Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Computer Engineering

Referents
Prof. Mariagrazia Graziano
Giovanna Turvani

Author
Sergio Mazzola
257926

October 2019

Contents

Introduction

The aim of this report is to describe the DLX microprocessor architecture that I developed in the context of the Microelectronic Systems course held by Prof. Mariagrazia Graziano during the academic year 2018/'19. The project encompassed the design of a custom DLX from HDL specification down to its simulation, synthesis and physical design; the whole process will be analyzed in the following, along with the design choices which led to the final implementation.

Features

My architecture, which is meant to be a **DLX-pro**, presents the following features:

- **Extended instruction set** including: `addui`, `subui`, `lhi`, `jr`, `jalr`, `srai`, `seqi`, `slt`, `sgti`, `lb`, `lbu`, `lhu`, `sb`, `sltui`, `sgtui`, `sleui`, `sgeui`, `sra`, `addu`, `subu`, `seq`, `slt`, `sgt`, `sltu`, `sgtu`, `sgeu`, `mult`;
- **extended datapath** to perform all instructions: integer multiplier, jump register logic, multiple format alignments for data memory;
- **architectural optimizations**: latency-optimized adder, latency-optimized multiplier, structural logic unit, structural general-purpose comparator, power-optimized ALU;
- **windowed register file** with a stack for spill and fill operations;
- **data hazards detection unit** with **forwarding logic** management and stalls issuing;
- **control hazard management logic** with branch history table and anticipated target address computation.

Moreover, to further extend the project, the following phases have been carried out:

- **behavioural simulation** with custom asm file;
- **advanced synthesis** with frequency-constrained optimization of area and dynamic power;
- **post-synthesis simulation** and power optimization with back-annotated switching activity;
- **physical design** with violations fix and post-routing analysis.

Project structure

The described custom DLX architecture has all the basic properties of the common DLX microprocessor, limited to its integer part:

- 32-bit Load-Store RISC architecture;
- 32-bit Instruction Register with 3 possible instruction formats (Register-Register, Immediate, Jump);
- five-stages pipeline (Instruction Fetch, Instruction Decode, Execution, Memory, Write-Back);
- integer Register File with 32 general-purpose registers (for each routine);
- byte-addressable, big-endian data memory with 4-bytes words.

The datapath is managed by an hardwired control unit, which allows the correct execution of a total of 55 instructions; 54 of them comes from the integer part of the DLX instruction set architecture (ISA), while the `mult` is a modification of a floating-point instruction, which required some changes to the assembler script as described in Appendix B. The whole instruction set is presented in Appendix A.

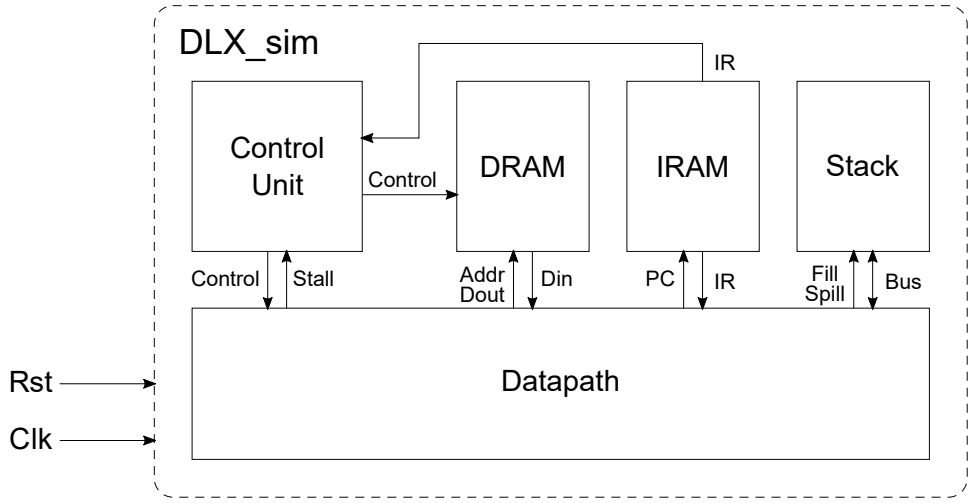

Figure 1: Structure of the DLX top-level entity employed for simulation purposes.

The top-level entity which has been used for simulation, the file `/src/a-DLX_sim.vhd` (see Appendix C for project files organization), is shown in Fig.1. Its interface only contains `Clk` and `Rst` (asynchronous and active-low reset) signals, being all the other signals distributed internally among datapath, control unit and memories (instruction memory, data memory, stack). All these components are analyzed in Ch.1, Ch.2 and Ch.3. For the simulation to take place, two files are needed to be placed in the ModelSim directory:

- *test.asm.mem*: it is the output of the Perl asm assembler and contains a 32-bit hexadecimal word for each line representing an instruction; at reset, its content is loaded in the instruction memory to constitute the firmware of the DLX;
- *data.mem*: it is used to load some initial data in the data memory at reset, and should contain a 32-bit hexadecimal word for each line.

The simulation is fully automatized by the ModelSim script `/sim/simulation.tcl`, which compiles all the needed VHDL source files in `/src` and the DLX testbench `/sim/testbenches/TEST-a-DLX_sim.vhd`. Note that all scripts in the project contain a *project directory* variable, which has to be set to the right path basing on the machine on which the script is being executed.

As memories are employed for simulation purposes only, a similar wrapper but connecting just datapath and control unit has been employed for synthesis, thus moving all the signals of memories interfaces to the entity ports; Ch.4 deals with the synthesis process. Physical design is instead described in Ch.5.

CHAPTER 1

Control Unit

Due to the strongly incremental nature of this project, I decided to base my DLX on an **Hard-wired Control Unit**, whose simplicity allows agile modifications, at least during design stage.

The control unit has been designed to drive a **five-stages pipelined datapath**: it receives the instruction register of the fetched instruction and returns a control word which the IF stage is fed with; such control word, also including the ALU opcode and the Register File Management Logic (RML) signals, is then pipelined to match the remaining stages of the datapath. Two additional inputs are needed to perfectly match the datapath pipeline and take into account possible alterations of the pipeline flow: `stall` and `misprediction`; more about their origin can be found in Sec.2.5.

1.1 Control Unit components

The control unit, whose scheme is shown in Fig.1.1, is made up of three main components:

- The Control Words ROM;
- the ALU Opcode LUT;
- the Register File Management Logic.

The **Control Words ROM** is a read-only memory containing the control words for the execution of every instruction throughout the whole pipeline, and is addressed by the opcode field of the input `IR` (see Fig.A.1 for IR fields). The ROM contains 62 words, each one composed of 30 bits for control signals; actually only 55 instructions exist, but their opcodes describe address spaces which are not always contiguous: see Appendix A for ISA characterization. Since the ROM is accessed only once for each instruction, the control word is extracted in its entirety and then pipelined, dropping the used signals stage by stage.

The **ALU Opcode LUT** is a look-up table which outputs the opcode for the ALU operation needed by the instruction. The LUT is indexed by the func field of the instruction register if its opcode corresponds to an R-type instruction, to differentiate among all possible R-type operations, whose opcode is always `0x00`; otherwise, for other instructions, the LUT gets addressed by the opcode field.

The **Register File Management Logic**, finally, is a small module useful to manage context switches of the windowed register file signalling subroutines calls and returns. It monitors the opcode and the source register of the input `IR`, raising the output `RF_CALL` if the instruction is a Jump & Link (`jal`, `jalr`) and `RF_RET` if the instruction is a Jump Register to Reg[`r23`]

(`jr r23`), which is the default register where the link address is saved (link register is not `r31`, see Subsec.2.3.3).

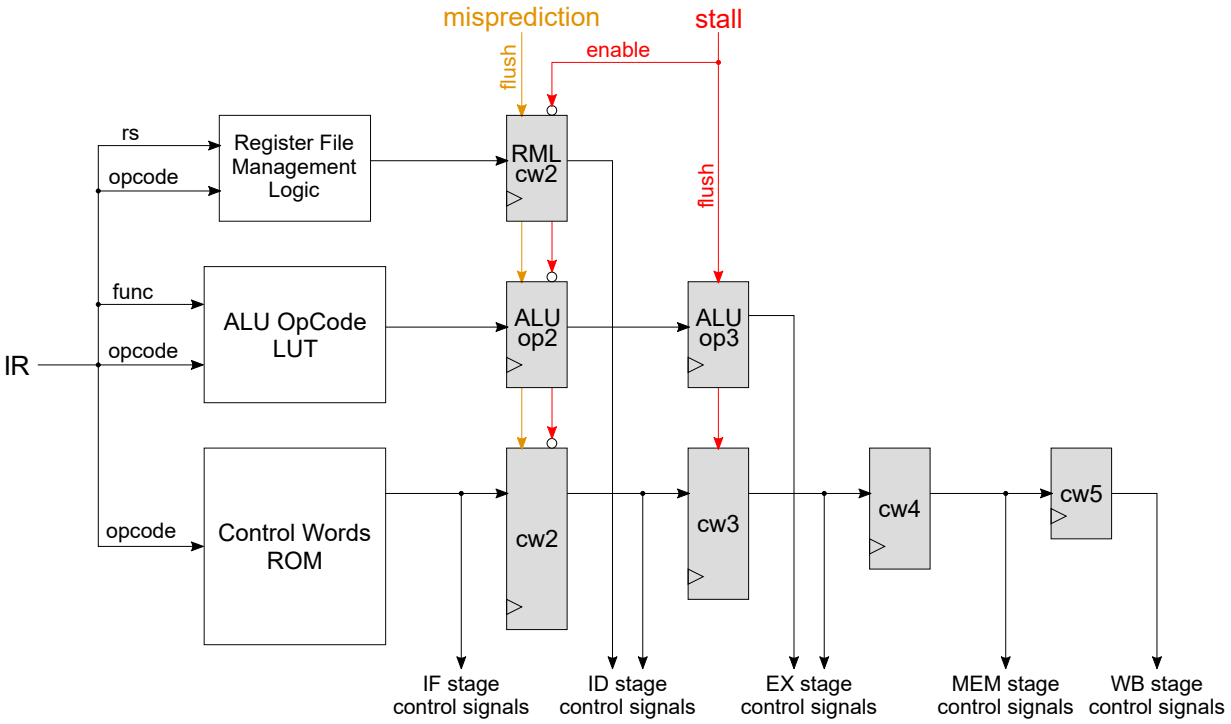

Figure 1.1: Scheme of the DLX hardwired control unit and its register to pipeline control signals.

