Politecnico di Torino

Microelectronic Systems

DLX Microprocessor: Design & Development
Final Project Report

Master degree in Electronics Engineering
Master degree in Computer Engineering

Referents: Prof. Mariagrazia Graziano, Giovanna Turvani

Authors: group 12

Tesser Andrea, Vianello Nicola

October 29, 2019

Features

32-bit RISC CPU:

- Five stages pipeline with forwarding
- Automatic hazard detection
- Up to 4GB of instructions memories
- Up to 4GB of data memories
- Branch target buffer with 4-way 8-set associative cache, 8-bit TAG, LRU policy replacement
- 4-cycle hardware multiplier

45 nm **CMOS technology**

- Max clock frequency: 411 MHz
- Supply voltage: 1.1 V
- Total power dissipation: 3.79 mW
- Silicon area: 29 000 µm^2

Supported instruction set
Register-to-register operations: ADD, ADDU, AND, OR, SGE, SLE, SLL, SNE, SRL, SUB, SUBU, XOR, SGEU, SGT, SGTU, SLEU, SLT, SLTU, SRA, SEQ, ROR, ROL, MULT.
Immediate operations: ADDI, ADDUI, SUBI, SUBUI, ANDI, ORI, XORI, SGEI, SGEUI, SGTI, SGTUI, SLEI, SLEUI, SLTI, SLTUI, SLLI, SRLI, SRAI, SNEI, SEQI, ROLI, RORI, MULTI.
Load and store: LW, LH, LHU, LB, LBU, SW, SH, SB.
Jump: J, JR, JAL, JALR, BEQZ, BNEZ.
Other instructions: NOP.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

DLX

The top level view of the processor is shown in figure 1.1, it has in addition to the clock and the reset ports, an interface for the IRAM (Instruction RAM) and another for the DRAM (Data RAM). Inside it we find the following blocks:

- The datapath: it is the block where both the memories are connected from the outside except for the DRAM write enable that is drive by the CU (Control Unit). Furthermore it receives internally the control from the CU, the FU (Forwarding Unit), and the HDU (Hazard Detection Unit), and the only internal output signal is the misprediction one, used by the BPU (Branch Prediction Unit) to inform the HDU that an hazard occurs.
- A set of registers that are used to keep track of all the instructions running inside the pipeline.
- The CU, which receives the instruction in the ID stage and according to this gives the appropriate control outputs.
- A set of registers that propagate the CU outputs, in order to send the signals to the datapath at the right time depending on the pipeline stage where they are used.
- The FU that reading all the instruction in execution from the second stage to the last stage can detect a data dependance and, in the case this happens, it can drive the datapath to forward the datum in a stage of the pipeline to another stage.
- The HDU, according on the instruction in the second and in the third stage and on the misprediction signal it has the ability to stop some pipeline stage and to discards a fetched instruction, in order to avoid any kind of hazard.

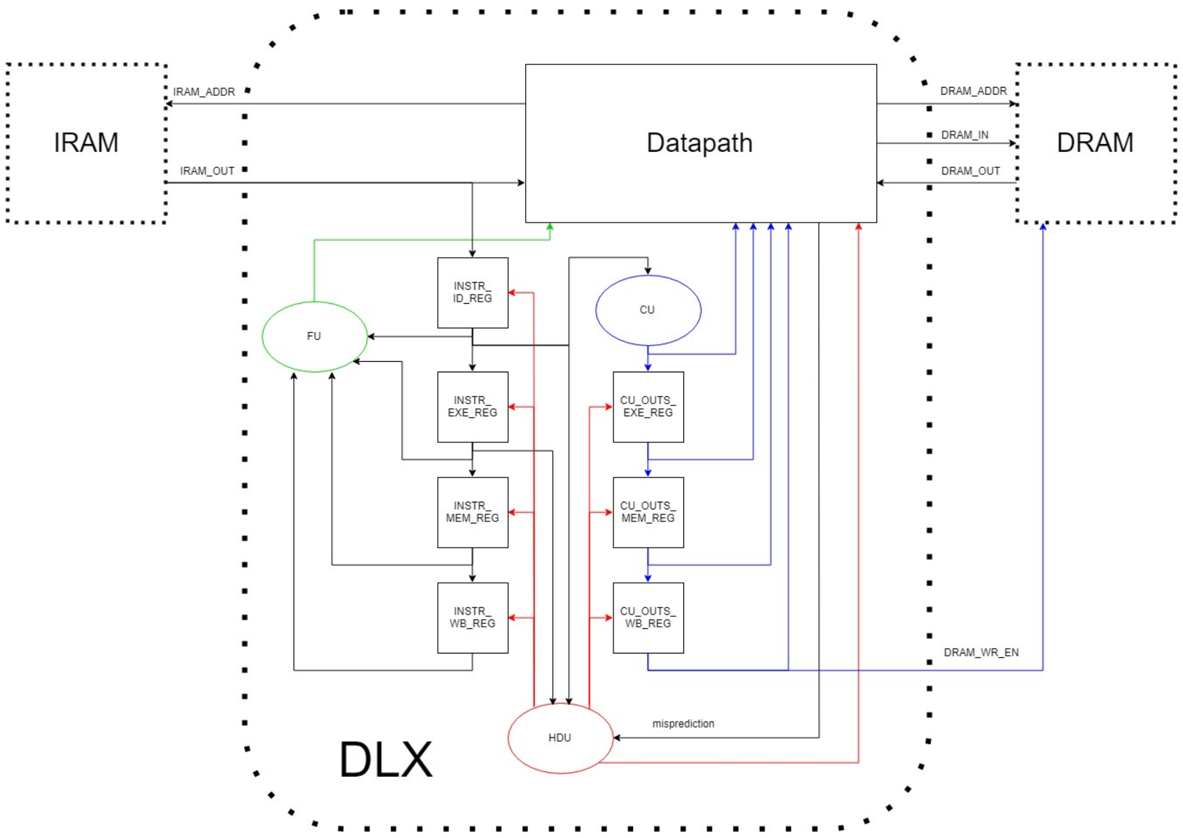


Figure 1.1: DLX diagram.

CHAPTER 2

Datapath

The block diagram of the whole datapath is shown in figure 2.1. It is divided into five pipeline stages:

1. The first is called IF (Instruction Fetch) and has the task of fetch the instructions from the IRAM. Here is collocated the PC (Program Counter) register, the adder that sum four to the value of the PC, and ideally the IRAM remembering however that it is connected from the outside. In the diagram the BPU is shown in this stage, because we can think of it as a MUX that drives the PC input and receives in input the NPC (Next program counter), the actual address computed in the next stage, and the output of the cache inside it. However it is important to remember, that the control circuit is actually sequential and works both in the IF/ID step where it stores the information of a possible prediction, and in the ID/EXE step where it updates the cache based on the result of the prediction. For more accurate information about how it works see section 2.1.

2. The second stage is called ID (Instruction Decode) and has the task of decode the fetched instruction, to read the RF (Register file), and to compute the actual address following a jump. Here is also when the CU receives the fetched instruction and gives the appropriate control outputs. As regards the jump address, there are an adder that sum the value of the NPC to the immediate offset and a MUX can takes this output or the value of a register based whether the jump has an immediate offset or whether the address is taken from a register. Another MUX receives the so computed address and the value of the NPC and driven by the branch comparator (see section 2.4 for more details) send the actual address to the BPU which can compare it to the made prediction. Furthermore there are more MUX to forwarding the registers, to select whether the immediate field is of 26-bit (immediate jump) or of 16-bit (immediate operations, load/store, and conditional jump), and to select whether the RF write address are in the range 15 downto 11 (register to register operations), in the range 20 downto 16 (immediate operations and loads), or it is equal to 31 (subroutine calls: JAL, JALR).

3. The third is the EXE (EXEcute) stage, where the arithmetic and logic operations are carried out. This is do thanks to the ALU (see section 2.2) or to the multiplier in the case of a MULT or a MULTI operation. The two outputs will be selected by a MUX and another MUX determine whether the second operand is taken from the second register or from the immediate value. A last MUX is used for a particular case of forwarding, it will be explained in detail in chapter 3.2.

4. The fourth stage is aimed to read and write the DRAM. As for IRAM we can ideally place the DRAM here, without forgetting that it is not part of the datapath and will be connected from the outside. The address input and the data input of the memory are just connected to the operation output and to the second register output respectively, while the data output has

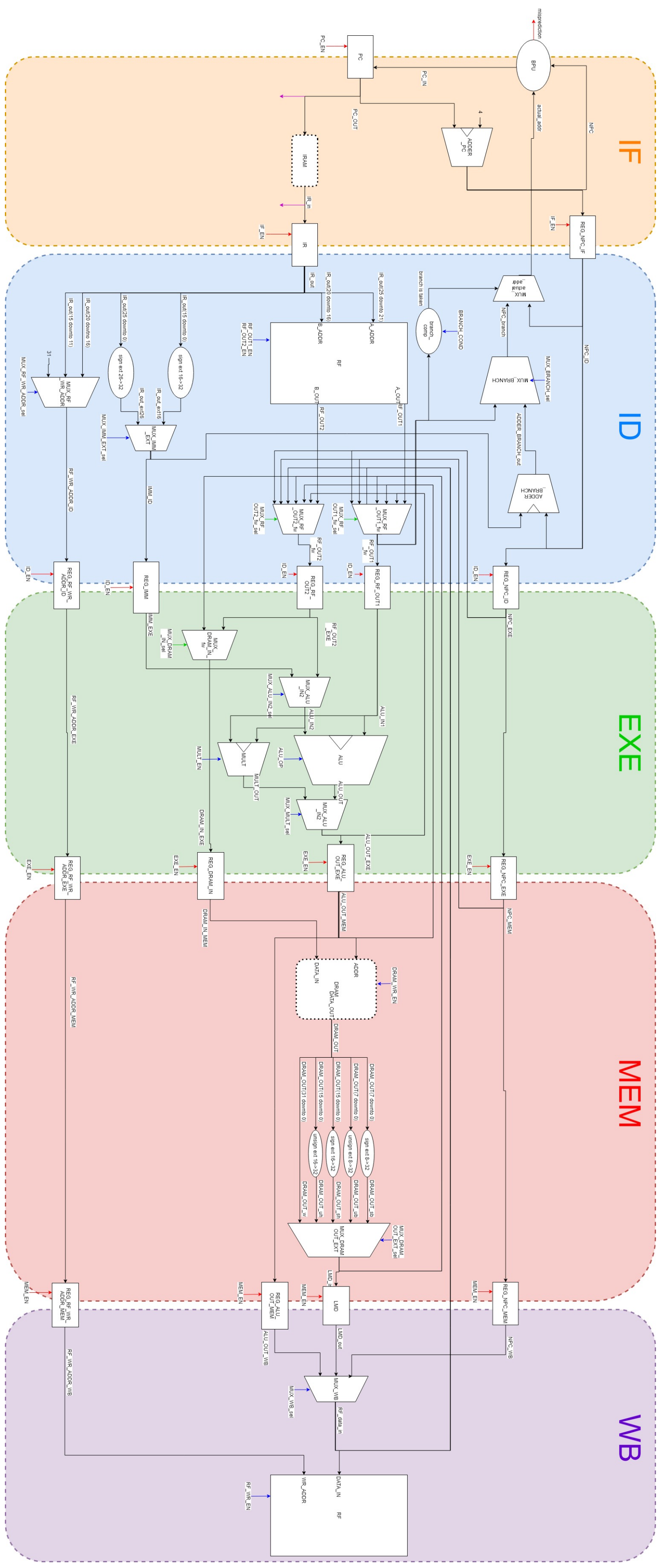


Figure 2.1: Datapath diagram. For clarity, the RF is represented twice, in the ID stage only for the reading-related ports, and in the WB stage for the write ports. The blue arrows represent the control from the CU, red arrows represent control from HDU, green arrows control from FU, purple arrows are signals read by the BPU.

a slightly more complicated circuit: there are a set of bitwidth-extenders whose outputs are selected by a MUX controlled by the CU, depending on the type of load required (byte/half-word/word, signed/unsigned) only one extender is read.

5. The fifth and last stage, WB (Write-Back), is where the RF is write. The only additional hardware is a MUX that choose the datum to write beetween the DRAM output, the logic/arithmetic output, or the value of PC+4. The latter is propagated since the IF stage using a register for each pipeline stage and is used when a subroutine call occurs and so it is necessary to save the return address in the register 31 which has the purpose of link register.

2.1 Branch Prediction Unit

The adopted BPU is a branch target buffer using a set-associative cache and a LRU policy replacement. It is positioned in the fetch stage, so as to receive the value of PC+4 and to drive the PC register (see figure 2.1). In normal conditions the value of PC+4 is reported directly to the output, but as soon as a jump instruction (conditional or unconditional) is fetched, the value of the program counter is read and a search is performed in the cache memory according on its set and tag fields. If the search is successful (there is a hit), the corresponding datum is read from the cache memory and is sent out instead of PC+4, this datum is the prediction on what will be the address following the fetched jump. If instead no data is stored in the cache with a set and a tag corresponding to those of the current program counter (there is a miss) it means that a jump has not yet be found with these parameters, in this case the PC+4 is output anyhow. In both cases (hit or miss) at the rising edge of the clock, program counter set and tag, the received address and the output one, and in the case of a hit also in which line it occurred, are temporarily saved, moreover a flag is raised to make the BPU understand that in the next clock cycle it will have to check if the prediction was correct and, if not, to make the necessary corrections. After the clock cycle, once the flag is raised, the prediction is compared to the actual computed address and if they are different the HDU is informed that it will have to discard the fetched instruction, losing a clock cycle (see chapter 3.3). Furthermore in this phase the cache memory is updated: if before there was a miss a new data is added using the saved set and tag and memorizing the address computed by the datapath, if all the lines within the set are already used the less recently used one is discarded. If before there was a hit but the prediction is wrong the address is only updated with the correct one and the counters that indicate the order in which the lines of the set were accessed are adjusted. If instead the hit has led to a correct prediction, only this second action is performed.

In figure 2.2 the procedure just described is shown schematically.

2.2 ALU

The ALU structure is shown in figure 2.3, it consists on an adder/subtractor, a barrel shifter and a logic block. The adder/subtractor has the task to compute the addition or the subtraction between the two inputs, the latter is needed not only when the corresponding instruction is executed but also when conditional-set instructions are performed. In fact this output is read by the logic block together with the carry out and the two operands, and depending on these all the comparisons are made (see subsection 2.2.2 for the exact used logics). Last but not less important component is the barrel shifter, it allows to perform not only logical and arithmetic shifts in the two directions, but also to rotate the input data. The block to use, the type of operation and the output to read will be established by a third input driven by the control unit.

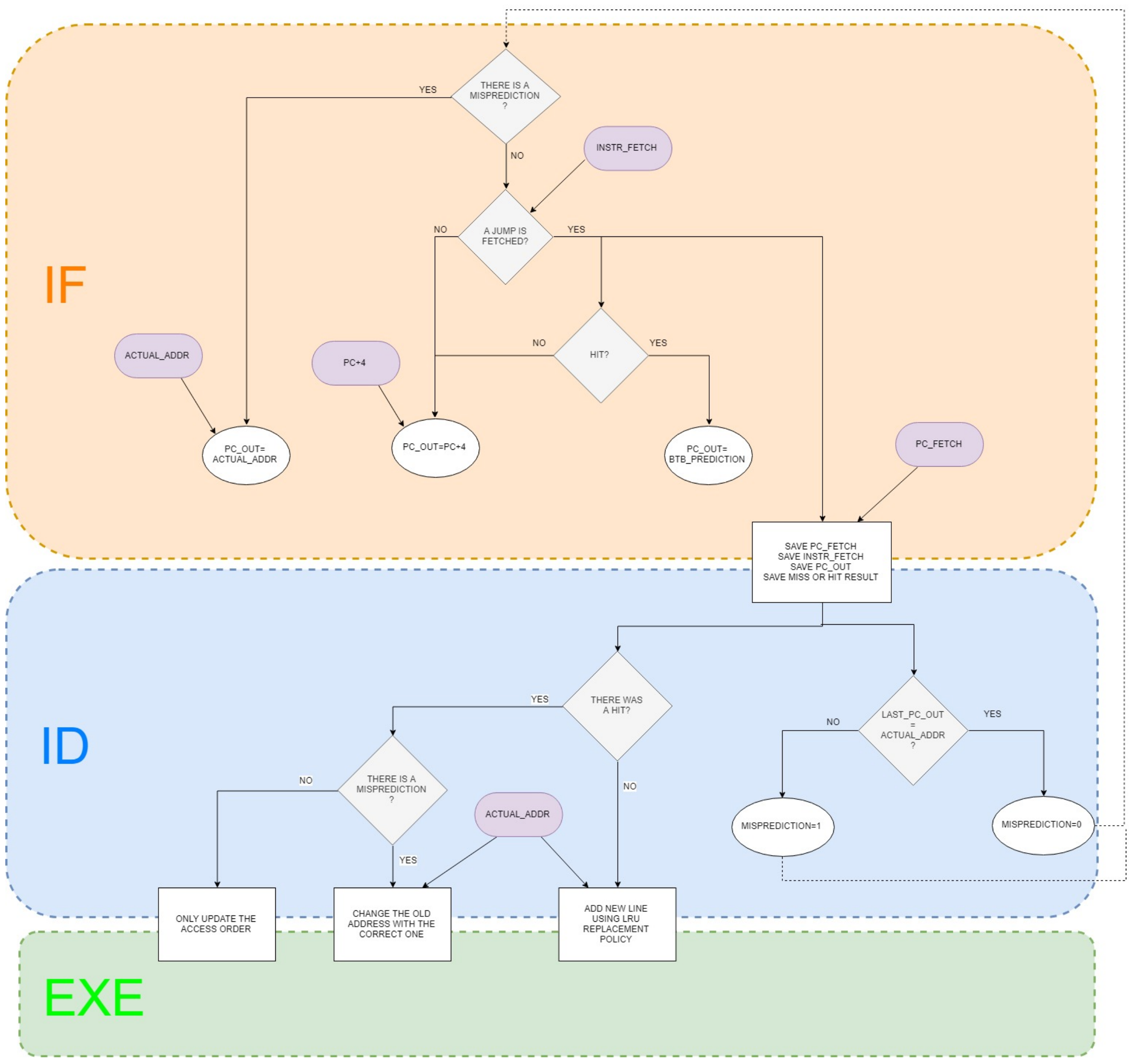


Figure 2.2: Flowchart of the BPU operation. The elliptical signals are combinational output, the squares content is perfomed at the rising clock edge, and the purple signal are the input from the outside.

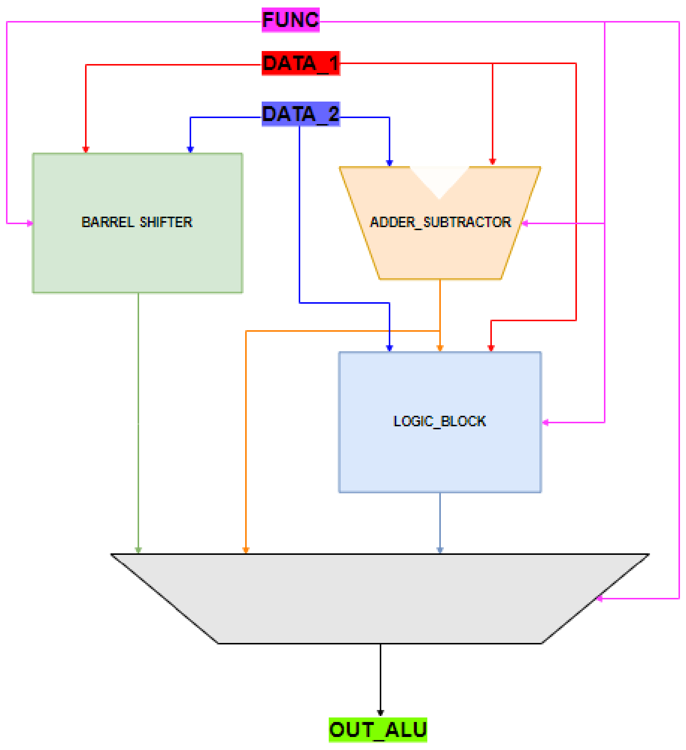


Figure 2.3: ALU internal diagram.

2.2.1 Adder/Subtractor

The operations on high number of bits are high time consuming, so, the architecture belonging to the Pentium 4 is used to reduce the delay of this block. This implementation is a tree adder, a particular family of the Carry Look Ahead that has the advantage of using particular network to compute the carry propagation in a faster way compared to others. To obtain a subtractor the circuit of figure 2.4 are adopted:

- An Add/Sub signal is added, when it is set to 0 an addition is performed, when it is 1 it perform a subtraction;
- Every bit of the input B are sent in a xor gate with the Add/Sub signal, so, if this one is 0, nothing happens and B pass unchanged, while if it is 1, the xor gate invert B;

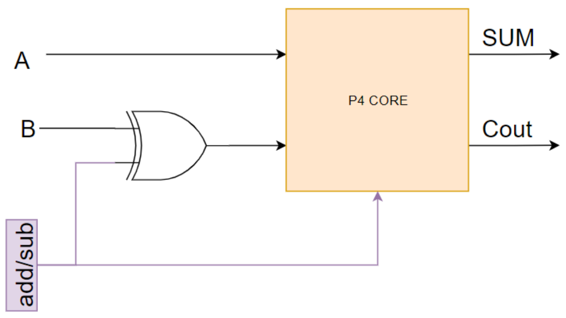


Figure 2.4: Circuit used to perform the subtraction.

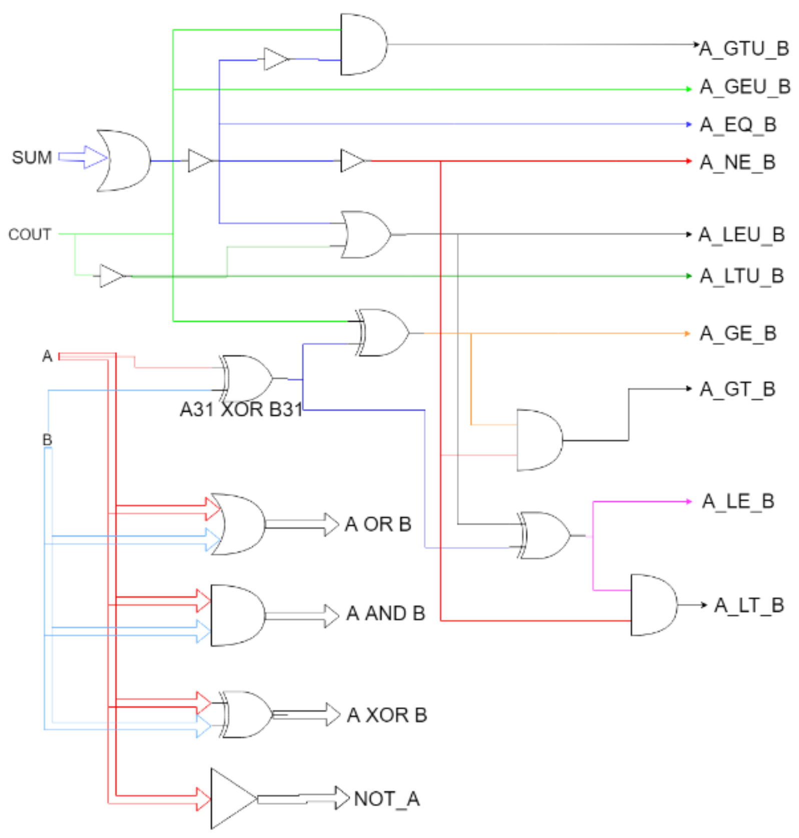


Figure 2.5: Logic block internal diagram.

- The Add/Sub signal is always summed with A and B through the carry in, in order to obtain a 2'complements operation.

2.2.2 Logic Block

This block is designed to perform logic functions and comparisons. The figure 2.5 report the internal architecture of the block. It take as input:

- The result of the adder/subtractor;
- The carry out;
- Input operand A;
- Input operand B.

The input operand A and B are used to compute the bitwise NOT, XOR, AND and OR on 32 bit, that doesn't make any distinction on the operation type (signed or unsigned). The MSBs of this two operand are send to a 2-port XOR gate in order to make distinction between signed or unsigned operation. The other two inputs, SUM and COUT, are combined as reported in table 2.1 to make unsigned comparison between the operand (SUM is on 32 bit, so, to obtain a useful signal of width one, it is put in a OR and then inverted). The signed one are derived from these as reported in table 2.2.

A $\geq$ **B**	COUT
A $>$ **B**	[OR(SUM)] AND COUT
A $\leq$ **B**	NOT(COUT) OR (OR(SUM))
A $<$ **B**	NOT(COUT)
A $\neq$ **B**	OR(SUM)
A $=$ **B**	NOT[OR(SUM)]

Table 2.1: Unsigned comparison logic.

A $\geq$ **B**	[A(31) XOR B(31)] XOR GEU
A $>$ **B**	GE AND A!=B
A $\leq$ **B**	[A(31) XOR B(31)] XOR LEU
A $<$ **B**	LE AND A$\neq$B

Table 2.2: Signed comparison logic

2.2.3 Barrel Shifter

This block is used to execute the following operations:

- Shift Left Logical (SLL);
- Shift Right Logical (SRL);
- Shift Right arithmetical (SRA) ;
- Rotate Left (ROL);
- Rotate Right (ROR).

All this operations are completed in a single clock cycle thanks to the use of a barrel shifter architecture that can shift a data using only combinational circuits. It is implemented with a structure composed by a cascade of four multiplexers. It takes two input operands: one is the operand to shift/rotate, the other keep the number of positions to shift or to rotate. In figure 2.6a the flow of the operations is shown.

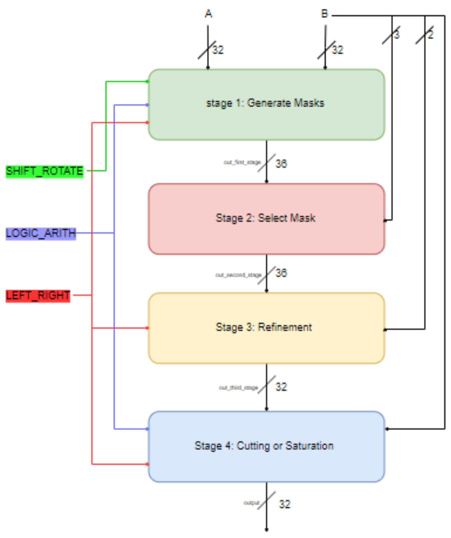
The first stage of the circuit is dedicated to the creation of the masks. These are built with the same basic block, reported in figure 2.6b, which, based on the received control signals, handle the signal A to generate the correct masks. It has been chosen to create 8 different mask, and to do this the signal is extended on 36-bits.

The second stage (figure 2.6c) takes in input all the masks generated on the first stage and, with the bits 2, 3 and 4 of the second input operand B, select the correct one. All the signal are still on 36 bits.

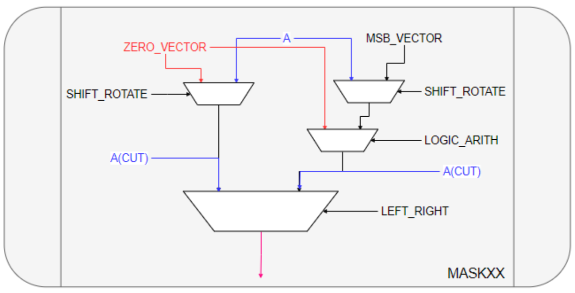
The third stage (figure 2.6d) does the remaining work on the selected mask. To select the correct refinement, the multiplexer reads three signals: LEFTRIGHT control and the two least significant bits of the second operand B.

The fourth stage is aim to correctly manage the saturations. The operands are on 32 bits, but rotation and shifting operation on the operand A can be expressed only on 5 bit, so, if a value on operand B greater than 32 is detected, can happens two things:

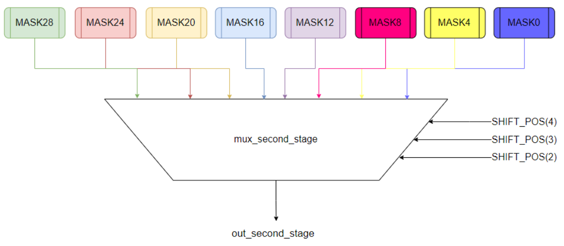
- In case of a Rotate operation: are taken only the five least significant bit of the operand B;
- In case of a Shift operation: the output of the barrel shifter is saturate to ones in case of an arithmetic right shift, otherwise to zeros.



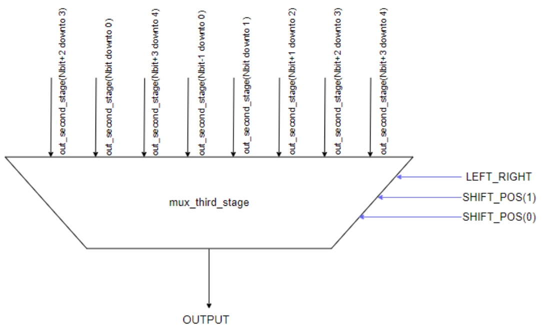
(a) Stage flow.



(b) Masks generation.



(c) Second stage.


out_second_stage(Nbit+2 downto 3)
out_second_stage(Nbit downto 0)
out_second_stage(Nbit+3 downto 4)
out_second_stage(Nbit-1 downto 0)
out_second_stage(Nbit downto 1)
out_second_stage(Nbit+1 downto 2)
out_second_stage(Nbit+2 downto 3)
out_second_stage(Nbit+3 downto 4)
mux_third_stage
LEFT_RIGHT
SHIFT_POS(1)
SHIFT_POS(0)
OUTPUT

(d) Third stage.

Figure 2.6: Shifter diagrams.

2.3 Multiplier

In order to compute multiplication by hardware, a multiplier based on the booth's algorithm is adopted. The used adders is the same already implemented in the ALU block (see subsection 2.2.1), this because the algorithm requires a long chain of adders (in our case, for 32-bit operands, fifteen adder in cascade are used), and it could be very slow if other slower adders are used (as RCA). Due to the length of this adders chain, the initial critical path of this block was about four times larger than the ALU's one, so, to making the delay comparable it has been split in four stages. This reduce the multiplication's latency, but allows to use a frequency almost equal to that would be had without this component. In order to reduce the general power consumption, these registers are enabled only when the multiplication is effectively needed.

2.4 Branch Comparator

The branch comparator is a combinational component that receives a test condition from the CU and a word to test. The condition can be: never, always, equal to zero, not equal to zero. If the condition is verified in the word to test an output is rised.

This component is used in the ID stage to drive the selection input of the MUX that send the correct address to the BPU (figure 2.1), so if the condition is not verified the send address is PC+4 otherwise if the condition is verified the correct address is that computed by the dedicated hardware.

2.5 Register File

The used RF has two output ports and one writing port, these numbers are sufficient to avoid any structural hazard due to this component.

A special mention have to be made for registers 0 and 31. The first is hardwired to zero, this is necessary to ensure that it cannot be changed even after a program error, in this way there is always a register that can be used to store an immediate value in a register, to move a register into another, or to use a direct addressing mode for load/store. The second is used as Link Register, this means that when a subroutine call occurs (thanks to an instruction JAL o JALR) the return address is saved in this register, so that you can return from the subroutine by performing a jump to the address contained in this register.

CHAPTER 3

Sequential units

As mentioned in chapter 1 and illustrate in figure 1.1, there are three totally independent units that drive the datapath:

- The CU, which send to the datapath the correct signals in order to carry out a specific instruction;
- The FU, to detect a data dependence and forward the data between the pipeline stages;
- The HDU, so that the hazards are automatically avoided.

