complete list is provided in table 4.1.

Name	Class	Opcode/Func	Original opcode/func	Added/Modified
sll	r	0x00	0x04	M
mflo	r3	0x12	-	A
mfhi	r3	0x10	-	A
mult	r4	0x0E	0x0E	M
bgez	b1	0x01	-	A
bltz	b1	0x00	-	A
beq	b2	0x04	-	A
bne	b2	0x05	-	A
blez	b	0x06	-	A
bgtz	b	0x07	-	A
beqz	b	0x10	0x04	M
bnez	b	0x11	0x05	M
jr	jr	0x08	0x12	M
jalr	jr	0x09	0x13	M
nop	n	0x00	0x15	M

Table 2.1: Instructions added or modified in the dlxasm.pl file

CHAPTER 3

IF stage

In this chapter we will show how the IF stage is structured and how each block works. In figure 3.1 the block diagram of the stage is shown.

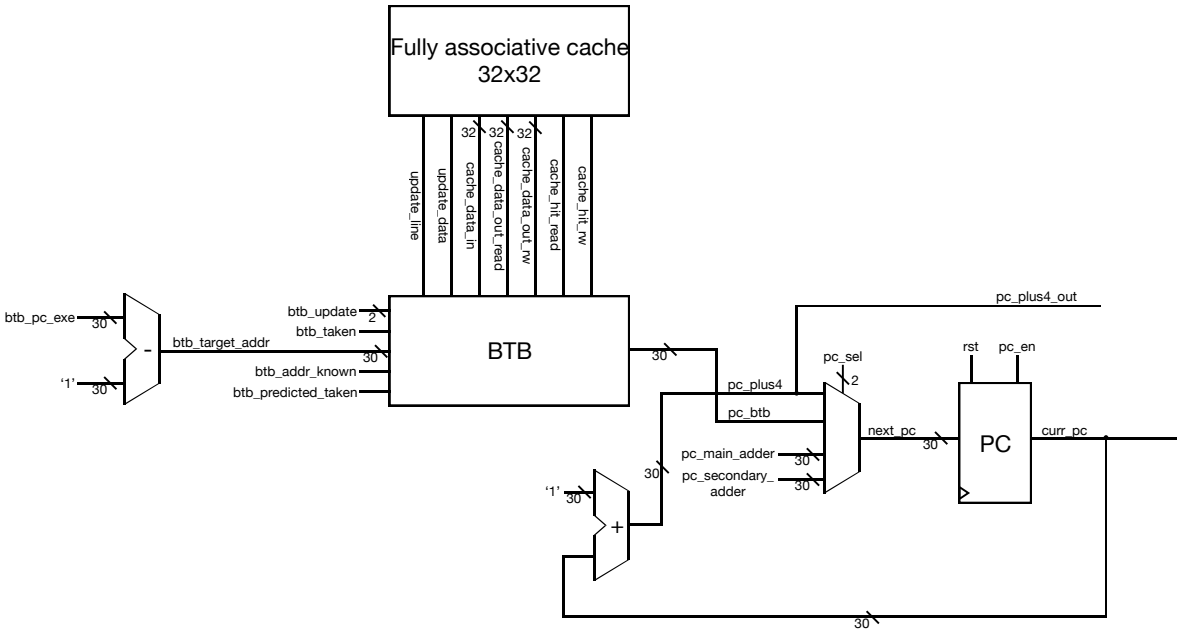

Figure 3.1: Block diagram of the IF stage

There are:

1. a register storing the value of the current PC.
2. A mux that selects among its 4 inputs which will be the next PC value.
3. An adder that calculates *current pc* + 4
4. A *Branch Target Buffer* (BTB) able to recognize and predict the behavior of known branches and jumps
5. A 32-entries, 32 bits, fully associative cache that it is used by the BTB to recognize known branches and jumps

6. A subtractor calculating *btb pc exe* − 4, that is the PC of the instruction in the EXE stage. It is used when updating the BTB content.

In this stage the PC is used on 30 bits instead of 32 because instructions are aligned on 32 bits boundaries, therefore the 2 LSBs are always 0 and it was useless to add more HW to handle them.

3.1 Branch Target Buffer

The *branch target buffer* is a hardware structure used to known in the IF stage if the instruction being fetched is a known branch or jump. It does so by accessing in read mode its cache using as address the current pc (this connection is not shown in figure 3.1). For the differences between read mode and write mode of the cache refer to subsection 3.1.1. Our BTB operates over 3 values, as shown in figure 3.2.

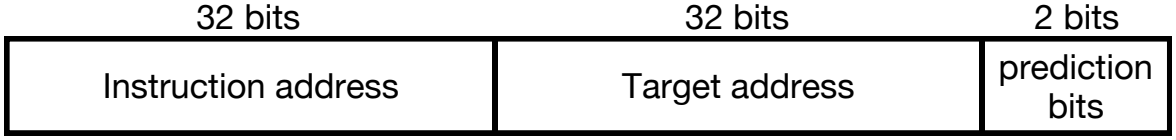

Figure 3.2: Branch target buffer values

1. The *instruction address* is the memory address where the instruction resides. In our implementation corresponds to the TAG value in the cache.
2. The *target address* is where the PC branch or the jump would go in case the jump condition is verified.
3. The *prediction bits* are used by the FSM inside the BTB to predict whether a branch will be taken or not. The state machine of the prediction bits is shown in figure 3.3.

The target address, concatenated with the prediction bits, form a 32 bit value which corresponds to the data stored inside the cache.
Now let's take a closer look to the FSM implementation.

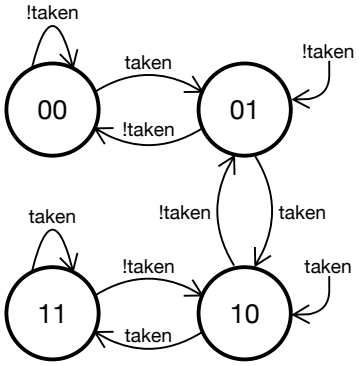

Figure 3.3: Branch target buffer FSM representation

There are two entry points in the FSM: one that enters the state `01` and one that enters the state `10`. The former is used to add a new branch as non-taken, while the latter is used for the opposite. In the delivered implementation only the *taken* entry point is used, but since at the beginning of the

development we did not know if we would have added also non-taken branch to the BTB, we felt that this degree of flexibility would have been nice to have.

Besides this particular, the FSM is straightforward: the 2 bits are used to store the state of a branch, and their values have the following meanings:

1. `00`: strongly not taken.
2. `01`: weakly not taken.
3. `10`: weakly taken.
4. `11`: strongly taken.

The value of `btb_taken` is 0 for the first two cases and 1 for the last two, provided that the address that is being considered is known to the cache (that is, we have $hit = 1$ in read mode).

The BTB of course needs to be updated every time a branch is executed or discovered. This is achieved by writing in `btb_update 01` to update only the history bits or `10` to add a new branch, by concatenating the target address to the history bits and by setting the cache's write address to the instruction's PC.

3.1.1 Fully associative cache

A key component of the BTB is the cache, where as already said all the known branches and jumps are stored. We decided to use a 32-entries fully associative cache to have the highest possible hit rate, since every miss could hurt the pipeline's performance. Its general, simplified structure is shown in figure 3.4.

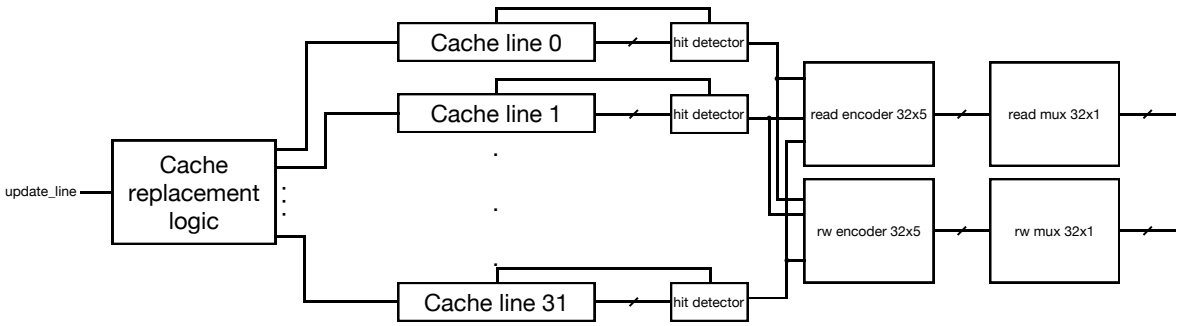

Figure 3.4: Cache general representation

This cache support two read operations and a write operation per clock cycle and implements a FIFO replacement logic. Each cache line has a hit detector, which is used to detects a hit in case of a read, and to issue a line update (only the data is updated, not the tag) if requested by the BTB. The hits signals then enter two 32-entries encoders which in turn drive two 32-entries muxes, used to output the data from the cache. To illustrate exactly how a cache line and a hit detector work we'll refer to picture 3.5.

