



**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 3, 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

Feature	i
1 Introduction	1
1.1 Abstract	1
1.2 Workflow	1
2 Hardware Architecture	2
2.1 Overview	2
2.2 Pipeline Stages	3
2.3 Control Unit	3
2.3.1 Internal organization of the Control Unit	4
2.4 Memory Interface	5
2.4.1 Signals and Timing	5
2.4.2 Memory Addressing	6
2.4.3 Memory Data Size: the MAS[1:0] signal	6
2.5 Instruction Set	7
3 Fetch Stage	8
3.1 Instruction Register	8
3.2 Program Counter	8
3.3 Jump and Branch Management	8
4 Decode Stage	9
4.1 Instruction Decode	9
4.2 Register File and Windowing	9
4.2.1 Decoder	10
4.2.2 Connection Matrix	11
4.2.3 Register File	13
4.2.4 Select Block	14
4.2.5 Output Selection	14
4.3 Hazard Control	15
4.4 Comparator	15
4.5 Jump and Branch decision	15
4.6 Next Program Counter computation	15
5 Execute Stage	16
5.1 ALU: Arithmetic Logic Unit	16
5.1.1 Adder	16
5.1.2 Multiplier	16

5.1.3	Logic Operands	16
5.1.4	Shifting	17
5.2	Set-Like Operations unit	18
6	Memory Stage	19
6.1	Load-Store Unit	19
6.2	Address Mask Unit	19
7	Write Back Stage	20
8	Testing and Verification	21
8.1	Test Benches	21
8.2	Simulation	21
8.3	Post Synthesis Simulation	21
9	Physical Design	22
9.1	Synthesis	22
9.2	Place and Route	22
10	Conclusions	23

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 synthesized using synopsis, and a post-synthesis 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 compiling VHDL code, simulating it, adding wave and then synthesizing 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 controlling 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	Fetch 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 <i>UNSIGNED_ID</i> 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.

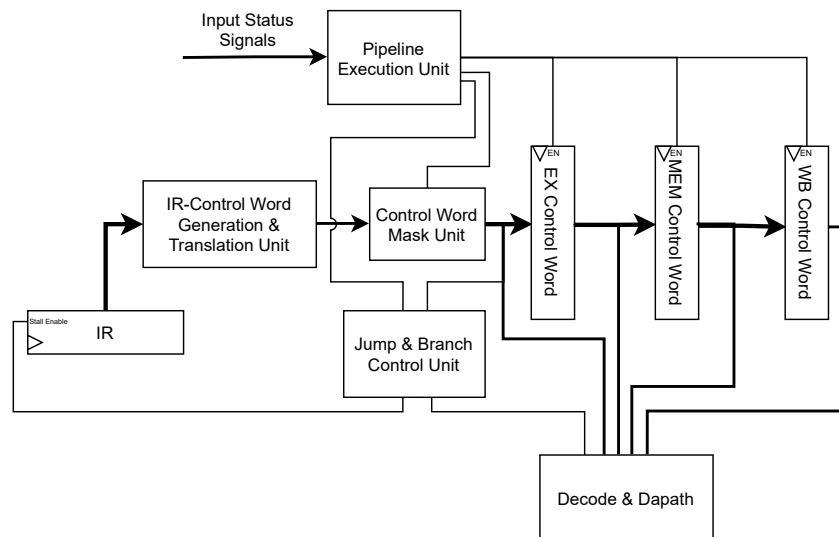


Figure 2.2: Schematic of the DLX Control Unit

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.

31	24	23	16	15	8	7	0
Word at address A							
Half at address A				Word at address A+2			
Byte at address A		Byte at address A+1		Byte at address A+2		Byte at address A+3	

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:

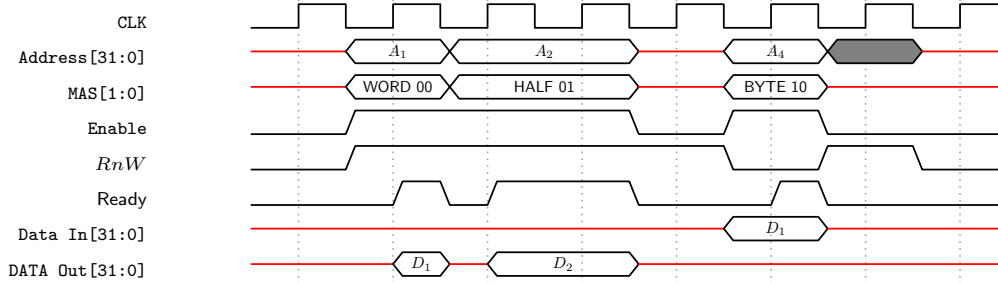
- A[31:0]
- DATA.SIZE[1:0] or MAS[1:0]

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, $A[0]$. However, 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
10 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.