1.2 Control signals

Each control word is composed of 30 bits, for a total of 26 control signals described in the following. A precise characterization of control signals for the ISA can be found in the attached Excel table.

IF control signals

- `MUXImmTA_SEL`: selection signal for the immediate to be used (Imm16 or Imm26) for Target Address computation;
- `NPC_LATCH_IFID_EN`: enable signal for Next Program Counter register of IF/ID;
- `TA_LATCH_IFID_EN`: enable signal for Target Address register of IF/ID;
- `PC_LATCH_IFID_EN`: enable signal for Program Counter register of IF/ID.

ID control signals

- `MUXImm_SEL`: selection signal for the extended immediate to be stored in ID/EX.Imm;
- `RF_RD1EN`: enable for read port 1 of Register File;
- `RF_RD2EN`: enable for read port 2 of Register File;

- RegA_LATCH_IDEX_EN: enable signal for register A of ID/EX;
- RegB_LATCH_IDEX_EN: enable signal for register B of ID/EX;
- RegIMM_LATCH_IDEX_EN: enable signal for Immediate register of ID/EX;
- LPC_LATCH_IDEX_EN: enable signal for Link Program Counter register of ID/EX.

EX control signals

- MUXB_SEL: signal to select the second operand for the ALU (ID/EX.B or ID/EX.Imm);
- ALUOUT_LATCH_EXMEM_EN: enable signal for ALU output register of EX/MEM;
- RegB_LATCH_EXMEM_EN: enable signal for register B of EX/MEM;
- LPC_LATCH_EXMEM_EN: enable signal for Link Program Counter register of EX/MEM.

MEM control signals

- DRAM_WE: enable for write port of Data Memory;
- DRAM_RE: enable for read port of Data Memory;
- DRAMOP_SEL: select whether to perform a store of a word or of a byte in the Data Memory;
- MUXLPC_SEL: select whether the Link Program Counter should be PC+4 or PC+8, where PC is the address of the Jump & Link instruction;
- MUXLMD_SEL: select which format and extension of Data Memory output has to be taken;
- LMD_LATCH_MEMWB_EN: enable signal for Load Memory Data register of MEM/WB;
- ALUOUT_LATCH_MEMWB_EN: enable signal for ALU output register of MEM/WB;
- LPC_LATCH_MEMWB_EN: enable signal for Link Program Counter register of MEM/WB.

WB control signals

- RF_WE: enable signal for write port of Register File;
- MUXWrAddr_SEL: selection signal for the write-back register address;
- MUXWB_SEL: selection signal for the data to write-back to the Register File.

Many control signals are only used to individually enable each pipeline register: this is a form of power optimization at architectural level, since enabling just the strictly necessary registers at each clock cycles only partially activates the combinational logic of each stage, leaving idle the unused logic and thus saving a considerable amount of dynamic power due to the reduced switching activity.

The other control signals, generated by the ALU Opcode LUT and by the RML are:

- RF_CALL: signals the call of a subroutine;
- RF_RET: signals the return from a subroutine;
- ALU_OPCODE: selects the operation that the ALU has to perform on its input operands.

CHAPTER 2

Datapath

The **datapath** is the component of the DLX in which, under the supervision of the control unit, instructions are actually processed and their related data is elaborated. The datapath I designed, whose simplified scheme is shown in Fig.2.1, is composed of the following five pipeline stages:

- **Instruction Fetch** (IF): a new instruction is fetched from the instruction memory, addressed by the Program Counter; Next Program Counter (PC+4) and branch/jump Target Address get computed, preparing one of them to become the new Program Counter basing on the signals coming from Branch & Jump Logic;
- **Instruction Decode** (ID): the register file is accessed and operands addressed by `rs` and `rt` fields are read; also, the 16-bit immediate field is extended; with Reg[`rs`] now available, the Branch & Jump logic checks, in case of a branch, if the prediction was correct and possibly corrects it;
- **Execution** (EX): the Arithmetic-Logic Unit performs the operation requested by the control unit and Reg[`rt`] is saved in case of a store instruction;
- **Memory** (MEM): the data memory is accessed in read (for loads) or write (for stores) mode with the address computed by the ALU; in case of a load, the output data gets extended and stored in the LMD; if a Jump & Link is being executed, the correct Link Program Counter is calculated;
- **Write-Back** (WB): when a write-back is needed, the correct data to store and its address are muxed and sent to the register file, whose write port gets enabled.

A detailed scheme of the datapath is shown in the attached PDF document.

In the following, the datapath will be analyzed by being broken into its main components, which are:

- integer Register File;
- integer Arithmetic-Logic Unit;
- Branch & Jump Logic;
- Hazard Detection Unit.

Memories are instead analyzed in Ch.3.

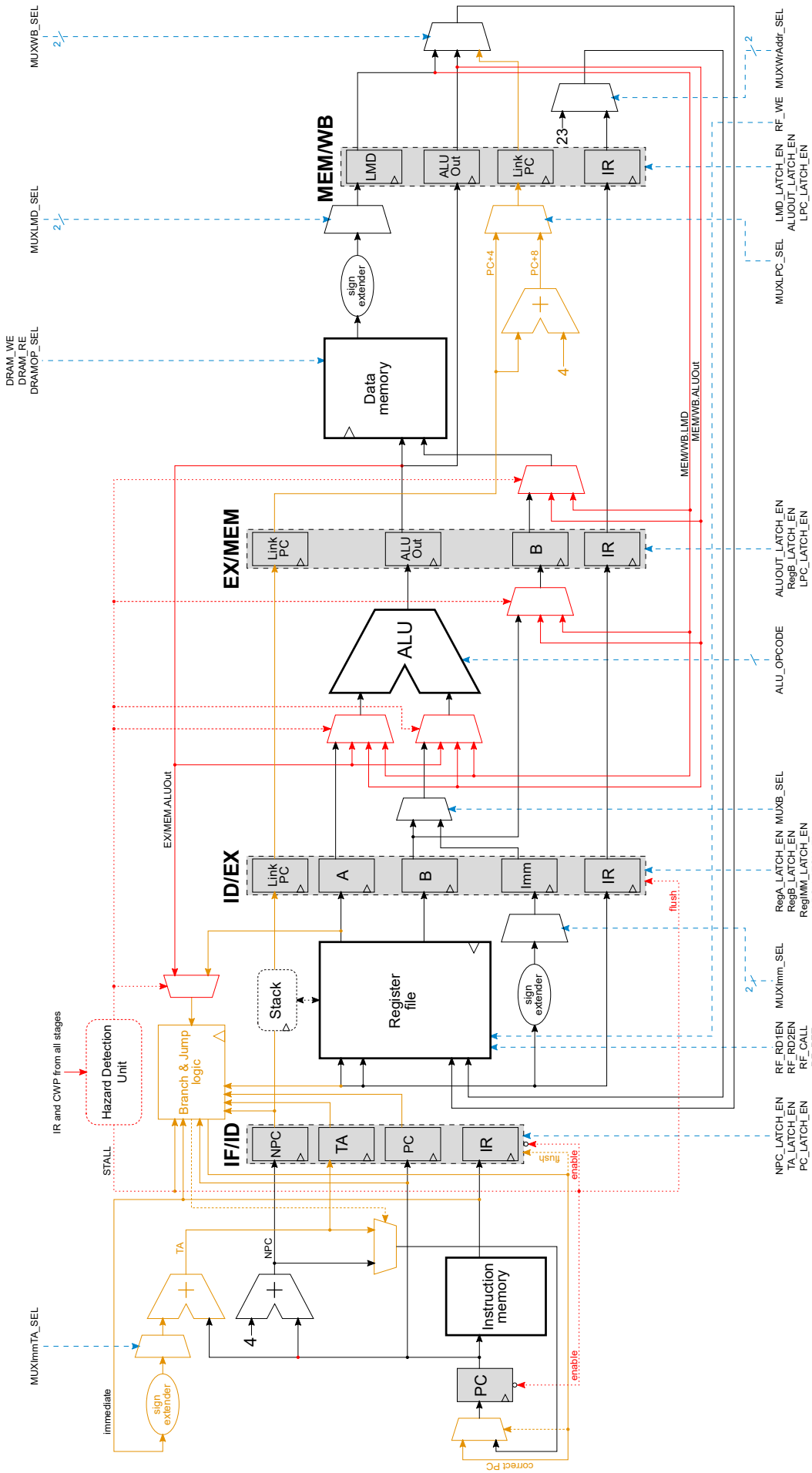

Figure 2.1: Simplified datapath scheme; yellow components are part of the Branch & Jump management logic, red components make up the hazard detection unit and the forwarding logic, while blue signals are control signals generated by the control unit.

2.1 Register File

The **Register File** is the portion of storage directly accessible by instructions being executed in the processor datapath; my implementation of the DLX Register File features windowing capabilities for quick context switches and possesses a total of 136 physical registers of 32 bits each, organized into 8 windows of 32 registers as in Fig.2.2. Each window is logically further subdivided into 4 groups of 8 registers each:

- GLOBALS: 8 registers for global variables, common to all windows;
- IN: 8 registers for input variables from parent routine (in common with OUT group of preceding window);
- LOCALS: 8 registers for local variables;
- OUT: 8 register for output variables for child subroutine (in common with IN group of following window).

The interface of the Register File, shown in Fig.2.2, offers two asynchronous read ports with read enable and one synchronous write port with write enable; the registers of a window are accessed by instructions with a 5-bit virtual address, ranging from 0 to 31, which has to be translated into a physical address, ranging from 0 to 135, by means of the Current Window Pointer (CWP).