A cache line is composed of a 30 bits `tag` field, that stores the PC of the branch/jump, a validity bit used to discriminate invalid data after a reset and a 32 bits `data` field that keeps the value of the target address along with the history bits. The tag enter two equality comparators to check if a match exists either with the read address, the write address, or both. To determine if a hit has occurred this is not sufficient though, because the data stored in the tag may be invalid. To solve this issue an AND is executed between the validity bit and each comparator's output. The write part has an additional AND between its hit value and the `update_data` signal to determine if this line have to update its `data` field.

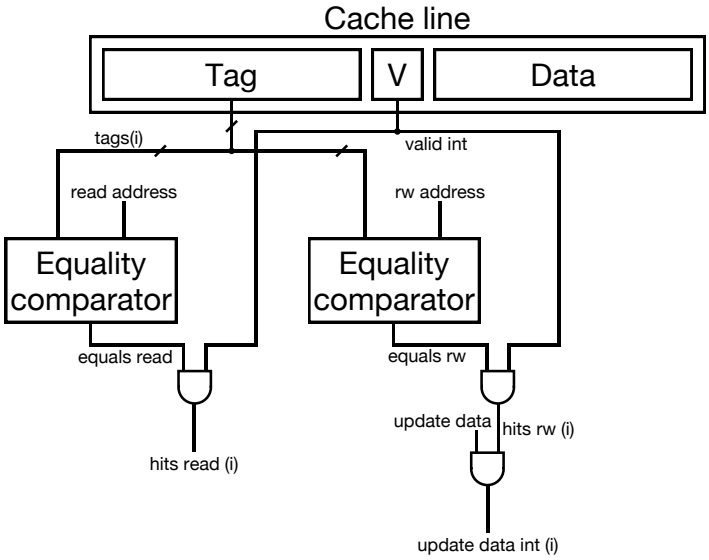

Figure 3.5: Cache line and hit detectors

3.2 Program counter selection

There are 4 possible next program counter values to choose at each clock cycle:

1. $PC + 4$
2. PC_{BTB}
3. $PC_{main\ adder}$
4. $PC_{secondary\ adder}$

$PC + 4$ is the default choice when the BTB doesn't recognize the address and the EXE have not executed a branch or a jump. PC_{BTB} is taken whenever the BTB recognizes the address and the EXE have not executed any branch or jump. $PC_{main\ adder}$ is used when the EXE have executed a `jr` or `jalr`. Finally, $PC_{secondary\ adder}$ is the value used when any branch or jump except `jr` and `jalr` is executed in the EXE stage.

CHAPTER 4

ID stage

In the ID stage instructions are decoded and the register file is accessed. Its block diagram is shown in picture 4.1

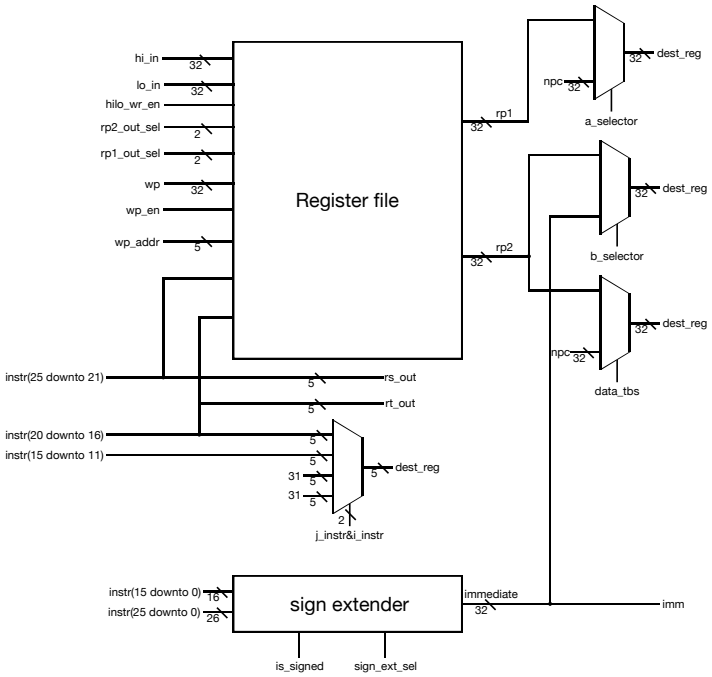

Figure 4.1: Block diagram of the ID stage

4.1 Register file

The *register file* is one of the two main components in this stage. It has 2 read ports and 3 write ports, although a write operation never use all of them at the same time. It contains 32 *general purpose registers* (GPR), as stated by the ABI, and 2 additional registers called `lo` and `hi`, accessible only by special instructions and used to store multiplications' results. Among the GPR there is a special register, the register number 0, that cannot be written and always contains the value `0x00000000`. The full list of the ABI-compliant registers is shown in table 4.1:

In the ID stage the RF is accessed in read, since it is the WB stage to access it in write. To select

Name	Number	Use	Directly accessible
`$zero`	0	constant 0	Yes
`$at`	1	assembler temporary	Yes
`$v0-$v1`	2-3	values for function returns and expression evaluation	Yes
`$a0-$a3`	4-7	function arguments	Yes
`$t0-$t7`	8-15	temporaries	Yes
`$s0-$s7`	16-23	saved temporaries	Yes
`$t8-$t9`	24-25	temporaries	Yes
`$k0-$k1`	26-27	reserved for OS kernel	Yes
`$gp`	28	global pointer	Yes
`$sp`	29	stack pointer	Yes
`fp`	30	frame pointer	Yes
`$ra`	31	return address	Yes
`$lo`	32	stores the lower 32 bits of a multiplication	No
`$hi`	33	stores the higher 32 bits of a multiplication	No

Table 4.1: DLX registers. Taken from [4]

the registers the bits 25-21 and 20-11 are used, because they represent the `rs` and `rt` fields in an R instruction. These values are not the only ones used by the RF to decide which values its read ports should take: as illustrated in table 4.1, there are two non-directly accessible registers, `lo` and `hi`. To perform the final output selection two control signals are used, `rp1_out_sel` and `rp2_out_sel`, that drive 2 internal muxes that choose the value of the read ports as follows:

1. `00`: output the selected GP register.
2. `01`: output the `lo` register.
3. `10`: output the `hi` register.
4. `11`: output `0x00000000`.

4.2 Sign extender

This component is used by I and J instructions to extend their immediate fields on 32 bits. In fact, I instructions have their immediate stored in the lower 16 bits of the instruction, while the J have it stored on the lower 26 bits. Moreover, these bits could represent a signed or unsigned value, like in the case of a `addi` and a `addui`, therefore this has to be taken in account during the extension. As shown in figure 4.1, two controls signals enter the sign extender: `is_signed` and `sign_ext_sel`. The former is used to extend either with 0s or taking into account the sign, while the latter is used to decide if the extension should consider only 16 bits or 26.

4.3 Selection muxes

In the rightmost part of figure 4.1 3 muxes 2x1 can be seen. In a standard MIPS pipeline these would reside in the EXE stage, but since it is the slowest stage of the pipeline we decided to shorten its critical path by bringing them in the ID stage. The first mux is used to select between the read port 1 of the RF and the program counter coming from the stage before, while the second mux chooses among the RF's read port 2 and the immediate value coming from the sign extender. The output of these two muxes form respectively the `a` and `b` inputs of the EXE stage. The third mux is used to

select the output of the RF's read port 2, which is the data that would go inside the memory when executing a store, and the npc, that is used to execute in a single cycle a `jalr` instruction.

CHAPTER 5

EXE stage

In this stage it is performed the actual instructions' execution and the load/store memory address calculation. Its general block diagram is shown in figure 5.1.

