Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Group 40

Gabriel Ganzer

October 19, 2020

Contents

CHAPTER 1

Introduction

1.1 Architecture

The DLX (DELUXE) is a fully pipelined RISC processor based on the Harvard Architecture, i.e., it relies on two different memories for instructions and data, allowing simultaneous instruction-fetching and data transactions, as shown in Figure 1.1.

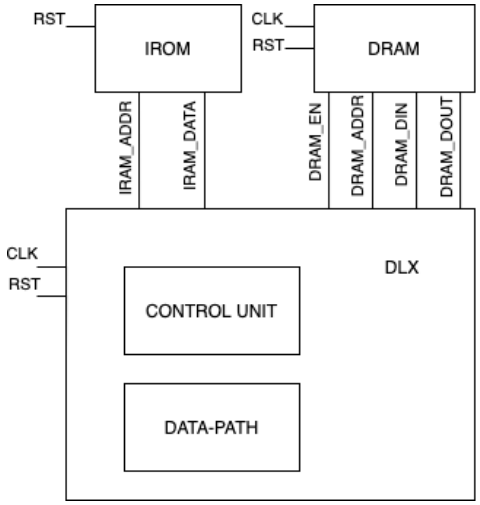

Figure 1.1: DLX block diagram.

The instruction memory is a ROM memory based on a VHDL description provided by Intel FPGA Support Resources. The memory behavior was adjusted to this project testbench by including a procedure that reads the file *"test.asm.mem"*, filling the memory array with its content.

The DRAM was also based on a Dual-Port SDRAM VHDL description provided by Intel FPGA Support Resources, also modified to handle external files.

The top-level entity consists of the DLX microprocessor itself, which in turn instantiates two components, the Control Unit (CU) and Data-path (DP). The CU in particular had to be implemented in one of three different design models: hardwired, fine-state machine, and micro-programmed. Further details on those components would be discussed in the following sections.

1.2 Implementation

The mandatory specifications for this project consisted of a VHDL description of a DLX Basic version able to run a small sub-set of instructions, followed by its synthesis and physical design.

Moreover, some optional specifications were suggested to achieve a DLX Pro version. The features added by the author are listed below.

- **Extended Instruction Set:** addu, addui, jalr, jr, lb, lbu, lhi, lhu, sb, seq, seqi, sgeu, sgeui, sgt, sgti, sgtu, sgtui, slt, slti, sltu, sltui, sra, srai, subu, subui, mult;
- **Optimized ALU:** Pentium4 adder/subtractor, UltraSPARC T2 3-level Shifter, UltraSPARC T2 Logic Unit using 2-level NAND gates, Multi-purpose Comparator, Zero Detector, and a multiplier that combines both Wallace Tree and Booth approaches. Power optimization was achieved by reducing switching activity through state assignment of all blocks;
- **Forwarding Unit:** data hazard control;
- **Clock-Gating:** Register File and Generic Registers gated for power optimization;
- **Scripts:** Scripts were used during design, simulation, and synthesis for automating such processes.

CHAPTER 2

Control Unit

The CU is the component responsible for managing the whole pipeline. Each time the Program Counter (PC) is incremented, a new instruction is fetched from the IROM by reading the address pointed by the PC value. The newly fetched instruction is then stored into the Instruction Register (IR) before entering the CU for decoding.

The hard-wired design style was chosen when describing the CU behavior due to its simplicity and liability. Two Look-up-Tables were used in the combinational logic that asserts a 32-bit Control Word (CW) signal and the ALU opcode. In each stage of the pipeline (FETCH, DECODE, EXECUTE, MEMORY ACCESS, and WRITE BACK) a portion of the CW is delivered to the data-path, as demonstrated in Figure 2.1.

The instruction set can be divided into three categories, register-to-register (R-type), immediate operations (I-type), and branching instructions (J-type). The instruction encoding changes according to the instruction type:

- **R-Type:** 6-bit OPCODE, 5-bit operand A (RS1), 5-bit operand B (RS2), 5-bit destination register (RD), and 11-bit FUNC;
- **I-Type:** 6-bit OPCODE, 5-bit operand A (RS1), 5-bit destination register (RD), and 16-bit immediate;
- **J-Type:** 6-bit OPCODE, 26-bit immediate.

Whenever a new instruction is received it is split into an OPCODE field used by both LUTs, and the FUNC field for determining the ALU operation. An implicit encoding for the ALU opcode signal was chosen as it provides better readability during the coding phase, as well as during testbench. The fragment of code below depicts this kind of signal assignment, extracted from the *000-globals.vhd* file.

```
type aluOp is (nopOp, addOp, subOp, andOp, orOp, xorOp, sllOp, srlOp,
    sraOp, multOp, lhiOp, gtOp, geOp, ltOp, leOp, eqOp, neOp, gtUOp,
    geUOp, ltUOp, leUOp);
```

New instructions are fetched at each clock cycle, entering the pipeline in order. Therefore, the DLX microprocessor can execute up to 5 instructions simultaneously, each allocated to one of the pipeline stages. The control word is shifted by one position at each positive edge of the clock cycle to ensure that the portion of control signals related to a certain stage is delivered to the data-path in the correct order. This behavior is highlighted in Figure 2.1.

Furthermore, three D-Type Flip-Flops store an enable signal coming from the ALU Opcode LUT that signalizes when a branch instruction is fetched, that are then delivered to an OR gate which in turn assign a FLUSH signal. This signal indicates that a jump is entering the pipeline, hence,