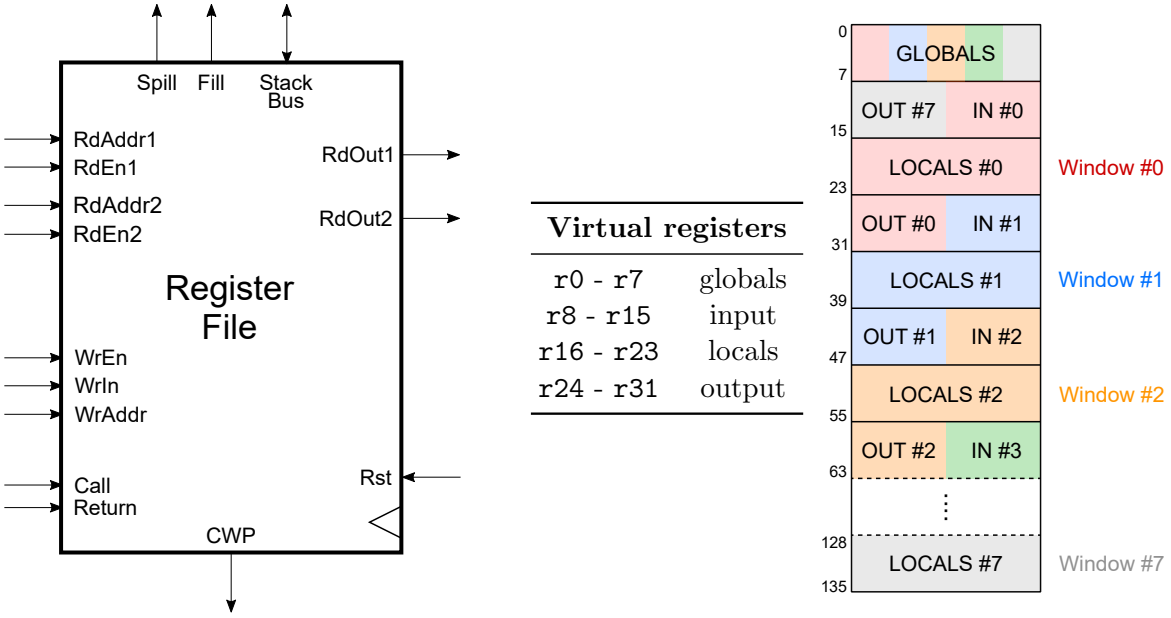

Figure 2.2: On the left, the windowed register file entity with its ports; on the right, a representation of the physical registers organization: boxes shaded with the same color belong to the same window.

The register interfaces with a stack memory, described in Sec.3.3, so that full windows can be spilled when the register file runs out of empty windows and then recovered when needed. The bus used to perform spill and fill transfers is four-registers-wide (i.e. 128 bits) so that 4 clock cycles only, instead of 16, are needed to complete the transfer of a full window of 16 registers.

2.1.1 Context switches

`CALL` and `RET` inputs, driven by the control unit, are used to manage context switches:

- when a subroutine **call** occurs, corresponding to a `jal` or `jalr` instruction, the CWP is increased by 1 to point to next window;
- when a subroutine **return** occurs, corresponding to a `jr r23` (Link Address is stored in `r23`, see Subsec.2.3.3), the CWP is decreased by 1 to point to parent routine window.

Context switches happen during Instruction Decode stage, while previous instructions belonging to old subroutines might still be in the pipeline; for this reason, we have to make sure that every instruction referring to a register in the register file is doing so with the correct CWP value, i.e. the same of the clock cycle during which the instruction was fetched. To solve this problem, the CWP is **pipelined**: when a change in the CWP occurs, its old value is propagated from stage to stage until Write-Back, and then, after no instruction of its related subroutine is in the pipeline anymore, it is discarded. The value of the CWP is also sent to the datapath by means of the `CWP` output, so that the Hazard Detection Unit is able to obtain physical registers addresses, as explained in Sec.2.4.

2.1.2 Read & Write operations

As already mentioned, every instructions has available only the 32 register of the routine it belongs to, so that it can address its virtual registers with a 5-bit address. The virtual address is then translated to a **physical address** to index one of the 136 actual registers thanks to the CWP, which points to the window of the current routine.

Since write operations occur during Write-Back stage, the CWP used to write-back is the last one of the pipeline; read operations, instead, use the first value of the pipeline, corresponding to Instruction Decode stage.

Due to the pipelined nature of the DLX, two instructions might want to access the register file at the same time, one during its Write-Back stage, and the other during Instruction Decode. If the instruction decoding its operands wants to read a physical register which is being written by the other instruction, a **structural hazard** occurs. To solve it, the input data from the write port is simply forwarded to the output port whenever the virtual addresses employed by the two instructions, even if different, correspond to the same physical register.

As in the original DLX, the register `r0` is tied to zero, so that any read operation outputs a 32-bit zero and write operations on such address have no effect. In my implementation, registers for global variables are placed in the lowest addresses of the register file (i.e. from 0 to 7); thanks to this, the address `00000` indexes a global register and not the first register of each window, so that only one physical register has to be tied to 0.

2.1.3 Spill & Fill operations

During a context switch, if the CWP reaches the Saved Window Pointer (SWP), a **spill** or a **fill** is needed, basing on the direction from which CWP approached SWP:

- after a call, CWP gets increased by 1; if the new `CWP = SWP-1`, then it means that no more empty windows are available, and we have to spill the window pointed by SWP;
- after a return, CWP gets decreased by 1; if the new `CWP = SWP-1`, then it means that the content of the windows where we are returning has been previously spilled, and now needs

to be filled back from the stack.

In any case, a transfer to/from the stack involves 16 registers (since IN and OUT groups are shared by two adjacent windows, only two groups of 8 registers each are to be transferred for each spill/fill): with a 4-registers-wide stack bus, **4 clock cycles** are needed for each transfer (plus 1 additional clock cycle for the call/return itself). Moreover, since the instruction requesting the context switch has to wait for the new window to be available, the whole pipeline has to stall, which is why `SPILL` and `FILL` signals are not only used by the stack to synchronize the transfers, but also by the datapath to issue **stalls**. The maximum number of windows which can be spilled, and thus the maximum number of nested subroutine, depends exclusively on the size of the stack.

2.2 Arithmetic-Logic Unit

The **Arithmetic-Logic Unit** (ALU) is the very piece of processor hardware which performs the main computation with the incoming data during the Execution stage.

The ALU I designed has two 32-bit input operands, `A` and `B`, an input for the ALU opcode and a 32-bit output. For the opcode I have used an implicit encoding, shown in the following piece of VHDL code along with its enumeration.

```
-- Implicit encoding for ALU OpCode
type aluOp is (
   NOP, LSL, RSL, RSA, ADD, SUB, ANDalu, ORalu,
   XORalu, EQ, NE, LT, GT, LE,  GE,  LTU, GTU,
   LEU, GEU, MULT, THR_B
);
```

To be able to perform all the listed operations, the ALU is provided with the following components:

- an integer **Prefix Adder**, based on the Intel Pentium 4 one, which also operates as a **Subtractor**;
- a radix-4 **Booth Multiplier** based on carry-save adders;
- a two-level NAND tree **Logic Unit**, based on the logic unit of the UltraSPARC T2;
- a structural **general-purpose Comparator**;
- a **general-purpose Barrel Shifter**.

The described ALU, whose structure is sketched in Fig.2.3, has been optimized at architectural level in terms of power and area. As far as power is concerned, **guarded evaluation** has been employed: the outputs of all components are not simply muxed basing on the ALU opcode, but their inputs are buffered in latches in such a way that their internal nets activate exclusively when actually needed and not at each operation, greatly reducing the overall switching activity and thus the dynamic power consumption.

To improve the area, **resource sharing** has been exploited: apart from additions and subtractions, an adder is also needed for comparisons and multiplications; a further internal muxing, based on the ALU opcode, has been built in order to share the same adder among addition, subtraction, multiplication and comparison operations. In the following, post-synthesis

area and dynamic power reports are shown for comparison. Note that both synthesis have been performed without any constraint.

Non-optimized ALU

```
Combinational area:  4297.762021
Noncombinational area:  188.860006
Total cell area:  4486.622027

Cell Internal Power = 583.2666 uW
Net Switching Power = 565.0035 uW
Total Dynamic Power = 1.1483 mW
```

Optimized ALU

```
Combinational area:  3296.272012
Noncombinational area:  723.254024
Total cell area:  4019.526036

Cell Internal Power = 133.6019 uW
Net Switching Power = 110.2785 uW
Total Dynamic Power = 243.8804 uW
```

From the reports, it can be noticed how sharing decreased the combinational area of the ALU of about 23%; however, due to guarding evaluation, which successfully decreased dynamic power of 79%, the additional latches increased noncombinational area of 283%, adding up for an overall area decrease of 10% with respect to the non-optimized ALU.

