Politecnico di Torino

III Facoltà di Ingegneria

Design of a RISC-V-lite processor

Master degree in Electriconic Engineering

Integrated Systems Architectures

Authors: Group 34

github repository link :

https://github.com/alessionaclerio22/Integrated-Systems-Architecture-Labs

Simone Di Blasi, Stefano Floridia, Alessio Naclerio

Contents

CHAPTER 1

RISC-V-lite Design

The main goal of this laboratory is to develop a VHDL model of a simplified version of the RISC-V processor. It must be compliant with the RV32I architecture, 32-bit fixed-width instruction set, but a reduced set of of instructions has to be implemented. The supported instruction are the following:

- arithmetic: *add*, *addi*, *lui*, *auipc*;
- branches: *beq*;
- loads: *lw*;
- shifts: *srai*;
- logical:*andi*, *xor*;
- compare: *slt*;
- jump and link: *jal*;
- stores: *sw*;

1.1 VHDL Model

The developed model is mainly composed of five pipeline stages and a control unit. Moreover, additional features have been introduced, such as a forwarding unit, an hazard detection unit and branch target buffer and branch history table, implementing a two-bit predictor strategy. A description of the various components is present in the following sections.

1.1.1 Constant and Type Package

A package called "my_package" has been created specifically for this design. It contains all the needed constants, such as the data parallelism, along with the instruction type declaration. In order to use its content, it is included in each file of the design.

1.1.2 Basic Components

Some basic components have been exploited in the design:

- **MUX21, MUX31, MUX41**: they are three behavioural multiplexers. They have a different number of inputs, but the size of each input and output is the same and it is equal to 32 bits. Their external interface is depicted in Figure 1.1, 1.2 and 1.3;

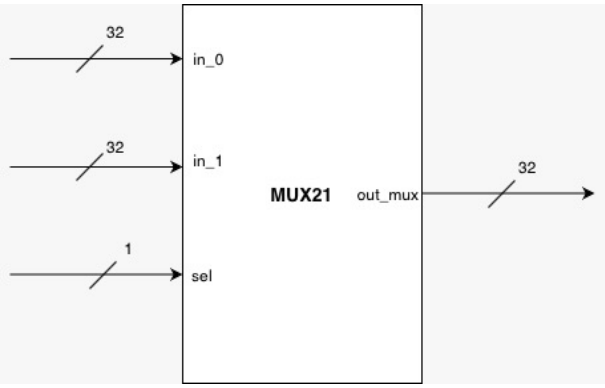

Figure 1.1: MUX21

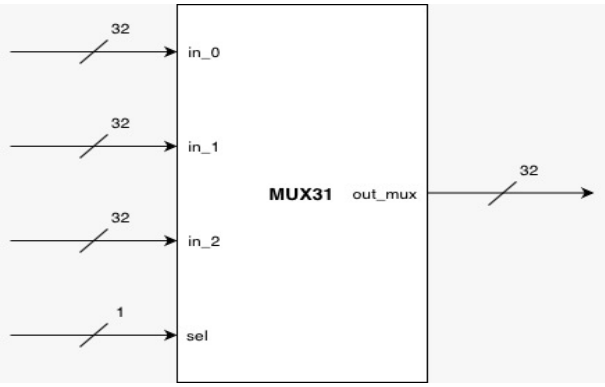

Figure 1.2: MUX31

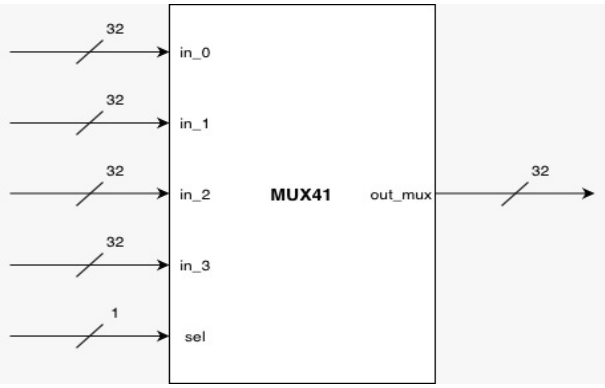

Figure 1.3: MUX41

- **Register**: it is an asynchronous reset behavioural register with an active low enable signal. Since it is used to store PC-related values, it is initialized with "0x00400000", in order to be compliant with the test code provided. Figure 1.4 shows its external interface.

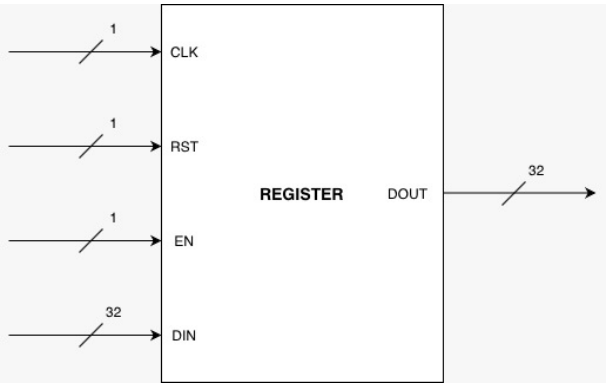

Figure 1.4: REGISTER

1.1.3 Fetch Stage

The fetch stage is the one in charge of managing the right selection of the program counter and providing it to the *IRAM*. It is mainly composed by the following components:

- **PC_Selector_Control_Unit**: based on the control signals it receives as inputs, it provides the proper selection signal to the 4-to-1 multiplexer in *Fetch_Stage* whosw output consists in the right PC to be given to the *IRAM*. The interface of this component is shown in Figure 1.5. It contains:
  - *IS_BRANCH_TAKEN*: signal coming from the branch target buffer, indicating that the branch has been predicted as taken;
  - *IS_BRANCH_TAKEN_D*: it corresponds to the signal *IS_BRANCH_TAKEN* delayed by one clock cycle;
  - *RESTORE_PC*: signal coming from *Branch_Control_Unit*, implying that a "taken" prediction has been proved wrong. In this case, the previous NPC has to be selected;
  - *TAKEN_BRANCH_FROM_ID*: it is a signal coming from *Decode_Stage*. If it is at '1', it means that the two operands pf the decoded *beq* instruction are equal;
  - *INSTRUCTION*: instruction currently in decode stage, encoded in the *INST* type;
  - *PC_MUX_SEL*: output signal which is given in input to the *MUX41* in *Fetch_Stage* as selection signal.

