Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Computer Engineering

Master degree in Electronics Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group_16

Battilana Matteo, La Greca Salvatore Gabriele, Pollo Giovanni

July 9, 2021

Feature

- Frequency - Slack - Area - Ecc

Grandes nacelles :

- Nacelle A318 PW
- Inverseur A320 CFM
- Inverseur A340 CFM
- Nacelle A340 TRENT
- Inverseur A330 TRENT
- Nacelles A380 TRENT900
- Nacelles A380 GP7200

Petites nacelles :

- Nacelle SAAB2000
- Inverseur DC8
- Inverseur CF34-8
- Inverseur BR710
- Nacelle F7X

Contents

Listings

CHAPTER 1

Introduction

1.1 Abstract

The goal of this project is to build from scratch a working implementation of a DLX. In order to achieve the goal, some known blocks, created during laboratories, were used.

The second step was the design of the datapath, done in the best possible way to obtain a high optimization and performance level. Some optimization examples that will be explained more in depth in the document are the use of the P4 adder and the Booth multiplier inside the ALU, the comparator and many others.

The third step was the design of the control unit. The choice fell on the microprogrammed version that guaranteed the pipeline implementation. In order to simplify the maintainability of the control unit, some struct-like constructs were used in the VHDL code.

In the fourth step, a very exhaustive testing has been executed. All the proposed asm codes were verified, but some well-known algorithm (bubble sort, Fibonacci and factorial) were written and tested.

Last but not least, the DLX has been synthetized using synopsis, and a post-synthesiss simulation has been executed.

Thanks to the datapath optimization and the synthesis optimization, the microprocessor proposed in this paper reached a peak speed of 400MHz.

1.2 Workflow

As many tools we used to automate the working process, all of them are explained in this section.

The first and most important tool was versioning control. The choice fell on GitHub. Thanks to this, the team managed all versions of the code and stepped back if any problem occurs. In addition to that, team communication and issue management were very straightforward, thanks to the possibilities offered by the tool. Another handy feature was the milestone, which allows the team to be on time and respect deadlines.

The used programming technique was the pair programming, that allows to write code and checking its correctness at the same time. This technique was particularly useful in difficult part of the projects. To exploit pair programming the extension used was Live Share for Visual Studio Code (`https://github.com/MicrosoftDocs/live-share`). Indeed, for the easier steps, the use of the branches and pull request gave the possibility of parallel working and drastically reduced the presence of conflicts.

The last thing to point out was the intensive use of scripting for compling VHDL code, simulating it, adding wave and then synthetizing the full project. All this scripts are reported in the appendix.

CHAPTER 2

Hardware Architecture

2.1 Overview

This DLX is a 32-bit RISC processor with a five-stage pipeline. The external interface is made mainly for memories connection (IRAM, DRAM and a DRAM for the Register File), and for the Clock and Reset signals. Inside we find the following blocks:

- Control Unit: it receives the fetched instruction from the IR register and starts to output the correct control signals towards all the pipeline stages. Moreover, it receives status signals from all other units about their working status like the comparator result (for branch decision), the status about possible hazards in the pipeline, Register File's Push & Pop operations under execution, and all the memories readiness. It's in charge of controling the entire pipeline and stop it in case of hazards or other situations that requires a stall.
- Decode Unit: part of the decode stage, it is in charge of keeping the status about all registers under use (for further hazard controls), computation of the new Program Counter (given a Jump or not), data comparison (for branches) and, the most important thing, the operation decode with the dispatch of all the operands towards the right ports of the DataPath.
- DataPath: the computational core of the processor. Made of 4 pipeline stages (Instruction Decode, Execution, Memory, Write Back) contains all the units capable of doing computation. In particular, we have the Register File (that manages all the registers of the core), the Arithmetic Logic Unit, the Load-Store Unit for data memory management, and other units useful for the correct operation of everything.
- IR and PC: two registers the compose the Instruction Fetch stage of the pipeline, they are in charge of keeping in memory the current instruction under execution and the address for the next instruction to execute, respectively.

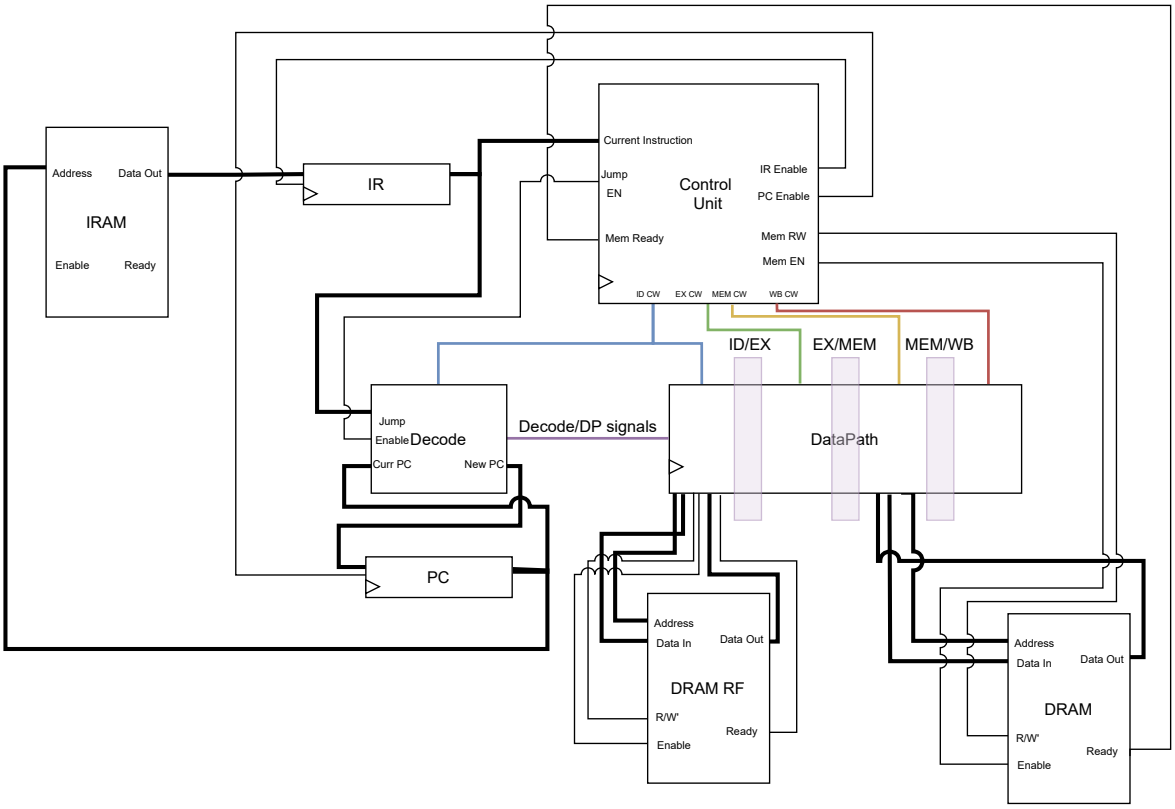


Figure 2.1: Schematic of the DLX

2.2 Pipeline Stages

2.3 Control Unit

The Control Unit is one of the most important part of the DLX processor. Its role is orchestrate all the jobs through the pipeline of the processor and to manage dangerous situations.

Its interface is made of input signals (status signals from other components) and of control signals (towards other components), as described in Table 2.1 and Table 2.2.

Signal Name	**Description**
IR_IN	Fetched Instruction, output from the IR Register
HAZARD_SIG	The Decode Stage is signaling a Hazard situation
BUSY_WINDOW	The Current RF Window has some registers in use
SPILL	The Register File has started a push operations towards the memory
FILL	The Register File has started a pop operations from the memory
IRAM_READY	The Instruction Memory has a data ready as output
LGET	Comparison status from the comparator inside the Decode Unit
DRAM_READY	Indicates if the DRAM is ready or not

Table 2.1: Input signals towards the Control Unit

Signal Name	Description
PIPLIN_IF_EN	IF Stage enable: enable of IR register
IF_STALL	IF Stage stall: a NOP is insterted in the IR register
PC_EN	Enable of the PC register
JUMP_EN	A JUMP must occur, the Decode Stage will compute the new PC
CALL	The Register File must start the context switch in a new window
RET	The Register File must restore the previous window
SEL_CMPB	Register or Immediate field as comparator input in the Decode Stage
UNSIGNED_ID	Indicates that all arithmetic units must consider operands as unsigned
NPC_SEL	Selects as new PC between the value of a REG NPC Adder's output
HAZARD_TABLE_WR1	Enables the logging of the current instruction in the hazard control
RF_RD1_EN	Enables the Read Port 1 in the RF
RF_RD2_EN	Enables the Read Port 2 in the RF
PIPLIN_ID_EN	ID Stage enable: enables all the ID pipeline REGs
PIPLIN_EX_EN	EX Stage enable: enables all the EX pipeline REGs
SEL_LGET	Used by the SET Comparator unit for selection of the SET operand
DRAM_WE	Enables the Write Operation in the DRAM
DRAM_RE	Enables the Read Operation in the DRAM
DRAM_ME	Enable signal for the DRAM
DATA_SIZE	Indicates the data of transfer towards/from the DRAM
UNSIG_SIGN_N	Same as *UNSIGNED_ID* but 2 clock cycles delayed
PIPLIN_MEM_EN	MEM Stage enable: enables all the MEM pipeline REGs
WB_MUX_SEL	Selects between MEM output and ALU out the content to write back
PIPLIN_WB_EN	WB Stage enable: enables the Write signal in the RF

Table 2.2: Output signals towards the Control Unit

2.3.1 Internal organization of the Control Unit

The Control Unit internally is organized mainly with a stage that converts the actual Instruction into a Control Word and then this control word is propagated in order to support all the pipeline stages, as shown in Figure 2.2.