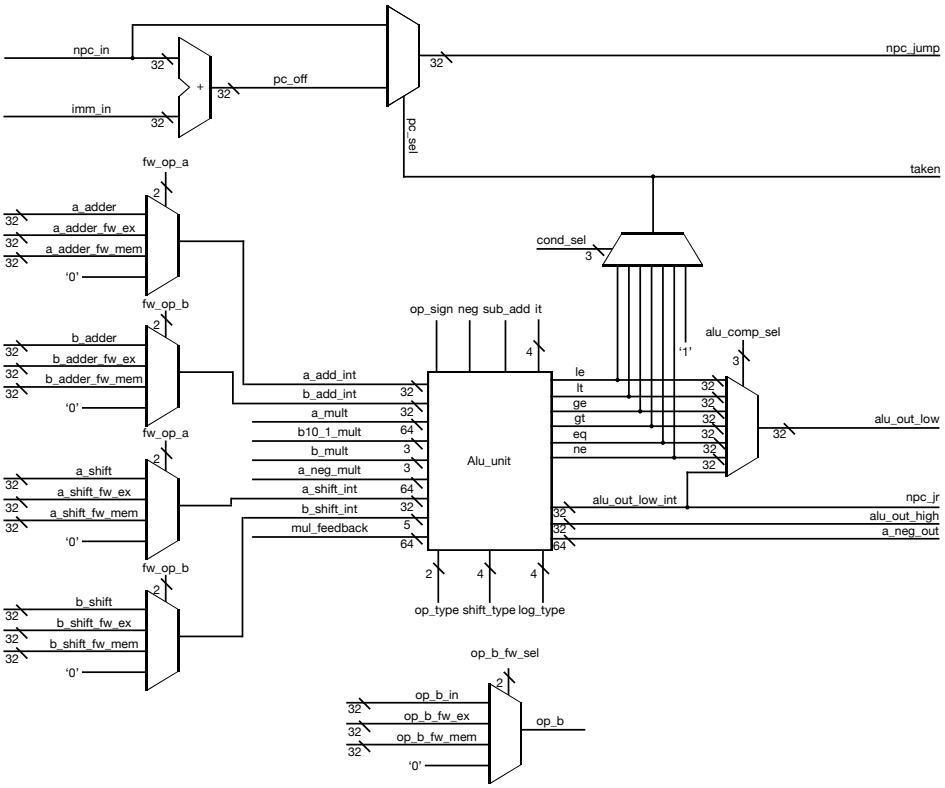

Figure 5.1: Block diagram of the EXE stage

The ALU incorporates the actual functional units, that are:

1. an adder
2. a shifter
3. a logical unit
4. a multiplier

Each of these units have its own inputs coming from separate ID/EXE registers, the only exception being the adder and the logical units. The rationale behind this is to reduce as much as possible the switching activity. These are, in fact, the most power hungry devices in the whole datapath, therefore anything that can be done to reduce their power consumption leads to great benefits.

As it can be observed in the leftmost part of figure 5.1, the adder (and hence the logical), as well as the shifter, support forwarding. Forwarding is also supported for the `op_b` signal, that in case of a store contains the data to be written in cache.

At the center there is the ALU, discussed in details in section 5.1, and two muxes: the one that goes toward the top of the page is used to send the `taken` signal to the control unit in case of a branch execution and to drive the topmost mux, which outputs the correct value of the program counter. The rightmost mux is, instead, used to select either the ALU's output signal or one of the internal comparator.

Finally, in the topmost part of figure 5.1 there is a second adder, used to calculate $PC_{i+1} = PC_i + immediate$, where i indicates the current cycle number.

5.1 ALU

The ALU encapsulate, as said before, the various components that perform the wide range of supported operations. Its block diagram is shown in figure 5.2.

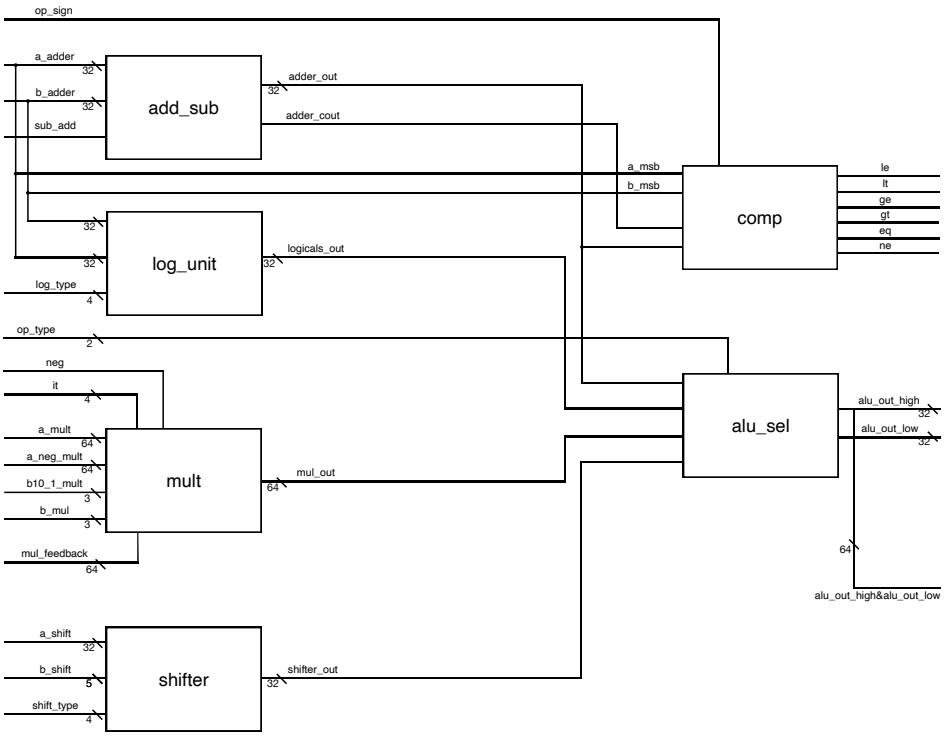

Figure 5.2: Block diagram of the ALU

The adder, the logical unit and the shifter all implements structures seen during the lecture, in particular:

1. the adder is an implementation of the P4.
2. the logical unit implements the one seen inside the T2.

3. the shifter is a smaller version than the one implemented in the T2.

For this reasons, for the various images and explanation of these units refer to [1].

5.1.1 Multiplier

The multiplication's algorithm is based on the Booth's algorithm explained during the lecture. However, we could not directly use the version developed for the laboratories. Indeed, that was a fully combinational circuit, which delivered its result after 5 *ns*. It was too slow, big and power hungry. Hence, we modified the original structure in such a way that it was pipelined and it used only a fraction of the components used in the original version. Its schematic is shown in figure 5.3

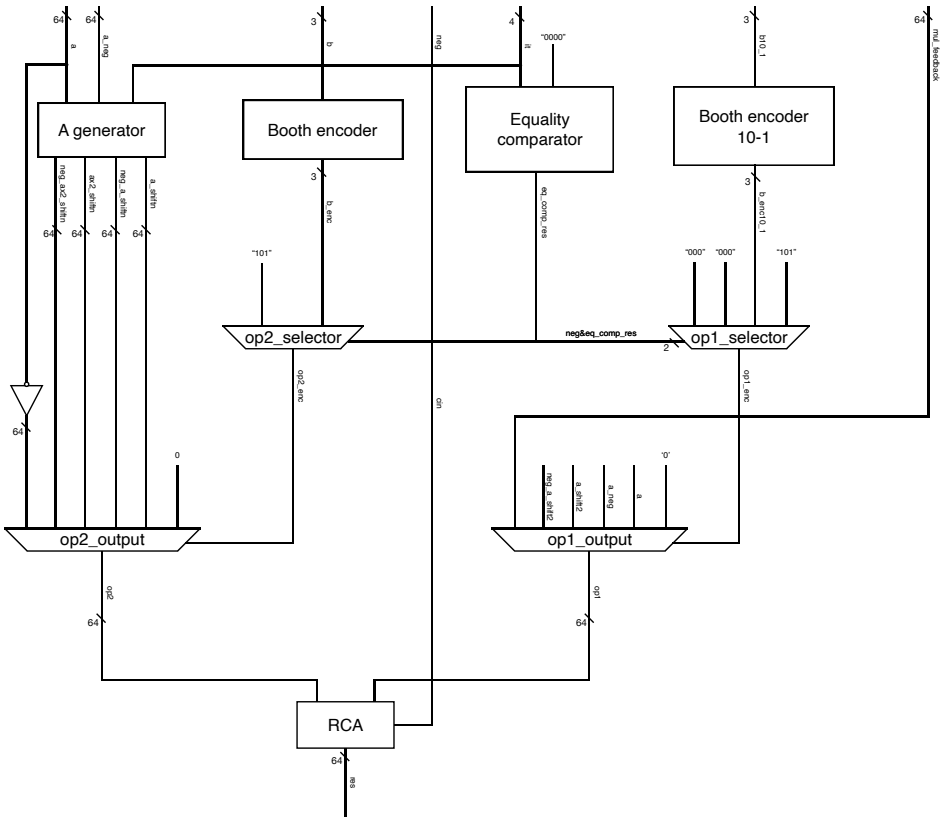

Figure 5.3: Multiplier schematic

Overall it is composed by:

1. a single 64-bits wide adder
2. two 6 inputs muxes 64 bits wide
3. one 4 inputs mux 3 bits wide
4. one 2 inputs mux 3 bits wide
5. an equality comparator
6. two Booth's encoder
7. a component called `a_generator`

It is clear that the area occupied is much smaller than the standard combinational version or even the pipelined one. This results also in huge power savings, since there are way less components that contributes to P_{leak}.