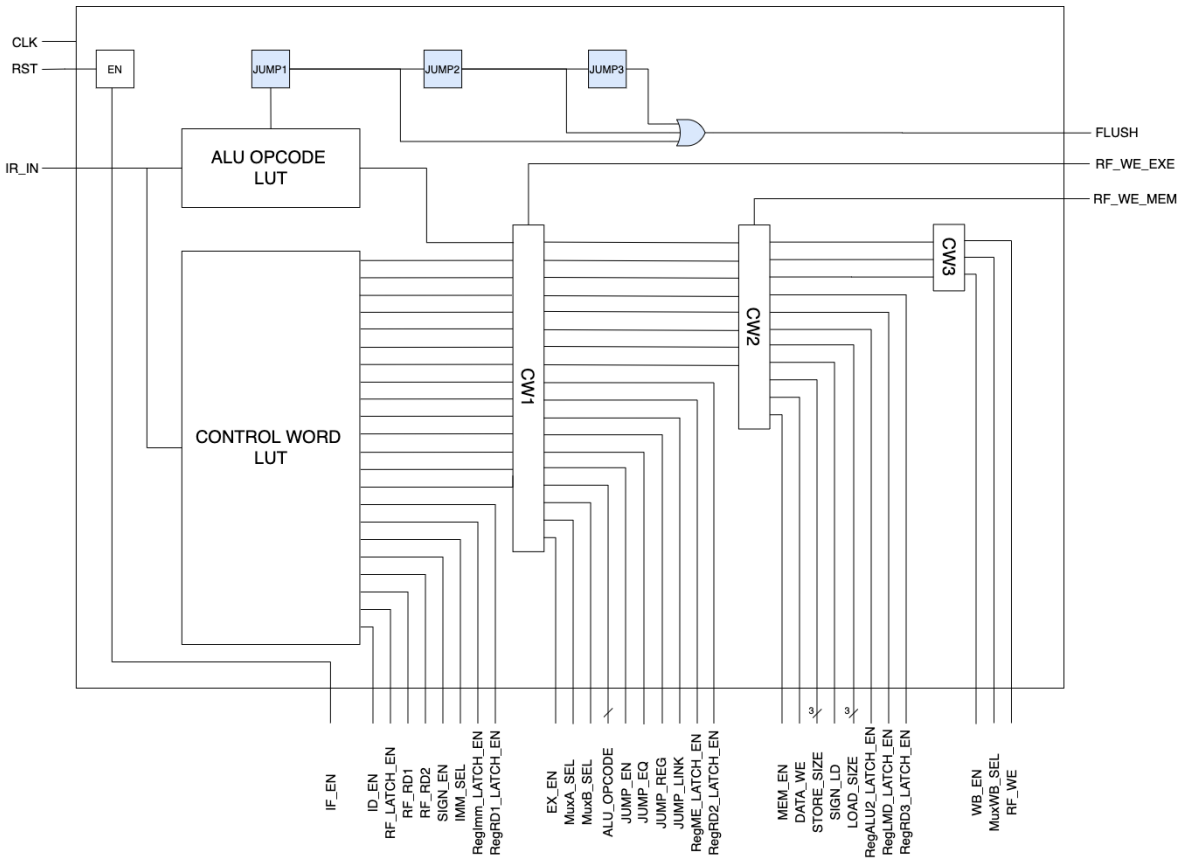

Figure 2.1: Harwired Control Unit.

any instruction that is fetched in the next 2 clock cycles must be discarded. In addition, two signals regarding the write to Register File signal are delivered to the Forwarding Unit, the first coming from the pipe related to the EXECUTE stage and the other from the MEMORY ACCESS pipe. In the following, a brief description of each one of the control signals is provided.

- **IF_EN:** enable signal for the registers involved in the FETCH stage.
- **ID_EN:** enable signal for the registers involved in the DECODE stage that are not gated.
- **RF_LATCH_EN:** enable signal when reading from the Register File.
- **RF_RD1:** RF read enable source register of operand A.
- **RF_RD2:** RF read enable source register of operand B.
- **SIGN_EN:** asserted when instruction implies an operation over signed immediate, the 26-bit immediate is always signed.
- **IMM_SEL:** select signal for the immediate multiplexer, whose assertion value depends on the type of instruction.
- **RegImm_LATCH_EN:** enable signal for gating the Immediate Register.
- **RegRD1_LATCH_EN:** enable signal for gating the Pipeline Destination Register 1.
- **EX_EN:** enable signal for the registers involved in the EXECUTE stage that are not gated.

- **MuxA_SEL:** select signal for choosing between register A and NPC.
- **MuxB_SEL:** select signal for choosing between register B and immediate register.
- **ALU_OPCODE:** ALU operation code.
- **JUMP_EN:** asserted when there is a direct branch.
- **JUMP_EQ:** asserted when branching instruction requires equality comparison.
- **JUMP_REG:** asserted when branching to a register value.
- **JUMP_LINK:** asserted when branching requires saving a return address.
- **RegME_LATCH_EN:** enable signal for gating the Memory Data-input Register.
- **RegRD2_LATCH_EN:** enable signal for gating the Pipeline Destination Register 2.
- **MEM_EN:** enable signal for the DRAM memory.
- **DATA_WE:** DRAM write enable signal.
- **STORE_SIZE:** select signal for choosing between the different lengths of data in STORE instructions.
- **SIGN_LD:** asserted when loading a signed half-word or byte.
- **LOAD_SIZE:** select signal for choosing between the different lengths of data in LOAD instructions.
- **WB_EN:** enable signal when writing to the Register File.
- **MuxWB_SEL:** select signal that determines the source of the data to be written into the RF.
- **RF_WE:** RF write enable signal during WRITE-BACK stage fed to forwarding unit also.

CHAPTER 3

Data-path

The data-path is the most critical unit of a microprocessor as it deals with the general data processing. Figure 3.1 depicts the internal architecture of this component. External ones such as IROM and DRAM were placed inside the data-path in that figure for simplicity. In that figure, the CU signals are highlighted in red while those generated in the Data-path itself are highlighted in blue.

The data-path is organized in five different stages, given as it follows:

- **INSTRUCTION FETCH (IF):** the PC register stores the address corresponding to the instruction that must be fetched from the IROM with its subsequent storage in the IR register. The address to the next instruction is then computed and saved into the NPC register if a branch is not to be performed in the following, storing the ALU result otherwise.
- **INSTRUCTION DECODE (ID):** source and destination addresses are then extracted according to the type of instruction currently executing and the corresponding signals for accessing the register file are asserted. In particular, in the case of an I-type instruction, the 16-bit immediate is extracted and sign-extended to a 32-bit word, with J-type instructions the 26-bit target address is extracted and sign-extended instead. After reading the operands from RF or extracting the immediate values, those are delivered to their corresponding registers for the next stage.
- **EXECUTION (EX):** the ALU performs the specified operation over the provided operands. These values may come from the previous or following stages according to the results obtained in the forwarding unit. Decisions regarding branch conditions also take place in this stage.
- **MEMORY ACCESS (MEM):** the DRAM is accessed if required by the instruction. In the case of a store the memory is accessed in write mode, the address to be written might be either the ALU result from the previous stage or the value currently in the following WRITE-BACK stage according to the forwarding unit decision. The data incoming is the value read from the register source B data-aligned to a word, half-word, or byte. In case of a load the memory is accessed in reading mode and the data outgoing is stored in the LMD register before going through its data-alignment and subsequent write-back to the destination address.
- **WRITE-BACK (WB):** in this last stage either the ALU result or the data read from the memory are eventually written into the register file. In case of a jump and link instruction, such as JAL, a return address consisting on the NPC value is stored into the R31 register instead.