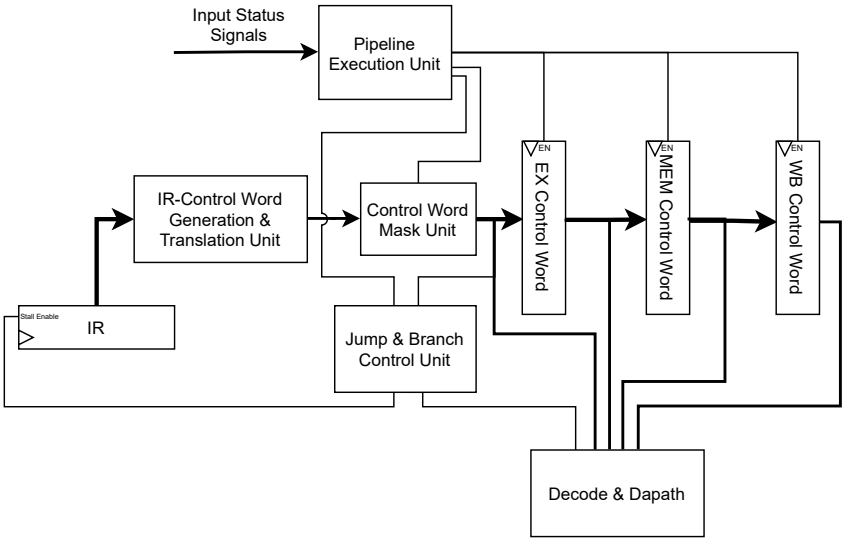


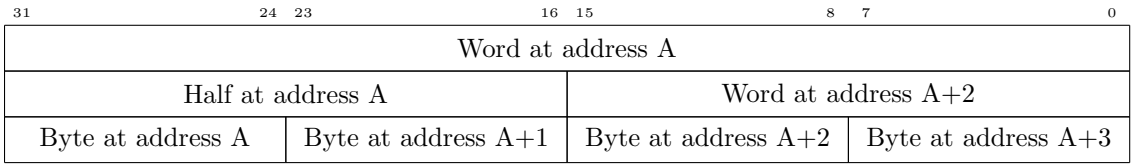
Figure 2.2: Schematic of the DLX Control Unit

It is made mainly by the following components:

- IR-Control Word Generation & Translation Unit: its role is to take the Instruction given as input and to output a valid Control Word related to that instruction.
- Pipeline Execution Unit: it's in charge of controlling the Pipeline Flow and stop it in case of needs and/or masking the actual Control Word in order to stop its propagation through the chain in case of Hazards.
- Control Word Mask Unit: when the *Pipeline Execution Unit* signals to mask the control word, this unit will do it by disabling all the bits in the CW that referees to the stage enabling signals.
- Jump & Branch Control unit: given the Control Word and other internal control signals, its job is to manage (by enabling or disabling them) signals like *CALL*, *RET*, *JUMP_EN* and *IF_STALL*.

2.4 Memory Interface

The DLX processor has a Harvard architecture, with two 32-bit data bus carrying instructions and data respectively. Only load and store instructions can access data from Data Memory. The data are stored in memory in a Big-Endian format.



The third RAM required, the one for the register file, is not under the direct access of the User. It's managed as a Stack memory by the Register File for automatic sub routines management and it has a dedicated interface. It can be merged with the Data Memory with an external logic.

All the subsequent paragraph are referred in the same way to all the three memory types.

2.4.1 Signals and Timing

The signals in the DLX processor bus interface can be grouped into tree categories:

- Address class signals
- Memory Request signals
- Data class signals

The *Address class signals* are:

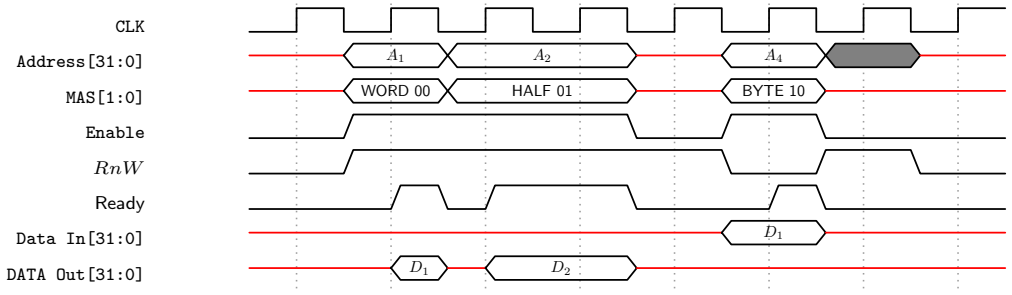
- A[31:0]
- DATA_SIZE[1:0] *or MAS[1:0]*

The *Memory Request signals* are:

- Enable

- $R\overline{W}$
- Ready

Moreover, all the memories connected must agree on a certain protocol both for writing and reading operations. The most important thing to take under consideration is the *Ready* signal: it must be high only when the operation is really completed. For example after a data read, the *Ready* stays at 1 only when the data is valid. If meanwhile the address changes, the *Ready* signal must go off.



If the memory in use can't accomplish to this timing, an external *Memory Control Unit* must be placed between the CPU and the Memory.

2.4.2 Memory Addressing

A[31:0] is the 32-bit address bus that specifies the address for the transfer. All addresses are byte addresses, so a burst of word accesses results in the address bus incrementing by four for each cycle.

The address bus provides 4GB of linear addressing space, and this can be used externally in different manners like in a SoC with memory mapped peripherals that shares the same address space.

When a word access is signaled the memory system ignores the bottom two bits, A[1:0], and when a halfword access is signaled the memory system ignores the bottom bit, A[0]. Howewer, the core already masks the two LSBs when needed.

2.4.3 Memory Data Size: the MAS[1:0] signal

The *MAS[1:0]* bus encodes the size of the transfer. The DLX processor can transfer word, halfword, and byte quantities and the processor indicates the size of the transfer through this signal.

When a halfword or byte read is performed, a 32-bit memory system can return the complete 32-bit word, and the processor extracts the valid halfword or byte field from it as shown in Table 2.3. For 8 and 16 bit memories, the data must be placed on the right byte lanes in the data bus.

DATA_SIZE[1:0]	A[1:0]	D[31:0]	DLX Register
00 WORD	00	0xAABBCCDD	0xAABBCCDD
01 HALF	00	0xAABB----	0x0000AABB
01 HALF	10	0x----CCDD	0x0000CCDD
10 BYTE	00	0xAA------	0x000000AA
10 BYTE	01	0x--BB----	0x000000BB
0 BYTE	10	0x----CC--	0x000000CC
10 BYTE	11	0x------DD	0x000000DD

Table 2.3: How data are read by the CPU in different MAS configurations

Instead, when the DLX processor performs a byte or halfword write, the data being written is replicated across the data bus, as shown in Figure 2.3. The memory system can use the most convenient copy of the data.

A writable memory system must be capable of performing a write to any single byte in the memory system. This is required for the correct working of the DLX.

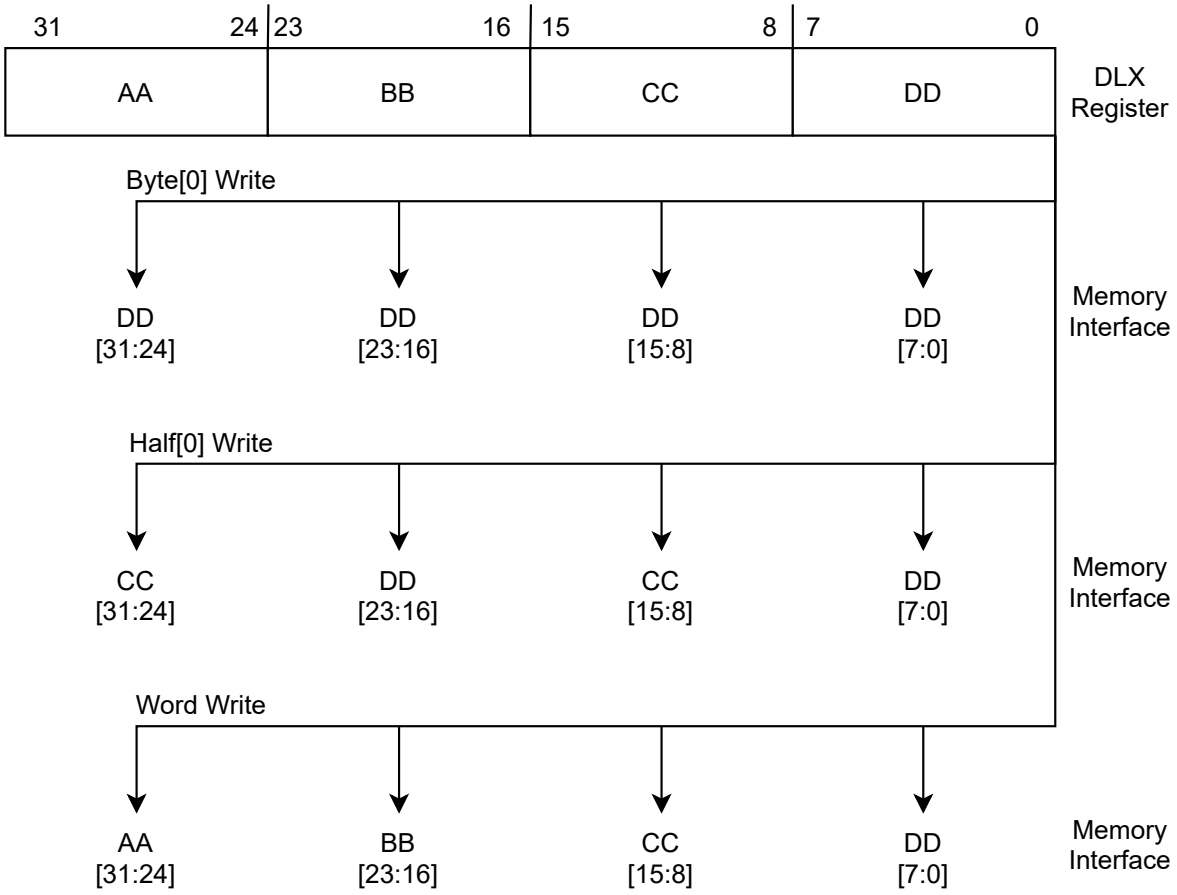


Figure 2.3: Data Write Replication

2.5 Instruction Set

The Instruction Set supported by the DLX is made of different instructions, in particual regarding Integer operations. Instructions are on 32 bit and are grouped in 3 different types:

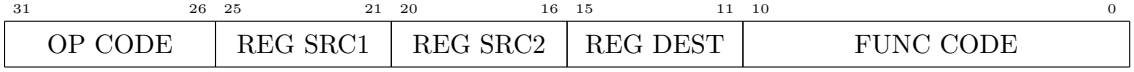


Figure 2.4: R-Type

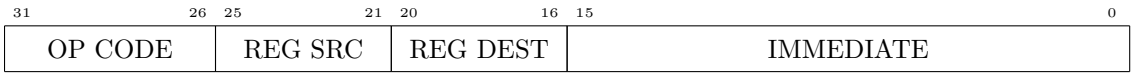


Figure 2.5: I-Type

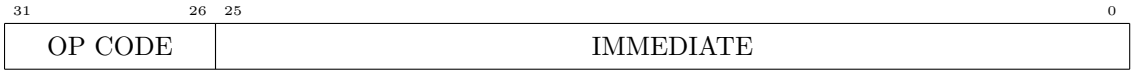


Figure 2.6: J-Type

In particular, all of them have in common the *OP CODE* field, used to identify the instruction.