3.1 Control Unit

The CU is developed using a mixture of the micro-programmed and the hardwired techniques. It is based on two look-up-table, the first is used when a register-to-register instruction is recognized and the other is used when the instruction to decode is an immediate or a jump instruction (the format and the meaning difference of the three types of instruction and how to distinguish these is explained in the appendix A). The LUTs are addressed directly by the FUNC field of the instruction in the first case, or by the OPCODE field in the case of an I-type instruction. This is very similar to the micro-programmed approach but it is important to notice that since there is no ”micro-program-counter”, it can be synthesize as a combinational circuit by the synthesizer. In this way the advantages of the two types of implementation are exploited: on the one hand it is very simple to join new instructions, to modify the existing ones, or to add new outputs, on the other hand it keep a lower area usage, the simplicity and speed of a cobinational circuit, and it is well suited to be used in a RISC pipeline context. For the correct timing of the signals within the pipeline, registers are used outside the control unit as you can see in figure 1.1, this to ensure that it is a completely combinational block.

3.2 Forwarding Unit

The FU is designed as a standalone block and it is completely independent from the control unit. It receives the instructions in execution in each stage of the pipeline from the ID stage to the WB stage (see figure 1.1) and according to these it controls the selection inputs of the forwarding MUXes (see figure 2.1). In particular there are three case of forwarding:

1. An instruction in the decode stage needs a datum that has been previously calculated by another instruction which has yet to finish the WB phase, or an instruction in the decode stage needs a datum that has already been read from the memory but that has yet to finish the WB phase.

In the case in which the load instruction has still to read the memory (this happens when the instruction that requires the datum is the immediately following one) there are two possibilities: if the immediately following instruction is a store that has to save the same datum that will come out from the memory it is possible to exploit the forwarding in the next clock cycle (see case 3), if instead the immediately following instruction is not a store or is a store but the datum is needed to compute the address to write on it is not possible to exploit the forwarding and the HDU will intervene inserting a stall in the execute stage and blocking all the precedent stages of the pipeline (see chapter 3.3).

2. An instruction in the decode stage needs register 31 (link register) and previously there was a JAL or JALR instruction that has yet to leave the pipeline. At first this may appear to be a special case of the previous one but it is important to note that it must be distinguished from it because in this case it is not possible to read the destination register of all the instructions in the pipeline and compare it with the source registers of the decoded instruction, but you must explicitly recognize the opcode of the subroutine call instructions, and furthermore in this case it is not possible to forward the data from the ALU output or from the DRAM output, but it must be taken directly from the registers that propagate the program counter from the IF stage to the WB stage (see figure 2.1).

3. As already mentioned, if a store must write immediately the same datum read by a load, it is possible to exploit the forwarding when the store is in the execute stage, in fact at this time the value to be written is already out of memory and it can be reported at the data input.

We also remember that the first two cases can occur for both the source registers and therefore both must be checked. Moreover if the data dependence refers to the register 0 the forwarding must not intervene because assuming that a hypothetical program tries to write, without any consequence, in the register 0, its subsequent reading must return zero in any time one tries to read it.

3.3 Hazard Detection Unit

The HDU is the block that has the task of controlling the progress of the pipeline stages, blocking some of them and inserting a stall if a hazard is detected. The specific cases in which it intervenes and how it do it are the following:

1. As already explained in chapter 3.2, if there is a data dependency between a load instruction and its next one, and the latter is not a store that have to write the datum just read, a data hazard occurs. This condition is verified in the decode stage, comparing the instruction with that in the execute stage, and in the event that a hazard is detected the program counter, all IF/ID registers, and all ID/EXE registers are disabled, and the registers in the EXE stage that propagate the control unit outputs are reset. The latter operation corresponds to the insertion of a stall that is a NOP instruction between the load and the next instruction, in this way an additional clock cycle is needed but once the data will be extracted from the memory it will be possible to use the forwarding to bring it back to the ALU input.

2. If a multi-cycle operation (i.e. a MULT or a MULTI) comes in the execute stage, a structural hazard occurs until it can leave this stage. This is the only hazard that requires a sequential circuit to be detected, in fact a counter starts to count from zero when the instruction reach the EXE stage, and as long as this count does not reach a value equal to the number of cycles required for complete the instruction all the IF/ID, ID/EXE, and EXE/MEM registers and the program counter are disabled, and bubbles are introduced in the MEM stage.

3. The last case is a control hazard and it occurs when the BPU makes a misprediction. When this happens a wrong instruction is entered in the fetch stage, so the HDU intervene substituting the mispredicted instruction with a NOP.

CHAPTER 4

Memories

Both the memorie (instruction RAM and data RAM) are described in VHDL only for simulations, because they are not synthesizable like a normal RAM with the used library and they are designed to be connected from the outside. The two addresses are overlapped, so both start from zero and an address for the IRAM corresponds to a different data respect to the same address but for the DRAM. Both are designed to have the same reading behavior and the same internal organization structure, but they have the difference that IRAM does not have a data input and the program is already inside it at the beginning of the simulation (with one of the modes that will be explained below), while the DRAM has a data input and a synchronous write enable to allow writing in runtime. So, here only the common features will be explained, while as regards the writing processes there will be two separate sections below.

The internal structure is divided in cell of 8-bit in order to allow the writing or the reading of a single byte, an half-word (two bytes), or a whole word (four bytes). Once send the address of a byte to the memory, it will output not only the corresponding byte, but also the three consecutive bytes, concatenated to form a single word in which the byte with the highest address is the most significant. Then it will be up to the datapath controlled by the CU to choose whether to take only one, two or all four bits and in the first two cases also whether to extend with or without sign, based on the type of load required by the program. Since there are no caches for data or for instructions which require to divide the memories into pages, it has been made possible to carry out this type of reading even for non-word-aligned addresses. In this way there is the advantage of having a better management of data and addresses in the DRAM, being able to exploit all the cells without leaving unused spaces.

Another feature of the used memories is the bit-width of the address inputs, that is always of 32-bit despite the physical size that you choose to use. If an attempt to read one or more bytes outside the physical size is made, these bytes will be read as if they were zeroes. This allows first of all a direct connection to the DLX without having open outputs, and furthermore in this way there is no risk that if following an error the address exceeds the physical size, the extra bits will be discarded reading an incomplete address (for example a memory with a 8-bit address would not be able to distinguish the addresses 0x00000000 and 0x00000100).

4.1 IRAM

We have used two different IRAM to testing the DLX, the first called *IRAM_test* was used during the development of the various units and consists of a list of programs already written, selectable by modifying a variable in the VHDL, which have the sole purpose of testing a certain feature. This allowed to quickly retest the correct behavior of the various parts each time a change was made, without perform several times the procedure required by the other IRAM.

The second IRAM, much more versatile, was used once the processor was completed to test the execution of more complicated programs. It allows to execute a program whose instructions have been previously written in a text file, they must be formatted in hexadecimal using the ascii coding and inserting a line break after each instruction, therefore to execute a program the following steps must be performed:

1. The program must be written in assembly language using the supported instructions, remember that it is not necessary to insert NOP to avoid hazards or dependencies, thanks to FU (chapter 3.2) and HDU (chapter 3.3).

2. The assembly code must be assembled using the assembler *dlxasm.pl*, this will generate the machine code but it is not yet in the required format.

3. The obtained code must be processed by the *conv2memory* script, which converts the machine code into the required format.

4. Finally the file thus obtained can be moved to the modelsim project folder, where it will be read when the simulation starts.

4.2 DRAM

The DRAM, unlike the IRAM, must be able to be written during the execution of the program, we also remember that it was chosen to adopt a little endianness, and that it must be possible to write arbitrarily, starting from the address indicated, all four bytes of the input word, only the two least significant bytes of the input word, or only the least significant byte of the input word. Moreover if you want to write only a byte or a half-word, the following three or two bytes respectively must not be modify. To do this it has been made possible to tell the memory by a special write enable, whether on the rising edge of the clock it has to memorize a word, a half-word, a byte, or if it doesn't have to do any writing. In all the first three cases the least significant byte is written to the indicated address, in addition to this in the first two cases also the second least significant byte is written to the address following the one indicated, finally in the case of a word writing also the third byte is written to the again following address and the most significant byte is written to the indicated address plus three.

Being a completely volatile memory, it goes without saying that it is not possible to initialize variables or define constants in the source code (unless you do it using a series of instructions), but it is a limit of the used memory and not of the developed DLX, in fact it would be totally possible to connect the data input and the address input of different types of memory (for example a ROM, an EEPROM and a RAM) together and exploit the extra bits of the address to enable only one memory so that they are not overlapped and corresponde to different address ranges. The data outputs can also be connected together if the memory supports the three-state output, otherwise a mux and a decoder are required to select the output here too based on the extra bits of the address.

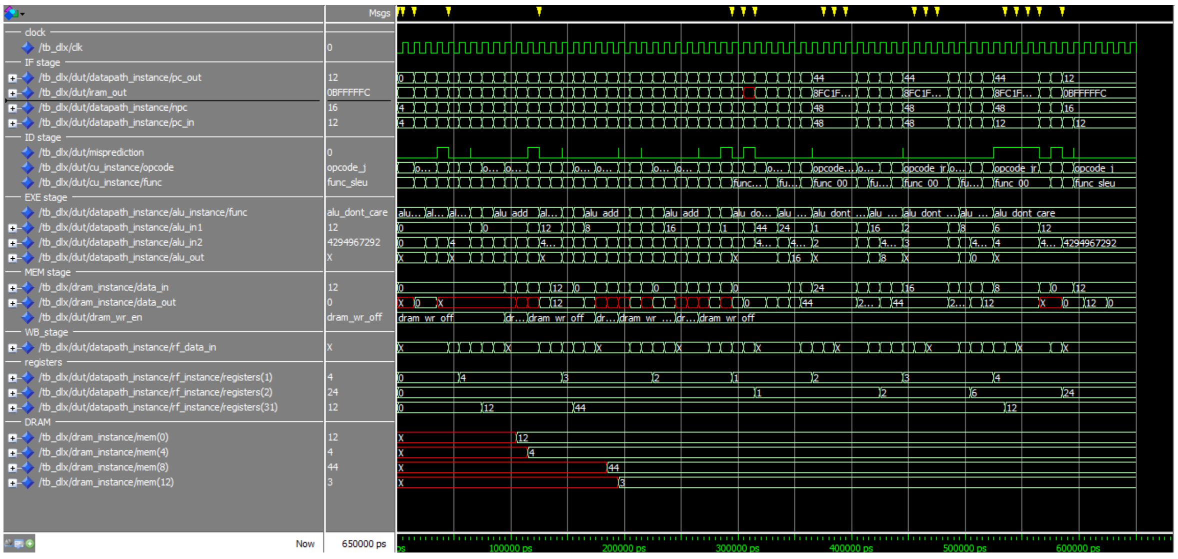
CHAPTER 5

Testbench

In the testbench the DLX, the IRAM, and the DRAM are connected together as shown in figure 1.1. Furthermore the clock signal is generated and a reset pulse is send to initialize all the sequential units. In the VHDL file is also possible to set the BPU parameters (TAG field size, SET field size, and the number of lines for each SET) the DRAM size, the IRAM size, and which of the two IRAM (see chapter 4.1) will be read.

In order to show most of the implemented features, it is possible to see in the figure 5.1 the result of a simulation in which a program for recursive factorial calculation is executed, the number to compute is passed by R1 and the result is taken from R2 (see appendix B to see the assembly code). Even if it is not possible to see the details there are some important things that you can appraise: first of all the value of the registers and the memory, in fact at the beginning in R1 is put the value to compute (in this case 4), and the recursion will overwrite this value each time with N-1 in order to compute the factorial of this value. Before overwriting R1, its value is saved in the stack (which begin at address 0) together with the return address (R13). Once N-1 becomes equal to 1 the first value is returned and R2 is set, the values saved in the stack begin to pop one by one and to multiply and accumulate with R2, until there are no more pushed numbers and the subroutine returns to the called instruction. Another interesting thing to notice is the misprediction signal, it is high every time a branch or a jump is executed for the first time, but the following time they are executed the BPU predict correctly the target address, unless the branch condition is not more verified or in the case of a JR the address to return is changed.

In this simulation occurs not only data dependencies forwarded thanks to the FU but also hazards due to mispredictions and to multi-cycle multiplications, in particular between 500 ns and 600 ns there are a situation in which both these types of hazard occurs at the same time and the HDU correctly handles the pipeline stages avoiding every type of conflict.



```
                                                       Msgs
— clock
  /tb_dlx/clk                                          0
— IF stage
  /tb_dlx/dut/datapath_instance/pc_out                 12
  /tb_dlx/dut/iram_out                                 0BFFFFFC
  /tb_dlx/dut/datapath_instance/npc                    16
  /tb_dlx/dut/datapath_instance/pc_in                  12
— ID stage
  /tb_dlx/dut/misprediction                            0
  /tb_dlx/dut/cu_instance/opcode                       opcode_j
  /tb_dlx/dut/cu_instance/func                         func_sleu
— EXE stage
  /tb_dlx/dut/datapath_instance/alu_instance/func      alu_dont_care
  /tb_dlx/dut/datapath_instance/alu_in1                12
  /tb_dlx/dut/datapath_instance/alu_in2                4294967292
  /tb_dlx/dut/datapath_instance/alu_out                X
— MEM stage
  /tb_dlx/dram_instance/data_in                        12
  /tb_dlx/dram_instance/data_out                       0
  /tb_dlx/dut/dram_wr_en                               dram_wr_off
— WB_stage
  /tb_dlx/dut/datapath_instance/rf_data_in             X
— registers
  /tb_dlx/dut/datapath_instance/rf_instance/registers(1)   4
  /tb_dlx/dut/datapath_instance/rf_instance/registers(2)   24
  /tb_dlx/dut/datapath_instance/rf_instance/registers(31)  12
— DRAM
  /tb_dlx/dram_instance/mem(0)                         12
  /tb_dlx/dram_instance/mem(4)                         4
  /tb_dlx/dram_instance/mem(8)                         44
  /tb_dlx/dram_instance/mem(12)                        3

                                               Now     650000 ps
```

Figure 5.1: Simulation in which a recursive factorial calculation program is runned.

CHAPTER 6

Synthesis

The top level entity has three parameters, all related to the BPU cahche, which must be set before starting the synthesis. In this design, to achieve a good hit ratio without exceeding in terms of area, they have been chosen:

- BPU_TAG_FIELD_SIZE equal to eight. This specify the bit width of the TAG field;
- BPU_SET_FIELD_SIZE equal to three. This configure the bit width of the SET field and consequently the number of sets that make up the set-associative cache, in this case eight set are created;
- BPU_LINES_PER_SET equal to four. This set the number of lines for each set of the cache.

The current design is analyzed taking into account three topic: Area, Delay and Power. First of all the longest path between any two registers is found, it is highlighted in figure 6.1 and it is possible to see the complete report in listing E.1. Thanks to the use of the function *custom_report* used in the script D.1, it is found that all the possible paths between the input registers of the EXE stage and the BPU are critical paths. The value thus found is set as timing constraint for combinational and sequential paths.

In order to achieve good results in terms of power consumption, an Ultra compile procedure with clock gating features is developed. The obtained results are reported in tables 6.1 and 6.2 in which also the power consumption and the used area of the most important blocks are reported in order to evaluate their impact on the architecture.

As further step, the area and power consumption variations based on the time constraint are analyzed. For this goal two different scripts are used in the Design Vision environment. For each of this script, five simulations are performed with the tune of the WCP (Worst critical path) parameter:

1. WCP = 20 ns, a very large timing constraint, so it doesn't have any impact on the synthesis;
2. WCP = 10 ns, it is still a large timing, but some effect can appears;
3. WCP = 5 ns, its effect became greater;
4. WCP = 3 ns, very near to the minimum delay, large effect on synthesis;
5. WCP = 2.4 ns, really strong timing constraint, the synthesizer try to do the best as possible to satisfy all the given constraint.

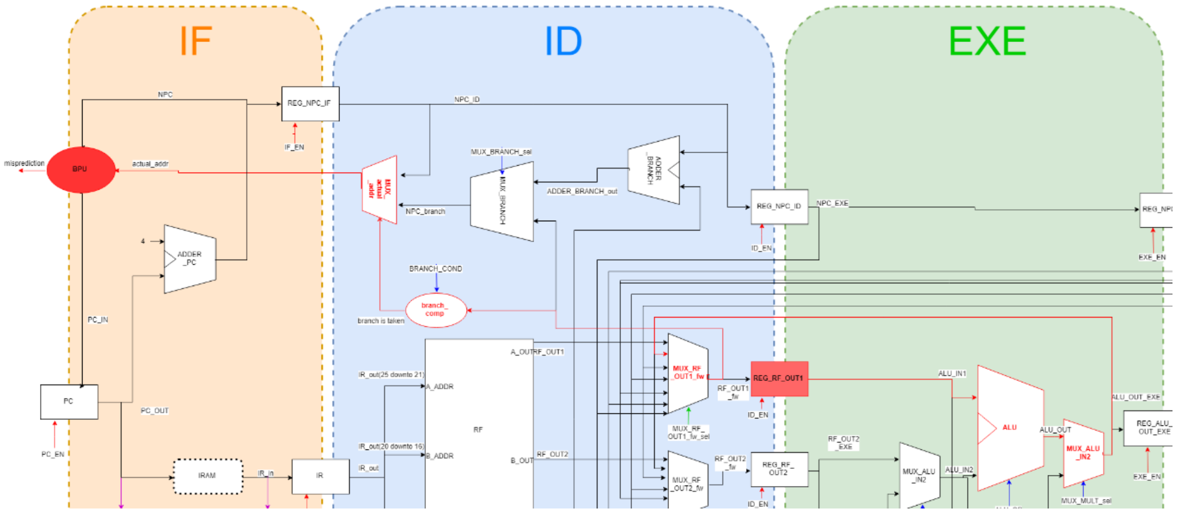


Figure 6.1: Critical path.

Component	Switch Power (µW)	Internal Power (µW)	Leakage Power (nW)	Total Power (µW)	%
DLX	878.851	2.42e+03	4.93e+05	3.79e+03	100.0
CU	9.263	5.209	3.01e+03	17.484	0.5
FU	12.538	12.389	4.43e+03	29.354	0.8
HDU	3.046	5.149	1.40e+03	9.591	0.3
BPU	166.370	353.422	1.61e+05	680.299	18.0
Datapath	772.068	2.06e+03	4.72e+05	3.31e+03	87.4
RF	31.903	209.550	1.16e+05	357.579	9.4
Mult	63.138	45.486	8.64e+04	194.975	5.1
ALU	72.408	66.170	4.48e+04	183.377	4.8
Logic_block	8.734	7.493	3.59e+03	19.819	0.5
Sparse_tree_adder	30.224	36.610	1.09e+04	77.709	2.1
Barrel_shifter	21.859	13.745	2.52e+04	60.768	1.6

Table 6.1: Power consumption of the entire DLX and of the most important blocks.

Component	Combinational Area	Non-combinational Area	Total Area	%
DLX	87.2480	563.3880	29000.3846	100.0
CU	117.8380	0.0000	117.8380	0.4
FU	184.3380	0.0000	184.3380	0.6
HDU	49.2100	10.6400	64.6380	0.2
BPU	2032.2400	7229.8802	9736.1323	33.6
datapath	15.1620	0.0000	27968.5706	96.4
RF	1933.2881	5277.4402	7359.1562	25.4
MULT	155.8760	511.7840	4808.4820	16.6
ALU	215.4600	0.0000	2280.9500	7.9
Barrel_shifter	20.4820	98.4200	1387.1900	4.8
Logic_block	153.2160	0.0000	153.2160	0.5
Sparse_tree_adder	57.7220	0.0000	525.0840	1.8

Table 6.2: Area of the whole DLX and of the most important blocks.

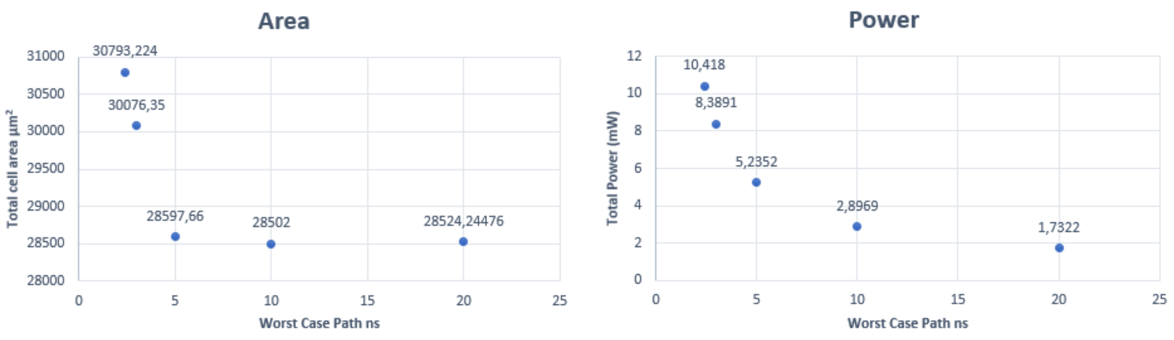


Figure 6.2: Trends of Area and Power with timing constraint variation under fixed area constraint.

6.1 Fixed area constraint analysis

The used script is the listing D.2, it give a fixed area constraint with the only variation of the WCP parameter. The results obtained from this first analysis are reported in the two graphs of figure 6.2. It is possible to see that, when the timing constraint is large enough, area and power have a pretty low value, whereas, with the decreasing of the delay, they become bigger and bigger.

6.2 Fixed power constraint analysis

This time, the power constraint is fixed and, as before, only the WCP parameter is changed between the synthesis. Since there are no constraint on the area, the synthesizer is free to allocate as much as it wants. The script used for this analysis is reported in listing D.3. The obtained results are shown in figure 6.3. As before, when the delay can be large, area and power value are low, while when the delay is reduced, they increase.

6.3 Consideration

It is possible to see that the result provided by the synthesis under fixed power constraint has a slightly better behaviour in terms of area occupation than the synthesis used for the final design. This is due

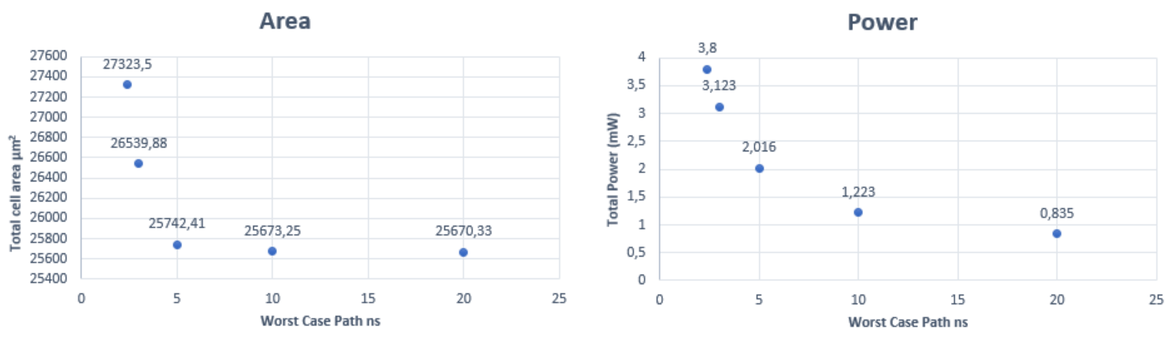


Figure 6.3: Trends of Area and Power with timing constraint variation under fixed power constraint.

by the Compile Ultra option *no_autoungroup* used in script D.1. This option forces the synthesizer to keep the hierarchy defined by the user and prevents any possible optimization between two different components. It was chosen to do this because in this way it was possible to get a better and deep analysis of the design.

CHAPTER 7

Place and Route

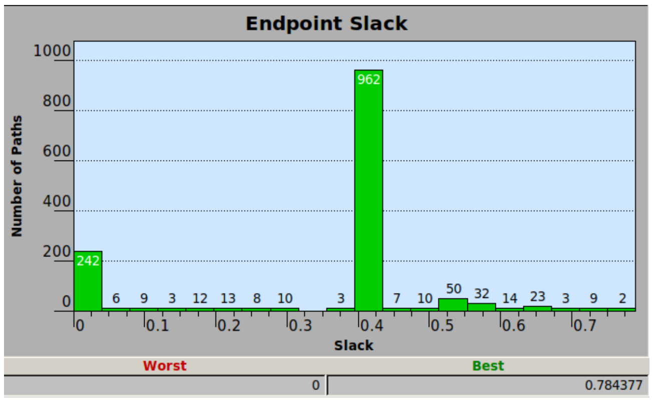
To conclude, the place and route in Innovus environment is performed. The SDC file and verilog netlist are extracted from the synthesis runned with the script of listing D.1 and the files default.view and DLX.globals are set properly: the first contains the MMMC file, the latter specified the design and the libraries.

In figure 7.2 it is possible to see an overview of the place and route. After this operation it is possible to obtain some information about the geometry and the timing of the routed design. In table 7.1 are collect some measurements on the layout area, and it is possible to compare this result with the one obtained during the synthesis phase: the result are quite similar.

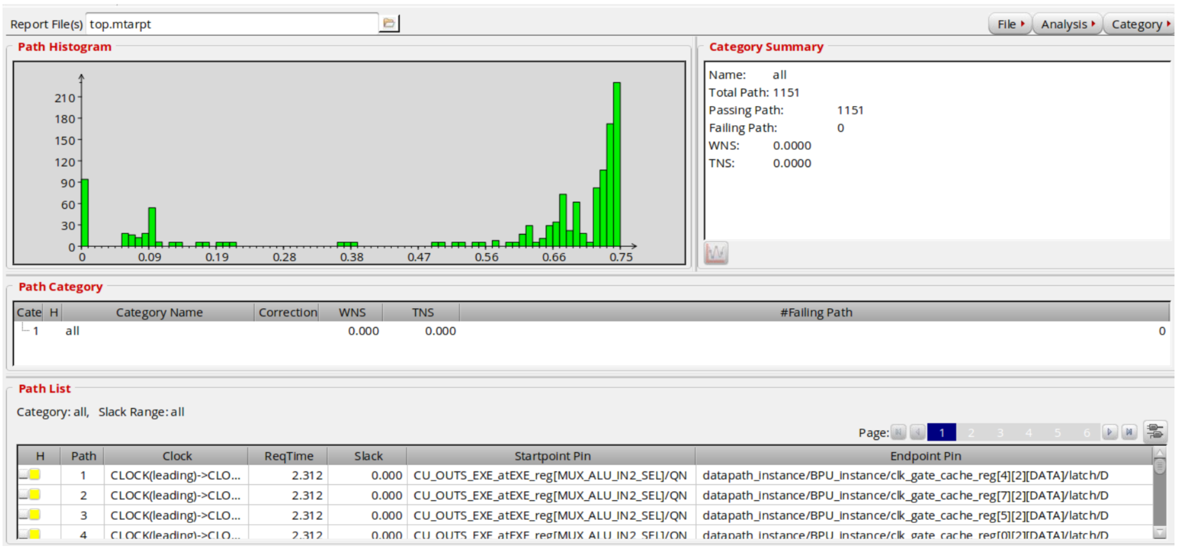
As regards the timing measurement, in figure 7.1a and 7.1b it is possible to see how the slack distribution is different in the two cases. In particular, the critical path is quite the same for both the analysis, always a path that starts from one of the EXE registers and ends up in the BPU unit, but, after the routing phase, some paths are moved to an higher value than the one which they have after the synthesis.

Module	**Gates**	**Cells**	**Area**
DLX	36421	13156	29064,5
FU	230	166	183,8
CU	150	114	120,2
BPU	12285	3503	9803,4
Datapath	35096	12598	28007,1
ALU	2818	1663	2249
RF	9266	2669	7394.5
MULT	5932	3188	4734,3
BARREL_SHIFTER	1718	956	1371,5
SPARSE_TREE_ADDER	649	334	518,4
LOGIC_BLOCK	186	190	148,7

Table 7.1: Area extracted by post-routing analysis.



(a) Slack distribution achieved after synthesis.



Name: all
Total Path: 1151
Passing Path: 1151
Failing Path: 0
WNS: 0.0000
TNS: 0.0000

Cate	H	Category Name	Correction	WNS	TNS	#Failing Path
1		all		0.000	0.000	0

Category: all, Slack Range: all

H	Path	Clock	ReqTime	Slack	Startpoint Pin	Endpoint Pin
🟨	1	CLOCK(leading)->CLO...	2.312	0.000	CU_OUTS_EXE_atEXE_reg[MUX_ALU_IN2_SEL]/QN	datapath_instance/BPU_instance/clk_gate_cache_reg[4][2][DATA]/latch/D
🟨	2	CLOCK(leading)->CLO...	2.312	0.000	CU_OUTS_EXE_atEXE_reg[MUX_ALU_IN2_SEL]/QN	datapath_instance/BPU_instance/clk_gate_cache_reg[7][2][DATA]/latch/D
🟨	3	CLOCK(leading)->CLO...	2.312	0.000	CU_OUTS_EXE_atEXE_reg[MUX_ALU_IN2_SEL]/QN	datapath_instance/BPU_instance/clk_gate_cache_reg[5][2][DATA]/latch/D
🟨	4	CLOCK(leading)->CLO...	2.312	0.000	CU_OUTS_EXE_atEXE_reg[MUX_ALU_IN2_SEL]/QN	datapath_instance/BPU_instance/clk_gate_cache_reg[0][2][DATA]/latch/D

(b) Slack distribution achieved after place and route.

Figure 7.1: Comparison between the slack distribution post-synthesis and post-place and route.

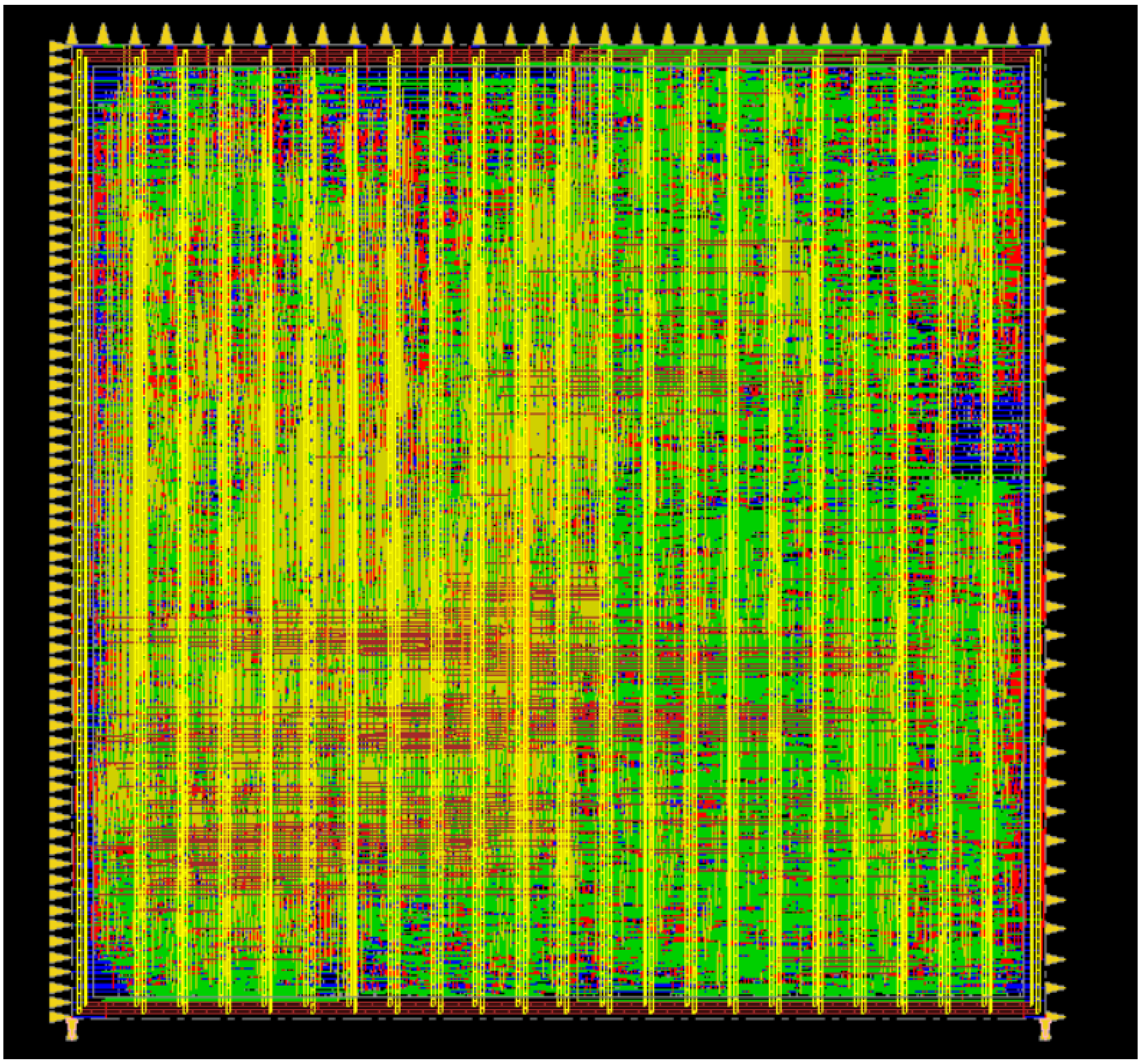


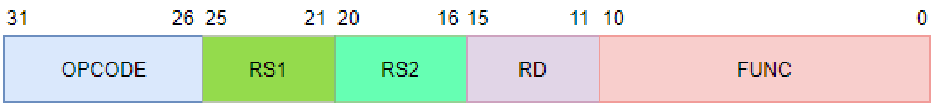
Figure 7.2: Overview of the physical layout obtained after the place and route operation.

