Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development
Final Project Report

Master degree in Electronics Engineering
Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group 06

Fogliato Martina, Monti Paolo

July 17, 2017

Contents

Introduction

Aim of the project : DLX Basic

The aim of this project was a VHDL description of a basic fully-pipelined DLX processor (completing the provided CU and designing the datapath) followed by a refinement from RTL architecture down to synthesis and physical design.

Our choices : DLX PRO

We decided to go beyond the simple DLX and add some features to achieve a pro-version of the processor.
In particular:

- Hardware support for **more instructions**: addu, addui, jalr, jr, lb, lbu, lhi, lhu, sb, seq, seqi, sgeu, sgeui, sgt, sgti, sgtu, sgtui, slt, slti, sltu, sltui, sra, srai, subu, subui, mult, call, ret. For the last three we made some modifications to the compiler (see Appendix A).
- **Optimized ALU**: Pentium 4 adder (with power optimization), three-level shifter, multi-purpose comparator, two-level nand logicals, Booth multiplier with Wallace Tree adders structure.
- **Branch and Jump management** : early calculation of jump target address, static prediction logic for branches, flush logic.
- **Forwarding** to prevent data hazards
- **Register File Windowed**
- **Physical Layout**

CHAPTER 1

General functioning and architecture

Let's start with an overview of our processor architecture, whose block diagram is shown in Figure 1.1.

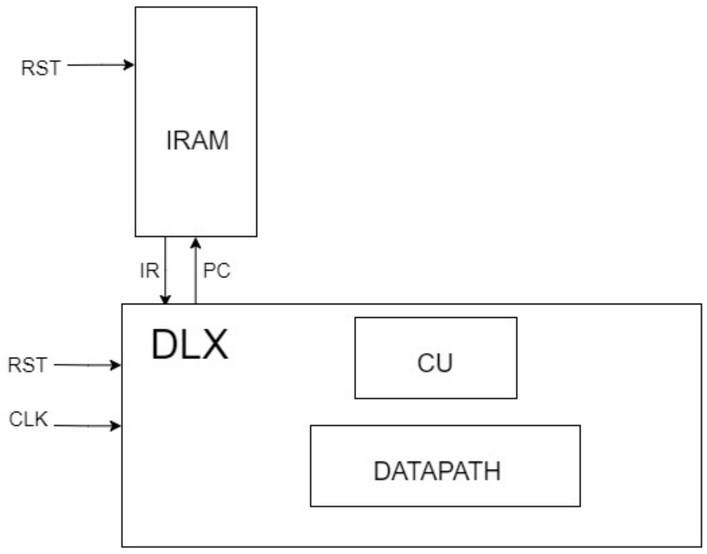


Figure 1.1: block diagram of DLX and IRAM

The asynchronous 32-bit entries IRAM was provided, and our choice was to keep it external with respect to the DLX itself. It is filled according to the content of the file *"test.asm.mem"*, produced by the compiler and containing the binary codes of the assembly instructions, and it is addressed by the processor through the Program Counter (PC). The IRAM returns to the processor the corresponding instruction (IR).
The DLX processor instantiates the Control Unit (Chapter 2) and the 5-stages Datapath (Chapter 3).
All these components are wrapped in a testbench (TB_DLX.vhd), which has been used for the simulation on Modelsim, where the IRAM is declared having 1024 entries and a process takes care of creating the 5 ns clock.
We decided to work with PC values which are multiple of 4, since all instructions that involve an offset (like branch and jump) provide it as multiple of 4. However, the memory locations are accessed with consecutive numbers, so we had to shift right the PC by two before addressing the IRAM.

CHAPTER 2

Control Unit

The CU is the manager of the DLX processor: by providing control signals it defines how the different components should deal with the IR coming from the IRAM and coordinates all the sub-units.
We chose to design an Hardwired Control Unit, with 5 stages of Pipeline (FETCH, DECODE, EXECUTE, MEMORY and WRITE BACK).
The CU receives in input, in addition to clock and reset signals, also the IR and a *flush* signal coming from the datapath, which allows Us to flush the pipeline when needed.
The block diagram of the Control Unit, together with the whole control word and the five stages, can be seen in Figure 2.1.

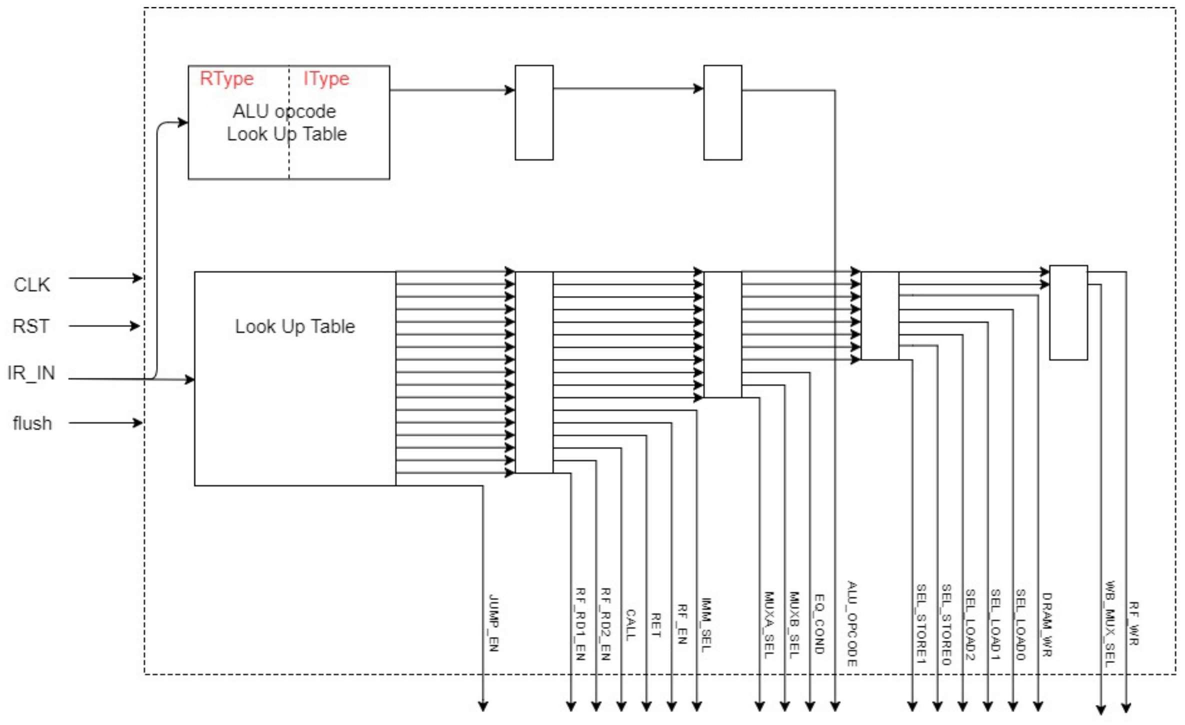


Figure 2.1: block diagram of CU and control word

According to the type of the instruction, the bits of the IR can assume different meanings:

- **R-Type**: 6 bits of OPCODE, RS1 on 5 bits, RS2 on 5 bits, RD on 5 bits and FUNC on 11 bits

- **I-Type**: 6 bits of OPCODE, RS1 on 5 bits, RD on 5 bits and IMMEDIATE on 16 bits
- **J-Type**: 6 bits of OPCODE, 26 bits of IMMEDIATE