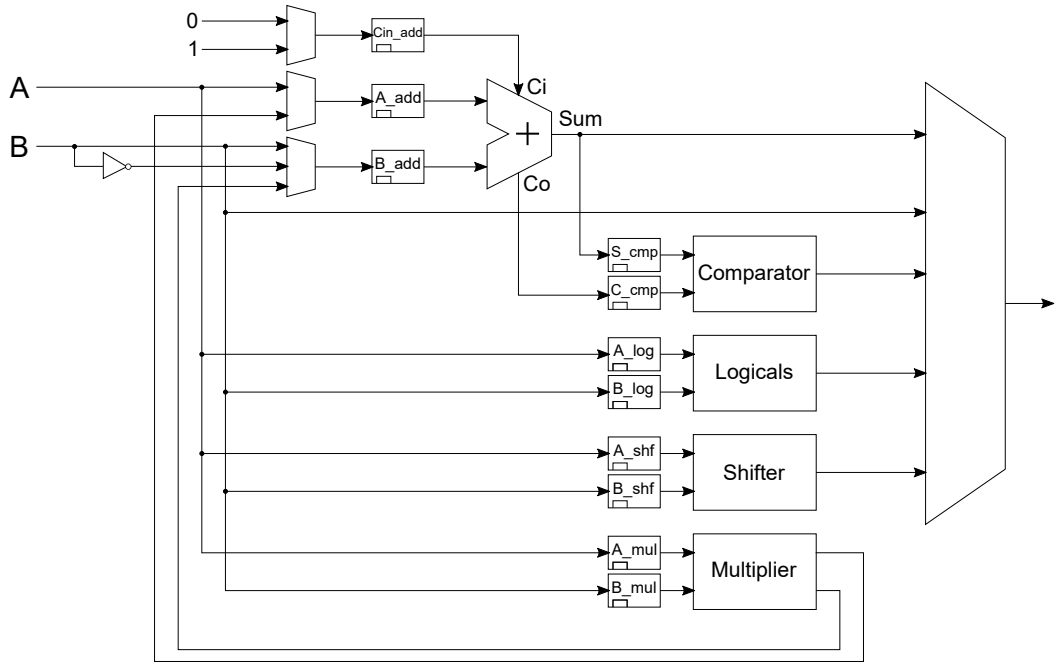

Figure 2.3: The simplified block scheme of the ALU; note that this is only an intuitive representation of the structure, highlighting its main details.

2.2.1 Adder

The 32-bit integer adder used in the ALU, which has been replicated also for Next Program Counter, Target Address and Link Program Counter computations, is based on the Pentium 4 sparse tree adder, belonging to the family of *Prefix adders*. Its structure, shown in Fig.2.4, is composed of a **sparse tree Carry Generator** for efficient carry look-ahead (CLA) logic which generates a partial carry every 4 bits. The resulting 8-bit carry array is sent to the **carry-select-based Sum Generator**, composed of 8 RCA-based carry-select adders, which

generates the final sum in blocks of 4 bits each. This structure is an architectural optimization of a basic CLA, and represents a good trade-off between area and latency.

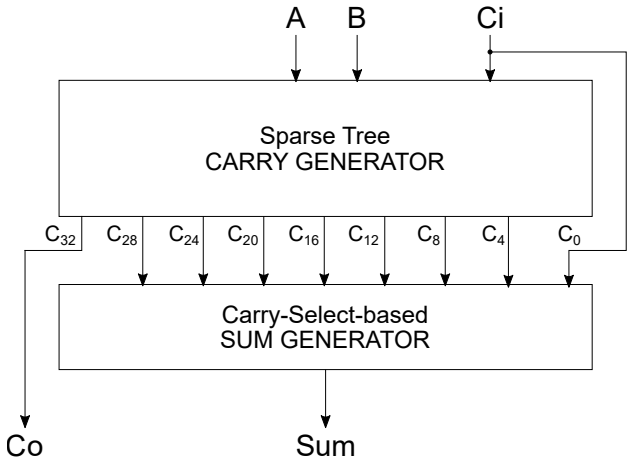

Figure 2.4: Structure of the designed sparse tree adder.

In the ALU, the inputs of the adders are muxed so that it can be used for additions, subtractions and multiplications (taking as inputs the partial results of the multiplier component); also, its outputs are sent both to the output of the ALU and to the comparator input.

2.2.2 Multiplier

For the integer multiplier, a structure based on the **modified radix-4 Booth Multiplier** algorithm, shown in Fig.2.5, has been employed.

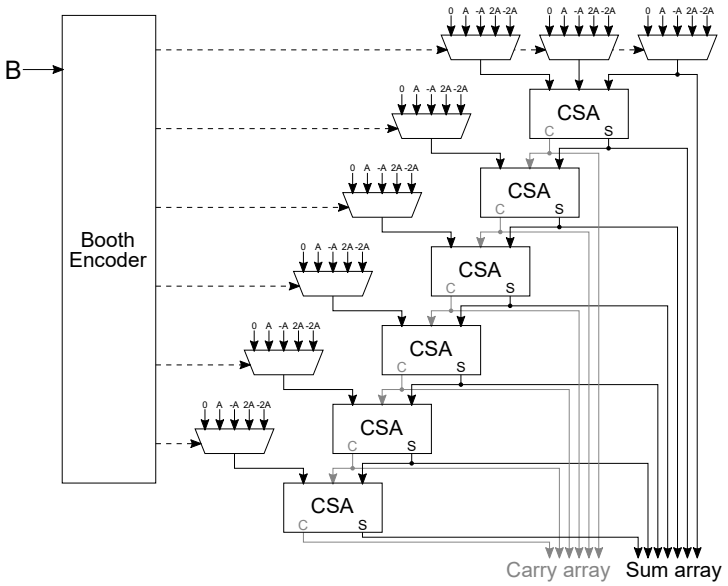

Figure 2.5: Structure of the designed Booth multiplier, showing the 6 CSA in series; note that gray lines stand for carry-related data and dashed lines for multiplexers drivers.

The multiplier takes as inputs the 16 less significant bits of both `A` and `B` operands and returns

a 32-bit output. As a form of architectural latency-area trade-off, original ripple-carry adders for partial sums have been replaced by a series of **carry-save adders**. Also, the multiplier lacks of a final adder to add up the final carry and sum arrays; the multiplier entity outputs indeed such two arrays, leaving the final sum to the ALU adder.

When it comes to obtain shifted values of the `A` multiplicand, as a form of further optimization, a logic to perform only one shift and use its results for all partial sums has been developed: the idea is to operate at each partial sum stage on a portion of data which is two bits forward, which intrinsically correspond to shift `A` two bits left at each iteration, as the Booth algorithm requires; in this way, the width of the carry-save adders can be the same for all instances and the combinational logic does not have to deal with unnecessary zero values. Bits which would be only summed to zeros are instead directly brought to the output port.

In the following, post-synthesis area and timing reports are shown to compare a non-optimized RCA-based Booth multiplier and the described one. Both synthesis are unconstrained and the arrival time refers to the path with maximum delay. For a consistent comparison, and solely for the purpose of this report, an adder like the one described in Subsec.2.2.1 has been synthetized together with the optimized Booth multiplier.

Non-optimized Multiplier

```
Combinational area:  1945.258007
Noncombinational area:  0.000000
Total cell area:  1945.258007

Startpoint:  A[0] (input port)
Endpoint:  Y[31] (output port)
  data arrival time 4.41 ns
```

Optimized Multiplier

```
Combinational area:  1380.540015
Noncombinational area:  306.432003
Total cell area:  1686.972018

Startpoint:  B_i[1] (input port)
Endpoint:  Y[31] (output port)
  data arrival time 3.08 ns
```

2.2.3 Logic Unit

The logic unit I designed for the ALU of my DLX, shown in Fig.2.6, resorts to the **UltraSPARC T2 Logic Unit** structure, composed of two levels of 32-bit-wide NAND gates and driven by means of 4 control signals S_0, S_1, S_2 and S_3 to select the logical operation to perform. Note that such selection signals are first extended to 32 bits and then sent to NAND gates inputs.

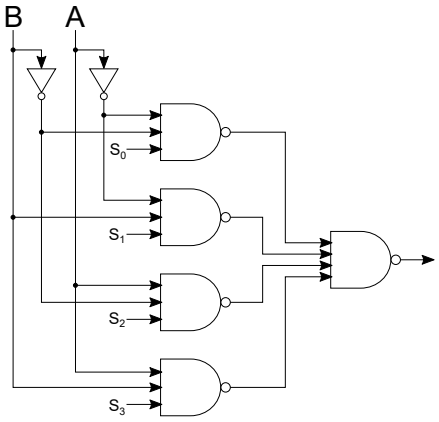

Figure 2.6: Structure of the designed logic unit; note that all the represented wires are actually 32-bit-wide buses.

In addition to its simplicity of implementation and functioning, this structure represents a very good trade-off in terms of area and latency: only NAND gates are used and no muxing is needed. The implemented boolean function is

$$\mathrm{OUT} = \overline{A}\,\overline{B}\,S_0 + \overline{A}\,B\,S_1 + A\,\overline{B}\,S_2 + A\,B\,S_3 \tag{2.1}$$

where each literal represents a 32-bit operand. From Eq.2.1 we can derive the following selection signals for the logic operations needed by the ISA.

$$\texttt{S}_3\texttt{=1},\ \texttt{S}_2\texttt{=0},\ \texttt{S}_1\texttt{=0},\ \texttt{S}_0\texttt{=0} \rightarrow \texttt{and}$$
$$\texttt{S}_3\texttt{=1},\ \texttt{S}_2\texttt{=1},\ \texttt{S}_1\texttt{=1},\ \texttt{S}_0\texttt{=0} \rightarrow \texttt{or}$$
$$\texttt{S}_3\texttt{=1},\ \texttt{S}_2\texttt{=0},\ \texttt{S}_1\texttt{=0},\ \texttt{S}_0\texttt{=1} \rightarrow \texttt{xor}$$

2.2.4 Comparator

The **general-purpose comparator** I employed for the ALU resorts to a structure, shown in Fig.2.7, capable of efficiently performing comparisons between two signed or unsigned integers. The comparator exploits the result of the subtraction `A − B` of the two operands, which is computed by an external adder (the one included in the ALU); by analyzing the sum and the carry outputs of the adder, the correct outputs to signal the relative magnitude of `A` and `B` are set to `1`.

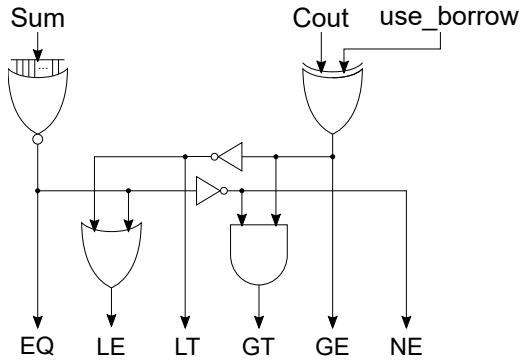

Figure 2.7: Structure of the designed general-purpose comparator.

