Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: Group 52

Edoardo Manfredi, Lorenzo Ruotolo, Agostino Saviano

October 16, 2023

Contents

CHAPTER 1

Introduction

The DLX implemented by this project is a complete 32-bit, five stages pipelined CPU that follows the MIPS ISA. It is composed of a datapath subdivided in the classic 5 stages (Fetch, Decode, Execute, Memory Access, Write Back) managed by a control unit that works in tandem with a forwarding unit for data forwarding between stages and a hazard detection unit for data and control hazards.

With respect to the base version, this DLX sports:

- data forwarding between stages for higher instruction throughput;
- control and data dependencies hazard detection;
- extendend instruction set: instructions for unsigned operands, set true on various conditions (equality, inequality, greatness or lessness than and more), JR for jumping to address pointed by a register;
- double frequency write-read register file that is capable of reading in the first half of a c.c. and writing in the second one;
- additional custom synthesis and simulation scripts.

1.1 Supported instructions

The following Table 1.1 provides an overview of the Instruction Set Architecture (ISA) for the DLX processor and it includes details about the instruction's opcode, operand format, full name, and a brief description of its function. This ISA is composed of a variety of instructions, including arithmetic, logical, data transfer, control flow, and comparison operations.

OPCODE	Operands	Full Name	Function
ADD	rd, rs, rt	Add	rd = rs + rt
ADDU	rd, rs, rt	Add	rd = rs + rt
SUBI	rt, rs, imm	Subtract Imm.	rt = rs - imm
SUBU	rd, rs, rt	Subtract	rd = rs - rt
AND	rd, rs, rt	AND	rd = rs & rt
OR	rd, rs, rt	OR	rd = rs | rt
XOR	rd, rs, rt	XOR	rd = rs ^ rt
SLL	rd, rs, imm	Shift Left Logical	rd = rs « imm
SRL	rd, rs, imm	Shift Right Logical	rd = rs » imm
SEQ	rd, rs, rt	Set if Equal	rd = (rs == rt) ? 1 : 0
SNE	rd, rs, rt	Set if Not Equal	rd = (rs != rt) ? 1 : 0
SLT	rd, rs, rt	Set if Less Than	rd = (rs < rt) ? 1 : 0
SLTU	rd, rs, rt	Set if Less Than Uns.	rd = (rs < rt) ? 1 : 0
SGT	rd, rs, rt	Set if Greater Than	rd = (rs > rt) ? 1 : 0
SGTU	rd, rs, rt	Set if Greater Than Uns.	rd = (rs > rt) ? 1 : 0
SLE	rd, rs, rt	Set if Less Than or Equal	rd = (rs <= rt) ? 1 : 0
SLEU	rd, rs, rt	Set if Less Than or Equal Uns.	rd = (rs <= rt) ? 1 : 0
SGE	rd, rs, rt	Set if Greater Than or Equal	rd = (rs >= rt) ? 1 : 0
SGEU	rd, rs, rt	Set if Greater Than or Equal Uns.	rd = (rs >= rt) ? 1 : 0
ADDI	rt, rs, imm	Add Imm.	rt = rs + imm
ADDUI	rt, rs, imm	Add Uns. Imm.	rt = rs + imm
SUBUI	rt, rs, imm	Subtract Uns. Imm.	rt = rs - imm
ANDI	rt, rs, imm	AND Imm.	rt = rs & imm
ORI	rt, rs, imm	OR Imm.	rt = rs | imm
XORI	rt, rs, imm	XOR Imm.	rt = rs ^ imm
SLLI	rt, rs, imm	Shift Left Logical Imm.	rt = rs « imm
SRLI	rt, rs, imm	Shift Right Logical Imm.	rt = rs » imm
SEQI	rd, rs, rt	Set if Equal Imm.	rd = (rs == rt) ? 1 : 0
SNEI	rd, rs, rt	Set if Not Equal Imm.	rd = (rs != rt) ? 1 : 0
SLTI	rd, rs, rt	Set if Less Than Imm.	rd = (rs < rt) ? 1 : 0
SGTI	rd, rs, rt	Set if Greater Than Imm.	rd = (rs > rt) ? 1 : 0
SLEI	rd, rs, rt	Set if Less Than or Equal Imm.	rd = (rs <= rt) ? 1 : 0
SGEI	rd, rs, rt	Set if Greater Than or Equal Imm.	rd = (rs >= rt) ? 1 : 0
SLTUI	rd, rs, rt	Set if Less Than Uns. Imm.	rd = (rs < rt) ? 1 : 0
SGTUI	rd, rs, rt	Set if Greater Than Uns. Imm.	rd = (rs > rt) ? 1 : 0
SLEUI	rd, rs, rt	Set if Less Than or Equal Uns. Imm.	rd = (rs <= rt) ? 1 : 0
SGEUI	rd, rs, rt	Set if Greater Than or Equal Uns. Imm.	rd = (rs >= rt) ? 1 : 0
BEQZ	rs, imm	Branch Equal to Zero	if (rs == 0) PC = PC + imm
BNEZ	rs, imm	Branch Not Equal to Zero	if (rs != 0) PC = PC + imm
LW	rt, offset(rs)	Load Word	rt = *(int*)(offset + rs)
SW	rt, offset(rs)	Store Word	*(int*)(offset + rs) = rt
J	target	Jump	PC = target
JAL	target	Jump and Link	$LR = PC + 4; PC = target
JR	rs	Jump Register	PC = rs
NOP	-	No Operation	No operation

Table 1.1: DLX Instruction Set Architecture (ISA)

CHAPTER 2

Functional Schema

2.1 Datapath

The following schematics (Fig. 2.1) shows how the Datapath has been organized. The different stages of the pipeline can be clearly differentiated by the different register blocks in gray. Stage registers are aligned to those blocks and signals and discrete components are highlighted as shown in the legend.

2.1.1 Instruction Fetch

This is the first stage of the pipeline. The Program Counter contains the address of the instruction to be fetched next from the instruction memory, or IRAM. It could be stopped of being updated, by the *PREFETCH_ENABLE* signal when needed. Since most of the jump and branching logic uses the immediately successive address to compute the target address, it gets precomputed with an adder in this stage to be sent to the IF/ID NPC register. PC is also updated directly with $PC + 4$ if there are not branch taken in execute stage.