APPENDIX A

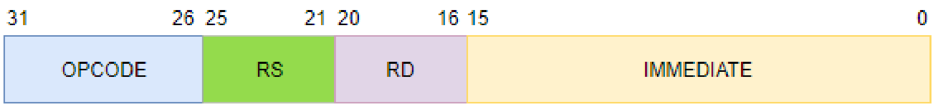
Instruction Set

The instruction is expressed on a word of 32 bit. To recognize the different instructions, the six most significant bits are read. These bits compose the OPCODE field. Basing on this, it is possible to recognize three different types of instruction:

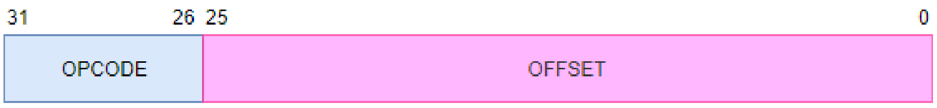
- If the OPCODE is equal to zero it is an R-type instruction. This corresponds to a register-to-register operation. In this case also the six least significant bits are read, which compose the FUNC field, in order to understand the required operation. The numbers of the registers to use for these instructions are taken from the fields shown in the figure A.1a.
- If the OPCODE is different from zero, it is an I-type or a J-type instruction and the operation is read directly by this field. The other fields are different in the two cases:
  - If the OPCODE is that of a Jump or a Jump And Link (see the list below), the instruction is a J-type and only the address offset is specified in the instruction as shown in figure A.1c.
  - In all the other cases we have an I-type instruction. This means that it is a register-to-immediate operation, a conditional branch, or a load/store operation, so also the immediate value must be specified. In figure A.1b are reported the format also for this type of instruction.



(a) R-type.



(b) I-type.



(c) J-type.

Figure A.1: Format and relative fields of the three types of instruction.

R-type instructions

- **SLL Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1 << Rs2$
  FUNC=0x04

- **SRL Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1 >> Rs2$
  FUNC=0x06

- **SRA Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1(31)^{Rs2}\ \#\#\ (Rs1 >> Rs2)\_Rs2..0$
  FUNC=0x07

- **ROL Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1((31 - Rs2)\_downto\_0)\ \#\#\ Rs1(31\_downto\_(32 - Rs2))$
  FUNC=0x08

- **ROR Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1((Rs2 - 1)\_downto\_0)\ \#\#\ Rs1(31\_downto\_Rs2)$
  FUNC=0x09

- **MULT Rd, Rs1, Rs2** (4 clock cycles required)
  $Rd \leftarrow Rs1 * Rs2$
  FUNC=0x1E

- **ADD Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1 + Rs2$
  FUNC=0x20

- **ADDU Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1 + Rs2$
  FUNC=0x21

- **SUB Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1 - Rs2$
  FUNC=0x22

- **SUBU Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1 - Rs2$
  FUNC=0x23

- **AND Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1\ and\ Rs2$

FUNC=0x24

- **OR Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1\ or\ Rs2$
  FUNC=0x25

- **XOR Rd, Rs1, Rs2**
  $Rd \leftarrow Rs1\ xor\ Rs2$
  FUNC=0x26

- **NOT Rd, Rs1, Rs2** (Rs2 neglected)
  $Rd \leftarrow not\ Rs1$
  FUNC=0x27

- **SEQ Rd, Rs1, Rs2**
  $if\ Rs1 = Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  FUNC=0x28

- **SNE Rd, Rs1, Rs2**
  $if\ Rs1\ \neq Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  FUNC=0x29

- **SLT Rd, Rs1, Rs2**
  $if\ Rs1 < Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are signed.
  FUNC=0x2A

- **SGT Rd, Rs1, Rs2**
  $if\ Rs1 > Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are signed.
  FUNC=0x2B

- **SLE Rd, Rs1, Rs2**
  $if\ Rs1 \leq Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are signed.
  FUNC=0x2C

- **SGE Rd, Rs1, Rs2**
  $if\ Rs1 \geq Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are signed.
  FUNC=0x2D

- **SLTU Rd, Rs1, Rs2**
  $if\ Rs1 < Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$

All operands are unsigned.
FUNC=0x3A

- **SGTU Rd, Rs1, Rs2**
  $if\ Rs1 > Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are unsigned.
  FUNC=0x3B

- **SLEU Rd, Rs1, Rs2**
  $if\ Rs1 \leq Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are unsigned.
  FUNC=0x3C

- **SGEU Rd, Rs1, Rs2**
  $if\ Rs1 \geq Rs2\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are unsigned.
  FUNC=0x3D

I-type Instructions

- **ADDI Rd, Rs, #imm**
  $Rd \leftarrow Rs + \#imm$
  OPCODE=0x08

- **ADDUI Rd, Rs, #imm**
  $Rd \leftarrow Rs + \#imm$
  OPCODE=0x09

- **SUB Rd, Rs, #imm**
  $Rd \leftarrow Rs - \#imm$
  OPCODE=0x0A

- **SUBUI Rd, Rs, #imm**
  $Rd \leftarrow Rs - \#imm$
  OPCODE=0x0B

- **ANDI Rd, Rs, #imm**
  $Rd \leftarrow Rs\ and\ \#imm$
  OPCODE=0x0C

- **ORI Rd, Rs, #imm**
  $Rd \leftarrow Rs\ or\ \#imm$
  OPCODE=0x0D

- **XORI Rd, Rs, #imm**
  $Rd \leftarrow Rs\ xor\ \#imm$
  OPCODE=0x0E

- **SLLI Rd, Rs, #imm**
  $Rd \leftarrow Rs << \#imm$
  OPCODE=0x14

- **NOP**
  No operation
  OPCODE=0x15

- **SRLI Rd, Rs, #imm**
  $Rd \leftarrow Rs >> \#imm$
  OPCODE=0x16

- **SRAI Rd, Rs, #imm**
  $Rd \leftarrow Rs(31)^{\#imm}\ \#\#\ (Rs1 >> \#imm)$
  OPCODE=0x17

- **SEQI Rd, Rs, #imm**
  $if\ Rs = \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  OPCODE=0x18

- **SNEI Rd, Rs, #imm**
  $if\ Rs\ \neq \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  OPCODE=0x19

- **SLTI Rd, Rs, #imm**
  $if\ Rs < \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operand are signed.
  OPCODE=0x1A

- **SGTI Rd, Rs, #imm**
  $if\ Rs > \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are signed.
  OPCODE=0x1B

- **SLEI Rd, Rs, #imm**
  $if\ Rs \leq \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are signed.
  OPCODE=0x1C

- **SGEI Rd, Rs, #imm**
  $if\ Rs \geq \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are signed.
  OPCODE=0x1D

- **ROLI Rd, Rs, #imm**
  $Rd \leftarrow Rs((31 - \#imm)\_downto\_0)\ \#\#\ Rs(31\_downto\_(32 - \#imm))$
  OPCODE=0x1E

- **RORI Rd, Rs, #imm**
  $Rd \leftarrow Rs((\#imm - 1)\_downto\_0)\ \#\#\ Rs(31\_downto\_\#imm)$
  OPCODE=0x1F

- **LB Rd, #imm(Rs)**
  $Rd <= sign\_extended\ \#\#\ M[\#imm + Rs]$
  OPCODE=0x20

- **LH Rd, #imm(Rs)**
  $Rd <= sign\_extended\ \#\#\ M[\#imm + Rs + 1]\ \#\#\ M[\#imm + Rs]$
  OPCODE=0x21

- **LW Rd, #imm(Rs)**
  $Rd <= M[\#imm+Rs+3]\ \#\#\ M[\#imm+Rs+2]\ \#\#\ M[\#imm+Rs+1]\ \#\#\ M[\#imm+Rs]$
  OPCODE=0x23

- **LBU Rd, #imm(Rs)**
  $Rd <= 0^{24}\ \#\#\ M[\#imm + Rs]$
  OPCODE=0x24

- **LHU Rd, #imm(Rs)**
  $Rd <= 0^{16}\ \#\#\ M[\#imm + Rs + 1]\ \#\#\ M[\#imm + Rs]$
  OPCODE=0x25

- **SB Rs, #imm(Rd)**
  $M[\#imm + Rd] = Rs(7\_downto\_0)$
  OPCODE=0x28

- **SH Rs, #imm(Rd)**
  $M[\#imm + Rd] = Rs(7\_downto\_0)$
  $M[\#imm + Rd + 1] = Rs(15\_downto\_8)$
  OPCODE=0x29

- **SW Rs, #imm(Rd)**
  $M[\#imm + Rd] = Rs(7\_downto\_0)$

$M[\#imm + Rd + 1] = Rs(15\_downto\_8)$
$M[\#imm + Rd + 2] = Rs(23\_downto\_16)$
$M[\#imm + Rd + 3] = Rs(31\_downto\_24)$
OPCODE=0x2B

- **SLTUI Rd, Rs, #imm**
  $if\ Rs1 < \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are unsigned.
  OPCODE=0x3A

- **SGTUI Rd, Rs, #imm**
  $if\ Rs1 > \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are unsigned.
  OPCODE=0x3B

- **SLEUI Rd, Rs, #imm**
  $if\ Rs1 \leq \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are unsigned.
  OPCODE=0x3C

- **SGEUI Rd, Rs, #imm**
  $if\ Rs1 \geq \#imm\ \ Rd \leftarrow 1;\ else\ Rd \leftarrow 0$
  All operands are unsigned.
  OPCODE=0x3D

- **MULTI Rd, Rs, #imm** (4 clock cycles required)
  $Rd \leftarrow Rs1 * \#imm$
  OPCODE=0x1E

J-type Instructions

- **J label**
  $PC \leftarrow PC + offset$
  OPCODE=0x02

- **JAL label**
  $R31 \leftarrow PC + 4;\ \ PC \leftarrow PC + offset$
  OPCODE=0x03

- **BEQZ Rs, label**
  $ifRs = 0\ \ PC \leftarrow PC + offset$
  OPCODE=0x04

- **BNEZ Rs, label**
  $ifRs \neq 0 \;\; PC \leftarrow PC + offset$
  OPCODE=0x05

- **JR Rs**
  $PC \leftarrow Rs$
  OPCODE=0x12

- **JALR Rs**
  $R31 \leftarrow PC + 4; \;\; PC \leftarrow Rs$
  OPCODE=0x13

APPENDIX B

Assembly programs

Listing B.1: factorial_recursive.asm

```
addui r1,r0,4       ;N
addui r30,r0,0      ;stack pointer
jal factorial
end_program:
j end_program

; parameter passed in r1, return value in r2

factorial:
seqi r2,r1,1            ;set r2 if N is 1
bnez r2,end_factorial   ;if r2 is set return 1, otherwise...
sw 0(r30),r31           ;save link register on the stack
sw 4(r30),r1            ;save N on the stack
addui r30,r30,8         ;update stack pointer
addui r1,r1,-1          ;compute N-1
jal factorial           ;compute factorial(N-1)
lw r1,-4(r30)           ;load N from the stack
lw r31,-8(r30)          ;load link register from the stack
addui r30,r30,-8        ;update stack pointer
mult r2,r2,r1           ;compute N*factorial(N-1)
end_factorial:
jr r31
```

APPENDIX C

VHDL

In this appendix are reported only the most important VHDL files. The trivial components like Full Adder, MUX, Register, and so on are omitted from here.

Listing C.1: 000-my_package.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.math_real.all;

package my_package is

    type std_logic_vector_intrange is array (integer range <>) of std_logic;    --standard logic
        ↪ vector with integer range

    --function that receives an integer greater or equal to 1 and returns its base-2 logarithm
        ↪ rounded up to the nearest greater (or equal) integer
    function log2_ceiling (N: positive) return natural;

    --operator xor between a std_logic_vector and a signle std_logic
    function "xor" (vector: std_logic_vector; one_bit: std_logic) return std_logic_vector;

    --nor all bits of a generic width std_logic_vector
    function nor_vector (vector: std_logic_vector) return std_logic;

end package my_package;

package body my_package is

    function log2_ceiling (N: positive) return natural is
    begin
        return natural(ceil(log2(real(N))));
    end function log2_ceiling;

    function "xor" (vector: std_logic_vector; one_bit: std_logic) return std_logic_vector is
        variable temp: std_logic_vector(vector'range);
    begin
        for i in vector'range loop
            temp(i):=vector(i) xor one_bit;
        end loop;
        return temp;
    end function "xor";

    function nor_vector (vector: std_logic_vector) return std_logic is
        variable temp: std_logic:='0';
    begin
        for i in vector'range loop
            temp:= temp or vector(i);
        end loop;
        return not temp;
    end function nor_vector;

end package body my_package;
```

Listing C.2: 001-DLX_package.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

package DLX_package is

    constant BYTE_SIZE : integer := 8;
    constant HALFWORD_SIZE : integer := 2*BYTE_SIZE;
    constant WORD_SIZE : integer := 4*BYTE_SIZE;
    constant DOUBLEWORD_SIZE : integer := 8*BYTE_SIZE;
    subtype BYTE_range is integer range BYTE_SIZE-1 downto 0;
    subtype BYTE1_range is integer range 2*BYTE_SIZE-1 downto BYTE_SIZE;
    subtype BYTE2_range is integer range 3*BYTE_SIZE-1 downto 2*BYTE_SIZE;
    subtype BYTE3_range is integer range 4*BYTE_SIZE-1 downto 3*BYTE_SIZE;
    subtype HALFWORD_range is integer range HALFWORD_SIZE-1 downto 0;
    subtype HALFWORD1_range is integer range 2*HALFWORD_SIZE-1 downto HALFWORD_SIZE;
    subtype WORD_range is integer range WORD_SIZE-1 downto 0;
    subtype DOUBLEWORD_range is integer range DOUBLEWORD_SIZE-1 downto 0;
    subtype byte is std_logic_vector(BYTE_range);
    subtype halfword is std_logic_vector(HALFWORD_range);
    subtype word is std_logic_vector(WORD_range);
    subtype doubleword is std_logic_vector(DOUBLEWORD_range);

    constant IF_STAGE   : integer := 0;
    constant ID_STAGE   : integer := 1;
    constant EXE_STAGE  : integer := 2;
    constant MEM_STAGE  : integer := 3;
    constant WB_STAGE   : integer := 4;
    constant N_STAGE    : integer := 5;

    --all possible R-type instruction
    type FUNC_type is (
        FUNC_00,
        FUNC_01,
        FUNC_02,
        FUNC_03,
        FUNC_SLL,           -- shift left logical
        FUNC_05,
        FUNC_SRL,           -- shift right logical
        FUNC_SRA,           -- shift right arithmetical
        FUNC_ROL,           -- rotate left
        FUNC_ROR,           -- rotate right
        FUNC_0A,
        FUNC_0B,
        FUNC_0C,
        FUNC_0D,
        FUNC_0E,
        FUNC_0F,
        FUNC_10,
        FUNC_11,
        FUNC_12,
        FUNC_13,
        FUNC_14,
        FUNC_15,
        FUNC_16,
        FUNC_17,
        FUNC_18,
        FUNC_19,
        FUNC_1A,
        FUNC_1B,
        FUNC_1C,
        FUNC_1D,
        FUNC_MULT,          -- multiplication (signed)
        FUNC_1F,
        FUNC_ADD,           -- addition (with overflow trap)
        FUNC_ADDU,          -- addition (without overflow trap)
        FUNC_SUB,           -- subtraction  (with overflow trap)
        FUNC_SUBU,          -- subtruction (without overflow trap)
        FUNC_AND,           -- bitwise and
        FUNC_OR,            -- bitwise or
        FUNC_XOR,           -- bitwise xor
        FUNC_NOT,           -- bitwise not
        FUNC_SEQ,           -- Set if equal to
        FUNC_SNE,           -- Set if not equal to
        FUNC_SLT,           -- set if less than (signed)
        FUNC_SGT,           -- set if greather than (signed)
        FUNC_SLE,           -- set if less than or equal to (signed)
        FUNC_SGE,           -- set if greater than or equal to
        FUNC_2E,
        FUNC_2F,
        FUNC_MOVI2S,        --NOT IMPLEMENTED
        FUNC_MOVS2I,        --NOT IMPLEMENTED
        FUNC_MOVF,          --NOT IMPLEMENTED
        FUNC_MOVD,          --NOT IMPLEMENTED
        FUNC_MOVFP2I,       --NOT IMPLEMENTED
        FUNC_MOVI2FP,       --NOT IMPLEMENTED
        FUNC_MOVI2T,        --NOT IMPLEMENTED
        FUNC_MOVT2I,        --NOT IMPLEMENTED
        FUNC_38,
        FUNC_39,
        FUNC_SLTU,          -- set if less than (unsigned)
        FUNC_SGTU,          -- set if greather than (unsigned)
        FUNC_SLEU,          -- set if less than or equal to (unsigned)
        FUNC_SGEU,          -- set if greather than or equal to (unsigned)
        FUNC_3E,
        FUNC_3F
    );

    --all possible I-type instruction
    type OPCODE_type is (
        OPCODE_RTYPE,       -- if the OPCODE is equal to this the instrution is a R-type
        OPCODE_01,
        OPCODE_J,           -- jump
        OPCODE_JAL,         -- jump and link
        OPCODE_BEQZ,        -- branch if equal to zero
        OPCODE_BNEZ,        -- branch if not equal to zero
        OPCODE_BFPT,        --NOT IMPLEMENTED
        OPCODE_BFPF,
        OPCODE_ADDI,        -- addition immediate (with overflow trap)
        OPCODE_ADDUI,       -- addition immediate (without overflow trap)
        OPCODE_SUBI,        -- subtraction immediate (with overflow trap)
        OPCODE_SUBUI,       -- subtraction immediate (without overflow trap)
        OPCODE_ANDI,        -- bitwise and immediate
        OPCODE_ORI,         -- bitwise or immediate
        OPCODE_XORI,        -- bitwise xor immediate
        OPCODE_LHI,         --NOT IMPLEMENTED
        OPCODE_RFE,         --NOT IMPLEMENTED
        OPCODE_TRAP,        --NOT IMPLEMENTED
        OPCODE_JR,          -- jump register
        OPCODE_JALR,        -- jump and link register
        OPCODE_SLLI,        -- shift left logical immediate
        OPCODE_NOP,         -- no operation
        OPCODE_SRLI,        -- shift right logical immediate
        OPCODE_SRAI,        -- shift right arithmetical immediate
        OPCODE_SEQI,        -- set if equal to immediate
        OPCODE_SNEI,        -- set if not equal to immediate
        OPCODE_SLTI,        -- set if less than immediate (signed)
        OPCODE_SGTI,        -- set if greather than immediate (signed)
        OPCODE_SLEI,        -- set if less than or equal to immediate
        OPCODE_SGEI,        -- set if greather than or equal to immediate
        OPCODE_ROLI,        -- rotate left immediate
        OPCODE_RORI,        -- rotate right immediate
        OPCODE_LB,          -- load byte (signed)
        OPCODE_LH,          -- load halfword (signed)
        OPCODE_22,
        OPCODE_LW,          -- load word
        OPCODE_LBU,         -- load byte (unsigned)
        OPCODE_LHU,         -- load halfword (unsigned)
        OPCODE_LF,          --NOT IMPLEMENTED
        OPCODE_LD,          --NOT IMPLEMENTED
        OPCODE_SB,          -- store byte
        OPCODE_SH,          -- store halfword
        OPCODE_2A,
        OPCODE_SW,          -- store word
        OPCODE_2C,
        OPCODE_2D,
        OPCODE_SF,          --NOT IMPLEMENTED
        OPCODE_SD,          --NOT IMPLEMENTED
        OPCODE_30,
        OPCODE_31,
        OPCODE_32,
        OPCODE_33,
        OPCODE_34,
        OPCODE_35,
        OPCODE_36,
        OPCODE_37,
        OPCODE_ITLB,        --NOT IMPLEMENTED
        OPCODE_39,
        OPCODE_SLTUI,       -- set if less than immediate (unsigned)
        OPCODE_SGTUI,       -- set if greather than immediate (unsigned)
        OPCODE_SLEUI,       -- set if less than or equal to immediate (unsigned)
        OPCODE_SGEUI,       -- set if greather than or equal to immediate(unsigned)
        OPCODE_MULTI,       -- multiplication immediate
        OPCODE_3F
    );

    subtype OPCODE_range    is integer range 31 downto 26;
    subtype REG1_range      is integer range 25 downto 21;
    subtype REG2_range      is integer range 20 downto 16;
    subtype REG3_range      is integer range 15 downto 11;
    subtype FUNC_range      is integer range 5 downto 0;
    subtype IMM_range       is integer range 15 downto 0;
    subtype J_OFFSET_range  is integer range 25 downto 0;

    constant OPCODE_SIZE    : integer := OPCODE_range'high-OPCODE_range'low+1;          --OPCODE
        ↪ field size
    constant RF_ADDR_SIZE   : integer := REG1_range'high-REG1_range'low+1;              --
        ↪ register address field size
    constant FUNC_SIZE      : integer := FUNC_range'high-FUNC_range'low+1;              --FUNC
        ↪ field size
    constant IMM_SIZE       : integer := IMM_range'high-IMM_range'low+1;                --IMM
        ↪ field size
    constant J_OFFSET_SIZE  : integer := J_OFFSET_range'high-J_OFFSET_range'low+1;      --OFFSET
        ↪ field size

    constant N_REG          : integer := 2**RF_ADDR_SIZE;

    constant N_RTYPE    : integer := 2**FUNC_SIZE;      --number of possible R-type instructions
    constant N_ITYPE    : integer := 2**OPCODE_SIZE;    --number of possible I-type instructions

    --type conversion functions for FUNC_type
    function to_integer (FUNC: FUNC_type) return natural;
    function to_std_logic_vector (FUNC: FUNC_type) return std_logic_vector;
    function to_func_type (FUNC_N: integer range 0 to N_RTYPE-1) return FUNC_type;
    function to_func_type (FUNC_FIELD: std_logic_vector(FUNC_SIZE-1 downto 0)) return FUNC_type;

    --type conversion functions for OPCODE_type
    function to_integer (OPCODE: OPCODE_type) return natural;
    function to_std_logic_vector (OPCODE: OPCODE_type) return std_logic_vector;
    function to_opcode_type (OPCODE_N: integer range 0 to N_ITYPE-1) return OPCODE_type;
    function to_opcode_type (OPCODE_FIELD: std_logic_vector(OPCODE_SIZE-1 downto 0)) return
        ↪ OPCODE_type;

    --function to get directly opcode, func or registers number from an instruction
    function get_opcode (INSTR: word) return OPCODE_type;
    function get_func (INSTR: word) return FUNC_type;
    function get_Rs1 (INSTR: word) return natural;
    function get_Rs2 (INSTR: word) return natural;
    function get_Rd (INSTR: word) return natural;

    --instruction encoding functions
    function INSTR_RTYPE (FUNC: FUNC_type; RS1,RS2,RD: integer range 0 to N_REG) return word;
    function INSTR_ITYPE (OPCODE: OPCODE_type; RS, RD: integer range 0 to N_REG; IMM: integer
        ↪ range -2**(IMM_SIZE-1) to 2**(IMM_SIZE-1)-1) return word;
    function INSTR_JTYPE (OPCODE: OPCODE_type; OFFSET: integer range -2**(J_OFFSET_SIZE-1) to
        ↪ 2**(J_OFFSET_SIZE-1)-1) return word;

end package DLX_package;

package body DLX_package is

    function to_integer (FUNC: FUNC_type) return natural is
    begin
        return FUNC_type'pos(FUNC);
    end function to_integer;

    function to_std_logic_vector (FUNC: FUNC_type) return std_logic_vector is
    begin
        return std_logic_vector(to_unsigned(to_integer(FUNC), FUNC_SIZE));
    end function to_std_logic_vector;

    function to_func_type (FUNC_N: integer range 0 to N_RTYPE-1) return FUNC_type is
    begin
            return FUNC_type'val(FUNC_N);
    end function to_func_type;

    function to_func_type (FUNC_FIELD: std_logic_vector(FUNC_SIZE-1 downto 0)) return FUNC_type
        ↪ is
    begin
        return to_func_type(to_integer(unsigned(FUNC_FIELD)));
    end function to_func_type;

    function to_integer (OPCODE: OPCODE_type) return natural is
    begin
        return OPCODE_type'pos(OPCODE);
    end function to_integer;

    function to_std_logic_vector (OPCODE: OPCODE_type) return std_logic_vector is
    begin
        return std_logic_vector(to_unsigned(to_integer(OPCODE), OPCODE_SIZE));
    end function to_std_logic_vector;

    function to_opcode_type (OPCODE_N: integer range 0 to N_ITYPE-1) return OPCODE_type is
    begin
            return OPCODE_type'val(OPCODE_N);
    end function to_opcode_type;

    function to_opcode_type (OPCODE_FIELD: std_logic_vector(OPCODE_SIZE-1 downto 0)) return
        ↪ OPCODE_type is
    begin
        return to_opcode_type(to_integer(unsigned(OPCODE_FIELD)));
    end function to_opcode_type;

    function get_opcode (INSTR: word) return OPCODE_type is
    begin
        return to_opcode_type(INSTR(OPCODE_range));
    end function get_opcode;

    function get_func (INSTR: word) return FUNC_type is
    begin
        return to_func_type(INSTR(FUNC_range));
    end function get_func;

    function get_Rs1 (INSTR: word) return natural is
    begin
        return to_integer(unsigned(INSTR(REG1_range)));
    end function get_Rs1;
    function get_Rs2 (INSTR: word) return natural is
    begin
        return to_integer(unsigned(INSTR(REG2_range)));
    end function get_Rs2;

    function get_Rd (INSTR: word) return natural is
    begin
        if get_opcode(INSTR)=OPCODE_RTYPE then
            return to_integer(unsigned(INSTR(REG3_range)));
        else
            return to_integer(unsigned(INSTR(REG2_range)));
        end if;
    end function get_Rd;

    function INSTR_RTYPE (FUNC: FUNC_type; RS1,RS2,RD: integer range 0 to N_REG) return word is
    begin
        return to_std_logic_vector(OPCODE_RTYPE) & std_logic_vector(to_unsigned(RS1,RF_ADDR_SIZE)
            ↪ ) & std_logic_vector(to_unsigned(RS2,RF_ADDR_SIZE)) & std_logic_vector(
            ↪ to_unsigned(RD,RF_ADDR_SIZE)) & "00000" & to_std_logic_vector(FUNC);
    end function INSTR_RTYPE;

    function INSTR_ITYPE (OPCODE: OPCODE_type; RS, RD: integer range 0 to N_REG; IMM: integer
        ↪ range -2**(IMM_SIZE-1) to 2**(IMM_SIZE-1)-1) return word is
    begin
        return to_std_logic_vector(OPCODE) & std_logic_vector(to_unsigned(RS,RF_ADDR_SIZE)) &
            ↪ std_logic_vector(to_unsigned(RD,RF_ADDR_SIZE)) & std_logic_vector(to_signed(IMM,
            ↪ IMM_SIZE));
    end function INSTR_ITYPE;

    function INSTR_JTYPE (OPCODE: OPCODE_type; OFFSET: integer range -2**(J_OFFSET_SIZE-1) to
        ↪ 2**(J_OFFSET_SIZE-1)-1) return word is
    begin
        return to_std_logic_vector(OPCODE) & std_logic_vector(to_signed(OFFSET,J_OFFSET_SIZE));
            ↪
    end function INSTR_JTYPE;

end package body DLX_package;
```

Listing C.3: 002-ALU_package.vhd

```
use work.my_package.all;        --for log2_ceiling

package ALU_package is

    --all possible ALU operation
    type ALU_OP_type is (
        ALU_DONT_CARE,  -- The operation don't care
        ALU_ADD,        -- Addition
        ALU_SUB,        -- Subtraction
        ALU_XOR,        -- Bitwise xor
        ALU_AND,        -- Bitwise and
        ALU_OR,         -- Bitwise or
        ALU_NOT,        -- Bitwise not
        ALU_SLL,        -- Shift left logical
        ALU_SRL,        -- Shift right logical
        ALU_SRA,        -- Shift right arithmetical
        ALU_ROL,        -- Rotate left
        ALU_ROR,        -- Rotate right
        ALU_SGE,        -- Set if greather than or equal to (SIGNED)
        ALU_SGEU,       -- Set if greather than or equal to (UNSIGNED)
        ALU_SGT,        -- Set if greather to (SIGNED)
        ALU_SGTU,       -- Set if greather to (UNSIGNED)
        ALU_SLE,        -- Set if less than or equal to (SIGNED)
        ALU_SLEU,       -- Set if less than or equal to (UNSIGNED)
        ALU_SLT,        -- Set if less to (SIGNED)
        ALU_SLTU,       -- Set if less to (UNSIGNED)
        ALU_SNE,        -- Set if not equal to
        ALU_SEQ         -- Set if equal to
        );

    constant N_ALU_OP : integer := ALU_OP_type'pos(ALU_OP_type'high)+1;      --number of possible
        ↪ ALU operations
    constant ALU_OP_SEL_SIZE : integer := log2_ceiling(N_ALU_OP);           --ALU operation
        ↪ selector size

end package ALU_package;
```

