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, Giulia Santoro, Giovanna Turvani

Authors: ms17.50

Mauro Guerrera 222218
Claudio Poidomani 231090

November 1, 2017

Contents

CHAPTER 1

Introduction

The aim of this project is to implement a full-fledged processor based on the *DLX* architecture. The project covers the steps of digital design, simulation and synthesis stages.
The project is part of *Microelectronic Systems* course, held by Professor Mariagrazia Graziano in the first year of Embedded Systems Master Degree at Politecnico di Torino. This report introduces the processor architecture and provides a technical analysis of CPU's main components.

1.1 CPU structure

The *DLX* is based on the *Harvard* architecture, with separate Instruction RAM and Data RAM, which store the code and the data respectively. Since these memories are not synthesizable, they are out of the DLX top level entity.

There are 3 main components inside the DLX (figure 1.1). The control unit is in charge of every decision. It manages the normal flow of instructions and it is able to stall the pipeline and to forward data. The data path performs the actual operations (additions, multiplication, branches, etc.). The DMA can take control over the Datapath to perform routines like spill and fill (see chapter 5.2).

The processor has 8 interrupt lines (IRQ0 : IRQ7), with increasing priority. When an interrupt occurs, the corresponding line will be set to 1. An ack signal is generated when the interrupt is detected, so that the line can go back to 0 (see Chapter 6.2).

The DBG pin is used in *debug mode* to execute an instruction at a time (see Chapter 2).

The CRASH pin is set when the CPU stopped due to an unresolvable error.

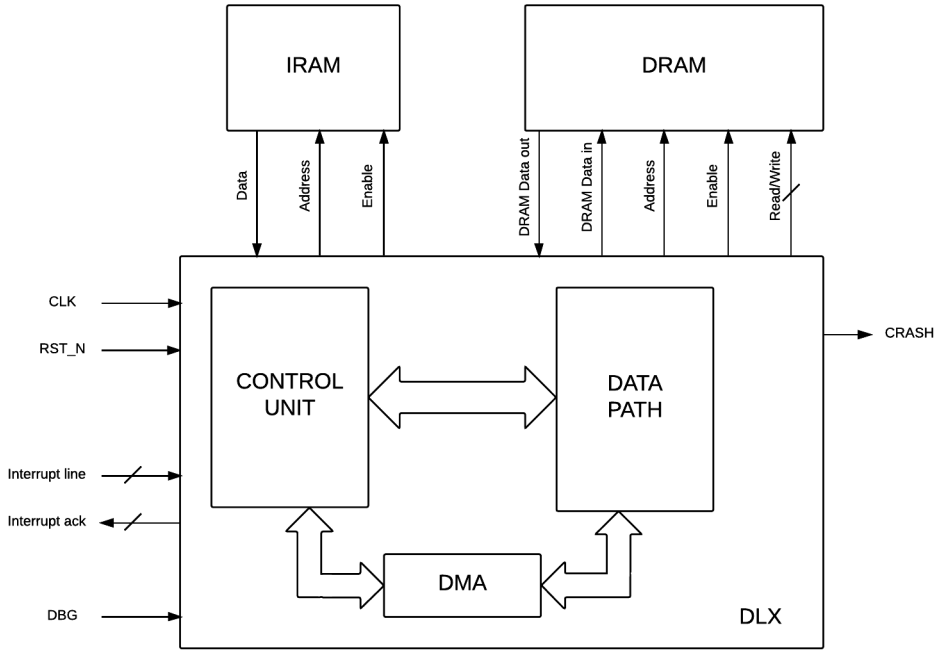

Figure 1.1: DLX structure

1.2 CPU pipeline

This DLX is a processor based on a 5-stage pipeline. The five stages are *Fetch*, *Decode*, *Execute*, *Memory* and *Write back*. The basic pipeline has been extended to include multiple execution units. In particular, a 3-stage floating-point multiplier and a 4-stage floating-point adder have been added. The final structure of the DLX pipeline is shown in figure 1.2.

The behaviour of the multiple execution units is different depending on the CPU mode. The DLX has 4 possible modes: *starting mode*, *debug mode*, *standard mode* and *fast mode*. For instance, in standard mode, just one execution unit at a time can be running. CPU modes are explained in depth in Chapter 2.

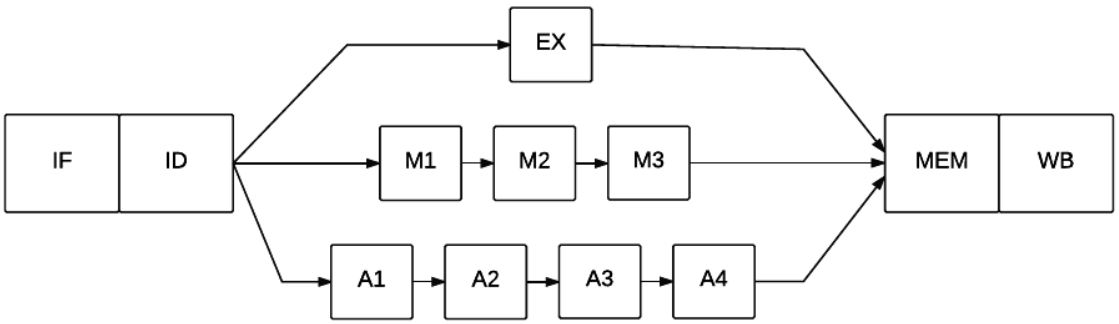

Figure 1.2: DLX pipeline

1.3 Memory

Although the DLX is based on the Harvard architecture, Instruction RAM and Data RAM are treated like there was a single RAM. The first addresses belong to the IRAM, and the following ones to the DRAM (figure 1.3). So, for instance, it is not possible to save a data on address 0 (it belongs to the IRAM).
Both memories store data in big-endian format. The IRAM is word-addressable, and a misaligned address will raise an exception (see Chapter 6). The DRAM can be addressed differently depending on the instruction. For example, the instruction `lb` can address every byte of the memory, an instruction like `lh` needs to be half-word-aligned, while the `lw` has to be word-aligned.
The IRAM is a read only memory, and the read process is asynchronous. The DRAM can be both read and written. The write process happens during the falling edge of the clock, while the read process is asynchronous. The *DLX* uses the technique of memory mapping to exchange data with I/O peripherals.
It is possible at compilation time to reserve some of the DRAM space to allocate a stack (see appendix B. If so, the instruction `push` and `pop` are the only instructions allowed to use the stack.

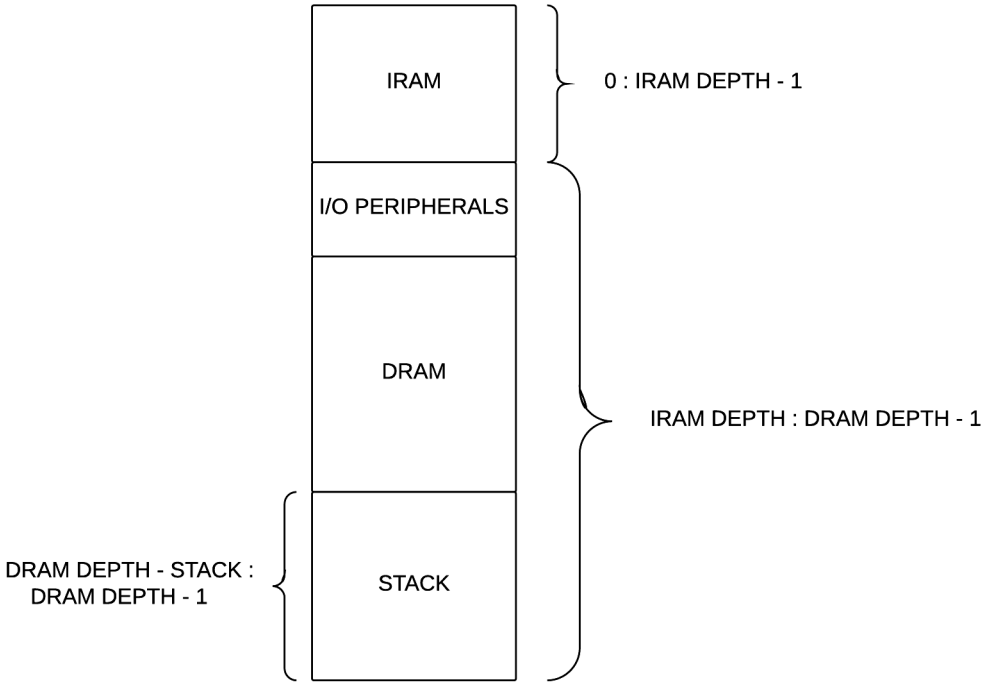

Figure 1.3: Memory organization

CHAPTER 2

CPU modes

There are four different modes in which the DLX can be running. It is possible to select the mode at compile time. It is not possible to change it at runtime.

