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 5, 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.

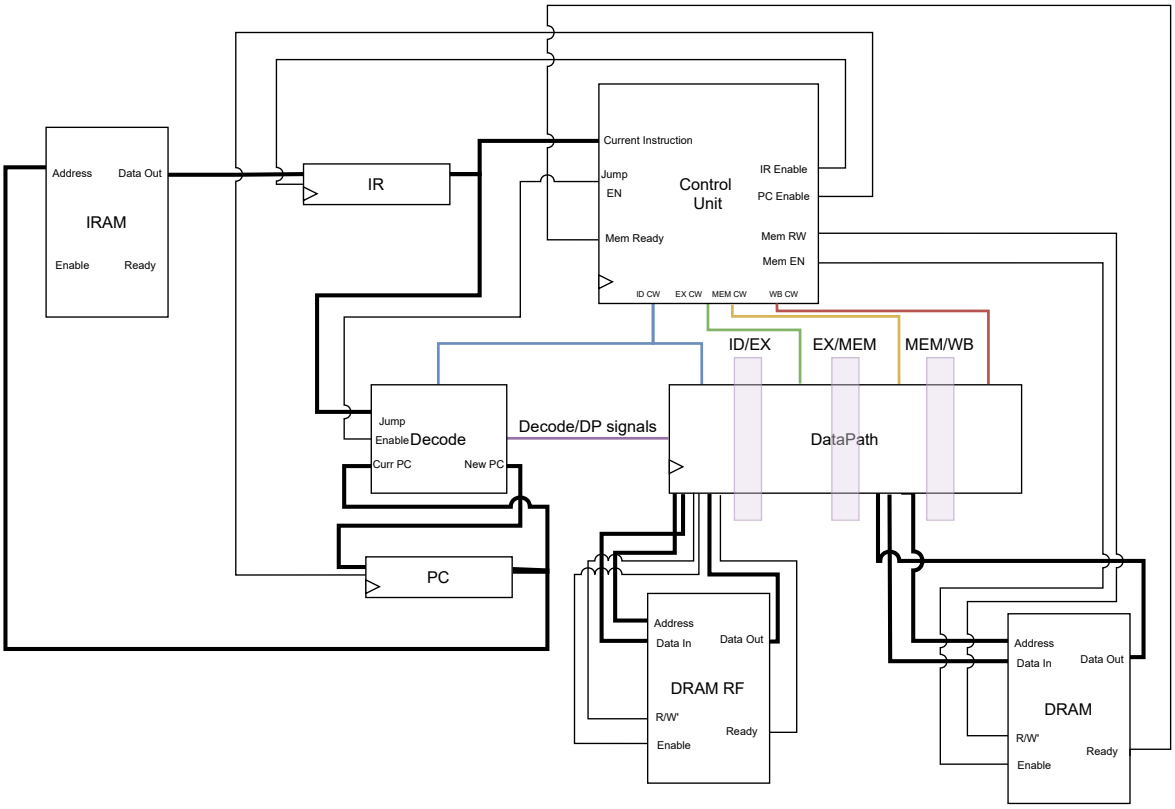


Figure 2.1: Schematic of the DLX

2.2 Pipeline Stages

2.3 Control Unit

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

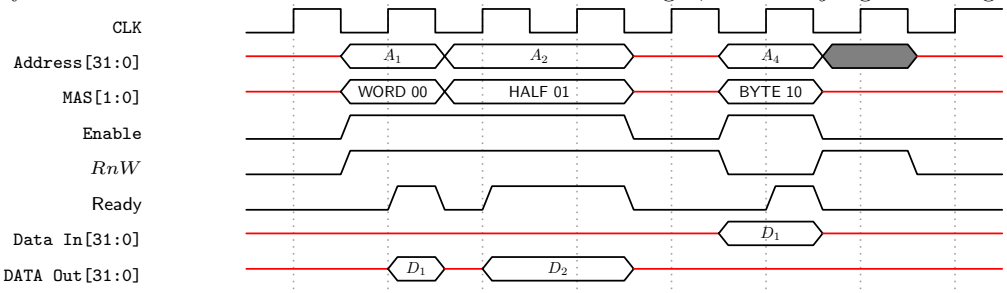
The *Address class signals* are:

- 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. For 8 and 16 bit memories, the data must be placed on the right byte lanes in the data bus.

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

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.