A MUX, *NOP_MUX* is present just after the IRAM to select between the instruction inside the memory and a NOP to be sent to the Instruction Register *IR*. This is useful for hazards as it provides an easy way to introduce stalls into the pipeline.

2.1.2 Instruction Decode / Register Fetch

In the ID stage, the Control Unit reads the fetched instruction from the Instruction Register and begins to send out the signals needed to manage the whole pipeline.

Instructions that make use of registers fetch their values here: register fields from the instructions go directly to the selection ports *RS1* and *RS2* and the register file outputs their value into registers A and B. In case of an instruction with an immediate field, instead, its value is transformed and selected according to the needs of the fetched instruction:

- for signed and unsigned I-TYPE instructions (branch instructions are included) the immediate field gets extended sign-wise;
- for jump instructions the same operation is done but the result is sent to a different MUX

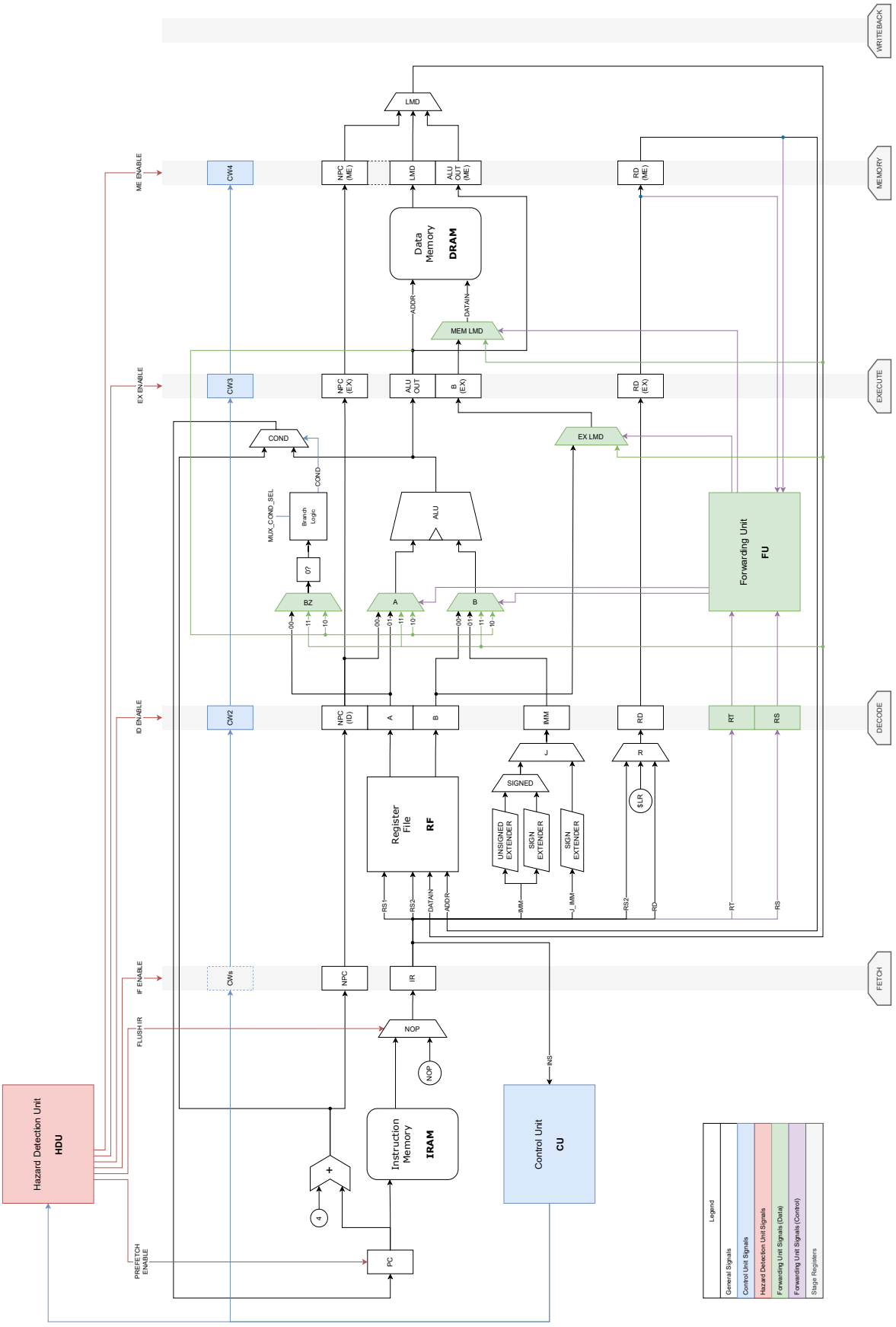

Figure 2.1: DLX Diagram

because the size of the immediate for J-TYPE instruction is different.

Together with stage registers *NPC* for the ID stage, *A*, *B* and *IMM*, there is also *RD*. It keeps track of the destination register of the current instruction, needed for later in the Write Back stage. Its value is controlled by MUX *R*, which selects between the registers in the fields of the instruction for I-TYPE and R-TYPE, or the Link Register in case of a Jump and Link instruction. In addition *RS* and *RD* registers are updated, with the first and second instruction operands respectively. These register are used by the Forwarding Unit to check data dependency when the instruction goes in execute stage and instruction in memory and write back stage.

2.1.3 Execute

Arguably the most important stage. It contains the ALU together with MUXes for selecting operands feeding it: registers *A* and *B*, immediate values, addresses from the Next Program Counter and operands forwarded from other stages controlled by signals coming from the FU. The ALU is not only responsible for all arithmetic and logic operations, but also for address computations. Further details on the implementation of the ALU are contained in the appropriate section.

In addition to the ALU, there are blocks for managing branch and jump instructions. A zero detector for the A register needed for BEQZ and BNEZ, a small combinational block for the branching logic that outputs whether the branch is taken or not according to the instruction and the zero detector, and a MUX controlled by the previously mentioned block that writes to the PC the address computed by the ALU if the branch is to be taken.

Stage registers are present for capturing the outputs: *ALU_OUT* for the output of the ALU, *B (EX)* for loading words into memory bypassing the ALU altogether. Registers *NPC (EX)* and *RD (EX)* are also present to pass the values of *NPC* and *RD* to the next stage.

The muxes at the input of the ALU have additional inputs, allowing the Forwarding Unit to select as ALU input data coming from memory or write back. The same idea is applied for the zero detector, in order to enable forwarding paths for branches instructions, and for the value written in *B_EX*. This last one enables the forwarding for `sw` instructions.