The following section will discuss some details regarding the internal functional units that compose the ALU, as well as a brief explanation of the Forwarding Unit behavior. Comments over the register

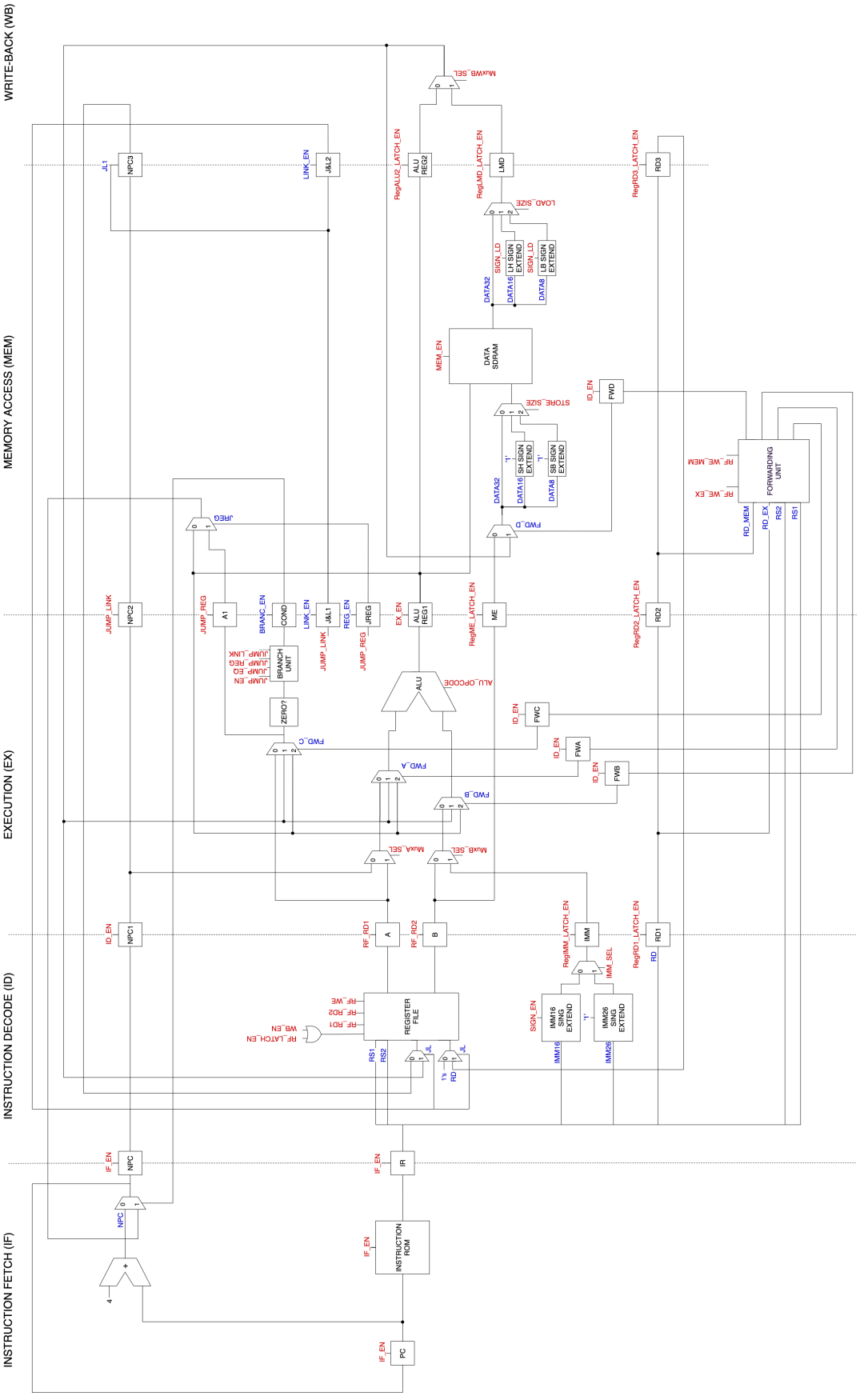

Figure 3.1: Fully pipelined Data-path.

file will be skipped as it was described as a regular 32-bit width and 32-register length dual-port synchronous read/write component. The branch unit will be skipped also as it consists of a simple static branch prediction implementation, i.e., jumping instructions are predicted to be taken always.

3.1 ALU

The ALU depicted in Figure 3.2 is the core of the execution stage able to perform arithmetic and bitwise operations on both sign and unsigned integer numbers. It takes as input the two operands the signal encodes the operation to be performed. Both operands go through a decoder omitted in that figure. This approach aims to reduce the switching activity of those units, hence, decreasing power consumption due to Joule Effect by staging the operands and asserting only the inputs of those functional units involved in a particular operation.