The functioning of the comparator is based on the 2's compliment subtraction of the operands: the adder performs `A − B` and the comparator verifies whether such operation required a borrow or not; note that, in 2's compliment subtractions, *borrow flag* is the compliment of *carry flag*. When `A` and `B` are unsigned, or they have the same sign, there are no ambiguities: if `A − B` requires a borrow (borrow = `1`, carry = `0`), then `A < B`, while when it does not (borrow = `0`, carry = `1`) then `A ≥ B`. The same applies for signed data with same sign.

On the other hand, when `A` and `B` are signed data with opposite signs, the carry flag assumes an opposite meaning: if `A < B` then `A` is negative and `B` is positive, thus `A − B` is always a sum of two negative numbers which always returns a carry = `1` due to the sum of their sign bits; else, if `A ≥ B` then `A` is positive and `B` is negative, making `A − B` a sum of two signed positive number which will always return a carry = 0 due to the `0` sign bits, which absorb a possible carry out.

In conclusion, we need to **compliment the carry flag** when a signed comparison of operands with different signs has to be performed: to do so, an additional `use_borrow` input is used, setting it to `0` for unsigned operations and to `A(31) xor B(31)` for signed ones.

2.2.5 Other operations

As for the shifter, a simple **general-purpose Barrel Shifter** has been implemented exploiting the default VHDL shift operators; driven by a 2-bit selection signal, it is capable of three operations, which are performed on the operand `A` basing on the number of bits specified by the 5 least significant bits of `B`: logically shift left, logically shift right and arithmetically shift right.

A specific ALU behaviour has been implemented for the **load high immediate** instruction: `lhi` requires to extend the 16-bit immediate field with 16 zeros in the low half; since immediate extension is performed during Instruction Decode stage, when a `lhi` occurs the ALU simply forwards the operand `B`, which contains the extended immediate, from the input to the output port.

Finally, similarly to the components inputs, the **output** of the ALU is guarded with a latch: it indeed switches only when a new operation is actually performed; when the ALU opcode corresponds to a `nop`, it persists in the previous state.

2.3 Branch & Jump Logic

The instruction set of my DLX implementation, shown in Appendix A, includes 6 instructions which are able to modify the value of the Program Counter (PC): `beqz`, `bnez`, `j`, `jal`, `jr`, `jalr`. Due to their nature, these instructions can create control hazards: with the aim to handle them, I have developed a specific module embedded within the DLX datapath, the **Branch & Jump Logic**. Its main characteristics are:

- 32-entries **Branch History Table** (BHT) with **2-bit prediction** scheme to predict branches (`beqz`, `bnez`) outcome during IF stage;
- extra adder, instance of the one described in Subsec.2.2.1, to anticipate the computation of branches and J-type jumps (`j`, `jal`) **target address** to IF stage;
- **flush** of wrongly fetched instruction in case of branch misprediction and PC correction during ID stage;
- **anticipation of J-type jumps** management to IF stage, to completely eliminate their related control hazards;
- **one delay slot for I-type jumps** (`jr`, `jalr`).

The interface of the entity is shown in Fig.2.8.

2.3.1 Additional datapath logic

The additional logic needed in the datapath to correctly handle control hazards is highlighted in yellow in Fig.2.1. It consists of:

- a **multiplexer** in IF stage to choose the *immediate* to sum to PC (Imm16 for branches, Imm26 for I-type jumps) to compute the Target Address (TA), driven by the signal `MUXImmTA_SEL` from control unit;
- an extra **adder** in IF stage to sum the selected immediate to PC to compute the *Target Address*; since such computation happens in IF stage directly using PC instead of NPC, a modification to the assembler script has been needed, as described in Appendix B;

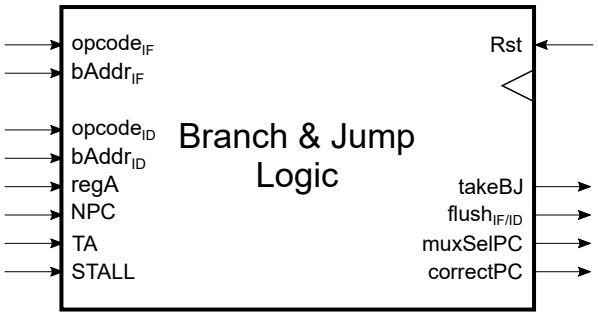

Figure 2.8: Entity of the Branch & Jump Logic; $\texttt{opcode}_{\texttt{IF}}$ is taken directly from instruction memory output, while $\texttt{bAddr}_{\texttt{IF}}$ is taken from the PC register; $\texttt{opcode}_{\texttt{ID}}$, $\texttt{bAddr}_{\texttt{ID}}$, `NPC` and `TA`, instead, come from IF/ID registers.

- a **multiplexer** in IF stage, driven by `takeBJ` output of Branch & Jump Logic, to decide next value for PC (PC+4 for normal instructions, TA for branches and J-type jumps);
- another **multiplexer** at the input of PC register, driven by `muxSelPC` output, to be able to correct the PC when needed (branch misprediction or I-type jump); this multiplexer has a priority over the previous one, so that even if another branch or a jump are fetched after a mispredicted branch or a jump register, the correct PC is loaded anyway;
- an **adder** and a **multiplexer** in the MEM stage to be able to select between PC+4 and PC+8 for the Link Address to write-back respectively for `jal` and `jalr`.

2.3.2 Branches management

Prediction (IF stage)

Branch prediction is based on a BHT with 32 entries of 2 bits each; if the instruction fetched during IF stage is a branch ($\texttt{opcode}_{\texttt{IF}}$ = `beqz` or `bnez`), the memory is addressed with the least significant 5 bits of the branch address (input $\texttt{bAddr}_{\texttt{IF}}$), excluding the very 2 LSBs which are always 0 (because PC addresses single byte of instruction memory).

Basing on the content of the addressed entry, the prediction is computed and used to drive `takeBJ` output to select TA if the branch is predicted as taken, or NPC otherwise. Since the memory is small and is addressed only by a portion of the bits of a branch address, more than one branch might refer to the same BHT entry; for this reason, the prediction of the BHT is just an hint, which may need to be corrected during ID stage, when the operand to compare against zero is actually available.

Verification (ID stage)

If the instruction in the ID stage is a branch ($\texttt{opcode}_{\texttt{ID}}$ = `beqz` or `bnez`), the prediction used in the IF stage (the BHT entry indexed by $\texttt{bAddr}_{\texttt{ID}}$) is compared against the actual outcome of the branch, after having fetched the register A from the register file.

If the prediction was correct, no action is taken but update the prediction itself if needed; if instead a misprediction occurred, IF/ID registers are flushed on next clock rising edge and the correct PC (chosen between `NPC` and `TA` inputs, taken from IF/ID) is latched to PC register, raising `muxPC_SEL`; also, the related prediction in the BHT gets updated.

Note that, logically, the BHT memory belongs to the IF/ID-level registers so that, if a stall occurs, the update of the predictions needs to be stalled too (because the branch in the ID stage stalls due to register A not being available), in order for it to correctly happen.

Every 2-bit entry of the BHT represents a 4-stages finite state machine (FSM), whose current state corresponds to the prediction sent to IF stage. At reset, every entry corresponds to a *Not Taken* prediction. The FSM state diagram is represented in Fig.2.9.

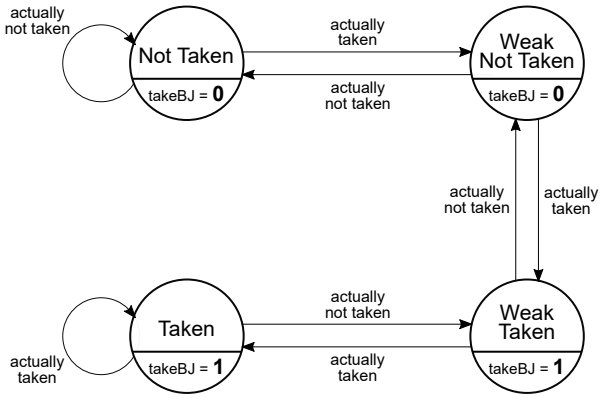

Figure 2.9: The state diagram of the Moore FSM representing the 2-bit prediction scheme of the designed BHT.

2.3.3 Jumps management

J-type Jumps

J-type Jumps are unconditional jumps relative to the current PC; they are implemented with `j` and `jal` instructions. When one of these two instructions is fetched, the `takeBJ` output is unconditionally set to `1`, so that the Target Address is used.

As far as the Jump & Link (`jal`) is concerned, it also saves the address of its following instruction (PC+4) to allow its use as a return address. The so called Link Address is written-back in register `r23`, and not in `r31` as in the classical DLX architecture: since a windowed register file is employed, `r31` is indeed an output register, which is thus accessible by a potential child subroutine that could corrupt it; `r23` is instead the last register of the LOCALS group, which is accessible only by the routine which writes it.

I-type Jumps

I-type jumps, implemented by `jr` and `jalr` instructions, are absolute jumps to the address contained in a register. Since register A is fetched only during ID stage, the instruction fetched after a Jump Register is still the one pointed by NPC, PC+4.

In ID stage, instead, the content of the register A is available and is latched to the PC register raising `muxPC_SEL`; however, to maximize the instructions per cycle (IPC), the previously fetched instruction is not flushed, so that a **delay slot** exists for `jr` and `jalr` instructions. If no instructions are available to fill the delay slot, it must be filled with a `nop` at compile time.

Due to the delay slot, the Link Address which must be written-back by `jalr` is PC+8, to point to the second instruction after the Jump & Link. To do so, an extra adder and a mul-

tiplexer, driven by `MUXLPC_SEL` control signal, are instantiated in MEM stage; the multiplexer output is PC+4 for `jal` and PC+8 for `jalr`.

