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: Allioux Coralie, Romero R. Gabriel, Valente Simone

June 20, 2022

Contents

CHAPTER 1

Project Overview

1.1 Basic-DLX

As a reminder, the requirements for the basic DLX have been developed as follows:

- 5-stage pipeline
- Instruction subset:
  - add/addi
  - and/andi
  - or/ori
  - sge/sgei
  - sle/slei
  - sll/slli
  - sne/snei
  - srl/srli
  - sub/subi
  - xor/xori
  - beqz
  - bnez
  - j
  - jal
  - lw
  - nop
  - sw
- Datapath related to pipeline and instruction subset
- Basic synthesis
- Basic physical design

1.2 Pro-DLX

Furthermore, some pro features have been added to the basic DLX:

- ALU: logic functions are based on the T2 processor
- Verification of the processor using a UVM testbench

1.3 File hierarchy

1.3.1 Main tree

The project tree is composed of three main folders: `DLX_vhd`, `DLX_vhd` and `UVM_testbench`. More details can be found about those folders in the next sections.

In addition, the folders `asm_example`, `assembler.bin` and the script `assembler.sh` have as purpose to convert an assembly file to a binary file, readable by our DLX.

Those sources are mainly taken from the original sources given at the beginning of the project.

```
/
├── DLX_vhd/ ..... DLX source files
│   ├── sim/ ..... Simulation scripts
│   ├── test_bench/
│   └── compile_DLX.bash ..... Compilation script for simulation
├── DLX_vhd_fully_synthesizable/ ..... DLX source files for synthesis purposes
│   ├── sim/
│   ├── test_bench/
│   ├── compile_DLX.bash
│   └── syn_DLX.scr ..... Synthesis script
├── UVM_testbench/ ..... Files for the UVM testbench
├── asm_example/
├── assembler.bin/
└── assembler.sh
```

1.3.2 DLX_vhd tree

Here the full tree inside the `DLX_vhd` folder, which contains the entire DLX with pro functionalities (control unit, datapath, T2 ALU).

```
/
└── DLX_vhd/
    ├── a.b-Datapath.core/ ..... VHDL source files of Datapath components
    │   ├── a.b.a-mux21.vhd
    │   ├── a.b.b-adder.vhd
    │   ├── a.b.c-register.vhd
    │   ├── a.b.d-ALU.vhd ..... T2-based ALU
    │   ├── a.b.d.a-NAND3.vhd
    │   ├── a.b.d.b-NAND4.vhd
    │   ├── a.b.d.c-Shifter.vhd
    │   ├── a.b.e-branching_unit.vhd
    │   ├── a.b.f-extender.vhd
    │   ├── a.b.g-mem.vhd
    │   └── a.b.h-registerfile.vhd
    └── sim/ ..... Simulation scripts
```

```
│   ├── simulate_DLX.do ..... Script to launch simulation on ModelSim
│   └── wave_DLX.do ..... Default configuration of waves for simulation
├── test_bench/
│   ├── ALU_tb/ ..... Standalone ALU testbench.
│   └── TB_DLX.vhd
├── 000-Log2.vhd
├── 001-globals.vhd
├── a-DLX.vhd ..... DLX top entity
├── a.a-CU_HW.vhd ..... Hardwired Control Unit
├── a.b-Datapath.vhd
├── a.c-IRAM
└── compile_DLX.bash ..... Compilation script for simulation
```

Note that the naming of VHDL source files for the DLX is following the component hierarchy: `a-DLX.vhd` being the top entity.

It also contains a special folder for the Datapath components, called `a.b-Datapath.core`.

For basic test and simulation, some testbenches and scripts are included. The testbenches are obviously located within the folder `test_bench`. On the other hand, some simulation scripts for ModelSim are located in `sim` folder, in addition of the `compile_DLX.bash` script. The procedure to be followed to use those scripts is available in section 1.4.

1.3.3 DLX_vhd_fully_synthesizable tree

The `DLX_vhd_fully_synthesizable` folder is quite similar to the previous one, described in section 1.3.2. The version of the VHDL sources has been modified in order to make the synthesis possible: the memories are now out of the DLX.

Moreover, a script for synthesizing this version of the DLX can be found. To know how to use, refer to section 1.5. Regarding the `reports` folder, it keeps the results obtained after synthesizing.

```
/
└── DLX_vhd_fully_synthesizable/
    ├── a.b-Datapath.core/
    │   └── <datapath_components>.vhd ..... Same as DLX_vhd
    ├── reports/ ..... Results of synthesis
    ├── sim/
    │   └── <simulation_scripts>.do ..... Same as DLX_vhd
    ├── test_bench/
    │   └── <testbenches>.vhd ..... Same as DLX_vhd
    ├── .synopsys_dc.setup
    ├── <DLX_components>.vhd ..... Same as DLX_vhd
    ├── compile_DLX.bash
    └── syn_DLX.scr ..... Synthesis script for Synopsys
```

1.3.4 UVM_testbench tree

The UVM testbench folder contains all the files related to the test that have been used for assessing the performance of the DLX. The folder `tb` contains all the SystemVerilog files of the test, each file represents a different class declaration that are used to create the objects of the testbench.

```
/
└── UVM_testbench/
    └── tb/
```

```
    ├── command_monitor.svh
    ├── coverage.svh
    ├── dlx_if.sv
    ├── dlx_macros.svh
    ├── dlx_pkg.sv
    ├── dlx_wrap.sv
    ├── driver.svh
    ├── env.svh
    ├── normal_sequence.svh
    ├── normal_test.svh
    ├── output_monitor.svh
    ├── output_transaction.svh
    ├── scoreboard.svh
    ├── sequence_item.svh
    └── top.sv
├── compile.f
├── compile_synth.f
├── compile_vhdl.f
└── sim.do ..... Script for launching the UVM simulation
```

For launching the UVM test it is necessary to run the script `sim.do` using `vsim`:

```
vsim -c -do sim.do
```

Notice that this script is in charge of compiling both the DLX processor files (using `vcom`) and the testbench files (using `vlog`). Besides, setting the `UVM_VEBOSITY` to the value `UVM_LOW` means that only the final count of errors is displayed when the test is finished, if the number of errors found is zero, then the DLX has successfully overcome the UVM test.

A final remark regarding the test has to be added: When using a new file as the input of the instruction memory of the DLX, besides changing the name of the file in `a.c-IRAM.vhd`, it is necessary to change the name of the file also in the `scoreboard.svh` file, since the instructions file is also used by the UVM testbench.

1.4 Simulation with Modelsim

1.4.1 Full DLX simulation

As shown in subsections 1.3.2 and 1.3.3, a dedicated folder called `sim/` contains some scripts to be used while simulating the DLX on *ModelSim*.

By assuming that the commands `vsim` and `vlib` are available, the following commands allow to execute the simulation by using our scripts.

Go to the source folder `DLX_vhd` and create the library:

```
cd DLX_vhd/
vlib work
```

Compile all DLX sources by executing the `compile_DLX.bash` script.

```
./compile_DLX.bash
```

Launch ModelSim by using the provided file `sim/simulate_DLX.do`. This will open some waveforms for the `test_bench/TB_DLX.vhd` testbench.

```
vsim -do sim/simulate_DLX.do
```

1.4.2 ALU simulation

As highlighted in the tree of subsection 1.3.2, an specific test has been developed for covering the ALU standalone. This test corresponds to a SystemVerilog test and all the required files are inside the folder `ALU_tb`. The test is a randomized test and performs 1000 executions for random inputs and operation types.

The SystemVerilog test can be launched using the following command in the `ALU_tb` folder:

```
vsim -c -do sim.do
```

Here, the scoreboard is in charge of checking if the actual output of the ALU corresponds to the expected output that has been created by the internal ALU model. Then, in this case, we are taking advantage of the SystemVerilog objects in order to create a testbench that corroborates that the ALU is providing the correct output.

1.5 Synthesis with Synopsys

As mentioned in subsection 1.3.3, the script `syn_DLX.scr` automates the synthesis of the DLX without, with external memories.

By using the `design_vision` command and giving the synthesis script, some reports will be created. More details about it will be found in section 3.1.

```
design_vision -no_gui -f syn_DLX.scr
```

CHAPTER 2

DLX Functional Schema

2.1 Control unit

The implemented control unit, shown on Figure 2.1, is a **hardwired** one, composed of 18 signals, organized in 5 stages. They are listed in Table 2.1, followed by a short description. Most of them are the same than the one suggested in the DLX guide. For more information about how and by which components they are used, see section 2.2.