We inserted in the CU three Look Up Tables. The first one is used for all instructions, each entry corresponds to a different one (except for R-Type instructions which are all assigned to the first entry) and contains the bits of the control word which is destined to the datapath in order to allow it to efficiently manage the operations. The other two configure the ALU for the correct type of operation; one for the R-Type operations and the other for the I-Type operations.
To improve flexibility and readability of the VHDL code, rather than directly specify the bits for the ALU Opcode, we declared a type aluOp, in a separate file (000-globals.vhd), which we used in the Look Up Table:

```
type aluOp is ( NOP,
                ADDOP, SUBOP, MULOP,
                ANDOP, OROP, XOROP,
                SLLOP, SRLOP, SRAOP,
                GTOP, GETOP, LTOP, LETOP, EQOP, NEQOP,
                GTUOP, GETUOP, LTUOP, LETUOP,
                LHIOP );
```

Whenever a new instruction is received from the IRAM the CU splits Opcode and Function fields. The Opcode is used to access the first Look Up Table and retrieve the corresponding control word. Then, if the instruction is an I-Type, the Opcode is used again to access the second LUT and retrieve the ALU bits; instead, if the instruction is an R-Type, the Function filed is used.
We declared 5 signals, which correspond to 5 control words for the 5 stages of the pipeline; at each clock cycle they shift by one stage, exactly as if pipeline registers are interleaved. In this way, each bit of the CW is available for the datapath in the correct stage of the pipeline.

2.1 Control Word

Going into the details of all the bits of the Control Word of our CU:

- **JUMP_EN** : is equal to 1 whenever the current instruction is a Jump Instruction (J, JAL, JALR, JR).
- **RF_RD1_EN** and **RF_RD2_EN** : Read Enable for the two read ports of the Register File.
- **CALL** and **RET** : they refer to the operation of calling a new subroutine and returning from a subroutine. They are necessary in order to select the appropriate window in the Register File.
- **RF_EN** : it is the enable of the Register File itself and it is equal to 1 for all instructions that need to access registers.
- **IMM_SEL** : according to the type of instruction the processor is executing, the Immediate Field can have different lengths. Whenever this signal equals 1 the 16-bit Immediate (Branches and I-Type) is selected, otherwise the 26-bit Immediate (Jumps) does, both sign-extended. It is used as selection signal of a multiplexer in the datapath.
- **MUXA_SEL** : selects either the value at the output of the Register File (when it is equal to 0), or the value of the Next Program Counter (1) for some jumps and branches.

- **MUXB_SEL** : selects either the value at the output of the Register File (when it is equal to 1), or the Immediate (0).
- **EQ_COND** : this bit is used to identify the condition of branches; it is equal to 1 in case of BEQZ, equal to 0 for BNEZ.
- **ALU_OPCODE** : specify which kind of operation the ALU should execute, among the ones listed in the VHDL code in the previous page.
- **SEL_STORE2** and **SEL_STORE1** : these two bits are passed together to the DRAM to specify the type of store operation; if "11" it is a Store Word (SW), if "10" it is a Store Byte (SB), if "01" a Store Halfword (SH).
- **SEL_LOAD2**, **SEL_LOAD1** and **SEL_LOAD0** : again these three bits are passed together to the DRAM for load operations; if "111" it corresponds to Load Word (LW), if "101" or "001" it means Load Halfword respectively with (LH) and without (LHU) sign extension, if "110" or "010" Load Byte with (LB) and without (LBU) sign extension.
- **DRAM_WR** : it indicates that the instruction needs to write in the DRAM.
- **WB_MUX_SEL** : it is a selection signal that decides which one between the result of the ALU and the output of the DRAM (in case of Load) is going to be written back in the Register File.
- **RF_WR** : enables the write port of the Register File.

Example: **JALR R10**
Saves a return address in register 31 and does an absolute jump to the target address contained in the specified register (R10 in this case).
Basically this instruction sums the content of R10 and R0 (always 0) and use the result as new PC.
The ALU Opcode corresponds to ADDOP.
The corresponding control word is:

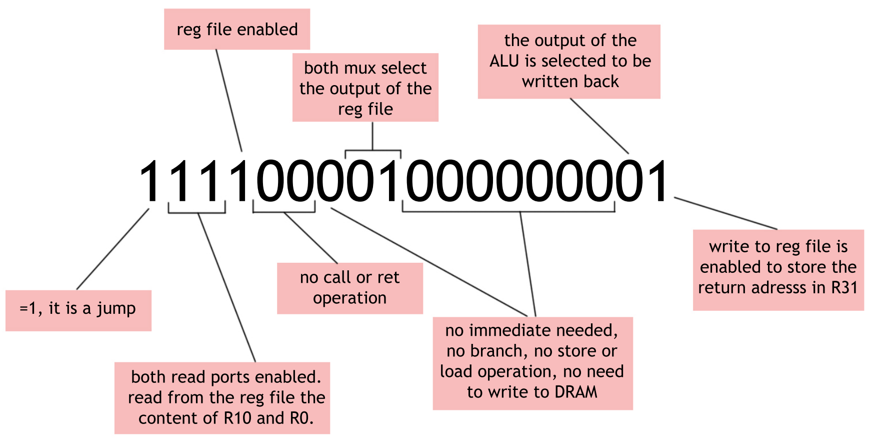

reg file enabled
both mux select the output of the reg file
the output of the ALU is selected to be written back
11110000100000000l
=1, it is a jump
no call or ret operation
write to reg file is enabled to store the return adresss in R31
no immediate needed, no branch, no store or load operation, no need to write to DRAM
both read ports enabled. read from the reg file the content of R10 and R0.

Figure 2.2: JALR CW example

2.2 Flush

Sometimes a flush mechanism is needed to be sure the pipeline is able to correctly process the instructions in the following clock cycles.
As an example, when a branch is mispredicted we need to be sure that the already fetched instructions do not have a visible impact on the execution.
The CU manages the flush mechanism receiving from the datapath the *flush* signal, which assumes a certain value according to how many stages of the pipeline need to be flushed:

- 10 : used in case of branches, for which it takes two clock cycles to recognize a correct or incorrect prediction. It flushes stage 3 and 4 of the pipeline, together with the ALU bits.
- 11 : used in case of jumps, for which only the following instruction (fetched before the jump is taken), should be flushed. It flushes the fourth stage of the pipeline only.