2.4 Hazard Detection Unit & forwarding logic

When a data dependency exists between the instructions being executed in a pipeline, a series of data hazards can occur. The one which can actually create problems in my implementation of the DLX architecture is the *read-after-write* (RAW) hazard: it is the result of an instruction reading a register which is the destination register of a previous instruction.

To solve the mentioned hazard, which can happen in all the forms listed in the attached Excel table, I have designed a combinational **Hazard Detection Unit** (HDU) which constantly monitors the instructions in every stage and their operand registers, taking action when they are in conflict. The module is able to issue stalls and drive 5 multiplexers for the forwarding logic; they are highlighted in red in Fig.2.1. The entity interface is shown in 2.10.

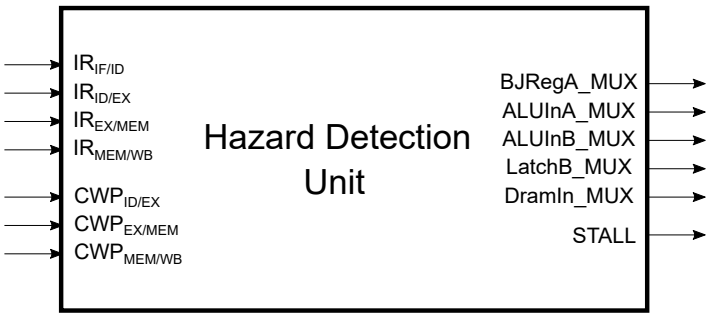

Figure 2.10: Entity of the Hazard Detection Unit.

Since a windowed register file is employed in the datapath and context switches are supported, it may happen that instructions belonging to different routines refer to the same physical register with different virtual addresses. In order to correctly detect data hazards, the HDU has first to **translate virtual addresses** taken from instruction registers to physical addresses and then compare them. To perform such translation, the Current Window Pointer related to each stage of the pipeline has to be considered, to make sure to correlate the instruction in each stage to the subroutine it actually belongs to.

2.4.1 Forwarding Logic

The Hazard Detection Unit drives the control signals for **five multiplexers**, listed along with their possible inputs in Tab.2.1.

Basing on the relative positions of two instructions, on their types and on their operands, the HDU drives the selection signals of such multiplexers. As an example, if a `sub r1,r2,r3` is followed by an `addi r4,r1,#8` then EX/MEM.IR[rd] = ID/EX.IR[rs] and the output `ALUInA_MUX` is set to `01` to forward the result of the subtraction from EX/MEM.ALUOut to the input A of the ALU.

It is important to point out that, for a consistent hazards detection, *write-after-write* (WAW) hazards have to be taken into account too. For example, suppose that the following three instructions are in the pipeline.

Table 2.1: List of the forwarding logic multiplexers with all their possible inputs.

MULTIPLEXER	POSSIBLE INPUTS	SELECTION SIGNAL
Branch & Jump Logic register A input	Register File Read output 1 EX/MEM.ALUout	`BJRegA_MUX`
ALU input A	ID/EX.A EX/MEM.ALUOut MEM/WB.ALUOut MEM/WB.LMD	`ALUInA_MUX`
ALU input B	ID/EX.B or ID/EX.Imm EX/MEM.ALUOut MEM/WB.ALUOut MEM/WB.LMD	`ALUInB_MUX`
EX/MEM.B register	ID/EX.B MEM/WB.ALUOut MEM/WB.LMD	`LatchB_MUX`
Input data of Data Memory	EX/MEM.B MEM/WB.ALUOut MEM/WB.LMD	`DramIn_MUX`

```
addi r1, r0, #5
subi r1, r0, #1
mult r3, r1, r2
```

To respect the data flow, the `mult` has to use the value `-1` for `r1` because it is the result of the latest write operation, and not `5`. To grant this property, in the processes which implement the HDU behaviour, assignments related to instructions in earlier stages of the pipeline are the last ones, so that write operations of *younger* instructions have priority over *older* ones.

The only data hazards which are not managed by the HDU are the ones generated by Jump & Link instructions, which write their Link Address in register `r23`; if a subsequent instruction has its `rs` or `rt` source field pointing to the same physical register to which a `jal` or a `jalr` writes its Link Address and a RAW occurs, inconsistent data is read: the hazard is thus to be managed at compile time, inserting independent instructions in between. A space of at least two instruction slots is needed between the Jump & Link and the instruction needing `r23`; a third slot is not needed since the register file is able to manage the hazard anyway, basing on the physical addresses of read and write ports.

2.4.2 Stalls

When forwarding cannot solve data hazards one or more **stalls** might need to be issued. Suppose, for example, that an `add r3,r1,r2` follows a `ld r1,20(r0)`; during the EX stage of the `add`, Mem[20] has not been accessed yet and the data does not exist in the pipeline, so that a stall is needed before activating the forwarding from MEM/WB.LMD to input A of the ALU.

The behaviour and the consequences of a stall in the datapath are analyzed in Sec.2.5.

2.5 Pipeline flow alterations

As mentioned in the description of previous datapath components, there are several reasons for which the ordinary pipeline flow might be altered, but they all converge in two events:

- **stall** of IF and ID stages and issuing of a **bubble** (i.e. a `nop`) which propagates starting from EX stage;
- **flush** of the IF stage, which gets filled with a new instruction, and fetch of a **bubble** propagating from ID stage.

The **stall** signal might be raised for two main reasons: a stall, raised by the HDU, is needed to avoid a *data hazard* or the windowed register file is in the middle of a *spill* or *fill* operation. The VHDL assignment statement for the stall signal is indeed the following.

```
STALL <= HDU_stall or RF_spill or RF_fill;
```

`STALL` is an active-low enable for PC and IF/ID registers: when it is `1`, such registers are not updated on the next clock rising edge and their content remains the same, making the instructions in IF and ID stage stall. Also, `STALL` acts as a synchronous reset for ID/EX registers: when raised, their content gets flushed on the next clock rising edge so that a bubble is issued to EX stage.

The **flush** signal is instead driven exclusively by Branch & Jump Logic and it is raised when a *branch misprediction* occurs: if during the ID stage of a branch the prediction used during IF stage turns out to be different from the actual outcome of the comparison against 0, the wrongly fetched instruction needs to be aborted. `BJ_flushIFID` thus act as a synchronous reset for IF/ID registers, which are flushed to fetch a bubble that starts propagating from the ID stage, while the correct address is latched to PC register to fetch the correct instruction.

Registers of the IF/ID group are subject to both `STALL` and `BJ_flushIFID`. To grant a consistent behaviour when a stall and a misprediction occur during the same clock cycle, the active-low enable `STALL` has priority over the synchronous flush `BJ_flushIFID`.

CHAPTER 3

Memories

For the purpose of the simulation, three memories have been designed and connected to the DLX Datapath and Control Unit:

- Instruction Memory;
- Data Memory;
- Stack.

Such components are instantiated in the simulation wrapper only and they have not been considered for synthesis and physical design.

3.1 Instruction memory

The **Instruction Memory** is the piece of storage where the firmware of the DLX is loaded. It has been designed as a read-only memory, which gets loaded only at reset with the content of the file *test.asm.mem*, placed in the simulation directory. Each word of the memory is 32-bit wide, to contain one instruction. For the simulations, a 256-words Instruction Memory has been instantiated, corresponding to a total of 1 MB storage for instructions. If the firmware file is shorter than the memory size the remaining words get filled with `nop` instructions.

Since the PC addresses single bytes of the Instruction Memory, its value is divided by 4 before actually using it, so that 4-bytes instruction words can be correctly addressed.

3.2 Data memory

The **Data Memory** is a random-access memory immediately under the Register File in the memory hierarchy. It is the source for load instructions and receives the output of store instructions. As in the classical DLX architecture, the data memory I implemented is big-endian and byte-addressable, and has a default word size of 4 bytes; it is provided with an asynchronous read port with read-enable and a synchronous write port with write-enable and has the possibility to perform store operations on single bytes (`sb`, taking only the least significant byte of the input data) and on whole words (`sw`) basing on the control signal `DRAMOP_SEL`. Read and write ports share the same address bus since only one operation at a time can take place.

To be able to perform all kinds of loads (`lb`, `lbu`, `lhu`, `lw`), the memory outputs all possible formats of the data corresponding to the given address (word, half-word, byte); their extension

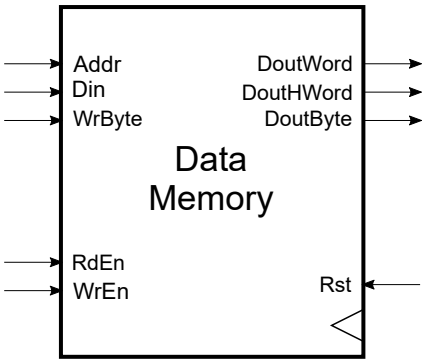

Figure 3.1: Data Memory entity.

and selection is then left to the datapath. When indexing words or half-words, addresses must be aligned to such formats; since exceptions have not been implemented, addresses of words and half-words are automatically aligned to be divisible respectively by 4 and by 2. The interface of the Data Memory is shown in Fig.3.1.

At reset, the content of the memory is initialized to the one of the file *data.mem*, placed in the simulation folder. If the file is shorter than memory size, the remaining words are initialized to 0. For the simulations, a 128-words Data Memory has been instantiated, corresponding to a storage of 512 kB.

3.3 Stack

The **Stack** is a LIFO memory used to hold registers when the number of nested subroutines is higher than the number of windows in the register file, so that no more physical space is available for a new subroutine. The Stack I implemented is filled in an increasing fashion, and its stack pointer (SP) indexes the last full slot.

As long as the `SPILL` or `FILL` inputs coming from the register file are at `1`, the Stack establishes a synchronous transfer with the register file, exchanging registers by means of the data bus. Such bus has a width of 128 bits, allowing a transfer of 4 32-bits registers per clock cycle and reducing the needed stalls.

For the simulations, a stack capable of holding 128 32-bit registers has been instantiated, corresponding to a storage of 512 kB in which a maximum number of 8 windows can be spilled.