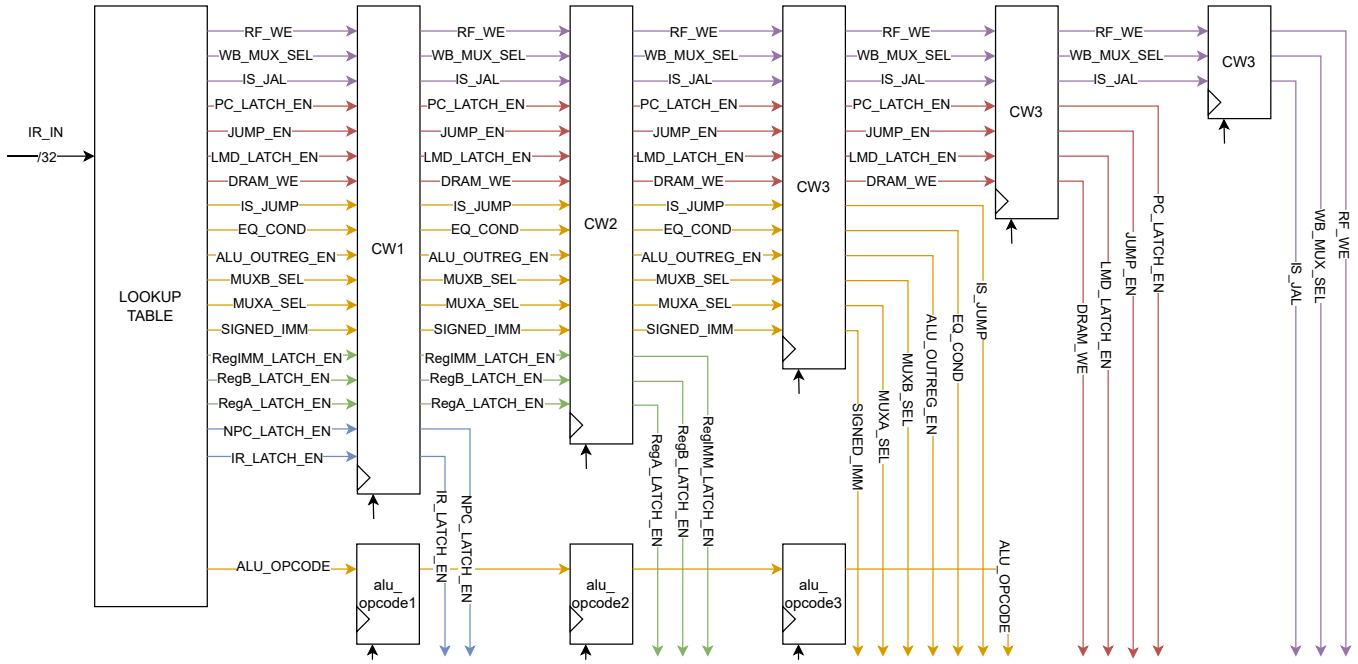


Figure 2.1: Hardwired Control Unit

The definition of the number of control signals is fully modular, using constants defined in `001-globals.vhd` file. With this method, it is easier to modify the control unit by adding or removing a control signal, which has been useful while implementing the DLX by functionalities.

The list of those constants are sum-up in Table 2.2.

Thank to this method, adding or removing a control signal is done by very few steps:

Stage	Control signals	Description
IF	IR_LATCH_EN	Enable instruction register
	NPC_LATCH_EN	Enable new program counter register
ID	RegA_LATCH_EN	Enable read register A in register file
	RegB_LATCH_EN	Enable read register B in register file
	RegIMM_LATCH_EN	Enable immediate register
Exec	SIGNED_IMM	Immediate is signed
	MUXA_SEL	Mux selector for ALU input A
	MUXB_SEL	Mux selector for ALU input B
	ALU_OUTREG_EN	Enable ALU output register
	EQ_COND	1 for instruction BEQZ, 0 for BNEZ
	IS_JUMP	Current instruction is of J-type
	ALU_OPCODE	Operation to be executed by ALU
Mem	DRAM_WE	Write enable in DRAM
	LMD_LATCH_EN	Read enable in DRAM
	JUMP_EN	Current instruction is of a J-Type or a conditionnal branch
	PC_LATCH_EN	Enable program counter register
WB	IS_JAL	Mux selector for write back the PC value, in case of JAL instruction
	WB_MUX_SEL	Mux selector for write back data, between DRAM and ALU output
	RF_WE	Write enable for register file

Table 2.1: Control signals list with description

- Update the constant *CW_SIZE* in `001-globals.vhd`
- Update the constant *NB_SIG_S**X***, replacing ***X*** by the corresponding stage number. Note that this step is needed only for the first four stages, it is not required for *WB* stage.
- Update the content of the signal *cw_mem* in `a-a.CU_HW.vhd`, in particular the number of columns.
- Update the association between the target control signal and the corresponding bit in control word. This step could require more modification, like updating the index of the following control signals of the same stage.

Constant name	Description
CW_SIZE	Number of control signals, i.e. size of control word in bit
NB_SIG_S1	Number of control signals for stage 1, i.e. IF
NB_SIG_S2	Number of control signals for stage 2, i.e. ID
NB_SIG_S3	Number of control signals for stage 3, i.e. EXEC
NB_SIG_S4	Number of control signals for stage 4, i.e. MEM
CW1_SIZE	Number of bit for the control word associated to stage 1, i.e. IF
CW2_SIZE	Number of bit for the control word associated to stage 2, i.e. ID
CW3_SIZE	Number of bit for the control word associated to stage 3, i.e. EXEC
CW4_SIZE	Number of bit for the control word associated to stage 4, i.e. MEM
CW5_SIZE	Number of bit for the control word associated to stage 5, i.e. WB

Table 2.2: Constants used by the hardwired control unit, to manage the control words size

2.2 Datapath overview

The full schematic of the DLX Datapath is depicted on Figure 2.2. As mentioned earlier, the 5 stages are clearly visible with the associated control signals: IF, ID, EXEC, MEM and WB.

The next sections aim at providing more details about components and choices we have done for implementing our DLX.

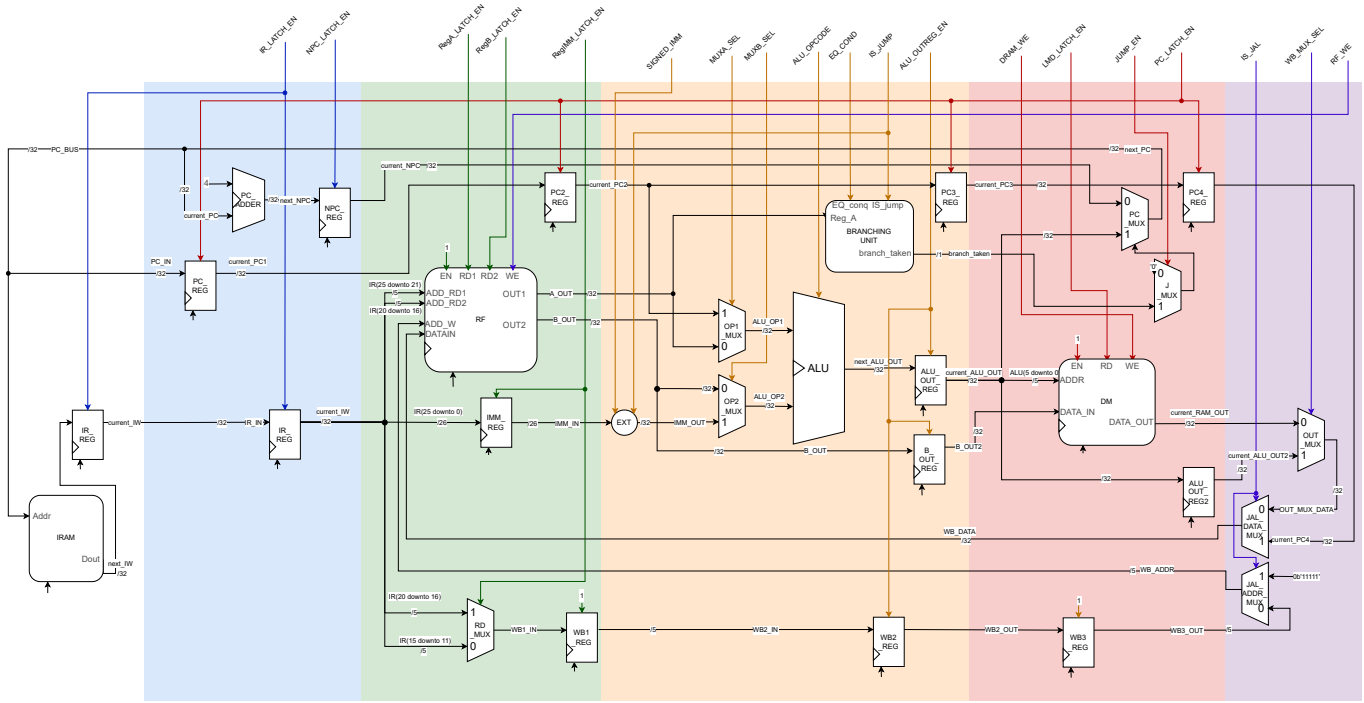


Figure 2.2: DLX Datapath schematic

2.3 Personalized components in Datapath

This section presents personalized components in order to match the specifications of the DLX.

2.3.1 ALU

The ALU, depicted on Figure 2.3, is composed of five parts:

- An **adder/subtractor**
- The **T2 logicals**: using NAND gates as discussed in the Microelectronics Systems course
- A **comparator**
- The **T2 shifter**: barrel shifter discussed in the Microelectronics Systems course
- A **multiplexer** to select the output using *FUNC*, which is actually described using a *case-when* construct

The adder, substractor and comparator are nothing special: they are described with the native operation of VHDL, i.e. $+, -, >=, <=, /=$.

On the contrary, the shift and logical instructions are implemented taking example from the T2.