CHAPTER 3

Datapath

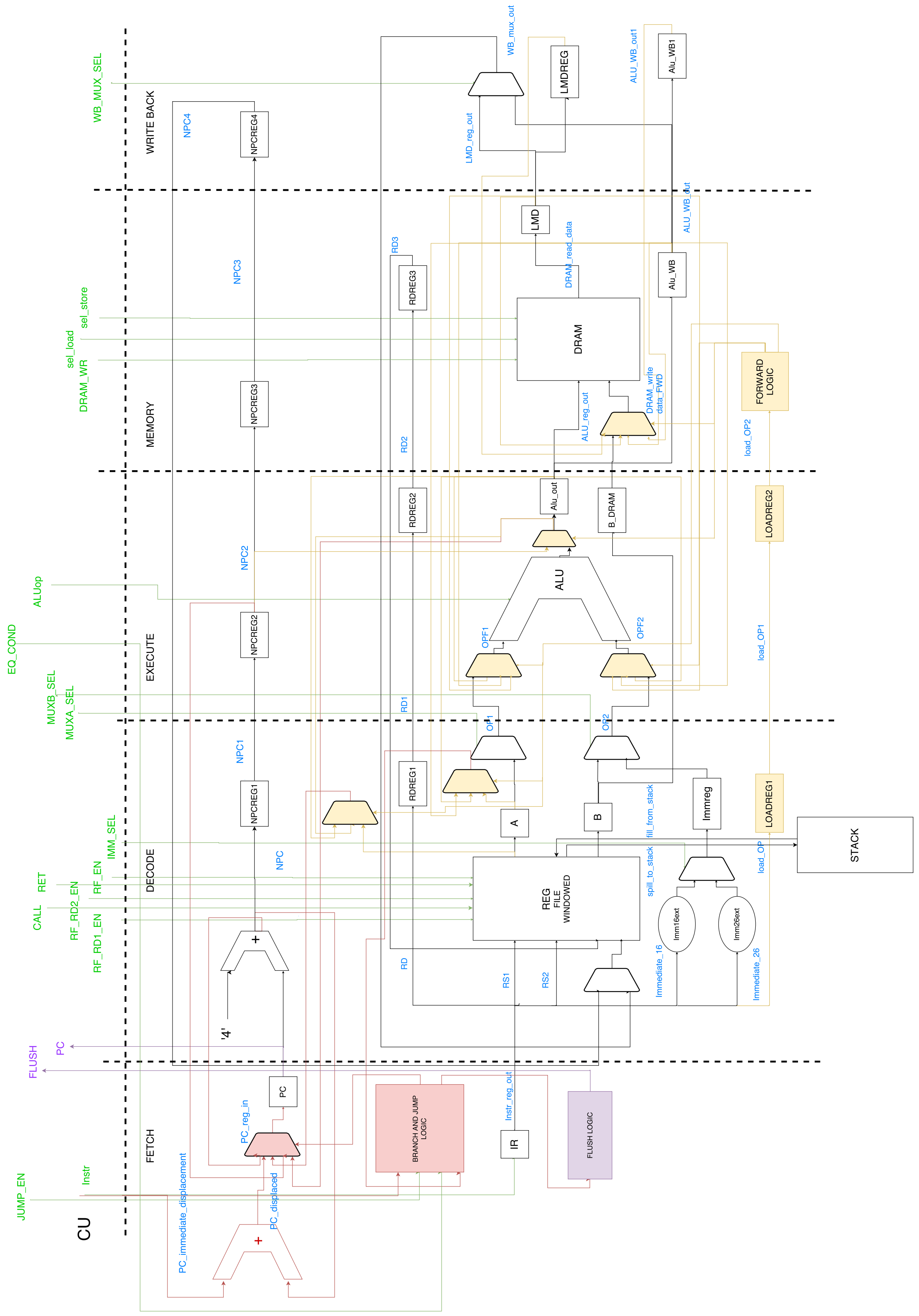
The datapath, the data processing unit of the DLX, is where all the functional units we designed reside. It is where each instruction is processed and executed and the Data Memory is accessed.
As said, it is organized in five different stages:

- **FETCH** : the instruction coming from the IRAM is saved into the IR register. The new target address, in case of branches and jumps, is computed and the new PC is selected.
- **DECODE** : the Register File is accessed to get the needed operands according to the Control Word of the instruction. The output data of the Reg File are saved into two registers, A and B. The immediate filed is sign extended and either the 16-bit or 26-bit one is selected. The condition of branch instructions is checked.
- **EXECUTE** : the ALU performs the specified operation on the two provided operands, exploiting the forwarding logic as well. The result is saved in a register.
- **MEMORY** : if required, the DRAM is accessed. In case of Store operation, the operand B (coming from the forwarding logic or not) is saved in the DRAM which is accessed in write mode, while in case of Load operation the DRAM is accessed in read mode and the value is stored in the LMD register.
- **WRITE-BACK** : the Reg File may be accessed in write mode to store the result of the ALU computations, or the value just loaded from the DRAM or the value of NPC (in case of JALR and JAL)

It is possible to analyze the datapath architecture subdividing it into its most important functionalities and units, that will be detailed in the following sections:

1. **Windowed Register File**
2. **Arithmetic and Logic Unit, ALU**
3. **DRAM**
4. **Branch and Jump Logic**
5. **Forwarding Logic**

In the block diagram below inputs are shown in green , outputs are violet and internal signals blue. The Branch and Jump logic is red, while Forwarding Logic is yellow.


CU
FETCH
DECODE
EXECUTE
MEMORY
WRITE BACK
JUMP_EN
Instr
FLUSH
PC
CALL
RET
RF_RD2_EN
RF_RD1_EN
RF_EN
IMM_SEL
MUXB_SEL
MUXA_SEL
EQ_COND
ALUop
DRAM_WR
sel_load
sel_store
WB_MUX_SEL
PC_immediate_displacement
PC_displaced
PC_reg_in
PC
'4'
NPC
NPCREG1
NPC1
NPCREG2
NPC2
NPCREG3
NPC3
NPCREG4
NPC4
BRANCH AND JUMP LOGIC
IR
Instr_reg_out
FLUSH LOGIC
RD
RS1
RS2
REG FILE WINDOWED
RDREG1
RD1
RDREG2
RD2
RDREG3
RD3
A
B
OP1
OP2
OPF1
OPF2
ALU
Alu_out
B_DRAM
ALU_reg_out
DRAM
DRAM_write data_FWD
DRAM_read_data
LMD
LMD_reg_out
LMDREG
WB_mux_out
Alu_WB
ALU_WB_out
Alu_WB1
ALU_WB_out1
Immreg
Imm16ext
Imm26ext
Immediate_16
Immediate_26
fill_from_stack
spill_to_stack
STACK
LOADREG1
load_OP
load_OP1
LOADREG2
load_OP2
FORWARD LOGIC

3.1 Windowed Register File

In a processor the register file is at the top of the memory hierarchy, it is a directly accessible fast storage. Adding the windowing feature means allocating register space for more than one procedure so that whenever a new one is called, that procedure gets a new set of registers.
Consequently, although the programmer can access only one window at any instant, there is a different window associated with each procedure call. Each window is organized into: IN registers (data passed from the parent to child subroutine), locals (registers which are local to the current subroutine only) and OUT registers (that will be the IN of the following procedure). Globals, instead, are shared among all windows.
The main features of our Register File (a.b.l-W_RegFile.vhd) are:

- 4 windows
- 8 registers for each group (IN, OUT, locals)
- 8 globals
- 32-bit entries
- 2 read ports, 1 write port
- Simultaneous read/write operations
- Synchronous write, asynchronous read
- Active high enables
- Active low, asynchronous reset