2.1 Starting mode

Immediately after the reset, the CPU is in *starting mode*. This mode is used to execute just the first instruction of the code. This is a special instruction that set some parameters in the status register. In particular, it can set:

- CPU mode: it is the mode in which all the rest of the program will run.
- Stack protection : To reserve a portion of the memory to the stack, and activate the protection on it, which means that only a *push* or a *pop* instruction can access the stack.
- Exceptions : To enable or disable exceptions (see Chapter 6).
- Branch delay slot: To specify if the compiler uses this option (see Appendix B).

All these option will be set during the decode phase of the first instruction. After that, the program itself can start.

2.2 Standard mode

In standard mode, just one execution unit is allowed to be running at a time. For example, in a program like Code 2.1, the addi operation will be stalled in decode stage until the multiplication is finished, since the addi uses the integer execution unit. It is worth noting that in Code 2.2, the *multu* doesn't need to be stalled, since it uses the same execution unit as the *mult*.

```
1 mult f1 , f2 , f3
2 addi r1 , r1 , r4
```

Code 2.1: instructions with different execution units

```
1 mult f1 , f2 , f3
2 multu f4 , f5 , f6
```

Code 2.2: instructions with the same execution unit

This means that in standard mode every instruction terminates in order. As a consequence, the CPU can guarantee precise exceptions. So, assuming an instruction i generates an exception, all the instructions before i will complete, all instructions starting from i will be flushed, the routine relative to the exception will be executed, and at the end the program will resume from the faulty instruction.

2.3 Fast mode

In *fast mode*, multiple execution units are allowed to run at the same time. As a result, the CPU won't need to be stalled as much as in standard mode, and the overall performance will be improved. In a program like Code 2.1, the *addi* can enter the integer execution stage without being stalled, while the *mult* instruction is still in the multiplication execution stage. Since the multiplication execution unit is 3 stages long and the integer execution unit just 1, the *addi* instruction will finish before the *mult*.
The behaviour of the CPU in Code 2.2 is the same for fast mode and standard mode.
The main difference between fast mode and standard mode is that the instructions in fast mode can finish out of order. This results in an increased difficulty when try to guarantee precise exceptions. In our project, nothing has been done to deal with precise exceptions in fast mode. An exception can still be handled (if they are enabled), but when the CPU goes back to the program, it may produces incorrect results. For instance, the same instruction may be executed twice, even if it was written just once in the program.
In conclusion, if precise exceptions is a must, the only solution is to run the program in standard mode, even if it's a little bit slower.

2.4 Debug mode

The *debug mode* is a useful tool to test and debug a program. The CPU will execute just one instruction of the program, while keeping the next instruction stalled in the *fetch* stage. Whenever the input pin *DBG* is set, the instruction in the *fetch* stage will be executed, and the next one will be blocked.
An hardware debouncing circuit has been added to the pin *DBG* (Figure 2.1), so it is not possible to accidentally execute two instructions together.

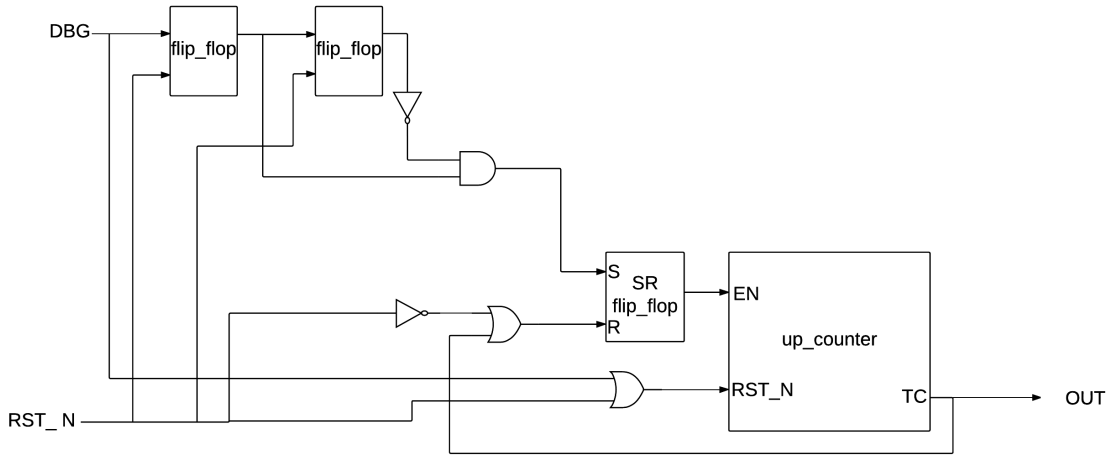

Figure 2.1: Debouncing circuit

Note that the *DBG* pin must stay at 1 until the terminal count is set, otherwise the counter will be reset. In our case, it must stay at 1 for 4 clock cycles (2-bit counter).

CHAPTER 3

Datapath

The complete structure of the data path is shown in figure 3.1. The green components represent the pipeline registers. Note that just the main 5 stages are shown in the figure. All the execution stages of the multiple execution units are inside the ALU block.

3.1 Instruction Fetch stage

The instruction fetch is the first stage of the data path. Normally, in this stage, the instruction at the address pointed by the program counter is sent in input to the instruction register, and the next program counter is calculated. The multiplexer driven by *IR_MUX_SEL* can select different things to send to the IR other than the output of the IRAM:

- During a *SPILL* or a *FILL*, the DMA can take control and fetch instructions that are not in the IRAM.
- There are some occasions when a `NOP` is fetched. For example, if branch delay slot is disabled, the instruction after a `jump` has to be flushed. This is done by fetching a `NOP` instead. Another example is in debug mode. While waiting for the input *DBG*, the CPU continues to fetch `NOPs`.

The value of the next program counter is also selected in the fetch stage, by the multiplexer driven by *JUMP_MUX_SEL*.

- *ECP_LUT* stores all the addresses where the routines to handle exceptions are. When an exception occurs.
- *EPC*, the exception program counter, stores the address of the instruction that has raised an exception, which is also the first instruction that has to be executed again after the exception routine is finished.
- *START_CODE* is the address of the first instruction of the program. This is used to jump at the program after the *starting mode* phase is over.

3.2 Instruction Decode stage

In the decode stage, the instruction in the IR is decoded. Two main changes have been made with respect to a normal decode stage.

- The *BRANCH* block, that has to decide whether to jump in case of branch operations, has been moved from the execution stage to the decode stage. The reason is that it is now possible to know if a branch is taken one clock cycle sooner, meaning that fewer instructions have to be flushed.
- the multiplexer driven by *BR_FWD_MUX_SEL* has been added to allow forwarding to branch operations, in order to reduce the stalls in case of data dependencies.

3.3 Execution stage

The execution stage is where the majority of the computational results are calculated. The main component of this stage is the ALU (see Chapter 4). The multiplexers *Mux A* and *Mux B* have been extended in order to allow forwarding to the ALU inputs. There is also another multiplexer used for forwarding, and it's for all the store operations. Finally, there is a small adder, outside the ALU. During a `push` or a `pop` operations, the ALU will compute the memory address, while this small adder will update the stack pointer.

3.4 Memory stage

Memory stage is where read and write operations in the Data RAM happen. The multiplexer outside the DRAM is used to extend correctly the output of the DRAM in case of particular load operations.