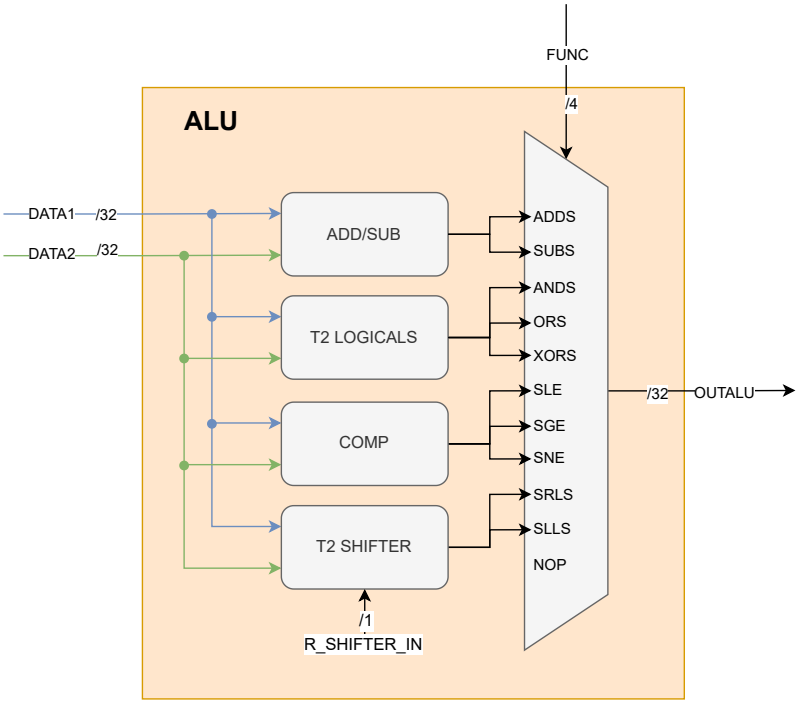


Figure 2.3: ALU Overview

T2 Shifter

The implementation of the T2 shifter is depicted on Figure 2.4. It has three inputs and one output:

- *DATA1* which is the data to be shifted;
- *DATA2* tells how much *DATA1* must be shift;
- *R_SHIFTER_IN* configures the shifter as right or left;
- *OUTPUT* beeing the result after applying the shift operation.

The shifter is also divided in two mirroring parts:

- Right shift **SRL**: $CONF = 0$, in Orange on Figure 2.4
- Left shift **SLL**: $CONF = 1$, in Violet on Figure 2.4

Note that the input signal *CONF*, is called *R_SHIFTER_IN* in the ALU component. This signal is defined thanks to the *FUNC* signal: if the current operation is **SRL**, then *R_SHIFTER_IN* is set to 1, otherwise 0.

The shifter uses some 40-bit *MASKS*, composed of a concatenation of *DATA1* and zeros, as shown on the top of Figure 2.4.

DATA2 is used as a selector in two steps: to select the **MASK** and the *COARSE* value. Note that only the 5 LSBs of *DATA2* are actually exploited.

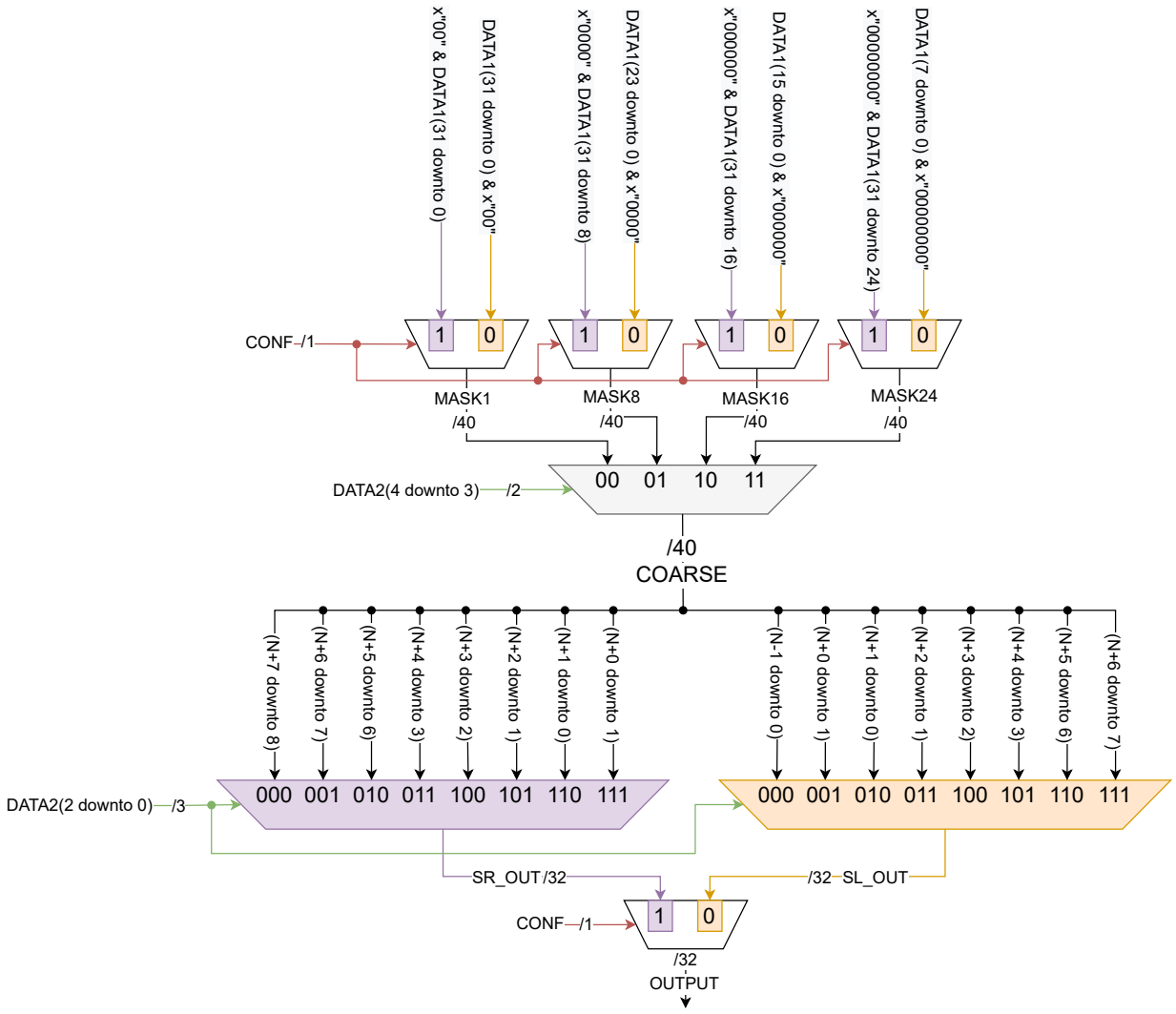


Figure 2.4: T2 Shifter implementation

T2 Logicals

The T2 Logicals, which its implementation is represented on Figure 2.5, needs three inputs and one output:

- *DATA1*, which is the first operand;
- *DATA2*, which is the second operand;
- *S*, which selects the logical operation to be applied;
- *Y_LOGIC*, which is the result of the operation *S* between *DATA1* and *DATA2*.

It is mainly composed of **NAND** gates: 32 **NAND4** (32-bit DATA) and 4×32 **NAND3**.

This implementation defines six logic operations: **AND**, **OR**, **XOR**, **NAND**, **NOR**, **XNOR**. As the requested instruction does not include the last three instruction, this part of the T2 Logicals is not exploited by our DLX. However, including them is straightforward: those instructions could be included as a **alu_type** in `001-globals.vhd` and in the process **P_LOGIC** of `a.b.d-ALU.vhd`.

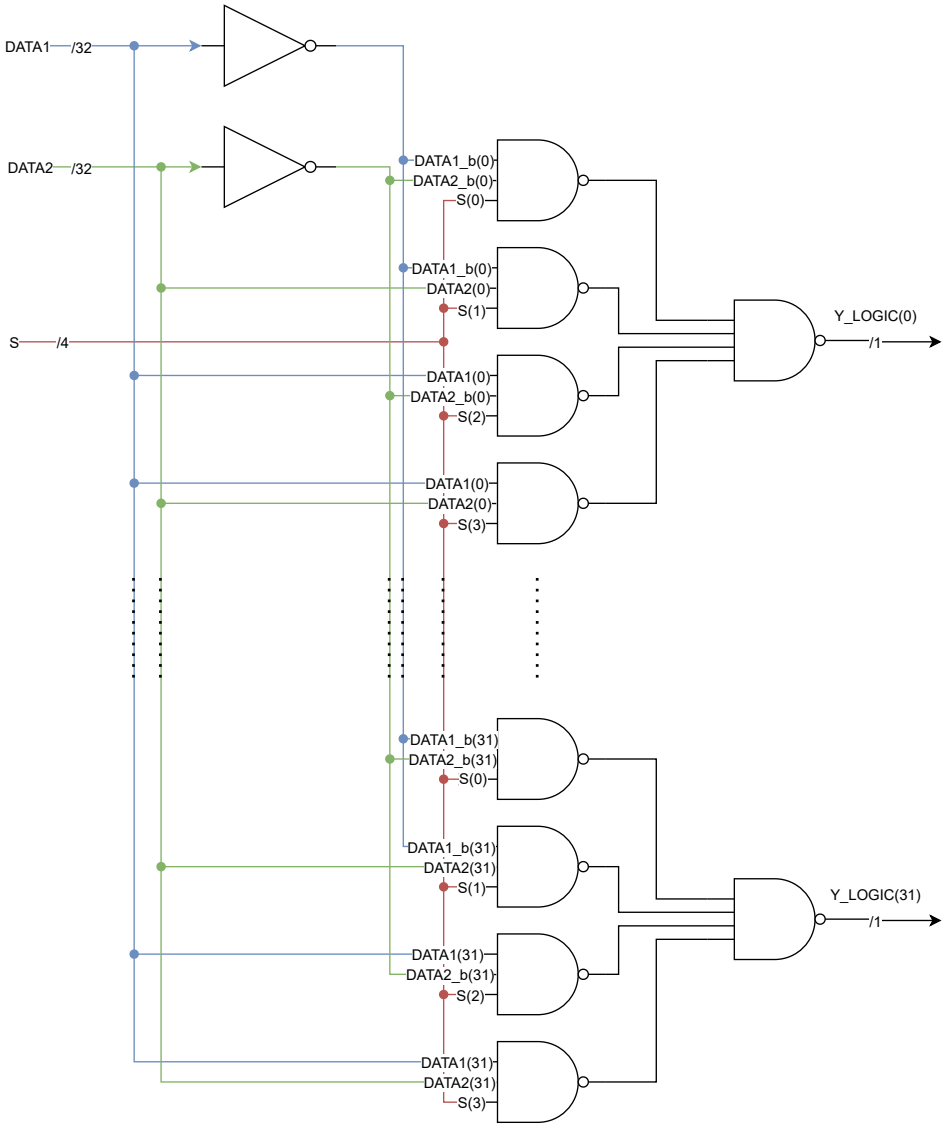


Figure 2.5: T2 Logicals implementation

The Table 2.3 recaps the possible value that the selection signal S can take.