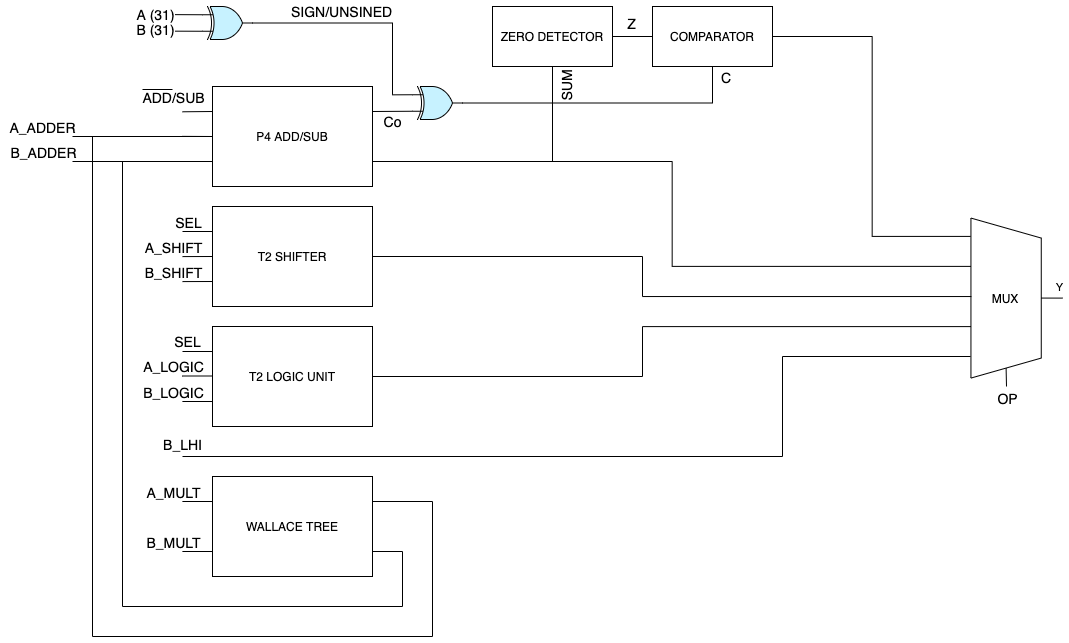

Figure 3.2: ALU block.

The internal functional units added to the project are the following:

- Pentium4 Adder/Subtractor;
- UltraSPARC T2 3-level Shifter;
- UltraSPARC T2 Logic Unit;
- Signed/Unsigned Multi-purpose Comparator;
- Zero Detector;
- Wallace Tree.

In addition to the components mentioned before, the ALU also implements a combinational logic that determines the operation sign, taking both operands MSB and performing an XOR. This signal is then compared again with the carry-out asserted by the adder/subtractor.

The instruction *LHI* was considered as a special case, whose operand B value has its lower 16-bit field concatenated with the higher one and then cleared.

Finally, the Integer Multiplier combines a Wallace Tree Multiplier approach with the shifting strategy from the Booth Multiplier. Its design was inspired in the UltraSPARC T2 multiplier strategy, that reuses the already available adder when summing the final partial multiplications, hence, the result is given by that block instead, reducing area consumption.

3.1.1 Pentium4 Adder/Subtractor

This unit consists of a 32-bit adder/subtractor revised version from the net-lists delivered in the lab session, which now includes a bitwise xor network between operand B and the carry-in signal, asserted in case of subtraction.

3.1.2 UltraSPARC T2 3-level Shifter

The shifter presented during classes was modified to work on 32-bit operands, as it is demonstrated in Figure 3.3. This hardware implements all logic and arithmetic shifts described in the DLX Instruction Set Architecture (ISA).

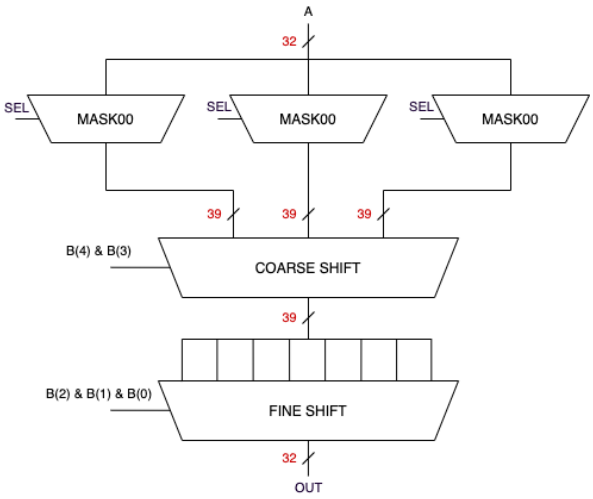

Figure 3.3: T2 3-level Shifter.

It is assumed that operand A represents the operand to be shifted and the lower bits of operand B represent the amount of position to shift, in a range starting from 0 up to 32. The structure is organized as described below:

- **1st-level Mask Generator:** according to the selection signal the first level produces three masks, shifted by 0 (MASK00), 8 (MAKS08) and 16 bits (MASK16), also extending operand A to 39 bits. The selection signal is asserted to:
  - 00: logic shift left, replacing vacant bits with 0's;
  - 01: logic shift right, replacing vacant bits with 0's;
  - 10: arithmetic shift right, replacing vacant bits with MSB.
- **2nd-level Coarse Shift:** one of the masks is selected according to the shift magnitude given by B(4) and B(3);
- **3rd-level Fine Shift:** the final result is produced.

3.1.3 UltraSPARC T2 Logic Unit

The logic unit was based on the UltraSPARC T2 one presented in the lectures, consisting of a 2-level NAND gate network. The first level performs a NAND operation between operands, their respective inverted values and the select signal that determines the final logic operation to be performed, being those: *AND*, **NAND**, **OR**, **NOR**, **XOR**, and *XNOR*. The second level performs a NAND operation between the four outputs of the previous stage, presenting the final result.

3.1.4 Signed/Unsigned Multi-purpose Comparator

The Multi-purpose Comparator presented in the lectures was slightly modified to deal with unsigned operations as well, as depicted in Figure 3.4 and mentioned when discussing the ALU block. The XOR gates and the adder/subtractor block were placed in that figure for simplicity, the comparator itself takes as input the carry signal after the comparison and the output Z given by the Zero Detector unit. It provides hardware support for several operations: $<$, $>$, $\leq$, $\geq$, $=$, and $\neq$.

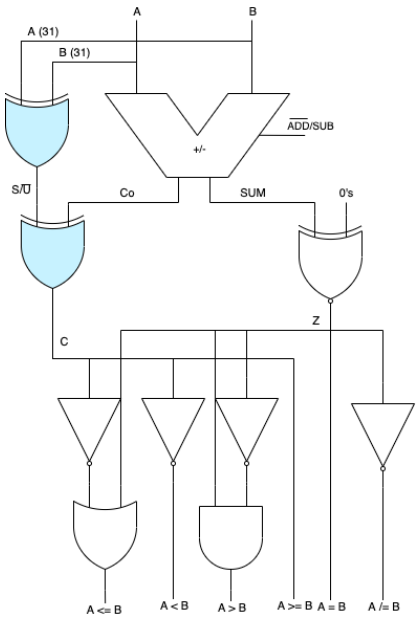

Figure 3.4: Sign/Unsigned Comparator.

