Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development

Final Project Report

Master degree in Electronics Engineering

Authors: group_19

Castagno Fabio - s242018
Rossi Luca - s239951

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

October 30, 2017

Contents

CHAPTER 1

Summary

The project consists in the design, simulation and synthesis of a 5 stage pipelined DLX processor core in VHDL. After the basic functionalities it implements also: an extended ISA and an advanced ALU. Everything was optimized for delay and power efficiency. Physically implementation with place, route and clock tree synthesis. Our choice is to make the PRO version of the DLX, with some features added as:

- expanded ISA
- data-path optimization
- hazard control (some particular cases)
- static branch prediction

CHAPTER 2

DLX

The DLX has a RISC architecture (Reduced Set Instruction Computer). The block diagram is shown in figure 2.1. It uses 32 integer general purpose registers and 32 floating point registers. In this typical architecture there are two different memories, one for the instructions and one for data.

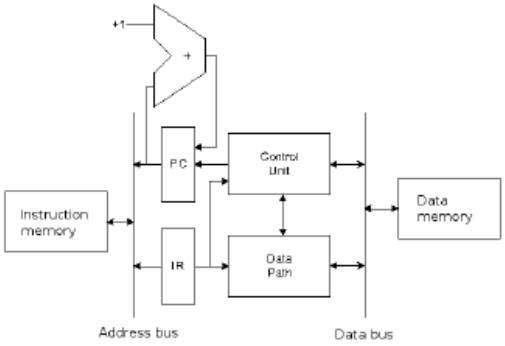

Figure 2.1: General schematic of a microprocessor based system

It uses only 2 addressing modes:

- **Immediate**: the content of a register is added to a value and the result is stored back
- **Displacement**: the content of a register is added to the content of memory at one specific address

The main structure of the system is composed by:

- **Program Counter**: register that always points the next instruction cell memory
- **Instruction register**: register that contains the instruction to be executed
- **Control Unit**: coordinates all the necessary actions in order to execute the stream of instructions
- **Data Path**: collection of units that are able to perform the data processing operations, it typically contains adders, multiplier, registers ecc.

The DLX datapath structure is shown in the following image 2.2. From this starting point we make our customized version.

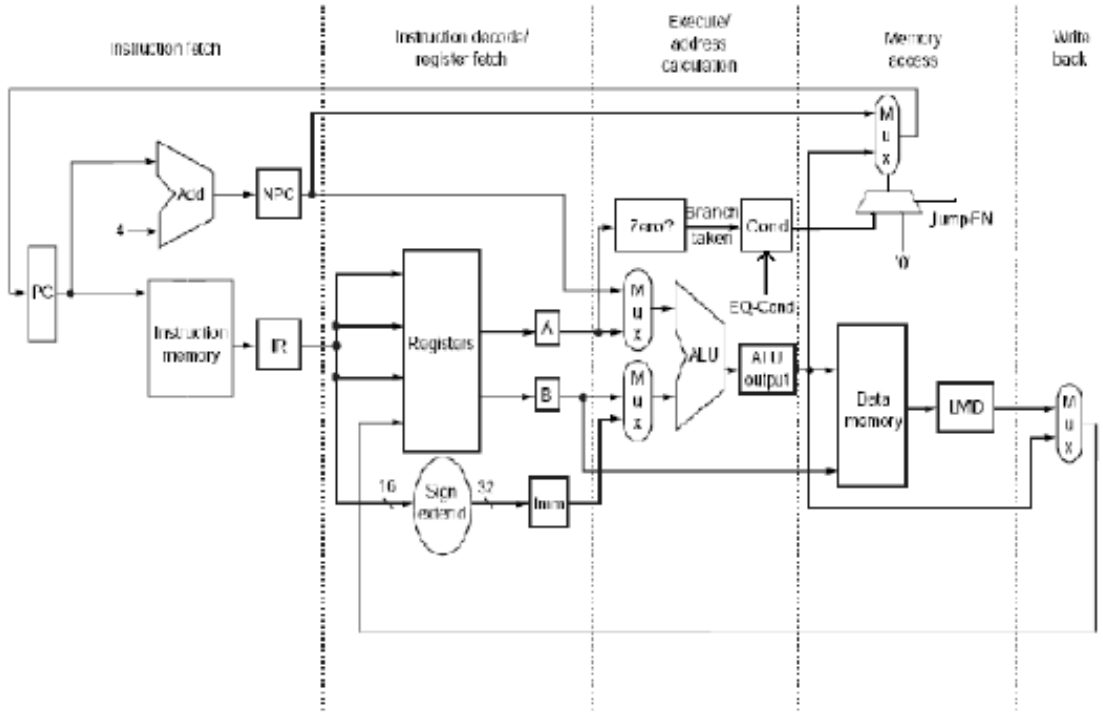

Figure 2.2: DLX datapath

There are 3 types of instructions (I - R - J) all 32 bits width with a 6-bit primary opcode.

- **I-type**: normally they are load and store instructions, operations with immediates or conditional branches
- **R-type**: they are ALU register to register operations
- **J-type**: they are jump, trap or return from exception

Every DLX instruction is implemented in at most five clock cycles:

IF ID EX MEM WB

- **Instruction fetch (IF)** cycle: Sends out the PC and fetches the instructions from memory into the instruction register (IR) and increments the PC by 4 to address the next one.
- **Instruction decode/register fetch (ID) cycle**: Decodes the instruction and accesses the register file (RF) to read the registers. Decoding is done in parallel with reading registers, which is possible because these fields are at a fixed location in the DLX instruction format. This technique is known as fixed-field decoding.
- **Execution/effective address cycle (EX)**: The ALU operates on the operands (A or B/Imm or both) prepared in the previous cycle and the result is stored in a register.
- **Memory access/branch completion (MEM) cycle**: Accesses memory if needed. If instruction is a load, the data returned by the memory is placed in a register. Otherwise if it is a store, the data from the register is written into memory. In both cases the address used is the one computed in the prior cycle.
- **Write-back(WB) cycle**: Writes the result into the register file, whether it comes from the memory system or from ALU.

The processor is fully pipelined, this approach exploits parallelism and is based on concurrently perform different phases of processing. The general assumption is that the phases are independent between different operations and can be overlapped. If this condition is not respected the processor stalls the downstream phases. So multiple operations can be processes simultaneously for every different phase. The following timing diagram shows how it works:

Instruction cycle	1	2	3	4	5	6	7	8
i	IF	ID	EX	MEM	WB			
i+1		IF	ID	EX	MEM	WB		
i+2			IF	ID	EX	MEM	WB	
i+3				IF	ID	EX	MEM	WB
i+4					IF	ID	EX	MEM

Figure 2.3: Pipeline timing

With this approach we must pay attention to some problems that can occur, for example hazards. There are some techniques that try to prevent hazards.

CHAPTER 3

Control Unit

The top entity of our DLX is composed of two macroblocks, the hardwired CU and the datapath. The main reason that brings us to choose an hardwired Control Unit implementation instead of a Microprogram one is the high speed of this kind of architecture. The main drawback is the lower flexibility and possibility to manage complex instructions. The implementation is made through use of combinational logic units, featuring a finite number of gates that are able to generate the results that we need according to the instruction that we provide. It can be seen as a big look-up table. The CU receives the instruction word and two signals, clock and reset. The aim of the CU is to take these informations, elaborate them and produce the right control signals to the datapath. In order to delay the execution of each stage (pipeline), flip-flops are added.

3.1 Control Word

The control word size is 42 bits and each instruction has its own unique configuration. In Appendix A are reported all the instructions that our DLX is able to handle. In the following is shown the exact composition of the control word and the function of every single bit that compose it.

Listing 3.1: Control Word

```
-- FETCH stage signals
        FS_Rst                  <= CWSF(41); -- reset to fetch stage
        FATCH_En                <= CWSF(40); -- enables the fetch stage registers
        FATCHRstMux21_Sel       <= CWSF(39); -- 1 reset of the stage if branch or jump 0
            FS_Rst
        PCMux41_Sel             <= CWSF(38 downto 37); --1 1 Beqz 10 Bneqz 01 Jump 00 '0'
        IRAM_Rst                <= CWSF(36); -- reset of IRAM

        -- DECODE stage signals
        DECODE_En               <= CWSD(35); -- eanbles the decode stage registers
        DECODE_Rst              <= CWSD(34); -- reset the decode stage registers
        RF_Rst                  <= CWSD(33); -- reset the register file
        RF_RD1                  <= CWSD(32); -- read on port 1 of register file
        RF_RD2                  <= CWSD(31); -- read on port 2 of register file
        R1Mux21A_Sel            <= CWSD(30); -- (1 AddrR1=R0 0 AddrR1=AddR1)
        R2Mux21B_Sel            <= CWSD(29); -- (1 AddrR2=R0 0 AddrR2=AddR2)
        RWMux41WR_Sel           <= CWSD(28 downto 27);-- (11 AddrWR=R31 10 AddrWR=R0 01
            AddrWR=AddrR2 00 AddrWR=AddrR3)
        ImmMux21_Sel            <= CWSD(26); --(1 Imm26 0 Imm16)

        -- EXECUTE stage signals
        EXECUTE_En              <= CWSE(25); -- enables the execute stage registers
        EXECUTE_Rst             <= CWSE(24); -- resets the execute stage registers
        OPBMux41_Sel            <= CWSE(23 downto 22); -- (11=4 10=0 01=Imm 00=B)
        OPAMux21_Sel            <= CWSE(21); -- (1=PC_ret 0=A)
        ALU_Sel                 <= CWSE(20 downto 17); -- selection signals for the ALU
            operation
        ALU_Unsign              <= CWSE(16); -- execute an unsigned operation in the ALU
        ALU_Arith_logN          <= CWSE(15); -- (1 artithmetic shift 0 logical shift)
        StatusMux81_Sel         <= CWSE(14 downto 12); -- select the flag of the ALU that we
            want to propagate

        -- MEMORY stage signals
        MEMORY_En               <= CWSM(11); -- enables the memory stage registers
        MEMORY_Rst              <= CWSM(10); -- reset the memory stage registers
        DATAMEM_En              <= CWSM(9); -- enables the data memory
        DATAMEM_Rst             <= CWSM(8); -- reset the data memory
        DATAMEM_Read_Wrn        <= CWSM(7); -- (1 read from the memory o write to the memory)
        DATAMEM_Word            <= CWSM(6); -- parallelism of the memory operation is 32 bit
        DATAMEM_HalfWord        <= CWSM(5); -- parallelism of the memory operation is 16 bit
        DATAMEM_Byte            <= CWSM(4); -- parallelism of the memory operation is 8 bit
        DATAMEM_Unsign          <= CWSM(3); -- the number that we want store or read is
            unsigned
```

```
39
        -- WRITE BACK stage signals
41      RF_WR                   <= CWSWB(2);
        WBMux41_Sel             <= CWSWB(1 downto 0); -- (11=Memory_data 10=ALU_data 01=
            Status_data 00='0')
```

During the rising edge of the clock, the control word for the current instruction is determined through a process, while on the falling edge, it is sent to the chain of registers that performs the pipeline. The three different kind of instructions will be treated separately but the way we handle them is similar.