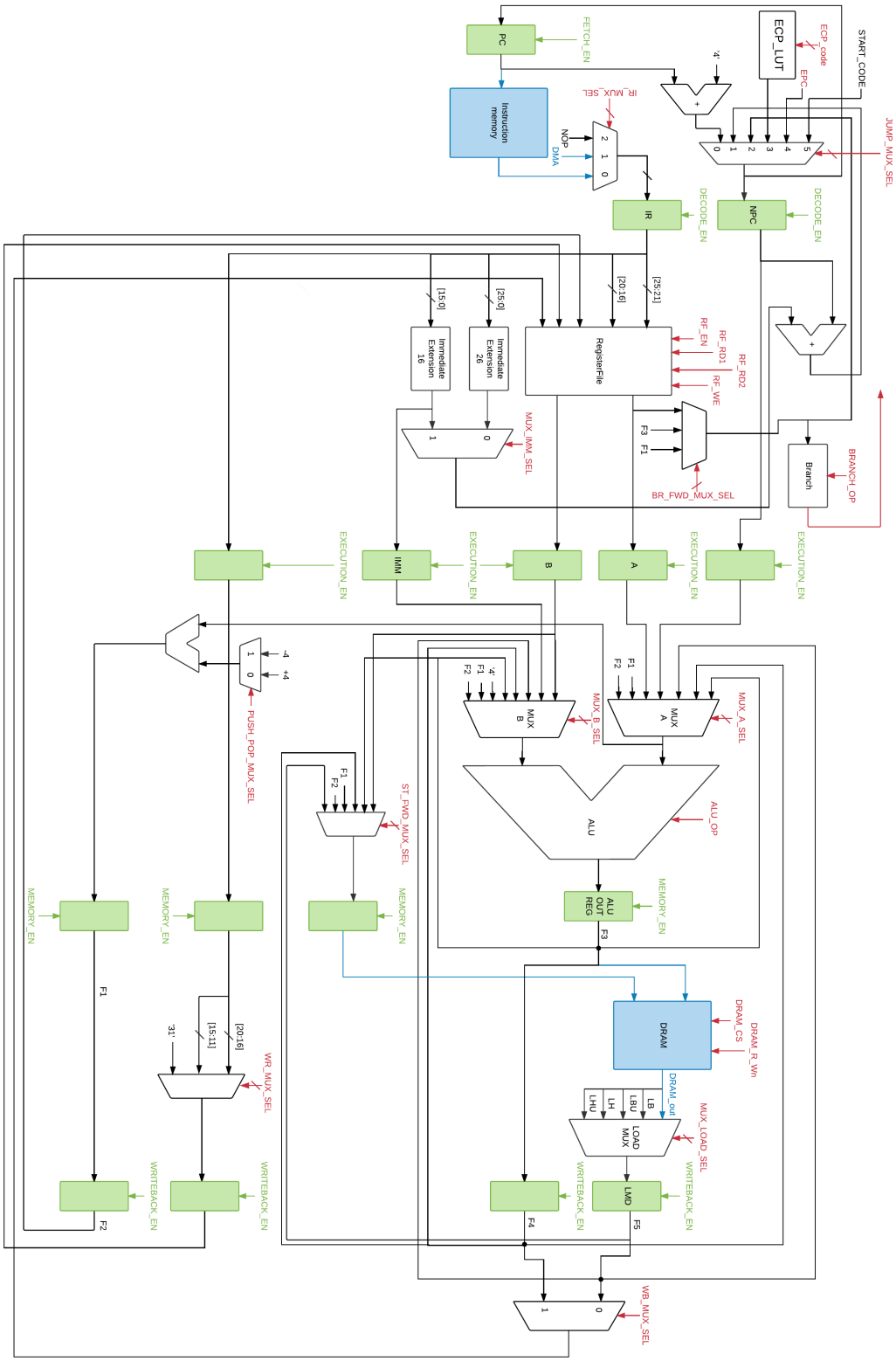

Figure 3.1: DLX datapath structure

CHAPTER 4

ALU

In this chapter the main components inside the ALU will be explained. The ALU structure is shown in figure 4.1. The floating point units aren't shown in this figure, but they will be explain in Chapters 4.5.2 and 4.6. The input *Data1* and *Data2* go to the correct component through some latches. The latches have been inserted so that just one component will perform some operation, while the others can remain idle, thus saving power. The output multiplexer select the correct output.

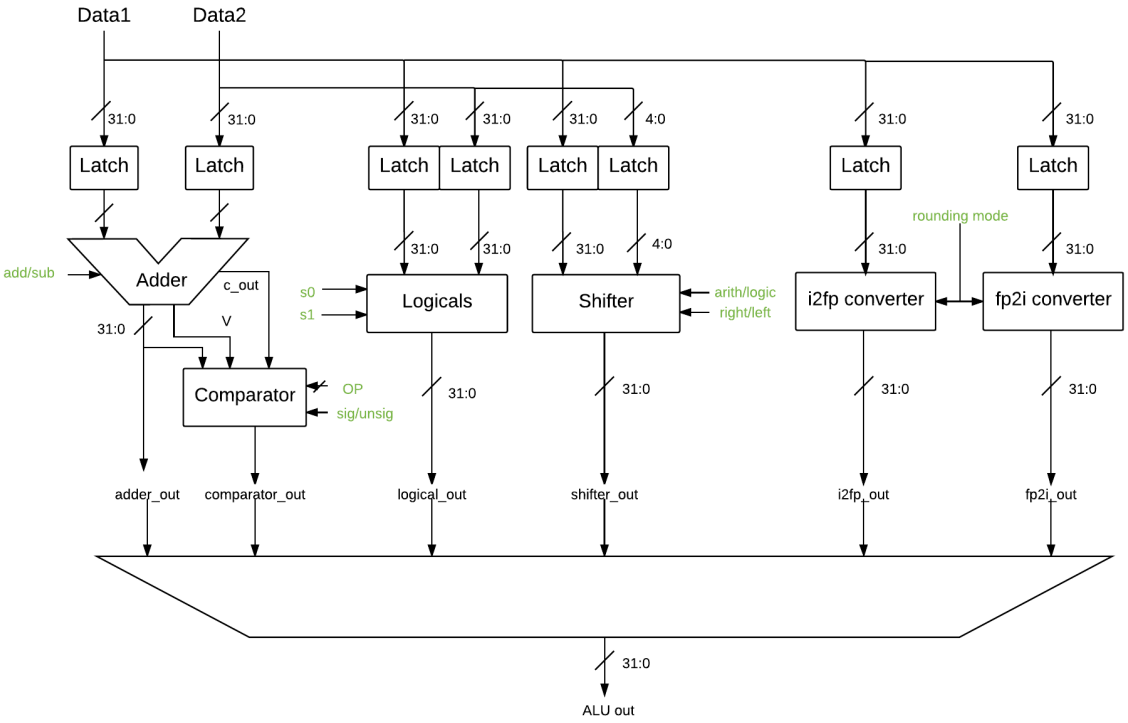

Figure 4.1: ALU structure

4.1 Adder

The adder used is the adder of the Pentium 4, the same implemented in the second laboratory. It is able to perform additions and subtractions on 32-bit operands. It can distinguish between signed and unsigned operations. In case of a signed operation, the output v is set in case of overflow, while for unsigned operations just the carry bit might be set, while the overflow is kept to a value 0.

4.2 Comparators

The comparators are used to compare two operands. The adder will perform the operation $Data1 - Data2$. The comparators will take the result, the overflow flag and the carry flag in order to calculate the final result. The schematic is shown in figure 4.2.

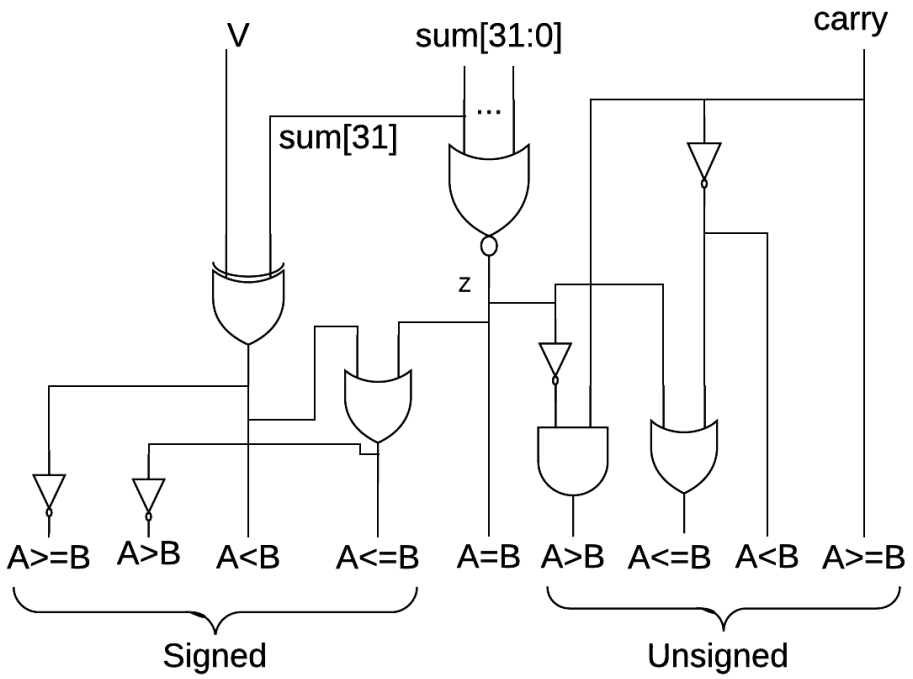

Figure 4.2: Comparators schematic

4.3 Logicals

The logicals block is used to compute bitwise operations *AND*, *OR* and *XOR*. The selection bits *s0* and *s1* are used to specify the operation, while *R1* and *R2* are the inputs (figure 4.3).