Operation	*S3*	*S2*	*S1*	*S0*
AND	0	0	0	0
OR	0	0	1	1
XOR	0	1	0	1

Table 2.3: T2 Logicals: Selection signal S values

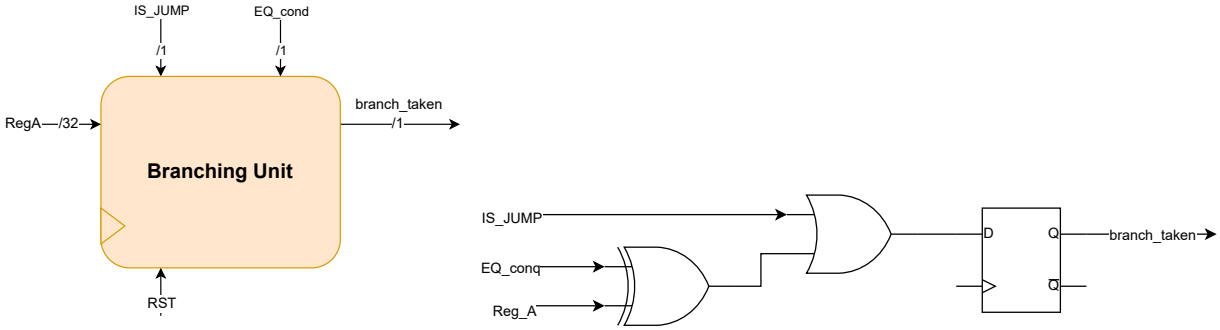
2.3.2 Register file

The exploited register file is actually the one implemented during the exercise 1 of the third lab, done in class.

2.3.3 Branching Unit

The Branching Unit is part of the stage *EXEC* of the pipeline. This component is depicted on Figure 2.6. Its role is to determine if a branch must be taken or not, i.e. the value of PC must be updated with a new value or not.

This component is then dedicated for the instructions **beqz**, **bnez**, **j** and **jal**.



(a) Branching unit block

(b) Branching unit implementation

Figure 2.6: Branching Unit Overview

The Branching Unit is composed of three inputs and one output as shown in Figure 2.6a: the two control signals *IS_JUMP* and *EQ_cond*; the value of *Reg_A* from the register file; while the output corresponds to *branch_taken*.

This component respects the truth table in Table 2.4. This truth table can be implemented with two gates, as shown on Figure 2.6b. Indeed, in case of a **J-Type** instruction, i.e. **j** or **jal**, the branch is automatically taken. On the other hand, for the conditional branch like **beqz** and **bnez**, a comparison to 0 must be done on the value of *Reg_A* in order to determine if the branch is taken or not. For this, the *EQ_cond* control signal specifies which type of comparison should be applied:

- $EQ\_conq = 0$: **BNEZ** instruction
- $EQ\_conq = 1$: **BEQZ** instruction

IS_JUMP	*EQ_conq*	*Reg_A*	**branch_taken**
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	–	–	1

Table 2.4: Branching Unit Truth Table

In addition, a flip-flop is needed to keep the result in the current stage for one clock cycle and by a consequence, respect the pipeline flow.

2.3.4 Extender

The Extender component, depicted on Figure 2.7, is used to extend to 32-bit the immediate of a **I-type** or **J-Type** instruction. Depending of the type instruction, an immediate can be:

- Signed or Unsigned
- 16-bit long or 26-bit long

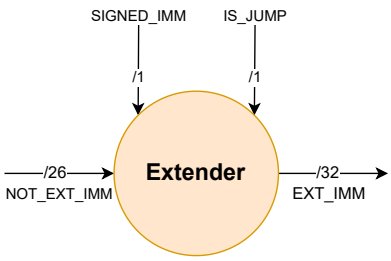


Figure 2.7: Extender Overview

The Table 2.5 recaps the behavior of the extender, controlled by the values of input control signals *SIGNED_IMM* and *IS_JUMP*.

Signal values by immediate charactheristic			**Operation to be applied**	**Associated instructions**
SIGNED_IMM	0	Unsigned	Zeros must be added on MSB until 32-bit are reached	ANDi, ORi, XORi, SLLi, SRLi
	1	Signed	The MSB of the immediate must be replicated until 32-bit are reached	J-TYPE, BNEZ, BEQZ, ADDI, SUBi, SNEi, SLEi, SGEi, LW, SW
IS_JUMP	0	16-bit long	Only the 16-bit LSBs of the input NOT_EXT_IMM must be taken into account, i.e. the 16th bit is the actual MSB	I-TYPE
	1	26-bit long	All bit from input NOT_EXT_IMM must be considered, i.e. MSB is the 26th bit	J-TYPE

Table 2.5: Extender behavior by instruction characteristics

Note that by default, the control signal *SIGNED_IMM* is set to 1 when the current instruction is a **R-type** one. Even if obviously, this signal does not give any concrete information because **R-Type** does not possess any immediate.

2.3.5 External IRAM

The top entity of the DLX, visible on Figure 2.8, is done with four components:

- Datapath, see section 2.2
- Control Unit, see section 2.1
- Instruction register
- IRAM

The control signals driven by the CU must correspond to the current instruction word given as input to DP. However, being an hardwired CU, to have the correct value in *CW1* (i.e. control word associated to the current instruction word) takes one clock cycle. This means that the CU must process the instruction in advance: one clock cycle earlier than the DP. In other words, the **IR** in stage IF should be updated by the **IR_LATCH_EN** control signal at the value of the current instruction word.

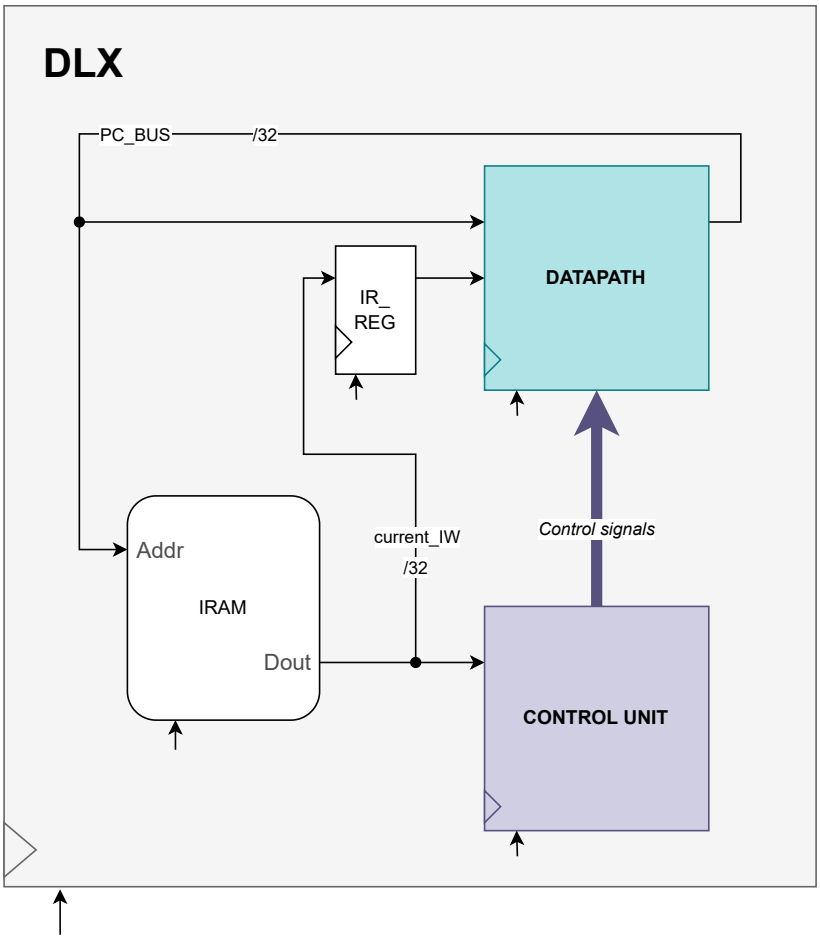


Figure 2.8: DLX Overview: IRAM and PC register external from Datapath

For this reason, the IRAM and an additional instruction register are outside to control better how to drive the instruction word to the datapath and the control unit.

Note that to test bigger assembly programs, the size of the IRAM must be updated and so the associated global *IRAM_SIZE* in `001-globals.vhd`. This value will propagate in the entire DLX entity, where it is needed.

2.3.6 JAL multiplexers

Some dedicated components within the Datapath are dedicated to the **JAL** instruction to support it. This section explains the reasons why some modifications were requested for **JAL** and how it has been thought.

The next address (the one after the **JAL** instruction) needs to be store in **R31** before applying the jump. The address of the target register is fixed and cannot be deduced from the instruction word. In addition, the ALU is already used to perform the jump, i.e. the addition between the current value of *PC_REG* and the *immediate* given by the instruction label. Therefore, there were still no way to propagate the *PC_REG* value until the **register file** *DATAIN*.
That is why some modifications within the Datapath for this particular instruction are required.

In order to support the **JAL** instruction, two multiplexers, and one control signal to control them, have been added in the stage WB: the control signal *IS_JAL*; the muxes *JAL_DATA_MUX* and *JAL_ADDR_MUX*. In addition, three

registers are included to propagate the value of **PC** in stages ID, EXEC and MEM.

Figure 2.9 recaps the connection around those two dedicated muxes.

As said before, the address just after the **JAL** must be stored in **R31**. It means that the *ADD_W* of the **register file** must be equal to $0b11111$, which is fixed, and *DATAIN* to $PC + 4$.