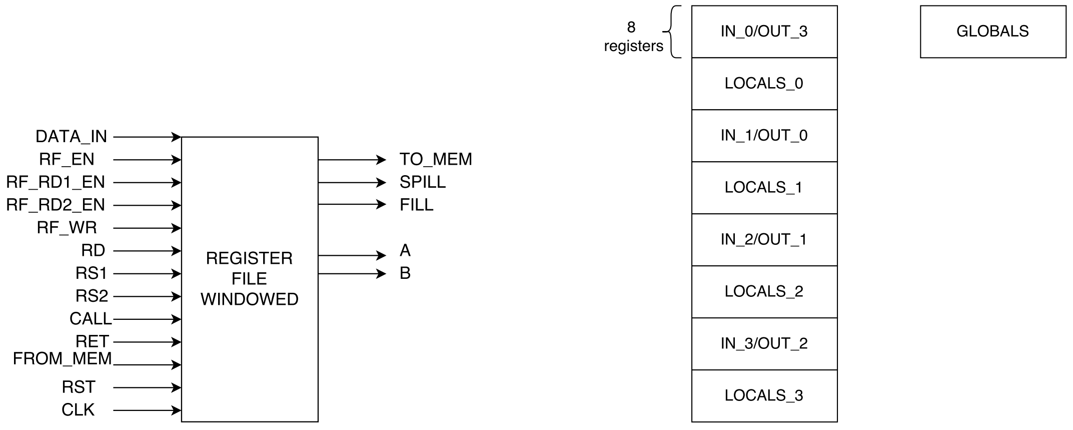


Figure 3.1: Register File Windowed, Block diagram and general structure with 4 windows

At reset, the i^{th} register is initialized to i (i.e. the first register is reset to 1, the second to 2 and so on), while keeping R0 always to 0, in order to simplify debugging operations and verify easily the correct functioning of the architecture.
At the input of the Reg File we need to specify destination and source registers, according to the content of IR:

- destination register R_D : it is where the result of the operation is going to be stored. It is equal to "11111" (R31) in case of JALR and JAL, to IR(15 downto 11) for R-Type instructions, otherwise IR(20 downto 16). Since R_d is used during the Write Back stage, but the instruction is fetched earlier, we added pipeline registers.

- source registers R_{S1} and R_{S2} : they are both present only in R-Type instructions, while in I-Type only R_{S1} is used, and correspond respectively to IR(25 downto 21) and IR(20 downto 16).

At each clock cycle, if the Write Enable is active, the data at the input of the register file is written in the destination register. Whenever the Read Enable is high, instead, the content of the source register(s) is produced at one or both read ports of the Register File, in an asynchronous way, in order not to have a delay of one clock cycle.
To avoid possible spurious signals due to the asynchronous read, the output of the read ports is then stored into two registers, A and B.

3.1.1 CALL and RET

The peculiar behavior of the Windowed Register File with respect to a normal one can be seen whenever the assembly code contains CALL or RET instructions.
We used for this purpose the two instructions TRAP and RETURN FROM EXCEPTION (RFE), changing their name in the compiler.
CALL corresponds to the calling of a new subroutine, which means that the currently active window, pointed by the CWP (Current Window Pointer), should shift by 16 registers. In this way the OUT registers of the previous subroutines become the IN of the new one, while the new subroutine has its own locals and OUT. The CWP is incremented by 16.
RET corresponds to return from a subroutine and means that the currently active window should shift back by 16 registers, in order to recover the data of the parent subroutine. The CWP is decremented by 16.
The globals are shared among all windows and we placed them at the end of the Register File.

3.1.2 SPILL and FILL

Problems arise when all windows have been used and we need to find room for a new subroutine. The spill operation empties one of the windows by saving its content into a stack we implemented. A pointer called SWP (Saved Window Pointer) points to the most recently spilled window.
SWP is initialized to 2N(F-2), where N is the number of registers in each group (8) and F is the overall number of windows (4), since we decided that a spill is necessary before accessing the last window, or the subroutine could modify its OUT registers, corrupting the IN registers of the first window.
Whenever, after a CALL, SWP and CWP are equal a spill operation is needed.
When RET operation is performed, we need to be sure that the correct data are present in the previous window, i.e. it has not been spilled. In case SWP = CWP - 2*N (CWP is about to move into the location pointed by SWP) correct data must be restored and a fill operation is required, during which data are retrieved from the stack.
Both spill and fill are synchronous and require 16 clock cycles, since one data a cycle is stored in or removed from the stack. During this period the pipeline must stall because the Register File is unavailable.
That is why in the datapath we added a signal called *PCen* which influences the enable of PC register. *PCen* is equal to 1 only when neither spill or fill are active, otherwise is 0 and the PC cannot be modified anymore, stalling the whole pipeline.

3.1.3 Stack

The stack (a.b.k-Stack.vhd) is a 256 entries synchronous memory which is only used by the Register File as support for spilling and filling.
A pointer called *sp* points always to the first empty location in the stack. When write enable is active

the stack stores the words at its input incrementing *sp* every time, while when read enable is active data present inside the stack are presented at the output one by one.
Reset is active high and, when disabled, the output of the stack is 'Z'.

3.2 Arithmetic and Logic Unit, ALU

The ALU is the core of the execution stage of the datapath and performs arithmetic and bitwise operations on integer numbers.

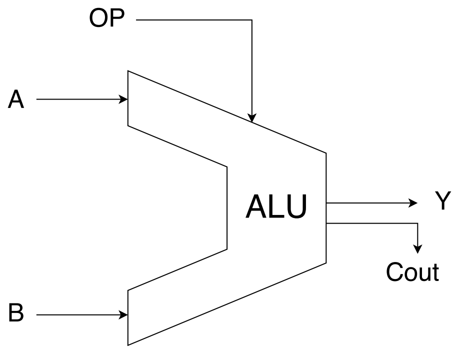


Figure 3.2: ALU Block diagram

It takes as input the two operands and the type of operation that should be performed (OP which is of type AluOp) and gives back the result and a possible carry out. Actually we never use the carry out but it could be useful when exceptions should be detected.
We inserted in the ALU the following units, whose results are multiplexed and only the correct one is assigned to Y :

1. **Adder/Subtractor**
2. **Multiplier**
3. **Shifter**
4. **Logicals**
5. **Comparator**

The following sections describe the details of each of them. Section 3.2.7 gives an insight on how we assigned the signals to each ALU unit according to the the type of operation and eventually Section 3.2.8 provides some information about a power saving technique we implemented on the components of the ALU.

3.2.1 Adder/Subtractor

Our adder/subtractor is a 32-bit inputs 32-bit output revisited version of the Pentium 4 one we designed for the Laboratories, since we added xor gates between each bit of the second operand and the carry in, which works as a selection bit: when equals '0' an addition is performed, otherwise a subtraction.

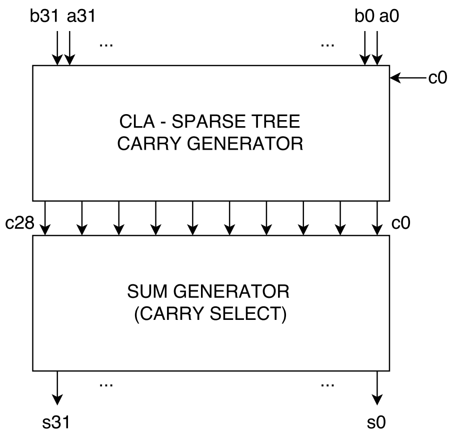

b31 a31
b0 a0
c0
CLA - SPARSE TREE CARRY GENERATOR
c28
c0
SUM GENERATOR (CARRY SELECT)
s31
s0

Figure 3.3: P4 adder Block Diagram

The sparse tree carry generator is based on the carry look-ahead adder and its task is to compute all carries in parallel. It is a sparse radix-2 carry-merge tree, that generates every fourth carry, increasing performances.
All the carries generated are then inputs of the sum generator block, based on carry select adders, which compute sum and carry out both in the case the carry in is 0 and in case is 1. After that, the actual carry in is used as selection signal of a multiplexer to choose the correct option.
This structure helps reducing the delay due to the carry propagation.