Operation	s0	s1
AND	0	1
OR	1	1
XOR	1	0

Table 4.1: Logicals operations

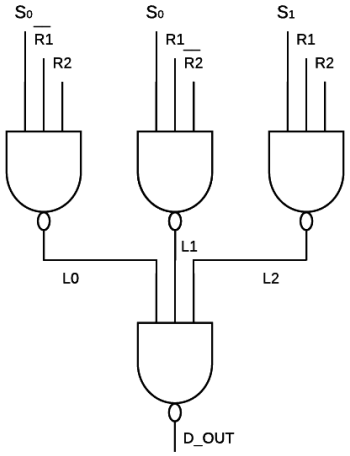

Figure 4.3: Logicals schematic

4.4 Shifter

The shifter can perform a shift operation up to 32 positions, both right and left, both logical and arithmetical. The structure is similar to the T2 shifter analyzed in class. It is organized on 3 different levels (figure 4.4). The first level generates 4 masks, different for left or right shift. The second level perform a coarse grain shift, and the third level a fine grain one. The second and third levels are shown in figure 4.5.

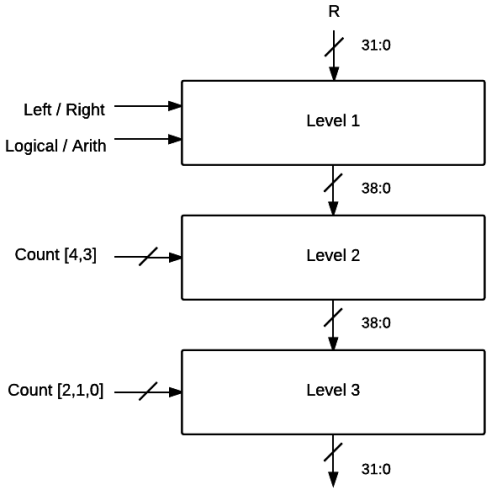

Figure 4.4: Shifter schematic

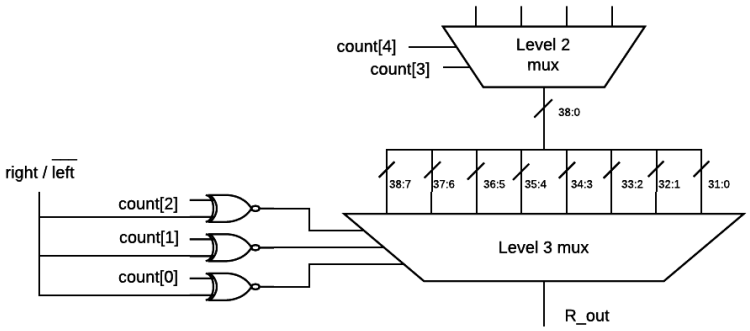

Figure 4.5: Shifter schematic

4.5 Multiplier

The DLX has a multiplier capable of doing both integer and floating-point multiplications. The integer multiplications can be both signed and unsigned. A floating point multiplication takes 2 floating-point registers on 32 bits, and save the result in a 32-bit register. An integer multiplication takes the lower 16 bits of 2 floating-point registers and it saves the result on a 32-bit register.

4.5.1 Integer multiplier

The integer multiplier is a block inside the floating-point multiplier (figure 4.6. The 16-bit inputs are extended (with or without sign) to 25 bits. The reason is that this same block will be used to multiply the mantissas in floating-point multiplications. Then, a block generates all the partial products according to the booth algorithm. The partial products will go through a series of carry save adders, the Wallace Tree, and a CLA will perform the last addition. At the end, the result reconstructor will select the correct 32 bits of the result.

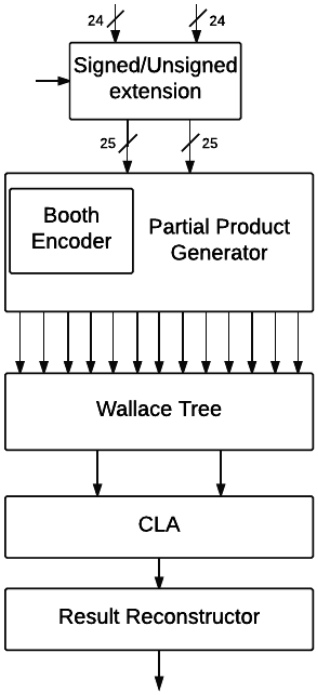

Figure 4.6: integer multiplier structure

Since it has its operand on 16 bits, and the final result on 32 bits, an integer multiplication cannot generate overflow (nor underflow).

4.5.2 Floating-Point multiplier

The floating-point multiplier extends the integer multiplier in order to perform the more difficult floating-point multiplication (figure 4.7). It is a 3-stage multiplier. The green lines in the figure is where pipeline registers have been added.
The sign evaluation is just a single xor port. The exponent evaluation is the sum of the biased exponents, and a subtraction of the bias, which would be counted 2 times otherwise. The mantissas of the inputs are unpacked (there is an implicit 1 in the structure of a floating-point number), then extended again with 0 (the booth algorithm works with signed numbers, while the mantissa is unsigned). Then, the 2 resulting values are multiplied, using the integer multiplier. The final result is then normalized and eventually rounded, according to the rounding mode. There are 4 possible rounding modes:

- *Minus infinity*: if the result need rounding, a +1 is added if the sign of the result is negative.
- *Plus infinity*: if the result need rounding, a +1 is added if the sign of the result is positive.
- *Zero*: The result is always truncated.
- *Nearest*: This is the classical way of rounding numbers.

The floating-point multiplier has some output flags used to notify the CPU in different scenarios:

- An overflow flag is set when the result of the multiplication is too big to be represented (note that $\infty * x = \infty$, without overflow).