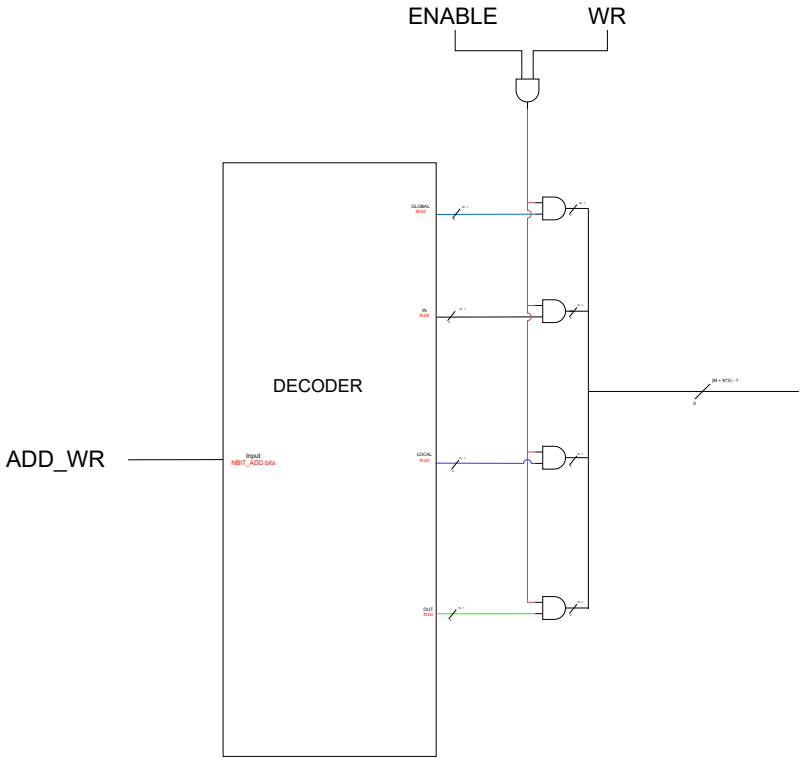


Figure 4.1: Schematic of the Decoder

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 the schematic.

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 choice, 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 by 1 because..... TODO

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.

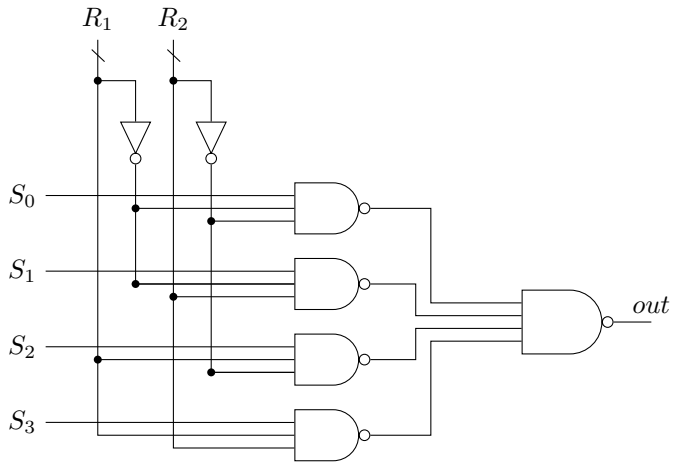


Figure 5.1: Logic unit

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 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

The unit perform the requested operation in three stages, sketched in figure 5.2:

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`.

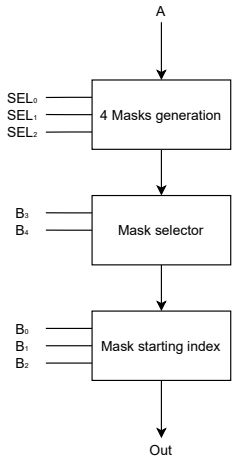


Figure 5.2: Blocks of the Shifter/Rotate Unit

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.

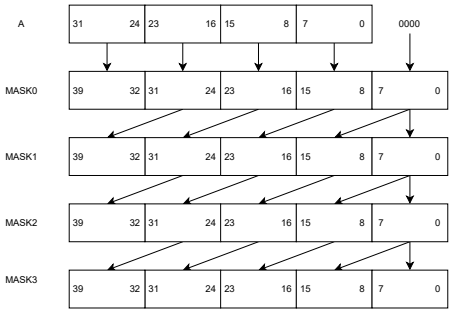


Figure 5.3: Masks for left shift

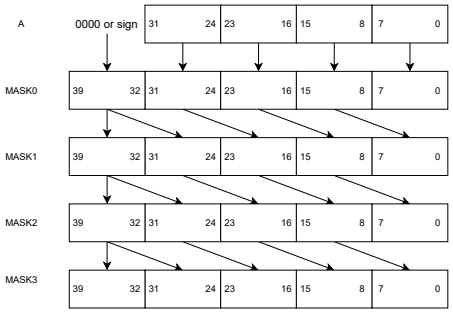


Figure 5.4: Masks for right shift

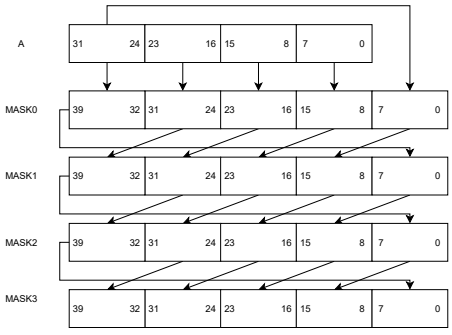


Figure 5.5: Masks for left rotate

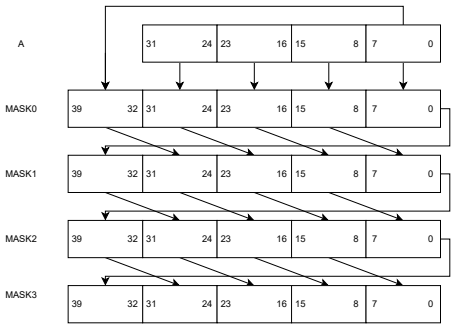


Figure 5.6: Masks for right rotate

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

- 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