3.2.2 Multiplier

Since all operations should be performed in order to fit the result on 32 bits, we chose to feed the multiplier with two 16-bit inputs, the lower 16 bits of the two operands A and B.
Our multiplier is based on radix-4 Booth's algorithm, which allows to decrease the number of partial products with respect to a traditional array multiplier.
During the laboratories we implemented a 32-bit Booth's multiplier with a "cascaded sum", meaning that each adder should wait for the result of the previous one to be generated. From the post-synthesis timing reports we obtained at best (*-map_effort high*) a delay of 7.33 ns.
In order to improve this result we modified the structure of the sum part using a Wallace Tree structure and Carry Save Adders, being able to reduce the delay up to 1.5 ns.
In this way several sums could be computed in parallel avoiding the delay of carry propagation at each step (it occurs only at the last one where the adder is no more a CSA but a P4 Adder), as it can be seen from Figure 3.4.
All multiplexers select the correct value, produced by the shifters, in parallel, at the beginning. The selection signals of these muxes, not shown in Figure for simplicity, are made up of three bits of operand B (b[i+1], b[i], b[i-1]), except the first one which is made up of just two bits and a 0 as b[i-1].

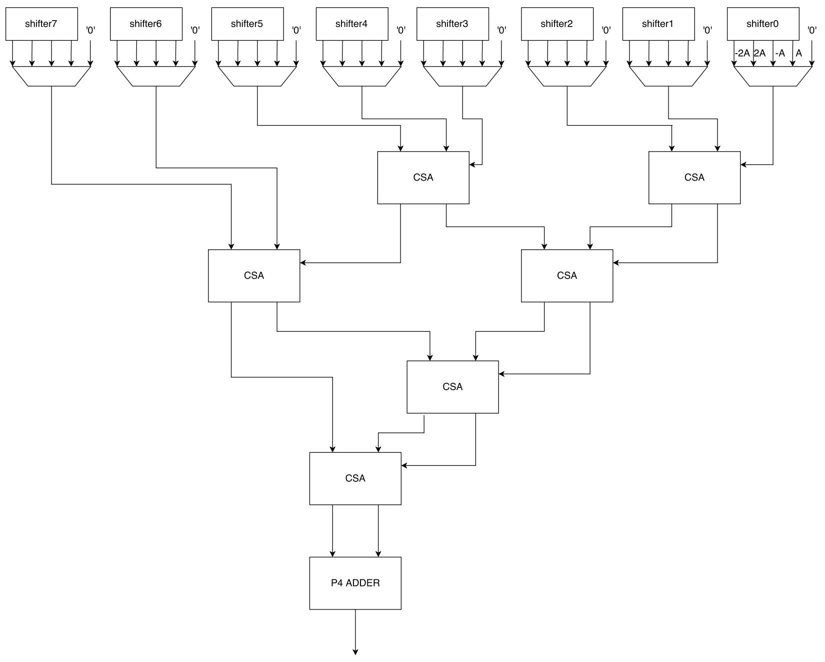


Figure 3.4: Booth's algorithm, Wallace Tree Multiplier

3.2.3 Shifter

As for the shifter, we designed a hardware support able to implement all shift operations, not only the DLX-basic ones: *sll*, *srl*, *srli*, *sra*, *srai*.
The shifter is based on the T2 one explained during lectures and is organized on three levels (assuming that A is the operand to be shiftedd and B controls the shift amount), a scheme in shown in Figure 3.5:

- First Level: according to the selection signal (00, 01 or 10) the first level produces three masks, shifted by 0 (mask00), 8 (mask08) and 16 bits (mask16) each. Moreover, operand A is extended to 39 bits.
  - 00 : the operand is shifted left replacing vacant positions with 0s and extended appending 7 0s to the right.
  - 01 : the operand is shifted right replacing vacant positions with 0s and extended appending 7 0s to the left.
  - 10 : the operand is shifted right with sign extension and extended replicating the sign bit.
- Second Level : the selection of one of the masks produced by the First Level is performed with a multiplexer which has as selection signal bits 3 and 4 of operand B.
- Third Level : the last level corresponds to a fine-grained shift which produces the final result by selecting the correct 32 bits among the 39 ones. The selection is performed by the 3 LSB of operand B. These are used affirmed in case of *sll* or *slli*, complemented in case of srl, *srli*, *sra* or *srai* as we want an opposite behavior.

3.2.4 Logicals

We based our logic unit on the one of UltraSPARC T2. It has two levels of NAND gates:

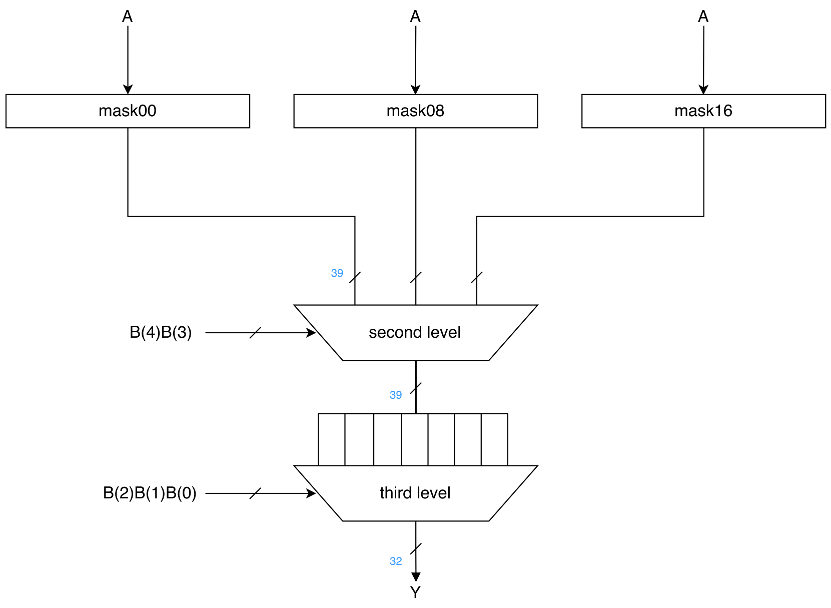


Figure 3.5: Three-level Shifter scheme

- First Level : It has four 64-bit inputs NAND gates. Each NAND has three inputs, two of them for the two operands A and B (affirmed or complemented) and the third one as selection signal.
- Second Level : it consists of one NAND only whose inputs are the four outputs of the previous level.

According to the value of the selection signal, assigned by the ALU opcode (see Section 3.2.6) a different logical operation is performed: AND, NAND, OR, NOR, XOR, XNOR.
Figure 3.6 shows the Block Diagram.