CHAPTER 4

Synthesis

After the design phase and a series of tests exploiting many asm benchmarks for simulations, the verified design has been synthesized with the use of Synopsys Design Vision. As already mentioned, memories discussed in Ch.3 have not been synthesized: a different wrapper has been indeed used, the file `/src/a-DLX_syn.vhd`, connecting exclusively the datapath and the control unit and leaving the memories interface to the entity ports. In order to verify and further optimize the synthesized netlist, a post-synthesis simulation (exploiting another wrapper, `/src/a-DLX_postsyn_sim.vhd`) has been carried out.

For the technology mapping stage, the Nangate Open Cell Library has been employed: it is an open-source, 45 nm standard-cell library provided for the purposes of research, testing, and exploring EDA flows.

4.1 Synthesis and optimization

Since no particular computational time or resources constraint were imposed for the synthesis step, I decided to exploit the **ungroup method**, which requires more memory and takes longer to compile the design but gives the best synthesis results. The ungroup method is based on the `ungroup` command, which flattens the hierarchy of the designs references specified as arguments; this exposes an higher number of optimization possibilities, but makes the design more complex to analyze.

The synthesis script I have written for this purpose is `/syn/synthesis.tcl`. With ungroup compilation, the design proved to be able to reach a clock period as small as 1.10 ns; however, a synthesis with 1.25 ns, corresponding to a clock frequency of 800 MHz, guaranteed an appropriate margin, also giving room for area and power optimization, which would otherwise explode. To ensure further margins for the subsequent physical design, clock non-idealities have been considered in the synthesis process. The following piece of code sets up a synthesis with **dynamic power and area optimization under latency and clock period constraints**. The `0.0` value set for `max_area` and `max_dynamic_power` tells the optimization engine to try its best to minimize such figures.

```
set delayConstraint 1.25
# Sequential constraint
set clockName "Clk"
create_clock -period $delayConstraint $clockName
```

```
# Clock network tolerances setup
set_clock_uncertainty 0.05 $clockName
set_clock_transition 0.05 $clockName
set_clock_latency 0.05 $clockName

# Combinational constraint
set_max_delay $delayConstraint -from [all_inputs] -to [all_outputs]

# Area & dynamic power minimization
set_max_area 0.0
set_max_dynamic_power 0.0
```

Before the compilation, performed with `map_effort` set to `high`, all designs references in the DLX top level design get recursively flattened; the final result after the `ungroup` command is a netlist of *gtech* cells without any hierarchy which maximizes the potential optimizations.

```
# Flatten hierarchy of all designs
ungroup -all -flatten
# Compile
compile -map_effort high
```

4.2 Post-synthesis simulation

With the aim to perform a post-synthesis simulation with the netlist obtained after the ungroup compilation, I created another wrapper to connect the synthesized DLX netlist, described in `/syn/netlists/DLX_postsyn_ungroup.vhdl`, and the non-synthesized memories.

To be able to simulate the netlist, I retrieved and compiled in ModelSim the VITAL (VHDL Initiative Towards ASIC Libraries) format of the Nangate Open Cell Library. The ModelSim script for the post-synthesis simulation is `/sim/post_syn_simulation.tcl`. The employed testbench is the same of the simulations performed during the design phase.

With the post-synthesis simulation, not only was I able to test the same asm benchmarks used before the synthesis phase to verify the correctness of the design, but I also collected more accurate power information, both in SAIF and VCD formats, as in the following piece of code. The benchmark which has been used to gather such information is `/test/vectorminmax.asm`, which explores a decent amount of features of my DLX.

```
power add -in /tb_dlx/DLX_uut/DLX/*
power add -internal /tb_dlx/DLX_uut/DLX/*
vcd  file "${outputsDir}/dlx_swa.vcd"
vcd add /tb_dlx/DLX_uut/DLX/*

# Run simulation
run 245 ns
# Report switching activity to file
power report -all -bsaif "${outputsDir}/dlx_swa.saif"
power report -all -file "${outputsDir}/dlx_swa.txt"
```

SAIF format is useful to back-annotate switching activity information in Synopsys tools,

to improve the accuracy of the power report and further optimize power consumption. VCD format can be instead imported in Cadence Innovus, for more accurate power analysis after the physical design.

The collected switching activity information have been back-annotated in Design Vision with another script, `/syn/post_syn_backannotation.tcl`, which performs a final compilation to try to further optimize power consumption.

```
# Read power back annotated file from ModelSim
read_saif -input ${simDir}/outputs/dlx_swa.saif -instance tb_dlx/dlx_uut/dlx
# Specify wire model (useful for physical design)
set_wire_load_model -name 5K_hvratio_1_4
# Compile
compile -map_effort high
```

4.3 Results

The results of the synthesis I have carried out are summed up in Tab.4.1. It can be noticed how the ungroup method is able to reach the desired clock period constraint of 1.25 ns (frequency of 800 MHz), while a normal synthesis, the one marked with *Constrained*, can only reach a slack of -0.19 ns. On the other hand, the imposition of such a low clock period constraint results in a dramatic increase of the dynamic power, which goes from the 2.39 mW of the unconstrained synthesis to about 31 mW, worsening of more than an order of magnitude.

Finally, the back-annotation of the switching activity computed with post-synthesis simulation seems to have helped improving the power consumption, reducing the dynamic power of about 1,3 mW, which accounts for the 4% of its total. Also, this final optimization greatly improved the combinational area, resulting in a total area reduction of about 17%.

For the purpose of a comparison with the physical design reports, it is useful to point out that the worst path in terms of max delay is the one going from `DLX_Datapath/PC_reg[1]` to `DLX_Datapath/PC_reg[29]`, which has a slack of 0 ns; the worst path in terms of min delay goes instead from `DLX_ControlUnit/cw3_reg[0]` to `DLX_ControlUnit/cw4_reg[0]` and has a slack of 0.03 ns.

Table 4.1: Results of the DLX-pro synthesis; the reported timing path is the worst one in terms of max delay.

SYNTHESIS	TIME				POWER		AREA	
	Startpoint	Endpoint	Arrival time	Slack	Dynamic	Leakage	Comb.	Non-comb.
Unconstrained	RF_state_reg[1]	stackBus_Out[31]	2.44 ns	N/A	2.39 mW	1.28 mW	30094.71	29637.72
Constrained	IR_EXMEM_reg[20]	ALUop2_reg[2]	1.41 ns	-0.19 (violated)	32.18 mW	1.46 mW	46114.56	28296.81
Constrained with ungroup	cw5_reg[3]	phy_regfile_reg[7][3]	1.22 ns	0.0 ns (met)	31.32 mW	1.71 mW	45859.20	30730.18
Back-annotated switching activity	PC_reg[1]	PC_reg[29]	1.22 ns	0.0 ns (met)	30.04 mW	1.32 mW	33081.09	30542.92

CHAPTER 5

Physical Design

The output of the last back-annotated post-synthesis netlist optimized for a clock frequency of 800 MHz, the file `/syn/netlists/DLX_backann_opt.v`, has been used as a start point for the physical design, carried out with the aid of Cadence Innovus.

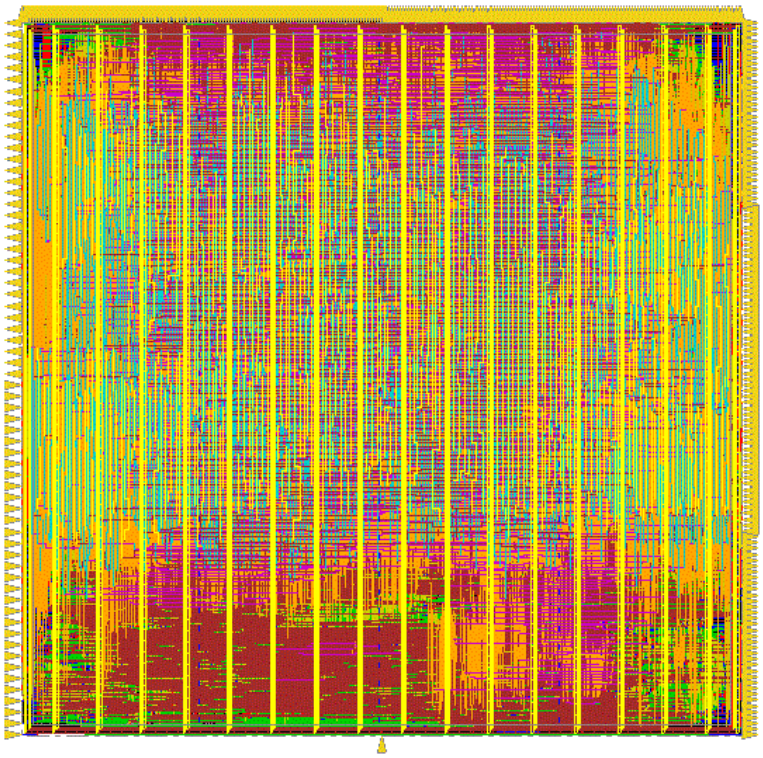

Figure 5.1: Physical design of the core after the post-routing optimization; I/O pins positions are also shown.

5.1 Physical design parameters

For the floorplan, a core utilization factor of 0.6 and an aspect ratio of 1 have been specified; also, a boundary of 5 µm has been created between the core and the die to insert `vdd` and `gnd`

power rings. *Wirebonding* is indeed supposed to be used for the final chip, which is also the reason why I employed the highest metal layers M9 and M10 for the power distribution. Two metal layers are used for power lines in order to avoid congestions; cells interconnections are instead placed from layer M1 up to M8.

As far as the I/O pins are concerned, I have used the bottom side for `Clk` and `Rst`; they are placed in the middle of the core side so that their distribution can be handled more homogeneously. Along the top side is placed the interface with the stack, on the right side the interface with the data memory and on the left the interface with the instruction memory.