3.1.5 Zero Detector

The Zero Detector is a simple XOR gate network that takes as one of the inputs the resulting output from the adder/subtractor and performs a bitwise operation with an array of 0's.

3.1.6 Integer Multiplier

The 32-bit Integer Multiplier takes as input the lower 16-bit field of operands A and B performing a signed integer multiplication. As in a regular Booth Multiplier the operand A is shifted by one position at each partial multiplication and operand B is grouped into sets of 3 bits. whose very first bit is actually '0' by convention.

The principle of Wallace tree multiplication involves by grouping the partial products into sets of 3 as well. The latency is reduced even further in the tree by using Carry Save Adder (CSA) blocks instead of regular Ripple-Carry Adder (RCA). Details of this implementation can be seen in Figure

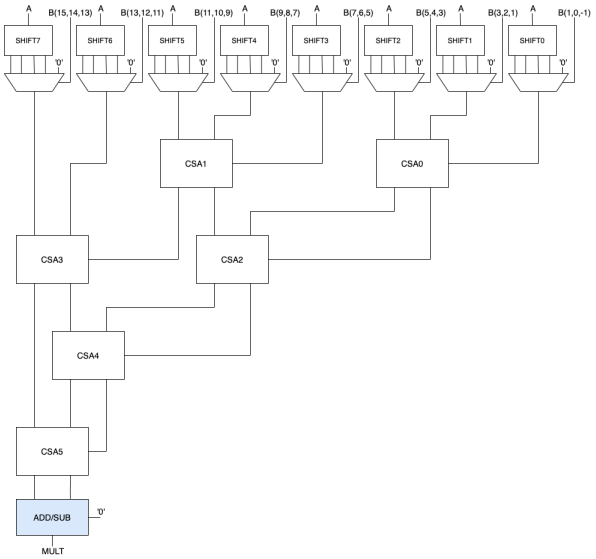

Figure 3.5: Integer Multiplier.

3.5. As mentioned in previous sections the Wallace Tree output is delivered to the adder block when performing the final sum, highlighted in blue in that figure.

Version	Delay $[\eta s]$	Area $[\mu m^2]$	Power $[\mu W]$
ALU Booth Multiplier	5.70	2691.920035	443.8564
ALU Integer Multiplier	2.87	2655.212035	520.8425

Table 3.1: Booth Multiplier versus Wallace Tree combined version.

Table 3.1 compares the metrics obtained with an ALU implementing a regular Booth Multiplier and the proposed design after an unconstrained synthesis. It is clear that the delay is reduced by half with the second implementation, however, the power consumption increased as the workload in the adder gates is also increased. As predicted, there is a small decrease in area consumption from one implementation to the other.

3.2 Forwarding Unit

Pipelined architectures have among some of their drawbacks the occurrence of stalls due to data hazards. When there is a true data dependency among instructions being processed by the pipeline simultaneously, such that one instruction must wait for the completion of the following ones as it depends on their result.

In order to avoid stalls and penalties, parallel components can be used to directly forward some results of instruction to another one before completion. Hence, read-after-Write (RAW) dependencies can be solved efficiently using a technique that allows data to skip the normal execution pipeline and to arrive directly where is needed.

In the DLX case, data is produced mainly in the ALU and DRAM, and fed to, again, the ALU and DRAM, but also the branching unit. Therefore, the Forwarding Unit (FU) checks both addresses and RF write enables signals for the instructions currently at the EXECUTE and MEMORY ACCESS stages to determine if there is a true data dependency, hence, taking a decision of which data must be forwarded.

CHAPTER 4

Synthesis

The synthesis phase started after completing the design and simulation, using **Design Vision**. A preliminary step consisted of synthesizing the DLX-basic version for later comparison with the subsequent DLX-pro synthesis results. The results were obtained using a 65nm ST-Microelectronics library, Table 4.1 reports those results achieved with an unconstrained synthesis of both design versions. The compilation took place using the command *compilation_ultra -gate_clock*, which is best suited for designs that have significantly tight timing constraints, providing better quality of results (QoR) and accurate clock gating. The significant increase in the data arrival delay from one version to another is mainly due to the insertion of the Integer Multiplier.

Version	Delay $[\eta s]$	Area $[\mu m^2]$	Power $[mW]$
DLX-basic	0.12	14518.281	1.164
DLX-pro	0.65	15110.662	12.462

Table 4.1: Comparison between DLX-basic and DLX-pro version.

The results presented in Table 4.2 were obtained with a static timing analysis performed over the DLX-pro version only. The clock signal was initially set to a **0.588** ηs, with the same value set for the maximum delay constraint from all inputs to all outputs. The period was then increased at each iteration, up to **0.769** ηs. Such values where chosen in compliance with available commercial values for clock frequency. Details on the commands used during synthesis are reported in Appendix B, the chart in Figure 4.1 depicts the results obtained with each clock period in a 2D design space exploration.

Frequency $[GHz]$	Delay $[\eta s]$	Slack	Area $[\mu m^2]$	Power $[mW]$
1.7	0.62	-0.08	15135.134	15.092
1.6	0.62	-0.04	15134.868	14.215
1.5	0.68	-0.01	15118.376	13.322
1.4	0.64	0	15113.056	12.463
1.3	0.65	0	15110.662	11.592

Table 4.2: Static timing analysis results.

The simulations with maximum delay lower than **0.7** ηs presented a slack violation. Therefore, only the simulations running with a **0.714** ηs e **0.769** ηs were able to meet all the imposed constraints.