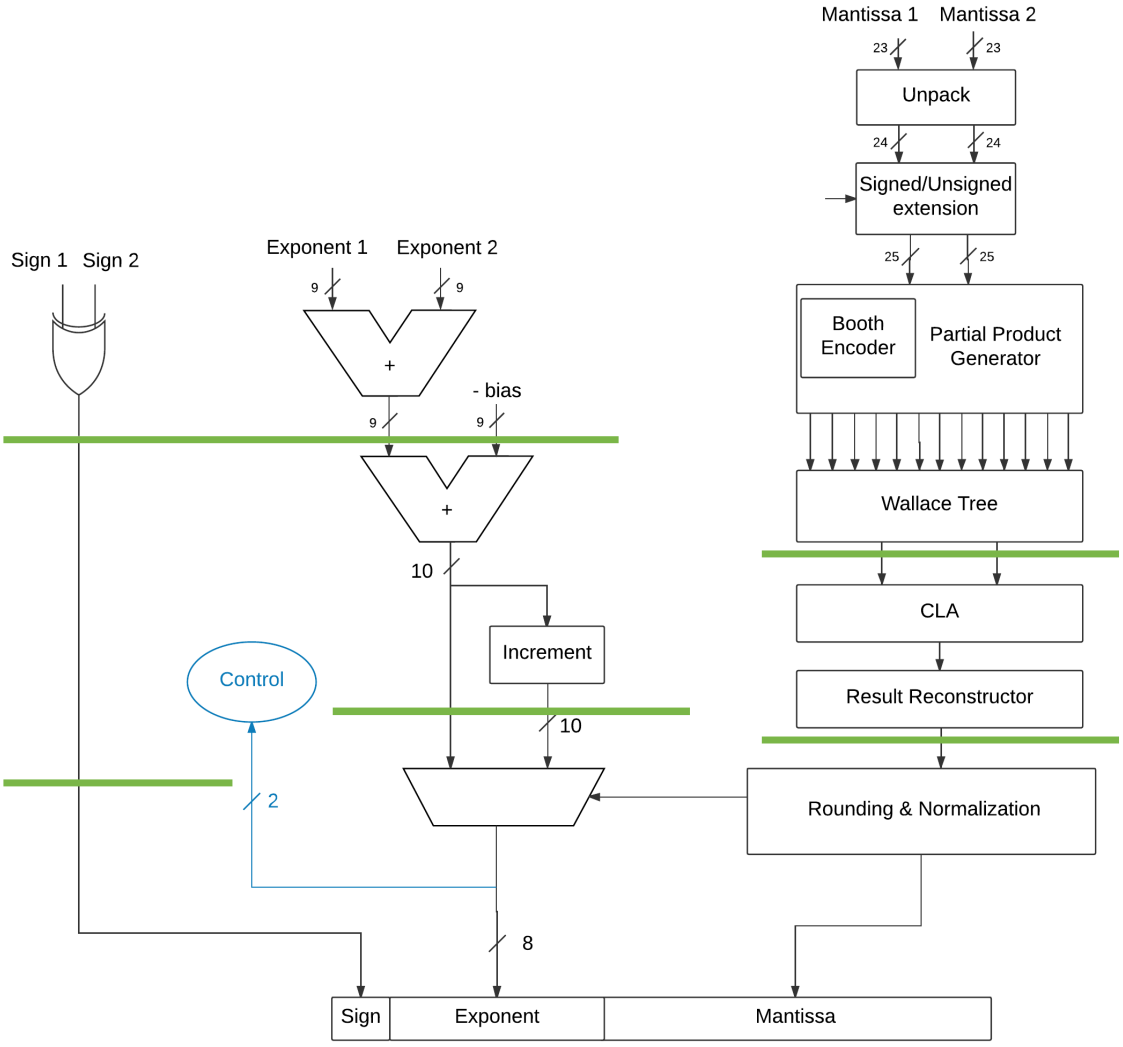

Figure 4.7: FP multiplier structure

- An underflow flag is set when the result of a multiplication is too small to be represented.
- An invalid flag is set when the multiplication is not possible. Example of that are the operations $NaN * x$, $\infty * 0$, $+\infty * (-\infty)$, etc.

4.6 Floating-Point Adder

The 4-stages floating-point adder is able to perform additions and subtractions on 32-bit floating-point registers (figure fig 4.8. The general idea of a floating-point addition is to find the greater of the two input, right shift the lower one until the 2 operands have the same exponent, then perform the addition/subtraction. At the end, normalize and round the result, adjust the final exponent, and evaluate the sign.
For a complete explanation of the algorithm, check out the Hennesy-Patterson book "Computer architecture, a quantitative approach".
As the floating-point multiplier, the adder supports 4 rounding modes, and it ca generate overflow,

underflow and invalid flags.

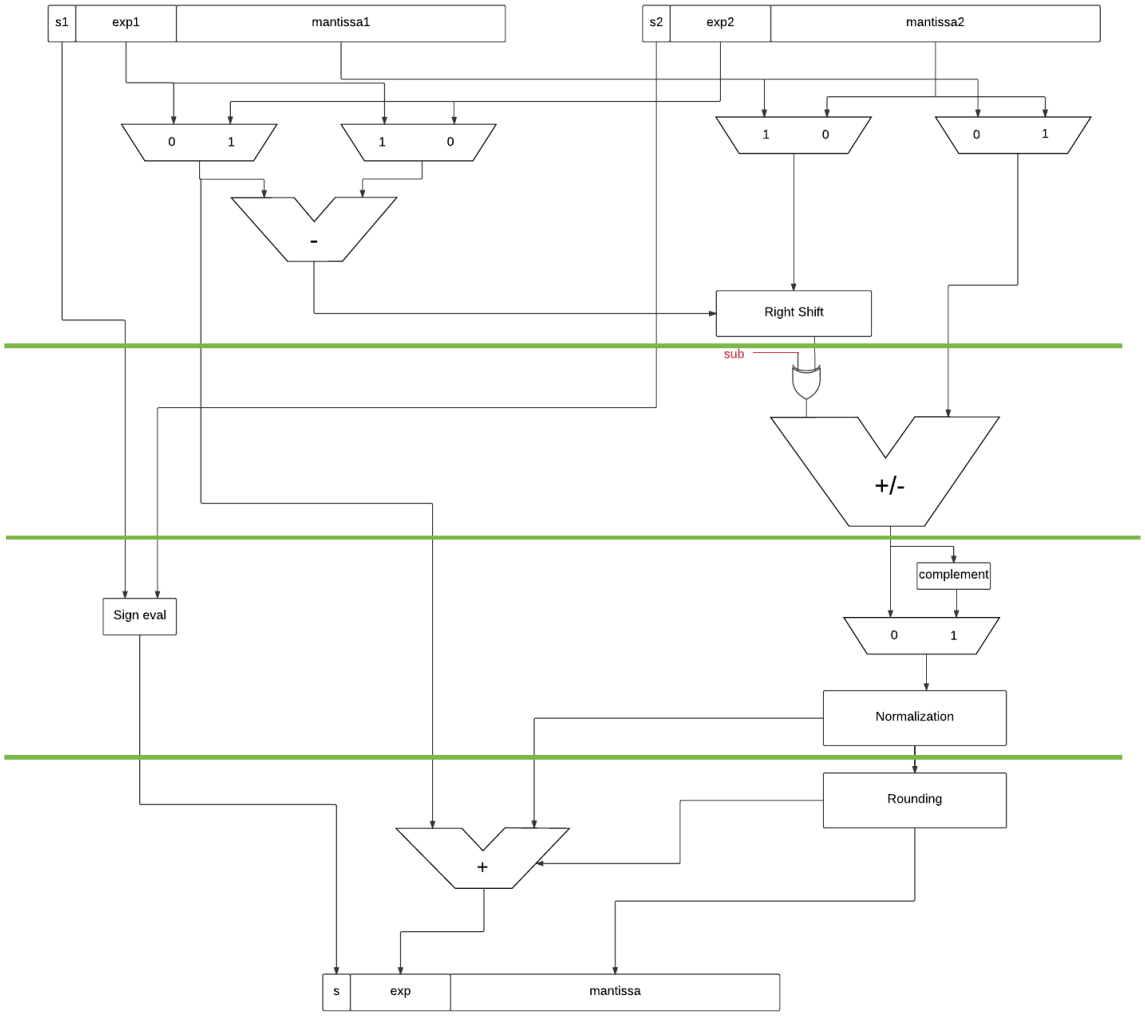

Figure 4.8: FP adder structure

CHAPTER 5

Windowed Register File

5.1 Register File

The DLX includes a Windowed Register File with 72 total registers for Integers and 72 total registers for Floating point values. The registers are divided in banks. Each bank has 8 32-bit registers (figure 5.1). Integer registers banks and floating point registers banks are treated in the same way: the first 8 banks are used to implement the window mechanism, with 3 banks active at a time (*IN*, *LOCAL*, *OUT*). Last bank is used for *GLOBLAS* registers and it is always active, for a total number of 32 active Integer registers and 32 active Floating point registers.
The *OUT* bank of the current window overlaps with the *IN* bank of the next window. This allows to share parameters between functions when using a `call` or a `ret` instruction. If a call or a return instruction move the current window over an already used bank, it is necessary to save current data inside the WRF to DRAM before overwriting registers. This is achieved through the spill/fill process (see 5.1.1).
Both integer and floating-point registers indexed from 0 to 23 refer to the 3 banks of the current window and are for general purpose use. Registers from 24 to 31 are global registers and some are dedicated for special purposes (figure 5.2):

- `R24` is the Zero register (it has been moved to `R24` because `R0` is part of the *IN* bank of wach window;
- `R25` is the Status Register and stores information about the current CPU status;
- `R26-R27-R30` are global general purpose registers;
- `R28` is the Control Register and it is used to mask interrupts and to set the rounding mode;
- `R29` is the Stack Pointer;
- `R31` is the Return Register;
- `F24` is the Zero register for the floating-point registers.

5.1.1 Spill and Fill

In order to manage the window mechanism, there are two registers called *CWP* (Current Window Pointer) and *SWP* (Saved Window Pointer), that store the index of the current window and the index of the next window to be saved, respectively. *CWP* is incremented on `call` and decremented on `ret`. When the value of *CWP* equals the value of *SWP*, a spill or a fill is needed, depending

on which instruction caused that condition. When *CWP* is incremented on a `call`, the three banks associated to the *SWP* are stored into the DRAM (SPILL) and the *SWP* is incremented by one. On the contrary, when a *ret* instruction decrements *CWP*, it is necessary to restore registers from DRAM to WRF (FILL) and to decrement *SWP*.
Spill and Fill processes introduce a considerable delay in the program execution, although they allow to perform a virtually unlimited sequence of `call` and `ret` instructions.