2.1.4 Memory

This stage is dedicated to the storing and retrieval of words from DRAM. Registers *ALU_OUT* and *B (EX)* are used in conjunction for memory accesses. For a store, just the address is used and the DRAM outputs the corresponding word into register *LMD*. For a load, address from *ALU_OUT* and word from *B (EX)* are provided to the DRAM, that takes care of storing the word. If no memory access is needed, DRAM is skipped. Port *DATAIN* is MUXed to provide a way for forwarded words to be obtained from write-back stage. In addition, *ALU_OUT* is connected with the muxes in execute stage in order to be reused.

2.1.5 Write Back

The last stage of the pipeline deals with writing back the results of a previous computation (if it was an arithmetic or logical operation) or a retrieval from memory (if it was a load instruction) to the register file at the very beginning of the pipeline. It is bare-bones, consisting of just a MUX *MUX_LMD* that selects between the NPC, the Loaded Memory Data and the ALU output, all from

the Memory stage. As before, two forwarding paths connecting the MUX output with memory and execute stages are also present.

2.2 Functional Blocks

2.2.1 Control Unit

The Control Unit is in charge of managing the datapath throughout its stages. Given an instruction from IR, it generates the corresponding control word that manages the various registers, MUXes and other control signals in the datapath. It also receives a block of *STALL* signals from the HDU (Hazard Detection Unit) that instructs the CU on when to stall the pipeline in case of data hazards.

It is implemented as an hybrid between an hardwired and a microcoded CU, with two processes that simply associate every instruction with a given control word and every ALU opcode with the corresponding operation, plus a process that manages the transition between every stage taking into consideration the *STALL* signals from the HDU.

The control word corresponding to the instruction in decode is purely combinational, changing as soon as IR is updated. In the next clock cycles, this control word is latched into a register, becoming a nop control word if a stall should be inserted. The control word is therefore propagated in memory and, finally, in write-back. The actual control word seen by the datapath is a combination of these register, selecting in each of them only the portion needed in a particular stage.

2.2.2 Register File

The register file included in the DLX contains 32 registers, each of word size equal to 32 bits. Following the MIPS ISA, register *R0* has its value hardcoded to 0 and register *R31* is the *Link Register* used for returning from subroutines.

It has been chosen to be a Double-read/Single-write type, as in: on every clock cycle, the register file can output the values of two registers at the time by providing addresses on ports *ADD_RD1* and *ADD_RD2* and requesting reads on ports *RD1* and *RD2*; a write can be made by setting an address on *ADD_WR* and a word on port *DATAIN*. Requested registers output their value on ports *OUT1* and *OUT2*. This is possible by having the writing take place in the first half of clock period, and the reading in the reaming half. This choice helps removing a stall when the data written into the RF in write-back stage should be used in decode stage. This is a kind of forwarding through the RF.

2.2.3 ALU

The ALU of the DLX operates on two inputs: *DATA1* and *DATA2*. It can obtain them from a multitude of sources, namely:

- from the register file, through 2 stage registers A and B;
- from the program counter, necessary for address computations in branching and jump instructions;
- from the instructions themselves, immediate values via the instruction register after the appropriate transformations;

- from the ALU itself, and in general from other stages' outputs through MUXes controlled by the forwarding unit.

The function to be computed on the operands is selected by a third input *FUNC* that receives a code representing it and the result is sent to the output *OUTALU*. A list of currently implemented functions follows. The implementation of each operation is behavioural unless specified otherwise.

Addition

$ALUOUT = DATA1 + DATA2$

A P4 adder is present inside the ALU to perform additions. Further details on its implementation together with the VHDL description are included in lab 2's zip file.

Subtraction

$ALUOUT = DATA1 - DATA2$

The P4 adder is also used for subtractions, by means of negating one of the inputs and setting the carry-in input of the adder to 1.

AND

$ALUOUT = DATA1 \wedge DATA2$

OR

$ALUOUT = DATA1 \vee DATA2$

XOR

$ALUOUT = DATA1 \oplus DATA2$

Logical Shift Left

$ALUOUT = DATA1 \ll DATA2$

Logical Shift Right

$ALUOUT = DATA1 \gg DATA2$

Set equal

$if (DATA1 == DATA2)\ then\ ALUOUT = 1\ else\ 0$

Predicate	Instructions
$LMD\_EN == 1$	sw
$MUX\_J\_SEL == 1$	j or jr or jal
$MUX\_COND\_SEL == 10$	bne
$MUX\_COND\_SEL == 01$	bez

Table 2.1: Predicates on control signals used by the Hazard Detection Unit to distinguish an instruction from another one.

Set not equal

$if(DATA1 \neq DATA2)\ then\ ALUOUT = 1\ else\ 0$

Set greater than or Equal (signed and unsigned)

$if(DATA1 \geq DATA2)\ then\ ALUOUT = 1\ else\ 0$

Set greater than (signed and unsigned)

$if(DATA1 > DATA2)\ then\ ALUOUT = 1\ else\ 0$

Set less than or Equal (signed and unsigned)

$if(DATA1 \leq DATA2)\ then\ ALUOUT = 1\ else\ 0$

Set less than (signed and unsigned)

$if(DATA1 < DATA2)\ then\ ALUOUT = 1\ else\ 0$

2.2.4 Hazard Detection Unit

The Hazard Detection Unit takes care of detecting data and control hazards in the pipeline and successively dispatching stage enable (or STALL, as written in the CU subsection) signals to the CU for indication on where and for how many stages to stall for. It does so by progressively checking for hazardous conditions in a process starting from IRAM and DRAM readiness, then for branches (assuming not taken), jumps and loads from memory giving precedence, in case of instructions of the same type, to instructions currently found in the execute stage. If no hazardous condition is found, a STALL_CLEAR signal is sent out.

The Hazard Detection Unit (HDU) controls the enable signals of the pipeline registers and flushes IR register when needed. The mechanism is similar to the one used by the forwarding unit. It reads control signals to identify an instruction or a class of them, and, if necessarily, it compares registers coming from the datapath in order to check if a dependence exists. The table that summarizes these signals is 2.1.