Our algorithm works in this way:

1. **Cycle 0**: The value of `a` is extended on 64 bits and negated by the 64 bits adder. Then it is sampled by the corresponding ID/EXE register.
2. **Cycle 1**: Both `booth_encoder` and `booth_encoder10_1` are used to calculate the first partial result.
3. **Cycle 2-15**: From `op1_output` is always chosen the `mul_feedback` signal, while from `op2_output` is selected one of the shifted values of `a` or its negation based on the output of `booth_encoder`.
4. **Cycle 16**: The final result is sampled and sent to the MEM stage.

Let's see in details what these components do and how they are implemented.

5.1.1.1 a generator

The `a_generator` takes in input the `a` operand, both affirmed and negated, on 64 bits. Then, based on the iteration number (that is, how many time a partial has been calculated), it outputs the value `A`, `-A`, `2A` and `-2A` correctly shifted. The name `A` refers to the one used in [1] to indicates a shifted version of `a`.

5.1.1.2 Booth encoder

There are two booth encoders: one called `booth_encoder` and one called `booth_encoder10_1`. The former always takes 3 bits from the LSBs of a shift register that contains the `b` operand, while the latter takes always in input the bits 1, 0 and -1 of `b`. This is used only for the calculation of the first partial, then its output is ignored.

For the truth table of the encoder please refer to [1].

5.1.1.3 Equality comparator

It is used to discriminate when `it = 0b0000`, because in this case the negation of `a` is performed. Hence, the adder's input muxes have to output the values to perform $0 - a$.

5.1.1.4 Multiplier's limitations

As it will be explained also in the following chapters, this multiplier presents some limitations. One is the number of cycles to output the result, because the pipeline has to be stalled. It is worth nothing that a superscalar processor would have alleviated this problem.

Another one is its inability to support forwarding: its operands come from the ID/EXE stage, that has to store them on 64 bits. Our forwarding logic, however, is able to forward only 32 bits at a time, therefore neither its output or its input can receive data from subsequent stages. This results in pipeline stalls to allow the fetch and the store of the correct values in the register file.

5.1.2 Comparator

This unit takes in input the adder's output and compare it in such a way to detect the following conditions:

$$a < b \quad a \leq b \quad a = b \quad a \neq b \quad a \geq b \quad a > b$$

One of these conditions is used to drive the secondary adder's mux and may be used (after being extended on 32 bits) by instructions like `seq` or `sleui`.

The logic that calculates these values is pretty straightforward, as this extract from the file `comparator.vhd` shows:

```
res_nor <= not or_reduce(res);
cout_not <= not cout;
lt_signed <= (a_msb and (not b_msb)) and op_sign;
gt_signed <= ((not a_msb) and b_msb) and op_sign;

eq <= res_nor;
ne <= not res_nor;
le <= (lt_signed or res_nor or ((not res_nor) and cout_not)) and (not gt_signed);
lt <= (lt_signed or ((not lt_signed) and (cout_not))) and (not gt_signed);
ge <= (cout or gt_signed) and (not lt_signed);
gt <= (gt_signed or (cout and (not res_nor))) and (not lt_signed);
```

5.2 Branch & Jump

The EXE stage is capable of executing any branch or jump in a single clock cycle, thanks to the secondary adder shown in figure 5.1. Jumps are treated as always taken branches: this allows to exploit the BTB, leading to a performance increase since there is no need to flush the pipeline after the first time a jump is encountered.

Branches always calculate the next pc as $PC_{i+1} = PC_i + immediate$, while jumps may also appear in the form $PC_{i+1} = GPR[rs]$, where i indicates the cycle number, GPR the general purpose register file and rs the index of the source register. Moreover, jumps like `jal` and `jalr`

2 tipi: pc + offset e pc = registro

CHAPTER 6

MEM stage

In this stage memory is read during a load and it is written during a store. Its block scheme is shown in figure 6.1.

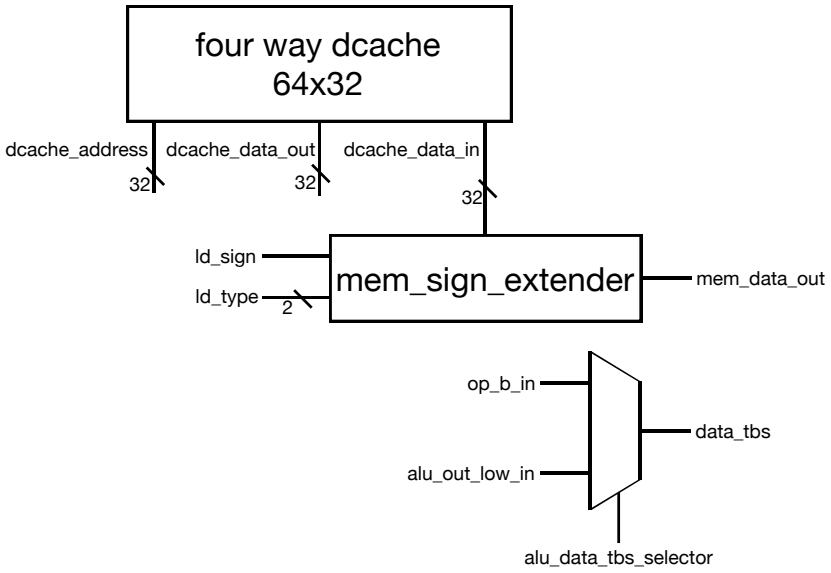

Figure 6.1: Block diagram of the MEM stage

The lowest mux is used when in the MEM stage when an operation different from a load and a store is executed. It selects `alu_out_low_in` when in the RF the data coming from the ALU must be stored, otherwise it selects `op_b_in`, used by `jalr` and `jal` instruction to save in the RF the value of the program counter.

6.1 Data cache

The main datapath component of this stage is the data cache. It is a 64-entries four-way associative cache, able to perform two reads, two writes, or one read and one write per clock cycle. This is due to the fact that the cache has two interfaces, one connected to the processor and another one connected to the memory controller. In figure 6.2 it is shown the block diagram for the processor's side.

As it can be seen, the cache's lines are divided among four blocks. Each line in a block corresponds to a different set, therefore a set is formed by considering all the lines having the same number. The