2.5.1 R-Type Instructions

R-Type Instructions are called in this way because all the operands are between registers. They all have in common the *OP CODE* equal to `0b00000` and each instruction is differentiated from another one thanks to the *FUNC CODE* field, as shown in Table 2.4.

MNEMONIC	FUNC CODE	EXAMPLE	OPERATION	DESCRIPTION
SLL	`0x04`	SLL R3, R2, R1	$R3 \leftarrow R2 \ll R1$	Logical shift left
SRL	`0x06`	SRL R3, R2, R1	$R3 \leftarrow R2 \gg R1$	Logical shift right
ROR	`0x08`	ROR R3, R2, R1	$R3 \leftarrow R2 \circlearrowright R1$	Right rotation
ROL	`0x09`	ROL R3, R2, R1	$R3 \leftarrow R2 \circlearrowleft R1$	Left rotation
MUL	`0x0E`	MUL R3, R2, R1	$R3 \leftarrow R2 * R1$	Integer multiplication
ADD	`0x20`	ADD R3, R2, R1	$R3 \leftarrow R2 + R1$	Integer Add
ADDU	`0x21`	ADDU R3, R2, R1	$R3 \leftarrow R2 + R1$	Integer Add
SUB	`0x22`	SUB R3, R2, R1	$R3 \leftarrow R2 - R1$	Integer subtraction
SUBU	`0x23`	SUBU R3, R2, R1	$R3 \leftarrow R2 - R1$	Integer subtraction
AND	`0x24`	AND R3, R2, R1	$R3 \leftarrow R2 \& R1$	Bitwise AND
OR	`0x25`	OR R3, R2, R1	$R3 \leftarrow R2 \mid R1$	Bitwise OR
XOR	`0x26`	XOR R3, R2, R1	$R3 \leftarrow R2 \oplus R1$	Bitwise XOR

Table 2.4: R-TYPE Instructions: logical and arithmetic

MNEMONIC	FUNC CODE	EXAMPLE	OPERATION	DESCRIPTION
SEQ	0x28	SEQ R3, R2, R1	$R3 \leftarrow R2 == R1$	1 if R2 eqaul R1

Table 2.5: R-TYPE Instructions: test instructions

2.5.2 J-Type Instructions

J-Type Instructions are groups of instructions made for .

MNEMONIC	FUNC CODE	EXAMPLE	OPERATION	DESCRIPTION

Table 2.6: J-TYPE Instructions

CHAPTER 3

Fetch Stage

3.1 Instruction Register

3.2 Program Counter

3.3 Jump and Branch Management

CHAPTER 4

Decode Stage

4.1 Instruction Decode

4.2 Register File and Windowing

The general structure of a register file is based on a decoder that takes the selection input (so the address of the desired register) and enables it (using also the enable signal). At this point, an input signal will contain the value to be written. On the other hand, a read signal is used to select among all the registers.

The DLX presented in this document has been enhanced in order to be able to manage subroutine in a transparent manner from the point of view of the user. For this reason, the DLX must be able to handle subroutines, and so the context switching, that consists in saving the registers content in order to be restored once the procedure has been completed. The straightforward solution is to save into the memory all registers but this is not feasible in terms of delay, since for 32 registers we will need 32 clock cycles; if you image this in a pipeline, this corresponds to a long stall each time a procedure is called.

A windowed register allows to reduce the overhead due to the context switch; the basic idea is to split the available registers in the physical register file into blocks, called *windows*. We have limited amount of physical registers in the register file, for this reason a finite number of windows are defined. Each window is assigned to a subroutine, so that the procedure can write only on those register. This is transparent from the point of view of the CPU, that sees all registers available. Thus, the physical register file has a wrapper around it with a logic and a Register Management Logic (MML) that allows to perform the translation between the CPU requests to corresponding window for the running procedure.

What if the number of called procedure is larger than the number of available windows? The main memory is involved only when there are no free windows in the register file. In this case, the oldest allocated window is swapped into the main memory, so that the new one can be allocated. Obviously, once all the recursion chain has been unrolled, the swapped window in the memory must be restored into the register file. All windows, so each procedure, has 4 blocks of 8 registers each one:

1. **IN**: the first block is dedicated to the data inherited from the parent routine (OUT);
2. **LOCALS**: contains the registers that are dedicated to the procedure;
3. **OUT**: is dedicated to the variables to be passed to the child routine, that is the IN of the next

sub-procedure

4. **GLOBAL**: the last block is common to every windows.

When a procedure is called, the first LOCALS and OUT blocks are allocated from the physical file register and assigned to it (because IN is the OUT of the previous one).

By calling many nested procedures, at some point there will be no free windows; for this reason the oldest is de-allocated from the physical FR and swapped to the main memory, the operation is called **SPILL**. This it accomplished by using a support pointer, called **Saved Window Pointer SWP** that stores the point of the spilled data, exactly the end of the LOCALS block (only IN and LOCAL are spilled, the OUT block is not spilled because is the IN of the next sub-procedure). In practice it define the position of the last free cell. Notice that this operation cannot be executed in one clock cycle: each register is spilled once at a clock cycle.

On the other hand, when the last procedure in the chain is finished, the other are unrolled; if some of them have been spilled, a **FILL** must be executed before the actual execution. This can be achieved by, firstly decrement CWP by 16 and check if now CWP >> SWP.

It's important to notice that the implementation of the entire register file has been implemented in Structural. Is is composed by several components:

- **Decoder**: it is used to generate a single enable signal from a signal on **NBIT_ADD** bits; in this way, a register is selected in order to perform a write. The register will check also if a write is requested;
- **Connection matrix**: this block allows to "highlight" the active windows, the block IN, LOCAL and OUT will be the default destination when writing and reading;
- **Register file**: this block corresponds to the physical registers, composed by rows of Flip-Flops;
- **Select block**: this block is used for the reading, is connected to all the registers and selects, using the read address, the single register to be read;
- **Address generator**: this block is used only when perform a FILL or a SPILL, it generates the address for the registers to be moved from/to the memory. The memory, in this case works exactly like a stack.

Additional, but less complex components, have been used in order to manage the management of the CWP and SWP.

4.2.1 Decoder

This block receives as input the *write address* on **NBIT_ADD** bits and outputs $\mathbf{2^{NBIT\_ADD} - 1}$ bits. It has the utility of converting the address of the register at which we need to write into its enable signal.

The idea is that if the input is $0b00010$ the output will be $0b00000000000000000000000000000100$. In fact if the input is decimal 2, it means that we need to write the second register of the *GLOBAL* block. In terms of enable it can be translated by having the bit with index 2 at one. In fact in the output we see that the it with index 2 has value 1, while the others are all 0.

The output is divided (in the schematic) in order to represent the group of bits. In particular we have that:

- M - 1 DOWNTO 0: bits associated to the *GLOBAL* register
- M + N - 1 DOWNTO M: bits associated to the *IN* register

- M + 2N - 1 DOWNTO M + N: bits associated to the *LOCAL* register
- M + 3N - 1 DOWNTO M + 2N: bits associated to the *OUT* register

On the top of the schematic (Figure 4.1) we can see an AND logic port between *ENABLE* and *WR* signals. If both *ENABLE* and *WR* are 1, it means that our register need to work. In fact, the output of the dedocer is anded with 1 and so we maintain the value. Otherwise, if one signal between *ENABLE* and *WR* is 0, the output will be 0 and so the AND with the output of the *decoder* will return all 0.

This signal goes into the *connection matrix*, which is the next block described.

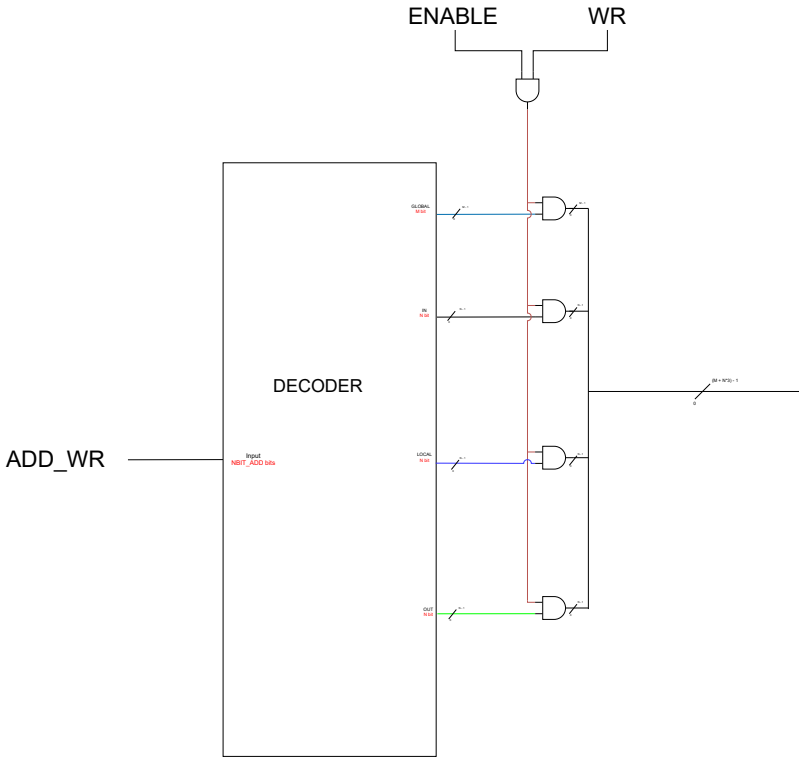


Figure 4.1: Schematic of the Decoder

4.2.2 Connection Matrix

With the previous block, we generated all our enable signals. The problem is that we have more windows. So how do we decide which window needs to be activated? Here comes the connection matrix. This block receives as inputs the signal coming from the decoder, the current window, the saved window and the address for the pop (fill) operation. The output is a signal that contains the enable signals ready for all the registers of all windows.