5.1.2 WRF read/write ports

The WRF has multiple read and wirte ports. In particular, there are:

- two general purpose read ports.
- one general purpose writing port.
- a write port reserved for the Stack Pointer.
- a write port reserved for the Status Register. This is also the only port that can write on it. It is not possible to write on the Status Register using the general purpose port.
- a read port for the Status Register.
- a read port for the Control Register.

The Read process is asynchronous, while the Write process happens on the falling clock edge (writing on the falling edge allows to avoid some data hazards, but it affects the critical path of the DLX, see Chapter 8 for further information). There are three control bits that allow to choose whether read/write ports address Integers or Floating point banks.

5.2 DMA

The DMA allows to access memory using the pipeline hardware while *Fetch* stage is stalled. For instance, when a SPILL or FILL process is requested, the Control Unit stalls the *Fetch* stage and the DMA output is connected directly to Instruction Register input (see Chapter 3). To handle SPILL/FILL process, there is an internal counter that generates all register indexes between 0 and 23. The process is repeated twice to allow SPILL/FILL of both Integer and Floating point registers. The DMA outputs a 32-bit value which contains the opcode of a `push|pushf` (or a `pop|popf`) and the register index generated by the counter. When the process starts, a control bit is set to signal that a spill/fill process is running. The bit is cleared when the spill/fill terminates, and the normal control flow can start again.

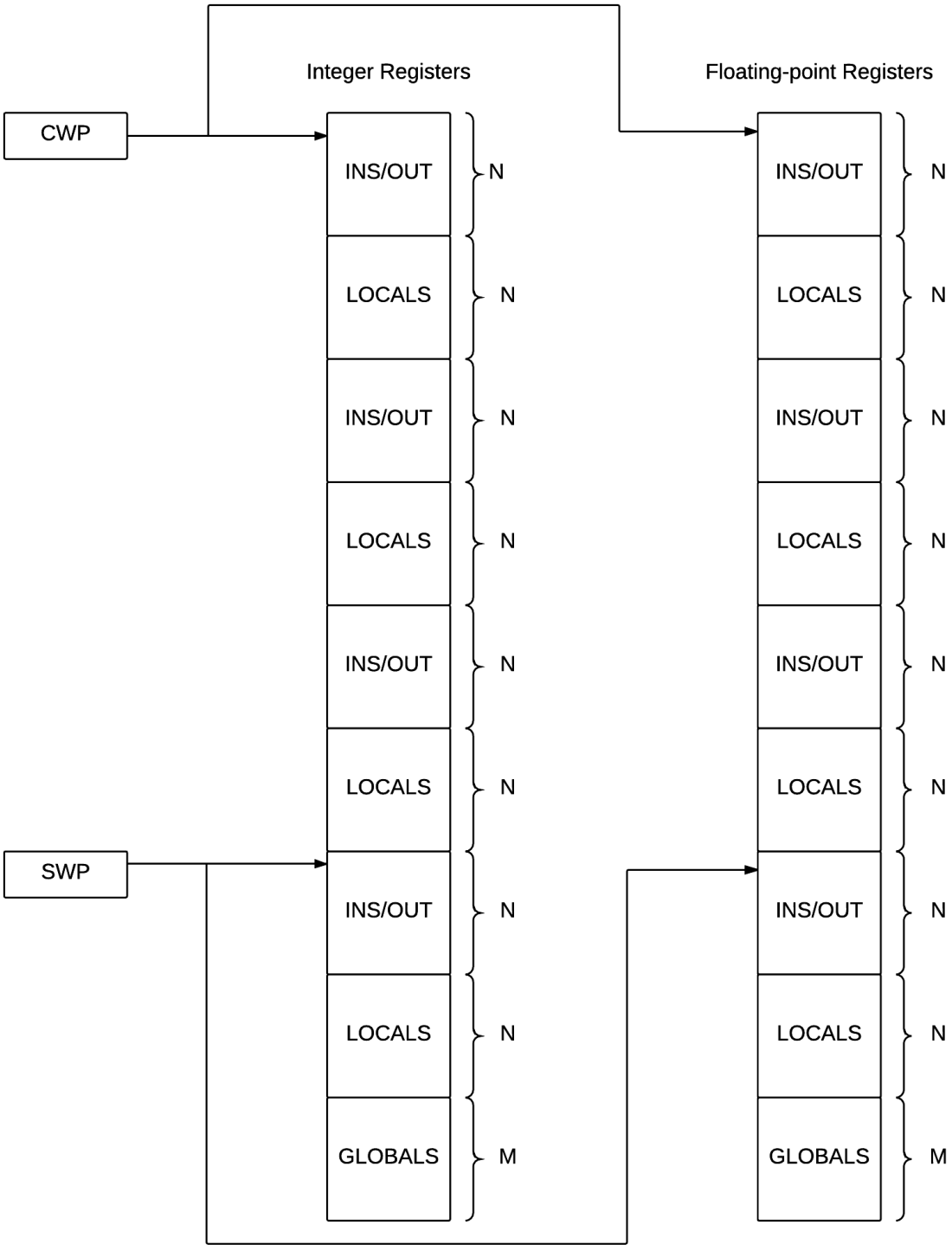

Figure 5.1: Windowed Register File

bit	R0/R23 - GP	R24	R25 - SR	R26 /27- GP	R28 - CR	R29 - SP	R30 - GP	R31 - RET ADDR
0	General purpose	0	CPU MODE	General purpose	ROUNDING MODE		General purpose	
1								
2			STACK PROTECTION					
3			IRAM PROTECTION					
4			ECP_LOCK					
5			BRANCH_DL_SL					
6			ECP MODE					
7								
8			IRQ line		IRQ mask			
9								
10								
11								
12								
13								
14								
15								
16			OF					
17			UF					
18			CR					
19			ZERO					
20			INVALID					
21								
22								
23								
24								
25								
26			RESERVED					
27								
28								
29								
30								
31								

Figure 5.2: Registers organization

CHAPTER 6

Exceptions and Interrupts

Exception definition

Since the definition of exceptions and interrupts is often confusing, in this project the following definitions are considered valid:

- Exception : An event caused by a faulty instruction. This event is internal to the processor, and it is synchronous. The same exception will always occur at the same time inside the CPU.
- Interrupt: An asynchronous external event caused by some hardware or software peripheral to raise the attention of the CPU.

6.1 Exceptions

The user can choose at compile time to enable or disable the exceptions. There are 3 possible options:

- Disabled: All exceptions (and interrupts) are disabled.
- Enabled: All exceptions (and interrupts) are enabled.
- No arith: Exceptions and interrupts are enabled, except for all the arithmetic ones (overflow, underflow and invalid operation).

Hereinafter are shown all the possible exceptions and their meaning

- Overflow: The operation inside the ALU has generated an overflow. This may come from a signed integer addition or subtraction, floating-point multiplication, floating-point addition and fp-to-integer conversions.
- Underflow: The operation inside the ALU has generated an underflow. This may come from a floating-point addition or a floting-point multiplication.
- invalid operation: The operation inside the ALU has generated an invalid operation. For example, a floating-point multiplication when one of the operands is a NaN.
- Invalid Opcode: The opcode of the instruction in the instructions register does not correspond to any valid operation.
- IRAM misaligned: The program counter tries to access the IRAM to an address that is not a multiple of 4. Since each instruction is stored in 4 bytes, an IRAM misaligned exception means that the CPU is try to read bytes of different instructions.

Table 6.1: Exception table

CODE	EXCEPTION
0	no ecp
1	overflow
2	underflow
3	invalid operation
4	invalid opcode
5	iram misaligned
6	iram reserved
7	dram misaligned
8	dram reserved
9	irq

- IRAM reserved: The program counter does not have the permission to access the memory location that it's pointing at. During a normal program, the program counter cannot access the DRAM, nor a portion of the IRAM which is reserved to the exceptions routines.
- DRAM misaligned: The address trying to access the DRAM is not correctly aligned. Note that the this depends on the instruction. For instance, a `lb` cannot be misaligned, it can read every byte of the DRAM, while a `lw` has to be aligned to read the correct 4 bytes of a word.
- DRAM reserved: An instruction is trying to read or write a data in a location where it has no permission to do so. This can happen when the address is pointing to a location of the IRAM, or if a normal load or store are trying to access the stack. Remember that if the stack has been allocated, only `push|pushf` and `pop|popf` are allowed to use it.
- Interrupt: An external interrupt has been detected.