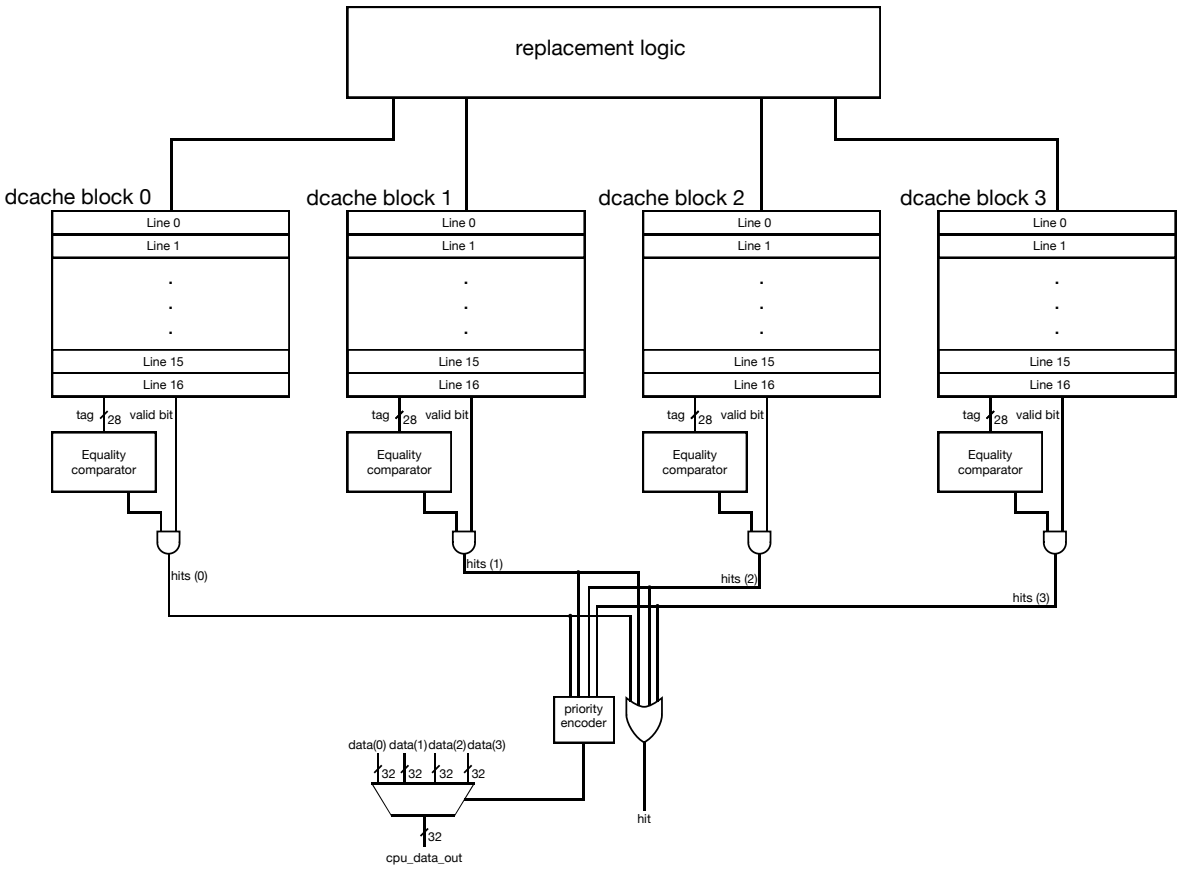

Figure 6.2: Dcache's processor side

structure of the line is the same as the one shown in figure 3.5, but with a difference: if the data being stored comes from the CPU it is saved in big endian. In fact, the DLX is expected to work in big endian, therefore every time data comes out the CPU it must do so in this format. To reuse some modules we already had, however, we preferred to internally work in little endian. We've identified the cache as the best place to perform DLX-RAM data conversion transparently.

Moving on, each block has its own logic to detect if the line corresponding to the chosen set is generating a hit. The detection logic works in the same way as in the fully associative cache explained in section 3.1.1 and will not be discussed again. The four hits signal generated by the blocks then enter a priority encoder, which generate a suitable encoding for the data mux. Moreover, these signals converge into a 4-inputs OR gate that outputs the final `hit` signal.

6.1.1 Replacement logic

On top of the cache's blocks there is the replacement logic, shown in figure 6.3.

This unit receives in feedback the `hits` signal generated by the four blocks and the encoding value for the data mux, moreover receives either from the memory controller or the control unit an update signals. These information are combined together to decide how and if to update or replace the data contained in the cache. When an update is requested for data not present yet in the cache and there are free blocks, a FIFO scheme is used and both `replacement` and `dirty_bit` are low. On the other hand, when an update is requested for data already known to the cache, `ram_out_enc` forwards the value of the input `mux_enc` and it keeps `replacement` low. However, if new data must be written and there's no room in the blocks, an eviction must be performed. In this case the signal `replacement` is

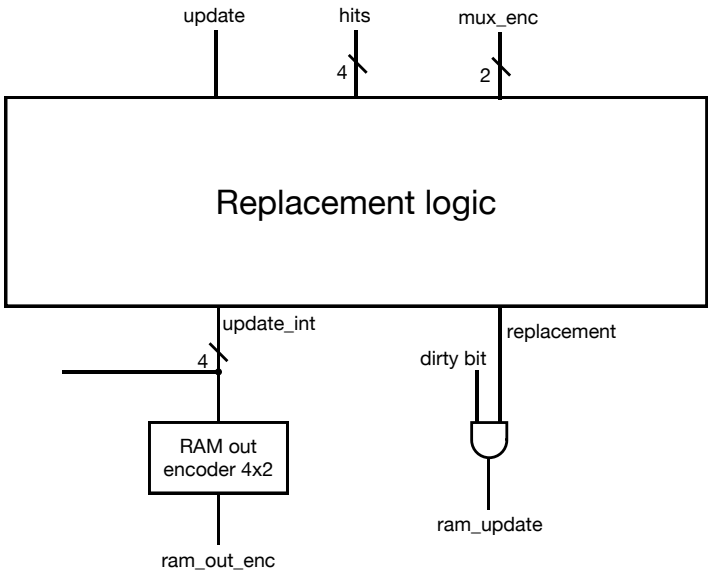

Figure 6.3: Replacement logic

raised to 1. Also the `dirty_bit` is set to 1 since we are considering the case in which all the blocks are used. Hence, the AND goes to 1 and the both the control unit and the memory controller receive it. For the explanation of how cache eviction and cache misses are handled please refer to section 8.4.

6.2 Sign extender

The DLX, as seen in table 1.1, supports many types of load operation: with or without sign extension, and on different sizes of data. The job of the sign extender is therefore the one of taking in input the data read from the cache and to extend it properly (if necessary). It is driven by the control unit through the `ld_sign` anf `ld_type` signals. The former is set to 0 when load is unsigned, to 1 when it is signed. The latter is a 2-bit signals that specifies from where 0-extension or sign-extension should take place. The possible values are:

1. `00`: use 32 bits. In this case no operation is performed on the incoming data.
2. `01`:use 16 bits.
3. `10`: use 8 bits.
4. `11`: reserved. In this particular implementation it works in the same way as `00`.

CHAPTER 7

WB stage

This is the last stage of the pipeline. It is conceptually part of the ID stage since it drives the write signals of the register file. It is composed only by a 2x1 mux, which decides the value of the `wp` port of the register file shown in figure 4.1. The choice is made among the data read from the cache in the MEM stage or the one coming from a calculation performed in the EXE stage. It also drives:

1. `wp_en`, set to 1 when an operation has to store data in a GP register.
2. `hilo_wr_en`, set to 1 when the result of a multiplication has to be stored in `lo` and `hi`.
3. `wp_addr`, which is the value of the destination register.

CHAPTER 8

Control Unit

The DLX's control unit is mainly based on the hardwired approach, although some operations are handled by in-stage FSMs. The former approach has been chosen because most of the control signals of an operation never changes during its execution. For example, for an `add` instruction the value of `rp1_out_sel` is always `00` and will never change, no matter what happens in the pipeline. Because of this, reading the values from a control register and writing them directly in the component is the fastest and cheapest way to control the datapath. However, not all the signals can be known a-priori, or they need to change over time to handle complex operations: the former case is represented by the forwarding signals for example, while the latter from the multiplication. Due to this we had to resort to in-stage FSM able to correctly manage these situations.

The control unit is by far the most complex component in the DLX, therefore to ease the explanation the rest of the chapter is divided in sections, each of which explains what happens in the related stage.

8.1 IF stage

The PC register in the datapath is used to access the *instruction ROM*, or IROM, where all the instructions to be executed are stored. The IROM's output is then sent to both the control unit, to analyze the instruction's *opcode* and *func* fields, and to the IF/ID registers for decoding the rest of the fields in the ID stage. The control units hosts two ROMs: one contains all the control words (also referred to as *CW*) related the I and J instructions, while the other one stores all the CW belonging to the R instructions. The former is accessed using the opcode field, the latter using the func field. This distinction had to be done because all R instructions have the same opcode, which is equal to `0x00`.

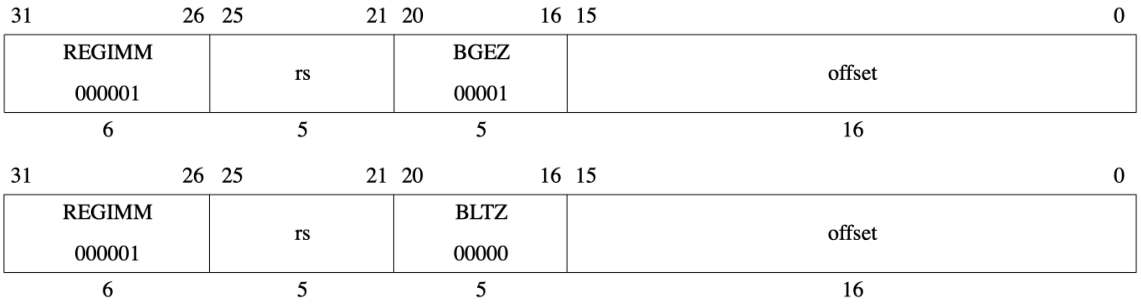

Figure 8.1: bltz and bgez instruction format. Taken from [3]

The control unit recognize this pattern and it chooses the right CW for each instruction. Nonetheless this is not the only particular case. As already shown in table 1.1, `bgez` and `bltz` shares the same opcode. Their difference reside in the bits 20-16, which are set to `00001` for the former, to `00000` for the latter (figure 8.1). This information cannot be used to differentiate the branches directly in the CW's ROM, hence their final CW is generated dynamically.

This part of the control unit also decide from which source must come the next value of the PC register. Four cases have to be distinguished:

1. The EXE stage have not executed a branch and the BTB predicts as not taken the current fetched instruction: in this case $PC + 4$ is chosen.
2. The EXE stage have not executed a branch and the BTB predicts as taken the current fetched instruction: in this case PC_{BTB} is chosen.
3. The EXE has executed a `jr` or a `jalr`: the prediction of the BTB is discarded and $PC_{main\ adder}$ is chosen.
4. the EXE has executed a branch or a jump different from `jr` or `jalr`: the prediction of the BTB is discarded and $PC_{secondary\ adder}$ is chosen.