3.2 R-type instruction

They are ALU register to register operations. When the CU detects an instruction that belongs to an R-type, it immediately goes to watch the FUNCTION field. At this point, according to its internal LUT, it sets properly the Control Word and the ALU op-code.
In order to have a better view on modelsim, we also set a variable IR, that holds the name of the instruction that is processed. A little abstract of the code is reported here:

Listing 3.2: R TYPE operation

```
case conv_integer(unsigned(IR_OPCODE)) is
2       when conv_integer(unsigned(RTYPE)) =>
                case conv_integer(unsigned(IR_FUNC)) is -- if RTYPE instruction then watch
                    function code
4                       when conv_integer(unsigned(RTYPE_ADD)) =>
                                CW <= CW_RTYPE_ADD;
6                               aluOpCod <= ADDop; -- only for a clear view on modelsim
                                IR <= addr; -- only for a clear view on modelsim
8                       when conv_integer(unsigned(RTYPE_AND)) =>
                                CW <= CW_RTYPE_AND;
10                              aluOpCod <= ANDop;
                                IR <= andr;
12                      when conv_integer(unsigned(RTYPE_OR)) =>
                                CW <= CW_RTYPE_OR;
14                              aluOpCod <= ORop;
                                IR <= orr;
16  . . .
```

3.3 J-type instruction

They are normal jump instructions. In our case we have both absolute and relative jump instructions. The procedure is the same that we use for the R-type instructions. The four jump types that the DLX can handle are reported here.

Listing 3.3: J-TYPE operation

```
when conv_integer(unsigned(JTYPE_JABS)) =>
2       CW <= CW_JTYPE_JABS;
        Jump <= '1';
4       Jump_In <= '1';
        aluOpCod <= ADDop;
6       IR <= j;
when conv_integer(unsigned(JTYPE_JAL)) =>
8       CW <= CW_JTYPE_JAL;
        Jump <= '1';
10      Jump_In <= '1';
        aluOpCod <= ADDop;
12      IR <= jal;
when conv_integer(unsigned(JTYPE_JR)) =>
14      CW <= CW_JTYPE_JR;
        BranchInst <= '1';
16      JumpR_In <= '1';
        JumpRInst <= '1';
18      aluOpCod <= ADDop;
        IR <= jr;
20 when conv_integer(unsigned(JTYPE_JALR)) =>
        CW <= CW_JTYPE_JALR;
22      BranchInst <= '1';
        JumpR_In <= '1';
24      JumpRInst <= '1';
        aluOpCod <= ADDop;
26      IR <= jalr;
```

3.4 I-type instruction

Normally they are load and store instructions, operations with immediates or conditional branches. The handle of the branches instructions is discussed in a dedicated section. The procedure is the same of the previous two cases as we can see here below.

Listing 3.4: I TYPE operation

```
when conv_integer(unsigned(ITYPE_ADDI)) =>
        CW <= CW_ITYPE_ADDI;
        aluOpCod <= ADDop;
        IR <= addi;
when conv_integer(unsigned(ITYPE_ANDI)) =>
        CW <= CW_ITYPE_ANDI;
        aluOpCod <= ANDop;
        IR <= andi;
when conv_integer(unsigned(ITYPE_BEQZ)) =>
        BranchZ_In <= '1';
        BranchInst <= '1';
        CW <= CW_ITYPE_BEQZ;
        aluOpCod <= ADDop;
        IR <= beqz;
. . .
```

3.5 Branch and Jump management

If the instruction read from the IRAM is either a branch or a jump, an internal signal is risen in the CU according to the different kind of instructions.

Listing 3.5: Branch detection

```
when conv_integer(unsigned(ITYPE_BEQZ)) =>
        BranchZ_In <= '1';
        BranchInst <= '1';
        CW <= CW_ITYPE_BEQZ;
        aluOpCod <= ADDop;
        IR <= beqz;
when conv_integer(unsigned(ITYPE_BNEQZ)) =>
        BranchNZ_In <= '1';
        BranchInst <= '1';
        CW <= CW_ITYPE_BNEQZ;
        aluOpCod <= ADDop;
        IR <= bnez;
```

If the previously one was a **J, JAL**, the CU, through the jump signal, will change the input address of the IRAM, so the next instruction will not be the one located at the next sequential address, but the one situated at the address determined thanks to the **BrancAdd**. In this way we can fetch the right instruction without loosing 1 clock cycle.
Instead for the other kind of branch and jump, the procedure is more complicated because we have to read the data from RF without affecting the execution of the previous instructions. In order to do that we have to wait until the decode phase in which the access to RF is performed. At this point for the branch we are able to determine if it will happens or not. Due to this delay, in case of taken branch, we will always execute a wrong instruction fetch and consequently we will loose 1 clock cycle. The same thing happens with **JR, JALR** because we have to read from the RF the branch address. To avoid that a wrong fetched instruction affects the state of the processor, a FLUSH operation is performed on the pipeline through a proper signal.

Listing 3.6: FLUSH

```
if (Branch_In='1') then -- eliminate the effect of CW if branch occurs
        CWSF <= (others => '0');
        end if;
        Branch_In <= '0';
        BranchInst <= '0';
        Jump <= '0';
        JumpRInst <= '0';
        Jump_In <= '0';
        JumpR_In <= '0';
        StoreInst <= '0';
        LHIInst <= '0';
```

There are also two dedicated bits of the control word whose aim is to determine if the branch will occur or not and, based on this decision, also correct the DLX program counter.

Listing 3.7: Jump and branch affection

```
--CW(37) and CW(38) are two bit of selection of a multiplexer that determine if the next
    instruction will be affected by a jump or a branch
if (JumpR_In='1') then
        CW(38) <= '0';
        CW(37) <= '1';
        JumpR_In <= '0';
        JumpRInst <= '0';
        BranchInst <= '0';
end if;
if (BranchZ_In='1') then
        CW(38) <= '1';
        CW(37) <= '0';
        BranchZ_In <= '0';
        BranchInst <= '0';
end if;
        if (BranchNZ_In='1') then
        CW(38) <= '1';
        CW(37) <= '1';
        BranchNZ_In <= '0';
        BranchInst <= '0';
end if;
```

We decided to manage these operations in this way because is quiet simple and intuitive and also because is easier to test. 42 bits for the control word are a lot, but allow us to have under constant control the general situation during the test phase.

CHAPTER 4

Data Path

The full structure of the datapath is shown in Appendix B, here we will analyze how it is made stage by stage explaining our choices. All the control signals are generated inside the CU and then send to the datapath of DLX. Every stage has its own enable and reset signal, except for the write-back one.

4.1 Fetch

At the beginning of this stage the program counter is uploaded and so the address of the next instruction is available. Then there is a logic that is able to understand if there was a branch or not and according to this it gives the right address to the **PCadd** and the **IRAM**.
The branch detection occurs in this phase, based on a static prediction that is always a not-taken type. If there is a misprediction, then the PC is updated to the right value. The IRAM reads the address and produces at the output the stream of 32 bits of the current instruction. Then the new one sent to a register waiting to be processed by the next stage.
In this phase also the program counter is updated. The PCadd always adds 4 because we work with 32 bits instructions with memory width of 8 bits.

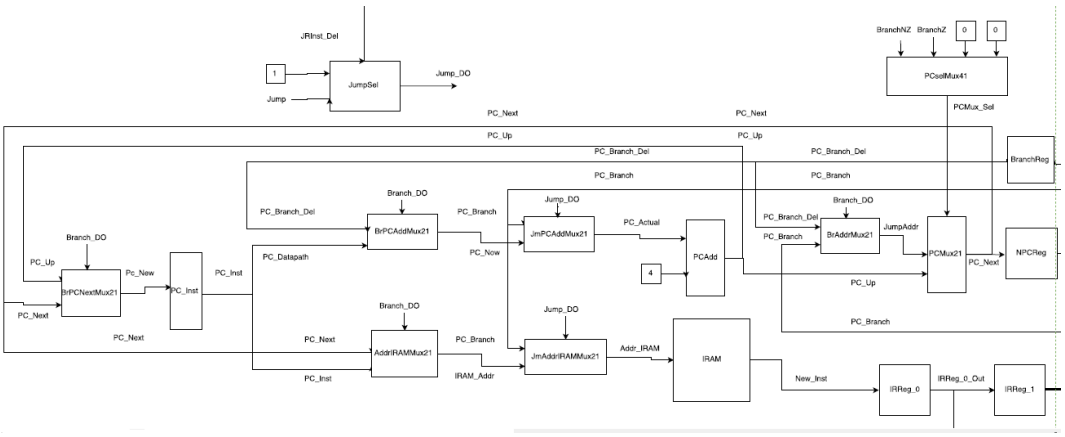

Figure 4.1: Fetch

4.2 Decode

In this phase the opcode fetched from the memory is splitted into 5 different fields:

- **RF_AddrR1** read address for port 1 or RF
- **RF_AddrR2** read address for port 2 or RF
- **RF_AddrR3** write port RF address
- **ImmediateJ** target jump address
- **ImmediateI** immediate operand for I-type instruction

There are 5 output registers listed here below:

- **PCRetReg** stores return address for JAL and JALR
- **Areg** stores RF output form port1
- **Breg** stores RF output from port2
- **ImmReg** stores the extended operand (32 bits) for I-type instructions
- **RFWRAddrID** stores the write address for the current operation

In this stage is also present the logic used to determine, in case of branch instruction, if it has to be taken or not. This information is sent back to the previous stage so in the next clock cycle we know if we have to flush or not the pipeline.

4.2.1 Register file

The most important part of this stage is the register file, where all the data are stored.
To handle the particular case in which we want both to write and read the same memory location, we decided to give the priority to the write phase, and the data that is being written in the memory is also reported to the output ready to be read. This allows us to remove some glitches and in this very particular case to save 1 clock cycle.
In the following abstract of code this mechanism is shown:

Listing 4.1: Register file

```
IF (CLK'EVENT AND CLK = '1') THEN
        IF (ENABLE = '1') THEN
        IF (WR = '1') THEN
                REGISTERS(to_integer(unsigned(ADD_WR))) <= DATAIN;
                        --SIMULTANEOU R/W
                        IF ADD_WR = ADD_RD1 THEN
                                OUT1 <= DATAIN;
                        END IF;
                        IF ADD_WR = ADD_RD2 THEN
                                OUT2 <= DATAIN;
                        END IF;
                END IF;
        ELSE--WHEN ENABLE '0' HIGH IMPEDANCE
                OUT1 <= (OTHERS => 'Z');
                OUT2 <= (OTHERS => 'Z');
        END IF;
END IF;
```

4.3 Execute

In this stage all the operations on the data take place. There are three registers:

- **StatusRegEx** stores the status information of the previous ALU operation
- **ALUReg** stores the ALU output
- **DMEMAddrReg** depending on the instruction type, it can contains the data to be stored in the next mem stage

The StatusMux81 is a 8 to 1 multiplexer whose task is to select which ALU flag we want to save in the RF (greater, equal, lower, ...).