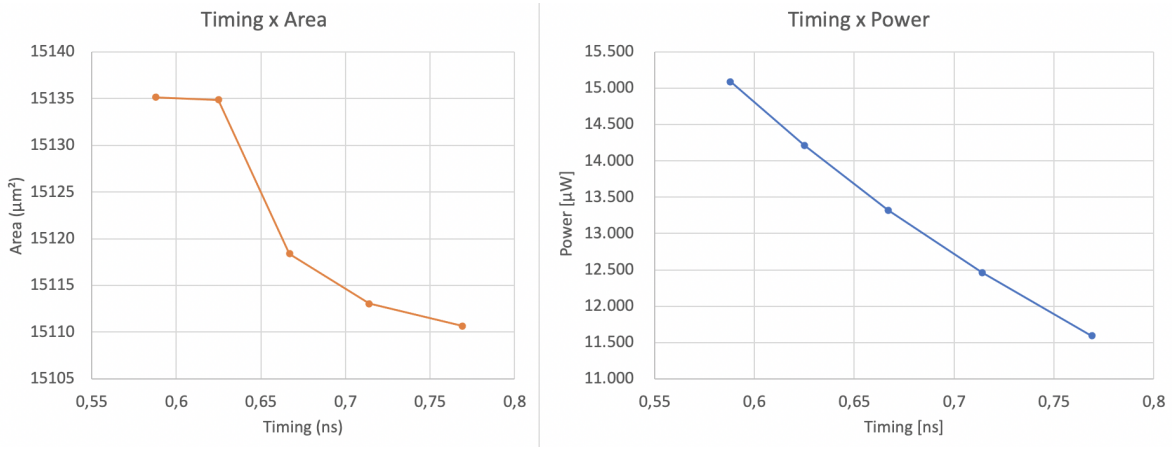

Figure 4.1: Static timing analysis.

Considering a trade-off between all the design aspects, the simulation running at $1.4GHz$ has shown a better performance overall, thus, the one chosen as the final optimization.

The ALU was optimized to reduce circuit switching activity. In addition, all 3-input multiplexers instantiated throughout the data-path were designed using the one-hot encoding technique to reduce power consumption as well. The Table 4.3 compares both design styles regarding the switching power with the **0.714** ηs timing constraint.

Version	Switching Power $[\mu W]$
Non-optimized ALU	636.735
Power-optimized ALU	213.591

Table 4.3: Comparison between switching power achieved before and after optimizing the ALU circuit.

The actual implementation of the clock gating technique consists of simply inserting an AND gate between the clock signal and the enable signal. The tool performs an exhaustive search over the VHDL description file for declarations that correspond to an enable signal, hence, such description must be carefully written. The results obtained are reported bellow.

Number of Clock gating elements: 49
Number of Gated registers: 1475 (96.66%)
Number of Ungated registers : 51 (3.34%)
Total number of registers: 1526

The final version delivered to the physical layout implementation was the DLX-pro with optimized ALU and gated registers able to operate correctly with a **1.4 GHz** clock frequency, accounting setup and hold times. Other parameters are reported below.

Clock period $[\eta s]$	Delay $[\eta s]$	Area $[\mu m^2]$	Power $[mW]$
0.71	0.65	15110.662	12.462

Table 4.4: Final version results.

CHAPTER 5

Physical Layout

The final step of this project requires using the netlists obtained during synthesis to implement the DLX physical layout over the **Cadence Innovus**. The same steps given during the lab session related to this subject were followed and Figure 5.1 illustrates the final schematic obtained after routing. The PWR and GND distribution is achieved through two rings around the die and 5 vertical lines using Metal 9 and Metal 10.

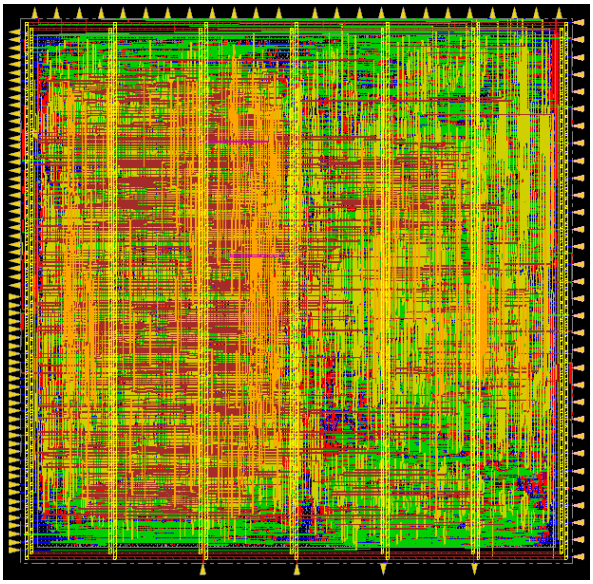

Figure 5.1: Final schematic.

The final gate count, with a gate area of **0.7980** μm^2 is reported in Table 5.1

Level	Gates	Cells	Area [μm^2]
0 Module DLX	19117	8155	15255.4
1 Module DP/ALU0	3432	1884	2738.7
1 Module DP/RF0	10495	4087	8375.0

Table 5.1: Gate count results.

REFERENCES

[1] N. Anand, G. Joseph, and S. S. Oommen. "Performance analysis and implementation of clock gating techniques for low power applications". In: *2014 International Conference on Science Engineering and Management Research (ICSEMR)*. 2014, pp. 1–4.

[2] J. Balfour, R. Harting, and W. Dally. "Operand Registers and Explicit Operand Forwarding". In: *IEEE Computer Architecture Letters* 8.2 (2009), pp. 60–63.

[3] *Design Compiler® User Guide*. F-2011.09-SP2. Synopsys, Inc. Dec. 2011.

[4] Abhishek Kumar et al. "Design of SDRAM Memory Controller using VHDL". In: (Jan. 2010).

[5] Hadeel Mahmood and Safaa Omran. "Pipelined MIPS processor with cache controller using VHDL implementation for educational purposes". In: (Dec. 2014), pp. 82–87. DOI: 10.1109/ICECCPCE.2013.6998739.

[6] *OpenSPARC^{TM} T2 Core Microarchitecture Specification*. Revision 05. Sun Microsystems, Inc. July 2007.

[7] Kirat Singh and Shivani Parmar. "Vhdl Implementation of A Mips-32 Pipeline Processor". In: (Jan. 2012). DOI: 10.5281/zenodo.33247.

[8] S. B. Wijeratne et al. "A 9-GHz 65-nm Intel® Pentium 4 Processor Integer Execution Unit". In: *IEEE Journal of Solid-State Circuits* 42.1 (2007), pp. 26–37.