5.2 Routing and optimization

To avoid introducing setup and hold time violations, a careful optimization has been performed. After the placement of standard cells and I/O pins, I ran a first **post clock-tree synthesis** (CTS) **optimization** by means of the command `optDesign`. Such command tries to solve setup time violations in the first place, then hold time ones. However, to correct setup violations new hold violations might be created, without the possibility to solve them during the following step. I avoided this problem correcting only hold time violations first, with

```
optDesign -postCTS -hold
```

and then running a full optimization with

```
optDesign -postCTS -setup -hold
```

The timing reports after the post CTS optimization did not signal any violation, over a total of 9985 paths.

After such optimization, I performed the **routing** to actually establish physical interconnections among the standard cells and then set

```
setAnalysisMode -analysisType onChipVariation
```

to be able to perform timing reports. Post-routing timing reports returned a total of 98 paths violating hold time constraint, and no violations of setup time; the slack histogram for both setup time and hold time is shown in Fig.5.2

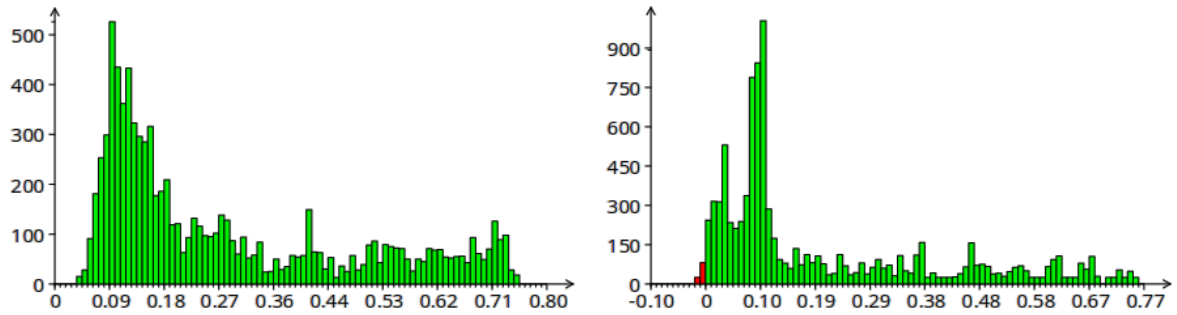

Figure 5.2: Slack histograms of post-route timing analysis; on the left, the histogram for the setup time, on the right, the one for the hold time highlighting 98 violations.

With the same principle explained above, I performed a **post-routing optimization** with the following commands

```
optDesign -postCTS -hold
optDesign -postCTS -setup -hold
```

This actually solved all the violations introduced by routing; the slack histograms representing post-routing optimization timing reports are shown in Fig.5.3.

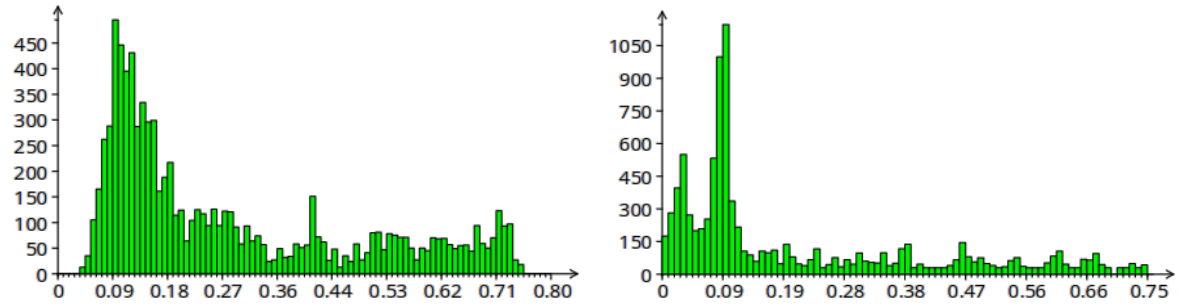

Figure 5.3: Slack histograms of post-route optimization timing analysis; on the left, the histogram for the setup time, on the right, the one for the hold time.

Differently from what reported for the post-synthesis back-annotated netlist, the max delay path goes from `DLX_Datapath/RegisterFile/phy_regfile_reg[122][31]/D` to `DLX_Datapath/RegisterFile/RF_state_reg[1]/Q` and has a slack of 0.039 ns, while the min delay path starts from `stackBus_In[100]` to end in `DLX_Datapath/RegisterFile/phy_regfile_reg[76][4]/D`, with a slack of 0 ns.

No other violations have been reported, including max fanout load, max capacitance, connectivity verification and geometry verification. A picture of the routed and optimized core is shown in Fig.5.1. The final area occupation was reported to be 62492.4 µm^2, or about 0.06 mm^2, with 36262 standard cells composed of a total of 78311 gates. The total area of the core is 101804.052 µm^2 (0.1 mm^2), with an effective utilization factor of the chip equal to 61.385%. A total of 447 I/O pins are present. The total wires length over all the 10 metal layers is 1213991.09 µm (1.21 m), with an average of 32.0636 µm per net.

As far as the power analysis is concerned, it was based on the VCD switching activity annotation file obtained from the post-synthesis simulation of the DLX; the obtained results substantially differed from the ones of the synthesis report, especially in terms of switching power which was estimated to be 566.2 µW after synthesis, but revealed to reach 20.3 mW after physical design analysis. The total reported power consumption is 61.2 mW.

APPENDIX A

Instruction Set Architecture

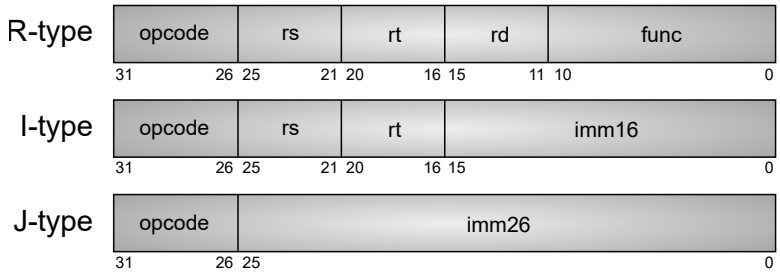

Figure A.1: Instruction register semantics basing on the instruction type.

Instruction set encoding

R-type instructions

MNEMONIC	FUNC	MNEMONIC	FUNC
sll	0x04	sltu	0x3A
srl	0x06	sgtu	0x3B
sra	0x07	sleu	0x3C
add	0x20	sgeu	0x3D
addu	0x21	mult	0x40
sub	0x22		
subu	0x23		
and	0x24		
or	0x25		
xor	0x26		
seq	0x28		
sne	0x29		
slt	0x2A		
sgt	0x2B		
sle	0x2C		
sge	0x2D		

J-type instructions

MNEMONIC	OPCODE
j	0x02
jal	0x03

I-type instructions

MNEMONIC	OPCODE	MNEMONIC	OPCODE
beqz	0x04	seqi	0x18
bnez	0x05	snei	0x19
addi	0x08	slti	0x1A
addui	0x09	sgti	0x1B
subi	0x0A	slei	0x1C
subui	0x0B	sgei	0x1D
andi	0x0C	lb	0x20
ori	0x0D	lw	0x23
xori	0x0E	lbu	0x24
lhi	0x0F	lhu	0x25
jr	0x12	sb	0x28
jalr	0x13	sw	0x2B
slli	0x14	sltui	0x3A
nop	0x15	sgtui	0x3B
srli	0x16	sleui	0x3C
srai	0x17	sgeui	0x3D

APPENDIX B

Assembler modifications

Many features of my custom DLX-pro are different from the original implementation of such architecture; in order to assemble asm code which could be simulated correctly, I have had to make some modifications to the original Perl script of the assembler. They are listed in the following.

- to introduce an **integer multiplication** instruction in the ISA, I have substituted the entry `"mult" => "f,0x0e"` in the instruction table `instTbl`, corresponding to the floating-point instruction with opcode `0x0e` and mnemonic `mult`, with `"mult" => "r,0x40"`, which represents an R-type instruction with `0x40` as func field; integer multiplication is discussed in Subsec.2.2.2;
- the original assembler supposes that the **Branch Target Address** computation is performed starting from the Next Program Counter value (which is PC+4), and thus 4 is subtracted from the immediate before encoding it in the Imm16 field when the instruction is a branch (`beqz`, `bnez`): `$dst -= $addrt + 4`; in my architecture, as described in Sec.2.3, the potential Target Address of a branch is computed in the IF stage directly with PC, so that the 16-bit immediate must be encoded as written in the asm code, without any subtraction: `$dst -= $addrt`;
- since also **Jump Target Address** computation is anticipated to IF stage, the same modification of the previous point applies also to J-type instructions (`j`, `jal`).

APPENDIX C

Project files organization

In the following, the main directories of the project and their organization is reported.

```
dlx_project
├── docs ... Documentation of the project, including the report attachments.
│   └── attachments
├── libs ... Libraries which might not be included in ModelSim by default, needed for post-
│            synthesis simulation.
│   ├── NangateOpenCellLibrary
│   └── vital2000
├── phy ... Physical design directory, containing the Innovus project files.
│   ├── outputs ... Outputs of the physical design: post-routing optimized netlist and delay
│   │               annotations file.
│   └── reports
├── script ... Scripts related to the project: assembler script and the script to create the
│              firmware file.
│   └── assembler.bin
├── sim ... Simulation directory, containing behavioural simulation and post-synthesis simu-
│           lation scripts, firmware and data memory files.
│   ├── outputs ... Outputs of the simulations: nets switching activity annotations.
│   └── testbenches ... Testbenches for simulation of whole DLX and single components.
├── src ... Directory for the VHDL surce file of the DLX, containing all its entities and the
│           wrappers for behavioural simulation, synthesis and post-synthesis simulation.
│   └── a.b-DataPath.core
│       ├── 000-Adder.core
│       └── d-ALU.core
│           ├── b-Logicals.core
│           ├── c-Comparator.core
│           └── d-Multiplier.core
├── syn ... Synthesis directory containing the scripts for initial synthesis and back-annotated
│           optimization.
│   ├── netlists ... Post-synthesis netlists and constraints files.
│   └── reports
└── test ... Tests in asm language for DLX simulation.
```