We have a specific structure for each block:

- GLOBAL: the global is the simplest, because it is connected directly to the output
- IN: for this block we AND the IN bits coming from the decoder with the bit (that is extended) of the related window. For example if we are evaluating the IN of the first window, we will AND the IN bits with the bit 0 of the current window.

- OUT: for this block we AND the OUT bits coming from the decoder with the bit (that is extended) of the previous related window. For example if we are evaluating the OUT of the first window, we will AND the OUT bits with the bit 4 of the current window (supposing our window has 5 bits).
- LOCAL: for this block we AND the LOCAL bits coming from the decoder with the bit (that is extended) of the related window. For example if we are evaluating the LOCAL of the first window, we will AND the LOCAL bits with the bit 0 of the current window.

For the IN and OUT we then an OR between the two outputs (the logic can be seen in the schematic), while for the LOCAL we don't have anything.

In addition to that, the connection matrix also manages the saved window, used for the pop (fill) operation. First we need to invert the addr_pop, because when we execute the pop operation, we restore data starting from the last one (we are using a STACK). The addr_pop_inverted is composed like this:

- 2N - 1 DOWNTO 0: we have the IN bits
- N - 1 DOWNTO 0: we have the LOCAL bits

The signal is splitted into two wires and is anded with the saved related saved window pointer.

In the end, we definitely OR the output of the previously described OR with the output of this AND. This is visible in Figure 4.2

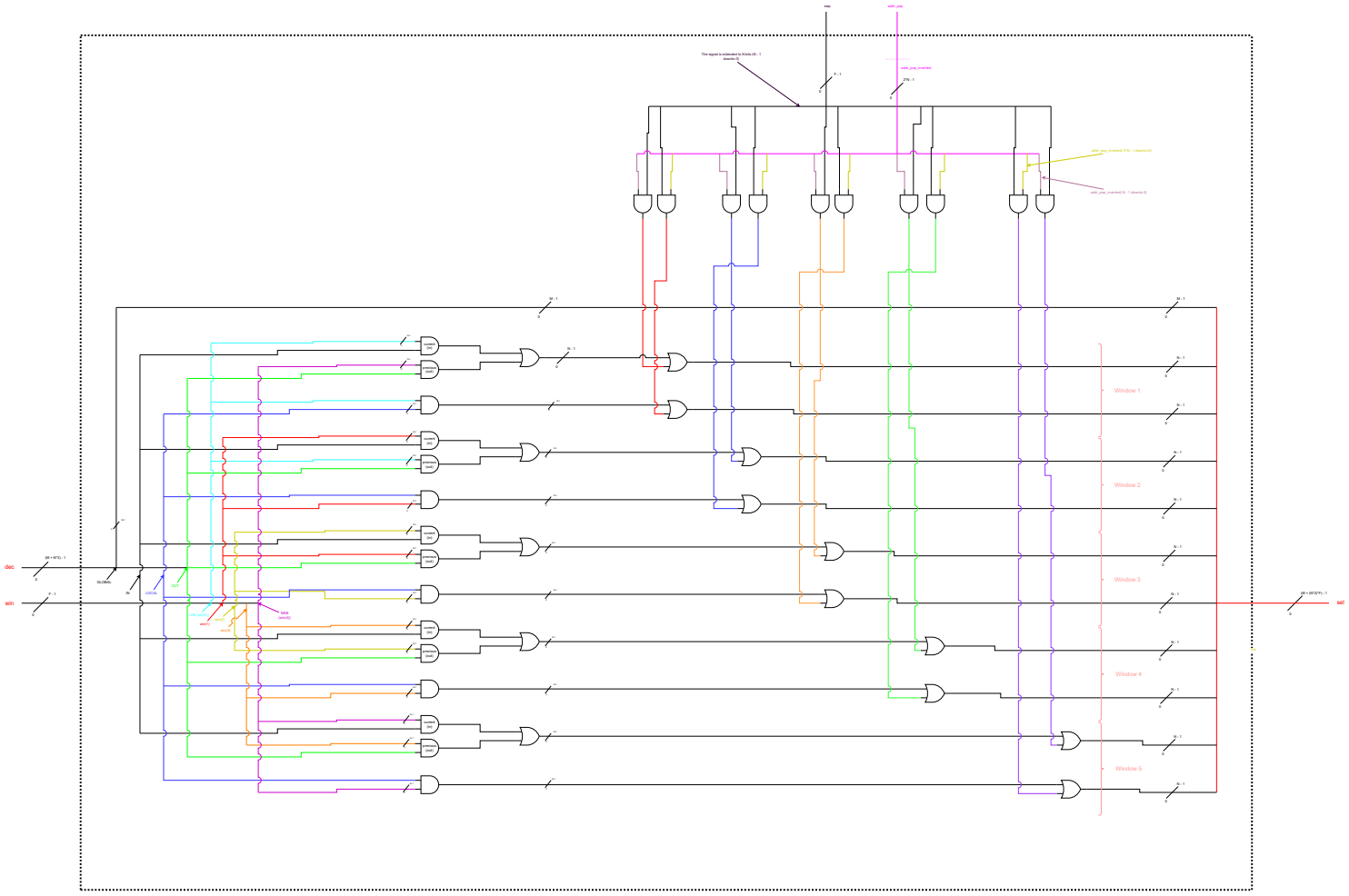


Figure 4.2: Connection matrix

4.2.3 Register File

The next block is the Register File, that is a sequence of registers. The important thing to notice in our design is how we managed the data that goes into the registers. We have two choices, data_in and from_mem. In order to choose we decided to use multiplexers. We have a multiplexer for each window. The signal used to drive the multiplexer is the saved window pointer, rotated right by 1 position and anded with the pop signal. In fact, we select from_mem only when the pop signal is 1, otherwise we need to select data_in. We use the saved window pointer shifted right by 1 because when the saved window pointer is, for example 00010 we need to restore the window 00001.

Indeed, for dataout there are no multiplexers, because there is no choice.

4.2.4 Select Block

This block is very simple and straightforward. It receives as input the current window, and the output of the register file (of all windows). The it selects the bit of the IN, LOCAL, OUT of the current window. The interface is shown in Figure 4.3.

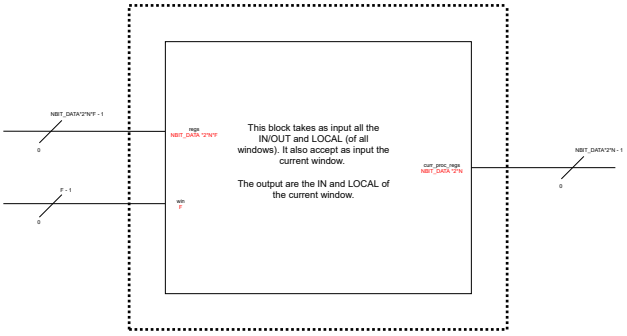


Figure 4.3: Interface of the select block

4.2.5 Output Selection

This is the stage that decide the two output Data1_Out and Data2_Out. The design is shown in Figure 4.4.

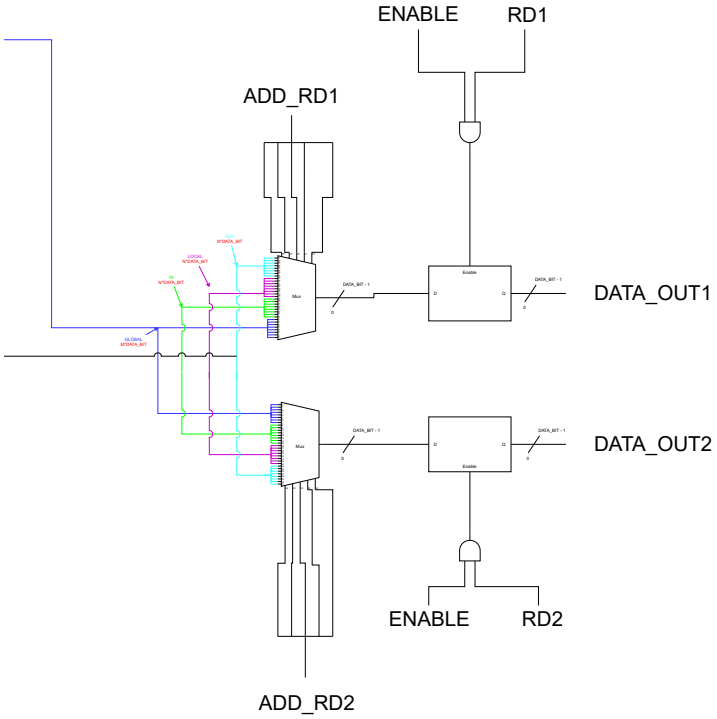


Figure 4.4: Design of the output selection

This stage receives the IN, LOCAL, OUT of the current window, thanks to the select block and the GLOBAL. The two addresses, ADD_RD1 and ADD_RD2, select the output of the multiplexer which goes into the register, used to respect the timing. The Enable of each register is the and of the ENABLE and the RD signal. The decision was made in order to stop reading when the circuit is not enabled, and so to have a granural and precise control of the circuit.

4.2.6 Next Window Calculator

This block is used to compute the next window, both for the current window and the saved window. The schematic is shown in Figure 4.5.