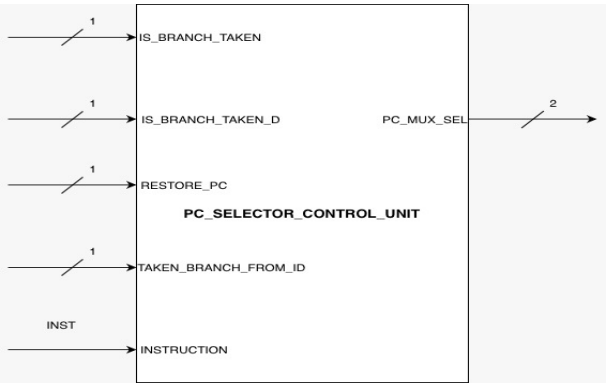

Figure 1.5: PC_SELECTOR_CONTROL_UNIT

- **Fetch_Stage**: it contains two registers, one for the PC and one for the PC to be restored in case of misprediction, a 4-to-1 multiplexer that selects the right PC to deliver to the *IRAM* and a simple adder, used to generate the NPC. Four possible PCs can be chosen by the MUX41:
  - *NEXT_PC*: it is generated adding 4 to the current PC;
  - *PC_FROM_ID_BRANCH*: it is the result of the addition between the immediate and the PC in the ID stage. It is selected when a *jal* instruction is present or in the situation of a not-taken prediction discovered to be wrong;
  - *PC_FROM_BTB*: it is the PC coming from the branch target buffer;
  - *RESTORED_PC*: it is the NPC (PC + 4) delayed of one clock cycle. It is selected when a taken prediction has been proved to be wrong.

  Figure 1.6 shows the external interface of *Fetch_Stage*. It consists of:
  - *CLK*: clock signal;
  - *RST*: asynchronous active high reset signal;
  - *STALL*: input signal meaning that a stall is needed;
  - *INSTRUCTION_ID*: input signal indicating the current instruction in *Decode_Stage*;
  - *SEL_MUX*: input selection signal for the internal *MUX41*;
  - *PC_TO_IMEM*: output PC given to the *IRAM*;
  - *PC_FROM_ID_BRANCH*: refer to the description above;
  - *PC_FROM_BTB*: refer to the description above;

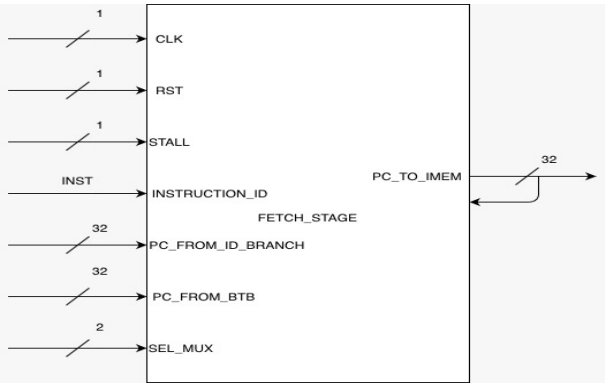

Figure 1.6: FETCH_STAGE

1.1.4 Decode Stage

The decode stage must understand which instruction has to be executed starting from the information retrieved from the *IRAM*. It must also generate the operands for the execute stage and perform operations related to branch and jump instructions. This stage contains the following components:

- **Register_File**: it is a register file that is asynchronous in reading and synchronous in writing. It also supports the special case in which the same location is read and written at the same time. It overcomes this issue by forwarding the data to be written to the right output port. As regards the first location of the register file, R0, it is tied to zero and it can only be read. Figure 1.7 shows its interface, which consists of:

- *CLK*: clock signal;
- *RST*: asynchronous active high reset signal;
- *WR*: write enable signal;
- *ADD_WR*: address for the write operation;
- *ADD_RD1*: first address for the read operation;
- *ADD_RD2*: second address for the read operation;
- *DATAIN*: input data to be written;
- *OUT1*: first output port providing the first operand in output;
- *OUT2*: second output port providing the second operand in output;

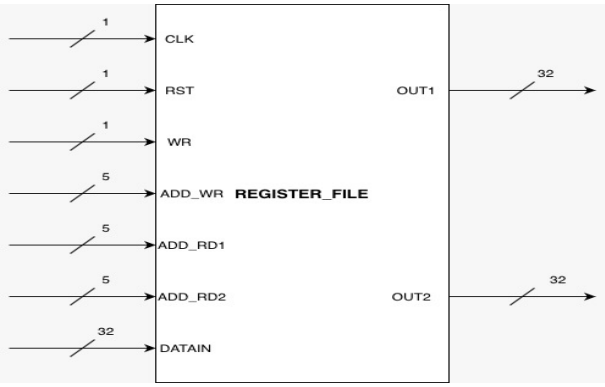

Figure 1.7: REGISTER_FILE

- **Instruction_Encoder**: it receives the opcode and funct field from the *IRAM* and generates a signal whose type is *INST*, the one defined in *"my_package"*. Figure 1.8 shows its interface, which consists of;
  - *OPCODE*: 7-bit-wide opcode field read from the fetched instruction;
  - *FUNCT*: 3-bit-wide funct field read from the fetched instruction;
  - *INSTRUCTION*: output signal of type *INST*, indicating which instruction has been fetched and is currently in decode stage.

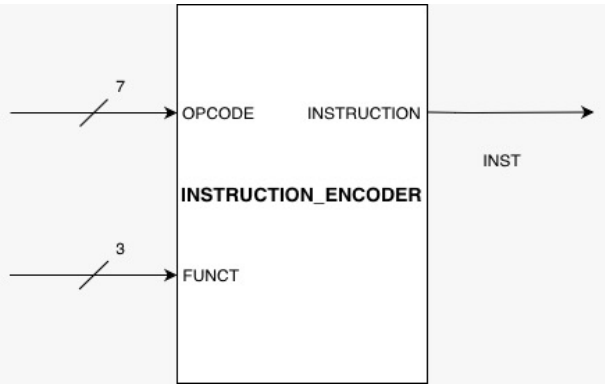

Figure 1.8: INSTRUCTION_ENCODER

- **Immediate_Generation**: based on the currently decoded instruction, it provides in output the proper 32-bit immediate. It will be used as operand in *Execute_Stage* or added to the PC to generate the target address, in case a *beq* or *jal* instruction is present. Figure 1.9 shows its interface, which consists of:
  - *INSTRUCTION*: 32-bit-wide encoding of the instruction that has been fetched and is currently in decode stage;
  - *INST_T*: *INST* type signal representing the instruction currently in decode stage;
  - *IMMEDIATE*: 32-bit output signal consisting in the decoded immediate.