The ALU is in charge to perform all the computation that are needed on the data, according to the ALUop signals sent by the CU. It has two inputs and one output for the data and it is composed by five components: Multiplier, Adder-Subtractor, Shifter, Comparator and Logic unit.

4.3.1 ADDER

Our adder-subtractor is based on the P4 adder, this architecture allows to reduce the carry propagation delay which is the bottleneck in these kind of structures.
The P4 adder is based on two substructures as shown in 4.2, a carry generator and a sum generator.

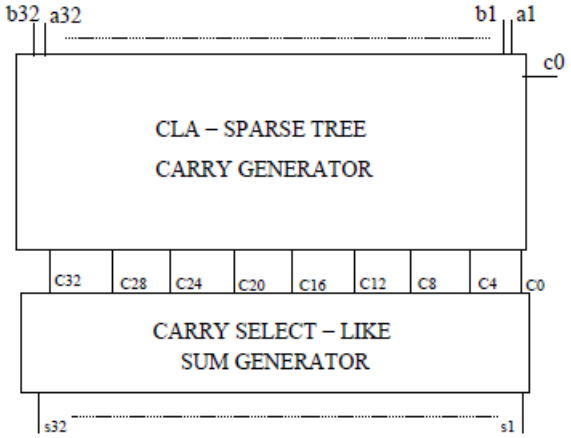

Figure 4.2: P4 structure

The first one is the **sparse tree** that has to generate all the carries for the final computation of the sum. Its structure is a quiet particular and to obtain it we need two "elementary" blocks:

- **G** block that produces the following output: $G_{i:j} = G_{i:k} + P_{i:k} * G_{k-1:j}$
- **PG** has two outputs: one is the same of G and the other one is $P_{i:j} = P_{i:k} * P_{k-1:j}$

The tree structure is shown in 4.3. As we can see it is not properly linear and the VHDL code is not so simple to understand at a first view. It consists of 4 cycles, one for the first row (0), one for the two consequent (1 and 2), a third one for the third and the last one for all the consequents rows. In our case the depth is 5. The full code is reported in Appendix C.

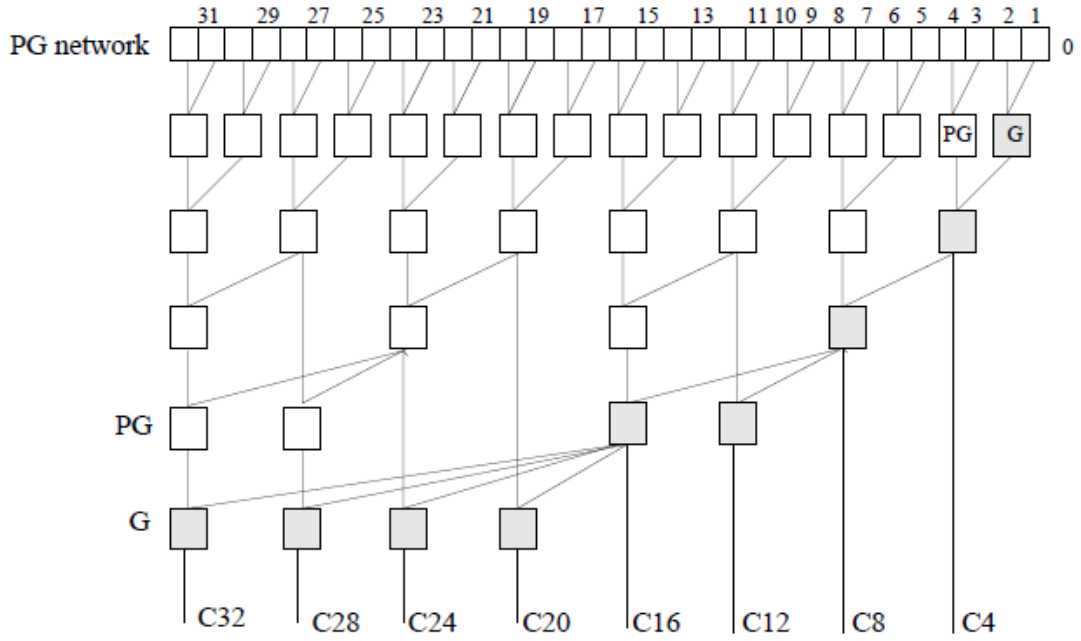

Figure 4.3: P4 carry tree structure

The structure of the top right PG was slightly modified in order to introduce the carry in and this permits both addition and subtraction.

The following part of the structure is the **carry select adder**. It is composed by several carry_select_blocks which generate 4 bits each one, by means of a small RCA supposing the input carry value. The component computes the result both for **carry-in = 0** and **carry-in = 1**. When the real one arrives from the above stage, a multiplexer chooses the right one.
The structure of this stage is presented here. In this way the sum generation and the carry generation phases occurs at the same time and so the speed is increased.

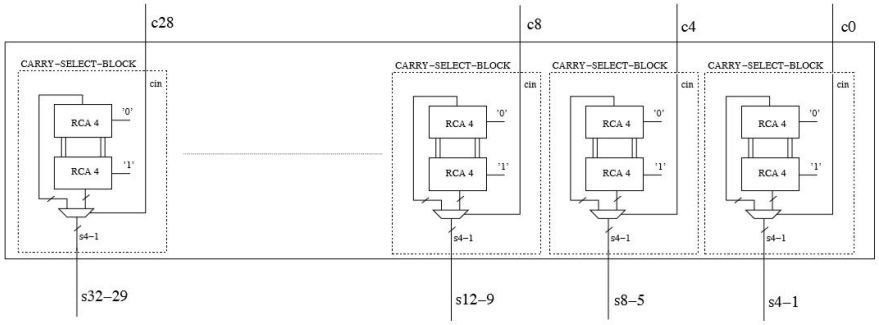

Figure 4.4: P4 Adder: carry select adders

4.3.2 COMPARATOR

The comparator gives us the relationship between the two input, in both signed and unsigned cases. In order to manage this feature we use an additional signal that tells us how to manage the two data. The other inputs are the difference between the two operands and their MSBs.
The logic is purely combinatorial and it works as follows:

Listing 4.2: Comparator

```
Temp(0) <= Diff (0);

Temp_g: for i in 1 to (Nbit-1) generate
        Temp(i) <= Temp(i-1) or Diff(i);
end generate;

Z <= not Temp(Nbit-1);
sign_cntrl <= (signA xor signB) and (not unsign);
AgB <= (Carry and ( not Z))xor sign_cntrl;
AgeqB <= Carry xor sign_cntrl;
AlB <= (not Carry) xor sign_cntrl;
AleqB <= ((not Carry) or (Z)) xor sign_cntrl;
AeqB <=  Z;
AnoteqB <= not Z;
```

4.3.3 MULTIPLIER

We have implemented a multiplier based on the Wallace tree architecture shown in the figure 4.5. At the end of the tree an adder is necessary to obtain the final value. These choices have been made to improve the performance because this architecture is one of the fastest. The Carry Save Adders are normal adders, one of the fastest type, with 3 inputs and 2 outputs (sum and carry).
The VHDL code for this structure is just the mapping of the tree.

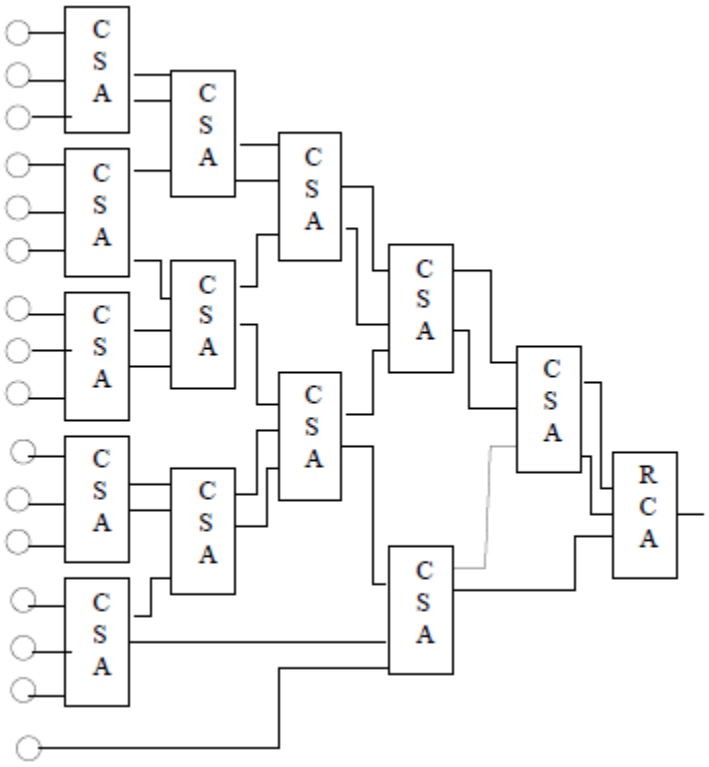

Figure 4.5: Wallace multiplier structure

This component is able to handle signed operands, this is done thanks to a logic that is present before the multiplier. It detects the signs of the operands and prepare them for the multiplication.
The following piece of code shows how we decide to handle it.

Listing 4.3: Sign handle

```
process(a, b, a_i, b_i)
        begin
                if((a(Nbit/2-1) = '1') and (b(Nbit/2-1) = '1')) then
                        a_i <= std_logic_vector(unsigned(not(a)) + 1);
                        b_i <= std_logic_vector(unsigned(not(b)) + 1);
                else
                        if(b(Nbit/2-1) = '1' and (a(Nbit/2-1) = '0')) then
                                a_i <= b;
                                b_i <= a;
                        else
                                a_i <= a;
                                b_i <= b;
                        end if;
                end if;
                for I in 0 to Nbit/2-1 loop
                        if(b_i(I)='1') then
                                init(I)(Nbit/2-1+I downto I) <= a_i;
                                if(a_i(Nbit/2-1) = '1') then
                                        init(I)(Nbit-1 downto Nbit/2+I) <= (others=>'1');
                                else
                                        init(I)(Nbit-1 downto Nbit/2+I) <= (others=>'0');
                                end if;
                        else
                                init(I) <= zero;
                        end if;
```

```
26                end loop;
        end process;
```

In order to avoid the overflow, we multiply only the lower 16 bits of the two operands. In this way we are sure that the result will always fit in 32 bits.