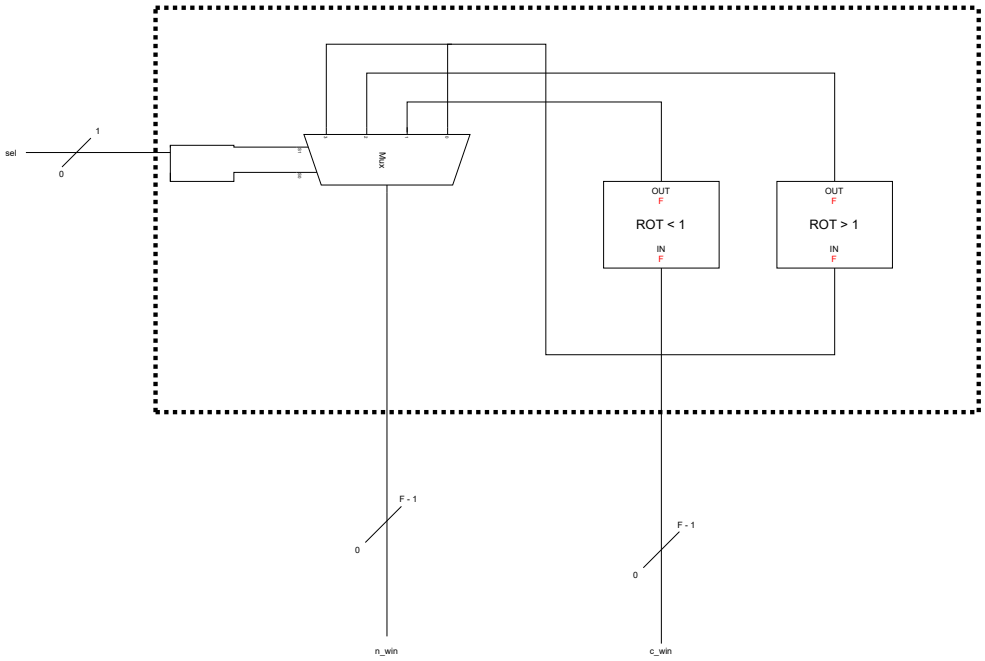


Figure 4.5: Design of the next window calculator

Inside this block there is a logic able to rotate right or left and a multiplexer, that allows to select the correct output based on what the circuit needs.

4.3 Hazard Control

4.4 Comparator

The straightforward way to implement a comparator, allows only to check if two operands, `A` and `B`, are equals. The solution is sketched at figure 4.6, that is based on N XNOR, where N is the number of bits of the operands and an AND gate with N inputs.

Even if this solution is extremely compact, it allows to perform only the equality comparison; since this DLX implementation has the ability to perform complex conditional branch instructions (refer to the Instruction section 2.5) and conditional set instructions (refer to the Set-Like Operations Unit

??) we need a more complex solution. In fact, if we need to perform a jump only if `A` > `B` (strictly greater) we need to check exactly this precise condition.

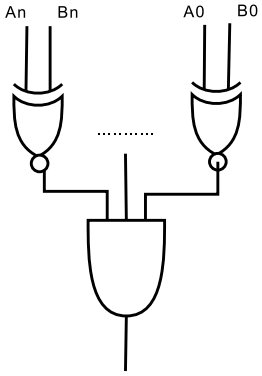


Figure 4.6: Design of the basic comparator

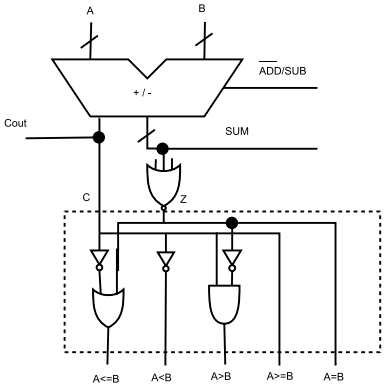


Figure 4.7: Design of the advanced comparator

The advance comparator exploits the comparison by performing a subtraction between `A` and `B` and then checking the result. This DLX implementation is based on a P4 adder that is able to perform subtraction and then a set of checks, the same that are in the 4.7 are performed in order to generate the comparison outputs. We can perform the comparisons using this boolean equations, where C is the carry-out and Z is the zero check (all bits of the results are zeros):

$$A > B \rightarrow C \cdot \overline{Z}$$
$$A \geq B \rightarrow C$$
$$A < B \rightarrow \overline{C}$$
$$A \leq B \rightarrow \overline{C} + Z$$
$$A = B \rightarrow Z$$
$$A \neq B \rightarrow \overline{Z}$$

In order to avoid to propagate six different signal, the outcomes of the comparisons are encoded into a signal `LGET` on two bits. The encoded value are the ones in the 4.1 table.

LGET	Case
01	$A < B$
00	$A \leq B$
11	$A > B$
10	$A \geq B$

Table 4.1: LEQ encoding

```
1  LGET <= "01" when (a_l_b = '1') else
2    "00" when (a_le_b = '1') else
3    "11" when (a_g_b = '1') else
4    "10" when (a_ge_b = '1') else
5    "00";
6
```

Listing 4.1: VHDL code for the encodig

The ordering of the comparison in the `when` statement is not casual nor follows the normal patterns but, the strictly lower comparison is done before the lower equals one, because, if the latter one is true it means that is also `A` lower than `B` but not vice-versa. Using this encoding, we can simply check the second bit in order to understand if `A` $\leq$ `B` or `A` $\geq$ `B`; instead, if we want to check only $<$ or $>$ comparisons we have to check also the first bit.

A further improvement has been done to the advanced comparator in order to manage comparison between both signed and unsigned numbers. The carry value works like this:

- Carry = 1: if $A > B$ in unsigned
- Carry = 0: if $A \leq B$ in unsigned

The advanced comparator works with unsigned numbers only. So it simply needs to be adapted for cases in which signed comparison and unsigned comparison are different. They are shown in the Table 4.2 highlighted in red. It is easy to notice that in the red lines A and B always have different sign. The logic must work when the UNSIG_SIGN_N bit is 0, that means the circuit is dealing with a signed number. In this case the carry bit must be complemented. Knowing this things, it's easy to derive the following logic:

```
i_cout_masked <= Cout xor (not(UNSIG_SIGN_N) and (A(A'length-1) xor B(B'length-1)));
```

A	B	Carry out	Signed comparison	Unsigned comparison
2	3	0	Less	Less
4	3	1	Greater	Greater
3	3	1	Equal	Equal
-3	3	1	Less	Greater
-2	3	1	Less	Greater
-5	3	1	Greater	Greater
3	2	1	Greater	Greater
3	4	0	Less	Less
3	3	1	Equal	Equal
3	-3	0	Greater	Less
3	-2	0	Greater	Less
3	-5	0	Greater	Less
2	-3	0	Greater	Less
4	-3	1	Greater	Less
3	-3	1	Greater	Less
-3	-3	1	Equal	Equal
-2	-3	1	Greater	Greater
-5	-3	1	Less	Less
-3	2	1	Less	Greater
-3	4	0	Less	Greater
-3	3	1	Less	Greater
-3	-3	0	Equal	Equal
-3	-2	0	Less	Less
-3	-5	0	Greater	Greater

Table 4.2: All cases of possible comparison

So, we the final unit corresponds to the one at figure 4.7 with an additional encoder before the output, that encode the six conditions into a signal on two bits. The resulting schema is the one in figure 4.8.

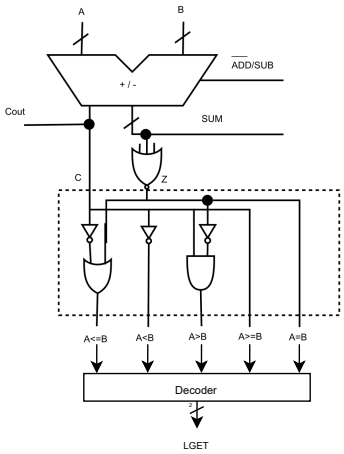


Figure 4.8: Final implementation of the comparator

4.5 Jump and Branch decision

4.6 Next Program Counter computation

CHAPTER 5

Execute Stage

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

5.1 ALU: Arithmetic Logic Unit

The Arithmetic Logic Unit can be seen as a block, that given a selection signal and the inputs is able to perform computation over the operands. The ALU implementation described in this document is based on the following block:

- Adder
- Multiplier
- Logic unit
- Shifter

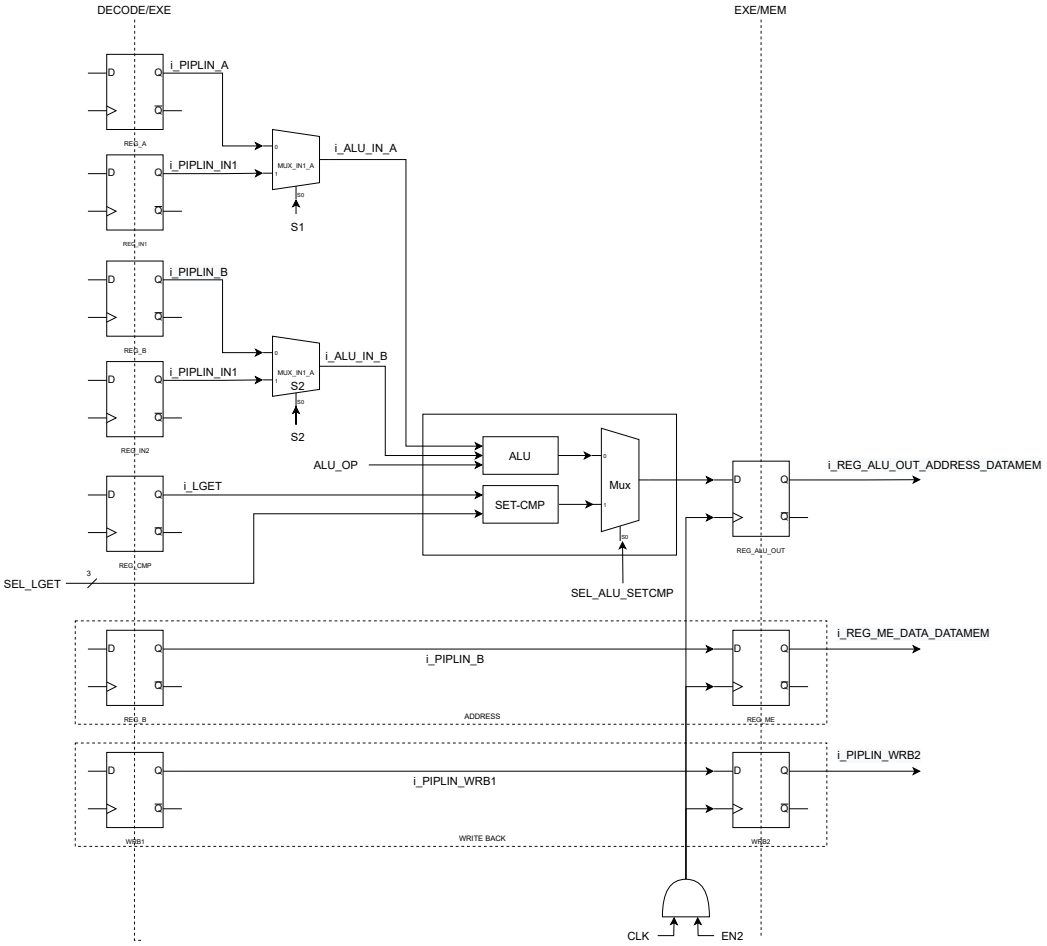


Figure 5.1: Execute stage

Each one of these blocks will be explained in the following sections.