These signals are needed only when the instruction is in a particular stage. For instance, when decoding a jump a nop should be fetched. As a consequence, the HDU needs to read the signal MUX_J_SEL of the instruction being in decode. In case of a jump an additional nop should be

fetched also when the jump is executing. As before, the HDU checks the predicate using the same signal but reading it from the control word of the instruction in the execute stage. In that way, the HDU behaves as a purely combinational block, avoiding keeping track of the state of the pipeline, at cost of adding extra signal connections. This choice seems to be reasonable, because only instructions that stall two clock cycles (i.e. jumps and branches) require the two copies.

Thanks to the FU there are only three cases in which a stall must be inserted. (1) Stalls waiting the completion of memory accesses, (2) data hazard due to use of loaded data, and (3) control hazard for jumps and branches.

1. Because it is likely that reading and writing the IRAM and DRAM take more than one clock cycle, the HDU should have a way to understand when these operations finish. Knowing a-priori the amount of stalls that a reading or a writing will take, designing the HDU accordingly would have not lead to an enough general solution. A better approach is exploiting the ready signals of memories, namely $IRAM\_READY$ and $DRAM\_READY$, which are routed to the HDU from the top level of the processor. Concerning the IRAM, as long as the ready is not asserted, IR must be filled with a nop instruction and the PC must be not increased. With regard to the DRAM, the pipeline must be frozen until the memory stage, but contrary to the previous case IR must maintain the instruction that is in decode. It can be said that the IRAM inserts a nop instruction at the beginning of the pipeline, while the DRAM inserts a bubble in the middle of that (after the memory stage). The bubble behaves like a nop, to the extent that it does not modify the state of the processor (i.e. register file and pipeline registers) and DRAM, although the bubble is not an actual instruction. Regarding all instructions that computes the target address in execute stage, jumping then into that address, they must insert a NOP instruction after them, because only after the execute stage the processor knows the address in which to jump. The penalty is two stalls in these cases. Taking jump instruction as example, when

2. When the instruction is a load, the loaded data can be reused in the pipeline after the memory stage, supposing that the memory requires just one cycle to retrieve it for simplicity. As a result, an instruction that needs the loaded data in the execute stage must wait at least one cycle in order to allow the load completing the memory stage, ensuring it is forwarded from the write back to the execute stage in the next clock cycle. Stalling for only one clock cycle is the ideal situation in which the DRAM has a delay less or equal to the clock cycle. If the memory takes more clock cycles to provide the result, additional stalls of type (1) are required. *lw* instruction is the only one that asserts the signal LMD_EN, therefore checking its value in the control word of the instruction in the execute stage allows blocking the instruction which arises the data dependence in the decode stage. Two additional checks on the corresponding register ensure if there is a data dependency or not. This check is somewhat anticipated in decode stage, while it could be delayed of one clock cycles. The difference would have been having an instruction blocked in the execute stage, blocking the next two instructions in decode and fetch. In terms of performance, this solution does not have obvious advantages with respect to the actual one, so they are equivalent.

3. Jumps and branches instructions compute the target address in the execute stage. Moreover, branches need to wait the execute stage to determine if the branch target address is taken or not. For those reasons, two nops must be inserted for sure after the jump instruction. In principle, inserting two nops would produce the correct result also in case of a branch, regardless of the branch outcome. However, in the case where the branch is not taken the two nops would be a waste, because the instruction to be executed after the branch would already be known when the branch is decoded. An optimization is assuming the branch not taken; it means that when the branch is in decode the next instruction is fetched anyway. This approach does not add any

Control Signal	Instructions
DRAM_READNOTWRITE	*sw*
DRAM_ENABLE	*sw* or *lw*
RF_WR	R-TYPE, I-TYPE

Table 2.2: Caption

clock cycles penalty if the prediction is correct, otherwise it is needed to add two nops, as in the non-optimised version. When a branch in execute, the signal coming from the branch logic is read. In case of incorrect prediction this signal is low, and the fetched instruction is flushed, ensuring that both IF/ID and ID/EX register are not updated, not propagating the instruction in the next stages. In that way, the erroneous fetched instruction disappears in favour of a NOP. To summarise, jumps stall the pipeline of two cycles, and branches do the same if the branch is not taken.

2.2.5 Forwarding Unit

The Forwarding Unit takes as input a subset of the control signals sent by the CU and of the register selection codes (not the values inside the registers themselves). Given those, it detects favourable conditions for source and destination registers between stages of the pipeline and if they match it forwards operands where they are needed by changing selection signals for the appropriate MUXes and skipping the write-back stage.

The forwarding unit helps improving the pipeline performances, reducing the number of stalls due to hazards in different situations. In order to do that, it receives signals both from the datapath and the control unit, and sends signals to the datapath to forward correctly the data. The situations in which the forwarding can be exploited are when an instruction has already produced the result without writing that in the register file, and a following instruction needs the data in the execute stage. The majority of forwarding paths links memory and write-back stages with the execute stage.

In order to distinguish an instruction from another, control signals that uniquely identify it are used. These control signals and their corresponding instruction are listed in Table 2.2.

The forwarding unit reads these signals only in the stage where the instruction they represent should be. For instance, RF_WR is read when $addr1, r2, r2$ is in the memory stage. That allows the forwarding occurs, if an instruction like $subi, r3, r1, \#1$ is in the execute stage.

There are three forwarding classes. The ALU inputs can be taken from the memory and write-back stages. For this purpose, the forwarding unit needs Rd and Rs registers of the instruction in execute stage and Rd of both instructions in memory and in write-back. It can then compare the registers in order to detect the data dependency, forwarding the result of the instruction in memory. Also fall in this case the offsets used to compute the target address in the *lw* and *sw* instructions. If the instruction is a branch instead, this forwarding path should not be enabled. In fact, RD_ID contains the value compared against zero, so the result should be forwarded to the unit that computes that, which is outside the ALU. When there are multiple instructions that read and write the same register subsequently, then the result in memory stage must take precedence over the write-back one. Basically, it is enough checking that the condition that activates the forwarding path between memory and execute is not true, before forwarding from write-back to execute.

As mentioned before, if the value compared against zero is computed by the previous instruction, then it should be forwarded from memory to execute, as input of the comparator. A similar case occurs

when it is available in write-back.

The remaining case to be treated involves *sw* instructions when a previous one instruction computes the value which must be written in memory stage. In such cases, the data input of DRAM is multiplexed with the register that stores the output of the memory itself. When the *sw* instruction is in the execute stage, the loaded value must be forwarded in advance and stored in a register in order to be used as input of the DRAM the next clock cycle.