4.3.4 Shifter

Our choice for the shifter was not to use the T2 one, but another version. The main reason is that we wanted a circuit on one single stadio instead of three with a logic upstream, able to recognize all the possible scenarios with shift and rotate operations. In particular this is able to handle:

- Rotate left
- Rotate right
- Shift arithmetic left
- Shift arithmetic right
- Shift logic left
- Shift logic right

The implementation is very simple but very fast, it consists of a series of multiplexer 32 to 1 where each input corresponds to one of the bit that must be shifted. In this way by properly setting the selector of the mpx we are able to rotate all the 32 bits. If we want a shift, the logic upstream masks the undesired bits according to logical or arithmetic (the sign is maintained). The logic is also able to detect if there is a shift or a rotate for more than half of the number of bits (e.g. SLL 22). In this case it reverses the rotation to a right shift of 10 and the result is the same. In Appendix C the VHDL code for both the component and the logic are presented.

4.3.5 Logic unit

Also in this case we decide not to use the T2 logic (2 level) and we made our own logic unit component that is able to perform the same tasks with only one level of depth. As consequence of this the area is increased but the latency is decreased. The unit computes this 7 typologies of bitwise operations:

- NOT
- AND
- OR
- XOR
- NAND
- NOR
- XNOR

Also in this case the implementation is quiet simple, we use the fundamental logic gates to perform the tasks. An extract of the code is shown here below.

Listing 4.4: Logic unit

```
1 notAf:          for i in 0 to (Nbit-1) generate
                    notA(i) <= not A(i);
3                   end generate;

5 notBf:          for i in 0 to (Nbit-1) generate
                    notB(i) <= not B(i);
7                   end generate;

9 AandBf:         for i in 0 to (Nbit-1) generate
```

```
                        AandB(i) <= A(i) and B(i);
11                      end generate;

13  Aorf:           for i in 0 to (Nbit-1) generate
                        AorB(i) <= A(i) or B(i);
15                      end generate;

17  AxorBf:         for i in 0 to (Nbit-1) generate
                        AxorB(i) <= A(i) xor B(i);
19                      end generate;

21  AnandBf:        for i in 0 to (Nbit-1) generate
                        AnandB(i) <= A(i) nand B(i);
23                      end generate;

25  AnorBf:         for i in 0 to (Nbit-1) generate
                        AnorB(i) <= A(i) nor B(i);
27                      end generate;

29  AxnorBf:        for i in 0 to (Nbit-1) generate
                        AxnorB(i) <= A(i) xnor B(i);
31                      end generate;
```

4.4 Memory

Load and Store operations are performed in this stage. It is possible to read and write in different size:

- Word - 32 bits
- Half word - 16 bits
- Byte - 8 bits

Before the **Data memory**, two multiplexers are used, one for the data and one for the address respectively. Those because the load and store operations are codified in different ways so is necessary to swap the operands according to the instruction to be performed.
At the end of the stage is also present a write-back register (**WBReg**) for those instructions that do not use the **Data memory**.

4.5 Write back

This last stage consists of a single multiplexer that, after chosing the right input, reports back the data to the **Register file**.
The possible inputs of the mpx are:

- **StatusRegMEM** - the output of the comparator
- **LMDReg** - the output of Data memory
- **WBReg** - the output of an ALU operation
- **0** - to put at 0 one location of the RF

CHAPTER 5

Simulation

In order to check the behavior of our micro-processor, we wrote and test some assembly programs trying to simulate the most common working conditions that could happened. In the following some examples are shown.
The software used for the simulation is Modelsim, distributed by Altera. All the simulations are functional, so the delays of the signals are not take into account. To make the simulation phase easier and faster to understand, some internal meaningful signals were added to the waveform window and a script is used for the compilation. The final version of it is attached in Appendix D.

5.1 Register file

In Figure 5.1 is possible to see the behavior of the Register File when read and write commands occur. As we can see the output is available only when the enable signal is '1'.

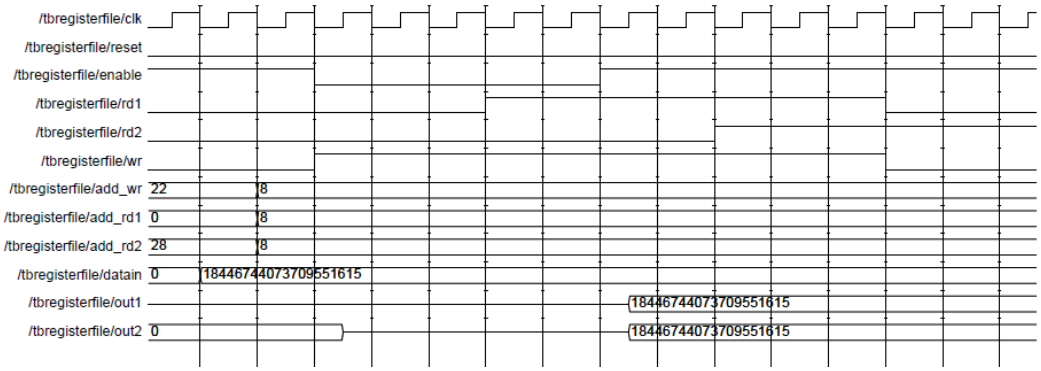

Figure 5.1: Read and wrire behavior

5.2 Generic program test

In order to simulate a common use of our DLX, we wrote three test programs to show the main features of the microprocessor.

- **Fibonacci series** it calculates and stores in memory the first N numbers of the series
- **4x4 matrix multiplication** it performs the multiplications between two 4x4 matrices
- **ISA** it tests the whole instruction set

The second one is explained here below.
This program stores in the memory the values of two matrices (4x4) and then calculates their product. The full program is reported in Appendix D while here are shown only some meaningful waves extracted from the simulation. As we can see it reports the Instruction register, the ALU operation code for every instruction and the Register file behavior. The result of this operation is stored in the main memory of the microprocessor.

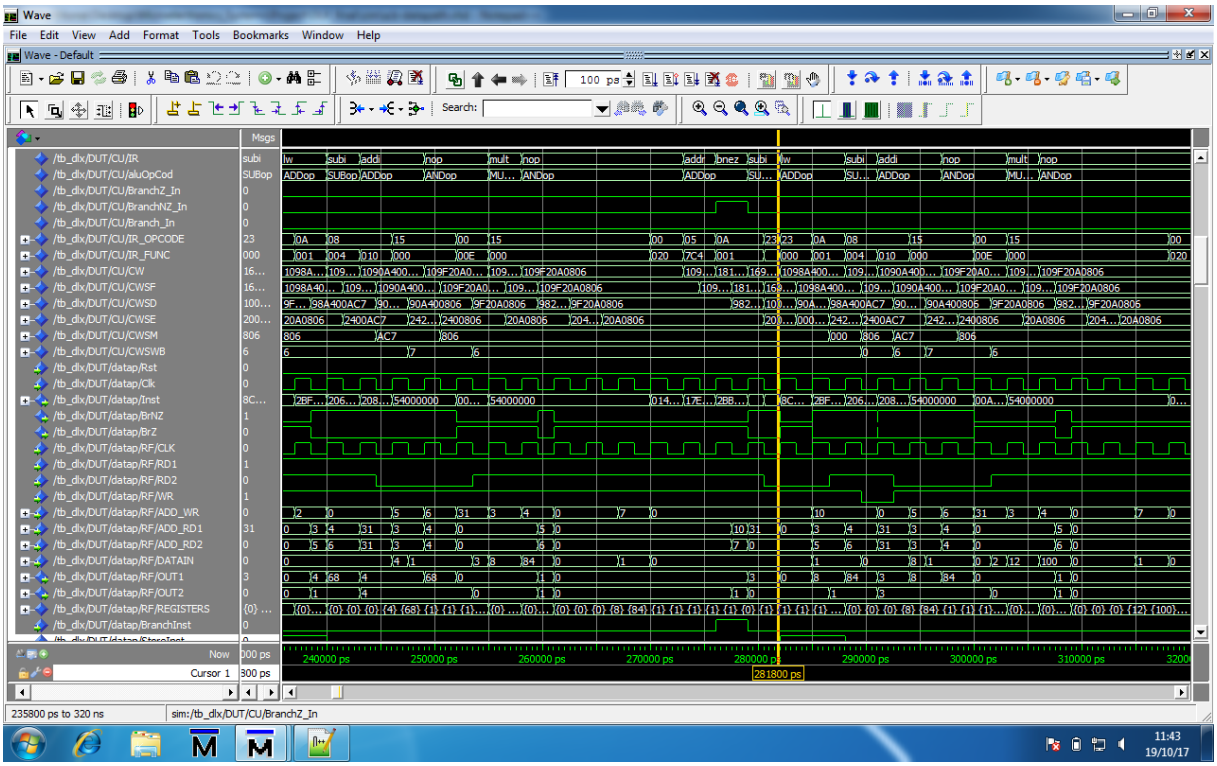

Figure 5.2: 4x4 matrices multiplication

CHAPTER 6

Synthesis

The tool used for the synthesis is design_vision, working in Synopsys environment. Also in this case some scripts were used in order to speed up the process. After a first synthesis, we tried to optimize the structure of the DLX both from timing and power point of view.
The top view structure of the component can be seen below, it consists of datapath and CU.

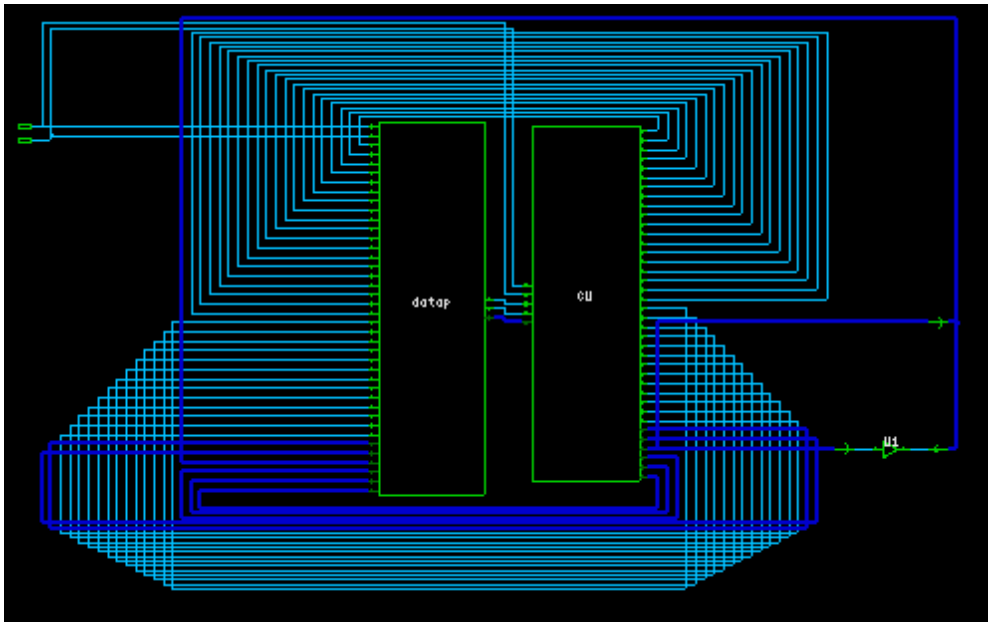

Figure 6.1: Structure of the DLX

To reduce the synthesis efforts, the memories were not synthesized. As we can see all the control signals start from the CU and go to the datapath.