Listing C.4: 003-datapath_package.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.DLX_package.all;   --for constants

package datapath_package is

    type BRANCH_COND_type is (
        BRANCH_NO,       --don't jump
        BRANCH_ALWAYS,  --jump always
        BRANCH_EQZ,      --jump if equal to zero
        BRANCH_NEZ       --jump if not equal to zero
    );

    subtype RF_addr is std_logic_vector(RF_ADDR_SIZE-1 downto 0);

    constant RF_ADDR_LR_n: integer:= 31;
    constant RF_ADDR_LR: RF_addr:= std_logic_vector(to_unsigned(RF_ADDR_LR_n, RF_ADDR_SIZE));

end package datapath_package;
```

Listing C.5: 004-CU_package.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.ALU_package.all;       --for ALU_OP_type
use work.datapath_package.all;  --for BRANCH_COND_type
use work.DLX_package.all;       --for FUNC_SIZE, OPCODE_SIZE and constants
use work.DRAM_package.all;      --for DRAM_WR_EN_type

package CU_package is

    type CU_OUTS_ID_type is record
        -- ID outputs
        MUX_BRANCH_sel: std_logic;                   --select whether the jump offset is taken
            ↪ from the immediate value or from a register
        MUX_IMM_EXT_sel     : std_logic;                     --extend from 16 bit or from 26 bit
        MUX_RF_WR_ADDR_sel  : std_logic_vector(1 downto 0); --select whether the RF write address
            ↪  is taken from bits 20-16 or from bits 15-11 of the instruction register out, or
            ↪ whether it is 31
        BRANCH_COND         : BRANCH_COND_type;              --branch condition
    end record CU_OUTS_ID_type;

    type CU_OUTS_EXE_type is record
        -- EX outputs
        MULT_EN             : std_logic;
        MUX_MULT_sel        : std_logic;        --select whether the output is taken from the alu
            ↪  or from the multiplier
        MUX_ALU_IN2_sel     : std_logic;        --select whether the second operand is B or IMM
        ALU_OP              : ALU_OP_type;      --ALU operation selector
    end record CU_OUTS_EXE_type;

    type CU_OUTS_MEM_type is record
        -- MEM outputs
        DRAM_WR_EN          : DRAM_WR_EN_type;              --data RAM write enable
        MUX_DRAM_OUT_EXT_sel: std_logic_vector(2 downto 0); --select size and type of extension
             ↪ of data out from DRAM
    end record CU_OUTS_MEM_type;

    type CU_OUTS_WB_type is record
        -- WB outputs
        MUX_WB_SEL          : std_logic_vector(1 downto 0); --write back MUX sel
        RF_WR_EN            : std_logic;                    --register file write enable
    end record CU_OUTS_WB_type;

    type CU_OUTS_type is record
        ID  : CU_OUTS_ID_type;
        EXE : CU_OUTS_EXE_type;
        MEM : CU_OUTS_MEM_type;
        WB  : CU_OUTS_WB_type;
    end record CU_OUTS_type;

    type LUT_RTYPE_type is array (0 to N_RTYPE-1) of CU_OUTS_type;
    type LUT_ITYPE_type is array (1 to N_ITYPE-1) of CU_OUTS_type;  --start from 1 because if
         ↪ OPCODE=0 the instruction is R-type

    --ID outputs constants
    constant BRANCH_REG     : std_logic:= '0';
    constant BRANCH_IMM     : std_logic:= '1';
    constant IMM_EXT_16     : std_logic:= '0';
    constant IMM_EXT_26     : std_logic:= '1';
    constant RF_WR_2ND      : std_logic_vector:= "00";
    constant RF_WR_3TH      : std_logic_vector:= "01";
    constant RF_WR_31       : std_logic_vector:= "10";
    --EX outputs constants
    constant MULT_OFF       : std_logic:= '0';
    constant MULT_ON        : std_logic:= '1';
    constant OUT_ALU        : std_logic:= '0';
    constant OUT_MULT       : std_logic:= '1';
    constant ALU_IN1_PC_4   : std_logic:= '0';
    constant ALU_IN1_RF     : std_logic:= '1';
    constant ALU_IN2_RF     : std_logic:= '0';
    constant ALU_IN2_IMM    : std_logic:= '1';
    --MEM outputs
    constant EXT_SB         : std_logic_vector:= "000";
    constant EXT_UB         : std_logic_vector:= "001";
    constant EXT_SH         : std_logic_vector:= "010";
    constant EXT_UH         : std_logic_vector:= "011";
    constant EXT_W          : std_logic_vector:= "100";
    --WB outputs
    constant WB_NPC         : std_logic_vector:= "00";
    constant WB_LMD         : std_logic_vector:= "01";
    constant WB_ALU         : std_logic_vector:= "10";
    constant RF_WR_OFF      : std_logic:= '0';
    constant RF_WR_ON       : std_logic:= '1';

    --                                      ID
         ↪                 EXE                                                     MEM
         ↪                             WB
    --                                      MUX_BRANCH_sel  MUX_IMM_EXT_sel MUX_RF_WR_ADDR_sel
         ↪ BRANCH_COND     MULT_EN     MUX_MULT    MUX_ALU_IN2_sel ALU_OP          DRAM_WR_EN
         ↪     MUX_DRAM_OUT_EXT_sel    MUX_WB_SEL  RF_WR_EN
    --R-type instruction outs
    constant NOP_outs   :CU_OUTS_type:=(    ('-',           '-',            "--",
         ↪ BRANCH_NO),     (MULT_OFF,  '-',        '-',            ALU_DONT_CARE), (DRAM_WR_OFF,
         ↪    "---"),                 ("--",      RF_WR_OFF   ));
    constant ADD_outs   :CU_OUTS_type:=(    ('-',           '-',            RF_WR_3TH,
         ↪ BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,     ALU_ADD),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant ADDU_outs  :CU_OUTS_type:=(    ('-',           '-',            RF_WR_3TH,
         ↪ BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,     ALU_ADD),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SUB_outs   :CU_OUTS_type:=(    ('-',           '-',            RF_WR_3TH,
         ↪ BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,     ALU_SUB),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SUBU_outs  :CU_OUTS_type:=(    ('-',           '-',            RF_WR_3TH,
         ↪ BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,     ALU_SUB),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant AND_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_AND),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant OR_outs    :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_OR),        (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant XOR_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_XOR),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SGE_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SGE),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SGEU_outs  :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SGEU),      (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SGT_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SGT),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SGTU_outs  :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SGTU),      (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLE_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SLE),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLEU_outs  :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SLEU),      (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLT_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SLT),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLTU_outs  :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SLTU),      (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLL_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SLL),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SRL_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SRL),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SRA_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SRA),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant ROR_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_ROR),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant ROL_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_ROL),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SNE_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SNE),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SEQ_outs   :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_RF,    ALU_SEQ),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant MULT_outs  :CU_OUTS_type:=(   ('-',         '-',           RF_WR_3TH,
         ↪ BRANCH_NO),      (MULT_ON,   OUT_MULT,   ALU_IN2_RF,    ALU_DONT_CARE), (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    --I-type instruction outs
         ↪
         ↪
    constant ADDI_outs  :CU_OUTS_type:=(   ('-',         IMM_EXT_16,    RF_WR_2ND,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,   ALU_ADD),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant ADDUI_outs :CU_OUTS_type:=(   ('-',         IMM_EXT_16,    RF_WR_2ND,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,   ALU_ADD),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SUBI_outs  :CU_OUTS_type:=(   ('-',         IMM_EXT_16,    RF_WR_2ND,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,   ALU_SUB),       (DRAM_WR_OFF,
         ↪    "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SUBUI_outs :CU_OUTS_type:=(   ('-',         IMM_EXT_16,    RF_WR_2ND,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,   ALU_SUB),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant ANDI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_AND),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant ORI_outs   :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_OR),        (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant XORI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_XOR),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SGEI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SGE),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SGEUI_outs :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SGEU),      (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SGTI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SGT),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SGTUI_outs :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SGTU),      (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLEI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SLE),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLEUI_outs :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SLEU),      (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLTI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SLT),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLTUI_outs :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SLTU),      (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SLLI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SLL),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SRLI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SRL),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SRAI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SRA),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant ROLI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ROR),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant RORI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ROL),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SNEI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SNE),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant SEQI_outs  :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_SEQ),       (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    constant MULTI_outs :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪   BRANCH_NO),     (MULT_ON,   OUT_MULT,   ALU_IN2_IMM,    ALU_DONT_CARE), (DRAM_WR_OFF,
         ↪      "---"),                 (WB_ALU,    RF_WR_ON    ));
    --jump and DRAM instruction outs
         ↪
         ↪
    constant J_outs     :CU_OUTS_type:=(    (BRANCH_IMM,    IMM_EXT_26,     "--",
         ↪   BRANCH_ALWAYS), (MULT_OFF,  '-',        '-',            ALU_DONT_CARE), (DRAM_WR_OFF,
         ↪      "---"),                 ("--",      RF_WR_OFF   ));
    constant JR_outs    :CU_OUTS_type:=(    (BRANCH_REG,    '-',            "--",
         ↪   BRANCH_ALWAYS), (MULT_OFF,  '-',        '-',            ALU_DONT_CARE), (DRAM_WR_OFF,
         ↪      "---"),                 ("--",      RF_WR_OFF   ));
    constant JAL_outs   :CU_OUTS_type:=(    (BRANCH_IMM,    IMM_EXT_26,     RF_WR_31,
         ↪   BRANCH_ALWAYS), (MULT_OFF,  '-',        '-',            ALU_DONT_CARE), (DRAM_WR_OFF,
         ↪      "---"),                 (WB_NPC,    RF_WR_ON    ));
    constant JALR_outs  :CU_OUTS_type:=(    (BRANCH_REG,    IMM_EXT_26,     RF_WR_31,
         ↪   BRANCH_ALWAYS), (MULT_OFF,  '-',        '-',            ALU_DONT_CARE), (DRAM_WR_OFF,
         ↪     "---"),                 (WB_NPC,    RF_WR_ON    ));
    constant BEQZ_outs  :CU_OUTS_type:=(    (BRANCH_IMM,    IMM_EXT_16,     "--",
         ↪ BRANCH_EQZ),     (MULT_OFF,  '-',        '-',            ALU_DONT_CARE), (DRAM_WR_OFF,
         ↪     "---"),                 ("--",      RF_WR_OFF   ));
    constant BNEZ_outs  :CU_OUTS_type:=(    (BRANCH_IMM,    IMM_EXT_16,     "--",
         ↪ BRANCH_NEZ),     (MULT_OFF,  '-',        '-',            ALU_DONT_CARE), (DRAM_WR_OFF,
         ↪     "---"),                 ("--",      RF_WR_OFF   ));
    constant LW_outs    :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ADD),       (DRAM_WR_OFF,
         ↪     EXT_W),                 (WB_LMD,    RF_WR_ON    ));
    constant LH_outs    :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ADD),       (DRAM_WR_OFF,
         ↪     EXT_SH),                (WB_LMD,    RF_WR_ON    ));
    constant LHU_outs   :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ADD),       (DRAM_WR_OFF,
         ↪     EXT_UH),                (WB_LMD,    RF_WR_ON    ));
    constant LB_outs    :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ADD),       (DRAM_WR_OFF,
         ↪     EXT_SB),                (WB_LMD,    RF_WR_ON    ));
    constant LBU_outs   :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     RF_WR_2ND,
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ADD),       (DRAM_WR_OFF,
         ↪     EXT_UB),                (WB_LMD,    RF_WR_ON    ));
    constant SW_outs    :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     "--",
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ADD),       (DRAM_WR_W,
         ↪      "---"),                 ("--",      RF_WR_OFF   ));
    constant SH_outs    :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     "--",
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ADD),       (DRAM_WR_H,
         ↪      "---"),                 ("--",      RF_WR_OFF   ));
    constant SB_outs    :CU_OUTS_type:=(    ('-',           IMM_EXT_16,     "--",
         ↪ BRANCH_NO),      (MULT_OFF,  OUT_ALU,    ALU_IN2_IMM,    ALU_ADD),       (DRAM_WR_B,
         ↪      "---"),                 ("--",      RF_WR_OFF   ));

end package CU_package;
```

Listing C.6: DRAM_package.vhd

```
package DRAM_package is

    type DRAM_WR_EN_type is (
        DRAM_WR_OFF,
        DRAM_WR_B,
        DRAM_WR_H,
        DRAM_WR_W
    );

end package DRAM_package;
```

Listing C.7: FU_and_HDU_package.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.DLX_package.all;       --for get_opcode and constants

package FU_and_HDU_package is

    --FU package
    type FU_OUTS_type is record
        MUX_RF_OUT1_sel             : std_logic_vector(2 downto 0);
        MUX_RF_OUT2_sel             : std_logic_vector(2 downto 0);
        MUX_DRAM_IN_sel             : std_logic;
    end record FU_OUTS_type;

    constant RF_forward_NO          : std_logic_vector(2 downto 0):= "000";
    constant RF_forward_EXE_ALU     : std_logic_vector(2 downto 0):= "001";
    constant RF_forward_MEM_ALU     : std_logic_vector(2 downto 0):= "010";
    constant RF_forward_MEM_DRAM    : std_logic_vector(2 downto 0):= "011";
    constant RF_forward_WB          : std_logic_vector(2 downto 0):= "100";
    constant RF_forward_MEM_NPC     : std_logic_vector(2 downto 0):= "101";
    constant RF_forward_EXE_NPC     : std_logic_vector(2 downto 0):= "110";
    constant DRAM_forward_NO        : std_logic:= '0';
    constant DRAM_forward_WB        : std_logic:= '1';

    function is_not_a_jump (INSTR: word) return boolean;
    function is_not_a_jump_imm (INSTR: word) return boolean;
    function is_not_a_store (INSTR: word) return boolean;
    function is_a_load (INSTR: word) return boolean;
    function is_a_subroutine_call (INSTR: word) return boolean;
    function is_a_mult (INSTR: word) return boolean;

    --HDU package
    type HDU_OUTS_type is record
        PC_EN   : std_logic;
        IF_EN   : std_logic;
        ID_EN   : std_logic;
        EXE_EN  : std_logic;
        MEM_EN  : std_logic;
        WB_EN   : std_logic;
        ID_bubble: std_logic;
        EXE_bubble: std_logic;
        MEM_bubble: std_logic;
        WB_bubble: std_logic;
    end record HDU_OUTS_type;

end package FU_and_HDU_package;

package body FU_and_HDU_package is

    function is_not_a_jump(instr: word) return boolean is
        variable opcode: OPCODE_type;
    begin
        opcode:=get_opcode(instr);
        return opcode/=OPCODE_BEQZ and opcode/=OPCODE_BNEZ and opcode/=OPCODE_JAL and opcode/=
            ↪ OPCODE_J and opcode/=OPCODE_JR and opcode/=OPCODE_JALR;
    end function is_not_a_jump;

    function is_not_a_jump_imm(instr: word) return boolean is
        variable opcode: OPCODE_type;
    begin
        opcode:=get_opcode(instr);
        return opcode/=OPCODE_JAL and opcode/=OPCODE_J;
    end function is_not_a_jump_imm;

    function is_not_a_store (INSTR: word) return boolean is
    begin
        return get_opcode(INSTR)/=OPCODE_SW and get_opcode(INSTR)/=OPCODE_SH and get_opcode(INSTR
            ↪ )/=OPCODE_SB;
    end function is_not_a_store;

    function is_a_load (INSTR: word) return boolean is
        variable OPCODE: OPCODE_type;
    begin
        OPCODE:=get_opcode(INSTR);
        return OPCODE=OPCODE_LW or OPCODE=OPCODE_LH or OPCODE=OPCODE_LHU or OPCODE=OPCODE_LB or
            ↪ OPCODE=OPCODE_LBU or OPCODE=OPCODE_LHI;
    end function is_a_load;

    function is_a_subroutine_call (INSTR: word) return boolean is
    begin
        return get_opcode(INSTR)=OPCODE_JAL or get_opcode(INSTR)=OPCODE_JALR;
    end function is_a_subroutine_call;

    function is_a_mult (INSTR: word) return boolean is
    begin
        return (get_opcode(INSTR)=OPCODE_RTYPE and get_func(INSTR)=FUNC_MULT) or get_opcode(INSTR
            ↪ )=OPCODE_MULTI;
    end function is_a_mult;

end package body FU_and_HDU_package;
```

Listing C.8: TB_DLX.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.DRAM_package.all;       --for DRAM_WR_EN_type
use work.DLX_package.all;        --for word type

entity TB_DLX is
end entity TB_DLX;

architecture test of TB_DLX is

    component DLX is
        generic(
            BPU_TAG_FIELD_SIZE  : integer :=8;
            BPU_SET_FIELD_SIZE  : integer :=3;
            BPU_LINES_PER_SET   : integer :=4
        );
        port(
            IRAM_ADDR    : out   word;
            IRAM_OUT     : in    word;
            DRAM_ADDR    : out   word;
            DRAM_IN      : out   word;
            DRAM_OUT     : in    word;
            DRAM_WR_EN   : out   DRAM_WR_EN_type;
            CLK          : in    std_logic;
            RST          : in    std_logic
        );
    end component DLX;

    component IRAM_test is
        generic(MEM_SIZE : integer := 1024);
        port(
            DATA_OUT    : out word;
            ADDR        : in word
            );
    end component IRAM_test;

    component IRAM_file is
        generic(MEM_SIZE : integer := 1024);
        port(
            DATA_OUT    : out word;
            ADDR        : in word
            );
    end component IRAM_file;

    component DRAM is
        generic(
            MEM_SIZE: positive:= 1024);
        port(
            DATA_IN     : in word;
            DATA_OUT    : out word;
            ADDR        : in word;
            WR_EN       : in DRAM_WR_EN_type;
            CLK         : in std_logic);
    end component DRAM;

    signal IRAM_ADDR, IRAM_OUT, DRAM_ADDR, DRAM_IN, DRAM_OUT: word;
    signal DRAM_WR_EN : DRAM_WR_EN_type;
    signal CLK, RST : std_logic;

    constant clk_period : time := 10ns;
    constant BPU_TAG_FIELD_SIZE : integer:=8;
    constant BPU_SET_FIELD_SIZE : integer:=3;
    constant BPU_LINES_PER_SET : integer:=4;
    constant DRAM_SIZE  : integer := 1024;
    constant IRAM_SIZE  : integer := 1024;
    constant IRAM_test_or_file : string := "file";

begin
```

```
    DUT: DLX
        generic map(
            BPU_TAG_FIELD_SIZE=>BPU_TAG_FIELD_SIZE,
            BPU_SET_FIELD_SIZE=>BPU_SET_FIELD_SIZE,
            BPU_LINES_PER_SET=>BPU_LINES_PER_SET
        )
        port map(
            IRAM_ADDR=>IRAM_ADDR,
            IRAM_OUT=>IRAM_OUT,
            DRAM_ADDR=>DRAM_ADDR,
            DRAM_IN=>DRAM_IN,
            DRAM_OUT=>DRAM_OUT,
            DRAM_WR_EN=>DRAM_WR_EN,
            CLK=>CLK,
            RST=>RST
        );

    IRAM_test_gen: if IRAM_test_or_file="test" generate
        IRAM_instance: IRAM_test
            generic map(MEM_SIZE=>IRAM_SIZE)
            port map(
                DATA_OUT=>IRAM_OUT,
                ADDR=>IRAM_ADDR
            );
    end generate IRAM_test_gen;

    IRAM_file_gen: if IRAM_test_or_file="file" generate
        IRAM_instance: IRAM_file
            generic map(MEM_SIZE=>IRAM_SIZE)
            port map(
                DATA_OUT=>IRAM_OUT,
                ADDR=>IRAM_ADDR
            );
    end generate IRAM_file_gen;

    DRAM_instance: DRAM
        generic map(
            MEM_SIZE=>DRAM_SIZE
        )
        port map(
            DATA_IN=>DRAM_IN,
            DATA_OUT=>DRAM_OUT,
            ADDR=>DRAM_ADDR,
            WR_EN=>DRAM_WR_EN,
            CLK=>CLK
        );

    clk_proc: process
    begin
        CLK<='0';
        wait for clk_period/2;
        CLK<='1';
        wait for clk_period/2;
    end process clk_proc;

    RST<='0', '1' after clk_period;

end architecture test;
```

Listing C.9: b-IRAM_file.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.DLX_package.all;   --for types definitions

entity IRAM_file is
```

```
    generic(MEM_SIZE : integer := 1024);
    port(
        DATA_OUT    : out word;
        ADDR        : in word
        );
end entity IRAM_file;

architecture behavioral of IRAM_file is

    type MEM_type is array (0 to MEM_SIZE-1) of byte;

    signal MEM: MEM_type;
    signal ADDR_to_int: natural;
    signal byte0_out, byte1_out, byte2_out, byte3_out: byte;

begin

    ADDR_to_int<=to_integer(unsigned(ADDR(30 downto 0)));

    byte0_out <= MEM(ADDR_to_int) when ADDR_to_int<=MEM_SIZE-1 else
                 (others=>'0');

    byte1_out <= MEM(ADDR_to_int+1) when ADDR_to_int+1<=MEM_SIZE-1 else
                 (others=>'0');

    byte2_out <= MEM(ADDR_to_int+2) when ADDR_to_int+2<=MEM_SIZE-1 else
                 (others=>'0');

    byte3_out <= MEM(ADDR_to_int+3) when ADDR_to_int+3<=MEM_SIZE-1 else
                 (others=>'0');

    DATA_OUT <= byte3_out & byte2_out & byte1_out & byte0_out;

    file_read_proc: process
        file file_to_read: text;
        variable file_line : line;
        variable n_byte : integer := 0;
        variable instruction : word;
    begin
        file_open(file_to_read,"test.mem",READ_MODE);
        while (not endfile(file_to_read)) loop
            readline(file_to_read,file_line);
            hread(file_line,instruction);
            MEM(n_byte) <= instruction(byte_range);
            MEM(n_byte+1) <= instruction(byte1_range);
            MEM(n_byte+2) <= instruction(byte2_range);
            MEM(n_byte+3) <= instruction(byte3_range);
            n_byte := n_byte + 4;
        end loop;
        wait;
    end process file_read_proc;

end behavioral;
```

Listing C.10: b-IRAM_test.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.DLX_package.all;       --for types and functions
use work.datapath_package.all;  --for constants

entity IRAM_test is
    generic(MEM_SIZE : integer := 1024);
    port(
        DATA_OUT    : out word;
        ADDR        : in word
        );
end entity IRAM_test;
```

```vhdl
architecture behavioral of IRAM_test is

    type IRAM_mem_type is array (0 to MEM_SIZE-1) of word;

    constant test_simple_add: IRAM_mem_type:=(
        INSTR_ITYPE(OPCODE_ADDUI,0,1,10),       --R1<-1
        INSTR_ITYPE(OPCODE_ADDUI,0,2,7),        --R2<-7
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_RTYPE(FUNC_ADDU,1,2,3),           --R3<-R1+R2
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_simple_dram: IRAM_mem_type:=(
        INSTR_ITYPE(OPCODE_ADDUI,0,1,32767),    --R1<-32767
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_RTYPE(FUNC_ADDU,1,1,1),           --R1<-R1+R1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_RTYPE(FUNC_ADDU,1,1,1),           --R1<-R1+R1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_SW,0,1,0),           --MEM(0+R0)<-R1 (word)
        INSTR_ITYPE(OPCODE_SH,0,1,4),           --MEM(4+R0)<-R1 (halfword)
        INSTR_ITYPE(OPCODE_SB,0,1,6),           --MEM(6+R0)<-R1 (byte)
        INSTR_ITYPE(OPCODE_LW,0,2,0),           --R2<-MEM(0+R0) (word)
        INSTR_ITYPE(OPCODE_LH,0,3,4),           --R3<-MEM(4+R0) (halfword)
        INSTR_ITYPE(OPCODE_LBU,0,4,6),          --R4<-MEM(6+R0) (byte)
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_simple_jump: IRAM_mem_type:=(
        INSTR_ITYPE(OPCODE_ADDUI,2,2,1),        --R2<-R2+1
        INSTR_ITYPE(OPCODE_ADDUI,1,1,1),        --R1<-R1+1
        INSTR_JTYPE(OPCODE_J, -8),              --jump to first instruction
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_simple_conditional_branch: IRAM_mem_type:=(
        INSTR_ITYPE(OPCODE_ADDUI,0,1,3),        --R1<-3
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_ADDUI,1,1,-1),       --R1<-R1-1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_BEQZ,1,0,400),       --if R1=0 branch forward
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_BNEZ,1,0,-36),       --if R1!=0 branch behind
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_forwarding_ALU: IRAM_mem_type:=(
        --test ALU output at EXE to ALU input 1
        INSTR_ITYPE(OPCODE_ADDUI,0,1,5),        --R1<-5
        INSTR_ITYPE(OPCODE_ADDUI,1,2,1),        --R2<-R1+1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test ALU output at MEM to ALU input 1
        INSTR_ITYPE(OPCODE_ADDUI,0,1,1),        --R1<-1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_ADDUI,1,2,1),        --R2<-R1+1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test ALU output at WB to ALU input 1
        INSTR_ITYPE(OPCODE_ADDUI,0,1,7),        --R1<-7
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_ADDUI,1,2,1),        --R2<-R1+1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test DRAM output at MEM to ALU input 1
        INSTR_ITYPE(OPCODE_SW,0,2,0),           --MEM(0+R0)<-R2 (word)
        INSTR_ITYPE(OPCODE_LW,0,1,0),           --R1<-MEM(0+R0) (word)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_ADDUI,1,2,1),        --R2<-R1+1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test DRAM output at WB to ALU input 1
        INSTR_ITYPE(OPCODE_SW,0,2,0),           --MEM(0+R0)<-R2 (word)
        INSTR_ITYPE(OPCODE_LW,0,1,0),           --R1<-MEM(0+R0) (word)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_ADDUI,1,2,1),        --R2<-R1+1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test ALU output at EXE to ALU input 2
        INSTR_ITYPE(OPCODE_ADDUI,0,1,5),        --R1<-5
        INSTR_RTYPE(FUNC_ADDU,0,1,2),           --R2<-R0+R1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test ALU output at MEM to ALU input 2
        INSTR_ITYPE(OPCODE_ADDUI,0,1,1),        --R1<-1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_RTYPE(FUNC_ADDU,0,1,2),           --R2<-R0+R1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test ALU output at WB to ALU input 2
        INSTR_ITYPE(OPCODE_ADDUI,0,1,7),        --R1<-7
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_RTYPE(FUNC_ADDU,0,1,2),           --R2<-R0+R1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test DRAM output at MEM to ALU input 2
        INSTR_ITYPE(OPCODE_SW,0,1,0),           --MEM(0+R0)<-R1 (word)
        INSTR_ITYPE(OPCODE_LW,0,3,0),           --R3<-MEM(0+R0) (word)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_RTYPE(FUNC_ADDU,2,3,1),           --R1<-R2+R3
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test DRAM output at WB to ALU input 2
        INSTR_ITYPE(OPCODE_SW,0,1,0),           --MEM(0+R0)<-R1 (word)
        INSTR_ITYPE(OPCODE_LW,0,3,0),           --R3<-MEM(0+R0) (word)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_RTYPE(FUNC_ADDU,2,3,1),           --R1<-R2+R3
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_forwarding_branch: IRAM_mem_type:=(
        --test ALU output at EXE to branch comparator
        INSTR_ITYPE(OPCODE_ADDUI,0,1,1),        --R1<-1
        INSTR_ITYPE(OPCODE_BNEZ,1,0,+12),       --if R1!=0 branch to next test
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test ALU output at MEM to branch comparator
        INSTR_ITYPE(OPCODE_ADDUI,0,1,0),        --R1<-0
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_BEQZ,1,0,+12),       --if R1=0 branch to next test
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test ALU output at WB to branch comparator
        INSTR_ITYPE(OPCODE_ADDUI,0,1,1),        --R1<-1
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_BNEZ,1,0,+12),       --if R1!=0 branch to next test
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test DRAM output at MEM to branch comparator
        INSTR_ITYPE(OPCODE_SW,0,1,0),           --MEM(0+R0)<-R1 (word)
        INSTR_ITYPE(OPCODE_LW,0,2,0),           --R2<-MEM(0+R0) (word)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_BNEZ,2,0,+12),       --if R2!=0 branch to next test
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test DRAM output at WB to branch comparator
        INSTR_ITYPE(OPCODE_LW,0,3,0),           --R3<-MEM(0+R0) (word)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_BNEZ,3,0,+100),      --if R3!=0 branch forward
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_forwarding_DRAM: IRAM_mem_type:=(
        --initialize dram
        INSTR_ITYPE(OPCODE_ADDUI,0,1,1),        --R1<-1
        INSTR_ITYPE(OPCODE_SB,0,1,100),         --MEM(100+R0)<-R1 (byte)
        --test DRAM output at MEM to DRAM in
        INSTR_ITYPE(OPCODE_LB,0,2,100),         --R2<-MEM(100+R0) (byte)
        INSTR_ITYPE(OPCODE_SB,0,2,0),           --MEM(0+R0)<-R2 (byte)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test DRAM output at WB to DRAM in
        INSTR_ITYPE(OPCODE_LB,0,3,100),         --R3<-MEM(100+R0) (byte)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_SB,0,3,1),           --MEM(1+R0)<-R3 (byte)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test ALU output at MEM to DRAM in
        INSTR_ITYPE(OPCODE_ADDUI,0,1,5),        --R1<-5
        INSTR_ITYPE(OPCODE_SB,0,1,0),           --MEM(0+R0)<-R1 (byte)
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        --test ALU output at WB to DRAM in
        INSTR_ITYPE(OPCODE_ADDUI,0,2,8),        --R2<-8
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_SB,0,2,1),           --MEM(1+R0)<-R2 (byte)
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_HDU_load: IRAM_mem_type:=(
        --initialize dram
        INSTR_ITYPE(OPCODE_ADDUI,0,1,1),        --R1<-1
        INSTR_ITYPE(OPCODE_SB,0,1,0),           --MEM(0+R0)<-R1 (byte)
        --test
        INSTR_ITYPE(OPCODE_LB,0,2,0),           --R2<-MEM(0+R0) (byte)
        INSTR_ITYPE(OPCODE_ADDUI,2,1,1),        --R1<-R2+1
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );
    constant test_HDU_branch: IRAM_mem_type:=(
        INSTR_ITYPE(OPCODE_ADDUI,0,1,1),        --R1<-1
        INSTR_ITYPE(OPCODE_BNEZ,1,0,+4),        --if R1!=0 branch
        --the next instruction should not be executed
        INSTR_ITYPE(OPCODE_ADDUI,0,2,1),        --R2<-1
        --the next instruction should be executed
        INSTR_ITYPE(OPCODE_ADDUI,0,3,1),        --R3<-1
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_HDU_subroutine: IRAM_mem_type:=(
        INSTR_ITYPE(OPCODE_ADDUI,0,1,5),        --R1<-5
        INSTR_ITYPE(OPCODE_ADDUI,0,2,4),        --R2<-4
        INSTR_JTYPE(OPCODE_JAL,32),             --subroutine call
        INSTR_ITYPE(OPCODE_ADDUI,3,1,0),        --R1<-R3
        INSTR_JTYPE(OPCODE_J,-4),               --trap
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_ITYPE(OPCODE_NOP,0,0,0),
        INSTR_RTYPE(FUNC_ADDU,1,2,3),           --R3<-R1+R2
        INSTR_ITYPE(OPCODE_JR,31,0,0),          --return to subroutine
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_mult: IRAM_mem_type:=(
        INSTR_ITYPE(OPCODE_ADDUI,0,1,5),        --R1<-5
        INSTR_ITYPE(OPCODE_ADDUI,0,2,7),        --R2<-7
        INSTR_RTYPE(FUNC_MULT,1,2,3),           --R3<-R1*R2
        INSTR_ITYPE(OPCODE_MULTI,1,1,3),        --R1<-R1*3
        INSTR_ITYPE(OPCODE_ADDUI,0,1,0),        --R1<-0
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant test_BPU: IRAM_mem_type:=(
        INSTR_ITYPE(OPCODE_ADDUI,0,1,5),        --R1<-5
        INSTR_ITYPE(OPCODE_ADDUI,1,1,-1),       --R1<-R1-1
        INSTR_ITYPE(OPCODE_BEQZ,1,0,100),       --if R1=0 branch forward
        INSTR_ITYPE(OPCODE_BNEZ,1,0,-12),       --if R1!=0 branch behind
        others=>INSTR_ITYPE(OPCODE_NOP,0,0,0)
    );

    constant program_to_run: IRAM_mem_type:= test_BPU;

begin

    DATA_OUT <= program_to_run(to_integer(unsigned(Addr))/4) when to_integer(unsigned(ADDR))<=
        ↪ MEM_SIZE else
            (others=>'-');

end behavioral;
```