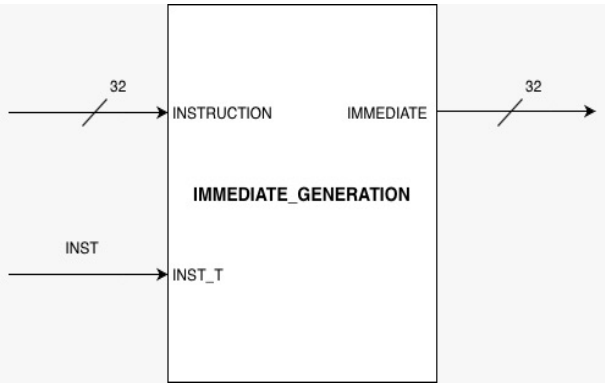

Figure 1.9: IMMEDIATE_GENERATION

- **Branch_Forwarding_Unit**: since the handling of branch operations is performed in *Decode_Stage*, a forwarding mechanism must be introduced in order to always select the right operands. Hence, when a *beq* is decoded, its source registers are checked and forwarding is required when they are equal to the destination register of the instruction in the memory stage. In case a *beq*, *sw* or a *nop* (*addi x0, x0, 0*) is in the memory stage, bypassing must not be done, since they either do not have a destination register or must be ignored. Figure 1.10 shows this component interface, which consists of:
  - *RST*: asynchronous active high reset signal;
  - *EX_MEM_RD*: destination register of the instruction in memory stage;
  - *IF_ID_RS1*: first source register of the instruction in decode stage;
  - *IF_ID_RS2*: second source register of the instruction in decode stage;
  - *IF_ID_OPC*: *INST* type input signal which refers to the instruction currently in the decode stage;
  - *EX_MEM_OPC*: *INST* type input signal which refers to the instruction currently in the memory stage;
  - *FW_EX_MEM_RS1*: output signal that, when it is '1', indicates the need of forwarding towards RS1;
  - *FW_EX_MEM_RS2*: output signal that, when it is '1', indicates the need of forwarding towards RS2;

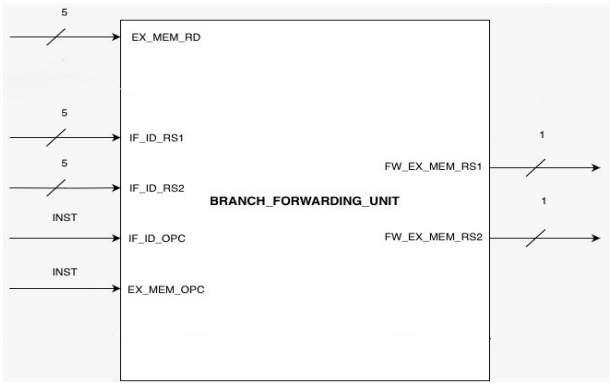

Figure 1.10: BRANCH_FORWARDING_UNIT

- **Decode_Stage**: it contains the *Register_File* and *Immediate_Generation* instances, but also the description of other useful behavioural components. An adder is used in order to calculate the sum of PC and IMM and to generate the target address for a branch instruction. In order to take into account a possible forwarding, a selection is made between the RF ouputs and the values coming from the memory stage, according to *Branch_Forwarding_Unit* outputs. Moreover, a comparison is carried out on the two operands if a *beq* is decoded, resulting in a 1-bit signal indicating if the branch is taken or not.
  - *CLK*: clock signal;
  - *RST*: asynchronous active high reset signal;
  - *INSTRUCTION*: 32-bit encoding of the instruction currently in the decode stage;
  - *INSTRUCTION_WB*: 32-bit encoding of the instruction currently in the write-back stage. It is useful in order to know the right destination register to be written;
  - *INST_T*: *INST* type input signal which refers to the instruction currently in the decode stage;
  - *PC*: program counter of the instruction in decode stage;
  - *RF_WRITE_IN*: input data to be written in *Register_File*;
  - *WR*: input write enable signal for *Register_File*;
  - *STALL*: input signal meaning that a stall is required;
  - *FW_EX_MEM_RS1*: input signal that, when it is '1', indicates the need of forwarding towards RS1;
  - *FW_EX_MEM_RS2*: input signal that, when it is '1', indicates the need of forwarding towards RS2;
  - *OUTALU_EX_MEM*: output of the *ALU* in memory stage;
  - *PC_BRANCH_ID*: output signal identifying the target address of a *beq* instruction;
  - *EFFECTIVE_OUTCOME*: output signal asserted if the two operands of a *beq* are equal;
  - *A*: first operand read from *Register_File*;
  - *B*: second operand read from *Registe_File*;
  - *IMM*: decoded immediate on 32 bits;
  - *INSTRUCTION_OUT*: output signal forwarding *INSTRUCTION* to the execute stage;
  - *PC_OUT*: output signal forwarding *PC* to the execute stage;

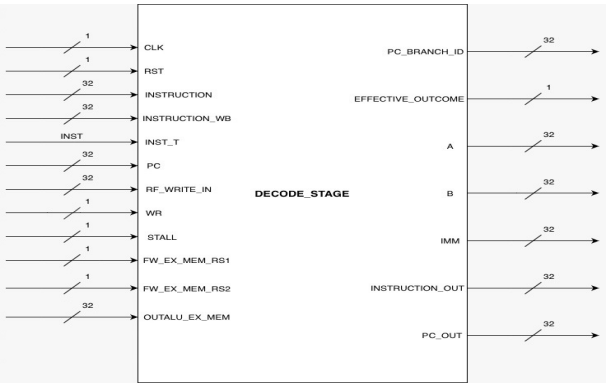

Figure 1.11: DECODE_STAGE

1.1.5 Execute Stage

The execute stage is in charge of performing the arithmetic and logic operations according to the previously decoded instruction. It acts on the operands received by the decode stage or the ones bypassed from the memory or write-back stage, if *Forwarding_Unit* asserts one of its outputs. The following components are present:

- **Execute_Mux_Sel_Encoder**: it generates the selection signals for the *Execute_Stage* multiplexers, according to the ones received from *Control_Unit* and *Forwarding_Unit*. Figure 1.12 shows its interface, which consists of:
  - *FW_EX_MEM_A*: input signal that, if asserted, indicates the need of forwarding the first operand from the memory stage;
  - *FW_MEM_WB_A*: input signal that, if asserted, indicates the need of forwarding the first operand from the write-back stage;
  - *FW_EX_MEM_B*: input signal that, if asserted, indicates the need of forwarding the second operand from the memory stage;
  - *FW_MEM_WB_B*: input signal that, if asserted, indicates the need of forwarding the second operand from the write-back stage;
  - *FW_EX_MEM_C*: input signal that, if asserted, indicates the need of forwarding the data to be written in *DRAM* (in case a *sw* is present in the execute stage) from the memory stage;
  - *FW_MEM_WB_C*: input signal that, if asserted, indicates the need of forwarding the data to be written in *DRAM* (in case a *sw* is present in the execute stage) from the write-back stage;
  - *CU_MUX_SEL_B*: input signal form *Control_Unit*;
  - *MUX_SEL_A*: selection signal for the 3-to-1 multiplexer in *Execute_Stage*;
  - *MUX_SEL_B*: selection signal for the first 4-to-1 multiplexer in *Execute_Stage*;
  - *MUX_SEL_C*: selection signal for the second 4-to-1 multiplexer in *Execute_Stage*.