The base concept is that, internally, the 4 units are selected through a multiplexer that takes two out five bits from a selection signal called `OP`. Having 5 bits to describe the type of operation, the possible combinations and their relative operations are:

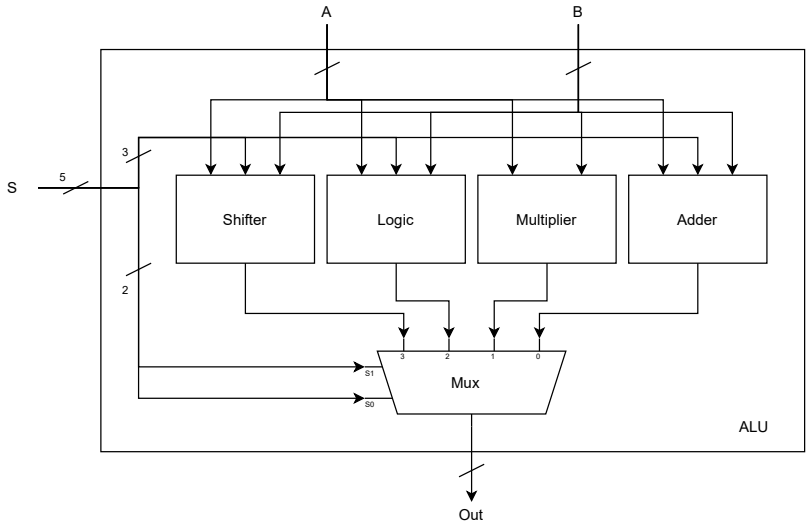


Figure 5.2: ALU unit

OP	Unit	Operation
000 00	ADD	ADD
001 00		SUB
000 01	MUL	MUL
000 10	LOGIC	AND
001 10		NAND
010 10		OR
011 10		NOR
100 10		XOR
101 10		XNOR
000 11	SHIFT	SHIFT RIGHT
001 11		SHIFT LEFT
010 110		ARITH SHIFT RIGHT
011 11		ARITH SHIFT LEFT
100 11		ROTATE RIGHT
101 11		ROTATE LEFT

Table 5.1: ALU operations encoding

The two LSBs are the ones used as selection input for the multiplexer that select from which ALU unit takes the result. In fact, the univocally define the unit to be used. The remaining three MSBs are used as input for the units that compose the ALU in order to select the correct operation.

5.1.1 Adder

The straightforward way to implement an adder is to use the Ripple Carry Adder structure, that is composed of $N-1$ Full Adder and one Half Adder (the first), where N is the number of bits of the two operands. This solution is not optimal from a timing point of view due to the time needed to propagate the carry, that defines the critical path, that is the bottleneck.

Since the sum and the subtraction is one of the most common operation, the DLX includes an adder

that is based on a CLA - Carry Look Ahead (Sparse Tree) and a Carry Select Like Adder. The complete structure can be seen at figure 5.3.

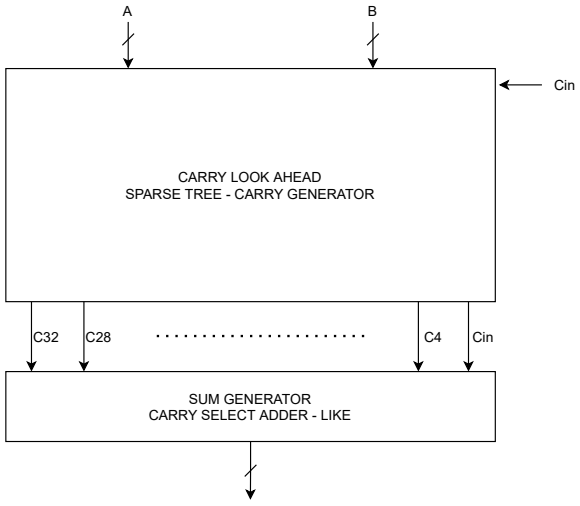


Figure 5.3: Booth's multiplier on 32 bits

As said before, the adder is composed of two blocks:

- **Carry Select Like Adder**: The main point of the Carry Select Adder is that it doubles the complexity of the adder itself in order to obtain better performances. It is composed by two RCA, in order to perform two sums in parallel.

  The idea is to compute both the results, on 4 bits in this case, for both when the carry-in is equal to '0' or '1'. In this way, the results is computed in parallel for all the stages, even if the carry-in is '0' or '1'; then the carry-in is used to mux against the two results (on 4 bits) and the two carry-outs. The carry-out will be used as result selection signal for the next Carry Select unit.

  We are paying complexity in order to reduce the addition computation time, in fact by having a carry out that is used as carry in for the next state, there is still propagation but is lower.

  The DLX implementation, instead of using a straightforward implementation of the Carry Select Adder, it uses a modified version of it. It has been implemented using *CLA - Sparse Tree Carry Generator* and a *Carry Select Like Adder*. The base idea is to use the CLA in order to compute a carry every n bits, then these carries are fed into the sum generator that uses them to compute the results in parallel.

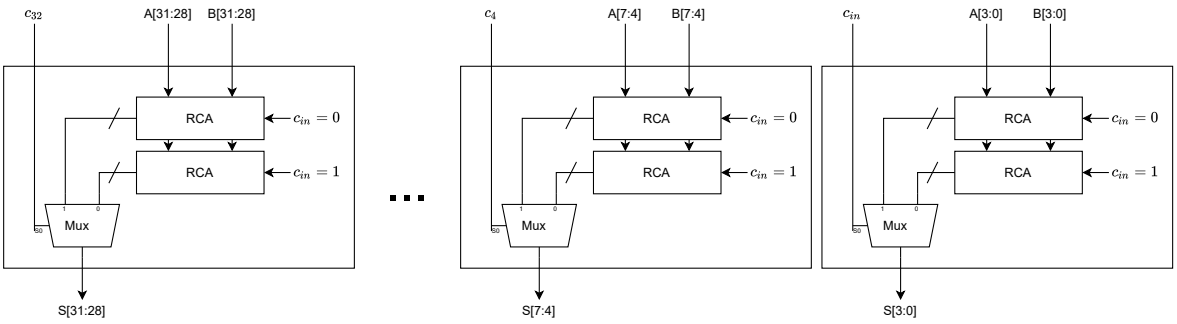


Figure 5.4: Carry Select Like Adder block for a 32 bits implementation

- **Carry Look Ahead - Sparse Tree**: this block is used to compute the carry out every 4 bits. The idea behind the CLA is to compute several carries simultaneously and to avoid waiting until the correct carry propagates from the stage of the adder in which it has been generated. This is done thanks to the *propagate (P)* (that is 1 if the carry-in is equal to the carry-out) and *generate (G)* (that is one if carry-in is 0 and carry-out is 1).

a	b	c_{in}	out	c_{out}	p	g
0	0	0	0	0	0	0
0	0	1	1	0	0	0
0	1	0	1	0	1	0
0	1	1	0	1	1	0
1	0	0	1	0	1	0
1	0	1	0	1	1	0
1	1	0	0	1	0	1
1	1	1	1	1	0	1

Table 5.2: Computation of propagate and generate bits

$$g = a \oplus b \qquad p = a \cdot b \tag{5.1}$$

The base idea is to write any s_i, that is the i-esim bit of the sum and c_i, the carry-out at i index in function of p and g. We now use p and g to express the same result:

$$s_1 = a \oplus b \oplus c_{in} = p_1 \oplus c_0$$
$$c_1 = a \cdot b + a \cdot c + b \cdot c = a_1 \cdot b_1 + (a_1 + b_1) \cdot c_0 = g_1 + p_1 \cdot c_0$$

The crucial point is that it's possible to compute the carry at i position only using the initial carry-in c_{in} and p and g generate in the current and previous blocks. Tree are a family of Carry Look Ahead that differ for the carry-logic. They are based always on *propagate* and *generate*. We have that

$$carry = g + p \cdot c_{in}$$
$$G_{i:j} = G_{i:k} + P_{i:k} \cdot G_{k-1:j}$$
$$P_{i:j} = P_{i:k} \cdot P_{k-1:j}$$

where

- $i \geq k > j$
- $G_{x:x} = g_x$ and $P_{x:x} = p_x$
- $g_0 = C_{in}$

The white and gray blocks in the Sparse Tree block at 5.6, that are used in the `carry_generator` block, are PG and G blocks. Normally two blocks are used, the first G generates only $G_{i:j}$ and the other PG both $G_{i:j}$ and $P_{i:j}$. The base idea is to combine their outputs and take only the G one as carries.

So, Carry Look Ahead - Sparse Tree needs a starting block that generates all the p and g for all the couples of bit using the 5.1 equation. The spars tree and the PG network structure is shown in the figure 5.6, in the code this block is called `prop_gen_generic` and is made of `prop_gen` block in order to compute p and g.