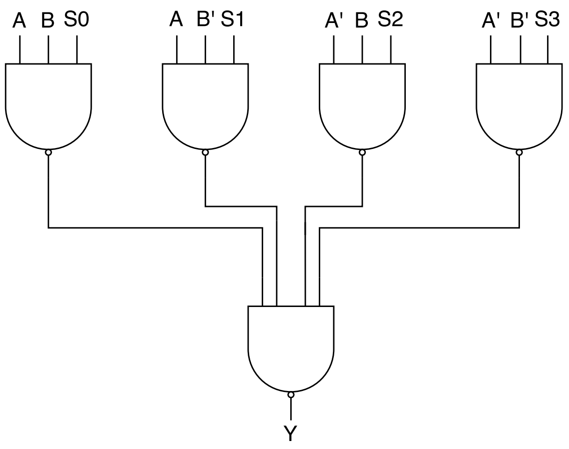


Figure 3.6: Logical Unit Block Diagram

3.2.5 Comparator

The general purpose comparator provides hardware support for several operations: $<$, $>$, $\geq$, $\leq$, $=$, $!=$. It is based on the one we saw during lectures, but modified to deal efficiently with signed numbers

too. The block diagram in shown in Figure 3.7, which includes an adder as well even if the addition is actually performed by the P4 adder previously described.
Together with the operands to be compared a *sign* bit is passed to the comparator, whose value is A(31) xor B(31), for those operations that require a signed comparison. This is because the only case in which problems could arise are the ones in which the signs of the two operands are different, otherwise the comparator can operate normally.
Whenever *sign* is equal to 1 (the operands have different sign) the carry out must be complemented, so we used a xor between *cout* and *sign*, highlighted in the diagram.

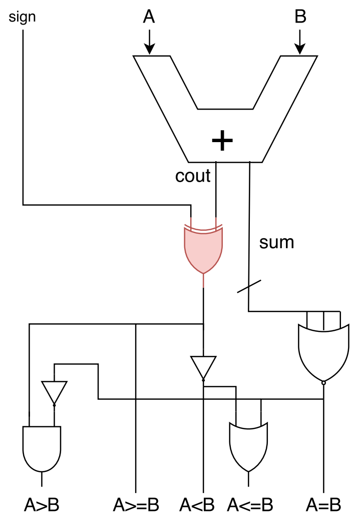


Figure 3.7: Comparator Block Diagram

3.2.6 LHI

The instruction Load Halfword Immediate (LHI) loads the 16 bits immediate into the most significant part of the register, clearing the least significant part.
We decided to treat it as a special case of instruction managed by the ALU, with the following code:

```
out_lhi <= B(15 downto 0) & x"0000";
```

3.2.7 Opcodes

We report here the input assignment to the functional units of the ALU for each instruction.
The *add_sub* signal is the carry in of the adder and defines whether the instruction is an addition or a subtraction. The above mentioned *sign* signal assumes value 0 for unsigned comparisons, A(31) xor B(31) for signed ones. The selection signals of logicals and shifter are, respectively, *sel_log* and *sel_shift*.
It can be noticed that each unit has its own copy of A and B operands; this choice is related to the power saving technique we implemented, described in the following section.

```
process(A, B, OP)
    begin
      case OP is
        when ADDOP => add_sub <= '0'; A_add <= A; B_add <= B;  -- A + B
        when SUBOP => add_sub <= '1'; A_add <= A; B_add <= B;  -- A - B
        when MULOP => A_mul <= A; B_mul <= B; -- A * B
        when ANDOP => sel_log <= "0001"; A_log <= A; B_log <= B;  -- A and B
        when OROP => sel_log <= "0111"; A_log <= A; B_log <= B; -- A or B
        when XOROP => sel_log <= "0110"; A_log <= A; B_log <= B;  -- A xor B
        when SLLOP => sel_shift <= "00"; A_sht <= A; B_sht <= B;  -- A sll B
        when SRLOP => sel_shift <= "01"; A_sht <= A; B_sht <= B;  -- A srl B
        when SRAOP => sel_shift <= "10"; A_sht <= A; B_sht <= B;  -- A sra B
        when GTUOP => add_sub <= '1'; sign <= '0'; A_add <= A; B_add <= B;  -- A > B
        when GETUOP => add_sub <= '1'; sign <= '0'; A_add <= A; B_add <= B;  -- A >= B
        when LTUOP => add_sub <= '1'; sign <= '0'; A_add <= A; B_add <= B; -- A < B
        when LETUOP => add_sub <= '1'; sign <= '0'; A_add <= A; B_add <= B;  -- A >= B
        when GTOP => add_sub <= '1'; sign <= A(31) xor B(31); A_add <= A; B_add <= B; -- A > B
        when GETOP => add_sub <= '1'; sign <= A(31) xor B(31); A_add <= A; B_add <= B; -- A >= B
        when LTOP => add_sub <= '1'; sign <= A(31) xor B(31); A_add <= A; B_add <= B; -- A < B
        when LETOP => add_sub <= '1'; sign <= A(31) xor B(31); A_add <= A; B_add <= B; -- A >= B
        when EQOP => add_sub <= '1'; sign <= '0'; A_add <= A; B_add <= B; -- A == B
        when NEQOP => add_sub <= '1'; sign <= '0'; A_add <= A; B_add <= B; -- A /= B
        when NOP => NULL;
        when LHIOP => B_lhi <= B;
        when others => NULL;
      end case;
    end process;
```

3.2.8 Input state assignments

Every time the ALU is computing the result of a given operation only one of the functional units is actually needed.
However, if the operands A and B are fed to all units, all of them will be active and computing the result in parallel at the same time, but only one of them will be selected by the output multiplexer, all the others are useless and will be discarded.
This is why we chose to have several copies of the operands, one for each unit: when the operation the ALU has to perform involves a particular unit, only that one has its inputs modified, all the others maintain the previous values.
Doing so, we reduce the switching activity of the ALU and, consequently also the dynamic power dissipation.
We synthesized both versions of the ALU extracting the power reports.
The dynamic power report of the ALU without state assignment (after timing optimization) is as follows:

Cell Internal Power = 5.8580 mW (48%)
Net Switching Power = 6.4103 mW (52%)
*Total Dynamic Power = **12.2684 mW** (100%)*

While for the version with input state assignment:

Cell Internal Power = 967.8691 uW (49%)
Net Switching Power = 988.3012 uW (51%)
*Total Dynamic Power = **1.9562 mW** (100%)*

3.3 DRAM

While IRAM is where instructions reside, the DRAM is the portion of memory that stores data. It is the memory accessed by the processor whenever load or store operations should be executed.
We designed and instantiated a DRAM with 2^{12} 32-bit words, knowing that it will be left out during the synthesis step. Figure 3.8 shows the general block diagram of our memory. All write operations

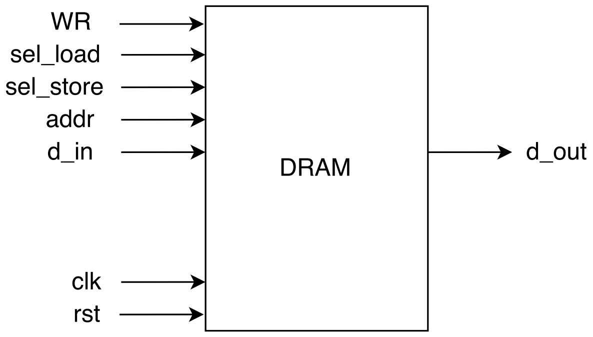


Figure 3.8: DRAM Block Diagram

(WR=1) are synchronous, while read operations (WR=0) are asynchronous as in the register file, not to loose clock cycles and have the needed data immediately available at the output.
All operations are defined by the two signals *sel_load* and *sel_store*, as described in Section 2.1, to specify the number of bytes (8, 16 or 32) to be read or written.
The *sel_load* signal has three bits and can assume the following values (the first bit indicates whether a sign extension should be performed or not):

- 111 : Load Word
- 101 : Load Halfword
- 110 : Load Byte
- 001 : Load Halfword Unsigned
- 010 : Load Byte Unsigned

The *sel_store* signal instead has two bits and can have value:

- 11 : Store Word
- 01 : Store Halfword
- 10 : Store Byte

To access the DRAM the address should have 12 bits, which come directly from the output of the ALU, since for L/S operations the ALU task is the computation of the effective address.
In order to avoid glitches caused by the asynchronous read at load time, a register is added at the output of the DRAM (called LMD in the Datapath).