CHAPTER 3

Simulation

3.1 Simulation

The simulation was performed using ModelSim or Vivado running locally in our machines.

3.1.1 Testbench

The testbench instantiates the DLX top level entity and the two memories IRAM and DRAM. These components are then connected together and the clock signal is generated. In order to bring the processor in a foreseeable state, the reset signal is asserted for half clock cycle. The testbench does not do anything apart that, because the IRAM automatically loads the program to be executed when the testbench starts.

3.1.2 Waveform

In order to verify the correct behavior, waveforms were checked by visual inspection. During the multiple debugging iterations, the script `wave.do` has been produced, avoiding to load manually one at a time the wave each time a new instance of ModelSim was opened. This script groups signals in their corresponding stage and includes signals coming from HDU, FU. It can be loaded by typing the following command in the terminal of ModelSim.

```
$ do wave.do
```

It has always been used as a starting point when it was necessarily to find a bug or checking a functionality.

3.1.3 Testing

Each implemented behavior has been tested with an ad-hoc program which stress the new feature. The problem with this approach was that the functionality tested up to that point could stop working, making it necessary to recheck all previous programs by hand. Because of these methodology-drawbacks, testing proved to be the most time-consuming and frustrating phase of the entire project.

Test Name	Description
`all_general_test.asm`	Executes all instruction supported by this processor
`dlx_div.asm`	Performs Euclidean division in an iterative manner
`dlx_mul.asm`	Performs multiplication in an iterative manner
`fwd_test_1.asm`	Checks forwarding paths of branch, load and store
`shift_imm.asm`	Immediate general testing
`bnez_lwsw.asm`	Checks for wrong forwarding paths when mixing instructions
`jr.asm`	Correctly forward and stall using a jr
`test_arithmetic_comments.asm`	General arithmetic instructions testing
`xor_swap.asm`	Checks forwarding when swapping two registers

Table 3.1: Programs executed to test the entire set of features.

3.1.4 Script

The `sim.sh` script is used to initialize the directory for the simulation. It creates the simulation directory if it does not exist, and copies the VHDL sources from `src`. Then it adds the simulator location in $\$PATH$, and creates the library called WORK. ModelSim compiler requires source file in compilation order, allowing to write them in a external text file, calling `vcom` with `-F` flag. The list of all components is written in `./src/components`. If the compilation has some error, the script exits, allowing to read the compilation outputs. Otherwise the simulation is started with the following command

```
$ vsim -t 10ps work.DLX_tb -voptargs=+acc
```

To be able to compile quickly the code after making changes, avoiding restarting the simulator from scratch, the last command is executed only if there are not vsim instances running. The entire script is listed A.1.

3.1.5 Tested programs

As mentioned before, ad-hoc programs have been written to test functionalities, that act as a kind of unit tests. For instance, to test that two nops were inserted after a `j`, only the minimum lines of assembly were written. So the code could be something like the following listed.

```
    addi r1, r1, #1
label:
    add  r1, r1, r1
    j    label
```

After the results by inspecting the waveform were satisfactory, a program requiring working the new feature and others ones at the same time is tested; these tests act as integration tests. At the end, the most relevant source files have been collected inside `res/asm_tests` and listed in table 3.1.

CHAPTER 4

Physical Design

4.1 Synthesis

4.1.1 Script

The synthesis phase was completely automated through scripting. The main bash script `syn.sh` (A.2) can be found inside the `scripts` folder. It consists of an initial general setup part, followed by loading a second TCL script `syn.tcl` (A.3) inside the invoked Synopsys Design Compiler `dc_shell` console for the actual synthesis part.

The script is expected to be run inside the root folder. The syntax is:

```
$ ./script/syn.sh [CLK_PERIOD]
```

The `CLK_PERIOD` is an optional argument that informs the synthesis tool about the desired target clock period value for the design in nanoseconds. This allows the tool to prioritize optimization efforts to achieve the specified clock period. If the argument is not specified, the default target will be set to 1 ns.

Setup

This part automates the general setup to allow the actual synthesis part to be effortlessly run later. It prepares several environment variables, which can be modified to match its usage in directory and file layouts differing from the one found in the submission.

It handles directory preparation for later, copying only the necessary source files from `$SRC_DIR`, while excluding testbenches and non-synthesizable blocks like IRAM and DRAM. Additionally, it copies other required resources from `$SYN_RES`, such as the library setup file and the component list for analysis. The script finally proceeds to the actual synthesis phase, where the Synopsys Design Compiler (`dc_shell`) is invoked.

Clock Period (ns)	Design Metrics		
	Worst Slack (ns)	Total Area (um^2)	Total Power (mW)
1.0	-0.48	16213	12.89
1.2	-0.27	16036	10.79
1.4	-0.09	15741	9.30
1.5	**0.00**	**15667**	**8.69**
1.6	0.00	15732	8.16
1.7	0.00	15579	7.71

Table 4.1: Comparison of different clock period results

Synthesis

After spawning the `dc_shell` and configuring the `$CLK_PERIOD` variable to match the value passed as an argument to the main bash script before, the TCL file `$SYN_TCL` is fed line by line to the shell.

The TCL script initializes various constants, including the component list parsed by the `components_syn` file, which was previously copied for use in the initial **analysis** phase. If the analysis ends without errors, the script proceeds to **elaborate** the design, providing synthesis-related information, such as the wire load model (in this case, `5K_hvratio_1_4` was used).

Next, it advances to the main phase: the design **compilation**.

```
$ compile -map_effort high
```

This is the critical segment of the script, where the Design Compiler synthesizes the design, aiming to minimize skews and meet the target clock period requirements throughout the entire design. The resulting Verilog **netlist** and SDC file are respectively stored in the `netlist` and `design_compiler_sdc` directories. Finally, the script generates three post-synthesis **reports** (timing, area, and power) inside the `reports` directory.

Following some final cleanup, the script concludes and the design is fully synthesized. All the generated files are organized by the target clock value specified at the beginning, allowing an easy comparison of multiple final reports to select the best clock period for the design.

4.1.2 Results

After testing with different clock periods, the value of 1.5 ns was selected. This choice was made based on a comparison of post-synthesis results using different periods, as shown in Table 4.1.