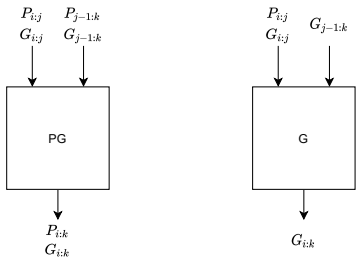


Figure 5.5: PG and P block

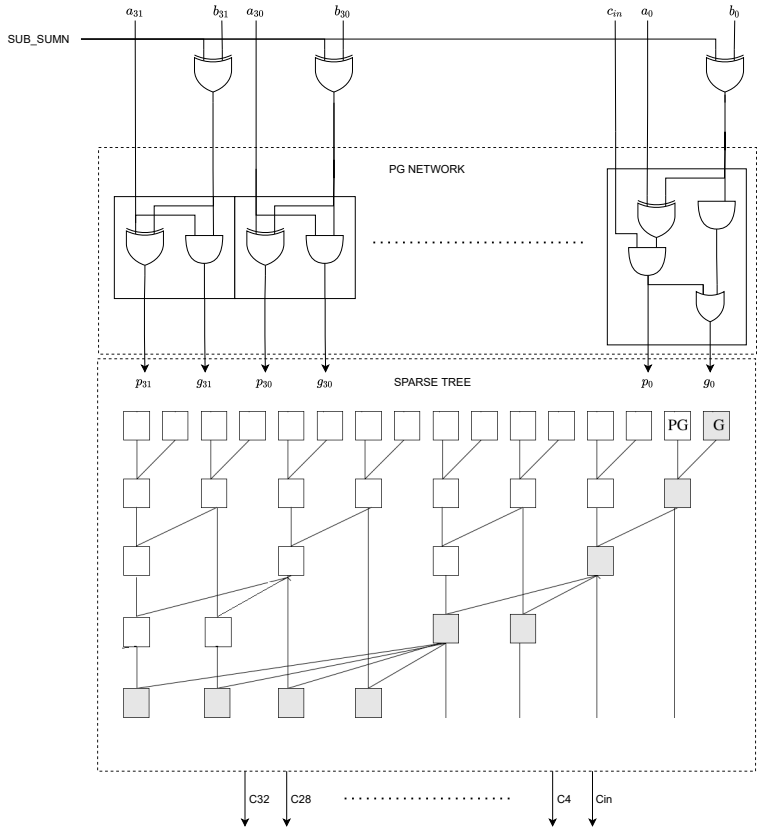


Figure 5.6: Carry Look Ahead - Carry Generator Block

In order to be able to perform both subtraction and sum, the first block must be modified and must include the logic to manage also the carry-in, that in case of a subtraction it is '1'. This is not enough to perform subtraction, in fact, an additional signal called `SUB_SUMN` is needed. So, the same structure can be used to implement the subtraction, by adding an XOR on each B input with the `SUB_SUMN` control signal.

We can say that a subtraction in 2's complement can be implemented as $A + \overline{B} + 1$; in order to implement this in the circuit we need to set the carry-in to '1' (so `SUB_SUMN` = '1') in order to add 1

and invert the B input by using the XOR. In fact:

x	y	$y \oplus q$
0	0	0
0	1	1
1	0	1
1	1	0

This solution is good because the PG and G block has the same delay, that is driven by G and since both include it, they are the same. Many paths have the same delay and the load on components is good since an output of a block is connected at maximum to 2 other block. These two factors bring a good *equilibrium* to the entire structure.

5.1.2 Multiplier

In order to overcome the limitation of the array multiplier, this DLX implementation includes a modified version of the Booth's multiplier, since the multiplex for the partial values to be added is only on two bits instead of three. The Booth's algorithm copes with 3 bits at a time, so the number of stages is $N/2$ (this corresponds to the number of the encoders) and this allows to speed up the result computation. The Booth's algorithm is the following:

```
i = 0
P = 0
while i ≤ M − 2 loop
        P = P + Vp( B_{i+1}, B_i, B_{i−1} )
        A = A ∗ 4
        i = i + 2
end loop
```

Where `P` is the final value of the product and during the algorithm execution it will contain the partial result; `M` represents the number of bit of the multiplicand, in this case B. The algorithm takes as convention that $B_{i-1} = 0$. The `Vp` is a lookup table (see 5.3), that return the value to add to `P`, according to the 3 bits selected. The value of A is multiplied by 4.

B_{i+1}	B_i	B_{i-1}	
0	0	0	0
0	0	1	+A
0	1	0	+A
0	1	1	+2A
1	0	0	-2A
1	0	1	-A
1	1	0	-A
1	1	1	0

Table 5.3: Booth's LUT

The Booth's multiplier work with 3 main components, supposing A is the multiplicand and B the multiplier:

1. $N/2$ encoders in order to take 3 bits from the operand B; the two LSB are used as selection signal for the multiplexers and the last one, the MSB, for the adder. In fact, as it's possible to notice from the table 5.3, when the MSB is 1 the value to be added to the partial result is

positive, negative otherwise. For this reason, the inputs to the multiplexers are only $\{0, A, 2A\}$. Since we need also to generate negative values, like $-A, -2A$, the MSB of the three bits is used as input for the adder. This signal, called `SUB_SUMn` is used to define if the operation is a sum or a subtraction; if it is 1, a subtraction is performed;

2. $N/2$ multiplexers that select only among $\{0, A, 2A\}$, since at each stage A must be multiplied by 4, a shift by two is done starting from A of the previous multiplexer;

3. $N/2$ ripple carry adders, that allow to preform the partial sums. Since the final results will be on $NBIT \cdot 2 + 1$ bits, the adders in each level have been optimized in order to work only with the minimum bits needed. In fact, the adder at i level, will generate the result on $NBIT + 2 \cdot i$ bits. As said before, an addition signal called `SUB_SUMn` has been added in order to be able to perform the subtraction. The Ripple Carry Adder has been select for its simplicity and since the multiplication is a less common instruction, it was not worth to use a more sophisticated adder. This allowed to reduce the total area of the multiplier itself.

The multiplexer implements the LUT and at the same time the $A = A \cdot 4$. The two values from the two multiplexer are summed together via an adder, this implements the partial sum. The overall structure can be observed at 5.7.

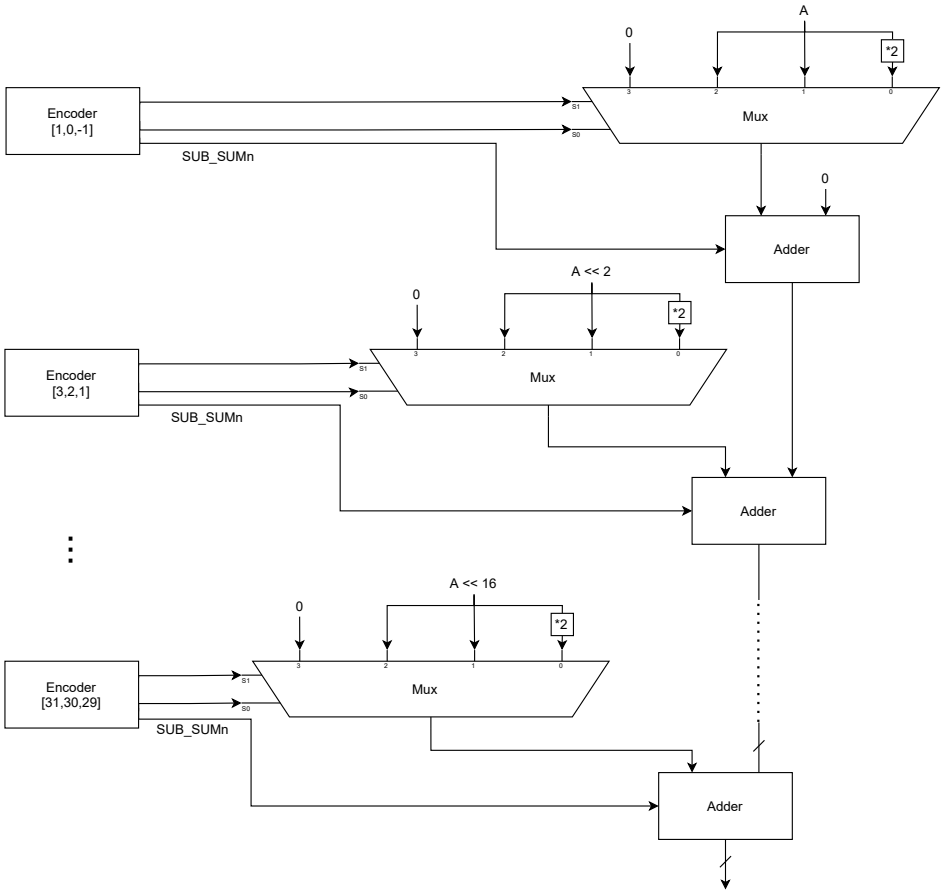


Figure 5.7: Booth's multiplier on 32 bits

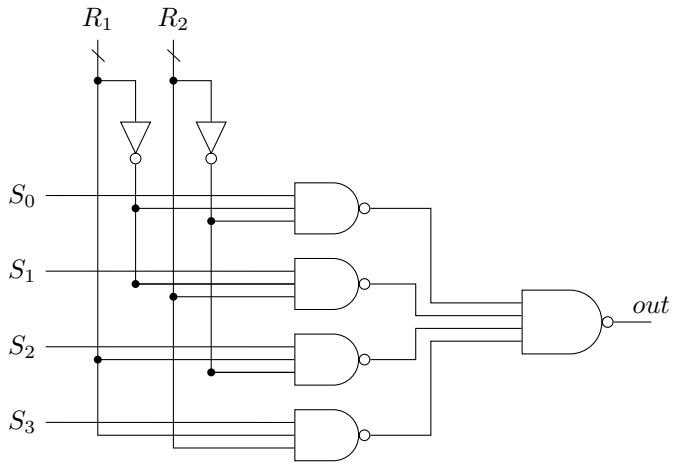


Figure 5.8: Logic unit

5.1.3 Logic Operands

The basic and most simple implementation of a logic unit is based on single logic gates on N bits whose outputs are muxed, in order to generate the correct output. The problem with this solution is that the number of input signals to the multiplexer is extremely high; this implementation does not only suffer from the point of view of the delay but, since each logic function is implemented with a specific gate, the total area is huge.