JAL_ADDR_MUX is used to select the address of the **R31** for the write address in case of **JAL** instruction, instead of the usual *WB3_REG* value.

Moreover, *JAL_DATA_MUX* is used to select the value of *PC_REG* as the data to be written in **register file**, instead of the value coming from **ALU** or **DRAM**, driven by *OUT_MUX_DATA*.

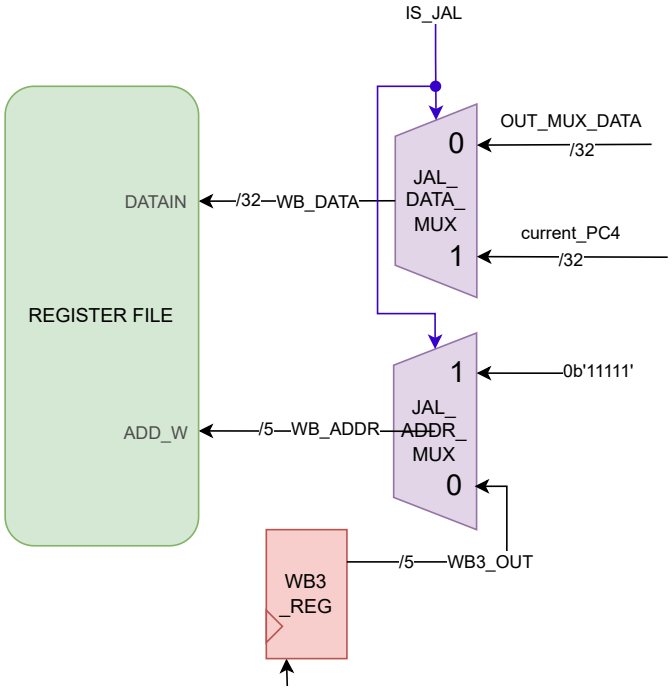


Figure 2.9: JAL multiplexers Context Overview

2.4 DLX fully synthesizable

In order to synthesize the DLX, the initial version has been changed to have external memories as recommended. This synthesizable version is depicted on Figure 2.10.

The IRAM and DRAM signals are then driven outside of the DLX:

- IRAM signals: *I_ADDR* (instruction address) and *I_DATA* (instruction word)
- DRAM signals: *D_WR* (write enable), *D_RR* (read enable), *D_DATA_IN* (data to be written), *D_ADDR* (address to be read or written) and *D_DATA_OUT* (read value from DRAM)

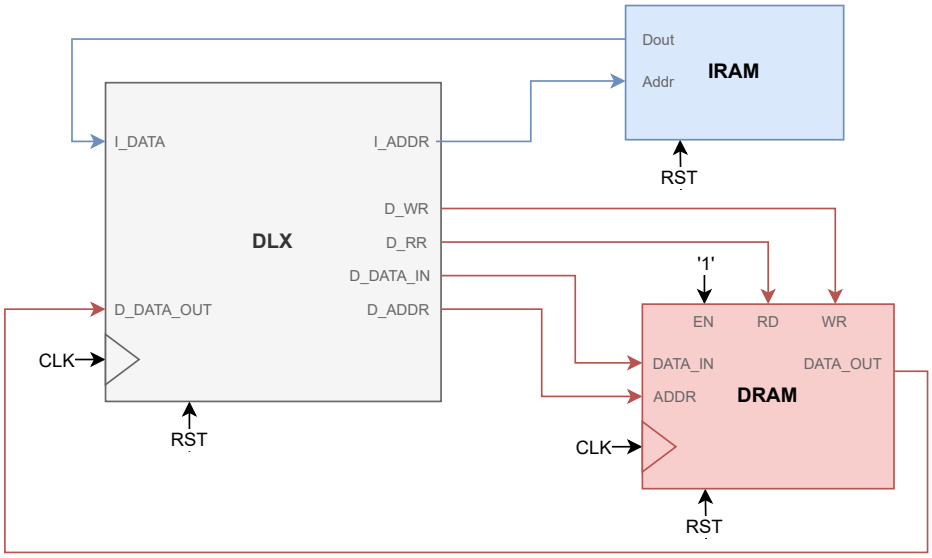


Figure 2.10: DLX synthesizable interfaced with external memories

CHAPTER 3

Implementation

3.1 Synthesis

As mentioned in Section 1.5 a single script is used to perform the synthesis of the DLX. The name of the script file is `syn_DLX.scr` and it can be found in the directory `/DLX_vhd_fully_synthesizable`.
At the beginning of this script all the files describing the synthesizable DLX are analyzed. Then with the line:

```
set min_clk_period 0.01
```

a variable is set to the value of 0.01 to represent a very low clock period that will be an "unachievable constraint" for Design Vision. By doing so the synthesis tool will find the lowest possible clock period for our design, trying to reach that "unachievable constraint".
The clock for the synthesis is then set in the following two lines:

```
set clk_period $min_clk_period
create_clock -name MY_CLK -period $clk_period CLK
```

and the synthesis is launched:

```
compile -map_effort high
```

In the reports of this synthesis the slack would be violated, even though the synthesis would be successful. Then, in order to perform a good analysis of the result, the clock is set to the lowest value found before, thanks to the following four lines:

```
set worst_path [get_timing_path]
set slack [get_attribute $worst_path slack]
set sorted_sl [lsort -real -increasing $slack]
set worst_slack [lindex $sorted_sl 0]
```

After that the synthesis is started again using the `-incremental mapping` option in order to re-use the results obtained in the previous synthesis, which are still valid even if the slack was negative. Now the timing report will show a null value for the slack.
The reports of the synthesized design are produced in the following three lines and saved in the `/reports` folder:

```
report_timing > reports/timing.txt
report_area > reports/area.txt
report_resources > reports/resources.txt
```

As last, the post-synthesis netlist of the DLX is saved.

3.1.1 Results

The files containing the data presented in the following are saved in the folder `/reports`.

Timing The clock period is found to be $0.98ns$. The critical path time is $0.94ns$ to which the setup time from the library, $0.04ns$, must be added. The sum turns out to be equal to the clock period, meaning the slack is 0 and performance is maximized.

Area A qualitative evaluation of the area occupation can be done looking at the quantity of cells used by Design Vision to synthesize the DLX. In Table 3.1 are listed all the different elements composing the DLX and their quantity. The reported cells are from the *Nangate Open Cell Library*, in which also their area occupation is listed. Design Vision can then compute the total area the DLX occupies, as reported in Table 3.2.

Number of ports	138
Number of nets	213
Number of cells	26
Number of combinational cells	23
Number of sequential cells	0
Number of macros	0
Number of buf/inv	23
Number of references	4

Table 3.1: Synthesized DLX composition

Combinational area	8166.2
Noncombinational area	6926.6
Total cell area	15092.8

Table 3.2: Synthesized DLX area occupation

3.1.2 Results with naive description of ALU

An effort was spent designing an ALU using explicitly described gates and shifters, instead of simple keywords provided by the VHDL language. An improvement in the performance is expected doing so. In Table 3.3 are presented the differences between the two choices. The synthesis with the naive implementation of the ALU can be launched with the same script, commenting the undesired line between the following two:

```
#analyze -library WORK -format VHDL {a.b-DataPath.core/a.b.d-ALU.vhd}
analyze -library WORK -format VHDL {a.b-DataPath.core/a.b.d-ALU_naive.vhd}
```

	Custom implementation	Naive implementation
Clock period [ns]	0.94	1.00
Combinational area	8166.2	7954.2
Noncombinational area	6926.6	6926.6
Total cell area	15092.8	14880.8

Table 3.3: Comparison with synthesis of naive ALU

It can be noticed the area occupation of the naive implementation is lower, while the timing is better in our description. This is a good situation that allows to find the best tradeoff between size and performance.

3.2 Physical design

After performing the synthesis of the DLX processor, this design is then going to be converted into a physical circuit. This section focuses on showing the results of the place and route of the circuit. The tool that allows to generate such physical structure corresponds to Innovus by Cadence.

The procedure for obtaining the circuit is very similar to the steps that have been followed during the laboratories of the Microelectronic Systems course. Some of the settings that have been chosen for this specific layout are described in the following lists. The order in which the lists are presented also correspond to the order that has been followed for developing the layout.

Floorplan

- Core aspect ratio = 1.0
- Core area utilization = 0.6
- Core margin towards die boundary: 5 µm

Power rings

- The rings are associated to VDD and GND nodes.
- Metal 9 is used for horizontal lines of the ring.
- Metal 10 is used for vertical lines of the ring.
- The rings width is chosen to be 0.8 µm

Vertical stripes

- The stripes are connected to both VDD and GND.
- Vertical stripes are included and use Metal 10 (as in the vertical stripes of the ring).
- The stripes width is chosen to be 0.8 µm.
- The distance between the stripes is 20 µm.

Horizontal power rails

- VDD and GND rails are also placed in the horizontal direction, according to the size of the standard cells.
- These metal conductors correspond to the Metal 1 layer.

Placement

- During this step, the metal layers 1 to 8 are blocked through the `placement blockage` parameter.

Fillers

- Fillers have been placed in the empty spaces of the layout.
- All the possible filler cells have been used in this design, i.e. from `FILLCELL_X1` to `FILLCELL_X32`

Routing

- The routing was performed using the NanoRoute routing tool.
- No specific settings were used along with NanoRoute, all the values were the default ones.

Besides, two different types of optimization of the circuit are performed during the process:

- **Post Clock-Tree-Synthesis (CTS) optimization**, which is performed before the placement of the fillers and it is expected to optimize the design considering both the setup and hold time of the registers.
- **Post routing optimization**, which is performed after doing the NanoRoute routing. The purpose of this step is to optimize the design in order to be compliant with the timing constraints that have been set by `.sdc` file that has been obtained from the synthesis.