As expected, selecting a lower clock period target to improve performance leads to increased area and power consumption of the design. This trade-off is illustrated in Fig. 4.1 (Area-Timing Analysis) and Fig. 4.2 (Power-Timing Analysis). In both plots, **red** points are related to negative slack designs, **black** to positive slack designs and finally the 1.5 ns **green** point is the selected target.

The rationale behind this choice becomes evident when examining the two graphs. Selecting a clock period higher than 1.5 ns would not have provided significant benefits, in fact, the area associated with the 1.6 ns clock period design is even higher. On the other hand, a slight reduction in the clock period could potentially yield improved performance, however, since the 1.4 ns clock period did not meet the slack constraints, further testing was considered unnecessary.

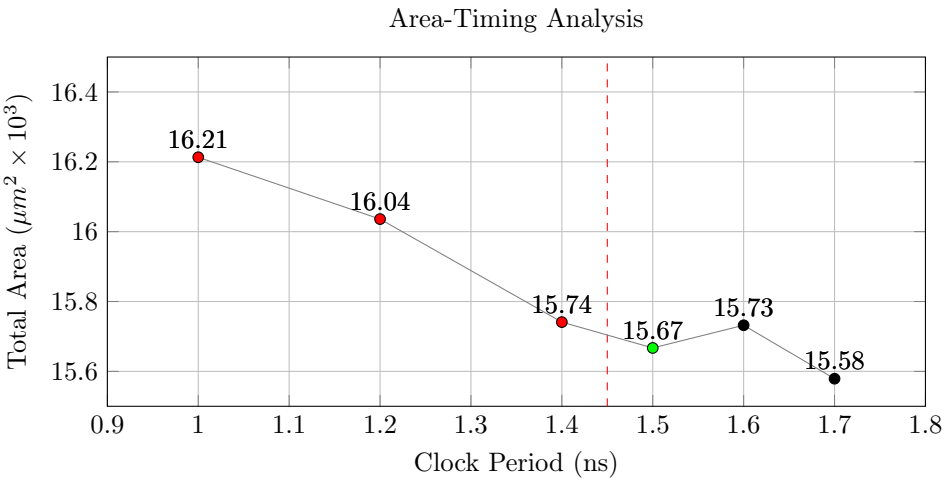

Figure 4.1: Area-Timing analysis

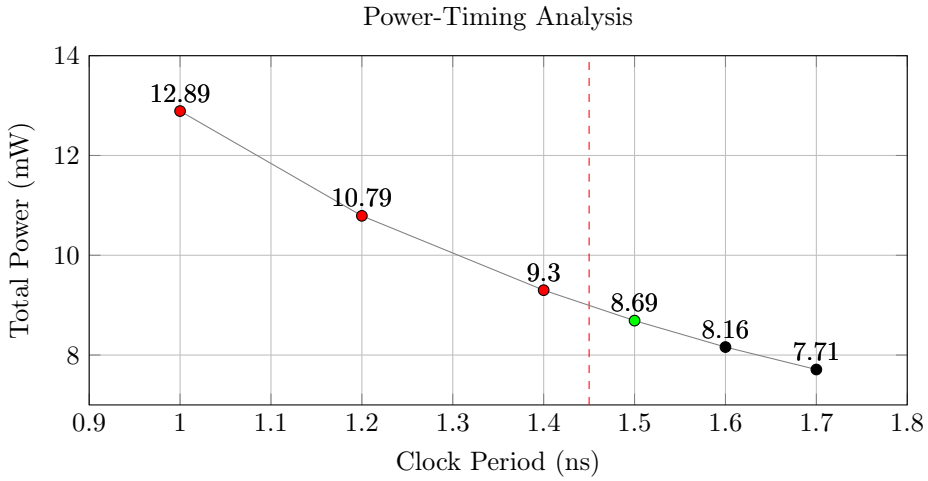

Figure 4.2: Power-Timing analysis

Eventually, given that no external constraints were imposed on timing, area, or power in the design, the decision to opt for a 1.5 ns clock period made sense. This means that the DLX processor is capable of operating at a frequency of up to approximately 0.66 GHz.

4.2 Post-Synthesis

The physical place and route phase were performed using the Cadence Innovus tool. A configuration file, similar to the one provided during lab sessions, was modified for the setup phase. The following steps are similar to those followed in the lab session.

First, the **floorplan** was defined with a core aspect ratio of 1.0, an utilization factor of 0.6, and core-to-die margins of 5µm. Then, the **power rails** were defined: the top-bottom rail used the M9 layer, while the left-right rail used the M10 layer. This was done to avoid congestion in the subsequent routing steps. To improve the distribution of power and ground signals towards the center, **vertical stripes** were added in the M10 layer with a distance of 20 between sets. Subsequently, **VDD and**

Ground routing to cells was performed to connect the standard cells that would be placed later with the main power rails.

With the power routing completed, the **logical cells** were placed using the remaining metal layers (M1 to M8). In addition to logical cells, also **filler cells** were placed in the remaining spaces not used by logical cells to improve the continuity of N+ and P+ wells in the rows.

Next, the **I/O pins** were added: all IRAM connections were placed on the left side of the chip, DRAM connections on the right, and the remaining single-wire clock and reset connections respectively on the top and bottom.

After that, optimizations were made, including **Clock-Tree-Synthesis** (CTS) optimization, to ensure that the design adhered to the specified timing constraints. The main **cell routing** was then performed using the NanoRoute tool. Finally, additional post-route optimizations were conducted with the same objective as the previous ones.

The design was successfully **verified** for connectivity and design rule compliance, and several reports were generated, such as the Standard Parasitic Exchange File (SPEF), that can be found inside the `report` folder.

4.2.1 Results

In Fig. 4.4 the Amoeba view of the processor is shown. As expected, the largest area is occupied by the Datapath. Notably, the Register File portion is highlighted in red, constituting more than half of the Datapath's total area. The remaining components are located in the top right corner: the Control Unit, Hazard Detection Unit, and Forwarding Unit occupy the rest of the chip's space. A general overview of the cells and connections is shown in Fig. 4.3.