6.1 ALU synthesis

For the DLX we create two different types of ALU named ALUv1 and ALUv2. The two have almost the same architecture and they are built with the same macro-components (multiplier, adder-substractor, etc...). The main difference between the two typologies is in how the operands are provided to them and how the result is computed. In ALUv1 the operations run in parallel so for each new set of inputs we compute the addition, the multipilication, etc.. Because the operands A and B are directly provided to each one of the macro-components. The output then is selected through a mpx which selector is the ALUsel signal. This type of architecture is very fast but has the main disadvantage to be very power consuming due to the high switching activity. Instead in ALUv2 we add a preliminary stage in which we prepare the input for the macro-components. In this phase a process identifies through the ALUsel signal which operation must be preformed and according to this it sets the enable signal for that component.

Listing 6.1: ALUv2 enable signal generation

```
inputgenMUL:    for i in 0 to (Nbit/2-1) generate
                        AinMUL(i) <= A(i) and enMUL;
                        BinMUL(i) <= B(i) and enMUL;
                end generate;

inputgen:       for i in 0 to (Nbit-1) generate
                        AinADDSUB(i) <= A(i) and enADDSUB;
                        BinADDSUB(i) <= B(i) and enADDSUB;
                        AinSHIFTER(i) <= A(i) and enSHIFTER;
                        BinSHIFTER(i) <= B(i) and enSHIFTER;
                        AinLOGIC(i) <= A(i) and enLOGIC;
                        BinLOGIC(i) <= B(i) and enLOGIC;
                end generate;

OUTgen: process (A,B,ALUsel,ADDSUBout,sumnsub,SHIFTERout,MULout,notAout,notBout,AandBout,
    AorBout,AxorBout,AnandBout,AnorBout,AxnorBout)
begin
        case (conv_integer(unsigned(ALUsel))) is
                when (conv_integer(unsigned(ADDcode))) =>
                        ALUout <= ADDSUBout;
                        sumnsub <= '0';
                        enADDSUB <='1';
                        enMUL <=  '0';
                        enLOGIC <= '0';
                        enSHIFTER <= '0';
                        Shift_Rotaten <= '0';
                        Right_LeftN <= '0';
                when (conv_integer(unsigned(SUBcode))) =>
                        ALUout <= ADDSUBout;
                        sumnsub <= '1';
                        enADDSUB <='1';
                        enMUL <=  '0';
                        enLOGIC <= '0';
                        enSHIFTER <= '0';
                        Shift_Rotaten <= '0';
                        Right_LeftN <= '0';
                when (conv_integer(unsigned(MULcode))) =>
                        ALUout<= MULout;
                        enADDSUB <='0';
                        enMUL <=  '1';
                        enLOGIC <= '0';
                        enSHIFTER <= '0';
                        Shift_Rotaten <= '0';
                        Right_LeftN <= '0';
```

Then each input of the macroblock is putted in a bitwise AND with the proper enable signal. In this way for each new operation we will have only one pair of non-zero inputs and so only one macro-component will work at a given time. Finally the right result is selected through a mpx and displayed to the output. Thanks to this architecture it is possible to reduce the switching activity and so the overall power consumption of the ALU. The main drawback is the insertion of the AND gates that increase the critical path of the circuit as well as the area. These are the synthesis power reports for the two architectures:

```
****************************************
Report : power
        -analysis_effort low
Design : ALU_v2
Version: Z-2007.03-SP1
Date   : Thu Oct 19 00:19:24 2017
****************************************


Library(s) Used:

    NangateOpenCellLibrary (File: /home/mariagrazia.graziano/do/libnangate/NangateOpenCellLibrary


Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

Design        Wire Load Model            Library
```

```
-------------------------------------------------
ALU_v2                 5K_hvratio_1_1    NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  =   1.6058 mW   (46%)
  Net Switching Power  =   1.8933 mW   (54%)
                         ---------
Total Dynamic Power    =   3.4992 mW  (100%)

Cell Leakage Power     =  97.0791 uW

****************************************
Report : power
        -analysis_effort low
Design : ALU_v1_Nbit32
Version: Z-2007.03-SP1
Date   : Thu Oct 19 01:05:12 2017
****************************************


Library(s) Used:

    NangateOpenCellLibrary (File: /home/mariagrazia.graziano/do/libnangate/NangateOpenCellLibrary


Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

Design        Wire Load Model            Library
------------------------------------------------
ALU_v1_Nbit32          5K_hvratio_1_1    NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  =   6.4383 mW   (52%)
  Net Switching Power  =   5.8908 mW   (48%)
                         ---------
Total Dynamic Power    =  12.3291 mW  (100%)

Cell Leakage Power     = 103.2780 uW
```

As we can see the ALUv1 consumes much more with respetc to the ALUv2. For this reason the last one has been implemented in the final acrhitecture of the DLX. In term of area and time there isn't a significant difference between the two versions.

6.2 DLX synthesis

The final script that we used is attached here and then commented step by step. After this we have further tried to improve the DLX in terms of power and area. The final results are displayed in the last part of this section.

Listing 6.2: Final synthesis script

```
analyze -library WORK -format vhdl {000-global.vhd}
analyze -library WORK -format vhdl {001-functions.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.a-iv.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.b-nd2.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.c-mux21.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.d-mux21N.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.e-fa.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.f-rcaN.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.g-PGblock.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.h-G.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.i-PG.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.l-
    SparseTreeCarryGenN.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.m-CSBlockN.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.n-CarrySumN.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.o-AddSubN.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.p-
    SparseTreeCarryGenNBM.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.q-CSBlockNBM.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.r-CarrySumNBM.vhd
    }
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.s-P4adderN.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.t-CSA.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.u-BoothMulWallace
    .vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.v-Comp.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.z-LogicFun_v2.vhd
    }
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.za-Mux2to1.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.zb-mux32to1.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.zc-
    bidir_shift_rot_N.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU.core/a.b.a.zd-
    bidir_shift_rot_N_interface.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.a-ALU_v2.vhd}

analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.c-fd_en.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.d-LatchEn.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.e-RegEn.vhd}

analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.h-mux41.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.i-mux41N.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.l-mux81.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.m-mux81N.vhd}
analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.n-mux161N.vhd}

analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.p-PC.vhd}


analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.q-registerfile.vhd}

analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.r-signExtension.vhd}

analyze -library WORK -format vhdl {a.b-Datapath.core/a.b.s-zerotest.vhd}

analyze -library WORK -format vhdl {a.b-datapath.vhd}

analyze -library WORK -format vhdl {a.a-CU_HW.vhd}

analyze -library WORK -format vhdl {a-DLX.vhd}

elaborate DLX -architecture Structure

create_clock -name "Clk" -period 2.97 Clk

compile -map_effort high

report_timing > Report_DLX_time_opt_time.txt
```

```
report_power > Report_DLX_time_opt_pow.txt
report_area > Report_DLX_time_opt_area.txt

write -hierarchy -format ddc -output DLX_structural_time_opt.ddc

create_clock -name "Clk" -period 3.5 Clk

compile -map_effort high -power_effort high

report_timing > Report_DLX_pow_time_opt_time.txt
report_power > Report_DLX_pow_time_opt_pow.txt
report_area > Report_DLX_pow_time_opt_area.txt

write -hierarchy -format ddc -output DLX_structural_time_pow_opt.ddc
```

After created the needed folder, we analyzed the VHDL files and we elaborted the whole structure of the micorprocessor. In the next step we created a clock signal and compiled the DLX architecture. Then we extracted timing, area and power reports. After this we generated a constraint for the clock period and we compiled again with high map effort. For the power optimization phase we reduced the constraint for the clock and we compiled again with high power effort option.
These are the final results obtained with slack met condition. The first one shows the power report with only time optimization.

```
Design        Wire Load Model            Library
------------------------------------------------
DLX                    5K_hvratio_1_1    NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  =   4.8464 mW   (86%)
  Net Switching Power  = 787.4512 uW   (14%)
                         ---------
Total Dynamic Power    =   5.6338 mW  (100%)

Cell Leakage Power     = 474.8714 uW
```

After some attempts, the minimum clock period achived is 2,97 ns and so the maximum working clock frequency is: 336,7 MHz.

The second one shows the power dissipation with both time and power optimization. In this case we are able to save about 14% of the power, but the maximum clock frequency is consequently decreased of about the same percentage. The maximum achieved clock frequency in this case is 289,86 MHz.

```
Design        Wire Load Model            Library
------------------------------------------------
DLX                   5K_hvratio_1_1    NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  =   4.1560 mW   (86%)
  Net Switching Power  = 678.8270 uW   (14%)
                         ---------
Total Dynamic Power    =   4.8349 mW  (100%)

Cell Leakage Power     = 469.9623 uW
```

CHAPTER 7

Place & Routing

The last step to be performed was the physical design implementation of our DLX using Encounter. We started by adding the power rings, stripes and cell placement. Then we performed the clock three synthesis and also the routing. During geometry check we had no violations. The gate count report was extracted and here are some of the information it provides:

Module	Gates	Cells	Area μm^2
DLX	25467	11744	20322,7
Datapath	23875	11181	19052,2
ALU	5206	3606	4154,4
MUL	3458	2326	2759,5
CU	1515	507	1209,5