After performing all these steps, the final version of the layout of the circuit is obtained. Figure 3.1 shows the physical view of the DLX, the pins have also been placed around the layout, most of them correspond to the buses that are used for communicating with the memories. On this figure it is also possible to notice the rings that are located around the processor and also the vertical stripes, since these are located in the upper layers.

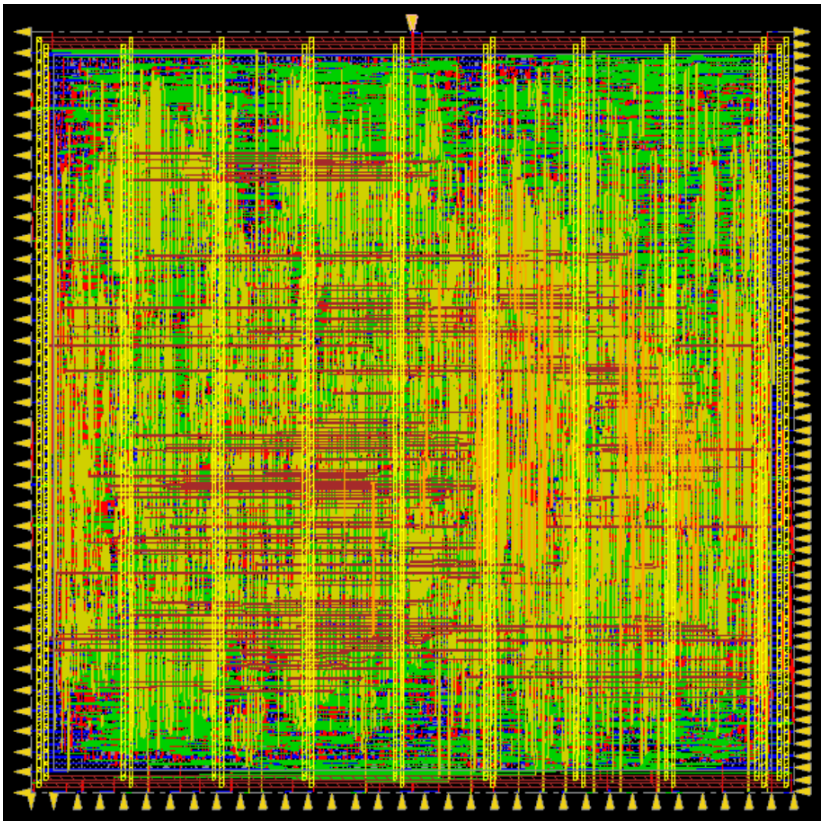
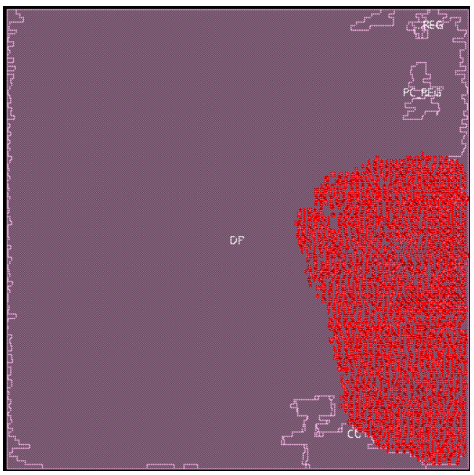
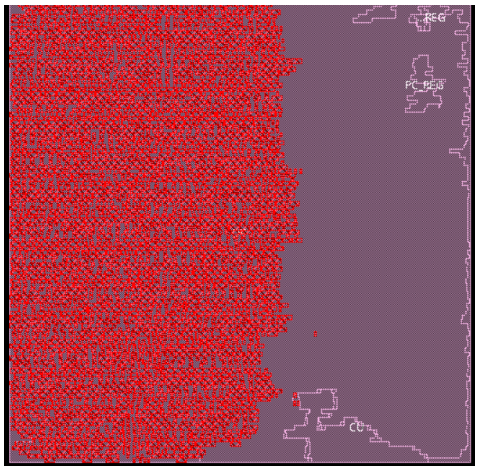


Figure 3.1: Physical view of the final design of the DLX processor

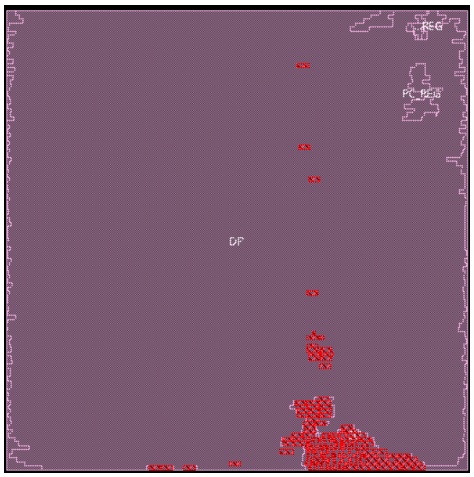
Besides the physical view of the design, the tool also allows to distinguish the regions of the circuit that correspond to the different components. This is performed through the amoeba view. For instance, Figure 3.2a shows the portion of the whole layout that corresponds to the ALU, Figure 3.2b shows that almost half of the total area of the circuit corresponds to the Register File. Finally, Figure 3.2c shows that the control unit uses only a small portion of the whole circuit area. The remaining area corresponds to other components that are within the datapath, and these figures highlight some of the relevant components.



(a) ALU highlighted in red



(b) Register file highlighted in red



(c) Control unit highlighted in red

Figure 3.2: Largest components of the DLX processor

After finishing the layout, it is necessary to perform some analysis that allow to confirm that the layout is compliant with the **design rules** and also with the connectivity of the circuit. For the first check we verify the geometry of the circuit; whereas for the second one, the **connectivity verification** is performed, this second verification checks that there are no floating wires or wrong connections. As a result, both of these validations showed **no errors** in the log, thus, we can conclude that the circuit is compliant with the layout design rules and also with the connectivity among the blocks.

Another action to be performed is the extraction of the **parasitic resistance and capacitance** of the metals that are in the layout. This process allows to export a `.spf` (Standard Parasitics Format) file that contains the model of the DLX processor. This file is subdivided into two sections: the first one corresponds to the part that models the parasitics that exist among the nodes (i.e. resistors and capacitors); the second part is the instance section and this one uses the subcircuits that contain the models of the instances that are used by the DLX processor, even the fillers are included in this file.

Finally, we also analyze the **timing performance** of the final post-layout circuit, for this purpose the tool allows to create reports both for the analysis of the setup and hold time violation of the registers. The setup time report

shows that the slack is still positive but with a very small margin (only 0.023 ns). Figure 3.3 shows the physical route that follows the critical path of this circuit. Besides, the histogram on the right side of the figure shows the distribution of the slack of the different paths of this circuit. For the hold time, the slack is slightly more robust (0.063 ns). All the reports (both post-CTS and post-route) are saved inside the `timingReports` folder.

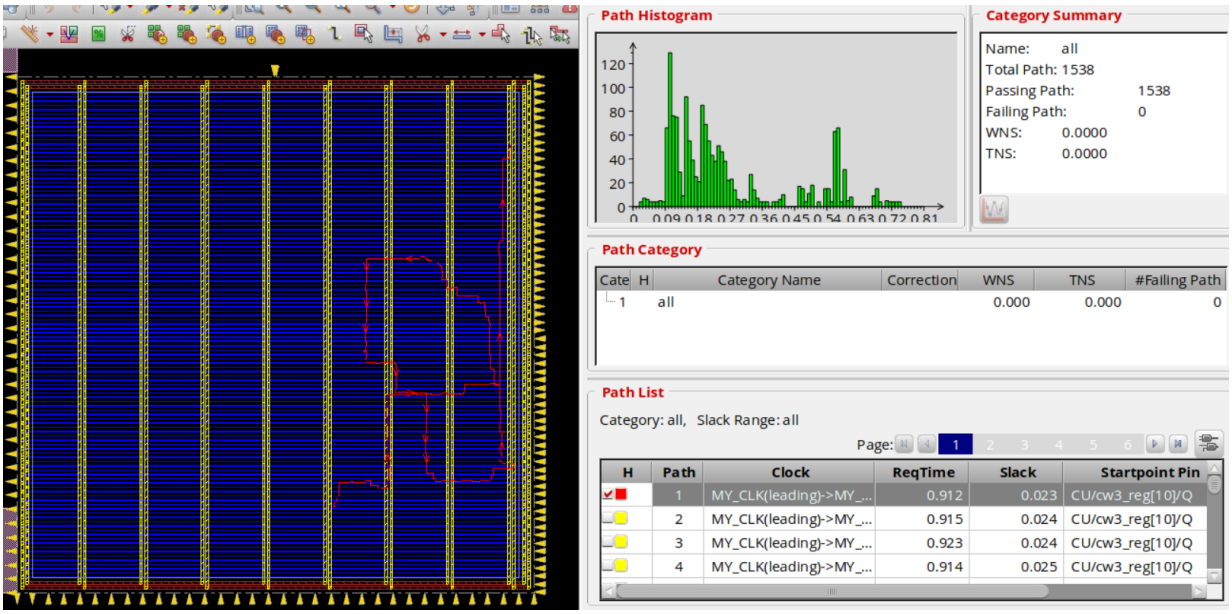


Figure 3.3: Critical path of the final design and histogram of path delays

CHAPTER 4

UVM

This chapter summarizes the main components of the UVM testbench that has been built for testing the behaviour of the DLX processor. Figure 4.1 represents the connections that exist between the components that comprise the testbench, each of these elements correspond to an object of the different classes that have been defined in the files mentioned in subsection 1.3.4.