8.2 ID stage

The bits used to drive the ID stage are all static, therefore they are directly read from the control unit's IF/ID register and are sent to the datapath. Here is it worth commenting how this stage interacts with the EXE stage. When a multiplication is in progress in the EXE stage it is the EXE stage's FSM to drive the ID/EXE registers, therefore all the ID's signals going to the ID/EXE registers are set in high impedance to avoid conflicts.

The ID stage is also in charge of detecting a `mult` instruction, and when it does so it may need to stall the pipeline. Unfortunately, when the multiplication's operand are sent to the ID/EXE stage, they are automatically extended on 64 bits by the registers themselves. This means that a `mult` cannot enters the EXE stage if one of its source operands is still being processed in the pipeline. Due to this limitation, when an hazard is detected for a multiplication is detected, the control unit's ID stage in conjunction with the *stall unit* (explained in section 8.6) stalls the IF and the ID stage, until all the data hazards have been resolved.

8.3 EXE stage

This is the biggest stage in the control unit's pipeline. It contains an FSM to handle the multiplication, which is the only instruction that requires more than a single clock cycle to execute. Its state diagram is shown in figure 8.2.

8.3.1 NORMAL_OP_EXE

This is the reset state and also the one where all the operations but `mul` are executed. Here few checks are executed, therefore to ease the reader's understanding we have provided a flow diagram, shown in figure 8.3.

The first check performed is whether the operation currently in the EXE stage is a multiplication or not. In the former case the EXE stage switch state as shown in figure 8.3 and stalls the pipeline, while in the latter case the state remains unchanged. If the operation is not a multiplication, the other notable case is to check if the instruction is a branch or jump. To do so, it is controlled if

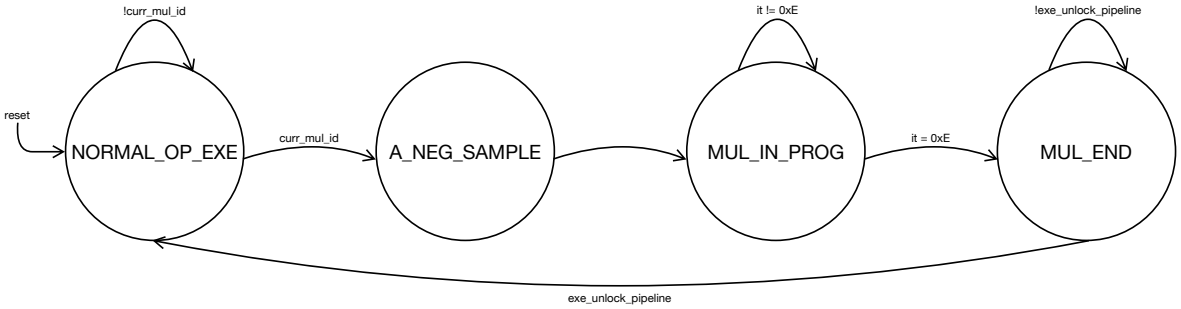

Figure 8.2: EXE stage state diagram

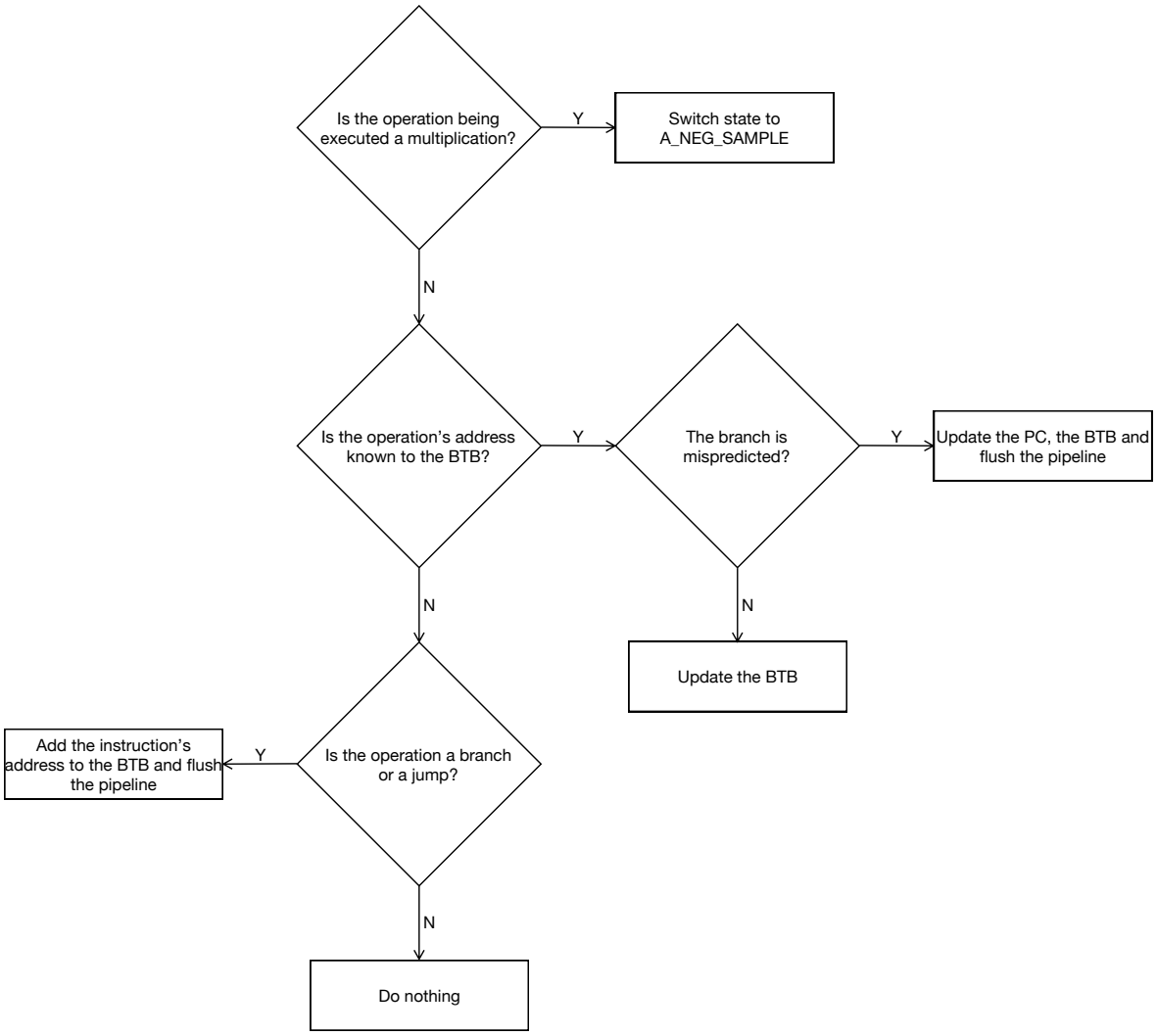

Figure 8.3: NORMAL_OP_EXE flow diagram

the BTB knew the instruction's address in the IF stage or not. If the address was known the only remaining check to be performed is whether the prediction was correct or not. In the first case the BTB prediction is updated, otherwise besides the BTB's update the pipeline must be flushed and the PC changed. If, instead, the address was not known, the FSM has to figure out if the instruction is a new branch or jump or not. In the latter case, besides writing as always the static control bits,

nothing is done. On the other hand, in the former case the address is added to the BTB, the pipeline is flushed and the PC is updated.

8.3.2 A_NEG_SAMPLE

This is the first state where the multiplication's execution starts. The EXE stage takes control over the ID/EXE registers, and uses the main adder to negate the value of the operand `a` on 64 bits. The next state of the FSM is automatically set to `MUL_IN_PROG`.

8.3.3 MUL_IN_PROG

The FSM iterates over this state 15 times, using the multiplier's adder to calculate each time the partial result. As soon as the 15th iteration is executed, the next state is changed to `MUL_END`. In this state the EXE still controls the ID/EXE registers.

8.3.4 MUL_END

This state is entered when the final result of the multiplication must be sampled by the EXE/MEM registers. After the result has been sampled, the EXE remains in this state (keeping therefore the pipeline locked) until the multiplication result has reached the WB stage. When this happens, the next state of the FSM is set to `NORMAL_OP_EXE` and the pipeline is unlocked. Again, this shows a limitation of the `mult`: its result cannot be forwarded to `mflo` and `mfhi` because it is on 64 bits and with out forwarding logic only 32 bits can be forwarded. Hence, to maintain data flow correctness the multiplication's result must be sampled in the RF before the next instruction can enters the EXE stage.