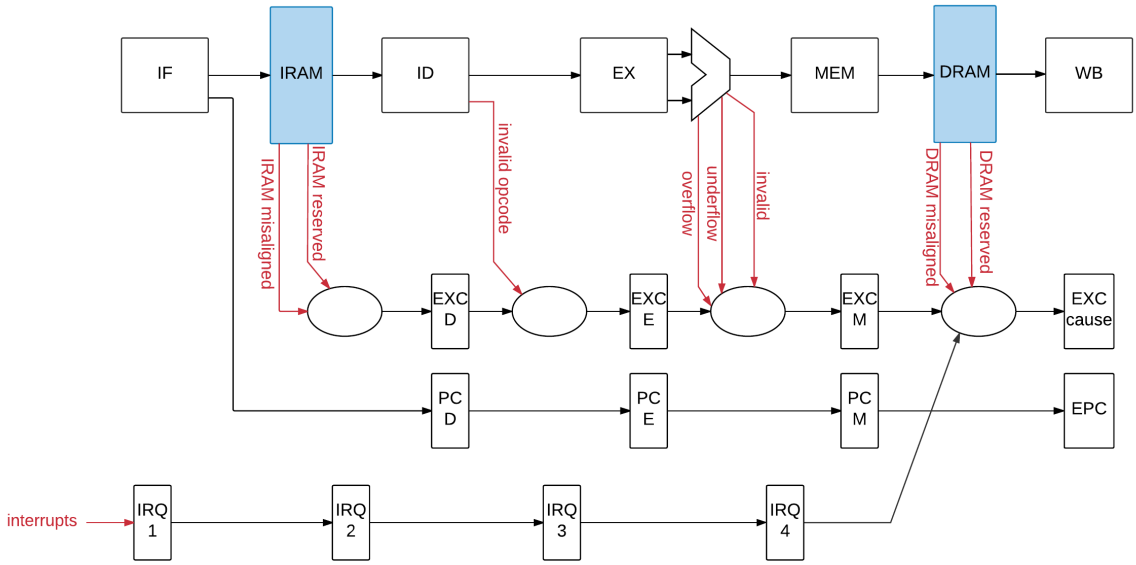

Figure 6.1: exceptions structure

6.1.1 Exception handling

Exceptions can be detected in different stages, from *Fetch* to *Memory* (figure 6.1), but they are always handled once they arrive in *memory stage*. Once captured, they are propagated through *exception pipeline* and detected by the Control Unit. The whole pipeline is flushed, preventing the instruction that caused the exception from completing its *Write Back* stage. As the exception is detected, the value of the *Program Counter* of the instruction that raised the exception is stored into the *Exception Program Counter*, and the address of the exception routine is loaded into the *Program Counter*. The *ECP* is then used to resume the program after the exception routine has been executed. The only instruction able to read the *EPC* is `rfe`, which should terminate every exception routine.

6.2 Interrupts

Interrupts are handled in the same way as exceptions, but are captured one stage early. This allows to complete the instruction that was in *Fetch* stage when the interrupt was captured. Interrupts can be masked by *Control Register* bits `15:8`. If set to 0, the corresponding interrupt line will not be detected. It is important to notice that the content of the *Control Register* is not forwarded, hence it is necessary to wait for the instruction that sets the mask bits to complete before the CU actually masks corresponding interrupts. Pending interrupts are stored into bits `15:8` of *Status Register*.

6.2.1 IRQ handshake protocol

There are 8 interrupt lines (`IRQ0:IRQ7`) with increasing priority. An *ACK* line is associated to each interrupt line, to notify the external peripheral that its request has been detected and that the CPU is ready to start the handler routine. When one or more interrupt lines become active, they are propagated through the interrupt pipeline and eventually detected by the exception check.
The *ACK* corresponding to the highest priority interrupt is set as soon as the interrupt is detected. When the peripheral receives and elaborates the *ACK* signal, it notifies the CPU by setting to 0 its interrupt line (*handshake*). The interrupt routine can start as soon as the peripheral completes the *handshake* procedure.
It is important to notice that the priority check over multiple active interrupt lines, as well as masks check, is performed in the last interrupt pipeline stage. This way, elder interrupts will have priority over newer ones, although some of them could have an higher priority.
The interrupt protocol of the DLX requires that any peripheral that activates its interrupt line, must keep it at 1 until the *handshake* procedure is completed. This is due to the fact that when an interrupt is detected, the interrupt pipeline is flushed and the interrupt detection is disabled until the routine completes. Once `rfe` instruction resumes the normal program execution, the interrupt detection is re-activated and any pending interrupt will be detected.

CHAPTER 7

Control Unit

The Hardwired Control Unit is the most complex part of the DLX processor: it performs multiple tasks, from deciding the instruction flow through the pipeline to avoid data and control hazards. It is also in charge of detecting exceptions and interrupts.
Instruction flow is controlled by an *Instruction Registers* pipeline that follows the progress of each instruction through the pipe.
Control signals associated to each instruction (*Control Word*) are stored into the *Control Word memory*. The *Control Word* of each instruction is set in *Fetch* stage and it is propagated through the pipeline, according to the Instruction Registers pipeline.
By checking opcodes and register indexes at different stages, it is possible to activate the correct control signals in order to avoid hazards. If forwarding is available, it is sufficient to substitute the selector of some multiplexers into the *Control Word* at the correct stage. If forwarding is not an option, the CU will stall some stages of the pipeline by deactivating the desired enable signals.
In addition to Control Word and Instruction Registers pipelines, also Interrupts and Exceptions will propagate from one stage to another through similar pipeline registers (see Chapter 6).

7.1 Hazards

Structural, Data and control hazards are handled by the two main processes of the Control Unit:

- *forwarding* process controls all dependencies between the source register(s) of the instruction in *Decode* stage and the destination register of the instruction in *Execution* and *Memory* stages.
- *stalls* process covers all possible cases when forwarding is not available and handles internal and external stall sources, as interrupts, exceptions, branches and DMA requests for spilling/filling.

It is worth noting that it's possible to have different kinds of hazard depending on the CPU mode.

7.1.1 Structural Hazards

A structural hazard occurs when multiple instructions complete the execution stage and try to access the memory stage together. Since there is a single port to write and read the DRAM, the only solution to this hazard is to stall the pipeline, and let just one instruction at a time into the memory stage. Note that this hazard is only possible in *fast mode*. In *standard mode*, just one execution unit at a time is active.

7.1.2 Control Hazards

As explained in Chapter 3, the *BRANCH* block has been moved to the decode stage. This allows to reduce the instructions lost to just one. Control hazards don't depend on the CPU mode

7.1.3 Data Hazards

Data hazards can be handled by the forwarding process, or by the stall process when it's not possible to use the forwarding. There are 2 different types of possible data hazards:

- RAW (read after write): this is the most common, it happens when an instruction has to write on a register, and the following instructions try to read it before it has been updated.
- WAW (write after write): it happens when two instructions have to write on the same register, but due to out of order execution, the final result stored in the register is not the correct one. This is possible only in fast mode, since in standard mode instructions complete in order.

A special case is represented by the *BRANCH* block: while any other instruction needs operands ready in *execution* stage, branch instructions need to know the correct value one stage before. In order to take advantage of the anticipation of the branch block to the *decode* stage, it is necessary to forward asynchronously data from *memory* stage. Forwarding from *execution* stage is not available for branch instructions.

CHAPTER 8

Synthesis

8.1 Synthesis of execution units

Floating point adder and multiplier units have gone through a first synthesis step, in order to determine the number of pipeline stages needed for each unit. The top level entity of the DLX has been synthesized to determine the delay on the critical path, which was between 2.8 and 3 `ns` (depending on synthesis options). According to these values, it has been decided to split the multiplier into 3 stages and the floating point adder into 4 stages. This allows a good tolerance margin, hence both execution units do not affect the overall critical path.

8.2 DLX synthesis

The final *DLX* has been synthesized trying different options and constraints

8.2.1 Results

Power

The final dynamic power consumption is 6.14 mW. This result has been obtained using the option `-clock_gate`. Without the clock gating option, the dynamic power consumption was around 32 mW.

Area

The total cell area obtained is 67070.3. This result has been obtained without any constraint on the area.

Timing

We had some difficulty understanding some of the timing reports. Using a clock of 2.8 ns, the slack was violated, saying that the data required time was 2.75 ns, and the data arrival time 2.90. However, after trying with a clock of 3 ns, the slack was violated again. The synthesis tool wasn't able to obtain a data arrival time of 2.90 ns. The first report with slack met was with a clock of 3.3 ns.