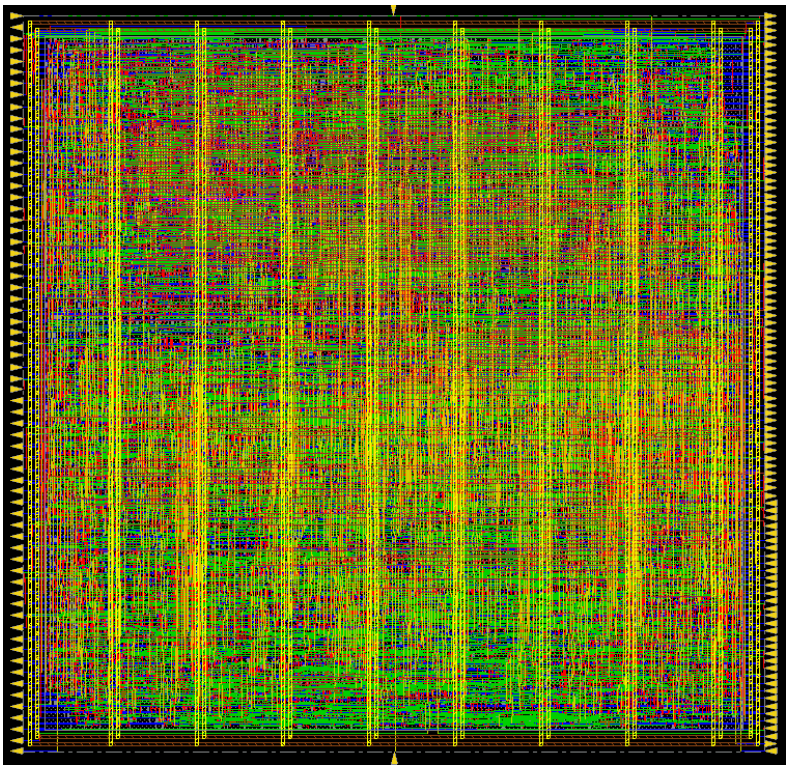

Figure 4.3: Synthesized DLX

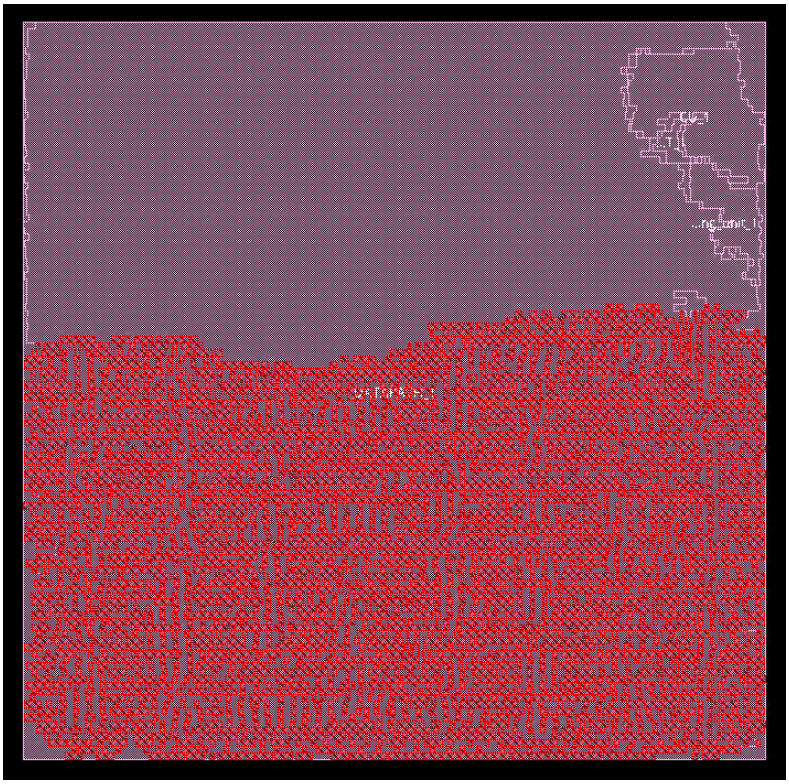

Figure 4.4: Amoeba view of the synthesized DLX

CHAPTER 5

Conclusions

Despite being a fairly featureful processor, many more improvements can be made to the architecture. One fundamental feature that is lacking is hardware support for multiplication, division and other multi-cycle operations. As said in the dedicated chapter for the ALU, building of a Long Latency ALU for these kinds of operation was planned but was put aside in favour of forwarding and hazard detection. Another possible improvement is faster address resolution for branch and jump instruction. This could be achieved by putting additional hardware for address computation inside the decode stage.

All in all, this project has seen a quite arduous road to completion. Such a complex endeavour required extensive testing and debugging, for which the tools have been proven to be severely lacking. Since a processor is tightly coupled and complicated, bugs in the hardware were especially difficult to spot. However, despite all these difficulties, building a working processor from scratch has been an extremely valuable and satisfying experience. Looking at a picture of, and reading the inner workings of a datapath can only make so much sense; actually building one together with a control unit and improving upon the whole project until it reaches a level that meets the expectations of the group: that is invaluable.

APPENDIX A

Scripts

A.1 Simulation

Listing A.1: Main BASH script for simulation.

```
#!/usr/bin/env bash

SRC_DIR='./src'
SYN_DIR='./syn'
SYN_RES='./res/syn'
SYN_TCL='./scripts/syn.tcl'
SYN_LIBRARYNAME='.synopsys_dc.setup'
CLK_PERIOD_DEF=1

GREEN="\e[32m"  # Green text color
RESET="\e[0m"   # Reset text color

echo -e "${GREEN}[INFO] Synthesis Setup...${RESET}"

if [ $# -lt 1 ]; then
    echo -e "${GREEN}[INFO] No clock period provided, using default $CLK_PERIOD_DEF ns period.${RESET}"
    CLK_PERIOD=$CLK_PERIOD_DEF
else
    # Save the first argument in CLK_PERIOD
    CLK_PERIOD="$1"
fi

if [ ! -d "$SYN_DIR" ]; then
    echo -e "${GREEN}[INFO] Folder $SYN_DIR does not exist yet, creating it now.${RESET}"
    mkdir -p "$SYN_DIR"
fi
mkdir -p "$SYN_DIR/work"
```

```
28 mkdir -p "$SYN_DIR/src"
29 mkdir -p "$SYN_DIR/src/datapath"
30
31 echo -e "${GREEN}[INFO] Copying sources from $SRC_DIR${RESET}"
32 find "$SRC_DIR" -maxdepth 1 -type f -name "*.vhd" -exec cp --update {} "
       $SYN_DIR/src/" \;
33 rsync -avm --include='*.vhd' --include='*/' --exclude='*' "$SRC_DIR/
       datapath/" "$SYN_DIR/src/datapath/"
34
35 echo -e "${GREEN}[INFO] Copying other resources from $SYN_RES${RESET}"
36 cp -ra --update $SYN_RES/* $SYN_DIR/
37 cp --update $SYN_RES/$SYN_LIBRARYNAME $SYN_DIR/
38
39 echo -e "${GREEN}[INFO] Setting up Design Vision${RESET}"
40 source /eda/scripts/init_design_vision
41
42 echo -e "${GREEN}[INFO] Setup done, launching Design Vision...${RESET}"
43 cd $SYN_DIR
44 dc_shell-t -no_gui << EOF
45     set CLK_PERIOD $CLK_PERIOD
46     source ../$SYN_TCL
47 EOF
```