Listing C.11: b-IRAM_file.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use std.textio.all;
use ieee.std_logic_textio.all;
use work.DLX_package.all;   --for types definitions

entity IRAM_file is
    generic(MEM_SIZE : integer := 1024);
    port(
        DATA_OUT    : out word;
        ADDR        : in word
        );
```

```
end entity IRAM_file;

architecture behavioral of IRAM_file is

    type MEM_type is array (0 to MEM_SIZE-1) of byte;

    signal MEM: MEM_type;
    signal ADDR_to_int: natural;
    signal byte0_out, byte1_out, byte2_out, byte3_out: byte;

begin

    ADDR_to_int<=to_integer(unsigned(ADDR(30 downto 0)));

    byte0_out <= MEM(ADDR_to_int) when ADDR_to_int<=MEM_SIZE-1 else
                 (others=>'0');

    byte1_out <= MEM(ADDR_to_int+1) when ADDR_to_int+1<=MEM_SIZE-1 else
                 (others=>'0');

    byte2_out <= MEM(ADDR_to_int+2) when ADDR_to_int+2<=MEM_SIZE-1 else
                 (others=>'0');

    byte3_out <= MEM(ADDR_to_int+3) when ADDR_to_int+3<=MEM_SIZE-1 else
                 (others=>'0');

    DATA_OUT <= byte3_out & byte2_out & byte1_out & byte0_out;

    file_read_proc: process
        file file_to_read: text;
        variable file_line : line;
        variable n_byte : integer := 0;
        variable instruction : word;
    begin
        file_open(file_to_read,"test.mem",READ_MODE);
        while (not endfile(file_to_read)) loop
            readline(file_to_read,file_line);
            hread(file_line,instruction);
            MEM(n_byte) <= instruction(byte_range);
            MEM(n_byte+1) <= instruction(byte1_range);
            MEM(n_byte+2) <= instruction(byte2_range);
            MEM(n_byte+3) <= instruction(byte3_range);
            n_byte := n_byte + 4;
        end loop;
        wait;
    end process file_read_proc;

end behavioral;
```

Listing C.12: c-DRAM.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.my_package.all;        --for log2_ceiling
use work.DLX_package.all;       --for types
use work.DRAM_package.all;      --for DRAM_WR_type

entity DRAM is
    generic(
        MEM_SIZE: positive:= 1024);
    port(
        DATA_IN     : in word;
        DATA_OUT    : out word;
        ADDR        : in word;
        WR_EN       : in DRAM_WR_EN_type;
        CLK         : in std_logic);
end entity DRAM;
```

```
architecture behavioral of DRAM is

    type MEM_type is array (0 to MEM_SIZE-1) of byte;

    signal MEM: MEM_type;
    signal ADDR_to_int: natural;
    signal byte0_out, byte1_out, byte2_out, byte3_out: byte;

begin

    ADDR_to_int<=to_integer(unsigned(ADDR(29 downto 0)));       --max 30 bits otherwise integer
         ↪ can overflow

    byte0_out  <= MEM(ADDR_to_int) when ADDR_to_int<=MEM_SIZE-1 and ADDR(31 downto 30)="00" else
                 (others=>'0');

    byte1_out  <= MEM(ADDR_to_int+1) when ADDR_to_int+1<=MEM_SIZE-1 and ADDR(31 downto 30)="00"
         ↪ else
                 (others=>'0');

    byte2_out  <= MEM(ADDR_to_int+2) when ADDR_to_int+2<=MEM_SIZE-1 and ADDR(31 downto 30)="00"
         ↪ else
                 (others=>'0');

    byte3_out  <= MEM(ADDR_to_int+3) when ADDR_to_int+3<=MEM_SIZE-1 and ADDR(31 downto 30)="00"
         ↪ else
                 (others=>'0');

    DATA_OUT  <= byte3_out & byte2_out & byte1_out & byte0_out;

    write_proc: process(CLK)
    begin
        if rising_edge(CLK) then
            if WR_EN=DRAM_WR_B or WR_EN=DRAM_WR_H or WR_EN=DRAM_WR_W then
                if ADDR_to_int<=MEM_SIZE-1 and ADDR(31 downto 30)="00" then
                    MEM(ADDR_to_int)<=DATA_IN(byte_range);
                end if;
            end if;
            if WR_EN=DRAM_WR_H or WR_EN=DRAM_WR_W then
                if ADDR_to_int+1<=MEM_SIZE-1 and ADDR(31 downto 30)="00" then
                    MEM(ADDR_to_int+1)<=DATA_IN(byte1_range);
                end if;
            end if;
            if WR_EN=DRAM_WR_W then
                if ADDR_to_int+2<=MEM_SIZE-1 and ADDR(31 downto 30)="00" then
                    MEM(ADDR_to_int+2)<=DATA_IN(byte2_range);
                end if;
                if ADDR_to_int+3<=MEM_SIZE-1 and ADDR(31 downto 30)="00" then
                    MEM(ADDR_to_int+3)<=DATA_IN(byte3_range);
                end if;
            end if;
        end if;
    end process write_proc;

end architecture behavioral;
```

Listing C.13: a-DLX.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.DLX_package.all;           --for word type
use work.CU_package.all;            --for CU_OUTS_type
use work.my_package.all;            --for log2_ceiling
use work.DRAM_package.all;          --for DRAM_WR_EN_type
use work.FU_and_HDU_package.all;    --for FU_OUTS_type and HDU_OUTS_type

entity DLX is
    generic(
        BPU_TAG_FIELD_SIZE  : integer :=8;
        BPU_SET_FIELD_SIZE  : integer :=3;
        BPU_LINES_PER_SET   : integer :=4
    );
    port(
        IRAM_ADDR   : out   word;
        IRAM_OUT    : in    word;
        DRAM_ADDR   : out   word;
        DRAM_IN     : out   word;
        DRAM_OUT    : in    word;
        DRAM_WR_EN  : out   DRAM_WR_EN_type;
        CLK         : in    std_logic;
        RST         : in    std_logic
    );
end entity DLX;

architecture structural of DLX is

    component reg is
        Generic (N : positive:= 1 );                              --number of bits
        Port(   D       : In    std_logic_vector(N-1 downto 0);     --data input
                Q       : Out   std_logic_vector(N-1 downto 0);     --data output
                EN      : In    std_logic;                          --enable active high
                CLK     : In    std_logic;                          --clock
                RST     : In    std_logic);                         --asynchronous reset active
                    ↪ low
    end component reg;

    component CU is
        port(
            INSTR_ID    : in word;                  --instruction register input
            CU_OUTS     : out CU_OUTS_type          --CU outs to datapath
        );
    end component CU;

    component HDU is
        port(
            INSTR_ID        : in word;
            INSTR_EXE       : in word;
            misprediction   : in std_logic;
            HDU_OUTS        : out HDU_OUTS_type;
            clk             : in std_logic;
            rst             : in std_logic
        );
    end component HDU;

    component FU is
        port(
            INSTR_ID    : in word;
            INSTR_EXE   : in word;
            INSTR_MEM   : in word;
            INSTR_WB    : in word;
            FU_OUTS     : out FU_OUTS_type
        );
    end component FU;

    component datapath is
        generic(
            BPU_TAG_FIELD_SIZE  : integer :=8;
            BPU_SET_FIELD_SIZE  : integer :=3;
            BPU_LINES_PER_SET   : integer :=4
        );
        port(
            IRAM_ADDR       : out   word;
            IRAM_OUT        : in    word;
            DRAM_ADDR       : out   word;
            DRAM_IN         : out   word;
            DRAM_OUT        : in    word;
            control_from_CU : in    CU_OUTS_type;
            control_from_FU : in    FU_OUTS_type;
            control_from_HDU: in    HDU_OUTS_type;
            misprediction   : out   std_logic;
            CLK             : in    std_logic;
            RST             : in    std_logic
        );
    end component datapath;

    --signals to pipelined the CU outputs
    signal CU_OUTS, CU_OUTS_pipelined                        : CU_OUTS_type;
    signal CU_OUTS_EXE_atEXE                                 : CU_OUTS_EXE_type;
    signal CU_OUTS_MEM_atEXE, CU_OUTS_MEM_atMEM              : CU_OUTS_MEM_type;
    signal CU_OUTS_WB_atEXE, CU_OUTS_WB_atMEM, CU_OUTS_WB_atWB  : CU_OUTS_WB_type;

    --signals to connect components
    signal HDU_OUTS : HDU_OUTS_type;
    signal FU_OUTS  : FU_OUTS_type;
    signal misprediction    : std_logic;

    --signals to record the instruction for each stage
    signal INSTR_ID, INSTR_EXE, INSTR_MEM, INSTR_WB : word;

begin

    IR_proc: process(CLK, RST) is
    begin
        if RST='0' then
            INSTR_ID<=INSTR_ITYPE(OPCODE_NOP,0,0,0);
            INSTR_EXE<=INSTR_ITYPE(OPCODE_NOP,0,0,0);
            INSTR_MEM<=INSTR_ITYPE(OPCODE_NOP,0,0,0);
            INSTR_WB<=INSTR_ITYPE(OPCODE_NOP,0,0,0);
        elsif rising_edge(CLK) then
            if HDU_OUTS.ID_bubble='1' then
                INSTR_ID<=INSTR_ITYPE(OPCODE_NOP,0,0,0);
            elsif HDU_OUTS.IF_EN='1' then
                INSTR_ID<=IRAM_OUT;
            end if;
            if HDU_OUTS.EXE_bubble='1' then
                INSTR_EXE<=INSTR_ITYPE(OPCODE_NOP,0,0,0);
            elsif HDU_OUTS.ID_EN='1' then
                INSTR_EXE<=INSTR_ID;
            end if;
            if HDU_OUTS.MEM_bubble='1' then
                INSTR_MEM<=INSTR_ITYPE(OPCODE_NOP,0,0,0);
            elsif HDU_OUTS.EXE_EN='1' then
                INSTR_MEM<=INSTR_EXE;
            end if;
            if HDU_OUTS.WB_bubble='1' then
                INSTR_WB<=INSTR_ITYPE(OPCODE_NOP,0,0,0);
            elsif HDU_OUTS.MEM_EN='1' then
                INSTR_WB<=INSTR_MEM;
            end if;
        end if;
    end process IR_proc;

    HDU_instance: HDU
        port map(
            HDU_OUTS=>HDU_OUTS,
            INSTR_ID=>INSTR_ID,
            INSTR_EXE=>INSTR_EXE,
            misprediction=>misprediction,
            clk=>clk,
            rst=>rst
        );

    FU_instance: FU
        port map(
            INSTR_ID=>INSTR_ID,
            INSTR_EXE=>INSTR_EXE,
            INSTR_MEM=>INSTR_MEM,
            INSTR_WB=>INSTR_WB,
            FU_OUTS=>FU_OUTS
        );

    CU_instance: CU
        port map(
            INSTR_ID=>INSTR_ID,
            CU_OUTS=>CU_OUTS
        );

    REGS_CU: process(CLK, RST)
    begin
        if RST='0' then
            CU_OUTS_EXE_atEXE<=NOP_outs.EXE;
            CU_OUTS_MEM_atEXE<=NOP_outs.MEM;
            CU_OUTS_WB_atEXE<=NOP_outs.WB;
            CU_OUTS_MEM_atMEM<=NOP_outs.MEM;
            CU_OUTS_WB_atMEM<=NOP_outs.WB;
            CU_OUTS_WB_atWB<=NOP_outs.WB;
        elsif rising_edge(CLK) then
            if HDU_OUTS.EXE_bubble='1' then
                CU_OUTS_EXE_atEXE<=NOP_outs.EXE;
                CU_OUTS_MEM_atEXE<=NOP_outs.MEM;
                CU_OUTS_WB_atEXE<=NOP_outs.WB;
            elsif HDU_OUTS.ID_EN='1' then
                CU_OUTS_EXE_atEXE<=CU_OUTS.EXE;
                CU_OUTS_MEM_atEXE<=CU_OUTS.MEM;
                CU_OUTS_WB_atEXE<=CU_OUTS.WB;
            end if;
            if HDU_OUTS.MEM_bubble='1' then
                CU_OUTS_MEM_atMEM<=NOP_outs.MEM;
                CU_OUTS_WB_atMEM<=NOP_outs.WB;
            elsif HDU_OUTS.EXE_EN='1' then
                CU_OUTS_MEM_atMEM<=CU_OUTS_MEM_atEXE;
                CU_OUTS_WB_atMEM<=CU_OUTS_WB_atEXE;
            end if;
            if HDU_OUTS.WB_bubble='1' then
                CU_OUTS_WB_atWB<=NOP_outs.WB;
            elsif HDU_OUTS.MEM_EN='1' then
                CU_OUTS_WB_atWB<=CU_OUTS_WB_atMEM;
            end if;
        end if;
    end process REGS_CU;

    CU_OUTS_pipelined<=(
        ID=>CU_OUTS.ID,
        EXE=>CU_OUTS_EXE_atEXE,
        MEM=>CU_OUTS_MEM_atMEM,
        WB=>CU_OUTS_WB_atWB
    );

    datapath_instance: datapath
        generic map(
            BPU_TAG_FIELD_SIZE=>BPU_TAG_FIELD_SIZE,
            BPU_SET_FIELD_SIZE=>BPU_SET_FIELD_SIZE,
            BPU_LINES_PER_SET=>BPU_LINES_PER_SET
        )
        port map(
            IRAM_ADDR=>IRAM_ADDR,
            IRAM_OUT=>IRAM_OUT,
            DRAM_ADDR=>DRAM_ADDR,
            DRAM_IN=>DRAM_IN,
            DRAM_OUT=>DRAM_OUT,
            control_from_CU=>CU_OUTS_pipelined,
            control_from_FU=>FU_OUTS,
            control_from_HDU=>HDU_OUTS,
            misprediction=>misprediction,
            CLK=>CLK,
            RST=>RST
        );

    DRAM_WR_EN<=CU_OUTS_pipelined.MEM.DRAM_WR_EN;

end architecture structural;
```

Listing C.14: a.a-datapath.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.datapath_package.all;  --for BRANCH_COND_type and constants
use work.CU_package.all;        --for control_from_CU_type
use work.ALU_package.all;       --for ALU_OP_type
use work.DLX_package.all;       --for word type and constants
use work.my_package.all;        --for log2_ceiling
use work.FU_and_HDU_package.all;        --for FU_outs_type and HDU_outs_type

entity datapath is
    generic(
        BPU_TAG_FIELD_SIZE  : integer :=8;
        BPU_SET_FIELD_SIZE  : integer :=3;
        BPU_LINES_PER_SET   : integer :=4
    );
    port(
        IRAM_ADDR       : out   word;
        IRAM_OUT        : in    word;
        DRAM_ADDR       : out   word;
        DRAM_IN         : out   word;
        DRAM_OUT        : in    word;
        control_from_CU : in    CU_OUTS_type;
        control_from_FU : in    FU_OUTS_type;
        control_from_HDU: in    HDU_OUTS_type;
        misprediction   : out   std_logic;
        CLK             : in    std_logic;
        RST             : in    std_logic
    );
end entity datapath;

architecture structural of datapath is

    component reg is
        Generic (N : positive:= 1 );                                --number of bits
        Port(   D       : In    std_logic_vector(N-1 downto 0);     --data input
                Q       : Out   std_logic_vector(N-1 downto 0);     --data output
                EN      : In    std_logic;                          --enable active high
                CLK     : In    std_logic;                          --clock
                RST     : In    std_logic);                         --asynchronous reset active
                    ↪ low
    end component reg;

    component RCA is
        generic (N:  integer := 8);                         --number of bits
        port (  A:  In  std_logic_vector(N-1 downto 0);     --data input 1
                B:  In  std_logic_vector(N-1 downto 0);     --data input 2
                Ci: In  std_logic;                          --carry in
                S:  Out std_logic_vector(N-1 downto 0);     --data output
                Co: Out std_logic);                         --carry out
    end component RCA;

    component  adder_P4 is
        GENERIC (N_BIT  : integer := 32);   --number of bits. Must be a number from 4 to 32 and a
            ↪ multiple of 4
        PORT   (A       : in  std_logic_vector(N_BIT - 1 downto 0);     -- input operand 1
                B       : in  std_logic_vector(N_BIT - 1 downto 0);     -- input operand 2
                add_sub : in  std_logic;                                -- carry-in
                Cout    : out std_logic;                                -- carry-out
                SUM     : out std_logic_vector(N_BIT -1 downto 0));     -- ouput sum
    end component adder_P4;

    component RF is
        generic(N_bit:      positive := 64;     --bitwidth
                N_reg:      positive := 32);    --number of address bits, the number of registers
                    ↪ is equal to 2**N_address
        port(   CLK:        IN std_logic;       --clock
                RST:        IN std_logic;       --asynchronous reset, active low
                WR_EN:      IN std_logic;       --synchronous write, active high
                ADD_WR:     IN std_logic_vector(log2_ceiling(N_reg)-1 downto 0);    --writing
                    ↪ register address
                ADD_RD1:    IN std_logic_vector(log2_ceiling(N_reg)-1 downto 0);    --reading
                    ↪ register address 1
                ADD_RD2:    IN std_logic_vector(log2_ceiling(N_reg)-1 downto 0);    --reading
                    ↪ register address 2
                DATA_IN:    IN std_logic_vector(N_bit-1 downto 0);      --data to write
                OUT1:       OUT std_logic_vector(N_bit-1 downto 0);     --data to read 1
                OUT2:       OUT std_logic_vector(N_bit-1 downto 0));    --data to read 2
    end component RF;

    component MUX_2to1 is
        Generic (N: integer:= 1);                               --number of bits
        Port (  IN0:    In  std_logic_vector(N-1 downto 0);     --data input 1
                IN1:    In  std_logic_vector(N-1 downto 0);     --data input 2
                SEL:    In  std_logic;                          --selection input
                Y:      Out std_logic_vector(N-1 downto 0));    --data output
    end component MUX_2to1;

    component MUX_4to1 is
    Generic (N: integer:= 1);   --number of bits
    Port (  IN0, IN1, IN2, IN3  : In    std_logic_vector(N-1 downto 0);     --data inputs
            SEL                 : In    std_logic_vector(1 downto 0);       --selection input
            Y                   : Out   std_logic_vector(N-1 downto 0));    --data output
    end component MUX_4to1;

    component MUX_8to1 is
        Generic (N: integer:= 1);   --number of bits
        Port (  IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7  : In    std_logic_vector(N-1 downto 0);
            ↪     --data inputs
                SEL                 : In    std_logic_vector(2 downto 0);       --selection input
                Y                   : Out   std_logic_vector(N-1 downto 0));    --data output
    end component MUX_8to1;

    component ALU is
        generic (N : integer := 32);                                    -- number of bit
        port(   FUNC            : IN ALU_OP_type;                       -- operation to do
                DATA1, DATA2    : IN std_logic_vector(N-1 downto 0);    -- data inputs
                OUT_ALU         : OUT std_logic_vector(N-1 downto 0));  -- data output
    end component ALU;

    component boothmul_4stage is
        generic(N: positive :=8);                           --number of input bits
        port(
            A, B: in std_logic_vector(N-1 downto 0);        --input operands
            EN: in std_logic;
            CLK: in std_logic;                          --clock signal
            RST: in std_logic;                          -- reset signal
            P: out std_logic_vector(2*N-1 downto 0)     --output product
        );
    end component boothmul_4stage;

    component branch_comp
        generic(N: integer:= 32);   --number of data-in bits
        port(
            BRANCH_COND     : in BRANCH_COND_type;                  --condition to take branch
            DATA_IN         : in std_logic_vector(N-1 downto 0);    --data to test
            BRANCH_IS_TAKEN : out std_logic);                       --high if the branch is taken
    end component branch_comp;

    component BPU is
        generic(
            TAG_FIELD_SIZE  : integer := 8;
            SET_FIELD_SIZE  : integer := 3;
            LINES_PER_SET   : integer := 4
        );
        port(
            clk             : in std_logic;
            rst             : in std_logic;
            instr_fetch     : in word;
            pc_fetch        : in word;
            pc_in           : in word;
```

```
            pc_out          : out word;
            misprediction   : out std_logic;
            actual_addr     : in word;
            IF_EN           : in std_logic;
            ID_EN           : in std_logic
        );
    end component BPU;

    signal PC_IN, PC_OUT, ADDER_BRANCH_out, NPC_BRANCH, NPC, NPC_ID, NPC_EXE, NPC_MEM, NPC_WB,
        ↪ actual_addr  : word;
    signal IR_in, IR_out, IR_out_ext16, IR_out_ext26
        ↪            : word;
    signal IMM_ID, IMM_EXE
        ↪            : word;
    signal RF_WR_ADDR_ID, RF_WR_ADDR_EXE, RF_WR_ADDR_MEM, RF_WR_ADDR_WB
        ↪            : RF_addr;
    signal RF_OUT1, RF_OUT1_fw, RF_OUT2, RF_OUT2_fw, RF_OUT2_EXE
        ↪            : word;
    signal ALU_IN1, ALU_IN2, ALU_OUT, ALU_OUT_EXE, ALU_OUT_MEM, ALU_OUT_WB
        ↪            : word;
    signal MULT_OUT
        ↪            : doubleword;
    signal branch_is_taken
        ↪            : std_logic;
    signal LMD_in, LMD_out
        ↪            : word;
    signal DRAM_IN_EXE, DRAM_IN_MEM, DRAM_OUT_sb, DRAM_OUT_ub, DRAM_OUT_sh, DRAM_OUT_uh,
        ↪ DRAM_OUT_w         : word;
    signal RF_DATA_IN
        ↪            : word;

    constant word_4: word:= std_logic_vector(to_unsigned(4,WORD_SIZE));

begin

    -------------------------------------IF stage

    IRAM_ADDR<=PC_out;
    IR_in<=IRAM_OUT;

    PC: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>PC_IN,
            Q=>PC_OUT,
            EN=>control_from_HDU.PC_EN,
            CLK=>clk,
            RST=>rst
        );

    ADDER_PC: RCA
        generic map(N=>WORD_SIZE)
        port map(
            A=>PC_out,
            B=>word_4,
            S=>NPC,
            Ci=>'0',
            Co=>open
        );

    BPU_instance: BPU
        generic map(
            TAG_FIELD_SIZE=>BPU_TAG_FIELD_SIZE,
            SET_FIELD_SIZE=>BPU_SET_FIELD_SIZE,
            LINES_PER_SET=>BPU_LINES_PER_SET
        )
        port map(
            clk=>clk,
            rst=>rst,
            instr_fetch=>IR_in,
            pc_fetch=>PC_OUT,
            pc_in=>NPC,
```

```vhdl
            pc_out=>PC_IN,
            misprediction=>misprediction,
            actual_addr=>actual_addr,
            IF_EN=>control_from_HDU.IF_EN,
            ID_EN=>control_from_HDU.ID_EN
        );

    REG_NPC_IF: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>NPC,
            Q=>NPC_ID,
            EN=>control_from_HDU.IF_EN,
            CLK=>clk,
            RST=>rst
        );

    IR: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>IR_in,
            Q=>IR_out,
            EN=>control_from_HDU.IF_EN,
            CLK=>clk,
            RST=>rst
        );

    -------------------------------------------------ID stage

    MUX_BRANCH: MUX_2to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>RF_OUT1_fw,
            IN1=>ADDER_BRANCH_out,
            SEL=>control_from_CU.ID.MUX_BRANCH_sel,
            Y=>NPC_BRANCH
        );

    ADDER_BRANCH: adder_P4
        generic map(N_BIT=>WORD_SIZE)
        port map(
            A=>NPC_ID,
            B=>IMM_ID,
            add_sub=>'0',
            Cout=>open,
            SUM=>ADDER_BRANCH_out
        );

    branch_comp_instance: branch_comp
        generic map(N=>WORD_SIZE)
        port map(
            BRANCH_COND=>control_from_CU.ID.BRANCH_COND,
            DATA_IN=>RF_OUT1_fw,
            BRANCH_IS_TAKEN=>branch_is_taken
        );

    MUX_actual_addr: MUX_2to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>NPC_ID,
            IN1=>NPC_branch,
            SEL=>branch_is_taken,
            Y=>actual_addr
        );

    MUX_RF_OUT1_fw: MUX_8to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>RF_OUT1,
            IN1=>ALU_OUT_EXE,
            IN2=>ALU_OUT_MEM,
            IN3=>LMD_IN,
            IN4=>RF_DATA_IN,
            IN5=>NPC_MEM,
            IN6=>NPC_EXE,
            IN7=>(others=>'-'),
            SEL=>control_from_FU.MUX_RF_OUT1_sel,
            Y=>RF_OUT1_fw
        );

    MUX_RF_OUT2_fw: MUX_8to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>RF_OUT2,
            IN1=>ALU_OUT_EXE,
            IN2=>ALU_OUT_MEM,
            IN3=>LMD_IN,
            IN4=>RF_DATA_IN,
            IN5=>NPC_MEM,
            IN6=>NPC_EXE,
            IN7=>(others=>'-'),
            SEL=>control_from_FU.MUX_RF_OUT2_sel,
            Y=>RF_OUT2_fw
        );

    REG_NPC_ID: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>NPC_ID,
            Q=>NPC_EXE,
            EN=>control_from_HDU.ID_EN,
            CLK=>clk,
            RST=>rst
        );

    MUX_RF_WR_ADDR: MUX_4to1
        generic map(N=>RF_ADDR_SIZE)
        port map(
            IN0=>IR_out(REG2_range),
            IN1=>IR_out(REG3_range),
            IN2=>RF_ADDR_LR,
            IN3=>(others=>'-'),
            SEL=>control_from_CU.ID.MUX_RF_WR_ADDR_sel,
            Y=>RF_WR_ADDR_ID
        );

    REG_RF_WR_ADDR_ID: reg
        generic map(N=>RF_ADDR_SIZE)
        port map(
            D=>RF_WR_ADDR_ID,
            Q=>RF_WR_ADDR_EXE,
            EN=>control_from_HDU.ID_EN,
            CLK=>clk,
            RST=>rst
        );

    RF_instance: RF
        generic map(
            N_bit=>WORD_SIZE,
            N_reg=>N_REG
        )
        port map(
            CLK=>clk,
            RST=>rst,
            WR_EN=>control_from_CU.WB.RF_WR_EN,
            ADD_WR=>RF_WR_ADDR_WB,
            ADD_RD2=>IR_out(REG2_range),
            ADD_RD1=>IR_out(REG1_range),
            DATA_IN=>RF_data_in,
            OUT1=>RF_OUT1,
            OUT2=>RF_OUT2
        );

    REG_RF_OUT1: reg
```

```vhdl
        generic map(N=>WORD_SIZE)
        port map(
            D=>RF_OUT1_fw,
            Q=>ALU_IN1,
            EN=>control_from_HDU.ID_EN,
            CLK=>clk,
            RST=>rst
        );

    REG_RF_OUT2: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>RF_OUT2_fw,
            Q=>RF_OUT2_EXE,
            EN=>control_from_HDU.ID_EN,
            CLK=>clk,
            RST=>rst
        );

    IR_out_ext16<=std_logic_vector(resize(signed(IR_out(IMM_range)), WORD_SIZE));
    IR_out_ext26<=std_logic_vector(resize(signed(IR_out(J_OFFSET_range)), WORD_SIZE));

    MUX_IMM_EXT: MUX_2to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>IR_out_ext16,
            IN1=>IR_out_ext26,
            SEL=>control_from_CU.ID.MUX_IMM_EXT_sel,
            Y=>IMM_ID
        );

    REG_IMM: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>IMM_ID,
            Q=>IMM_EXE,
            EN=>control_from_HDU.ID_EN,
            CLK=>clk,
            RST=>rst
        );

    ---------------------------------------EXE stage

    MUX_DRAM_IN_fw: MUX_2to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>RF_OUT2_EXE,
            IN1=>LMD_IN,
            SEL=>control_from_FU.MUX_DRAM_IN_sel,
            Y=>DRAM_IN_EXE
        );

    MUX_ALU_IN2: MUX_2to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>RF_OUT2_EXE,
            IN1=>IMM_EXE,
            SEL=>control_from_CU.EXE.MUX_ALU_IN2_sel,
            Y=>ALU_IN2
        );

    ALU_instance: ALU
        generic map(N=>WORD_SIZE)
        port map(
            FUNC=>control_from_CU.EXE.ALU_OP,
            DATA1=>ALU_IN1,
            DATA2=>ALU_IN2,
            OUT_ALU=>ALU_OUT
        );

    MULT: boothmul_4stage
        generic map(N=>WORD_SIZE)
        port map(
            A=>ALU_IN1,
            B=>ALU_IN2,
            EN=>control_from_CU.EXE.MULT_EN,
            CLK=>CLK,
            RST=>RST,
            P=>MULT_OUT
        );

    MUX_MULT: MUX_2to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>ALU_OUT,
            IN1=>MULT_OUT(WORD_range),
            SEL=>control_from_CU.EXE.MUX_MULT_sel,
            Y=>ALU_OUT_EXE
        );

    REG_ALU_OUT_EXE: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>ALU_OUT_EXE,
            Q=>ALU_OUT_MEM,
            EN=>control_from_HDU.EXE_EN,
            CLK=>clk,
            RST=>rst
        );

    REG_NPC_EXE: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>NPC_EXE,
            Q=>NPC_MEM,
            EN=>control_from_HDU.EXE_EN,
            CLK=>clk,
            RST=>rst
        );

    REG_DRAM_IN: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>DRAM_IN_EXE,
            Q=>DRAM_IN_MEM,
            EN=>control_from_HDU.EXE_EN,
            CLK=>clk,
            RST=>rst
        );

    REG_RF_WR_ADDR_EXE: reg
        generic map(N=>RF_ADDR_SIZE)
        port map(
            D=>RF_WR_ADDR_EXE,
            Q=>RF_WR_ADDR_MEM,
            EN=>control_from_HDU.EXE_EN,
            CLK=>clk,
            RST=>rst
        );

    ----------------------------------------------------------------MEM stage

    DRAM_ADDR<=ALU_OUT_MEM;
    DRAM_IN<=DRAM_IN_MEM;

    DRAM_OUT_sb<=std_logic_vector(resize(signed(DRAM_OUT(byte_range)), WORD_SIZE));
    DRAM_OUT_ub<=std_logic_vector(resize(unsigned(DRAM_OUT(byte_range)), WORD_SIZE));
    DRAM_OUT_sh<=std_logic_vector(resize(signed(DRAM_OUT(halfword_range)), WORD_SIZE));
    DRAM_OUT_uh<=std_logic_vector(resize(unsigned(DRAM_OUT(halfword_range)), WORD_SIZE));
    DRAM_OUT_w<=DRAM_OUT;

    MUX_DRAM_OUT_EXT: MUX_8to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>DRAM_OUT_sb,
            IN1=>DRAM_OUT_ub,
            IN2=>DRAM_OUT_sh,
            IN3=>DRAM_OUT_uh,
            IN4=>DRAM_OUT_w,
            IN5=>(others=>'-'),
            IN6=>(others=>'-'),
            IN7=>(others=>'-'),
            SEL=>control_from_CU.MEM.MUX_DRAM_OUT_EXT_sel,
            Y=>LMD_in
        );

    LMD: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>LMD_in,
            Q=>LMD_out,
            EN=>control_from_HDU.MEM_EN,
            CLK=>clk,
            RST=>rst
        );

    REG_NPC_MEM: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>NPC_MEM,
            Q=>NPC_WB,
            EN=>control_from_HDU.MEM_EN,
            CLK=>clk,
            RST=>rst
        );

    REG_ALU_OUT_MEM: reg
        generic map(N=>WORD_SIZE)
        port map(
            D=>ALU_OUT_MEM,
            Q=>ALU_OUT_WB,
            EN=>control_from_HDU.MEM_EN,
            CLK=>clk,
            RST=>rst
        );

    REG_RF_WR_ADDR_MEM: reg
        generic map(N=>RF_ADDR_SIZE)
        port map(
            D=>RF_WR_ADDR_MEM,
            Q=>RF_WR_ADDR_WB,
            EN=>control_from_HDU.MEM_EN,
            CLK=>clk,
            RST=>rst
        );

    ------------------------------------------------------------WB stage

    MUX_WB: MUX_4to1
        generic map(N=>WORD_SIZE)
        port map(
            IN0=>NPC_WB,
            IN1=>LMD_out,
            IN2=>ALU_OUT_WB,
            IN3=>(others=>'-'),
            SEL=>control_from_CU.WB.MUX_WB_sel,
            Y=>RF_data_in
        );

end architecture structural;
```