8.4 MEM stage

This stage, as all the previous ones, sends the static control bits to the datapath. When operations different from loads and stores pass through this stage, nothing more is done. Special attention must be given to stores and, more importantly, to loads. In fact, the formers may generate cache evictions, while the latter ones may generate cache misses. Thanks to the design of the data cache and of the memory controller cache eviction do not cause stalls, but cache misses do. Therefore, also in the MEM stage a FSM exists, of which the state diagram is shown in figure 8.4.

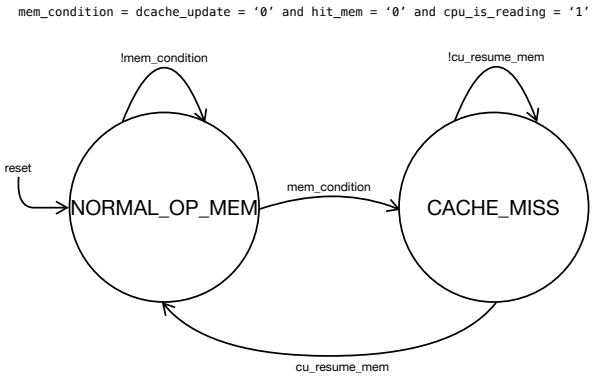

Figure 8.4: MEM stage state diagram

8.4.1 NORMAL_OP_MEM

The FSM starts in this state and remains always here for all operations but loads. When a load is executed, the content indexed by the address may have been swapped out in cache, so the cache's `hit` signal would stay at 0. When this situation is detected, the MEM stage along with the stall unit stalls the entire processor and then sets the next state to `CACHE_MISS`.

It is important to outline why `mem_condition` in figure 8.4 has that equation. Technically to detect the cache miss it would have been enough to look for `dcache_update = 0 AND hit_mem = 0`, since `dcache_update` signals the processor's will to perform a read (by not updating anything in the memory) and `hit_mem` signals if a hit or a miss have occurred. This is not sufficient however, because the cache is accessed at every cycle using as address the output of the ALU, no matter if the instruction was actually a load or not. For this reason in the CW there is a field called `cpu_is_reading`, which is set to 1 when the processor wants to actually access the cache.

8.4.2 CACHE_MISS

In this state the control unit hands over the control of the cache to the memory controller, which has recognized as well the cache miss in the previous cycle, by putting in high impedance `dcache_update` and `update_type_mem`. The CU remains in this state as long as the memory controller does not raise the `cu_resume` signal. When this happens, the pipeline is unlocked and the MEM stage FSM enters again the `NORMAL_OP_MEM` state.

8.4.2.1 Memory controller

The memory controller resides outside the control unit. It acts as a communication layer for the cache and the RAM, moreover as said before is capable of resolving cache misses. It has its own FSM to perform its job, shown in figure 8.5.

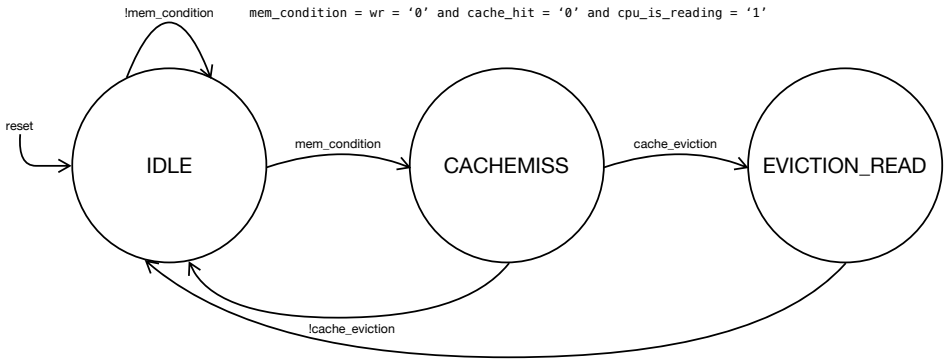

Figure 8.5: Memory controller state diagram

1. `IDLE`: this is the reset state. It stays here as long as no cache miss is detected and it keeps its cache's control signal in high impedance.

2. `CACHEMISS`: in this state it asks the cache to update its value. Two possible outcomes are possible here: if the cache is not signalling the need of an eviction the next state will be `IDLE`, otherwise it will be `EVICTION_READ`. In this case the `enable` signal of an internal register is enabled to sample the evicted data coming from the cache. In any case, after this state the control unit regains the cache's control.

3. `EVICTION_READ`: in this state the RAM is updated with the evicted data, then the next state is set to `IDLE`.

In the `IDLE` state, the memory controller may detect the occurrence of a data eviction during a write operation. This does not require a stall since it is able to directly forward the evicted data to the RAM.

8.5 WB stage

This is the final stage of the pipeline. Its job is fairly easy, since it has to forward the control signals to the datapath without any special exception. Besides this, the only other thing it does is to unlock the pipeline when a multiplication has reached this stage.

8.6 Stall unit

The stall unit has the duty of detecting data hazards and of handling all the signals required to stall the processor when needed. Hazards are detected by looking at `rs` and `rt` stored in the ID/EXE registers, and by confronting them with the `rd` registers stored in the EXE/MEM and MEM/WB registers. The `rd` registers are accompanied in the pipeline by a bit that specifies whether they are valid or not. This is necessary because to reduce the switching activity the registers not needed by an operation are disabled, therefore an invalid hazard could be detected. Data hazards are actually checked also from the ID stage, because as said in section 8.3 if a multiplication detects an hazard it has to stall.

This unit, before attempting to forward data, checks if it has to stall because the control unit has requested so. The sources of a stall can be:

1. A cache miss has occurred. This has the highest priority in the stall unit to guarantee data correctness. For example, a load could trigger a cache miss while a multiplication enters the EXE stage: if they have been served in the reversed order, the load would store in the register file the wrong data.

2. A multiplication is in the EXE stage. In this case `nop` operations are scheduled from the MEM stage onwards in order to allow to the operations in the MEM and WB stage to complete their execution.

3. A multiplication is in the ID stage and a data hazard has been detected.

If none of the above conditions are verified, the stall unit then controls what data can forwards to the EXE stage. Not all operations supports forwarding, as in the case of a `j`, or they could support the forwarding only of the `rs` register, as in the case of most —I— instructions. This information is embedded in the CW of the instruction in the EXE stage and it is encoded in two bits. The possible values are:

1. `00`: forwarding is not supported.

2. `01`: only `rs` can be forwarded.

3. `10`: both `rs` and `rt` can be forwarded (`rt` goes inside the adder's and shifter's operands).

4. `11`: both `rs` and `rt` can be forwarded (`rt` is forwarded in `op_b`, the signal where the data to be saved in the cache during a store is kept).

When the forwarding is supported by the instruction, it could be possible to generate a stall. In fact, if the instruction that should forward the data is a load, and said load is the MEM stage, it has not fetched yet the content of the memory. In this situation a bubble must be inserted in the pipeline.

CHAPTER 9

Simulation, Synthesis and Implementation

In this chapter we will discuss the flow we have followed from the DLX's simulation to implementation.

9.1 Simulation

The simulation has been performed on two tools: *Vivado* and *ModelSim*. The various components of the DLX have been developed by using the former tool, which is also the one we used to perform unit testing. The latter has been used to perform the final integration testing. In the *dlx_sim* folder a simulation script for ModelSim can be found, called `sim_script.tcl`.

This script first lets ModelSim to analyze the various VHDL files, then analyzed one of the provided testbenches and it runs a simulation for 100 ns.

We did not provided the testbenches we have written for testing the components in isolation because, during the final tests, we had to change many small things and we did not have enough time to modify them as well.

9.1.1 Test files

In the folder *tb* all the testbenches that can be used with `sim_script.tcl` are stored. By running all the scripts we have provided it is possible to verify the correctness of the entire DLX's ISA.

The files are:

1. `tb_arith.vhd`: it aims at testing all the various arithmetic instructions, either of type R and I, by also the forwarding logic.
2. `tb_branches.vhd`: it tests all the branches executed by the DLX along with the prediction abilities of the BTB.
3. `tb_jalr.vhd`: it verifies the correctness of the `jalr` instruction. This test is separated because otherwise `tb_branches` would have been too big.
4. `tb_jump_and_link.vhd`: it shows the ability of the processor to follows the ABI and to execute the `jalr` instruction. As a bonus, it demonstrate the ability of the control unit and memory controller to handle cache misses.
5. `tb_load_store.vhd`: it tests if the processor is capable to handle all the supported loads and stores.
6. `tb_mult.vhd`: it tests if the multiplication is properly executed in various scenarios, like when it would need forwarding and if it stalls the pipeline correctly.