CHAPTER 9

Notes and Open Issues

- In debug mode, if branch delay slot is enabled, the instruction following the jump will be executed in the same debug step as the jump.
- `call` and `ret` instructions are **not compatible** with Branch Delay Slot. This is due to the change of CWP. In fact, the instruction immediately after the `call` (or `ret`) will go through the *decode* stage with the old CWP, but when it arrives in *write back*, the CWP will be already updated. If Branch Delay Slot is enabled, a `NOP` instruction must be manually placed after any `call` or `ret`. If the branch delay slot is disabled, the instruction following the `call` (or `ret`) has to be flushed anyway, so the program will execute correctly.
- During a *SPILL* or a *FILL* process, interrupts are automatically disabled. If an exception occurs in the meantime, the CPU will crash. This is due to the difficulty of stopping the *SPILL* (*FILL*) process, executing the exception routine and then go back to complete the *SPILL* (*FILL*) process. A possible case of exception is if there is no more space in the stack.
  If a crash occurs, the output pin *CRASH* will be set, the entire pipeline will be flushed, and all the enables will go down. The only way to exit from a crash status is to reset the CPU.
- Nested exceptions/interrupts are currently not supported. During an interrupt/exception routine, interrupts and exceptions are automatically disabled. If another exception occurs, it won't be detected. If an interrupt occurs, it will be detected after the routine will be finished.
- When using `call` or `ret` instructions, if a Spill (Fill) process is performed, the Stack Pointer will be incremented (decremented) to store (restore) registers. This means that values stored in the stack before Spill will not be available anymore using `pop|popf` instructions. Similarly, if any `push| pushf` is performed after a Spill, it is strictly necessary to restore the correct Stack Pointer before the Fill process starts.
- The floating-point multiplier is not able to handle denormalized numbers, while the floating-point adder can. (Note that according to the ieee 754 standard, if the result of an addition is a denormal, the adder should anyway set an underflow flag).

APPENDIX A

DLX instructions

Table A.1 shows all the instructions supported by the DLX. The instructions `movfp2i` and `movi2fp` were originally R-type operations, and instructions `cvtf2i` and `cvti2f` were F-type operations, while now they are all general instructions. Some instructions were not present at all in the original DLX, and have been added:

- `push`: it stores the value of an integer register in the stack, at the address of the stack pointer, and then it decrements the stack pointer by 4.
- `pop`: it increments the stack pointer by 4, and then saves the value of the memory location corresponding to the stack pointer into an integer register.
- `pushf`: It's like a push, but with floating-point registers
- `popf`: It's like a pop, but with floating-point registers
- `call`: Like a `jal`, it's a relative jump, and it saves the return address in R31. The difference with the jal is that a call exploits the windowed register file, so after a call, the *CWP* will change.
- `ret`: It's similar to a `jr`, but it exploits the windowed register file.

General Instruction		R-type Instruction		F-type Instruction	
Mnemonics	Opcode	Mnemonic	Func	Mnemonic	Func
j	0x02	sll	0x04	addf	0x00
jal	0x03	srl	0x06	subf	0x01
beqz	0x04	sra	0x07	multf	0x02
bnez	0x05	add	0x20	mult	0x0E
addi	0x08	addu	0x21	multu	0x16
addui	0x09	sub	0x22		
subi	0x0A	subu	0x23		
subui	0x0B	and	0x24		
andi	0x0C	or	0x25		
ori	0x0D	xor	0x26		
xori	0x0E	seq	0x28		
lhi	0x0F	sne	0x29		
rfe	0x10	slt	0x2A		
jr	0x12	sgt	0x2B		
jalr	0x13	sle	0x2C		
slli	0x14	sge	0x2D		
nop	0x15	sltu	0x3A		
srli	0x16	sgtu	0x3B		
srai	0x17	sleu	0x3C		
seqi	0x18	sgeu	0x3D		
snei	0x19				
slti	0x1A				
sgti	0x1B				
slei	0x1C				
sgei	0x1D				
push	0x1E				
pop	0x1F				
lb	0x20				
lh	0x21				
lw	0x23				
lbu	0x24				
lhu	0x25				
lf	0x26				
sb	0x28				
sh	0x29				
sw	0x2B				
pushf	0x2C				
popf	0x2D				
sf	0x2E				
call	0x30				
ret	0x31				
movfp2i	0x32				
movi2fp	0x33				
cvtf21	0x34				
cvti2f	0x35				
sgtui	0x3B				
sleui	0x3C				
sgeui	0x3D				

Table A.1: Instruction set

APPENDIX B

Compiler

The compiler script `dlxasm.pl` has been extended and updated in order to support all new instructions and features included in the DLX processor. The script now generates two binary files, one with the IRAM content and another one with the DRAM content. The two files must be converted into `ASCII` text files, using the conversion tools `conv2iram` and `conv2dram`.
Since the IRAM is word-addressable, the IRAM file stores 4 bytes per line in hexadecimal format. The DRAM is byte-addressable, hence the DRAM file stores one byte per line.
The two output files are then copied to the DLX simulation directory and read by the ram entities to fill the memory locations.
The source file must have a defined structure, with different sections delimited by *compiler directives*. Exception and interrupt routines are supported and shall be placed into the source file. It is also possible to instantiate data bytes or words into the Data memory.
A generic source file will include the following sections:

- *Configuration* section, to generate the correct configuration word;
- *Exception and Interrupt routines* section (optional), to define exception handlers;
- *Text* section, which will include the main program;
- *Data* section (optional), to instantiate some data into the DRAM;

B.1 Configuration section

The compiler allows to specify different options, used to generate a configuration word that will be placed into the first address of the IRAM. This word is loaded into the Status Register of the CPU during the start-up phase. Each configuration option is not mandatory (if not specified, a default value will be used to generate the configuration word). If present, they must be placed at the beginning of the source file. The following options are available:

- `.mode std|fst|dbg`: it defines the CPU execution mode, which will be set as soon as the configuration word leaves the decode stage. It is possible to choose between *standard*, *fast* and *debug* mode. For further information about CPU modes, see Chapter 2. If not specified, the compiler uses `standard` as default value.
- `.ecp full|no_arith|disabled`: it defines the exception mode. *full* allows to trigger any exception, *no_arith* disables the detection of arithmetic exceptions and *disabled* allows no exception to be detected. For further information about exceptions, see Chapter 6. If not specified, the compiler uses `full` as default value.

- `.stack 0|1`: this option allows to enable stack automatic handling and protection. If enabled, the DLX reserves last DRAM addresses for stack space and prevents any instruction from accessing those locations, apart from `push|pushf` and `pop|popf`. If disabled, no space will be reserved for stack. `push|pushf` and `pop|popf` can still be used, but last addresses of DRAM will be available for any instruction. If not specified, the compiler uses `0` as default value.
- `.bds 0|1`: this option allows to use the *Branch Delay Slot*. If enabled, the DLX will always execute the instruction following any jump or branch instruction, instead of flushing it. It is important to notice that the compiler does not perform any reorder of instructions to exploit the Branch Delay Slot option. If active, it is necessary to modify source code manually. If not specified, the compiler uses `0` as default value.

B.2 Exception and Interrupt routines

It is possible to define routines for exceptions and interrupts, in order to handle them in a correct way. Up to 10 routines are available, each of them with a maximum of 4 instructions. The compiler directive used to start a routine is `.rX`, where `X` is the routine index, a number between 0 and 9 which identifies the exception or interrupt associated to the routine. Routines are placed by the compiler at a specified IRAM address, according to the routine index. They are not mandatory, but if present must be placed after configuration section and before text section. If more than one routine is declared, they must appear in ascending routine index order.
The following figure specifies the association between routine index and exception or interrupts:

B.3 Text section

It starts with the directive `.text` and it is followed by main program instructions. It is possible to define some labels, in order to simplify jump instructions. A label can be any sequence of characters, followed by a colon (for instance, `start:`).

B.4 Data section

It starts with the directive `.data` and allows to declare some data in DRAM space. The following directives are available:

- `.byte`
- `.word`
- `.float`

After each directive it is possible to declare an arbitrary number of values, separated by a comma. Since DRAM is byte-addressable, in order to avoid memory alignment problems it is possible to use `.space` or `.align` directives.
Using `.space X` will leave `X` empty bytes after last declared data, while `.align X` will move to the next address with last `X` bits at 0.

B.5 Comments

Comments can be placed at any point in the source code, using a semicolon.

The following is a source file example that uses all sections:

```
.mode fst
.ecp full
.stack 1
.bds 0

.r4
    subi r0,r24,#1
    rfe

.text
    addi r0,r24,1026
    lw r1, 0(r0)
    end:
    j end

.data
    .word 255
```

Code B.1: example code