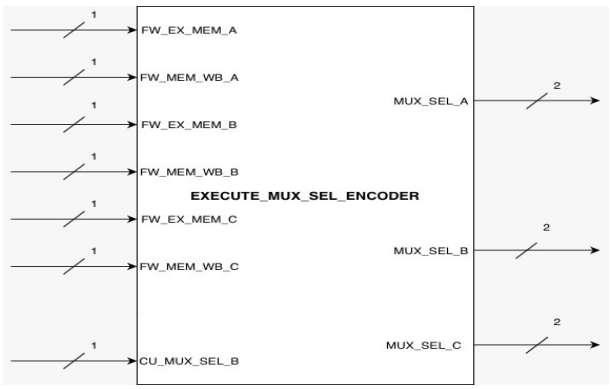

Figure 1.12: EXECUTE_MUX_SEL_ENCODER

- **Execute_Stage**: it contains three multiplexers and the *ALU*. The first multiplexer is a 3-to-1 mux that can choose between the first operand coming from the decode stage (A) and the forwarded data from the memory and write-back stages. The second one is a 4-to-1 mux that can select a value between the second operand coming from the decode stage (B), the immediate and the forwarded data from the memory and write-back stages. While the outputs of these two multiplexers are fed to the *ALU*, the one of the last mux consists in the data to be written in the data memory. It is a 3-to-1 mux which can choose between the second operand coming from the decode stage (B) and the forwarded data from the memory and write-back stages. Figure 1.12 shows its interface, which consists of:
  - *PC*: program counter of the instruction in execute stage;
  - *INSTRUCTION_IN*: 32-bit encoding of the instruction currently in execute stage;
  - *ALU_CONTROL*:
  - *A*: first operand coming from the decode stage;
  - *B*: second operand coming from the decode stage;
  - *IMM*: immediate coming from the decode stage;
  - *FW_OUTALU_EX_MEM*: data forwarded from the memory stage;
  - *FW_OUTALU_MEM_WB*: data forwarded from the write-back stage;
  - *MUX_SEL_A*: input selection signal for the 4-to-1 multiplexer;
  - *MUX_SEL_B*: input selection signal for the first 3-to-1 multiplexer;
  - *MUX_SEL_C*: input selection signal for the second 3-to-1 multiplexer;
  - *OUT_ALU*: output of the *ALU* which could be also used as address for the *DRAM*;
  - *INSTRUCTION_OUT*: output signal forwarding *INSTRUCTION_IN* to the memory stage;
  - *DATAIN_MEM*: output signal representing the data to be written in memory.

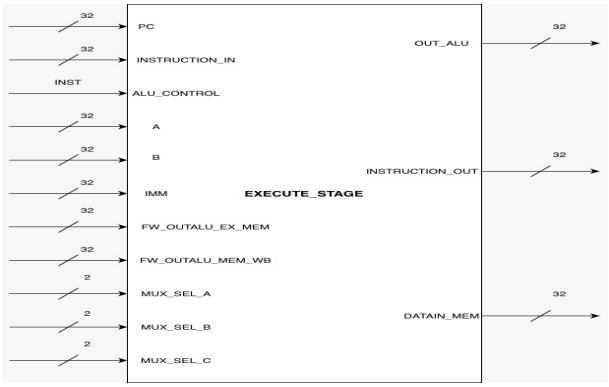

Figure 1.13: EXECUTE_STAGE

- **ALU**: it is the core of *Execute_Stage*. It carries out the needed operations on the received data, according to the signals received from *Control_Unit*. It is able to perform addition, logic operations (xor, and) and arithmetic right shifting, for which a separate component is used, **Shifter**. It is able of shifting the input operand towards right of a given number of positions Figure 1.14 shows *ALU* interface, which consists of:
  - *A*: first operand;
  - *B*: second operand;
  - *PC*: program counter of the instruction currently in execute stage;
  - *ALU_CONTROL*: control signal from the *Control_Unit*;
  - *OUT_ALU*: output of the *ALU*.

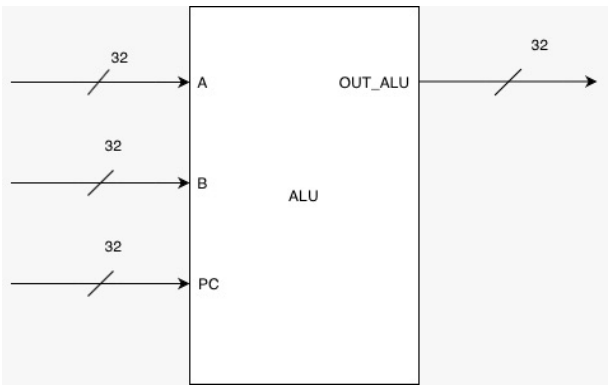

Figure 1.14: ALU

1.1.6 Control Unit

The component *Control_Unit* implements an hardwired control unit. Based on the currently decoded instruction, it generates the proper control word and the *INST* type control signal for the *ALU*. The total amount of control signals needed is 5, so the control word is on 5 bits. Moreover, in order to dispatch the proper signals to the subsequent stages, the control word has to be delayed by means of three registers. From the output of the first one, only the bit needed for the execute stage corresponding to *MUX_SEL_B* is considered, while the others are given in input to the following register. From the output of the second register, two bits from the remaining four are taken, the

ones corresponding to *MEM_EN* and *READNOTWRITE*, needed for the data ram in the memory stage. Finally, the output of the third register consists in the two control signals for the write-back stage, which are *MUX_SEL_WB* and *WR_RF*. The *ALU* control signal is managed separately and it is delayed only by one clock cycle, in order to be properly timed for the execute stage. Figure 1.15 shows the *Control_Unit* entity signals, which are the following:

- *CLK*: clock signal;
- *RST*: asynchronous active high reset signal;
- *STALL*: input signal meaning that a stall is required;
- *INSTRUCTION*: *INST* type instruction currently in the decode stage;
- *MUX_SEL_B*: control signal towards the *MUX41* of the execute stage;
- *ALU_INST*: control signal for the *ALU*;
- *MEM_EN*: control signal used for enabling the *DRAM*;
- *READNOTWRITE*: R/$\overline{W}$ control signal for the *DRAM*;
- *MUX_SEL_WB*: control signal to be given in input to the *MUX21* in the write-back stage as selection signal;
- *WR_RF*: write enable signal for the *Register_File*.