Listing C.15: a.b-CU.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.DLX_package.all;       --for word type, OPCODE_type, FUNC_type and constants
use work.ALU_package.all;       --for ALU_OP_type
use work.CU_package.all;        --for CU_OUTS_type and constants

entity CU is
    port(
        INSTR_ID    : in word;                  --instruction register output
        CU_OUTS     : out CU_outs_type          --CU outs to datapath
    );
end entity CU;

architecture behavioral of CU is

    constant LUT_RTYPE: LUT_RTYPE_type :=(
        FUNC_type'pos(FUNC_ADD)     =>  ADD_outs,
        FUNC_type'pos(FUNC_ADDU)    =>  ADDU_outs,
        FUNC_type'pos(FUNC_AND)     =>  AND_outs,
        FUNC_type'pos(FUNC_OR)      =>  OR_outs,
        FUNC_type'pos(FUNC_SGE)     =>  SGE_outs,
        FUNC_type'pos(FUNC_SLE)     =>  SLE_outs,
        FUNC_type'pos(FUNC_SLL)     =>  SLL_outs,
        FUNC_type'pos(FUNC_SNE)     =>  SNE_outs,
        FUNC_type'pos(FUNC_SRL)     =>  SRL_outs,
        FUNC_type'pos(FUNC_SUB)     =>  SUB_outs,
        FUNC_type'pos(FUNC_SUBU)    =>  SUBU_outs,
        FUNC_type'pos(FUNC_XOR)     =>  XOR_outs,
        FUNC_type'pos(FUNC_SGEU)    =>  SGEU_outs,
        FUNC_type'pos(FUNC_SGT)     =>  SGT_outs,
        FUNC_type'pos(FUNC_SGTU)    =>  SGTU_outs,
        FUNC_type'pos(FUNC_SLEU)    =>  SLEU_outs,
        FUNC_type'pos(FUNC_SLT)     =>  SLT_outs,
        FUNC_type'pos(FUNC_SLTU)    =>  SLTU_outs,
        FUNC_type'pos(FUNC_SRA)     =>  SRA_outs,
        FUNC_type'pos(FUNC_SEQ)     =>  SEQ_outs,
        FUNC_type'pos(FUNC_ROR)     =>  ROR_outs,
        FUNC_type'pos(FUNC_ROL)     =>  ROL_outs,
        FUNC_type'pos(FUNC_MULT)    =>  MULT_outs,
        others  => NOP_outs
    );

    constant LUT_ITYPE: LUT_ITYPE_type :=(
        OPCODE_type'pos(OPCODE_NOP)     =>  NOP_outs,
        OPCODE_type'pos(OPCODE_ADDI)    =>  ADDI_outs,
        OPCODE_type'pos(OPCODE_ADDUI)   =>  ADDUI_outs,
        OPCODE_type'pos(OPCODE_SUBI)    =>  SUBI_outs,
        OPCODE_type'pos(OPCODE_SUBUI)   =>  SUBUI_outs,
        OPCODE_type'pos(OPCODE_ANDI)    =>  ANDI_outs,
        OPCODE_type'pos(OPCODE_ORI)     =>  ORI_outs,
        OPCODE_type'pos(OPCODE_XORI)    =>  XORI_outs,
        OPCODE_type'pos(OPCODE_SGEI)    =>  SGEI_outs,
        OPCODE_type'pos(OPCODE_SGEUI)   =>  SGEUI_outs,
        OPCODE_type'pos(OPCODE_SGTI)    =>  SGTI_outs,
        OPCODE_type'pos(OPCODE_SGTUI)   =>  SGTUI_outs,
        OPCODE_type'pos(OPCODE_SLEI)    =>  SLEI_outs,
        OPCODE_type'pos(OPCODE_SLEUI)   =>  SLEUI_outs,
        OPCODE_type'pos(OPCODE_SLTI)    =>  SLTI_outs,
        OPCODE_type'pos(OPCODE_SLTUI)   =>  SLTUI_outs,
        OPCODE_type'pos(OPCODE_SLLI)    =>  SLLI_outs,
        OPCODE_type'pos(OPCODE_SRLI)    =>  SRLI_outs,
        OPCODE_type'pos(OPCODE_SRAI)    =>  SRAI_outs,
        OPCODE_type'pos(OPCODE_SNEI)    =>  SNEI_outs,
        OPCODE_type'pos(OPCODE_SEQI)    =>  SEQI_outs,
        OPCODE_type'pos(OPCODE_J)       =>  J_outs,
        OPCODE_type'pos(OPCODE_JR)      =>  JR_outs,
        OPCODE_type'pos(OPCODE_JAL)     =>  JAL_outs,
        OPCODE_type'pos(OPCODE_JALR)    =>  JAL_outs,
        OPCODE_type'pos(OPCODE_BEQZ)    =>  BEQZ_outs,
        OPCODE_type'pos(OPCODE_BNEZ)    =>  BNEZ_outs,
        OPCODE_type'pos(OPCODE_LW)      =>  LW_outs,
        OPCODE_type'pos(OPCODE_LH)      =>  LH_outs,
        OPCODE_type'pos(OPCODE_LHU)     =>  LHU_outs,
        OPCODE_type'pos(OPCODE_LB)      =>  LB_outs,
        OPCODE_type'pos(OPCODE_LBU)     =>  LBU_outs,
        OPCODE_type'pos(OPCODE_SW)      =>  SW_outs,
        OPCODE_type'pos(OPCODE_SH)      =>  SH_outs,
        OPCODE_type'pos(OPCODE_SB)      =>  SB_outs,
        OPCODE_type'pos(OPCODE_ROLI)    =>  ROLI_outs,
        OPCODE_type'pos(OPCODE_RORI)    =>  RORI_outs,
        OPCODE_type'pos(OPCODE_MULTI)   =>  MULTI_outs,
        others  => NOP_outs
    );

    signal OPCODE: OPCODE_type;
    signal FUNC: FUNC_type;

begin

    OPCODE<=get_opcode(INSTR_ID);
    FUNC<=get_func(INSTR_ID);

    CU_OUTS <=  LUT_RTYPE(to_integer(FUNC)) when OPCODE=OPCODE_RTYPE else
                LUT_ITYPE(to_integer(OPCODE));

end architecture behavioral;
```

Listing C.16: a.c-FU.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.DLX_package.all;       --for constants, types and functions
use work.FU_and_HDU_package.all;    --for FU_OUTS_type

entity FU is
    port(
        INSTR_ID    : in word;
        INSTR_EXE   : in word;
        INSTR_MEM   : in word;
        INSTR_WB    : in word;
        FU_OUTS     : out FU_OUTS_type
    );
end entity FU;

architecture behavioral of FU is

begin

    FU_process: process(INSTR_ID, INSTR_EXE, INSTR_MEM, INSTR_WB) is

        variable Rs1_ID, Rs2_ID, Rs2_EXE, Rd_EXE, Rd_MEM, Rd_WB : integer range 0 to N_REG;

    begin

        --compute usefull constants
        Rs1_ID  :=  get_Rs1(INSTR_ID);
        Rs2_ID  :=  get_Rs2(INSTR_ID);
        Rs2_EXE :=  get_Rs2(INSTR_EXE);
        Rd_EXE  :=  get_Rd(INSTR_EXE);
        Rd_MEM  :=  get_Rd(INSTR_MEM);
        Rd_WB   :=  get_Rd(INSTR_WB);

        --default assignment
        FU_OUTS.MUX_RF_OUT1_sel<=RF_forward_NO;
        FU_OUTS.MUX_RF_OUT2_sel<=RF_forward_NO;
        FU_OUTS.MUX_DRAM_IN_sel<=DRAM_forward_NO;
        --RF_OUT1 forwarding
        if Rs1_ID/=0 and Rs1_ID=Rd_EXE and is_not_a_jump(INSTR_EXE) and is_not_a_store(INSTR_EXE)
             ↪  then
            FU_OUTS.MUX_RF_OUT1_sel<=RF_forward_EXE_ALU;
        elsif Rs1_ID=31 and is_a_subroutine_call(INSTR_EXE) then
            FU_OUTS.MUX_RF_OUT1_sel<=RF_forward_EXE_NPC;
        elsif Rs1_ID/=0 and Rs1_ID=Rd_MEM and is_a_load(INSTR_MEM) then
            FU_OUTS.MUX_RF_OUT1_sel<=RF_forward_MEM_DRAM;
        elsif Rs1_ID/=0 and Rs1_ID=Rd_MEM and is_not_a_jump(INSTR_MEM) and is_not_a_store(
             ↪ INSTR_MEM) then
            FU_OUTS.MUX_RF_OUT1_sel<=RF_forward_MEM_ALU;
        elsif Rs1_ID=31 and is_a_subroutine_call(INSTR_MEM) then
            FU_OUTS.MUX_RF_OUT1_sel<=RF_forward_MEM_NPC;
        elsif Rs1_ID/=0 and Rs1_ID=Rd_WB and is_not_a_jump(INSTR_WB) and is_not_a_store(INSTR_WB)
             ↪  then
            FU_OUTS.MUX_RF_OUT1_sel<=RF_forward_WB;
        elsif Rs1_ID=31 and is_a_subroutine_call(INSTR_WB) then
            FU_OUTS.MUX_RF_OUT1_sel<=RF_forward_WB;
        end if;

        --RF_OUT2 forwarding
        if Rs2_ID/=0 and Rs2_ID=Rd_EXE and is_not_a_jump(INSTR_EXE) and is_not_a_store(INSTR_EXE)
             ↪  then
            FU_OUTS.MUX_RF_OUT2_sel<=RF_forward_EXE_ALU;
        elsif Rs2_ID=31 and is_a_subroutine_call(INSTR_EXE) then
            FU_OUTS.MUX_RF_OUT2_sel<=RF_forward_EXE_NPC;
        elsif Rs2_ID/=0 and Rs2_ID=Rd_MEM and is_a_load(INSTR_MEM) then
            FU_OUTS.MUX_RF_OUT2_sel<=RF_forward_MEM_DRAM;
        elsif Rs2_ID/=0 and Rs2_ID=Rd_MEM and is_not_a_jump(INSTR_MEM) and is_not_a_store(
             ↪ INSTR_MEM) then
            FU_OUTS.MUX_RF_OUT2_sel<=RF_forward_MEM_ALU;
        elsif Rs2_ID=31 and is_a_subroutine_call(INSTR_MEM) then
            FU_OUTS.MUX_RF_OUT2_sel<=RF_forward_MEM_NPC;
        elsif Rs2_ID/=0 and Rs2_ID=Rd_WB and is_not_a_jump(INSTR_WB) and is_not_a_store(INSTR_WB)
             ↪  then
            FU_OUTS.MUX_RF_OUT2_sel<=RF_forward_WB;
        elsif Rs2_ID=31 and is_a_subroutine_call(INSTR_WB) then
            FU_OUTS.MUX_RF_OUT2_sel<=RF_forward_WB;
        end if;

        --DRAM IN forwarding
        if Rs2_EXE/=0 then
            if Rs2_EXE=Rd_MEM and is_a_load(INSTR_MEM) then
                FU_OUTS.MUX_DRAM_IN_sel<=DRAM_forward_WB;
            end if;
        end if;

    end process FU_process;

end architecture behavioral;
```

Listing C.17: a.d-HDU.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.DLX_package.all;       --for constants, types and functions
use work.FU_and_HDU_package.all;    --for HDU_OUTS_type

entity HDU is   --hazard detection unit
    port(
        INSTR_ID         : in word;
        INSTR_EXE        : in word;
        misprediction    : in std_logic;
        HDU_OUTS         : out HDU_OUTS_type;
        clk              : in std_logic;
        rst              : in std_logic
    );
end entity HDU;
```

```
architecture behavioral of HDU is

    signal count: integer range 0 to 3;

begin

    HDU_proc: process(INSTR_ID, INSTR_EXE, misprediction, count) is

        variable Rs1_ID, Rs2_ID, Rd_EXE : integer range 0 to N_REG;

    begin

        --compute useful constants
        Rs1_ID  :=  get_Rs1(INSTR_ID);
        Rs2_ID  :=  get_Rs2(INSTR_ID);
        Rd_EXE  :=  get_Rd(INSTR_EXE);

        --default assignment
        HDU_OUTS.PC_EN      <='1';
        HDU_OUTS.IF_EN      <='1';
        HDU_OUTS.ID_EN      <='1';
        HDU_OUTS.EXE_EN     <='1';
        HDU_OUTS.MEM_EN     <='1';
        HDU_OUTS.WB_EN      <='1';
        HDU_OUTS.ID_bubble  <='0';
        HDU_OUTS.EXE_bubble <='0';
        HDU_OUTS.MEM_bubble <='0';
        HDU_OUTS.WB_bubble  <='0';

        --multicycle operations structural hazard
        if is_a_mult(INSTR_EXE) and count/=3 then
            HDU_OUTS.PC_EN<='0';
            HDU_OUTS.IF_EN<='0';
            HDU_OUTS.ID_EN<='0';
            HDU_OUTS.EXE_EN<='0';
            HDU_OUTS.MEM_bubble<='1';
        --load from DRAM data hazard
        elsif Rd_EXE/=0 and is_a_load(INSTR_EXE) and is_not_a_jump_imm(INSTR_ID) and        --
            ↪ immediate jumps does not need register file
        (Rs1_ID=Rd_EXE or (get_opcode(INSTR_ID)=OPCODE_RTYPE and Rs2_ID=Rd_EXE)) then       --
            ↪ stores can forward the value to store in the next pipeline stage but they need
            ↪ the register in which is cointain the address to proceed
            HDU_OUTS.PC_EN<='0';
            HDU_OUTS.IF_EN<='0';
            HDU_OUTS.ID_EN<='0';
            HDU_OUTS.EXE_bubble<='1';
        --branch misprediction control hazard
        elsif misprediction='1' then
            HDU_OUTS.IF_EN<='0';
            HDU_OUTS.ID_bubble<='1';
        end if;

    end process HDU_proc;

    count_proc: process(clk, rst)
    begin
        if rst='0' then
            count<=0;
        elsif rising_edge(clk) then
            if is_a_mult(INSTR_EXE) then
                if count=3 then
                    count<=0;
                else
                    count<=count+1;
                end if;
            end if;
        end if;
    end process count_proc;

end architecture behavioral;
```

Listing C.18: a.e-BPU.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.DLX_package.all;

entity BPU is
    generic(
        TAG_FIELD_SIZE   : integer := 8;
        SET_FIELD_SIZE   : integer := 3;
        LINES_PER_SET    : integer := 4
    );
    port(
        clk              : in std_logic;
        rst              : in std_logic;
        instr_fetch      : in word;
        pc_fetch         : in word;
        pc_in            : in word;
        pc_out           : out word;
        misprediction    : out std_logic;
        actual_addr      : in word;
        ID_EN            : in std_logic;     --fetch stage enable
        IF_EN            : in std_logic      --fetch stage enable
    );
end entity BPU;

architecture behavioral of BPU is

    constant N_SET: integer := 2**SET_FIELD_SIZE;
    constant DATA_SIZE: integer:= WORD_SIZE-2;

    subtype SET_range is integer range SET_FIELD_SIZE+1 downto 2;
    subtype TAG_range is integer range SET_FIELD_SIZE+TAG_FIELD_SIZE+1 downto SET_FIELD_SIZE+2;
    subtype DATA_range is integer range WORD_SIZE-1 downto 2;

    subtype SET_index_type is integer range 0 to N_SET-1;
    subtype LINE_index_type is integer range 0 to LINES_PER_SET-1;
    subtype LINE_index_type_ext is integer range -1 to LINES_PER_SET-1;
    subtype TAG_type is std_logic_vector(TAG_FIELD_SIZE-1 downto 0);
    subtype DATA_type is std_logic_vector(DATA_SIZE-1 downto 0);

    type CACHE_LINE_type is record
        TAG: TAG_type;
        DATA: DATA_type;
        YOUTH: LINE_index_type_ext;     --the higher this value, the more recently the data has
            ↪ been accessed, if equal to -1 this line is free
    end record CACHE_LINE_type;

    type CACHE_SET_type is array (LINE_index_type) of CACHE_LINE_type;

    type CACHE_type is array (SET_index_type) of CACHE_SET_type;

    signal cache : CACHE_type;

    --asyncronous signal used by read_proc
    signal prediction : word;
    signal hit_index : LINE_index_type_ext;

    --asynchronous signal used to read misprediction port
    signal misprediction_sig: std_logic;

    --synchronous signal used by write_proc
    signal last_pc_in : word;
    signal last_prediction : word;
    signal verify: std_logic;

    function is_a_branch(instr: word) return boolean is
        variable opcode: OPCODE_type;
    begin
        opcode:=get_opcode(instr);
        return opcode=OPCODE_BEQZ or opcode=OPCODE_BNEZ or opcode=OPCODE_JAL or opcode=OPCODE_J
            ↪ or opcode=OPCODE_JR or opcode=OPCODE_JALR;
    end function is_a_branch;

begin

    misprediction<=misprediction_sig;
    pc_out<=prediction;

    read_proc: process(cache, pc_in, instr_fetch, pc_fetch, misprediction_sig, actual_addr)
        variable set: SET_index_type;
        variable tag: TAG_type;
    begin
        --useful constants
        set:=to_integer(unsigned(pc_fetch(SET_range)));
        tag:=pc_fetch(TAG_range);
        --default assignments
        prediction<=pc_in;
        hit_index<= -1;
        if misprediction_sig='1' then
            prediction<=actual_addr;
        elsif is_a_branch(instr_fetch) then
            for i in 0 to LINES_PER_SET-1 loop
                if cache(set)(i).TAG=tag and cache(set)(i).YOUTH/=-1 then
                    hit_index<=i;
                    prediction<=cache(set)(i).DATA & "00";
                    exit;
                end if;
            end loop;
        end if;
    end process read_proc;

    write_proc: process(clk, rst)
        variable last_set: SET_index_type;
        variable last_tag: TAG_type;
        variable last_hit_index: LINE_index_type_ext;
        variable oldest_line: LINE_index_type;
        procedure update_youth(set: SET_index_type; index_to_update: LINE_index_type) is
            variable youth_to_update: LINE_index_type_ext;
        begin
            youth_to_update:=cache(set)(index_to_update).YOUTH;
            for i in 0 to LINES_PER_SET-1 loop
                if cache(set)(i).YOUTH>youth_to_update and cache(set)(i).YOUTH/=0 then
                    cache(set)(i).YOUTH<=cache(set)(i).YOUTH-1;
                end if;
            end loop;
            cache(set)(index_to_update).YOUTH<=LINES_PER_SET-1;
        end procedure update_youth;
        procedure get_oldest_line(set: SET_index_type) is
            variable smaller_youth: LINE_index_type_ext;
        begin
            smaller_youth:=LINES_PER_SET-1;
            oldest_line:=0;
            for i in 0 to LINES_PER_SET-1 loop
                if cache(set)(i).YOUTH<smaller_youth then
                    smaller_youth:=cache(set)(i).YOUTH;
                    oldest_line:=i;
                end if;
            end loop;
        end procedure get_oldest_line;
    begin
        if rst='0' then
            cache<=(others=>(others=>(TAG=>(others=>'0'), DATA=>(others=>'0'), YOUTH=>-1)));
            verify<='0';
            last_prediction<=(others=>'0');
            last_pc_in<=(others=>'0');
            last_set:=0;
            last_tag:=(others=>'0');
            last_hit_index:= -1;
        elsif rising_edge(clk) then
            if ID_EN='1' then
```

```
                if verify='1' then  --if there was a branch/jump fetch in the previous clock
                    ↪ cycle
                    if last_hit_index=-1 then   --if there was a cache miss, save the new address
                        ↪  (both if the prediction was correct or if it was not)
                        get_oldest_line(last_set);
                        cache(last_set)(oldest_line).TAG<=last_tag;
                        cache(last_set)(oldest_line).DATA<=actual_addr(DATA_range);
                        update_youth(last_set, oldest_line);
                    else    --if there was a cache hit
                        if misprediction_sig='1' then   --if there was a cache hit but a wrong
                            ↪ prediction update the correct address
                            cache(last_set)(last_hit_index).DATA<=actual_addr(DATA_range);
                            update_youth(last_set, last_hit_index);
                        else --if there was a cache hit and a correct prediction
                            update_youth(last_set, last_hit_index);
                        end if;
                    end if;
                    verify<='0';
                end if;
            end if;
            if IF_EN='1' then
                if is_a_branch(instr_fetch) and misprediction_sig='0' then
                    last_pc_in<= pc_in;
                    last_prediction<= prediction;
                    last_set:= to_integer(unsigned(pc_fetch(SET_range)));
                    last_tag:= pc_fetch(TAG_range);
                    last_hit_index:= hit_index;
                    verify<='1';
                end if;
            end if;
        end if;
    end process write_proc;

    misp_proc: process(verify, last_prediction, actual_addr)
    begin
        misprediction_sig<='0'; --default assignment
        if verify='1' and actual_addr/=last_prediction then
            misprediction_sig<='1';
        end if;
    end process misp_proc;

end architecture behavioral;
```

Listing C.19: a.a.a-ALU.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;
use work.ALU_package.all;   --for ALU_OP_type

entity ALU is
    generic (N : integer := 32);                                -- number of bit
    port(   FUNC            : IN ALU_OP_type;                   -- operation to do
            DATA1, DATA2    : IN std_logic_vector(N-1 downto 0);        -- data inputs
            OUT_ALU         : OUT std_logic_vector(N-1 downto 0));      -- data output
end ALU;

architecture BEHAVIORAL of ALU is

    COMPONENT barrel_shifter is
        generic (Nbit       : integer   := 32);                     -- number of bits
        port    (A          : IN std_logic_vector(Nbit-1 downto 0);         -- operand that we
            ↪ want modify
                B           : IN std_logic_vector(Nbit-1 downto 0);         -- number of position
                    ↪ of modification
                SHIFT_ROTATE: IN std_logic;                             -- select if it is a
                    ↪ shift=0 or a rotate=1 operation
                LOGIC_ARITH : IN std_logic;                             -- logic 0 arith 1
                LEFT_RIGHT  : IN std_logic;                             -- left 0 right 1
                OUTPUT      : OUT std_logic_vector(Nbit-1 downto 0));       -- operand modified
    END COMPONENT barrel_shifter;

    COMPONENT  adder_P4 IS
        GENERIC (N_BIT  : integer := 32);                               -- number of bits
        PORT    (A       : in  std_logic_vector(N_BIT - 1 downto 0);      -- input operand 1
                B       : in  std_logic_vector(N_BIT - 1 downto 0);      -- input operand 2
                add_sub : in  std_logic;                                 -- carry-in: 0 for ADD, 1
                    ↪  for SUB
                Cout    : out std_logic;                                 -- carry-out
                SUM     : out std_logic_vector(N_BIT -1 downto 0));      -- ouput sum
    END COMPONENT adder_P4;

    COMPONENT logic_block is
        Generic (N: integer:= 32);                                      -- number of bits
    Port (  SUM     :   In  std_logic_vector(N-1 downto 0);      -- sum provided by the sparse
         ↪ tree
            C_OUT   :   In  std_logic;                           -- carry out provided by sparse
                 ↪ tree
            A       :   In  std_logic_vector(N-1 downto 0);      -- input 1
            B       :   In  std_logic_vector(N-1 downto 0);      -- input 2
            A_GEU_B :   Out std_logic_vector(N-1 downto 0);      -- comparison A >= B for unsigned
                 ↪ , if true flag = 1, otherwise = 0
            A_GE_B  :   Out std_logic_vector(N-1 downto 0);      -- comparison A >= B for signed,
                 ↪ if true flag = 1, otherwise = 0
            A_GT_B  :   Out std_logic_vector(N-1 downto 0);      -- comparison A > B for signed,
                 ↪ if true flag = 1, otherwise = 0
            A_GTU_B :   Out std_logic_vector(N-1 downto 0);      -- comparison A > B for unsigned,
                 ↪  if true flag = 1, otherwise = 0
            A_LEU_B :   Out std_logic_vector(N-1 downto 0);      -- comparison A <= B for unsigned
                 ↪ , if true flag = 1, otherwise = 0
            A_LE_B  :   Out std_logic_vector(N-1 downto 0);      -- comparison A <= B for signed,
                 ↪ if true flag = 1, otherwise = 0
            A_LT_B  :   Out std_logic_vector(N-1 downto 0);      -- comparison A < B for signed,
                 ↪ if true flag = 1, otherwise = 0
            A_LTU_B :   Out std_logic_vector(N-1 downto 0);      -- comparison A < B for unsigned
                 ↪ , if true flag = 1, otherwise = 0
            A_NE_B  :   Out std_logic_vector(N-1 downto 0);      -- if A is different from B, if
                 ↪ true flag = 1, otherwise = 0
            A_EQ_B  :   Out std_logic_vector(N-1 downto 0);      -- if A is equal to B, if true
                 ↪ flag = 1, otherwise = 0
            NOT_A   :   Out std_logic_vector(N-1 downto 0);      -- make the bitwise not of the
                 ↪ input operand A
            A_AND_B :   Out std_logic_vector(N-1 downto 0);      -- make the bitwise AND between
                 ↪ input 1 and input 2
            A_XOR_B :   Out std_logic_vector(N-1 downto 0);      -- make the bitwise XOR between
                 ↪ input 1 and input 2
            A_OR_B  :   Out std_logic_vector(N-1 downto 0));     -- make the bitwise OR between
                 ↪ input 1 and input 2
    END COMPONENT logic_block;

    -- signals that manage the shifting type
    signal logic_arith, left_right,shift_rotate : std_logic;

    -- signal that manage the sparse tree
    signal add_sub      : std_logic;
         ↪

    -- ouput signal of component
    signal carry_out                        : std_logic;                        -- from adder
    signal out_st                           : std_logic_vector(N-1 downto 0);   -- from adder
    signal out_barrel_shifter               : std_logic_vector(N-1 downto 0);   -- from barrel
         ↪ shifter
    signal out_and,out_or,out_xor, out_not  : std_logic_vector(N-1 downto 0);   -- from logic
         ↪ block
    signal out_sge,out_sle                  : std_logic_vector(N-1 downto 0);      -- form logic
         ↪  block
    signal out_sgeu,out_sleu                : std_logic_vector(N-1 downto 0);      -- form logic
         ↪  block
    signal out_sgt,out_slt                  : std_logic_vector(N-1 downto 0);      -- form logic
         ↪  block
    signal out_sgtu,out_sltu                : std_logic_vector(N-1 downto 0);       -- form logic
        ↪  block
    signal out_seq,out_sne                  : std_logic_vector(N-1 downto 0);       -- form logic
        ↪  block

    BEGIN

        -- place the component for barrel shifter
        BRRL_SHFT : barrel_shifter
                    Generic map (N)
                    Port map(LOGIC_ARITH=>logic_arith, LEFT_RIGHT=>left_right,SHIFT_ROTATE=>
                        ↪ shift_rotate,
                            A=> DATA1, B=>DATA2, OUTPUT=> out_barrel_shifter);

        -- place the component for addition and subtraction
        SPARSE_TREE_ADDER: adder_P4
                    Generic map(N)
                    Port map(A=>DATA1, B=>DATA2, add_sub=>add_sub, Cout=>carry_out, SUM=>out_st);
                        ↪

        -- place the logic block, used to make all implemented logic function
        LGC_BLOCK: logic_block
                    Generic Map (N)
                    Port Map(SUM=>out_st, C_OUT=>carry_out,A=>DATA1,B=>DATA2,
                            A_GE_B=>out_sge ,A_GEU_B=>out_sgeu,
                            A_GT_B=>out_sgt ,A_GTU_B =>out_sgtu,
                            A_LE_B=>out_sle, A_LEU_B =>out_sleu,
                            A_LT_B  => out_slt, A_LTU_B => out_sltu,
                            A_NE_B=>out_sne, A_EQ_B=> out_seq, NOT_A => out_not,
                            A_AND_B=>out_and, A_XOR_B=>out_xor, A_OR_B=>out_or);

        -- statement that define the output of the alu
        with FUNC select OUT_ALU <=
            out_st  when ALU_ADD,
            out_st  when ALU_SUB,
            out_xor when ALU_XOR,
            out_and when ALU_AND,
            out_or  when ALU_OR,
            out_not when ALU_NOT,
            out_barrel_shifter when ALU_SLL,
            out_barrel_shifter when ALU_SRL,
            out_barrel_shifter when ALU_SRA,
            out_barrel_shifter when ALU_ROL,
            out_barrel_shifter when ALU_ROR,
            out_sge when ALU_SGE,
            out_sgeu when ALU_SGEU,
            out_sgt when ALU_SGT,
            out_sgtu when ALU_SGTU,
            out_sle when ALU_SLE,
            out_sleu when ALU_SLEU,
            out_slt when ALU_SLT,
            out_sltu when ALU_SLTU,
            out_sne when ALU_SNE,
            out_seq when ALU_SEQ,
            (others=>'-') when others;

        -- define if the operation needed a subtraction or an addition
        with FUNC select add_sub <=
            '0' when ALU_ADD,
            '1' when ALU_SUB,
            '1' when others;

        -- define if shift or rotate operation
        with FUNC select shift_rotate <=
            '0' when ALU_SLL,
            '0' when ALU_SRL,
            '0' when ALU_SRA,
            '1' when ALU_ROL,
            '1' when ALU_ROR,
            '-' when others;

        -- define if the operation is a shift
        with FUNC select logic_arith <=
            '0' when ALU_SLL,
            '0' when ALU_SRL,
            '1' when ALU_SRA,
            '1' when ALU_ROL,
            '1' when ALU_ROR,
            '-' when others;