7. `tb_r.vhd`: it checks whether the processor is able to execute all the supported R instructions, with and without forwarding.

The related assembly files can be found in the folder *test_assembly*.

9.2 Synthesis

Each time we developed a new datapath component, we have synthetized it to assess if it was able to meet our initial frequency target of 1 GHz. Among all the datapath components, the multiplier was the only one to not meet such standard, as it is able to reach a little bit more than 850 MHz if synthetized alone. When we synthetized the first version of the complete DLX (that is, also with the control unit and the memory controller) we used the `set_dont_touch` command to

1. use `compile_ultra` on critical parts of the design, that is the EXE stage and the control unit;
2. use `compile` on the rest of the processor, as it less time-sensitive.

The `set_dont_touch` command, in fact, tells to the tool to not modify anymore an elaborated and compiled entity. This removes some optimization opportunities, but it is very handy when dealing with big circuits such as a processor to reduce the synthesis time. The result of the synthesis didn't meet our performance target, since it reached only $800MHz$, but we still were satisfied. However, the more we have tested the DLX, the more we have found issues that needed to be fixed. Most of the issue were related to the control unit and the stall management, therefore it would have needed a complete redesign. Unfortunately, there was not enough time to do it, therefore we have been forced to add some hacks to make it work. This has severely affected the DLX's overall performance, as t_{CK} had to be raised to 1.9 ns. This results in $f_{CK} \approx 525\ MHz$, which is almost half of our initial frequency target. This value has been obtained by synthetizing the whole DLX with the `compile_ultra` command.

Below, the final time, area and power report are shown.

9.2.1 Time report

```
Startpoint: ctrl_u/curr_es_reg[1]
            (rising edge-triggered flip-flop clocked by clk)
  Endpoint: dp/id_exe_regs/b_mult_reg/curr_data_reg[23]
            (rising edge-triggered flip-flop clocked by clk)
  Path Group: clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  dlx_syn            5K_hvratio_1_1        NangateOpenCellLibrary

  Point                                                    Incr       Path
  ---------------------------------------------------------------------------
  clock clk (rise edge)                                    0.00       0.00
  clock network delay (ideal)                              0.00       0.00
  ctrl_u/curr_es_reg[1]/CK (DFF_X1)                        0.00       0.00 r
  ctrl_u/curr_es_reg[1]/Q (DFF_X1)                         0.09       0.09 r
  U4305/ZN (NAND3_X1)                                      0.04       0.13 f
  U4627/ZN (NAND2_X1)                                      0.04       0.17 r
  U4054/ZN (AND4_X2)                                       0.07       0.23 r
```

```
U4596/ZN (NAND4_X1)                                     0.05       0.29 f
U3980/ZN (NOR2_X2)                                      0.07       0.36 r
U3938/Z (BUF_X2)                                        0.05       0.41 r
U4032/ZN (INV_X1)                                       0.04       0.45 f
U7477/ZN (OAI22_X1)                                     0.05       0.50 r
U7153/ZN (INV_X1)                                       0.02       0.52 f
U7151/ZN (NAND2_X1)                                     0.03       0.55 r
U4560/ZN (XNOR2_X1)                                     0.07       0.62 r
U4341/ZN (NOR2_X1)                                      0.03       0.65 f
U4558/ZN (NOR2_X1)                                      0.07       0.72 r
U4550/ZN (NAND2_X1)                                     0.04       0.76 f
U4549/ZN (NOR2_X1)                                      0.05       0.80 r
U4529/ZN (AOI21_X1)                                     0.03       0.84 f
U4466/ZN (INV_X1)                                       0.06       0.90 r
U4459/ZN (AOI21_X1)                                     0.05       0.95 f
U4323/ZN (OAI21_X1)                                     0.05       1.00 r
U4322/ZN (XNOR2_X1)                                     0.06       1.06 r
U4319/ZN (NAND2_X1)                                     0.03       1.09 f
U4317/ZN (NOR2_X1)                                      0.04       1.13 r
U3971/ZN (AND4_X2)                                      0.07       1.20 r
U4452/ZN (NAND2_X1)                                     0.03       1.23 f
U4450/ZN (AND2_X1)                                      0.04       1.27 f
U4316/ZN (OAI21_X1)                                     0.04       1.32 r
U4315/ZN (NAND2_X1)                                     0.04       1.35 f
U4314/ZN (NAND2_X1)                                     0.04       1.40 r
U3973/ZN (OR2_X2)                                       0.05       1.44 r
U4443/ZN (AND2_X1)                                      0.04       1.48 r
U4390/ZN (NAND2_X1)                                     0.04       1.52 f
U4004/ZN (NAND2_X1)                                     0.06       1.59 r
U6576/ZN (NAND2_X1)                                     0.07       1.66 f
U4137/ZN (NOR2_X1)                                      0.11       1.77 r
U3277/ZN (AOI22_X1)                                     0.05       1.82 f
U3278/ZN (NAND3_X1)                                     0.04       1.86 r
dp/id_exe_regs/b_mult_reg/curr_data_reg[23]/D (DFF_X1)
                                                        0.01       1.87 r
data arrival time                                                  1.87

clock clk (rise edge)                                   1.90       1.90
clock network delay (ideal)                             0.00       1.90
dp/id_exe_regs/b_mult_reg/curr_data_reg[23]/CK (DFF_X1)
                                                        0.00       1.90 r
library setup time                                     -0.03       1.87
data required time                                                 1.87
-------------------------------------------------------------------------
data required time                                                 1.87
data arrival time                                                 -1.87
-------------------------------------------------------------------------
slack (MET)                                                        0.00
```

As it can be seen, the critical path starts in the EXE part of the control unit, specifically from the

register that holds the EXE FSM's state, and terminates in the multiplication's register `b` data.

9.2.2 Area report

```
Number of ports:                          600
Number of nets:                          6868
Number of cells:                         5918
Number of combinational cells:           5083
Number of sequential cells:               832
Number of macros:                           0
Number of buf/inv:                        899
Number of references:                      54

Combinational area:       10821.944148
Noncombinational area:    8594.991691
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          19416.935839
Total area:                 undefined
```

9.2.3 Power report

```
Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  =   6.9865 mW   (80%)
  Net Switching Power  =   1.7389 mW   (20%)
                         ---------
Total Dynamic Power    =   8.7254 mW  (100%)

Cell Leakage Power     = 463.1583 uW


                 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     11.1759        1.1986e+03           114.8263        1.2098e+03  (  13.17%)
register       6.7667e+03          72.8469        1.4489e+05        6.9844e+03  (  76.01%)
sequential         0.0000           0.0000              0.0000           0.0000  (   0.00%)
combinational    208.6468         467.4612        3.1815e+05         994.2623  (  10.82%)
--------------------------------------------------------------------------------------------------
```

```
Total          6.9865e+03 uW     1.7389e+03 uW     4.6316e+05 nW     9.1886e+03 uW
```

9.2.4 Synthesis files

The synthesis file are stored in the *dlx_syn* folder. Inside the folder it is provided a script called `syn_script.tcl`, capable of running the the whole synthesis process.

9.3 Implementation

CHAPTER 10

Conclusions

All in all, we have mixed feelings for this project. For a part we are satisfied by the amount of things we have been able to achieve, since we managed to add a large instruction set, a BTB, two caches, the forwarding unit and the ABI support.

However, we regret the way we implemented the multiplier. Actually, we regret to have added it at all.

In the earliest phase of development we have decided to rearrange the Booth's algorithm in such a way that it could be pipelined and would not take a lot of area. Unfortunately, we did not have enough time to implement it properly and this forced us to do compromise in order to keep it, compromises we would not have accepted by going back. It has hugely increased the complexity of the control unit and the amount of signals needed to handle all the possible stalls and corner cases. It also took a lot of time from us for testing and it had many problems, time that we could have dedicated to improving the rest of the system. This is all without even considering the fact that it is the only component in the EXE stage that, synthesized by itself with `compile_ultra`, is not able to reach the target frequency of 1 GHz.

The final frequency of our DLX is, indeed, "only" of 525 MHz, a decent value but far from our target. Sadly due to all the delays we had we did not have time to remove it and test everything again so in the end we kept it.

Bibliography

[1] Mariagrazia Graziano. *Microelectronic System Lecture Notes*. 2019.

[2] Peter Kankowski. *x86 Machine Code Statistics*. URL: `https://www.strchr.com/x86_machine_code_statistics`. (accessed: 16.10.20).

[3] Inc. MIPS Technologies. *MIPS64 Architecure For Programmers Volume II: The MIPS64 Instruction Set*. 2005.

[4] Wikipedia. *MIPS architecture*. URL: `https://en.wikipedia.org/wiki/MIPS_architecture`. (accessed: 17.10.20).