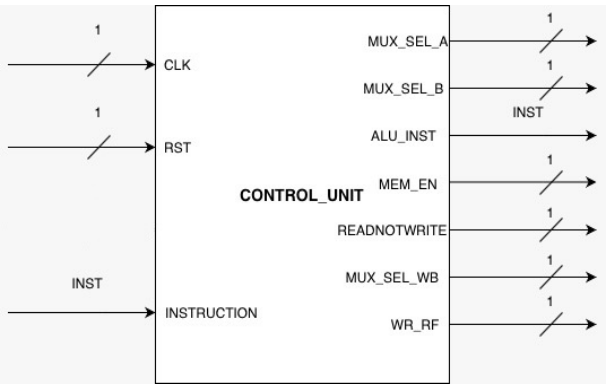

Figure 1.15: CONTROL_UNIT

1.1.7 Forwarding Unit

The Forwarding Unit allows the bypassing of data when one of the source registers in the execute stage is equal to the destination one of the instruction in memory or write-back stage, apart from some specific cases.

As regards the first operand of the execute stage, it is forwarded from the memory stage if the condition below is satisfied and if the instruction in memory is not a *lw*, case in which a stall is required, a *beq* or a *sw*, which are instruction not presenting a destination register. The bypassing from the write-back stage is needed when the first source register in the execute stage is equal to the destination one in write-back and the instruction in write-back is not a *beq* or a *sw*, which do not present a destination register. In this last case, a *lw* can be present in the write-back stage, since its

result is already available.

The bypassing of data towards the second operand can only be possible when the instruction in the execute stage presents a RS2 field in its encoding. These instructions are *add*, *slt*, *xor* and *sw*. Data is forwarded from memory stage towards the second operand of the execute when RS2 of the instruction in execute is equal to RD of the one in memory and the instruction in memory is not a *lw*, case in which a stall is required, *sw* or a *beq*, which do not have a destination register. Moreover, the case in which a *sw* is present in execute and its RS2 is equal to RD of the instruction in memory is managed separately, by activating the signal *FW_EX_MEM_C*. As a matter of fact, in this last case the forwarded data does not represent an operand of the *ALU*, but it consists in the data to be written in the data memory. On the other hand, data is forwarded from write-back stage towards the second operand of the execute stage when RS2 of the instruction in execute is equal to RD of the one in write-back and the instruction in write-back is not a *sw* or a *beq*, which do not have a destination register. Moreover, as explained before, the case in which a *sw* is present in execute stage and its RS2 is equal to RD of the instruction in write-back is managed separately, by activating the signal *FW_MEM_WB_C*. Figure 1.16 shows the *Forwarding_Unit* entity signals, which are the following:

- *RST*: asynchronous active high reset signal;
- *EX_MEM_RD*: destination register of the instruction in memory stage;
- *MEM_WB_RD*: destination register of the instruction in write-back stage;
- *ID_EX_RS1*: first source register of the instruction in execute stage;
- *ID_EX_RS2*: second source register of the instruction in execute stage;
- *ID_EX_OPC*: input *INST* type signal indicating the instruction currently in execute stage;
- *EX_MEM_OPC*: input *INST* type signal indicating the instruction currently in memory stage;
- *MEM_WB_OPC*: input *INST* type signal indicating the instruction currently in write-back stage;
- *FW_EX_MEM_A*: output signal that, if asserted, indicates the need of forwarding data from memory stage towards the first operand of the execute stage;
- *FW_MEM_WB_A*: output signal that, if asserted, indicates the need of forwarding data from write-back stage towards the first operand of the execute stage;
- *FW_EX_MEM_B*: output signal that, if asserted, indicates the need of forwarding data from memory stage towards the second operand of the execute stage;
- *FW_MEM_WB_B*: output signal that, if asserted, indicates the need of forwarding data from write-back stage towards the second operand of the execute stage;
- *FW_EX_MEM_C*: output signal that, if asserted, indicates the need of forwarding a value from memory stage towards the data to be written in memory (in case a *sw* is present in execute stage);
- *FW_MEM_WB_C*: output signal that, if asserted, indicates the need of forwarding a value from write-back stage towards the data to be written in memory (in case a *sw* is present in execute stage);

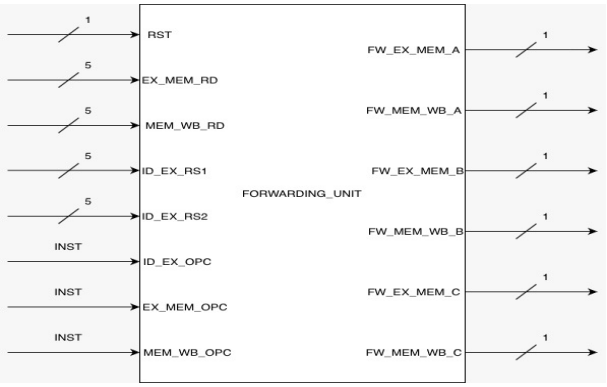

Figure 1.16: FORWARDING_UNIT

1.1.8 Hazard Unit

The Hazard Unit allows the detection of possible hazards and prevents them inserting a stall. The situations covered by *Hazard_Unit* are the following:

- the instruction in decode stage must not be a *lui*, *auipc* or a *jal*, since they do not have a destination register. Once this condition is verified, the possible considered situations are:
  - the case in which a *lw* is in execute stage and an instruction different from *beq* is in decode stage. The *STALL* signal is asserted when the source register of the instruction in decode is equal to the destination register of the *lw*. While all instructions have a RS1, some others do not present a RS2 and must avoided when this field is check;
  - the case in which a *beq* is in decode stage. If one of its source register is equal to the destination one of the instruction in execute (which must not be a *nop*), *STALL* is asserted;
  - the case in which a *lw* is in memory stage and a *beq* is is decode. In this case, if one of the source registers of the *beq* is equal to the destination register of the *lw*, *STALL* is asserted.