APPENDIX A

ModelSim Simulation Script

```
#### Compile ####

vcom {000-globals.vhd}
vcom {01-fa.vhd}
vcom {01-generic_mux21.vhd}
vcom {01-generic_mux31.vhd}
vcom {01-generic_rca.vhd}
vcom {01-generic_csa.vhd}
vcom {01-generic_register.vhd}
vcom {01-ffd.vhd}
vcom {01-ivx.vhd}
vcom {01-nand2.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.a-adder.core/a.b.a.a.a-
    pg_network.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.a-adder.core/a.b.a.a.b-
    g_block.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.a-adder.core/a.b.a.a.c-
    pg_block.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.a-adder.core/a.b.a.a.d-
    carry_select_block.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.a-adder.core/a.b.a.a.e-
    sum_generator.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.a-adder.core/a.b.a.a.f-
    sparse_tree.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.b-shifter.core/a.b.a.b.a-
    mask_generator.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.b-shifter.core/a.b.a.b.b-
    coarse_shift.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.b-shifter.core/a.b.a.b.c-
    fine_shift.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.c-logic.core/a.b.a.c.a-nand3
    .vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.c-logic.core/a.b.a.c.b-nand4
    .vhd}
```

```
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.f-multiplier.core/a.b.a.f.a-
    shifter_mult.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.f-multiplier.core/a.b.a.f.b-
    mux58_mult.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.a-adder.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.b-shifter.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.c-logic.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.d-comparator.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.e-zero_detector.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.f-booth_multiplier_v1.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.core/a.b.a.f-wallace_tree.vhd}
vcom {a.b-DataPath.core/a.b.a-ALU.vhd}
vcom {a.b-DataPath.core/a.b.b-register_file.vhd}
vcom {a.b-DataPath.core/a.b.c-sign_extend.vhd}
vcom {a.b-DataPath.core/a.b.d-branch_unit.vhd}
vcom {a.b-DataPath.core/a.b.e-forwarding_unit.vhd}
vcom {a.a-CU_HW.vhd}
vcom {a.b-DataPath.vhd}
vcom {a-DLX.vhd}
vcom {b-IROM.vhd}
vcom {c-DRAM.vhd}
vcom {test_bench/TB_DLX.vhd}

# Start simulation
vsim -t 10ps work.dlx_tb(testbench)

#### ControlUnit ####
add wave -group ControlUnit sim:/dlx_tb/uut/cu/*

#### DataPath ####
add wave -group DataPath -group PC sim:/dlx_tb/uut/pc/*
add wave -group DataPath -group IR sim:/dlx_tb/uut/ir/*
add wave -group DataPath -group MuxNPC sim:/dlx_tb/uut/dp/muxnpc/*
add wave -group DataPath -group RegNPC sim:/dlx_tb/uut/dp/regnpc/*
add wave -group DataPath -group MuxRFDATA sim:/dlx_tb/uut/dp/muxrfdata/*
add wave -group DataPath -group MuxRADDR sim:/dlx_tb/uut/dp/muxrfaddr/*
add wave -group DataPath -group RF sim:/dlx_tb/uut/dp/rf0/*
add wave -group DataPath -group SignExtIMM16 sim:/dlx_tb/uut/dp/
    signextimm16/*
add wave -group DataPath -group SignExtIMM26 sim:/dlx_tb/uut/dp/
    signextimm26/*
add wave -group DataPath -group MuxIMM sim:/dlx_tb/uut/dp/muximm/*
add wave -group DataPath -group RegA sim:/dlx_tb/uut/dp/rega/*
add wave -group DataPath -group RegB sim:/dlx_tb/uut/dp/regb/*
add wave -group DataPath -group RegIMM sim:/dlx_tb/uut/dp/regimm/*
add wave -group DataPath -group RegRD1 sim:/dlx_tb/uut/dp/regrd1/*
add wave -group DataPath -group RegNPC1 sim:/dlx_tb/uut/dp/regnpc1/*
add wave -group DataPath -group MuxA sim:/dlx_tb/uut/dp/muxa/*
add wave -group DataPath -group MuxB sim:/dlx_tb/uut/dp/muxb/*
```

```
add wave -group DataPath -group FwdA sim:/dlx_tb/uut/dp/muxfwda/*
add wave -group DataPath -group FwdB sim:/dlx_tb/uut/dp/muxfwdb/*
add wave -group DataPath -group FwdC sim:/dlx_tb/uut/dp/muxfwdc/*
add wave -group DataPath -group ALU0 sim:/dlx_tb/uut/dp/alu0/*
add wave -group DataPath -group ZERO sim:/dlx_tb/uut/dp/zero/*
add wave -group DataPath -group BU0 sim:/dlx_tb/uut/dp/bu0/*
add wave -group DataPath -group RegA1 sim:/dlx_tb/uut/dp/rega1/*
add wave -group DataPath -group FFBranch sim:/dlx_tb/uut/dp/ffdbranch/*
add wave -group DataPath -group FFJL1 sim:/dlx_tb/uut/dp/ffdjl1/*
add wave -group DataPath -group FFJREG sim:/dlx_tb/uut/dp/ffdjreg/*
add wave -group DataPath -group RegNPC2 sim:/dlx_tb/uut/dp/regnpc2/*
add wave -group DataPath -group RegALU1 sim:/dlx_tb/uut/dp/regalu1/*
add wave -group DataPath -group RegME sim:/dlx_tb/uut/dp/regme/*
add wave -group DataPath -group RegRD2 sim:/dlx_tb/uut/dp/regrd2/*
add wave -group DataPath -group MuxJR sim:/dlx_tb/uut/dp/muxjr/*
add wave -group DataPath -group FwdD sim:/dlx_tb/uut/dp/muxmem/*
add wave -group DataPath -group SignExtSH sim:/dlx_tb/uut/dp/signextsh/*
add wave -group DataPath -group SignExtSB sim:/dlx_tb/uut/dp/signextsb/*
add wave -group DataPath -group MuxSTORE sim:/dlx_tb/uut/dp/muxstore/*
add wave -group DataPath -group SignExtLH sim:/dlx_tb/uut/dp/signextlh/*
add wave -group DataPath -group SignExtLB sim:/dlx_tb/uut/dp/signextlb/*
add wave -group DataPath -group MuxLOAD sim:/dlx_tb/uut/dp/muxload/*
add wave -group DataPath -group RegALU2 sim:/dlx_tb/uut/dp/regalu2/*
add wave -group DataPath -group RegLMD sim:/dlx_tb/uut/dp/reglmd/*
add wave -group DataPath -group RegRD3 sim:/dlx_tb/uut/dp/regrd3/*
add wave -group DataPath -group FFJL2 sim:/dlx_tb/uut/dp/ffdjl2/*
add wave -group DataPath -group RegNPC3 sim:/dlx_tb/uut/dp/regnpc3/*
add wave -group DataPath -group MuxWB sim:/dlx_tb/uut/dp/muxwb/*
add wave -group DataPath -group FU0 sim:/dlx_tb/uut/dp/fu0/*

#### DRAM ####
add wave -group DRAM sim:/dlx_tb/dram/*

# Set unsigned as DEFAULT radix
radix -hex
# Run simulation
run 58 ns
# Print postscript waveform
#write wave p4adder.ps -start 0 -end 230000 -perpage 230000
```