        -- define the type of shift
        with FUNC select left_right <=
            '0' when ALU_SLL,
            '1' when ALU_SRL,
            '1' when ALU_SRA,
            '0' when ALU_ROL,
            '1' when ALU_ROR,
            '-' when others;

end BEHAVIORAL;
```

Listing C.20: a.a.a.a-barrel_shifter.vhd

```
library IEEE;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity barrel_shifter is
generic (Nbit       : integer   := 32);                         -- number of bits
port    (
        A           : IN std_logic_vector(Nbit-1 downto 0);         -- operand that we want
            ↪ modify
        B           : IN std_logic_vector(Nbit-1 downto 0);         -- number of position of
            ↪ modification
        SHIFT_ROTATE: IN std_logic;                                 -- select if it is a shift=0
            ↪ or a rotate=1 operation
        LOGIC_ARITH : IN std_logic;                                 -- logic 0 arith 1
        LEFT_RIGHT  : IN std_logic;                                 -- left 0 right 1
        OUTPUT      : OUT std_logic_vector(Nbit-1 downto 0));       -- operand modified
end  barrel_shifter;

architecture structural of  barrel_shifter is

    component MUX_2to1 is
        Generic (N: integer:= 1);                               --number of bits
        Port (  IN0:    In  std_logic_vector(N-1 downto 0);     --data input 1
                IN1:    In  std_logic_vector(N-1 downto 0);     --data input 2
                SEL:    In  std_logic;                          --selection input
                Y:      Out std_logic_vector(N-1 downto 0));    --data output
    end component MUX_2to1;

    component MUX_8to1 is
        Generic (N: integer:= 1);   --number of bits
        Port (  IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7  : In    std_logic_vector(N-1 downto 0);
            ↪     --data inputs
                SEL                     : In    std_logic_vector(2 downto 0);       --selection input
                Y                       : Out   std_logic_vector(N-1 downto 0));    --data output
    end component MUX_8to1;

    type matrix is array (Nbit/4 downto 1) of std_logic_vector(Nbit+3 downto 0);
    signal out_left, out_right, out_mux_right, out_first_stage: matrix ;

    -- signal to manage the mask creation
    signal zero_vector, MSB_vector,fill_vector_left, fill_vector_right : std_logic_vector(Nbit-1
        ↪ downto 0);

    -- signal used to propagate result from stage two to stage three
    signal out_second_stage : std_logic_vector(Nbit+3 downto 0);

    -- signal used to manage the mux of the third stage and the ouput stage
    signal out_barrel, out_control  : std_logic_vector(Nbit-1 downto 0);
    signal sel_mux_out              : std_logic;
    signal select_mux_3s            :std_logic_vector(2 downto 0);

    -- it holds only five bits of the B input signal because on 32 bit, it is possible has a max
        ↪ shift of 32 bit (log2(32) = 5)
    signal shift_pos                : std_logic_vector ( 4 downto 0);

BEGIN

    -- create a vector of all 0
    zero_vector <= (others => '0');

    -- Determine the mux control signal.
    -- If it is found that one bit is equal 1 between 31 and 6, means that the required shift is
        ↪ over 31 (max shift allowed)
    sel_mux_out<=not(   B(31) or B(30) or B(29) or B(28) or B(27) or B(26) or
                        B(25) or B(24) or B(23) or B(22) or B(21) or B(20) or
                        B(19) or B(18) or B(17) or B(16) or B(15) or B(14) or
                        B(13) or B(12) or B(11) or B(10) or B(9)  or B(8)  or
                        B(7)  or B(6));

    -- cut or saturate the signal provide on B port
    -- if a number major than 32 is detected, it saturate the output to the shifter
    cut_sat: process (B,LOGIC_ARITH,LEFT_RIGHT)
        begin
        if B > "00000000000000000000000000011111" then
                if LOGIC_ARITH='1' and LEFT_RIGHT='1' then -- if an arithmetic right shift is
                    ↪ required, it fill the output with all 1s
                    out_control<= (others=>'1');
                else
                    out_control<= (others=>'0');            -- in the other case with 0
                end if;
        else
            shift_pos <= B(4 downto 0); -- simple assignement if it is not detected a value
                ↪ greater than 31
            end if;
    end process cut_sat;

    -- it generates a signal with all the bits equal to the MSB. It is used in the arithmetical
        ↪ shift
    MSB_fill: for x in 0 to Nbit-1 generate
    MSB_vector (x)<= A(31);
    end generate MSB_fill;

    -- FIRST STAGE
        ↪ ----------------------------------------------------------------------------------------------------------
        ↪
    -- select if create the mask with zeros or with the bit received as operand for LEFT
        ↪ operation
     mux_shift_rotate_left: MUX_2to1
                            Generic map(N => Nbit )
                            Port map(IN0=> zero_vector, IN1=> A, SEL=> SHIFT_ROTATE, Y=>
                                ↪ fill_vector_left);

    -- create the eight possible general mask for left shift
    -- mask 0 : A(31 downto 0) & ("0000" or A(31 downto 28))
    -- mask 4 : A(27 downto 0) & ("00000000"or A(27 downto 24))
    -- ...other mask...
    -- mask 28: A(3 downto 0) & ("0000000000000000000000000000000" or A(31 downto 0))
    FIRST_STAGE_1: for x in 1 to Nbit/4 generate
        out_left(x) <= (A(35-4*x downto 0) & fill_vector_left(Nbit-1 downto Nbit-4*x));
    end generate FIRST_STAGE_1;


    -- select if create the mask with zeros or with the bit received as operand for RIGHT
        ↪ operation
    mux_shift_rotate_right: MUX_2to1
                            Generic map(N => Nbit )
                            Port map(IN0=> MSB_vector, IN1=> A, SEL=> SHIFT_ROTATE, Y=>
                                ↪ fill_vector_right);

    -- it select the type of filling. If rotate op must be at 1, as a ARITH operation
    FIRST_STAGE_2: for x in 1 to Nbit/4 generate
        right_MUX2to1 : MUX_2to1
                        Generic map(N => 4*x)
                        Port map(IN0=> (others=>'0'), IN1=> fill_vector_right(4*x-1 downto 0),
                            ↪ SEL=> LOGIC_ARITH, Y=> out_mux_right(x)(4*x-1 downto 0));
    end generate FIRST_STAGE_2;

    --  create the eight possible general mask for right shift
    -- if LOGIC shift:
    -- mask 0 : "0000" & A(31 downto 0)
    -- mask 4 : "00000000" & A(31 downto 4)
    -- ...other mask...
    -- mask 28: "00000000000000000000000000000000" & A(31 downto 28)
    -- if ARITHMETIC shift:
    -- mask 0 : ("1111" or A(3 downto 0)) & A(31 downto 0)
    -- mask 4 : ("11111111" or A(7 downto 0)) & A(31 downto 4)
    -- ...other mask...
    -- mask 28: ("11111111111111111111111111111111"or A(31 downto 0)) & A(31 downto 28)
    FIRST_STAGE_3: for x in 1 to Nbit/4 generate
        out_right(x) <= (out_mux_right(x)(4*x-1 downto 0) & A(Nbit-1 downto 4*x-4));
    end generate FIRST_STAGE_3;

    -- it select if it is a mask for right shift or a mask for left shift
    FIRST_STAGE_4: for x in 1 to Nbit/4 generate
        MASK_MUX2to1 : MUX_2to1
                    Generic map(N => Nbit + 4)
                    Port map(IN0=> out_left(x), IN1=> out_right(x), SEL=> LEFT_RIGHT, Y=>
                        ↪ out_first_stage(x));
    end generate FIRST_STAGE_4;

    -- SECOND STAGE
        ↪ ------------------------------------------------------------------------------------
    -- select the best mask approximation based on the 3 upper bits of shift_pos
    mux_second_stage: MUX_8to1
                    Generic map(N => Nbit + 4)
                    Port map(IN0=>out_first_stage(1),IN1=>out_first_stage(2),IN2=>out_first_stage
                        ↪ (3),
                    IN3=>out_first_stage(4),IN4=>out_first_stage(5),IN5=>out_first_stage(6),
                    IN6=>out_first_stage(7),IN7=>out_first_stage(8),SEL=>shift_pos(4 downto 2),Y
                        ↪ => out_second_stage);

    -- THIRD STAGE
        ↪ -------------------------------------------------------------------------------------
        ↪
    -- It select the right mask based on the last two bit of shift_pos and on the LEFT_RIGHT
        ↪ signal.
    -- if LEFT_RIGHT = LEFT it select one mask between input A and D
    -- if LEFT_RIGHT = RIGHT it select one mask between input E and H
    -- The different entries cut the input signal in order to provide the correct shifting
    select_mux_3s <= (LEFT_RIGHT & shift_pos(1) & shift_pos(0));
    mux_third_stage: MUX_8to1
                    Generic map(N => Nbit)
                    Port map(IN0=>out_second_stage(Nbit+3 downto 4),IN1=>out_second_stage(Nbit+2
                        ↪ downto 3),
                    IN2=>out_second_stage(Nbit+1 downto 2),IN3=>out_second_stage(Nbit downto 1),
                    IN4=>out_second_stage(Nbit -1 downto 0),IN5=>out_second_stage(Nbit downto 1),
                    IN6=>out_second_stage(Nbit+1 downto 2),IN7=>out_second_stage(Nbit+2 downto 3)
                        ↪ ,
                    SEL=>select_mux_3s,Y=> out_barrel);

    -- OUTPUT STAGE
        ↪ -------------------------------------------------------------------------------------
        ↪
    -- if sel_mux_out is = 0 -> pass A -> output obtained with the saturation
    -- if sel_mux_out is = 1 -> pass B -> output obtained with the three
    mux_out : MUX_2to1
            Generic map(N => Nbit)
            Port map(IN0=> out_control,IN1=> out_barrel,SEL=> sel_mux_out,Y=> OUTPUT);

end structural;
```

Listing C.21: a.a.a.b-logic_block.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;
use work.my_package.all;

entity logic_block is
    Generic (N: integer:= 32);                                   -- number of bits
    Port (  SUM     :   In  std_logic_vector(N-1 downto 0);      -- sum provided by the sparse
         ↪ tree
            C_OUT   :   In  std_logic;                           -- carry out provided by sparse
                ↪ tree
            A       :   In  std_logic_vector(N-1 downto 0);      -- input 1
            B       :   In  std_logic_vector(N-1 downto 0);      -- input 2
            A_GEU_B :   Out std_logic_vector(N-1 downto 0);      -- comparison A >= B for unsigned
                ↪ , if true flag = 1, otherwise = 0
            A_GE_B  :   Out std_logic_vector(N-1 downto 0);      -- comparison A >= B for signed,
                ↪ if true flag = 1, otherwise = 0
            A_GT_B  :   Out std_logic_vector(N-1 downto 0);      -- comparison A > B for signed,
                ↪ if true flag = 1, otherwise = 0
            A_GTU_B :   Out std_logic_vector(N-1 downto 0);      -- comparison A > B for unsigned,
                ↪  if true flag = 1, otherwise = 0
            A_LEU_B :   Out std_logic_vector(N-1 downto 0);      -- comparison A <= B for unsigned
                ↪ , if true flag = 1, otherwise = 0
            A_LE_B  :   Out std_logic_vector(N-1 downto 0);      -- comparison A <= B for signed,
                ↪ if true flag = 1, otherwise = 0
            A_LT_B  :   Out std_logic_vector(N-1 downto 0);      -- comparison A < B for signed,
                ↪ if true flag = 1, otherwise = 0
            A_LTU_B :   Out std_logic_vector(N-1 downto 0);      -- comparison A < B for unsigned
                ↪ , if true flag = 1, otherwise = 0
            A_NE_B  :   Out std_logic_vector(N-1 downto 0);      -- if A is different from B, if
                ↪ true flag = 1, otherwise = 0
            A_EQ_B  :   Out std_logic_vector(N-1 downto 0);      -- if A is equal to B, if true
                ↪ flag = 1, otherwise = 0
            NOT_A   :   Out std_logic_vector(N-1 downto 0);      -- make the bitwise not of the
                ↪ input operand A
            A_AND_B :   Out std_logic_vector(N-1 downto 0);      -- make the bitwise AND between
                ↪ input 1 and input 2
            A_XOR_B :   Out std_logic_vector(N-1 downto 0);      -- make the bitwise XOR between
                ↪ input 1 and input 2
            A_OR_B  :   Out std_logic_vector(N-1 downto 0));     -- make the bitwise OR between
                ↪ input 1 and input 2
end logic_block;

architecture Behavioural of logic_block is

signal nor_sum   : std_logic;
signal geu, ge   : std_logic;
signal gt, gtu   : std_logic;
signal leu, le   : std_logic;
signal lt, ltu   : std_logic;
signal neq, eq   : std_logic;
signal vector_zero:std_logic_vector(N-2 downto 0);
begin
    -- create the zero vector
    vector_zero <= (others=>'0');
    --simpler operation
    A_AND_B <= A and B;
    A_XOR_B <= A xor B;
    A_OR_B  <= A or  B;
    NOT_A   <= not(A);

    --nor all bits of the sum
    nor_sum <= nor_vector(sum);

    -- Now, thanks to the created signal nor_s3, it is possible make some comparison
    geu <= C_OUT;
    ge  <= C_OUT xor (A(N-1) xor B(N-1));
    gt  <= ge and neq;                                     -- ge = 1 and neq = 1
    gtu <= geu and neq;                                    -- geu = 1 and neq = 1
    leu <= nor_sum or (not(C_OUT));
    le  <= (nor_sum or (not(C_OUT))) xor (A(N-1) xor B(N-1));
    lt  <= le and neq;
    ltu <= leu and neq;

    neq <= not(nor_sum);
    eq  <= nor_sum;

    A_GEU_B <= vector_zero & geu;
    A_GE_B  <= vector_zero & ge;
    A_GT_B  <= vector_zero & gt;
    A_GTU_B <= vector_zero & gtu;

    A_LEU_B <= vector_zero & leu;
    A_LE_B  <= vector_zero & le;
    A_LT_B  <= vector_zero & lt;
    A_LTU_B <= vector_zero & ltu;

    A_NE_B  <= vector_zero & neq;
    A_EQ_B  <= vector_zero & eq;
end Behavioural;
```

Listing C.22: a.a.b-branch_comp.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use work.datapath_package.all;  --for BRANCH_COND_type

entity branch_comp is
    generic(N: integer:= 32);   --number of data-in bits
    port(
        BRANCH_COND     : in BRANCH_COND_type;                  --condition to take branch
        DATA_IN         : in std_logic_vector(N-1 downto 0);    --data to test
        BRANCH_IS_TAKEN : out std_logic);                       --high if the branch is taken
end entity branch_comp;

architecture behavioral of branch_comp is
begin

    branch_comparator_process: process(BRANCH_COND, DATA_IN)
    begin
        case BRANCH_COND is
            when BRANCH_NO =>
                BRANCH_IS_TAKEN<='0';
            when BRANCH_ALWAYS =>
                BRANCH_IS_TAKEN<='1';
            when BRANCH_EQZ =>
                if DATA_IN=(DATA_IN'range=>'0') then
                    BRANCH_IS_TAKEN<='1';
                else
                    BRANCH_IS_TAKEN<='0';
                end if;
            when BRANCH_NEZ =>
                if DATA_IN=(DATA_IN'range=>'0') then
                    BRANCH_IS_TAKEN<='0';
                else
                    BRANCH_IS_TAKEN<='1';
                end if;
        end case;
    end process branch_comparator_process;

end architecture behavioral;
```

Listing C.23: a.a.c-RF.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use work.my_package.all;    --for log2_ceiling

entity RF is
    generic(N_bit:      positive := 64;     --bitwidth
            N_reg:      positive := 32);    --number of registers
    port(   CLK:        IN std_logic;       --clock
            RST:        IN std_logic;       --asynchronous reset, active low
            WR_EN:      IN std_logic;       --synchronous write, active high
            ADD_WR:     IN std_logic_vector(log2_ceiling(N_reg)-1 downto 0);    --writing
                ↪ register address
            ADD_RD1:    IN std_logic_vector(log2_ceiling(N_reg)-1 downto 0);    --reading
                ↪ register address 1
            ADD_RD2:    IN std_logic_vector(log2_ceiling(N_reg)-1 downto 0);    --reading
                ↪ register address 2
            DATA_IN:    IN std_logic_vector(N_bit-1 downto 0);      --data to write
            OUT1:       OUT std_logic_vector(N_bit-1 downto 0);     --data to read 1
            OUT2:       OUT std_logic_vector(N_bit-1 downto 0));    --data to read 2
end entity RF;

architecture behavioral of RF is

    type    REG_ARRAY_TYPE is array(1 to N_reg-1) of std_logic_vector(N_bit-1 downto 0);    --
        ↪ array of registers type
    signal  REGISTERS : REG_ARRAY_TYPE;                                                      --
        ↪ registers instantiation

begin

    OUT1 <= REGISTERS(to_integer(unsigned(ADD_RD1))) when to_integer(unsigned(ADD_RD1))<=N_reg-1
        ↪ and to_integer(unsigned(ADD_RD1))>=1 else
            (others=>'0');

    OUT2 <= REGISTERS(to_integer(unsigned(ADD_RD2))) when to_integer(unsigned(ADD_RD2))<=N_reg-1
        ↪ and to_integer(unsigned(ADD_RD2))>=1 else
            (others=>'0');

    register_file_proc: process(CLK, RST)
    begin
        if RST='0' then
            REGISTERS <= (others=>(others=>'0'));
        elsif rising_edge(CLK) then
            if WR_EN = '1' then     --if write signal is active
                if (to_integer(unsigned(ADD_WR))>=1 and to_integer(unsigned(ADD_WR))<=N_reg-1)
                    ↪ then --and the write address is valid
                    REGISTERS(to_integer(unsigned(ADD_WR))) <= DATA_IN; --write register pointed
                        ↪ by ADD_WR with the value contained in DATAIN
                end if;
            end if;
        end if;
    end process;

end architecture behavioral;
```

Listing C.24: a.a.d-boothmul_4stage.vhd

```
library ieee;
library work;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;       --for types conversions and numeric functions
use work.my_package.all;        --for std_logic_vector_intrange

entity boothmul_4stage is
    generic(N: positive :=8);                       --number of input bits
    port(
        A, B: in std_logic_vector(N-1 downto 0);        --input operands
        EN: in std_logic;
        CLK: in std_logic;                          --clock signal
        RST: in std_logic;                          -- reset signal
        P: out std_logic_vector(2*N-1 downto 0)     --output product
    );
end entity boothmul_4stage;

architecture structural of boothmul_4stage is

    component MUX_8to1 is
        Generic (N: integer:= 1);   --number of bits
        Port (  IN0, IN1, IN2, IN3, IN4, IN5, IN6, IN7  : In    std_logic_vector(N-1 downto 0);
            ↪     --data inputs
                SEL                 : In    std_logic_vector(2 downto 0);       --selection input
                Y                   : Out   std_logic_vector(N-1 downto 0));    --data output
    end component MUX_8to1;

    component adder_P4 IS
    GENERIC (N_BIT  : integer := 32);   --number of bits. Must be a number from 4 to 32 and a
        ↪ multiple of 4
    PORT   (A       : in  std_logic_vector(N_BIT - 1 downto 0);     -- input operand 1
            B       : in  std_logic_vector(N_BIT - 1 downto 0);     -- input operand 2
            add_sub : in  std_logic;                                -- carry-in
            Cout    : out std_logic;                                -- carry-out
            SUM     : out std_logic_vector(N_BIT -1 downto 0));     -- ouput sum
    end component adder_P4;

    component booth_encoder is  --encoder to correctly drive booth multiplier MUXes
        Port (  input:  In  std_logic_vector_intrange(1 downto -1); --three bits of the number to
            ↪  multiplicate
                output: out std_logic_vector(2 downto 0));          --multiplexer input selector
    end component booth_encoder;

    type mux_out_type is array (N/2-1 downto 0) of std_logic_vector(2*N-1 downto 0);    --type
        ↪ used for signal mux_out
    type add_out_type is array (N/2-1 downto 0) of std_logic_vector(2*N-1 downto 0);    --type
        ↪ used for signal add_out
    type encoder_out_type is array (N/2-1 downto 0) of std_logic_vector(2 downto 0);    --type
        ↪ used for signal encoder_out
    type A_shifted_type is array (-N to N) of std_logic_vector(2*N-1 downto 0);         --type
        ↪ used for signal A_shifted

    signal encoder_out: encoder_out_type;                   --encoders outputs
    signal mux_out: mux_out_type;                           --MUXes outputs
    signal add_out_s1: add_out_type;                        --RCAs outputs
    signal add_out_s2: add_out_type;                        --RCAs outputs
    signal add_out_s3: add_out_type;
    signal add_out_s4: add_out_type;                        --RCAs outputs
    signal A_shifted: A_shifted_type;                       --signal that contains input A
        ↪ shifted to left of i-1 positions if i is positive,
                                                            --or contains -A shifted to left of |
                                                                ↪ i|-1 positions if i is
                                                                ↪ negative,
                                                            --where i is the array index.
                                                                ↪ A_shifted(0) contains all
                                                                ↪ zeros.
    signal B_ext: std_logic_vector_intrange(N-1 downto -1); --equal to input B but with the
        ↪ additional bit number -1, which is equal to 0.

begin
    -- process used to propagate the partial sum between the stages
    process(CLK, RST)
        begin
            if RST = '0' then                           -- asynchronous reset, active low
                add_out_s2(4) <=(others=>'0');
                add_out_s3(8) <=(others=>'0');
                add_out_s4(12)<=(others=>'0');
            elsif rising_edge(CLK) then                 -- positive edge triggered:
                if EN='1' then
                add_out_s2(4)   <= add_out_s1(4);       -- output of stage one, pass to stage
                    ↪  two
                add_out_s3(8)   <= add_out_s2(8);       -- output of stage two, pass to stage
                    ↪  three
                add_out_s4(12)  <= add_out_s3(12);      -- output of stage three, pass to
                    ↪ stage four
            end if;
        end if;
    end process;


    B_ext(-1)<='0';                             --connect bit -1 of B_ext to zero
    B_ext_generate: for i in 0 to N-1 generate  --connect all other bits to input B
        B_ext(i)<=B(i);
    end generate B_ext_generate;

    A_shifted_generate_positive: for i in 1 to N generate   --generate all A_shifted positive
         ↪ index
        A_shifted(i)<=std_logic_vector(shift_left(resize(signed(A), 2*N), (i-1)));
    end generate A_shifted_generate_positive;

    A_shifted(0)<=(others=>'0');    --connect A_shifted(0) to zero

    A_shifted_generate_negative: for i in -N to -1 generate --generate all A_shifted negative
         ↪ index
        A_shifted(i)<=std_logic_vector(shift_left(-resize(signed(A), 2*N), (-i-1)));
    end generate A_shifted_generate_negative;

    adder_generate_s1: for i in 1 to N/8 generate   --generate RCAs for stage 1
        adder: adder_P4 generic map(2*N)
            port map(
                A       =>  mux_out(i),
                B       =>  add_out_s1(i-1),
                add_sub =>  '0',
                SUM     =>  add_out_s1(i),
                Cout    =>  open );
    end generate adder_generate_s1;

    adder_generate_s2: for i in N/8+1 to N/4 generate   --generate RCAs for stage 1
        adder: adder_P4 generic map(2*N)
            port map(
                A       =>  mux_out(i),
                B       =>  add_out_s2(i-1),
                add_sub =>  '0',
                SUM     =>  add_out_s2(i),
                Cout    =>  open );
    end generate adder_generate_s2;

    adder_generate_s3: for i in N/4+1 to N/4+4 generate --generate RCAs for stage 1
        adder: adder_P4 generic map(2*N)
            port map(
                A       =>  mux_out(i),
                B       =>  add_out_s3(i-1),
                add_sub =>  '0',
                SUM     =>  add_out_s3(i),
                Cout    =>  open );
    end generate adder_generate_s3;

    adder_generate_s4: for i in N/4+5 to N/2-1 generate --generate RCAs for stage 4
        adder: adder_P4 generic map(2*N)
            port map(
                A       =>mux_out(i),
                B       =>add_out_s4(i-1),
                add_sub =>'0',
                SUM     =>add_out_s4(i),
                Cout    =>open );
    end generate adder_generate_s4;

    mux_generate: for i in 0 to N/2-1 generate  --generate all MUXes
        mux: MUX_8to1 generic map(2*N)
            port map(
                SEL=>encoder_out(i),
                Y=>mux_out(i),
                IN0=>A_shifted(0),
                IN1=>A_shifted(2*i+1),
                IN2=>A_shifted(-2*i-1),
                IN3=>A_shifted(2*i+2),
                IN4=>A_shifted(-2*i-2),
                IN5=>(others=>'-'),
                IN6=>(others=>'-'),
                IN7=>(others=>'-')
            );
    end generate mux_generate;

    encoder_generate: for i in 0 to N/2-1 generate  --generate all encoders
        encoder: booth_encoder
            port map(
                output=>encoder_out(i),
                input=>B_ext(2*i+1 downto 2*i-1)
            );
    end generate encoder_generate;

    add_out_s1(0)<=mux_out(0);  --there are no adder with index 0, the output of mux with index 0
        ↪  is directly connected to the second input of the adder with index 1

    P <= add_out_s4(N/2-1); --the final product is the output of the last adder

end architecture structural;
```

Listing C.25: a.a.d.a-booth_encoder.vhd

```
library ieee;
library work;
use ieee.std_logic_1164.all;
use work.my_package.all;    --for std_logic_vector_intrange

entity booth_encoder is --encoder to correctly drive booth multiplier MUXes
    Port (  input:  In  std_logic_vector_intrange(1 downto -1); --three bits of the number to
        ↪ multiplicate
            output: out std_logic_vector(2 downto 0));          --multiplexer input selector
end entity booth_encoder;

architecture behavioral of booth_encoder is

begin

    with input select
        output <=   "000" when "000",   --select 0
                    "001" when "001",   --select +A
                    "001" when "010",   --select +A
                    "011" when "011",   --select +2A
                    "100" when "100",   --select -2A
                    "010" when "101",   --select -A
                    "010" when "110",   --select -A
                    "000" when "111",   --select 0
                    "---" when others;  --don't care

end architecture behavioral;
```

Listing C.26: 05-adder_P4.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.my_package.all;            --for XOR operator between a std_logic and a std_logic_vector

-- Top level module of the Pentium 4 adder
```

```
ENTITY  adder_P4 IS
    GENERIC (N_BIT   : integer := 32);   --number of bits. Must be a number from 4 to 32 and a
        ↪ multiple of 4
    PORT   (A       : in  std_logic_vector(N_BIT - 1 downto 0);     -- input operand 1
            B       : in  std_logic_vector(N_BIT - 1 downto 0);     -- input operand 2
            add_sub : in  std_logic;                                -- carry-in
            Cout    : out std_logic;                                -- carry-out
            SUM     : out std_logic_vector(N_BIT -1 downto 0));     -- ouput sum
END ENTITY adder_P4;

architecture STRUCTURAL of adder_P4 is

    component sum_generator IS
        GENERIC(Nblocks         : positive := 8;    --number of carry select block
                bits_per_block  : positive := 4);   --number of bit per each block
                                                    --the number of input bits is equal to
                                                        ↪ Nblocks*bits_per_block
        PORT (  A               : in  std_logic_vector(bits_per_block*Nblocks - 1 downto 0);
            ↪ --data input 1
                B               : in  std_logic_vector(bits_per_block*Nblocks - 1 downto 0);
                    ↪ --data input 2
                CARRY_SELECT    : in  std_logic_vector(Nblocks-1 downto 0);
                    ↪ --carries from sparse tree
                SUM             : out std_logic_vector(bits_per_block*Nblocks - 1 downto 0));
                    ↪ --data output
    end component;

    component carry_generator IS
        GENERIC (Nbit   : positive := 32);                          --number of bits for the
            ↪ structure. Must be between 4 and 32 and a multiple of 4
        PORT   (    A       : in  std_logic_vector(Nbit - 1 downto 0);  --input operand 1
                    B       : in  std_logic_vector(Nbit - 1 downto 0);  --input operand 2
                    Cin : in  std_logic;                                --carry-in
                    Cout    : out std_logic_vector(Nbit/4 downto 0));   --carry-out generated by
                        ↪ the tree
    end component;

    signal tmp_co   : std_logic_vector(N_BIT/4  downto 0); -- used to connect the carries lines
        ↪ between the two modules
    signal B_xor    : std_logic_vector(N_BIT - 1 downto 0); -- B is changed if a subtraction
        ↪ incoming. If it is a sum, it doesn't change
BEGIN

    CLA_SPARSE_TREE: carry_generator
        GENERIC MAP (Nbit => N_BIT)
        PORT MAP  (A => A,  B => B_xor, Cin => add_sub, Cout => tmp_co);

    CSA: sum_generator
        GENERIC MAP(Nblocks=> N_BIT/4, bits_per_block => 4)
        PORT MAP(A => A, B => B_xor , CARRY_SELECT => tmp_co (N_BIT/4-1 downto 0), SUM => SUM);

    -- the most significant bit of the carries signal is the carry-out of the complete adder
    Cout <= tmp_co(N_BIT/4);

    -- Every bits of B pass trough a xor gates with the Cin. In this way, when a subtraction
        ↪ arrives, Cin is set to 1
    -- so it complements bit by bit the input B. If arrives a sum, nothing change and the input B
        ↪  is simply copied