3.4 Branch and Jump Logic

Branch and jump instructions modify the value of the Program Counter, so that instructions are no more executed in program order and the CPU can start fetching them from a different sequence of

memory cells.
A simplified scheme of all units involved in the branch logic we implemented can be seen in red color in the block diagram of the Datapath.

3.4.1 Branch Instructions

Branch instructions are also called conditional jumps because a certain condition should be true for the branch to be taken. Our DLX can process two types of branches:

- BEQZ : Branch if EQual to Zero. The PC is modified only if the content of the specified register is equal 0.
- BNEZ : Branch if Not Equal to Zero. The branch is taken only if the content of the tested register is not equal to zero

In both cases, whenever a branch instruction is fetched from the IRAM, the Register File is accessed to read the content of the specified register. EQ_COND bit from the Control Word is exploited to check whether the content of the register is what expected. If so, the ALU computes the new value the PC assumes in order to take the branch, otherwise instructions keep on being fetched in program order.
However, this means that without any prediction technique the new PC is computed at the EXECUTE stage and in the meanwhile other two instructions are fetched, perhaps the wrong ones.
The choices we made to deal with branches in a more efficient way can be summed up as follows:

- Insertion of **P4 adder** in the FETCH pipeline stage : avoids having to wait for the EXECUTE stage in order to compute the target address. This adder takes as input the NPC and the Immediate (16-bit for branches) directly taken from the asynchronous output of the IRAM rather than from the IR register, in order save a clock cycle.
- **Static Branch Prediction** based on branch direction : forward branches are predicted as not taken, backward branches as taken. A signal called *forward_branch* assumes value 1 if the MSB of the Immediate equals 0 (the branch implies a forward jump), 0 otherwise (backward jump). If the processed instruction is a branch and signal *forward_branch* is 1, the multiplexer at the input of PC register selects the output of the P4 adder, predicting the branch as taken. Otherwise it selects the NPC, predicting the branch as not taken.
- **Misprediction Check** : the correctness of the prediction can be checked only after the Register File has been accessed. The register to be tested is read and a signal *A_is_zero* equals 1 if the content is zero. Therefore, if *A_is_zero* and EQ_COND are equal the branch was taken. If the prediction was correct nothing is modified, while in case of misprediction two scenarios are possible:
  - The backward branch, predicted as taken, was untaken : signal *mispredict_taken* is raised and the multiplexer of PC selects the previously computed NPC. The incorrectly fetched instruction is flushed from the pipeline.
  - The forward branch, predicted as untaken, was taken : signal *mispredict_untaken* is raised and the multiplexer of the PC selects the output of the ALU to retrieve the correct target address. In this case two stages of the pipeline should be flushed.

3.4.2 Jump Instructions

Dealing with jumps was easier since they are unconditional, always taken and no prediction should be made about their outcome.
Our DLX is able to process 4 types of jumps:

- **J** : relative unconditional jump
- **JAL** : relative jump that saves the return address in register 31
- **JALR** : absolute jump to the target address stored in the specified register and saves the return address in register 31
- **JR** : absolute jump to the target address stored in the specified register

Like for branch instructions we can avoid time penalties by computing in advance the target address without waiting for the EXECUTE stage of the pipeline.
The logic we implemented to do so, can be summed up as follows:

- Jump **characterization** : set of signals that test the opcode of the combinational output of the IRAM to detect the type of jump among the four aforementioned to understand which kind of resources are going to be used.
- **P4 adder** : is the same already described for branches, which computes the target address during the FETCH stage of the pipeline, but for a jump has as input the 26-bit sign extended Immediate.
- Selection of the correct **NPC** : if the Jump is a J or JAL, that do not need to access the Register File to know where to jump, the output of the P4 adder is immediately selected as Next Program Counter. Otherwise, in case of JR or JALR, we cannot avoid losing one clock cycle in order to access the Register File. The content of the register read is then selected directly as Next Program Counter since they are absolute jumps and a flush mechanism is needed.

3.5 Forwarding Logic

Among the drawbacks of using a pipeline architecture there are stalls due to **data hazards**. They occur when there is a true data dependency among instructions being processed by the pipeline at the same time, so that one of them has to wait for the completion of the other one to use its results.
In order to avoid stalls and penalties it is possible to directly forward some results of an instruction to another one before completion.
The most common application of forwarding is related to the results of the ALU, but data dependency can involve any functional unit, especially when combined with other techniques, as branch prediction.
We tried to solve these hazard as much as possible and spot all critical combinations of instructions that may lead to stalls or worse, logical errors.
A simplified scheme of our forwarding logic can be seen in the block diagram of the Datapath in yellow color. It mainly involves multiplexers for the selection of the correct operand.
The detailed description of the whole logic is quite complex and we present here the 4 main situations in which we implemented forwarding:

1. At the two inputs of the **ALU** : two multiplexers are inserted which can select
   (a) the output of registers A and B (which contain the read value from the Reg File) as without forwarding, if forwarding is not needed
   (b) the output of the ALU computed by the previous instruction in case we need its results
   (c) the output of the ALU computed two instructions ago, which still did not have time to be written back in registers

   (d) the output of the DRAM (LMD register) in case we need the value loaded from memory two operations ago (for loads executed less than two operations ago we inserted a stall since they cannot be dealt with).

   We checked also that instructions are not jumps, branches or store, since forwarding on the ALU would be meaningless, and that the fetched instruction has not been flushed.

2. At the input of the **DRAM** : forwarding must be implemented for those store operations that need result just computed and not yet stored in registers. The multiplexer can select
   (a) the output of register B when forwarding is not needed
   (b) the output of the ALU computed by the previous operation
   (c) the output of the ALU computed two instructions ago, which still did not have time to be written back in registers
   (d) the value loaded by the previous instruction
   (e) the value loaded two instructions ago
3. For **Branches** : each branch instruction checks a condition on a specific register, so forwarding is needed when the value of this register is being modified by a previous instruction. The multiplexer in this case can select
   (a) the output of A register if forwarding is not needed
   (b) the output of the ALU computed by the previous operation
   (c) the output of the ALU computed two instructions ago, which still did not have time to be written back in registers
4. For **Jumps** : only in case of JR or JALR, that load in PC the value of a given register. As for branches, the content of this register can be modified by a previous instruction. The multiplexer can select
   (a) the output of the Register File if forwarding is not needed
   (b) the output of the ALU computed by the previous operation
   (c) the output of the ALU computed two instructions ago, which still did not have time to be written back in registers

CHAPTER 4

Synthesis

When satisfied with our design we proceeded with the next step, the Synthesis phase using **Design Vision**.
Since the DLX includes some critical blocks from the critical path point of view (like the Multiplier) and some multiple sub-design references (like the P4 adder), we chose a sort of Bottom Up compilation strategy.
The **compile-once-dont-touch** method has been exploited. "If the environments around the instances of a multiply referenced design are sufficiently similar [...] you compile the design, using the environment of one of its instances, and then you use the set_dont_touch command to preserve the sub-design during the remaining optimization"[1].
To automatize the process we wrote a *tcl* script which allows us to Analyze, Elaborate and Compile the design at once (*compileBottomUp.tcl*).
In this chapter we analyze the whole script and the result of the synthesis.

```
exec mkdir -p work
exec mkdir -p rpt
exec mkdir -p netlists

foreach file [exec find . -type f -name "*.vhd" | sort] {
        analyze -format vhdl $file}

elaborate DLX -architecture STRUCT -library WORK

current_design dlx
```