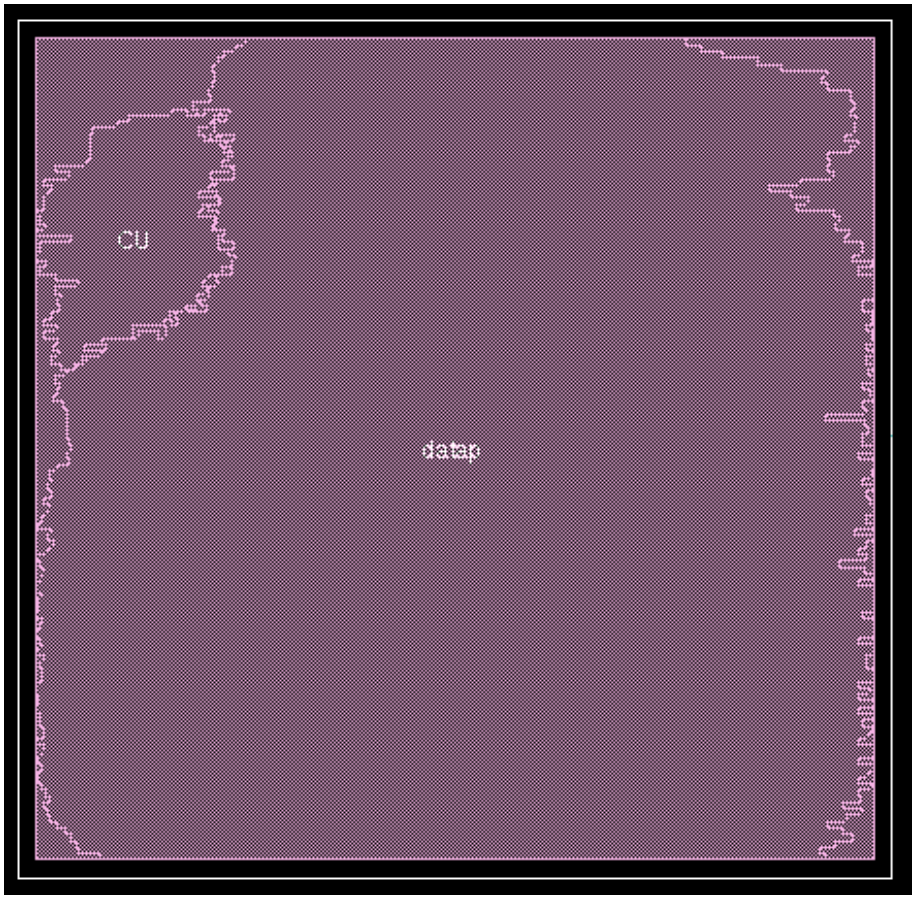

Figure 7.1: DLX ameoba view

APPENDIX A

Instructions

A.1 Instruction set

```
add
jalr
addi                    jr
and                     lb
andi                    lbu
beqz                    lhi
bnez                    lhu
j                       sb
jal                     seq
lw                      seqi
nop                     sgeu
or                      sgeui
ori                     sgt
sge                     sgti
sgei                    sgtu
sle                     sgtui
slei                    slt
sll                     slti
slli                    sltu
sne                     sltui
snei                    sra
srl                     srai
srli                    subu
sub                     subui
subi                    mult
sw                      rol
xor                     ror
xori                    roli
addu                    rori
addui
```

The explanation of each instruction is reported in the DLX project guide.

APPENDIX B

Datapath

B.1 Structure

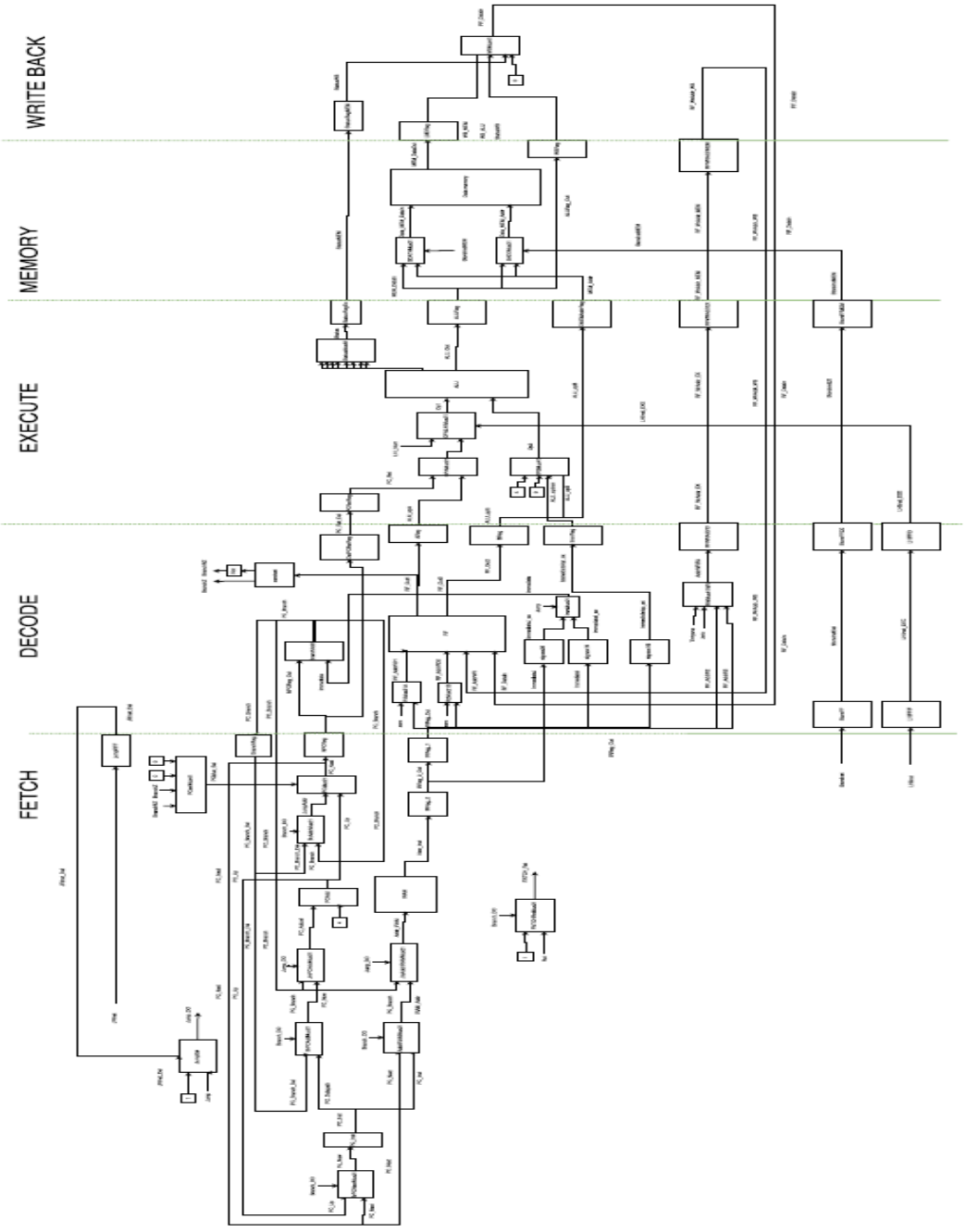

Figure B.1: Datapath structure

APPENDIX C

VHDL

C.1 SparseTreeCarryGenN

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
USE WORK.functions.all;

ENTITY SparseTreeCarryGenN IS
GENERIC ( Nbit: integer:=16 );
PORT (
                A :     IN std_logic_vector (Nbit-1 downto 0);
                B :             IN std_logic_vector (Nbit-1 downto 0);
                Cin :   IN std_logic;
                Cout :  OUT std_logic_vector ((Nbit/4) downto 0));
END SparseTreeCarryGenN;

ARCHITECTURE Structure OF SparseTreeCarryGenN IS

TYPE SignalVector IS ARRAY (log2_N(Nbit) DOWNTO 0) OF std_logic_vector(1 to Nbit);
SIGNAL prop : SignalVector;
SIGNAL gen : SignalVector;
SIGNAL propcin: std_logic;
SIGNAL gencin: std_logic;


COMPONENT G IS
PORT (
                G1:     IN std_logic;
                P1:     IN std_logic;
                G2:     IN std_logic;
                Gout:   OUT std_logic);
END COMPONENT;

COMPONENT PG IS
PORT (
                G1:     IN std_logic;
                P1:     IN std_logic;
                G2:     IN std_logic;
                P2:     IN std_logic;
                Gout:   OUT std_logic;
                Pout:   OUT std_logic);
END COMPONENT;

COMPONENT PGblock IS
PORT (
                A :     IN std_logic;
                B :     IN std_logic;
                G :     OUT std_logic;
                P :     OUT std_logic);
END COMPONENT;

BEGIN

rowgen: FOR row IN 0 TO log2_N(Nbit) GENERATE
        row0:   IF row=0 GENERATE
                el0: FOR I IN 1 TO Nbit GENERATE
                cin_prop:       IF ( I=1 ) GENERATE
                                Cinprop: PGblock PORT MAP ( A(i-1), B(i-1), gencin,
                                    propcin);
                                    gen(row)(i) <= (gencin OR (propcin AND Cin));
                                END GENERATE;
                other_prop:     IF (I>1) GENERATE
                                PGB: PGblock PORT MAP ( A(i-1), B(i-1), gen(row)(i),
                                    prop(row)(i));
                                END GENERATE;
                END GENERATE;
        END GENERATE;

        row1_2: IF row < 3 and row /= 0 GENERATE
```

```
                row1_2:FOR I IN 1 TO Nbit/2**row GENERATE
                        G_12: IF I=1 GENERATE
                                G1_2: G PORT MAP (gen(row-1)(2**(row)),prop(row-1)(2**(
                                    row)),gen(row-1)(2**(row)-row),gen(row)(2**row));
                        END GENERATE;
                        PG_12: IF I>1 GENERATE
                                PG1_2: PG PORT MAP (gen(row-1)(i*2**row),prop(row-1)(i
                                    *2**row),gen(row-1)(i*2**(row)-2**(row-1)),prop(row
                                    -1)(i*2**(row)-2**(row-1)),gen(row)(i*2**row),prop(
                                    row)(i*2**row));
                        END GENERATE;
                END GENERATE;
        END GENERATE;

        row3: IF row = 3 GENERATE
                el3:FOR I IN 1 TO Nbit/4 GENERATE
                        W_3:    IF (I mod 2)=1 GENERATE
                                        gen(row)(i*2**(row-1)) <= gen(row-1)(i*2**(row
                                            -1));
                                        prop(row)(i*2**(row-1)) <= prop(row-1)(i*2**(
                                            row-1));
                                END GENERATE;
                        G_3:    IF (I=2) GENERATE
                                        G3: G PORT MAP (gen(row-1)(i*2**(row-1)),prop(
                                            row-1)(i*2**(row-1)),gen(row-1)(i*2**(row
                                            -1)-2**(row-1)),gen(row)(i*2**(row-1)));
                                END GENERATE;
                        PG_3:   IF (((I mod 2)=0) and (I > 2)) GENERATE
                                PG3: PG PORT MAP (gen(row-1)(i*2**(row-1)),prop(row-1)(
                                    i*2**(row-1)),gen(row-1)(i*2**(row-1)-2**(row-1)),
                                    prop(row-1)(i*2**(row-1)-2**(row-1)),gen(row)(i
                                    *2**(row-1)),prop(row)(i*2**(row-1)));
                                END GENERATE;
                END GENERATE;
        END GENERATE;

        row3e: IF row > 3 GENERATE
                el3_e:FOR I IN 0 TO Nbit/4-1 GENERATE
                        W_3e:   IF ((I rem (2**(row-2)))<((2**(row-2))/2)) GENERATE
                                        gen(row)((i+1)*4) <= gen(row-1)((i+1)*4);
                                        prop(row)((i+1)*4) <= prop(row-1)((i+1)*4);
                                END GENERATE;

                        G_3e:   IF (((I rem (2**(row-2)))>=((2**(row-2))/2)) and (I <
                            2**(row-2))) GENERATE
                                        G3_E: G PORT MAP (gen(row-1)((i+1)*4),prop(row
                                            -1)((i+1)*4),gen(row-1)((i+1)*4-((i+1)-2**(
                                            row-3))*4),gen(row)((i+1)*4));
                                END GENERATE;