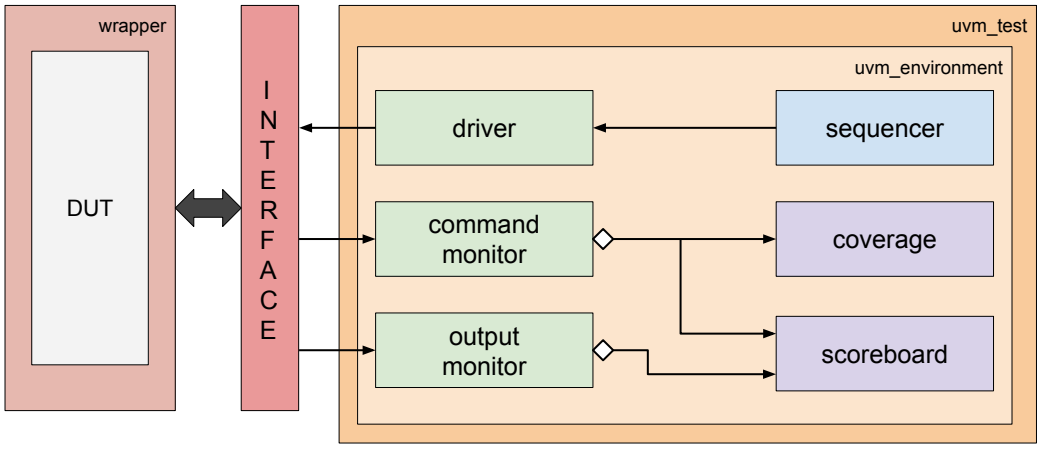


Figure 4.1: Structure of the UVM testbench.

The objective of this test is to check that the output of the different instructions correspond to the expected output. Consider that the different instructions have different types of outputs that allow to check if the instruction has been correctly executed. Therefore, it is necessary to sample different signals in the datapath of the DLX datapath (see Figure 2.2). Then, the distribution of the instructions and the relevant signals is organized as follows:

- *SW* instruction: in this case we observe the data memory in the expected address.
- *J-type*, *BEQZ* and *BNEZ* instructions: the value of the program counter is checked.
- *R-type* and other *I-type* instructions: we observe the register file in the expected destination address.

Besides the check of the expected output of the instructions, it is also relevant to check the correctness of the order in which these instructions are executed. For this reason, a check on the flow of the instructions has been added. The flow checker samples the value of the instruction register and then compares it with the expected value of the

instruction register. The expected value is calculated in the scoreboard using the assembly file that has been given as an input to the DLX processor.

The following sections describe the details of the different elements of the UVM testbench and also describe the design choices and characteristics that allow to meet the specifications of the test.

4.1 Wrapper

A wrapper for the device under test (DUT) is useful for all the cases in which direct mapping towards its ports is not feasible, for any reason such as the DUT being described in a different language (e.g. VHDL), as it occurs in this case.

The wrapper can do this after having instantiated the DUT. All other SystemVerilog objects willing to connect to the DUT can easily communicate to the wrapper instead.

4.2 Interface

As shown in Figure 4.1, the interface is in charge of connecting the signals of the `uvm_test` object and the DUT wrapper. This is done as follows:

1. Sample the relevant signals inside the datapath. In this case, we need to sample particularly the instruction word signal, the register file, the program counter and the data memory.
2. Send the commands (or instructions) to the command monitor. The instructions are obtained from sampling the instruction word signal inside the processor. This information will be used for both the coverage and the scoreboard.
3. Send the relevant signals of the DUT to the output monitor. For this purpose, the interface requires a handle of the output monitor, in order to use the *write_to_monitor* function.
4. Handle the start-up sequence of the processor by masking the first instructions and delaying the instruction word, as it occurs in the pipeline.
5. Allow the synchronization between the UVM test and the DUT by means of a task that delays for one clock cycle.
6. Generate the clock signal and define an initial reset routine. This second part is crucial for resetting all the registers in the DUT.

These functions of the interface are contained inside the different `always` statements of the interface. The file related to the interface is `dlx_if.sv`.

One of the most relevant functions of the interface corresponds to the part that samples the different signals inside the DUT according to the type of instruction that has been executed. The following code snippet shows how this is performed in the SystemVerilog interface:

```
1 // Obtain relevant word and set data_out
2 if ( instrType == i_type ) begin
3   if ( instrOpcode == itype_sw ) begin
4     reg1_val  = top.DUT.DLX_DUT.DP.RF.REGISTERS[int'(rs1)];
5     sw_addr   = int'(reg1_val + imm);
6     data_out = top.DUT.DLX_DUT.DP.RAM.DATAMEM[sw_addr];
7   end else if ( instrOpcode == itype_nop ) begin
```

```
8       data_out = 0;
9     end else if ( instrOpcode == itype_beqz || instrOpcode == itype_bnez ) begin
10      data_out = top.DUT.DLX_DUT.current_PC;
11    end else begin
12      data_out = top.DUT.DLX_DUT.DP.RF.REGISTERS[int'(rd)];
13    end
14  end else if ( instrType == r_type ) begin
15    data_out = top.DUT.DLX_DUT.DP.RF.REGISTERS[int'(rd)];
16  end else if ( instrType == j_type ) begin
17    data_out = top.DUT.DLX_DUT.current_PC;
18  end
19
20  if ( start_work_counter == 8)
21        output_monitor_h.write_to_monitor(data_out);
```

As shown in the previous listing, the instruction type and opcode define the data that will be set to the `data_out` variable. Then, the `write_to_monitor` method is used for writing the data that has been collected to the output monitor.

4.3 Sequence item

Even though the current test does not create any stimuli from the driver, since all the stimuli are within the instruction memory, it is still necessary to define a sequence item class that will be useful to create objects that contain the information about the instructions. In this case, the sequence item is expected to comprise all the possible fields that are within an instruction:

- `instr_type`: corresponds to an enumeration that can take three possible values depending on the three possible instruction types.
- `func_field`: variable related to the function field in the instruction. Used only for *R-type* instructions.
- `opcode_field`: 6-bits variable used for defining the opcode on the instruction. Used for all the instructions.
- `rs1`: address of the first source register of the instruction. Used for all instructions.
- `rs2`: address of the second source register of the instruction. Used only for *R-type*.
- `rd`: address of the destination register. Used for all instructions.
- `immediate`: value of the immediate. Not used for *R-type*.

These variables have been defined as random (`rand`) variables. However, in this test they are not randomized because the instructions are not generated by the driver but stored in the instruction memory.

4.4 Normal sequence

This sequence class extends from `uvm_sequence` in order to exploit the UVM factory. Then, we specify that this sequence will handle items of the type `sequence_item` (see section 4.3). This sequence has a task that is in charge of raising a flag when the test is being executed and also in charge of stopping the test. This is performed ash shown in the following listing:

```
1  task body();
2    instr_seq = sequence_item::type_id::create("instr_seq");
3
```

```
  i = 0;
  while (max_PC > current_PC) begin : normal_loop
    start_item(instr_seq);
    finish_item(instr_seq);
    i++;
    if (i > 1000) max_PC = 0;
  end : normal_loop
endtask : body
```

The value `max_PC` has been obtained from the scoreboard while it reads the instruction memory file. Then, the test will stops when either one of the two following events occur:

- The value of the current program counter reaches its maximum possible value (`max_PC`)
- The number of clock cycles reaches 1000.

The second condition is useful to avoid having an infinite test when there are infinite loops in the program that is being executed.

4.5 Testers

The tester is defined as a class called `normal_test` extended from `uvm_test`. In this class a handler of the environment and of the sequencer are declared (and built).

```
class normal_test extends uvm_test;
  `uvm_component_utils(normal_test);

  env               env_h;
  sequencer         sequencer_h;
```

After that, the `run_phase` task is coded. This task corresponds to the method that will be called when we do `run_test()` from the top module. This task holds a raised objection until the normal sequence detects the end of the program execution, using the conditions described in section 4.4. Afterwards, the objection is dropped and the test will finish.

```
task run_phase (uvm_phase phase);
    normal_sequence normal_seq;
    normal_seq = new("normal_seq");

    phase.raise_objection(this);
    normal_seq.start(sequencer_h);
    phase.drop_objection(this);
  endtask : run_phase
```

4.6 Driver

A class `driver` is created and it is extended from `uvm_driver` and it will handle items of the type `sequence_item`. A local handler of the DLX interface has been defined because, as shown in Figure 4.1, the driver is connected to the DUT through the interface. In fact, its aim is to retrieve the commands from the sequencer and then send them to the DUT through the interface.

```
class driver extends uvm_driver #(sequence_item);
  `uvm_component_utils(driver);
```

```
3
4   virtual dlx_if ddlx_if;
```

Then, the `build_phase` is created. During this function, the driver needs to retrieve a handler of the object of the interface. This process is performed using the UVM factory methods:

```
1  function void build_phase (uvm_phase phase);
2     if(!uvm_config_db #(virtual dlx_if)::get(null, "*", "ddlx_if", ddlx_if))
3       `uvm_fatal("DRIVER", "Could not retrieve the Interface ddlx_if");
4   endfunction : build_phase
```

The `run_phase` method is shown in the following listing:

```
1  task run_phase (uvm_phase phase);
2     sequence_item s_item;
3
4     forever begin : s_item_loop
5       seq_item_port.get_next_item(s_item);
6       ddlx_if.wait_one_cycle();
7       seq_item_port.item_done();
8     end : s_item_loop
9   endtask : run_phase
```

It contains an infinite loop, that waits for the commands from the sequencer and writes them to the DUT. In this task, another task from the interface is called, named `wait_one_cycle()`, thanks to which the execution of the `run_phase` will not finish until this interface task has been performed. Its aim is that of synchronizing the driver with the test. Once the `item_done` method is called, the sequencer understands that the driver is ready to receive a new command.