APPENDIX B

Design Compiler Synthesis Script

```
#####################################################################
exec mkdir -p work
exec mkdir -p report
exec mkdir -p netlist
analyze -library WORK -format vhdl {000-globals.vhd}
analyze -library WORK -format vhdl {01-fa.vhd}
analyze -library WORK -format vhdl {01-generic_mux21.vhd}
analyze -library WORK -format vhdl {01-generic_mux31.vhd}
analyze -library WORK -format vhdl {01-generic_rca.vhd}
analyze -library WORK -format vhdl {01-generic_csa.vhd}
analyze -library WORK -format vhdl {01-generic_register.vhd}
analyze -library WORK -format vhdl {01-ffd.vhd}
analyze -library WORK -format vhdl {01-ivx.vhd}
analyze -library WORK -format vhdl {01-nand2.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.a-adder.core/a.b.a.a.a-pg_network.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.a-adder.core/a.b.a.a.b-g_block.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.a-adder.core/a.b.a.a.c-pg_block.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.a-adder.core/a.b.a.a.d-carry_select_block.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.a-adder.core/a.b.a.a.e-sum_generator.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.a-adder.core/a.b.a.a.f-sparse_tree.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.b-shifter.core/a.b.a.b.a-mask_generator.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.b-shifter.core/a.b.a.b.b-coarse_shift.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.b-shifter.core/a.b.a.b.c-fine_shift.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.c-logic.core/a.b.a.c.a-nand3.vhd}
```

```
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.c-logic.core/a.b.a.c.b-nand4.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.f-multiplier.core/a.b.a.f.a-shifter_mult.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.f-multiplier.core/a.b.a.f.b-mux58_mult.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.a-adder.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.b-shifter.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.c-logic.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.d-comparator.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.e-zero_detector.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.core/a.b.
    a.f-wallace_tree.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.a-ALU.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.b-register_file
    .vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.c-sign_extend.
    vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.d-branch_unit.
    vhd}
analyze -library WORK -format vhdl {a.b-DataPath.core/a.b.e-
    forwarding_unit.vhd}
analyze -library WORK -format vhdl {a.a-CU_HW.vhd}
analyze -library WORK -format vhdl {a.b-DataPath.vhd}
analyze -library WORK -format vhdl {a-DLX.vhd}

# Elaborate
set_wire_load_model -name 5K_hvratio_1_4
elaborate DLX -architecture STRUCTURAL -library WORK -parameters "WIDTH =
    32"

# Characterize ALU
characterize  {DP/ALU0}
current_design ALU_WIDTH32_RADIX4_OPCODE6
compile_ultra
report_area > report/ALU_AREA.txt
report_timing > report/ALU_TIMING.txt
report_power > report/ALU_POWER.txt
set_max_delay 0.74 -from [all_inputs] -to [all_outputs]
compile_ultra
report_area > report/ALU_AREA_OPT.txt
report_timing > report/ALU_TIMING_OPT.txt
report_power > report/ALU_POWER_OPT.txt
current_design DLX_WIDTH32
```

```
# Clock gating
set_clock_gating_style -minimum_bitwidth 1 -max_fanout 1024 -
    positive_edge_logic {latch and} -control_point before
compile_ultra -gate_clock
current_design DLX_WIDTH32
report_clock_gating > report/REGS_GATED.txt
report_area > report/DLX_AREA.txt
report_timing > report/DLX_TIMING.txt
report_power > report/DLX_POWER.txt

# Static Timing Analysis
#0.588 cycle constraint
create_clock -name "CLK" -period 0.588 CLK
set_max_delay 0.588 -from [all_inputs] -to [all_outputs]
compile_ultra -incremental
report_area > report/DLX_AREA_588.txt
report_timing > report/DLX_TIMING_588.txt
report_power > report/DLX_POWER_588.txt
#0.625 cycle constraint
create_clock -name "CLK" -period 0.625 CLK
set_max_delay 0.625 -from [all_inputs] -to [all_outputs]
compile_ultra -incremental
report_area > report/DLX_AREA_625.txt
report_timing > report/DLX_TIMING_625.txt
report_power > report/DLX_POWER_625.txt
#0.666 cycle constraint
create_clock -name "CLK" -period 0.667 CLK
set_max_delay 0.667 -from [all_inputs] -to [all_outputs]
compile_ultra -incremental
report_area > report/DLX_AREA_667.txt
report_timing > report/DLX_TIMING_667.txt
report_power > report/DLX_POWER_667.txt
#0.714 cycle constraint
create_clock -name "CLK" -period 0.714 CLK
set_max_delay 0.714 -from [all_inputs] -to [all_outputs]
compile_ultra -incremental
report_area > report/DLX_AREA_714.txt
report_timing > report/DLX_TIMING_714.txt
report_power > report/DLX_POWER_714.txt
#0.714 cycle constraint
create_clock -name "CLK" -period 0.769 CLK
set_max_delay 0.769 -from [all_inputs] -to [all_outputs]
compile_ultra -incremental
report_area > report/DLX_AREA_0769.txt
report_timing > report/DLX_TIMING_769.txt
report_power > report/DLX_POWER_769.txt

create_clock -name "CLK" -period 0.714 CLK
```

```
set_max_delay 0.714 -from [all_inputs] -to [all_outputs]
compile_ultra -incremental

current_design DLX_WIDTH32
report_clock_gating > report/REG_FINAL.txt
report_area > report/DLX_AREA_FINAL.txt
report_timing > report/DLX_TIMING_FINAL.txt
report_power > report/DLX_POWER_FINAL.txt

write -hierarchy -format verilog -output netlist/dlx.v
write_sdc netlist/dlx.sdc
```