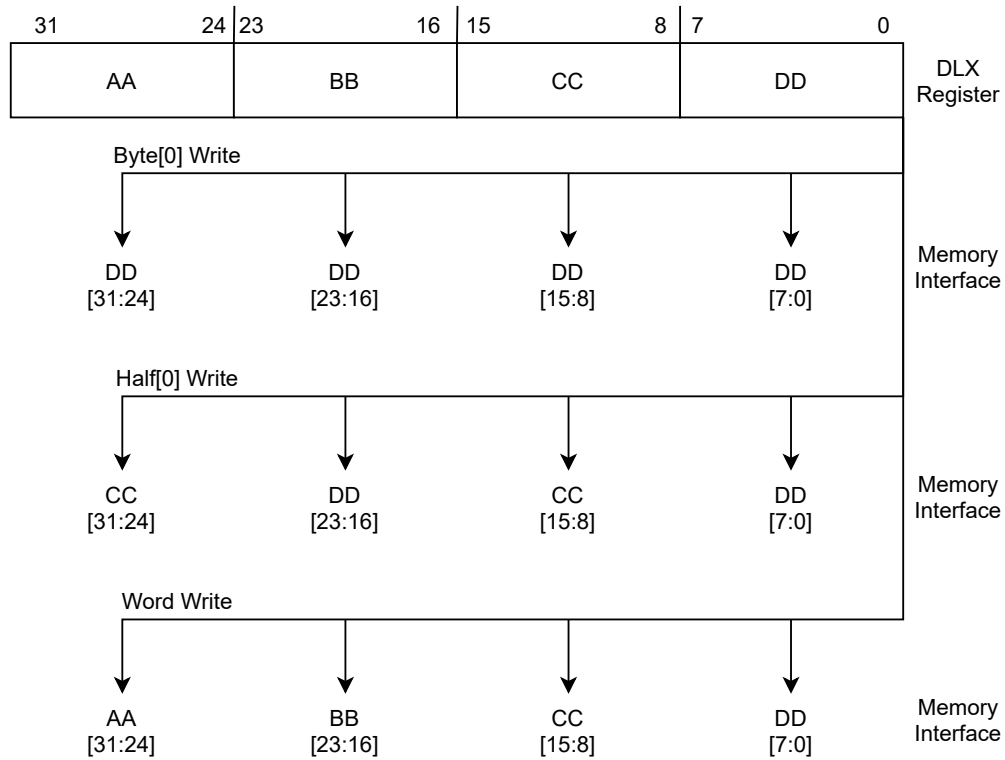


Figure 2.3: Data Write Replication

2.5 Instruction Set

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

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

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 is 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 it is the IN of the next sub-procedure). In practice it defines 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.

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.

4.2.1 Decoder

[illegible]

- $M - 1$ DOWNT0: bits associated to the *GLOBAL* register
- $M + N - 1$ DOWNT0 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.

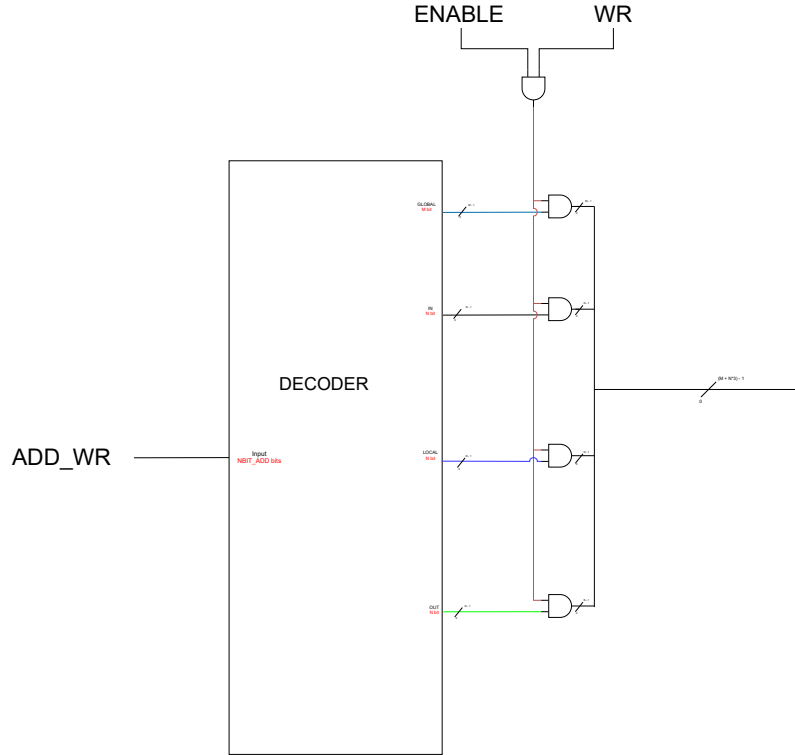


Figure 4.1: Schematic of the Decoder

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 DOWNT0 0`: we have the IN bits
- `N - 1 DOWNT0 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