    B_xor <= B xor add_sub;

end STRUCTURAL;
```

Listing C.27: 05.a-carry_generator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;    ----------------------------------------------SERVE?
use ieee.math_real.all;
use ieee.numeric_std.all;

-- CLA sparse tree module used in Pentium 4
ENTITY  carry_generator IS
    GENERIC (Nbit        : positive := 32);                             -- number of bits for the
        ↪  structure. Must be a power of two
    PORT   (A            : IN  std_logic_vector(Nbit - 1 downto 0);      --input operand 1
            B            : IN  std_logic_vector(Nbit - 1 downto 0);      --input operand 2
            Cin          : IN  std_logic;                                -- carry in
            Cout         : OUT std_logic_vector(Nbit/4 downto 0));       --carry out generated by
                ↪ the tree
END ENTITY carry_generator;

ARCHITECTURE STRUCTURAL OF carry_generator IS
    --propagate and generate network
    component PG_network IS
        PORT (
            op1               : in  std_logic;       --input bit 1
            op2               : in  std_logic;       --input bit 2
            g,p               : out std_logic);  --output propagate and generate
    end component;

    --general propagate block
    component PG_block IS
        PORT (
            P_ik , G_ik               : in  std_logic;
            P_k1j,G_k1j               : in  std_logic;
            P_ij, G_ij            : out std_logic);
    end component;

    --general generate block
    component G_block IS
        PORT (
            P_ik , G_ik               : in  std_logic;
            G_k1j                     : in  std_logic;
            G_ij                      : out std_logic);
    end component;

    -- We create two matrix, one for general propagate(p) and one for general generate(g)
    -- in this way, we have used the column to select the stage, and the row to select the
        ↪ product
    -- for example: g(0)(1) identify the general generate of the first element(1) provided by pg
        ↪ network(0)
    type matrix is array (4 downto 0) of std_logic_vector(Nbit downto 0);
    signal g, p : matrix ;

    -- temporal signal (tmp_p and tmp_g) are used only to semplify the generation of the last
        ↪ stage
    signal tmp_p,tmp_g : std_logic_vector(Nbit/4 -1 downto 0);

    -- This signal make easier the last stage implementation
    signal reference_gij : std_logic_vector((Nbit/4)-4 downto 0);

    -- co_tmp it is used to keep the temporal carry of the system
    signal co_tmp         : std_logic_vector(Nbit/4 - 1 downto 0);

    -- G_10 it is used only in case of carry in = 1. It modify the value provide to G block on
        ↪ first stage, after the pg network
    signal G_10: std_logic;


BEGIN
    -- particular case assignement
    p(0)(0) <= '0';
    g(0)(0) <= Cin;

    -- Create the pg network for the required number of bit. It is consider as stage 0, so the
        ↪ index of the column is 0.
    -- Stage 0
        ↪ ---------------------------------------------------------------------------------------------------------------
        ↪
    generation_PG_Network: for x in 1 to Nbit generate
```

```
        Block_PG_NET : PG_network
        PORT MAP(op1 => A(x-1), op2=>B(x-1), g=> g(0)(x), p=>p(0)(x));
    end generate generation_PG_Network;

    -- in case of carry in, g(0)(1) must be different (in lecture it is the value identify by G
         ↪ [1:0]), so:
    G_10 <= g(0)(1) or (p(0)(1) and g(0)(0));

    -- First stage
         ↪ ----------------------------------------------------------------------------------------------------------------
         ↪
    g_1 : G_block PORT MAP(P_ik => p(0)(2), G_ik =>g(0)(2) ,G_k1j => G_10, G_ij => g(1)(0));

    First_stage: for x in 1 to (Nbit-2)/2 generate
        Block_Stage_ONE : PG_block
        PORT MAP(P_ik => p(0)(x*2+2), G_ik =>g(0)(x*2+2) , P_k1j=>p(0)(x*2+1) , G_k1j =>g(0)(x
             ↪ *2+1) , P_ij => p(1)(x), G_ij =>g(1)(x) );
    end generate First_stage;

    -- Second stage
         ↪ ----------------------------------------------------------------------------------------------------------------
         ↪
    g_2 : G_block PORT MAP(P_ik => p(1)(1), G_ik =>g(1)(1) ,G_k1j => g(1)(0), G_ij => co_tmp(0));
         ↪

    Second_stage: for x in 1 to Nbit/4 -1 generate
        Block_Stage_TWO : PG_block
        PORT MAP( P_ik =>p(1)(x*2+1) , G_ik => g(1)(x*2+1), P_k1j=> p(1)(x*2), G_k1j =>g(1)(x*2)
             ↪ , P_ij =>p(2)(x), G_ij => g(2)(x));
    end generate Second_stage;

    -- Third Stage
         ↪ ----------------------------------------------------------------------------------------------------------------
         ↪
    if4: if Nbit>4 generate
        g_3 : G_block PORT MAP(P_ik =>p(2)(1), G_ik =>g(2)(1) ,G_k1j =>co_tmp(0), G_ij =>co_tmp
             ↪ (1));

        Third_stage: for x in 1 to (Nbit/8 -1 )generate
            Block_Stage_THREE : PG_block
            PORT MAP( P_ik => p(2)(x*2+1), G_ik =>  g(2)(x*2+1), P_k1j=>  p(2)(x*2), G_k1j =>  g
                 ↪ (2)(x*2), P_ij =>p(3)(x), G_ij =>g(3)(x) );
        end generate Third_stage;

    end generate if4;

    -- Fourth Stage
         ↪ ----------------------------------------------------------------------------------------------------------------
         ↪
    if8: if Nbit> 8 generate
        stage_c12_c16: for x in 0 to 1 generate
            g_4_c12_c16: G_block PORT MAP(P_ik =>p(2+x)(2-x), G_ik =>g(2+x)(2-x) ,G_k1j =>co_tmp
                 ↪ (1), G_ij =>co_tmp(2+x));
        end generate stage_c12_c16;
    end generate if8;

    if16_1: if Nbit>= 16 generate
        loopK: for k in 2 to Nbit/16 generate
            Fourth_stage : for x in 0 to 1 generate
                Block_stage_FOUR : PG_block PORT MAP( P_ik =>p(x+2)((-2+4*k)/(x+1)), G_ik =>g(x
                     ↪ +2)((-2+4*k)/(x+1)) , P_k1j=>p(3)(2*(k-1)), G_k1j => g(3)(2*(k-1)), P_ij
                     ↪ =>p(4)(x+2*k-2), G_ij =>g(4)(x+2*k-2) );
            end generate Fourth_stage;
        end generate loopK;
    end generate if16_1;
    --
         ↪ ----------------------------------------------------------------------------------------------------------------
         ↪
    -- Fifth stage.
    -- This are used to create a order sequence of p and g operand, so the last stage is simpler
    if16_2: if (Nbit>16) generate
        create_vector: for x in 1 to integer(log2(real(Nbit)))-4 generate
```

```
            tmp_p (4*x-1 downto 4*x-4) <=  ( p(4)(2*x+1) & p(4)(2*x) & p(3)(2*x) & p(2)(4*x));
            tmp_g (4*x-1 downto 4*x-4) <=  ( g(4)(2*x+1) & g(4)(2*x) & g(3)(2*x) & g(2)(4*x));
        end generate create_vector;
    end generate if16_2;

    -- The two following if, are used to create the PG block that preceding the G block that
        ↪ generate the carry52,carry56,carry60 and carry64
    if64_1: if Nbit>32 generate
        Stage_52_56 : for x in 0 to 1 generate
            Block_carry52_carry56 : PG_block PORT MAP( P_ik =>p(2+x)(12/(x+1)), G_ik =>g(2+x)
                ↪ (12/(x+1)) , P_k1j=>p(4)(5), G_k1j => g(4)(5), P_ij =>tmp_p(8+x), G_ij =>
                ↪ tmp_g(8+x) );
        end generate Stage_52_56;
    end generate if64_1;

    if64_2: if Nbit>32 generate
        Stage_60_64 : for x in 0 to 1 generate
            Block_carry60_carry64 : PG_block PORT MAP( P_ik =>p(4)(6+x), G_ik =>g(4)(6+x) , P_k1j
                ↪ =>p(4)(5), G_k1j => g(4)(5), P_ij =>tmp_p(10+x), G_ij =>tmp_g(10+x) );
        end generate Stage_60_64;
    end generate if64_2;

    -- i value is:
    -- 0 for 32 bit
    -- 1 for 64 bit
    -- This is used to create a vector with some value repeated.
    ------------------------------------ Example: if Nbit = 64 it creates:
    -- (co_tmp(7) & co_tmp(7) & co_tmp(7) & co_tmp(7) & co_tmp(7) & co_tmp(7) & co_tmp(7) &
        ↪ co_tmp(7) & co_tmp(3) & co_tmp(3) & co_tmp(3) & co_tmp(3))
    -- co_tmp(7) is used by all 8 block of the last stage between 32 and 62, while co_tmp(3) is
        ↪ used by the 4 block between 16 and 32
    -- k starts from 1 and not from 0
    if16_3: if Nbit > 16 generate
        selection_i : for i in 0 to integer(log2(real(Nbit)))-5 generate
            range_selection: for k in 4*(2**(i+1)-1)-(2**(i+2)-1) to 4*(2**(i+1)-1) generate
                reference_gij(k) <= co_tmp(-1+2**(i+2));
            end generate range_selection;
        end generate selection_i;
    end generate if16_3;

     -- This is used to generate the last stage of G block
    if16_4: if (Nbit>16) generate
        Fifth_stage : for x in 4 to Nbit/4 -1 generate
            Block_stage_FIVE : G_block PORT MAP(P_ik =>tmp_p(x-4), G_ik =>tmp_g(x-4) ,G_k1j =>
                ↪ reference_gij(x-3), G_ij => co_tmp(x));
        end generate Fifth_stage;
    end generate if16_4;
    -- END OF STAGES
        ↪ ------------------------------------------------------------------------------------------
        ↪

    -- Now, the carries obtained from the various stages are assigned, with also the
        ↪ concatenation of the carry in, to the output of the tree
    Cout <= co_tmp & Cin;

end STRUCTURAL;
```

Listing C.28: 05.a.a-PG_network.vhd

```
library ieee;
use ieee.std_logic_1164.all;

--Create the two starting signal for each pair of bit

ENTITY PG_network IS
    PORT (
        op1 : in  std_logic;     --input bit 1
        op2 : in  std_logic;     --input bit 2
        g, p: out std_logic);    --output propagate and generate
```

```
END ENTITY PG_network;

architecture STRUCTURAL of PG_network is
BEGIN

    g <= op1 and op2;   --generate
    p <= op1 xor op2;   --propagate

end STRUCTURAL;
```

Listing C.29: 05.b-sum_generator.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

ENTITY  sum_generator IS
    GENERIC(Nblocks         : positive := 8;     --number of carry select block
            bits_per_block  : positive := 4);    --number of bit per each block
                                                 --the number of input bits is equal to Nblocks*
                                                      ↪ bits_per_block
    PORT (
      A             : in  std_logic_vector(bits_per_block*Nblocks - 1 downto 0);    --data input
          ↪ 1
      B             : in  std_logic_vector(bits_per_block*Nblocks - 1 downto 0);    --data input
          ↪ 2
      CARRY_SELECT  : in  std_logic_vector(Nblocks-1 downto 0);                     --carries
          ↪ from sparse tree
      SUM           : out std_logic_vector(bits_per_block*Nblocks - 1 downto 0));   --data output
END ENTITY sum_generator;

architecture STRUCTURAL of sum_generator is

   component carry_select_block is
    generic(N: positive := 4);                             --number of bits of the block
    port(   A, B    :   in std_logic_vector(N-1 downto 0);      --data inputs
            S       :   out std_logic_vector(N-1 downto 0);     --data output
            Ci      :   in std_logic);                          --block carry-in
  end component;

BEGIN

    gen_block: for i in 1 to Nblocks generate   --generate "Nblocks" carry select blocks
    block_n : carry_select_block
        GENERIC MAP (N =>bits_per_block)         --each with "bits_per_block" bits
        PORT MAP(   A=> A((bits_per_block*i)-1 downto (i-1)*bits_per_block),        --split input
            ↪  A in "Nblocks" sub-signal each of "bits_per_block" bits and connect each sub-
            ↪ signal to each block input A
                    B=> B((bits_per_block*i)-1 downto (i-1)*bits_per_block),        --idem for
                        ↪ input B
                    S=> SUM((bits_per_block*i)-1 downto (i-1)*bits_per_block),      --put
                        ↪ together all the outputs of the singles blocks to the output of the
                        ↪ sum_generator
                    Ci => CARRY_SELECT(i-1));                                       --connect all
                        ↪  the input carries from the carry generator to the carry select of
                        ↪ each blocks
    end generate gen_block;

end STRUCTURAL;
```

Listing C.30: 05.b.a-carry_select_block.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity carry_select_block is
```

```
    generic(N: positive := 4);                              --number of bits of the block
    port(   A, B:   in std_logic_vector(N-1 downto 0);      --data inputs
            S:      out std_logic_vector(N-1 downto 0);     --data output
            Ci:     in std_logic);                          --block carry-in
end entity carry_select_block;

architecture structural of carry_select_block is

    component MUX_2to1 is
        Generic (N: integer:= 1);                               --number of bits
        Port (  IN0:    In  std_logic_vector(N-1 downto 0);     --data input 1
                IN1:    In  std_logic_vector(N-1 downto 0);     --data input 2
                SEL:    In  std_logic;                          --selection input
                Y:      Out std_logic_vector(N-1 downto 0));    --data output
    end component MUX_2to1;

    component RCA is
        generic (N:  integer := 8);                         --number of bits
        Port (  A:  In  std_logic_vector(N-1 downto 0);     --data input 1
                B:  In  std_logic_vector(N-1 downto 0);     --data input 2
                Ci: In  std_logic;                          --carry-in
                S:  Out std_logic_vector(N-1 downto 0);     --data output
                Co: Out std_logic);                         --carry-out
    end component RCA;

    signal S_0: std_logic_vector(N-1 downto 0);             --sum if carry-in is 0
    signal S_1: std_logic_vector(N-1 downto 0);             --sum if carry-in is 1

begin

    --sum with carry-in=0
    RCA_0: RCA  generic map(N)
                port map(A=>A, B=>B, Ci=>'0', S=>S_0, Co=>open);

    --sum with carry-in=1
    RCA_1: RCA  generic map(N)
                port map(A=>A, B=>B, Ci=>'1', S=>S_1, Co=>open);

    --select the actual sum depending on the effective carry-in
    SUM_SELECT_MUX:  MUX_2to1   generic map(N)
                                port map(IN0=>S_0, IN1=>S_1, SEL=>Ci, Y=>S);

end architecture structural;
```

APPENDIX D

Script

Listing D.1: final_script.tcl

```
#*******************************************************************************
# This script is used to synthesize the DLX
#*******************************************************************************
#analyze all possible files contained in the work folder
analyze {.} -autoread

#elaborate top entity, set the variable
elaborate DLX -architecture structural -library WORK -parameters "BPU_TAG_FIELD_SIZE = 8 ,
     ↪ BPU_SET_FIELD_SIZE = 3, BPU_LINES_PER_SET = 4"
#wires model
set_wire_load_model -name 5K_hvratio_1_4
#**************** CONSTRAINT THE SYNTHESIS ***********************
#timing constraint. WCP holds the minimum value of the delay
set WCP 2.43
# create a clock signal with a period equal to the worst critical path
create_clock -name "CLOCK" -period $WCP CLK
#forces a combinational max delay from each of the inputs
#to each of th output in case combinational paths are present
set_max_delay $WCP -from [all_inputs] -to [all_outputs]
#clock gating setting
set max_fanout 32
set minbit 1
set_clock_gating_style -minimum_bitwidth $minbit -max_fanout $max_fanout -control_point before -
     ↪ positive_edge_logic {latch and}

#compile ultra command, in order to perform an exact synthesis
#no_autoungroup is put in order to doesn't lost the hierarchy of the design
compile_ultra -timing_high_effort_script -no_autoungroup -gate_clock

#function to write the first 1000 critical path found
proc custom_report {} {
  echo [format "%-20s %-20s %7s" "From" "To" "Slack"]
     echo "--------------------------------------------------------"
      foreach_in_collection path [get_timing_paths -nworst 1000] {
        set slack [get_attribute $path slack]
        set startpoint [get_attribute $path startpoint]
        set endpoint [get_attribute $path endpoint]
        echo [format "%-20s %-20s %s" [get_attribute $startpoint full_name] \
             [get_attribute $endpoint full_name] $slack]
                        }
            }
#write the reports
report_timing > timing_DLX.rpt
report_area > area_DLX.rpt
report_power > power_DLX.rpt
# report that holds all the information for each block
report_area -hierarchy > area_DLX.rpt
```

```
report_power -hierarchy > power_DLX.rpt

#********************************************************
#write the netlist in verilog and vhdl languages
write -hierarchy -format verilog -output p4add_32bit_post_syn.v
write -hierarchy -format vhdl -output p4add_32bit_post_syn.vhdl
#generate the SDC file subsequently used in the Place and Route phase
write_sdc P4ADD.sdc
```

Listing D.2: area_script.tcl

```
# This script is used to synthesize the DLX from the point of view of area
#*****************************************************************************
#analyze files contained in the work folder
analyze {.} -autoread

#elaborate top entity, set the variable
elaborate DLX -architecture structural -library WORK -parameters "BPU_TAG_FIELD_SIZE = 8 ,
     ↪ BPU_SET_FIELD_SIZE = 3, BPU_LINES_PER_SET = 4"
#wires model
set_wire_load_model -name 5K_hvratio_1_4
#**************** CONSTRAINT THE SYNTHESIS **********************
# variable which is set to 20, 10, 5, 3, and 2.4 in order to see the variation
set WCP 20
# create a clock signal with a period equal to the worst critical path
create_clock -name "CLOCK" -period $WCP CLK
#forces a combinational max delay from each of the inputs
#to each of the output
set_max_delay $WCP -from [all_inputs] -to [all_outputs]
#area constraint
set_max_area 0.0
#lost every hierarchy of the circuit
ungroup -all -flatten
#compile ultra with the area ottimization constraint
compile_ultra -area_high_effort_script
#write report
report_timing > timing_area.rpt
report_area > area_area.rpt
report_power > power_area.rpt
#******************************************************************
```

Listing D.3: power_script.tcl

```
#*****************************************************************************
# This script is used to synthesize the DLX from the point of view of power
#*****************************************************************************
#analyze files contained in the work folder
analyze {.} -autoread
#elaborate top entity
elaborate DLX -architecture structural -library WORK -parameters "BPU_TAG_FIELD_SIZE = 8 ,
     ↪ BPU_SET_FIELD_SIZE = 3, BPU_LINES_PER_SET = 4"
#wires model
set_wire_load_model -name 5K_hvratio_1_4
#**************** CONSTRAINT THE SYNTHESIS ***********************
#give the constraint on dynamic and leakage power consumption
set_dynamic_optimization true
set_leakage_optimization true
# variable which is set to 20, 10, 5, 3, and 2.4 in order to see the variation
set WCP 20
# create a clock signal with a period equal to WCP
create_clock -name "CLOCK" -period $WCP CLK
#forces a combinational max delay from each of the inputs
#to each of the output
set_max_delay $WCP -from [all_inputs] -to [all_outputs]
#clock gating style
set max_fanout 32
set minbit 1
set_clock_gating_style -minimum_bitwidth $minbit -max_fanout $max_fanout -control_point before -
    ↪ positive_edge_logic {latch and}
#compile  ultra with clock gating option
compile_ultra -gate_clock
#write report
report_timing > timing_power.rpt
report_area > area_power.rpt
report_power > power_power.rpt
#*******************************************************************
```

APPENDIX E

Report

Listing E.1: timing_DLX.rpt

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : DLX_BPU_TAG_FIELD_SIZE8_BPU_SET_FIELD_SIZE3_BPU_LINES_PER_SET4
Version: F-2011.09-SP3
****************************************

 # A fanout number of 1000 was used for high fanout net computations.

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

  Startpoint: CU_OUTS_EXE_atEXE_reg[ALU_OP][4]
              (rising edge-triggered flip-flop clocked by CLOCK)
  Endpoint: datapath_instance/BPU_instance/clk_gate_cache_reg[0][2][DATA]/latch
            (negative level-sensitive latch clocked by CLOCK)
  Path Group: CLOCK
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  DLX_BPU_TAG_FIELD_SIZE8_BPU_SET_FIELD_SIZE3_BPU_LINES_PER_SET4
                     5K_hvratio_1_4        NangateOpenCellLibrary

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock CLOCK (rise edge)                                 0.00       0.00
  clock network delay (ideal)                             0.00       0.00
  CU_OUTS_EXE_atEXE_reg[ALU_OP][4]/CK (DFFS_X1)           0.00       0.00 r
  CU_OUTS_EXE_atEXE_reg[ALU_OP][4]/QN (DFFS_X1)           0.07       0.07 f
  datapath_instance/control_from_CU[EXE][ALU_OP][4] (
       ↪ datapath_BPU_TAG_FIELD_SIZE8_BPU_SET_FIELD_SIZE3_BPU_LINES_PER_SET4)
                                                          0.00       0.07 f
  datapath_instance/ALU_instance/FUNC[4] (ALU_N32)        0.00       0.07 f
  datapath_instance/ALU_instance/U44/ZN (INV_X1)          0.04       0.11 r
  datapath_instance/ALU_instance/U31/ZN (AND2_X1)         0.04       0.15 r
  datapath_instance/ALU_instance/U38/ZN (NAND3_X1)        0.04       0.19 f
  datapath_instance/ALU_instance/SPARSE_TREE_ADDER/add_sub (ADDER_P4_N_BIT32_1)
                                                          0.00       0.19 f
  datapath_instance/ALU_instance/SPARSE_TREE_ADDER/U18/ZN (INV_X1)
                                                          0.05       0.24 r
  datapath_instance/ALU_instance/SPARSE_TREE_ADDER/U3/ZN (INV_X1)
                                                          0.06       0.30 f
  datapath_instance/ALU_instance/SPARSE_TREE_ADDER/U23/Z (XOR2_X1)
                                                          0.12       0.42 f
```

```
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/B[17] (
    ↪ CARRY_GENERATOR_Nbit32_1)
                                                    0.00       0.42 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_PG_NET_18/op2 (
    ↪ PG_NETWORK_975)
                                                    0.00       0.42 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_PG_NET_18/U2/Z (XOR2_X1)
                                                    0.09       0.51 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_PG_NET_18/p (
    ↪ PG_NETWORK_975)
                                                    0.00       0.51 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_ONE_8/P_ik (
    ↪ PG_BLOCK_965)
                                                    0.00       0.51 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_ONE_8/U2/ZN (
    ↪ AOI21_X1)
                                                    0.05       0.57 r
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_ONE_8/U3/ZN (
    ↪ INV_X1)
                                                    0.03       0.59 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_ONE_8/G_ij (
    ↪ PG_BLOCK_965)
                                                    0.00       0.59 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_TWO_4/G_k1j (
    ↪ PG_BLOCK_954)
                                                    0.00       0.59 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_TWO_4/U2/ZN (
    ↪ AOI21_X1)
                                                    0.04       0.63 r
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_TWO_4/U3/ZN (
    ↪ INV_X1)
                                                    0.03       0.66 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_TWO_4/G_ij (
    ↪ PG_BLOCK_954)
                                                    0.00       0.66 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_THREE_2/G_k1j (
    ↪ PG_BLOCK_949)
                                                    0.00       0.66 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_THREE_2/U2/ZN (
    ↪ AOI21_X1)
                                                    0.04       0.71 r
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_THREE_2/U3/ZN (
    ↪ INV_X1)
                                                    0.03       0.74 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_Stage_THREE_2/G_ij (
    ↪ PG_BLOCK_949)
                                                    0.00       0.74 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_stage_FOUR_2_0/G_k1j (
    ↪ PG_BLOCK_947)
                                                    0.00       0.74 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_stage_FOUR_2_0/U2/ZN (
    ↪ AOI21_X1)
                                                    0.05       0.79 r
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_stage_FOUR_2_0/U3/ZN (
    ↪ INV_X1)
                                                    0.03       0.81 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_stage_FOUR_2_0/G_ij (
    ↪ PG_BLOCK_947)
                                                    0.00       0.81 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_stage_FIVE_6/G_ik (
    ↪ G_BLOCK_257)
                                                    0.00       0.81 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_stage_FIVE_6/U2/ZN (
    ↪ AOI21_X1)
                                                    0.05       0.87 r
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_stage_FIVE_6/U1/ZN (
    ↪ INV_X1)
                                                    0.04       0.91 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Block_stage_FIVE_6/G_ij (
    ↪ G_BLOCK_257)
                                                    0.00       0.91 f
```

```
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CLA_SPARSE_TREE/Cout[7] (
    ↪ CARRY_GENERATOR_Nbit32_1)
                                                        0.00       0.91 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CSA/CARRY_SELECT[7] (
    ↪ sum_generator_Nblocks8_bits_per_block4_1)
                                                        0.00       0.91 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CSA/block_n_8/Ci (carry_select_block_N4_241)
                                                        0.00       0.91 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CSA/block_n_8/SUM_SELECT_MUX/SEL (
    ↪ MUX_2to1_N4_241)
                                                        0.00       0.91 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CSA/block_n_8/SUM_SELECT_MUX/U2/Z (MUX2_X1)
                                                        0.08       0.98 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CSA/block_n_8/SUM_SELECT_MUX/Y[1] (
    ↪ MUX_2to1_N4_241)
                                                        0.00       0.98 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CSA/block_n_8/S[1] (carry_select_block_N4_241)
                                                        0.00       0.98 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/CSA/SUM[29] (
    ↪ sum_generator_Nblocks8_bits_per_block4_1)
                                                        0.00       0.98 f
datapath_instance/ALU_instance/SPARSE_TREE_ADDER/SUM[29] (ADDER_P4_N_BIT32_1)
                                                        0.00       0.98 f
datapath_instance/ALU_instance/LGC_BLOCK/SUM[29] (LOGIC_BLOCK_N32)
                                                        0.00       0.98 f
datapath_instance/ALU_instance/LGC_BLOCK/U22/ZN (NOR2_X1)
                                                        0.04       1.03 r
datapath_instance/ALU_instance/LGC_BLOCK/U13/ZN (AND4_X1)
                                                        0.07       1.09 r
datapath_instance/ALU_instance/LGC_BLOCK/U12/ZN (NAND2_X1)
                                                        0.05       1.14 f
datapath_instance/ALU_instance/LGC_BLOCK/U10/ZN (NAND2_X1)
                                                        0.05       1.19 r
datapath_instance/ALU_instance/LGC_BLOCK/U14/ZN (XNOR2_X1)
                                                        0.06       1.25 r
datapath_instance/ALU_instance/LGC_BLOCK/A_LE_B[0] (LOGIC_BLOCK_N32)
                                                        0.00       1.25 r
datapath_instance/ALU_instance/U36/ZN (NAND2_X1)        0.03       1.28 f
datapath_instance/ALU_instance/U102/Z (MUX2_X1)         0.07       1.35 f
datapath_instance/ALU_instance/U101/ZN (OAI211_X1)      0.04       1.39 r
datapath_instance/ALU_instance/OUT_ALU[0] (ALU_N32)     0.00       1.39 r
datapath_instance/MUX_MULT/IN0[0] (MUX_2to1_N32_5)      0.00       1.39 r
datapath_instance/MUX_MULT/U24/ZN (NAND2_X1)            0.03       1.42 f
datapath_instance/MUX_MULT/U22/ZN (NAND2_X1)            0.03       1.45 r
datapath_instance/MUX_MULT/Y[0] (MUX_2to1_N32_5)        0.00       1.45 r
datapath_instance/MUX_RF_OUT1_fw/IN1[0] (MUX_8to1_N32_0)
                                                        0.00       1.45 r
datapath_instance/MUX_RF_OUT1_fw/U28/ZN (NAND2_X1)      0.03       1.48 f
datapath_instance/MUX_RF_OUT1_fw/U26/ZN (NAND2_X1)      0.04       1.52 r
datapath_instance/MUX_RF_OUT1_fw/Y[0] (MUX_8to1_N32_0)
                                                        0.00       1.52 r
datapath_instance/branch_comp_instance/DATA_IN[0] (branch_comp_N32)
                                                        0.00       1.52 r
datapath_instance/branch_comp_instance/U12/ZN (NOR2_X1)
                                                        0.03       1.55 f
datapath_instance/branch_comp_instance/U11/ZN (XNOR2_X1)
                                                        0.06       1.61 f
datapath_instance/branch_comp_instance/BRANCH_IS_TAKEN (branch_comp_N32)
                                                        0.00       1.61 f
datapath_instance/MUX_actual_addr/SEL (MUX_2to1_N32_9)
                                                        0.00       1.61 f
datapath_instance/MUX_actual_addr/U2/Z (CLKBUF_X3)      0.08       1.69 f
datapath_instance/MUX_actual_addr/U27/Z (MUX2_X1)       0.10       1.79 r
datapath_instance/MUX_actual_addr/Y[29] (MUX_2to1_N32_9)
                                                        0.00       1.79 r
datapath_instance/BPU_instance/actual_addr[29] (
    ↪ BPU_TAG_FIELD_SIZE8_SET_FIELD_SIZE3_LINES_PER_SET4)
                                                        0.00       1.79 r
datapath_instance/BPU_instance/U323/ZN (NOR2_X1)        0.03       1.82 f
datapath_instance/BPU_instance/U321/ZN (NOR2_X1)        0.04       1.85 r
datapath_instance/BPU_instance/U667/ZN (OAI221_X1)      0.05       1.90 f
datapath_instance/BPU_instance/U375/ZN (NOR2_X1)        0.06       1.96 r
```

```
datapath_instance/BPU_instance/U372/ZN (NAND4_X1)        0.05       2.01 f
datapath_instance/BPU_instance/U370/ZN (OAI21_X1)        0.06       2.07 r
datapath_instance/BPU_instance/U58/ZN (INV_X1)           0.03       2.11 f
datapath_instance/BPU_instance/U198/ZN (AND2_X1)         0.05       2.16 f
datapath_instance/BPU_instance/U195/ZN (AOI21_X1)        0.05       2.21 r
datapath_instance/BPU_instance/U193/Z (BUF_X1)           0.05       2.27 r
datapath_instance/BPU_instance/U788/ZN (NOR2_X1)         0.03       2.30 f
datapath_instance/BPU_instance/clk_gate_cache_reg[0][2][DATA]/EN (
     ↪ SNPS_CLOCK_GATE_HIGH_BPU_TAG_FIELD_SIZE8_SET_FIELD_SIZE3_LINES_PER_SET4_92)
                                                         0.00       2.30 f
datapath_instance/BPU_instance/clk_gate_cache_reg[0][2][DATA]/test_or/ZN (OR2_X1)
                                                         0.05       2.35 f
datapath_instance/BPU_instance/clk_gate_cache_reg[0][2][DATA]/latch/D (DLL_X1)
                                                         0.01       2.36 f
data arrival time                                                   2.36

clock CLOCK (fall edge)                                  1.22       1.22
clock network delay (ideal)                              0.00       1.22
datapath_instance/BPU_instance/clk_gate_cache_reg[0][2][DATA]/latch/GN (DLL_X1)
                                                         0.00       1.22 f
time borrowed from endpoint                              1.15       2.36
data required time                                                  2.36
--------------------------------------------------------------------------
data required time                                                  2.36
data arrival time                                                  -2.36
--------------------------------------------------------------------------
slack (MET)                                                         0.00

Time Borrowing Information
--------------------------------------------------------------
CLOCK pulse width                                        1.22
library setup time                                      -0.05
--------------------------------------------------------------
max time borrow                                          1.17
actual time borrow                                       1.15
--------------------------------------------------------------
```

Listing E.2: area_DLX.rpt

```
****************************************
Report : area
Design : DLX_BPU_TAG_FIELD_SIZE8_BPU_SET_FIELD_SIZE3_BPU_LINES_PER_SET4
Version: F-2011.09-SP3
****************************************

Library(s) Used:

    NangateOpenCellLibrary (File: /home/mariagrazia.graziano/do/libnangate/
         ↪ NangateOpenCellLibrary_typical_ecsm.db)

Number of ports:                          164
Number of nets:                           397
Number of cells:                          201
Number of combinational cells:             86
Number of sequential cells:               108
Number of macros:                           0
Number of buf/inv:                          7
Number of references:                      16

Combinational area:       12648.034103
Noncombinational area:    16352.350523
Net Interconnect area:      undefined  (Wire load has zero net area)

Total cell area:          29000.384625
Total area:                 undefined
1
```

Listing E.3: power_DLX.rpt

```
****************************************
Report : power
        -analysis_effort low
Design : DLX_BPU_TAG_FIELD_SIZE8_BPU_SET_FIELD_SIZE3_BPU_LINES_PER_SET4
Version: F-2011.09-SP3
****************************************


Library(s) Used:

    NangateOpenCellLibrary (File: /home/mariagrazia.graziano/do/libnangate/
        ↪ NangateOpenCellLibrary_typical_ecsm.db)


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

Design        Wire Load Model            Library
------------------------------------------------
DLX_BPU_TAG_FIELD_SIZE8_BPU_SET_FIELD_SIZE3_BPU_LINES_PER_SET4
                       5K_hvratio_1_4    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  =   2.4157 mW   (73%)
  Net Switching Power  = 878.8506 uW   (27%)
                         ---------
Total Dynamic Power    =   3.2945 mW  (100%)

Cell Leakage Power     = 492.7463 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power    (   %    )
     ↪ Attrs
--------------------------------------------------------------------------------------------------
     ↪
io_pad             0.0000            0.0000            0.0000            0.0000  (   0.00%)
memory             0.0000            0.0000            0.0000            0.0000  (   0.00%)
black_box          0.0000            0.0000            0.0000            0.0000  (   0.00%)
clock_network    350.0800          350.0820        9.6281e+03          709.7900  (  18.74%)
register       1.8263e+03           61.8468        2.3770e+05        2.1259e+03  (  56.13%)
sequential         5.3910            0.3047        1.4431e+03            7.1388  (   0.19%)
combinational    233.8952          466.6210        2.4398e+05          944.4935  (  24.94%)
--------------------------------------------------------------------------------------------------
     ↪
Total          2.4157e+03 uW     878.8546 uW     4.9275e+05 nW     3.7873e+03 uW
1
```