                        PG_3e:  IF (((I rem (2**(row-2)))>=((2**(row-2))/2)) and (I >
                            2**(row-2))) GENERATE
                                        PG3_E: PG PORT MAP (gen(row-1)((i+1)*4),prop(
                                            row-1)((i+1)*4),gen(row-1)((i+1)*4-((i+1)
                                            -2**(row-3)-(4*(i/(2**(row-2)))*(2**(row-4)
                                            )))*4),prop(row-1)((i+1)*4-((i+1)-2**(row
                                            -3)-(4*(i/(2**(row-2)))*(2**(row-4))))*4),
                                            gen(row)((i+1)*4),prop(row)((i+1)*4));
                                END GENERATE;
                        END GENERATE;
                END GENERATE;
END GENERATE;

outgen: FOR I IN 0 TO Nbit GENERATE
        Cout_0: IF i=0 GENERATE
                        Cout(0) <= Cin;
                END GENERATE;
        Cout_ot:IF (((i mod 4)=0) and (i /= 0)) GENERATE
                        Cout(i/4) <= gen(log2_N(Nbit))(i);
                END GENERATE;
END GENERATE;

END Structure;
```

C.2 bidir_shift_rot_N_logic

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
use IEEE.numeric_std.all;
use work.global.all;
use work.functions.all;

ENTITY bidir_shift_rot_N_interface IS
generic (
        Nbit: integer := Nbit);
PORT (
        data_in         :       IN STD_LOGIC_VECTOR(Nbit-1 downto 0);
        moves           :       IN STD_LOGIC_VECTOR(log2_N(Nbit)-1 downto 0);
        tot             :       IN STD_LOGIC;
        shift_rotN      :       IN STD_LOGIC;
        right_leftN     :       IN STD_LOGIC;
        arith_logN      :       IN STD_LOGIC;
        data_out        :       OUT STD_LOGIC_VECTOR(Nbit-1 downto 0));
END bidir_shift_rot_N_interface;

ARCHITECTURE struct OF bidir_shift_rot_N_interface IS

COMPONENT bidir_shift_rot_N IS
        generic (
        Nbit: integer := Nbit);
PORT (
        data_in         :       IN STD_LOGIC_VECTOR(Nbit-1 downto 0);
        sel                     :       IN STD_LOGIC_VECTOR(log2_N(Nbit)-1 downto 0);
        shift           :       IN STD_LOGIC_VECTOR(Nbit-1 downto 0);
        dx                      :       IN STD_LOGIC;
        arith_logN      :       IN STD_LOGIC;
        data_out        :       OUT STD_LOGIC_VECTOR(Nbit-1 downto 0));
END COMPONENT;

signal move : integer range 0 to 16;
signal move0 : integer range 0 to 31;
signal selt     : std_logic_vector(log2_N(Nbit)-1 downto 0);
signal shiftt   : std_logic_vector(Nbit-1 downto 0);
signal r_l      : std_logic;

BEGIN

move0 <= to_integer(unsigned(moves));

        process(data_in, shift_rotN, right_leftN, move0)
        begin
                if(right_leftN = '1') then
                        if(move0 > 16) then
                                move <= 32-move0;
                                r_l <= '0';
                        else
                                move <= move0;
                                r_l <= '1';
                        end if;
                end if;
                if(right_leftN = '0') then
                        if(move0>15) then
                                move <= 32-move0;
                                r_l <= '1';
                        else
                                move <= move0;
                                r_l <= '0';
                        end if;
                end if;
        end process;

        process(move, r_l)
        begin
                if(r_l = '1') then
                        case move is
                                when 0 => selt <= "00000";
                                when 1 => selt <= "00001";
                                when 2 => selt <= "00010";
                                when 3 => selt <= "00011";
                                when 4 => selt <= "00100";
                                when 5 => selt <= "00101";
                                when 6 => selt <= "00110";
                                when 7 => selt <= "00111";
                                when 8 => selt <= "01000";
                                when 9 => selt <= "01001";
                                when 10 => selt <= "01010";
```

```
                        when 11 => selt <= "01011";
                        when 12 => selt <= "01100";
                        when 13 => selt <= "01101";
                        when 14 => selt <= "01110";
                        when 15 => selt <= "01111";
                        when 16 => selt <= "10000";
                end case;
        end if;
        if(r_l = '0') then
                case move is
                        when 0 => selt <= "00000";
                        when 1 => selt <= "11111";
                        when 2 => selt <= "11110";
                        when 3 => selt <= "11101";
                        when 4 => selt <= "11100";
                        when 5 => selt <= "11011";
                        when 6 => selt <= "11010";
                        when 7 => selt <= "11001";
                        when 8 => selt <= "11000";
                        when 9 => selt <= "10111";
                        when 10 => selt <= "10110";
                        when 11 => selt <= "10101";
                        when 12 => selt <= "10100";
                        when 13 => selt <= "10011";
                        when 14 => selt <= "10010";
                        when 15 => selt <= "10001";
                        when 16 => selt <= "10000";
                end case;
        end if;
end process;

process (move, shift_rotN, right_leftN, tot)
begin
        if (shift_rotN = '1') then
                if (right_leftN = '1') then
                        case move0 is
                                when 0 => shiftt <=
                                    "00000000000000000000000000000000";
                                when 1 => shiftt <=
                                    "10000000000000000000000000000000";
                                when 2 => shiftt <=
                                    "11000000000000000000000000000000";
                                when 3 => shiftt <=
                                    "11100000000000000000000000000000";
                                when 4 => shiftt <=
                                    "11110000000000000000000000000000";
                                when 5 => shiftt <=
                                    "11111000000000000000000000000000";
                                when 6 => shiftt <=
                                    "11111100000000000000000000000000";
                                when 7 => shiftt <=
                                    "11111110000000000000000000000000";
                                when 8 => shiftt <=
                                    "11111111000000000000000000000000";
                                when 9 => shiftt <=
                                    "11111111100000000000000000000000";
                                when 10 => shiftt <=
                                    "11111111110000000000000000000000";
                                when 11 => shiftt <=
                                    "11111111111000000000000000000000";
                                when 12 => shiftt <=
                                    "11111111111100000000000000000000";
                                when 13 => shiftt <=
                                    "11111111111110000000000000000000";
                                when 14 => shiftt <=
                                    "11111111111111000000000000000000";
                                when 15 => shiftt <=
                                    "11111111111111100000000000000000";
                                when 16 => shiftt <=
                                    "11111111111111110000000000000000";
                                when 17 => shiftt <=
                                    "11111111111111111000000000000000";
                                when 18 => shiftt <=
                                    "11111111111111111100000000000000";
                                when 19 => shiftt <=
                                    "11111111111111111110000000000000";
                                when 20 => shiftt <=
                                    "11111111111111111111000000000000";
                                when 21 => shiftt <=
                                    "11111111111111111111100000000000";
                                when 22 => shiftt <=
                                    "11111111111111111111110000000000";
                                when 23 => shiftt <=
                                    "11111111111111111111111000000000";
                                when 24 => shiftt <=
```

```
                    "11111111111111111111111100000000";
                when 25 => shiftt <=
                    "11111111111111111111111110000000";
                when 26 => shiftt <=
                    "11111111111111111111111111000000";
                when 27 => shiftt <=
                    "11111111111111111111111111100000";
                when 28 => shiftt <=
                    "11111111111111111111111111110000";
                when 29 => shiftt <=
                    "11111111111111111111111111111000";
                when 30 => shiftt <=
                    "11111111111111111111111111111100";
                when 31 => shiftt <=
                    "11111111111111111111111111111110";
        end case;
end if;

if (right_leftN='0') then
        case move0 is
                when 0 => shiftt <=
                    "00000000000000000000000000000000";
                when 1 => shiftt <=
                    "00000000000000000000000000000001";
                when 2 => shiftt <=
                    "00000000000000000000000000000011";
                when 3 => shiftt <=
                    "00000000000000000000000000000111";
                when 4 => shiftt <=
                    "00000000000000000000000000001111";
                when 5 => shiftt <=
                    "00000000000000000000000000011111";
                when 6 => shiftt <=
                    "00000000000000000000000000111111";
                when 7 => shiftt <=
                    "00000000000000000000000001111111";
                when 8 => shiftt <=
                    "00000000000000000000000011111111";
                when 9 => shiftt <=
                    "00000000000000000000000111111111";
                when 10 => shiftt <=
                    "00000000000000000000001111111111";
                when 11 => shiftt <=
                    "00000000000000000000011111111111";
                when 12 => shiftt <=
                    "00000000000000000000111111111111";
                when 13 => shiftt <=
                    "00000000000000000001111111111111";
                when 14 => shiftt <=
                    "00000000000000000011111111111111";
                when 15 => shiftt <=
                    "00000000000000000111111111111111";
                when 16 => shiftt <=
                    "00000000000000001111111111111111";
                when 17 => shiftt <=
                    "00000000000000011111111111111111";
                when 18 => shiftt <=
                    "00000000000000111111111111111111";
                when 19 => shiftt <=
                    "00000000000001111111111111111111";
                when 20 => shiftt <=
                    "00000000000011111111111111111111";
                when 21 => shiftt <=
                    "00000000000111111111111111111111";
                when 22 => shiftt <=
                    "00000000001111111111111111111111";
                when 23 => shiftt <=
                    "00000000011111111111111111111111";
                when 24 => shiftt <=
                    "00000000111111111111111111111111";
                when 25 => shiftt <=
                    "00000001111111111111111111111111";
                when 26 => shiftt <=
                    "00000011111111111111111111111111";
                when 27 => shiftt <=
                    "00000111111111111111111111111111";
                when 28 => shiftt <=
                    "00001111111111111111111111111111";
                when 29 => shiftt <=
                    "00011111111111111111111111111111";
                when 30 => shiftt <=
                    "00111111111111111111111111111111";
                when 31 => shiftt <=
                    "01111111111111111111111111111111";
        end case;
```

```
                        end if;
                        if (tot ='1' and shift_rotN = '1') then
                                shiftt <= (others => '1');
                        end if;
                else
                        shiftt <= "00000000000000000000000000000000";
                end if;
        end process;

        bidir1: bidir_shift_rot_N PORT MAP (data_in, selt, shiftt, right_leftN,
            arith_logN, data_out);

END struct;
```

C.3 bidir_shift_rot_N

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_signed.all;
USE work.global.all;
USE work.functions.all;

ENTITY bidir_shift_rot_N IS
generic (
        Nbit: integer := Nbit);
PORT (
        data_in         :       IN STD_LOGIC_VECTOR(Nbit-1 downto 0);
        sel                     :       IN STD_LOGIC_VECTOR(log2_N(Nbit)-1 downto 0);
        shift           :       IN STD_LOGIC_VECTOR(Nbit-1 downto 0);
        dx                      :       IN STD_LOGIC;
        arith_logN      :       IN STD_LOGIC;
        data_out        :       OUT STD_LOGIC_VECTOR(Nbit-1 downto 0));
END bidir_shift_rot_N;