4.7 Scoreboard

The scoreboard performs two main checks on the behaviour of the DLX; one aims at checking that every fetched instruction triggers the correct behaviour of the DLX. The other one checks for the correct flow of instructions to be respected, in order to account also for errors of the fetch and branch unit.

4.7.1 Single instruction check

The purpose of this part of the scoreboard consists on receiving both the command and the response to such command from the monitors described before and then check if the corresponding output corresponds to the expected output. In order to do so, three different data structures are created in the scoreboard. The purpose of these variables is to work as clones of the internal memory elements of the DUT:

```
1  bit [rf_size-1:0][word-1:0]   cloned_regFile;
2  bit [dram_size-1:0][word-1:0] cloned_datamem;
3  bit [word-1:0]                cloned_PC = 20;
```

Notice that due to the five-instruction delay that is expected due to the pipeline, the program counter is initialized as 20, which corresponds to the fifth instruction considering that the program counter changes with increments of four.

Then, these "clones" need to be addressed both for updating and for obtaining the current stored value for performing the comparisons. This is the purpose of the function *predict_output*. This method receives the incoming instruction and returns an object of type *output_transaction* that will be compared with the arriving output from the output monitor. Notice that the command monitor retrieves the command as 32-bit instruction, then the *predict_output* function needs to decompose the instruction according to the instruction type. This is performed using the following if-else statements with the parametric size of each of the fields.

```
    if (s_item.instr_type == i_type) begin // I-type instr
      rs1_addr  = instruction[msb_add_rd1:lsb_add_rd1];
      rd_addr   = instruction[msb_add_rd2:lsb_add_rd2];
      immediate = instruction[msb_add_rd3:0];
      rs1_val = cloned_regFile[rs1_addr];
    end else if (s_item.instr_type == r_type) begin // R-type instr
      rs1_addr  = instruction[msb_add_rd1:lsb_add_rd1];
      rs2_addr  = instruction[msb_add_rd2:lsb_add_rd2];
      rd_addr   = instruction[msb_add_rd3:lsb_add_rd3];
      funct     = instruction[lsb_add_rd3-1:0];
      rs1_val   = cloned_regFile[rs1_addr];
      rs2_val   = cloned_regFile[rs2_addr];
    end else if (s_item.instr_type == j_type) begin // J-type instruction
      j_immediate = instruction[ir_size-op_code_size-1:0];
    end else begin
      $fatal(1, "Unexpected instruction type. Does not correspond to any of the known
          instructions.");
    end
```

Then, a switch-case statement is used for replicating the operation according to the function that is performed by the instruction. Inside each of the cases the cloned data memory, program counter and register file are updated. The following listing shows a section of the switch-case structure for some of the instructions. Notice that the predicted output is created according to the operation and the clones are read and updated according to the operation.

```
    case (opcode)
      rtype:  begin
        case (funct)
          rtype_sll : begin
            predicted.data_o = rs1_val << rs2_val[4:0];
            cloned_regFile[rd_addr] = predicted.data_o;
          end
          rtype_srl : begin
            predicted.data_o = rs1_val >> rs2_val[4:0];
            cloned_regFile[rd_addr] = predicted.data_o;
          end
        (...)
        endcase
      itype_addi: begin
        predicted.data_o = rs1_val + immediate;
        cloned_regFile[rd_addr] = predicted.data_o;
      end
      itype_subi: begin
        predicted.data_o = rs1_val - immediate;
        cloned_regFile[rd_addr] = predicted.data_o;
      end
      itype_beqz: begin
        if (rs1_val == 0)
          predicted.data_o = signed'(cloned_PC) + immediate - 12;
        else
          predicted.data_o = signed'(cloned_PC);
        cloned_PC = predicted.data_o;
      end
      itype_bnez: begin
```

```
30          if (rs1_val != 0)
31            predicted.data_o = signed'(cloned_PC) + immediate - 12;
32          else
33            predicted.data_o = signed'(cloned_PC);
34          cloned_PC = predicted.data_o;
35        end
36        itype_lw: begin
37          predicted.data_o = cloned_datamem[immediate + rs1_val];
38          cloned_regFile[rd_addr] = predicted.data_o;
39        end
40        itype_sw: begin
41          predicted.data_o = cloned_regFile[rd_addr];
42          cloned_datamem[immediate + rs1_val] = predicted.data_o;
43        end
44        (...)
45      endcase
```

Inside the *predict_output* function it is necessary to update the value of the program counter:

```
1     // Increment Program Counter
2     cloned_PC = cloned_PC + 4
```

Finally, outside the *predict_output* function, the *compare* method of the *output_transaction* is used for performing the comparison among the two values. If the comparison shows that the values do not correspond, an error is raised. In summary, the following structure is used:

```
1     if (!predicted.compare(t))
2       `uvm_error("SELF CHECKER", {"FAIL: ", data_str})
3     else
4       `uvm_info ("SELF CHECKER", {"PASS: ", data_str}, UVM_HIGH)
```

Notice that the instruction (or command) is obtained using the FIFO provided by UVM: *uvm_tlm_analysis_fifo*. An instruction is always retrieved with the *try_get* method since the write in the command monitor is always performed first than the write to the output monitor, as defined in the interface.

4.7.2 Instruction flow check

After the check for the correctness of the instruction just issued, the function for checking the correct flow of the instructions is called, with the line:

```
1     predict_instr_flow(instr);
```

The first operation performed by this function is to increment the expected value of the program counter by one at the line:

```
1     predicted_PC = predicted_PC + 1;
```

Notice this variable is incremented by 1 at every clock cycle, differently from the DLX's PC that is incremented by 4, since it is used to address the clone of the IRAM declared ad a global matrix of bits at the line :

```
1     bit [iram_size-1:0][ir_size-1:0] cloned_IRAM;
```

Then the instruction word received as a parameter of the function, that is the one just fetched by the DLX, is compared with the one from the .mem file. This operation is done at the following lines:

```
1     if(cloned_IRAM[predicted_PC-1] != instruction)
2       `uvm_error("FLOW CHECKER", {"FAIL: flow violation"})
```

```
    else
      `uvm_info ("FLOW CHECKER", {"PASS: flow respected"}, UVM_HIGH)
```

Moreover if the instruction from the cloned IRAM is of jump or branch type, the `predicted_PC` is modified accordingly. Notice for every computation the expected values are used, i.e. the ones from the cloned RF and cloned IRAM, in order to always compute the correct PC.

4.8 Coverage

The coverage class is extended from `uvm_subscriber`, that eases the connection in the environment, since it contains *analysis_export*, and has a parameter of type *sequence_item*.

```
class coverage extends uvm_subscriber #(sequence_item);
  `uvm_component_utils(coverage)
```

After that, a covergroup with one coverpoint is defined.

```
covergroup IR_covg;

    coverpoint instr_type{
      bins itype = {i_type};
      bins rtype = {r_type};
      bins jtype = {j_type};
    }

endgroup
```

The `sample()` method is called to fill the bins every time a specific case is generated. Then in the `extract_phase` the method `get_coverage` is called, in order to store the value of the overall coverage in the variable `instr_coverage`

```
    instr_coverage = IR_covg.get_coverage();
```

The results of the coverage of the different bins are shown in figure 4.2. Eventually, in the *report_phase* the result is

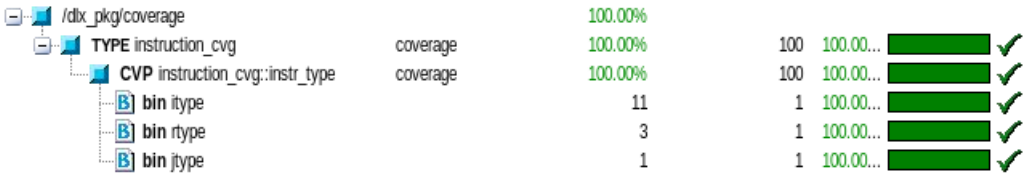


Figure 4.2: Coverage results

also print to monitor:

```
 function void report_phase(uvm_phase phase);
    `uvm_info("REPORT", $sformatf("Instruction coverage is %d", instr_coverage), UVM_LOW);
  endfunction: report_phase
```

The coverage obtained in the example in Figure 4.2, is 100%, even if only 15 instructions have been tested. This is due to the strategy of filling one bin out of three every time a new instruction type has been called. One simple way to improve the computation of the coverage here, is to create on bin for every single instruction in the instruction set.

4.9 Environment

Considering again figure 4.1, the environment comprises the sequencer, monitors, driver, coverage and scoreboard. The `env` is a class extended from `uvm_env`. Indeed it describes the whole UVM environment in which the test will be performed.
First of all it instantiates one object for every class described before, as follows:

```
class env extends uvm_env;
  `uvm_component_utils(env);

  sequencer        sequencer_h;
  coverage         coverage_h;
  scoreboard       scoreboard_h;
  driver           driver_h;
  command_monitor  command_monitor_h;
  output_monitor   output_monitor_h;
```

During the build phase, all the relevant objects have been created. A variable declared in the `DLX_package`, named `max_PC` is used to store the length of the program.

```
//a variable from the pkg that will store the length of the program
max_PC = scoreboard_h.read_mem_file();
```

Then the connect phase is defined, creating by that the connections using the analysis port objects.

```
function void connect_phase (uvm_phase phase);
    driver_h.seq_item_port.connect(sequencer_h.seq_item_export);

    command_monitor_h.ap.connect(coverage_h.analysis_export);
    command_monitor_h.ap.connect(scoreboard_h.instr_f.analysis_export);
    output_monitor_h.ap.connect(scoreboard_h.analysis_export);
endfunction : connect_phase
```

Using the handlers declared in the class `env`, we can access to the analysis port and connect the different elements. In this case, both the coverage and the scoreboard will act as subscribers of the different monitors' analysis port.