Three folders are created: *work*, *rpt* (to keep all timing and power reports) and *netlists* (to save .vhdl and .ddc files that may be useful to avoid recompiling each time).
Then, all .vhd files are analyzed in order, bottom up. We left off all memories (IRAM, stack, DRAM).

```
characterize DTP/ALU_block/mul/P4adder
current_design cla_adder_N32
compile
report_timing > rpt/cla_timing_unopt.txt
report_power > rpt/cla_power_unopt.txt
```

[1]from Design Compiler User Guide 8-25, Version X-2005.09, September 2005

```
set_max_delay 0.55 -from [all_inputs] -to [all_outputs]
compile -map_effort high
report_timing > rpt/cla_timing_opt.txt
report_power > rpt/cla_power_opt.txt
write -hierarchy -format vhdl -output netlists/adder.vhdl
write -hierarchy -format ddc -output netlists/adder.ddc
current_design dlx
set_dont_touch {DTP/ALU_block/mul/P4adder  DTP/ALU_block/adder_subtr/add
   DTP/jump_adder} true
```

First, the P4 Adder is compiled (with *-map_effort high*). We tried to reduce the maximum delay and eventually we chose the above reported value of 0.55 ns.
As said, power and timing reports are saved, together with .vhdl and .ddc files.
In our design the P4 adder is used three times : inside the ALU, to compute the target address as part of branch and jump instructions logic and as one of the adders of the Booth's algorithm Wallace Tree multiplier. The compile-one-don't-touch method allow us to resolve multiple instances like in this case.

```
characterize DTP/ALU_block/mul
current_design booth_mul_N16
compile
report_timing > rpt/mul_timing_unopt.txt
report_power > rpt/mul_power_unopt.txt
set_max_delay 1.5 -from [all_inputs] -to [all_outputs]
compile -map_effort high
report_timing > rpt/mul_timing_opt.txt
report_power > rpt/mul_power_opt.txt
write -hierarchy -format vhdl -output netlists/mul.vhdl
write -hierarchy -format ddc -output netlists/mul.ddc
current_design dlx
set_dont_touch {DTP/ALU_block/mul} true
```

The multiplier too is compiled on its own, since it is the component which will most likely influence the clock of the overall design.

```
characterize DTP/ALU_block
current_design alu
compile
report_timing > rpt/aluv2_timing_unopt.txt
report_power > rpt/aluv2_power_unopt.txt
set_max_delay 1.6 -from [all_inputs] -to [all_outputs]
compile -map_effort high
report_timing > rpt/aluv2_timing_opt.txt
report_power > rpt/aluv2_power_opt.txt
write -hierarchy -format vhdl -output netlists/aluv2.vhdl
write -hierarchy -format ddc -output netlists/aluv2.ddc
current_design dlx
set_dont_touch {DTP/ALU_block} true
```

Once both Adder and Multiplier are compiled, the whole ALU can be compiled as well and it can be seen that the max delay we set is slightly larger than the multiplier one.

```
current_design dlx
create_clock -name "Clk" -period 2 Clk
compile
report_timing > rpt/dlxv2_timing_unopt.txt
report_power > rpt/dlxv2_power_unopt.txt
create_clock -name "Clk" -period 1.7 Clk
set_max_delay 1.7 -from [all_inputs] -to [all_outputs]
compile -map_effort high
report_timing > rpt/dlxv2_timing_opt.txt
report_power > rpt/dlxv2_power_opt.txt
write -hierarchy -format vhdl -output netlists/dlx.vhdl
write -hierarchy -format ddc -output netlists/dlx.ddc
write -hierarchy -format verilog -output netlists/dlx.v
write_sdc netlists/dlx.sdc
```

Finally we compiled the whole microprocessor, creating a clock and setting it to 1.7 ns. Actually compiling we saw that the clock could be smaller than that, but we decided to leave some margin to account for uncertainties and process variations. A 2 ns clock would avoid any possible unlucky circumstance.
We also saved the verilog and sdc file in order to produce the physical layout.

We report some data from the timing and power reports:

Cell Internal Power = 13.6035 mW (82%)
Net Switching Power = 2.9305 mW (18%)
Total Dynamic Power = 16.5340 mW (100%)
Cell Leakage Power = 700.6071 uW

Data required time	*1.66*
Data arrival time	*-1.63*
Slack (MET)	*0.03*

CHAPTER 5

Physical Layout

As last step, similarly to what we did during Laboratories, we proceeded with the Physical design of our DLX using **Encounter**.
We followed all the suggested steps from the addition of Power Rings and Stripes and Cell Placement, to the Clock-Tree-Syntesis and eventually Routing. During geometry check we had no violations.
We modified some of the parameters that we used for Floorplanning during Laboratories since, being the DLX much larger than the designs which we worked with, we decided to avoid congestions.

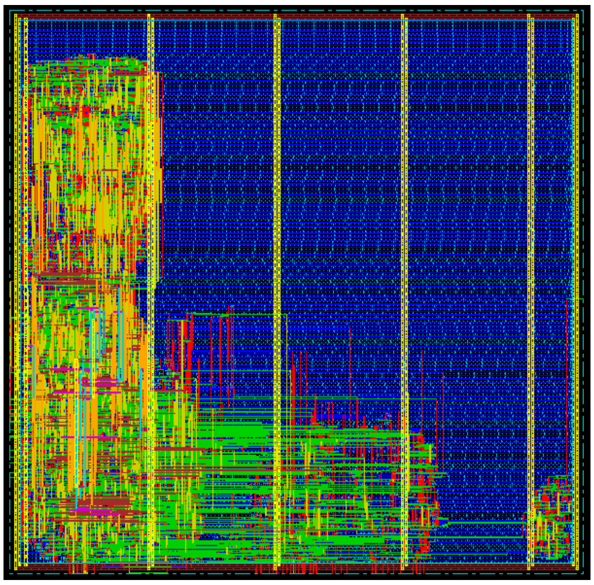


Figure 5.1: DLX Layout, post routing

Among the various reports we extracted, there was also the gate count one.
Here is some of the information it provides:

Unit	Number of gates	Area[μm^2]
total	11617	9270.4
ALU	5947	4745.2
mul	3355	2677.3

APPENDIX A

Compiler modifications

We made the following modifications to the code of the compiler:

- In the dlxasm.pl script, we changed line 135 and 136 in order to support instructions *call* and *ret* instead of *trap* and *rfe*.
- We changed line 104 so that the *mult* operation was considered as R-Type, making it compatible with our architecture.
- In the conv2memory script, we substituted the command

  ```
  od --width=4 -t xC $1 | awk '{print $2$3$4$5}'
  ```

  with the command

  ```
  xxd -c 4 $1.exe | awk '{print $2$3}' > ../test.asm.mem
  ```

  This is because the *od* executable would simply write an asterisk * when reading two identical lines of code instead of writing the line twice. This would cause the IRAM to interpret it wrongly. Adding a *-v* argument could have solved the problem, but we decided to use *xxd* because it is also OS independent.