In order to overcome the problems highlighted before, a more compact implementation has been chosen: the T2 logic unit.

This logic unit allows to perform AND, NAND, OR, NOR, XOR and XNOR using only 5 NAND gates, on two levels, and 4 selection signals. The schematic is the one in figure 5.8.

In order to compute one of the logical instructions, the select signals are properly activated as follow:

S_0	S_1	S_2	S_3	Operation
0	0	0	1	AND
1	1	1	0	$NAND$
0	1	1	1	OR
1	0	0	0	NOR
0	1	1	0	XOR
1	0	0	1	$NXOR$

Figure 5.9: Logic input signals with the relative operation

For example, in order to generate the AND logical operation, we have to select $S_3 = 1$, so that $out = R_1 \cdot R_2$; on the other hand, if we need NAND $S_0 = S_1 = S_2 = 1$ and $S_3 = 0$, so that $out = \overline{R_1} \cdot \overline{R_2} + \overline{R_1} \cdot R_2 + R_1 \cdot \overline{R_2} = \overline{R_1} \cdot \overline{R_2}$ that using the De Morgan law $out = \overline{R_1 \cdot R_2}$. This allows to obtain the best performances also because all paths work in parallel, compacting the area and the delay.

Since only the 3 bits are used to select among the logical operations (`S` signal), a direct correspondance is needed to generate the signal show in table 5.9. The following table, shows the conversion:

S	Decoded signal
000	0001
001	1110
010	0111
011	1000
100	0110
101	1001
110	0000

Table 5.4: Conversion table, from S input signal on 3 bits into 4 bits

5.1.4 Shifter

The implemented shifter allows to perform shift right, logical/arithmetical shift left and left/right rotate using the full operand A on 32 bits and 6 bits from the second one B and three *control signals*. Differently from the T2 version, it uses and addition signal in order to be able to manage also the rotate instruction. Our implementation takes three inputs:

- A: the operand to be shifted/rotated;
- B: only the 5 LSB [4,3,2,1,0] are used to select first the mask to be used and then the starting point from that mask;
- SEL: it encodes the operation type; the second bit is used to select among arithmetic and logic, the third bit is used to select the direction of the shift/rotate (left/right) and the first one is used only if the operation is a rotate. This is the encoding:

SEL	**Operation**
000	Shift logic right
001	Shift logic left
010	Shift arith right
011	Shift arith left
100	Rotate right
101	Shift right

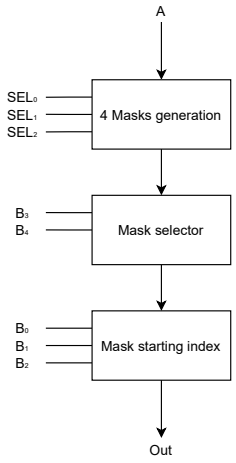


Figure 5.10: Blocks of the Shifter/Rotate Unit

The unit perform the requested operation in three stages, sketched in figure 5.10:

1. The first consist in preparing 4 possible "masks", each already shifted of 0, 8, 16, 32 left or right depending on the configuration. This allows to shift for all 32 bits. Basically it copies the input `A` into the 4 masks that will be used by the next stage. Being in 32 bits, the generated masks are in $32+8=40$ bits. The only different between this implementation and the T2 one, is that, in case of rotate, the additional 8 bits of the masks are filled with the corresponding 8 bits that are going "out" during the rotation.

2. The second level perform a coarse grain shift, that is basically consist on selecting one mask among the 4 possible masks generated in the previous stage. This selection is done by using the bits 4, 3 of `B`.

3. The third level, using the bits 2, 1, 0 of `B` and the selected mask, preform a fine grain refinement. The 3 bits allows to select the starting index from the mask, in fact it allows to select among 8 positions.

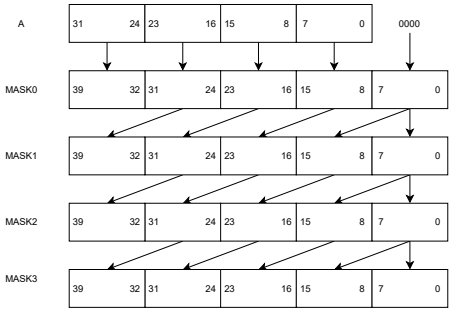


Figure 5.11: Masks for left shift

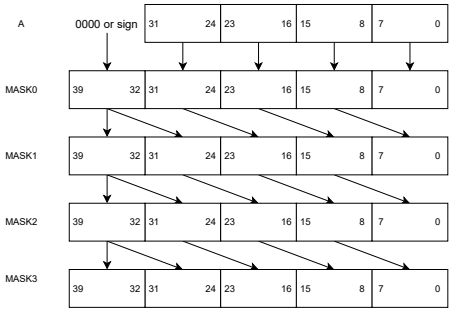


Figure 5.12: Masks for right shift

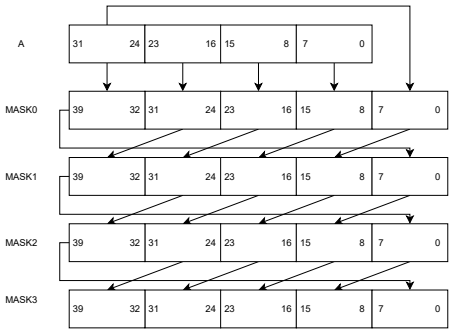


Figure 5.13: Masks for left rotate

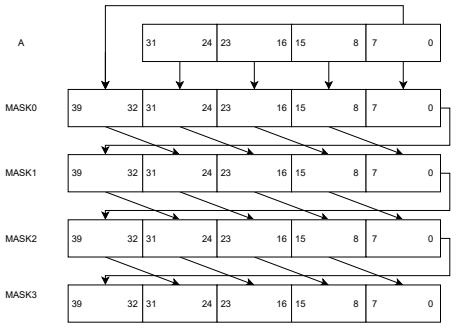


Figure 5.14: Masks for right rotate

Figure 5.15: Masks for shift unit on 32 bits

Examples
For example, if we need to perform a left left of 9 bits `A`, where `A=18`, the corresponding `B` value will be 1001; this means that the second masks will be taken and the output result will from the bit at position $40 - 1 = 39$ to the one at $39 - 32 = 7$ included.

$$\text{MASK 2: } 0\underbrace{\mathbf{0000000\ 00000000\ 00000000\ 00010010\ 0}}_{\text{shifted A}}0000000$$

On the other hand, if we need to perform a right shift the masks are generated in the opposite way, so the zeros are put in the MSB of the mask, shifted by 0, 8 ... positions. In this case we need also to distinguish between the an arithmetic and a logic shift; in the first case, instead of filling the "empty" bits with zero, the operand sign is used. For example, if we want to shift `A=-18` of `B=3` bits, the first mask is used:

$$\text{MASK 1: } 11111\underbrace{\mathbf{111\ 11111111\ 11111111\ 11111111\ 11101}}_{\text{shifted A}}110$$

In the last case, let's suppose to rotate right `A=1255` (=10011100111) by 5 position:

$$\text{MASK 1: } 11\underbrace{\mathbf{100111\ 00000000\ 00000000\ 0000100\ 111}}_{\text{rotated A}}00111$$

As you can see, in case of MASK 1 for the right rotation, the 8 LSB of `A` are copied into the 8 MSB of the mask.

5.2 Set-Like Operations Unit

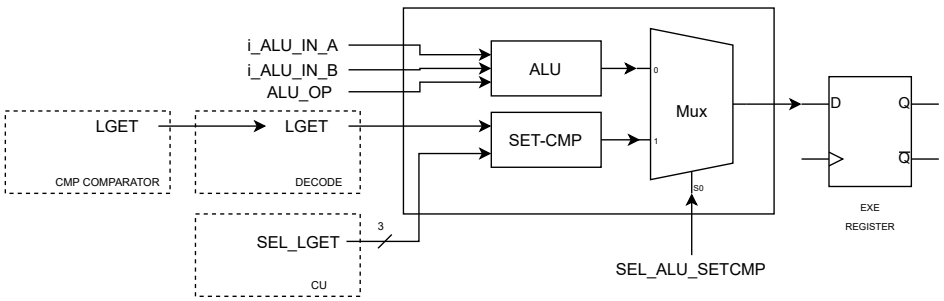


Figure 5.16: Execute stage with focus on the SET-CMP unit

The DLX execution stage includes also the possibility to set the value of a register accordingly to the outcome of the comparison of two operands; the operands can came from two sources:

1. Both from the register file, trough `REG_B` and `REG_A` (refer to figure 5.1);
2. One from the register file (`REG_A` by setting `S1 = 0`) and the immediate from `REG_IN2` (refer to figure 5.1).

The unit designed to perform this set operation is called `set_comparator`, that using a behavioural process, is able to generate the corresponding '1' or '0' to be set to the register, accordingly to the comparison result. The output on $NBIT_DATA$ bits will be muxed with the one coming from the ALU unit, using the CW that is configured in the Control Unit.

In order to decrease the area and since the comparison was already generated by the comparator unit in the decode stage (refer to section 4.4), the `set_comparator` unit takes the `LGET` signal that came from the decode unit and perform the following checks:

Operation	**VHDL implementation**
SEQ	`not LGET(0)`
SNE	`LGET(0)`
SLE	`LGET(1) = '0' or LGET(0) = '0'`
SLT	`LGET = "01"`
SGE	`LGET(1) = '1' or LGET(0) = '0'`
SGT	`LGET = "11"`

Table 5.5: Performed checks in order to generate '1' or '0' accordingly to the comparison outcome. Refer to table 4.1.

CHAPTER 6

Memory Stage

6.1 Load-Store Unit

- Unsigned things

6.2 Address Mask Unit

CHAPTER 7

Write Back Stage

Mux selects from Memory Output (LoadStore Unit) or ALU output.

Signal to enable register file write. Registers to delay the write register address

CHAPTER 8

Testing and Verification

8.1 Test Benches

8.2 Simulation

8.3 Post Synthesis Simulation

CHAPTER 9

Physical Design

9.1 Synthesis

9.2 Place and Route

CHAPTER 10

Conclusions