Figure 1.17 shows the *Hazard_Unit* entity signals, which are the following:

- *RST*: asynchronous active high reset signal;
- *ID_EX_RD*: destination register of the instruction in execute stage;
- *EX_MEM_RD*: destination register of the instruction in memory stage;
- *IF_ID_RS1*: first source register of the instruction in decode stage;
- *IF_ID_RS2*: second source register of the instruction in decode stage;
- *ID_EX_OPC*: input *INST* signal indicating the instruction in execute stage;
- *IF_ID_OPC*: input *INST* signal indicating the instruction in decode stage;
- *EX_MEM_OPC*: input *INST* signal indicating the instruction in memory stage;
- *STALL*: output signal meaning that a stall is required.

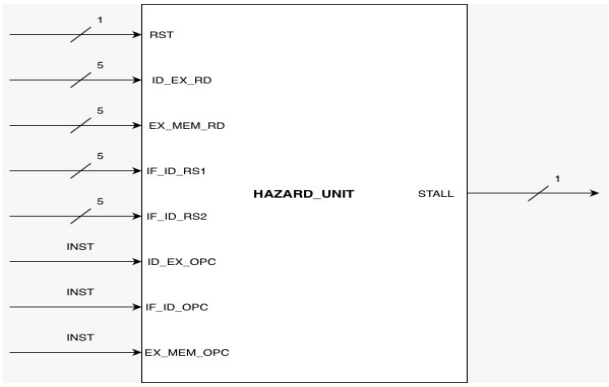

Figure 1.17: HAZARD_UNIT

1.1.9 Branch Management

The management of branch instructions is performed mainly by two components:

- *Branch_Target_Buffer*: it is the unit containing a Branch Target Buffer structure and implementing a two-bit predictor strategy. A Branch History Table, implemented as a 32x2 memory, contains the information needed in order to carry out the prediction according to the adopted strategy. The two-bit predictor allows four possible states to be assumed by each memory location. The evolution among these states is shown in figure 1.18, where "T" stands for a strong taken prediction, "W_T" for a weak taken one, "UT" for a strong not-taken one and "W_UT" for a weak not-taken one. When the effective outcome of the branch instruction is known, a redefinition of the associated prediction is performed according to the FSM showed in Figure 1.18. The Branch Target Buffer has been implemented as a 32x32 memory, where each location contains the target program counter of the related branch instruction. In order to address both the Branch Target Buffer and the Branch History Table, the program counter of the fetched instruction is used, especially the bits from 6 to 2. This would cause a quite high aliasing probability but it has been chosen since it is sufficient in order to carry out the given test program. Once the address is received, if the prediction made by the Branch History Table corresponds to a taken one, the Branch Target Buffer provides the target program counter of the branch instruction as output. Moreover, if it is the first time a branch is decoded, the target program counter is stored in the pointed location.

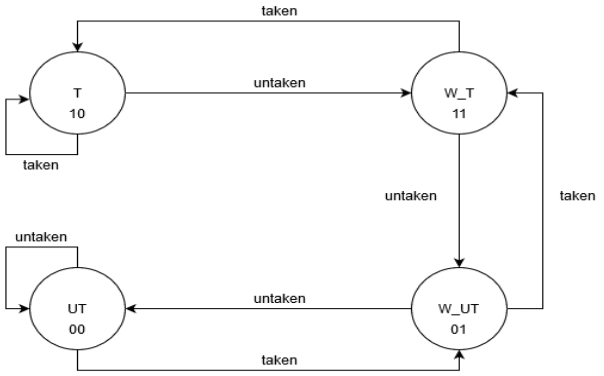

Figure 1.18: Two-bit predictor strategy FSM

The interface of this component is shown in figure 1.19 with the following signals:

- *CLK*: clock signal;
- *RST*: asynchronous active high reset signal;
- *PRED_T_NOT_T*: signal coming from *Branch_Control_Unit* meaning that a branch instruction was predicted taken but it resulted in a non-taken one;
- *PRED_T_T*: signal coming from *Branch_Control_Unit* meaning that a branch instruction was predicted taken correctly;
- *PRED_NOT_T_T*: signal coming from *Branch_Control_Unit* meaning that a branch instruction was predicted not taken but it resulted in a taken one;
- *PRED_NOT_T_NOT_T*: signal coming from *Branch_Control_Unit* meaning that a branch instruction was predicted not taken correctly;
- *UPDATE_EN*: input signal indicating the need of updating the branch prediction;
- *BTB_ADDR*: address on 5 bits to be given to the Branch Target Buffer structure;
- *UPDATE_TARGET_PC*: target program counter to be stored in the Branch Target Buffer location the first time a branch is decoded;
- *UPDATE_ADDRESS*: address used both to update the branch prediction and to store the target program counter in the Branch Target Buffer structure the first time a branch is decoded;
- *TARGET_PC*: output target program counter;
- *IS_TAKEN*: output signal indicating if a branch is predicted taken, if it is at '1', or untaken, if it is at '0'.

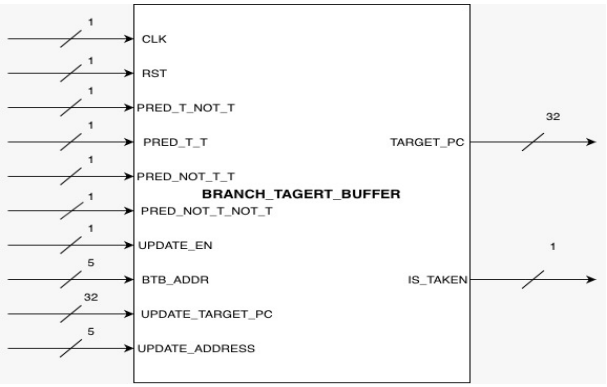

Figure 1.19: BRANCH_TARGET_BUFFER

- *Branch_Control_Unit*: this unit generates the proper control signals in order to redefine a branch instruction prediction, flush the instruction fetched if a misprediction occurred and restoring the right program counter when a branch instruction was predicted as taken but resulted in a not-taken one. It receives as inputs the effective outcome of the branch instruction (available in the decode stage), the branch prediction, the information about the instruction in decode and the information about the possible need of a stall, case in which all its output signals are at '0'. Its interface is shown in figure 1.20 with the following signals:
  - *EFFECTIVE_OUTCOME*: input signal indicating the effective outcome of the branch instruction;
  - *IS_BRANCH_TAKEN*: input signal corresponding to the branch prediction;