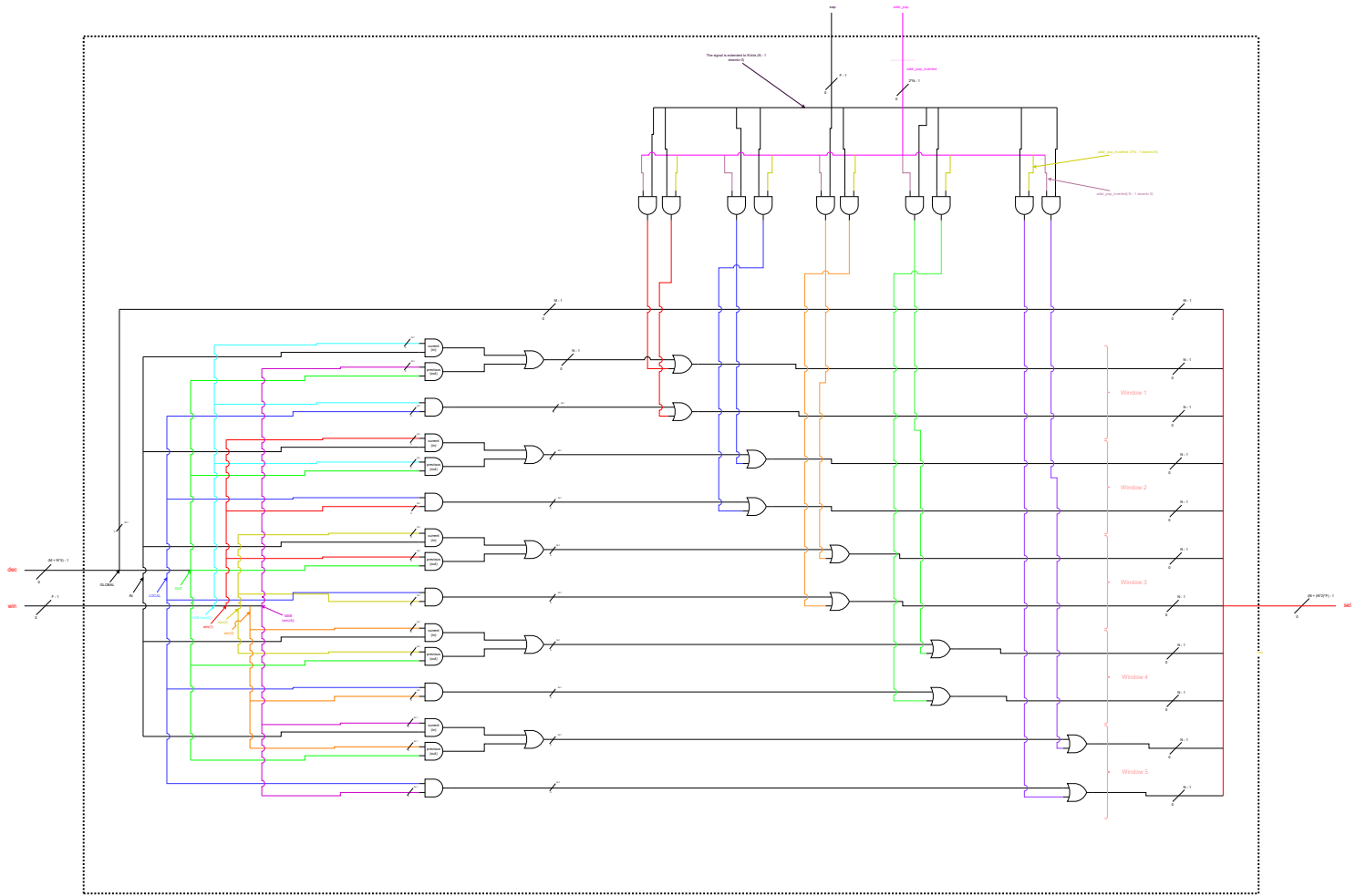


Figure 4.2: Connection matrix

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.