        -- data_in                      :       00000
        -- right_rot1   :       00001
        -- right_rot2   :       00010
        -- right_rot3   :       00011
        -- right_rot4   :       00100
        -- right_rot5   :       00101
        -- right_rot6   :       00110
        -- right_rot7   :       00111
        -- right_rot8   :       01000
        -- right_rot9   :       01001
        -- right_rot10  :       01010
        -- right_rot11  :       01011
        -- right_rot12  :       01100
        -- right_rot13  :       01101
        -- right_rot14  :       01110
        -- right_rot15  :       01111
        -- right_rot16  :       10000
        -- left_rot1    :       10001
        -- left_rot2    :       10010
        -- left_rot3    :       10011
        -- left_rot4    :       10100
        -- left_rot5    :       10101
        -- left_rot6    :       10110
        -- left_rot7    :       10111
        -- left_rot8    :       11000
        -- left_rot9    :       11001
        -- left_rot10   :       11010
        -- left_rot11   :       11011
        -- left_rot12   :       11100
        -- left_rot13   :       11101
        -- left_rot14   :       11110
        -- left_rot15   :       11111

ARCHITECTURE struct OF bidir_shift_rot_N IS

COMPONENT Mux2to1 IS
        PORT (  x1      : IN STD_LOGIC;
                        x2      : IN STD_LOGIC;
                        s       : IN STD_LOGIC;
                        f       : OUT STD_LOGIC);
END COMPONENT;

COMPONENT mux32to1 IS
        PORT (  A0, A1, A2, A3, A4, A5, A6, A7                  : IN    STD_LOGIC;
                        A8, A9, A10, A11, A12, A13, A14, A15    : IN    STD_LOGIC;
                        A16, A17, A18, A19, A20, A21, A22, A23  : IN    STD_LOGIC;
                        A24, A25, A26, A27, A28, A29, A30, A31  : IN    STD_LOGIC;
                        s       : IN    STD_LOGIC_VECTOR(5-1 downto 0);
                        F       : OUT   STD_LOGIC);
```

```
END COMPONENT;

SIGNAL muxout: STD_LOGIC_VECTOR(Nbit-1 downto 0);
SIGNAL dffout_t: STD_LOGIC_VECTOR(2*Nbit-1 downto 0);
SIGNAL dffout_tt: STD_LOGIC_VECTOR(2*Nbit-1 downto 0);
SIGNAL U: STD_LOGIC_VECTOR(Nbit-1 downto 0);
SIGNAL data: STD_LOGIC;

BEGIN

        stages: FOR I IN Nbit/2 TO Nbit+Nbit/2-1 GENERATE

                muxes: mux32to1 PORT MAP (data_in(I-Nbit/2), dffout_tt(I+1), dffout_tt(
                    I+2), dffout_tt(I+3), dffout_tt(I+4), dffout_tt(I+5), dffout_tt(I
                    +6), dffout_tt(I+7), dffout_tt(I+8), dffout_tt(I+9), dffout_tt(I
                    +10), dffout_tt(I+11), dffout_tt(I+12), dffout_tt(I+13), dffout_tt(
                    I+14), dffout_tt(I+15), dffout_tt(I+16), dffout_tt(I-15), dffout_tt
                    (I-14), dffout_tt(I-13), dffout_tt(I-12), dffout_tt(I-11),
                    dffout_tt(I-10), dffout_tt(I-9), dffout_tt(I-8), dffout_tt(I-7),
                    dffout_tt(I-6), dffout_tt(I-5), dffout_tt(I-4), dffout_tt(I-3),
                    dffout_tt(I-2), dffout_tt(I-1), sel, muxout(I-Nbit/2));
                data <= data_in(Nbit-1) and dx and arith_logN;
                mpxs: Mux2to1 PORT MAP (muxout(I-Nbit/2), data, shift(I-Nbit/2), U(I-
                    Nbit/2));
                dffout_t(I) <= data_in(I-Nbit/2);
        END GENERATE;
        dffout_tt(2*Nbit-1 downto Nbit+Nbit/2) <= dffout_t(Nbit-1 downto Nbit/2);
        dffout_tt(Nbit/2-1 downto 0) <= dffout_t(Nbit+Nbit/2-1 downto Nbit);
        dffout_tt(Nbit+Nbit/2-1 downto Nbit/2) <= dffout_t(Nbit+Nbit/2-1 downto Nbit/2)
            ;

        data_out <= U;

END struct;
```

APPENDIX D

SCRIPTS AND PROGRAMS

D.1 Final simulation script

```
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/000-global.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/001-functions.
    vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.a-iv.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.b-nd2.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.c-mux21.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.d-mux21N.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.e-fa.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.f-rcaN.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.g-PGblock.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.h-G.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.i-PG.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.l-SparseTreeCarryGenN.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.m-CSBlockN.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.n-CarrySumN.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.o-AddSubN.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.p-SparseTreeCarryGenNBM.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.q-CSBlockNBM.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.r-CarrySumNBM.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.s-P4adderN.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.t-CSA.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.u-BoothMulWallace.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.v-Comp.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.z-LogicFun_v2.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.za-Mux2to1.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.zb-mux32to1.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.zc-bidir_shift_rot_N.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU.core/a.b.a.zd-bidir_shift_rot_N_interface.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.a-ALU_v2.vhd

vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.b-DataMemory.vhd

vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.c-fd_en.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.d-LatchEn.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.e-RegEn.vhd
```

```
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.h-mux41.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.i-mux41N.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.l-mux81.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.m-mux81N.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.n-mux161N.vhd

vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.o-IRAM.vhd

vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.p-PC.vhd


vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.q-registerfile.vhd

vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.r-signExtension.vhd

vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-Datapath.
    core/a.b.s-zerotest.vhd

vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.b-datapath.
    vhd

vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a.a-CU_HW.vhd

vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/a-DLX.vhd
vcom -reportprogress 300 -work work /home/ms17.19/Project/DLX_final/sim/testbench/
    tb_DLX.vhd

vsim -t 100ps -novopt work.tb_DLX(TEST)
add wave *
add wave  sim:/tb_dlx/DUT/CU/*
add wave  sim:/tb_dlx/DUT/datap/*
add wave  sim:/tb_dlx/DUT/datap/RF/*
add wave  sim:/tb_dlx/DUT/datap/ALU/*
add wave  sim:/tb_dlx/DUT/datap/DataMem/*
```

D.2 4x4 matrix multiplication

```
addi r1,r0,#1 ;S11
addi r2,r0,#1 ;S12
addi r3,r0,#1 ;S13
addi r4,r0,#1 ;S14
addi r5,r0,#1 ;S21
addi r6,r0,#1 ;S22
addi r7,r0,#1 ;S23
addi r8,r0,#1 ;S24
addi r9,r0,#1 ;S31
addi r10,r0,#1 ;S32
addi r11,r0,#1 ;S33
addi r12,r0,#1 ;S34
addi r13,r0,#1 ;S41
addi r14,r0,#1 ;S42
addi r15,r0,#1 ;S43
addi r16,r0,#1;S44
sw 4(r0),r1 ; store S11
addi r17,r0,#1 ;S11
addi r18,r0,#1 ;S12
addi r19,r0,#1 ;S13
addi r20,r0,#1 ;S14
addi r21,r0,#1 ;S21
addi r22,r0,#1 ;S22
addi r23,r0,#1 ;S23
addi r24,r0,#1 ;S24
addi r25,r0,#1 ;S31
addi r26,r0,#1 ;S32
addi r27,r0,#1 ;S33
addi r28,r0,#1 ;S34
addi r29,r0,#1 ;S41
addi r30,r0,#1 ;S42
addi r31,r0,#1 ;S43
addi r1,r0,#1 ;S44
sw 8(r0),r2 ; store S12
sw 12(r0),r3 ; store S13
sw 16(r0),r4 ; store S14
sw 20(r0),r5 ; store S21
sw 24(r0),r6 ; store S22
sw 28(r0),r7 ; store S23
sw 32(r0),r8 ; store S24
sw 36(r0),r9 ; store S31
sw 40(r0),r10 ; store S32
sw 44(r0),r11 ; store S33
sw 48(r0),r12 ; store S34
sw 52(r0),r13 ; store S41
sw 56(r0),r14 ; store S42
sw 60(r0),r15 ; store S43
sw 64(r0),r16 ; store S44
sw 68(r0),r17 ; store S11
sw 72(r0),r18 ; store S12
sw 76(r0),r19 ; store S13
sw 80(r0),r20 ; store S14
sw 84(r0),r21 ; store S21
sw 88(r0),r22 ; store S22
sw 92(r0),r23 ; store S23
sw 96(r0),r24 ; store S24
sw 100(r0),r25 ; store S31
sw 104(r0),r26 ; store S32
sw 108(r0),r27 ; store S33
sw 112(r0),r28 ; store S34
sw 116(r0),r29 ; store S41
sw 120(r0),r30 ; store S42
sw 124(r0),r31 ; store S43
sw 128(r0),r1 ; store S44
addi r16,r0,#64 ;
addi r17,r0,#4 ; start address matrix 1
addi r10,r0,#0 ; reset accumulator
addi r31,r0,#4 ;number of row
addi r29,r0,#4 ;number of column
addi r20,r0,#160 ; memory start address for result matrix
addi r30,r0,#16 ;number of operations
addi r3,r0,#4 ; first address of matrix 1
addi r4,r0,#68 ; first address of matrix 2
addi r1,r0,#0
addi r2,r0,#0
nop
nop
mulsum:
lw r5,0(r3) ;load number of matrix 1
lw r6,0(r4) ;load number of matrix 2
subi r31,r31,#1 ;update counter of column
```

```
addi r3,r3,#4 ; update address row
addi r4,r4,#16 ; update column address
nop
nop
mult r7,r5,r6 ;product
nop
nop
nop
nop
nop
add r10,r10,r7 ; sum of product
bnez r31,mulsum
subi r29,r29,#1 ; update number of operation in row
subi r30,r30,#1 ; update number of calculation
addi r31,r0,#4 ; restore counter of column
subi r4,r4,#60 ; new column
nop
nop
nop
bnez r29,firstrow
addi r17,r17,#16 ; new row
addi r4,r0,#68 ; restore first address of matrix 2
addi r29,r0,#4 ;
nop
nop
nop
nop
firstrow:
add r3,r0,r17 ; new row start address
nop
sw 0(r20),r10 ; store result of multiplication 164 184 204 224
addi r20,r20,#4 ; update store address for result
addi r10,r0,#0 ; reset accumulator
nop
nop
bnez r30,mulsum
nop
nop
nop
nop
nop
```