- *INSTR*: input *INST* signal referring to the instruction currently in decode stage;
- *STALL*: input signal indicating that a stall is required;
- *UPDATE_EN*: output signal indicating the need of updating the branch prediction;
- *FLUSH*: output signal indicating the need of flushing the fetched instruction due to a misprediction;
- *RESTORE_PC*: output signal indicating that the program counter must be restored. This happens when a branch instruction was predicted taken but resulted in a non-taken one;
- *PRED_T_NOT_T*: signal coming from *Branch_Control_Unit* meaning that a branch instruction was predicted taken but it resulted in a non-taken one;
- *PRED_T_T*: signal coming from *Branch_Control_Unit* meaning that a branch instruction was predicted taken correctly;
- *PRED_NOT_T_T*: signal coming from *Branch_Control_Unit* meaning that a branch instruction was predicted not taken but it resulted in a taken one;
- *PRED_NOT_T_NOT_T*: signal coming from *Branch_Control_Unit* meaning that a branch instruction was predicted not taken correctly;

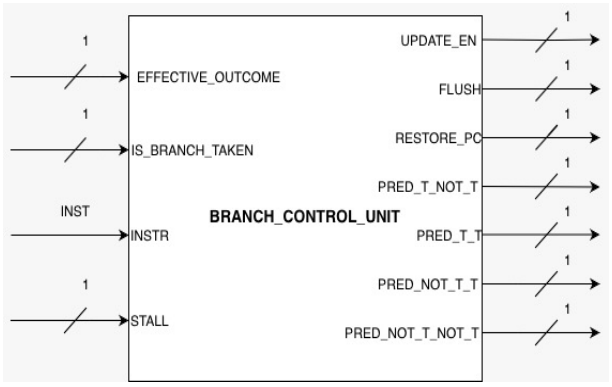

Figure 1.20: BRANCH_CONTROL_UNIT

1.1.10 RISCV

RISC-V is the microprocessor entity containing all the other components previously described. Moreover, it implements the pipeline registers in a process, also managing the cases in which a flush or a stall is required. The memory and write-back stages have not been implemented separately, since they respectively need an external data ram included in the testbench and a simple 2-to-1 multiplexer which can select between the output of the *ALU* and the data coming from the dram. The interface of *RISC-V* is shown in Figure 1.21 with the following signals:

- *CLK*: clock signal;
- *RST*: asynchronous active high reset signal;
- *OUTALU_IN_MEM*: output signal given to the *DRAM* representing the address where *DATAMEM_IN_MEM* should be written;
- *DATAMEM_IN_MEM*: output signal representing the data to be written in the *DRAM*;
- *MEMD_OUT_MEM*: input signal corresponding to the data read from the *DRAM*;

- *MEM_EN*: *DRAM* enable signal;
- *READNOTWRITE*: R/$\overline{W}$ signal to *DRAM*;
- *PC_TO_IMEM*: program counter to be delivered to the *IRAM*;
- *INSTRUCTION_OUT_IF*: 32-bit encoding of the fetched instruction from the *IRAM*.

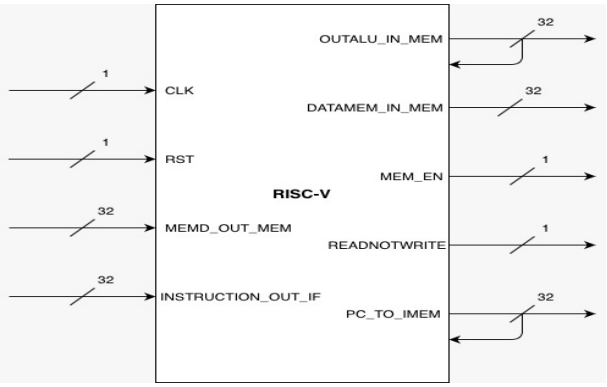

Figure 1.21: RISCV

1.1.11 Testbench

The testbench allows the connection of the *RISC-V* entity to an instruction memory and a data memory. These last two components are implemented respectively by:

- IRAM: it is a asynchronous 32x32 instruction memory, which is addressed by the program counter received by *RISC-V*. This PC has to be normalized by subtracting 0x00400000 to it in order to properly address the internal memory. The interface of this component is shown in Figure 1.22 with the following signals:
  - *RST*: asynchronous active high reset signal;
  - *PC_IN*: input program counter;
  - *INSTRUCTION*: output fetched instruction encoded on 32 bits.

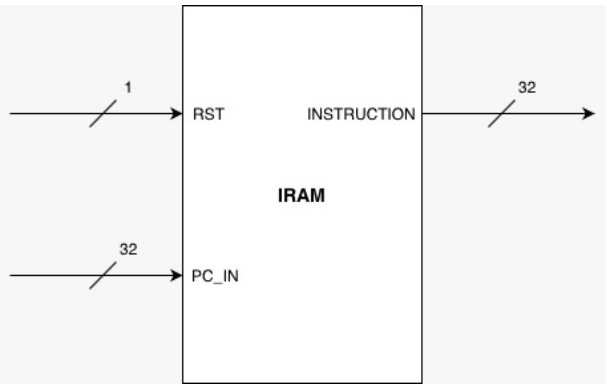

Figure 1.22: IRAM

- DRAM: it is a synchronous-writing, asynchronous-reading 8x32 memory. It is address by the *RISC-V* through the signal *OUTALU_IN_MEM* and it receives *DATAMEM_IN_MEM* as data to be written. If its enable signal, *ENABLE*, is at '1' and *READNOTWRITE* is at '1', *DATA_OUT* is provided in output and it corresponds to the read value. If *ENALBE* is '1' and *READNOTWRITE* si '0', *DATA_IN* is written into the pointed location. The interface of this component is shown in Figure 1.22 with the following signals:
  - *DATA_IN*: input data to be written in *DRAM*;
  - *ADDRESS*: address used both for both read and write operations;
  - *READNOTWRITE*: R/$\overline{W}$ signal;
  - *CLK*: clock signal;
  - *RST*: asynchronous active high reset signal;
  - *ENABLE*: enable signal;
  - *DATA_OUT*: read data output;

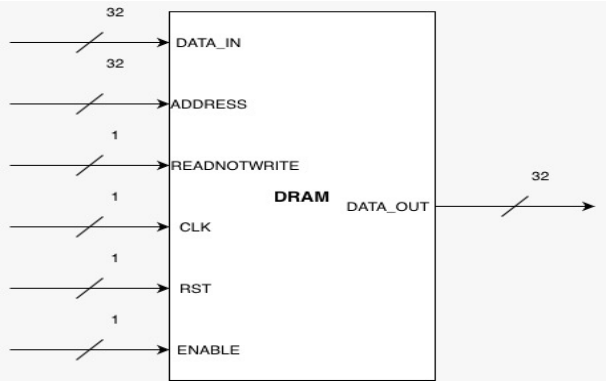

Figure 1.23: DRAM

- tb_RISC_V: it is the testbench file connecting *RISC-V* to *IRAM* and *DRAM*, allowing the design simulation. It instantiates a 1ns clock and a reset signal which goes low after 6ns activating the *RISC-V*. It also normalizes the address to be given to *DRAM* by subtracting 0x40040000 to it.