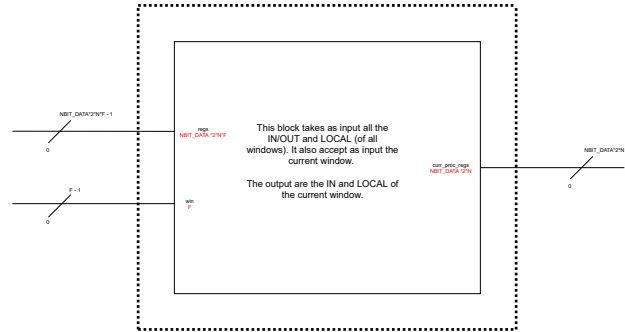


Figure 4.3: Interface of the select block

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.

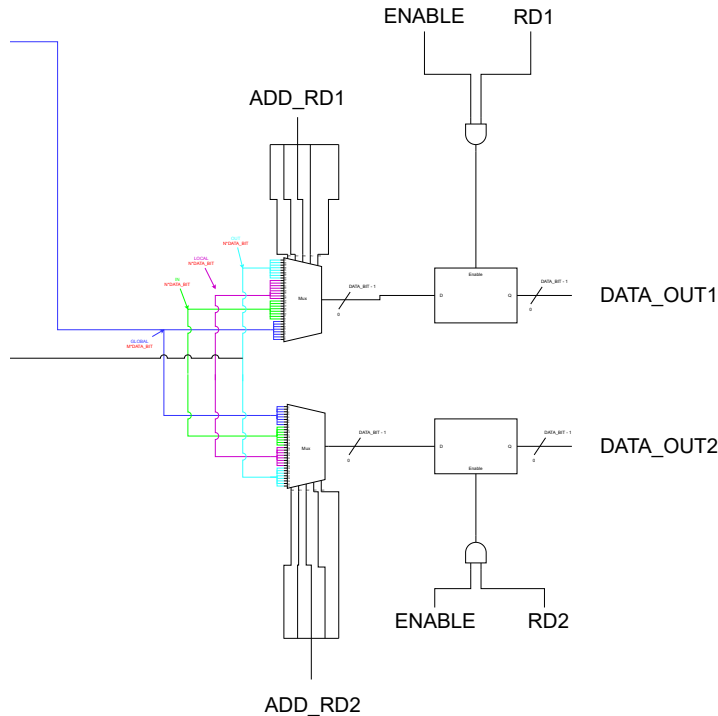


Figure 4.4: Design of the output selection

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.

4.3 Hazard Control

4.4 Comparator

- Unsigned things

4.5 Jump and Branch decision

4.6 Next Program Counter computation

CHAPTER 5

Execute Stage

5.1 ALU: Arithmetic Logic Unit

5.1.1 Adder

5.1.2 Multiplier

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 5.1.

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

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.

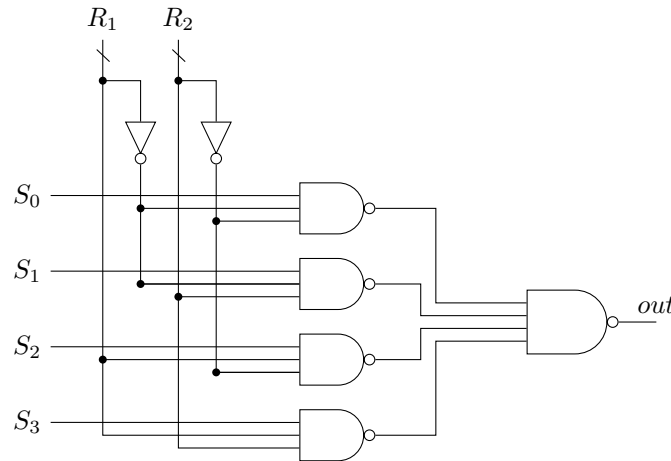


Figure 5.1: Logic unit

5.1.4 Shifting

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 an 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

The unit performs the requested operation in three stages:

1. The first consists 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 difference 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 performs a coarse grain shift, that is basically consists 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, perform 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.

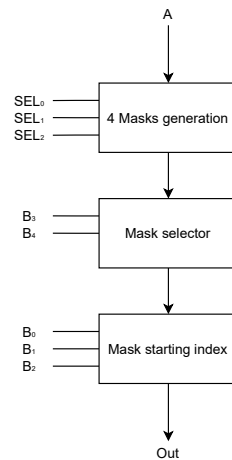


Figure 5.2: Blocks of the Shifter/Rotate Unit

Examples

For example, if we need to perform a left shift 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 be from the bit at position $40 - 1 = 39$ to the one at $39 - 32 = 7$ included.

MASK 2: 00000000 00000000 00000000 00010010 00000000
} shifted A

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:

MASK 1: 11111111 11111111 11111111 11111111 11101110
} shifted A

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

MASK 1: 11100111 00000000 00000000 0000100 11100111
} rotated A

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

- setcmp

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