A.2 Synthesis

A.2.1 Bash Script

Listing A.2: Main BASH script for synthesis.

```
1  #!/usr/bin/env bash
2
3  SRC_DIR='./src'
4  SYN_DIR='./syn'
5  SYN_RES='./res/syn'
6  SYN_TCL='./scripts/syn.tcl'
7  SYN_LIBRARYNAME='.synopsys_dc.setup'
8  CLK_PERIOD_DEF=1
9
10 GREEN="\e[32m"  # Green text color
11 RESET="\e[0m"   # Reset text color
12
13 echo -e "${GREEN}[INFO] Synthesis Setup...${RESET}"
14
15 if [ $# -lt 1 ]; then
16     echo -e "${GREEN}[INFO] No clock period provided, using default
         $CLK_PERIOD_DEF ns period.${RESET}"
17     CLK_PERIOD=$CLK_PERIOD_DEF
18 else
19     # Save the first argument in CLK_PERIOD
```

```
20      CLK_PERIOD="$1"
21  fi
22
23  if [ ! -d "$SYN_DIR" ]; then
24      echo -e "${GREEN}[INFO] Folder $SYN_DIR does not exist yet, creating
            it now.${RESET}"
25      mkdir -p "$SYN_DIR"
26  fi
27  mkdir -p "$SYN_DIR/work"
28  mkdir -p "$SYN_DIR/src"
29  mkdir -p "$SYN_DIR/src/datapath"
30
31  echo -e "${GREEN}[INFO] Copying sources from $SRC_DIR${RESET}"
32  find "$SRC_DIR" -maxdepth 1 -type f -name "*.vhd" -exec cp --update {} "
        $SYN_DIR/src/" \;
33  rsync -avm --include='*.vhd' --include='*/' --exclude='*' "$SRC_DIR/
        datapath/" "$SYN_DIR/src/datapath/"
34
35  echo -e "${GREEN}[INFO] Copying other resources from $SYN_RES${RESET}"
36  cp -ra --update $SYN_RES/* $SYN_DIR/
37  cp --update $SYN_RES/$SYN_LIBRARYNAME $SYN_DIR/
38
39  echo -e "${GREEN}[INFO] Setting up Design Vision${RESET}"
40  source /eda/scripts/init_design_vision
41
42  echo -e "${GREEN}[INFO] Setup done, launching Design Vision ...${RESET}"
43  cd $SYN_DIR
44  dc_shell-t -no_gui << EOF
45      set CLK_PERIOD $CLK_PERIOD
46      source ../$SYN_TCL
47  EOF
```

A.2.2 TCL Script

Listing A.3: TCL script for synthesis used inside Design Compiler.

```
1  #!/usr/bin/env tclsh
2
3  # colors
4  proc color {foreground text} {
5      return [exec tput setaf $foreground]$text[exec tput sgr0]
6  }
7
8  proc print_info {message} {
9          puts [ color 2 "\[INFO\] $message" ]
10 }
11
12 proc print_error {message} {
13         puts [ color 1 "\[ERROR\] $message" ]
14 }
```

```
15
16 puts "\n"
17 print_info "Starting␣Synthesis."
18
19 # clock
20 print_info "Clock␣period:␣$CLK_PERIOD␣ns"
21 set target_clk_ns $CLK_PERIOD
22
23 # constants
24 set design_entity "dlx"
25 set design_arch "rtl"
26 set fcomp_name "./components_syn"
27 set fcomp [open $fcomp_name r]
28 set fanalyze [read $fcomp]
29
30 # components analysis
31 print_info "Starting␣Analysis."
32 set errors 0
33 foreach component [split [string trim "$fanalyze" "\n"] "\n"] {
34         if {[analyze "$component" -library WORK -format VHDL] == 0} {
35                 set errors 1
36         }
37 }
38 if { $errors } {
39         print_error "Errors␣during␣Analysis"
40         exit
41 }
42 print_info "Done:␣Analysis"
43
44 # elaboration and synthesis settings
45 print_info "Starting␣Elaboration."
46 elaborate "${design_entity}" -update -architecture "${design_arch}"
       -library WORK
47 set_wire_load_model -name 5K_hvratio_1_4
48 create_clock -name "CLK" -period $target_clk_ns CLK
49 set_fix_hold "CLK"
50 set_max_delay $target_clk_ns -from [all_inputs] -to [all_outputs]
51 print_info "Done:␣Elaboration"
52
53 # design compile
54 print_info "Starting␣Compile."
55 compile -map_effort high
56 print_info "Done:␣Compile"
57
58 # generate netlist
59 print_info "Generating␣Netlist."
60 exec mkdir -p netlists
61 write -format verilog -hierarchy -output "./netlists/${design_entity}_${
       target_clk_ns}ns_postsyn.v"
```

```
62 exec mkdir -p design_compiler_sdc
63 write_sdc "./design_compiler_sdc/${design_entity}_${target_clk_ns}
       ns_postsyn.sdc"
64 print_info "Done: Netlist"
65
66 # generate reports
67 print_info "Generating Reports."
68 exec mkdir -p reports
69 report_timing > "./reports/${design_entity}_${target_clk_ns}ns_timing.rpt
       "
70 report_area   > "./reports/${design_entity}_${target_clk_ns}ns_area.rpt"
71 report_power  > "./reports/${design_entity}_${target_clk_ns}ns_power.rpt"
72 print_info "Done: Reports"
73
74 # final cleanup
75 print_info "Done: Synthesis. Cleaning up..."
76 exec rm -rf work
77 exit
```