1.2 Design Simulation

In order to simulate the design, the given test code, called *minv-rv.s*, has been exploited. The goal of this program is to calculate the minimum, in absolute value, among a set of operands stored in the data memory, which are: 10, -47, 22, -3, 15, 27, -4. At the end, the selected number is stored in the next free memory location.

Thanks to the presence of a sequence of instructions in which both bypassing and stalls are requested, the correct behaviour of *Hazard_Unit* and *Forwarding_Unit* has been checked. Moreover, two *beq* instructions are available in order to properly test *Branch_Target_Buffer*, *Branch_Control_Unit* and *Branch_Forwarding_Unit*.

1.2.1 Test Program Execution

The program is mainly divided in three sections. The first one, labeled "*__start*", consists of a set of *addi*, *lui* and *auipc* instructions initializing the content of some registers:

- *x16*: it is the register whose value identifies the current iteration of the loop. It is loaded with the total amount of values stored in memory. Once it has reached zero, it means that the result is ready;
- *x4*: it contains the pointer to the memory location whose value is currently processed;
- *x5*: it is loaded with the pointer of the memory location in which the final result will be stored;
- *x13*: it is initialized with the maximum positive value on 32 bits;

The second section, the one with the label "*loop*", performs the right operations in order to compute the minimum absolute value. It mainly consists in a loop in which:

- the value pointed by $x4$ is loaded from memory;
- a set of instructions transform this number into its unsigned equivalent and store it in $x10$;
- $x10$ is compared to the previous intermediate result ($x13$) and, if lower, replaced to it;
- the indexes are properly updated.

The instructions used for this part of code are:

```
beq x16,x0,done
lw x8,0(x4)
srai x9,x8,31
xor x10,x8,x9
andi x9,x9,0x1
add x10,x10,x9
addi x4,x4,0x4
addi x16,x16,-1
slt x11,x10,x13
beq x11,x0,loop
add x13,x10,x0
jal loop
```

Since *x16* is initialized with 7, seven total iterations are required to carry out the whole test program. While the execution of non-branch operations remains essentially the same through the various iterations, *beq* instructions behaviour may vary depending on the prediction and their effective outcome. As a matter of fact, an overview on branch instructions execution is reported in the the following Tables. As it can be noticed, the misprediction rate is significantly higher for the second *beq* than for the first one.

Loop Iteration	BHT	Prediction	Effective Outcome	Misprediction
0	00	NT	NT	No
1	00	NT	NT	No
2	00	NT	NT	No
3	00	NT	NT	No
4	00	NT	NT	No
5	00	NT	NT	No
6	00	NT	T	Yes

beq x16,x0,done behaviour

Loop Iteration	BHT	Prediction	Effective Outcome	Misprediction
0	00	NT	NT	No
1	00	NT	T	Yes
2	01	NT	T	Yes
3	11	T	NT	Yes
4	01	NT	T	Yes
5	11	T	T	No
6	10	T	T	No

beq x11,x0,loop behaviour

Moreover, at each iteration, an hazard would occur due to the fact that the destination register of the *lw* instruction is the same as the source register of the subsequent *srai*. This is detected by the *Hazard_Unit* and a stall of one clock cycle is inserted, in order to let the *srai* access to the right data thanks to bypassing from the write-back to the execute stage. Another stall is required by the *beq* instruction after the *slt*, in order to let the value of *x11* be computed and then forwarded to the *beq* thanks to the *Branch_Forwarding_Unit*, as shown in figure 1.25. In figure 1.24, the behaviour of the *Hazard_Unit* is shown in the described cases.

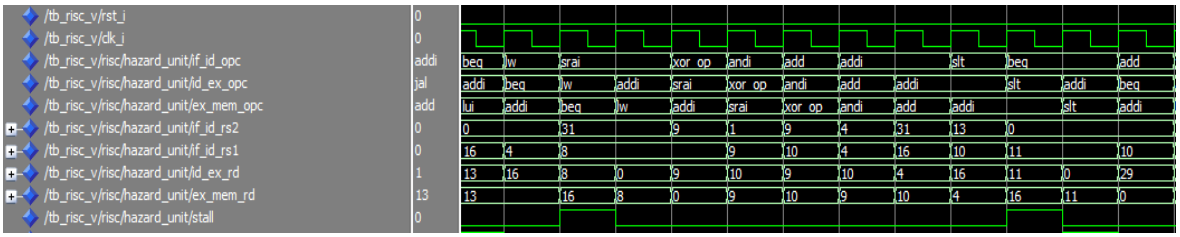

Figure 1.24: *Hazard_Unit* behaviour

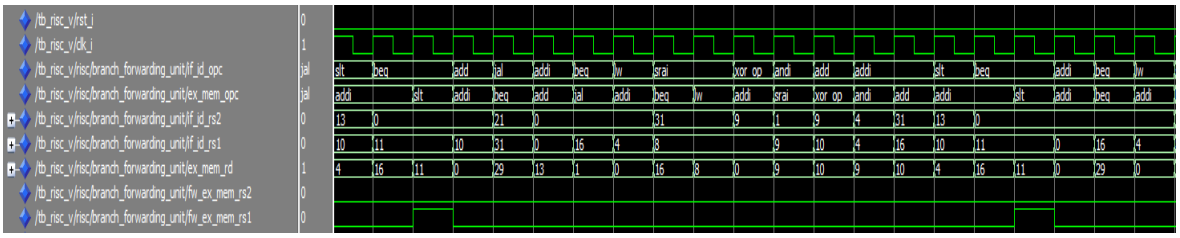

Figure 1.25: *Branch_Forwarding_Unit* behaviour

Bypassing is also needed in other situations. Data is forwarded from the memory to the execute stage in these cases:

- from *srai x9,x8,31* to *xor x10,x8,x9*
- from *andi x9,x9,0x1* to *add x10,x10,x9*

Data is forwarded from the write-back to the execute stage in the following situations (other than in the previously highlighted situation regarding the *lw* instruction):

- from *srai x9,x8,31* to *andi x9,x9,0x1*

- from *xor x10,x8,x9* to *add x10,x10,x9*

The *Forwarding_Unit* behaviour in these situations is reported in 1.26.

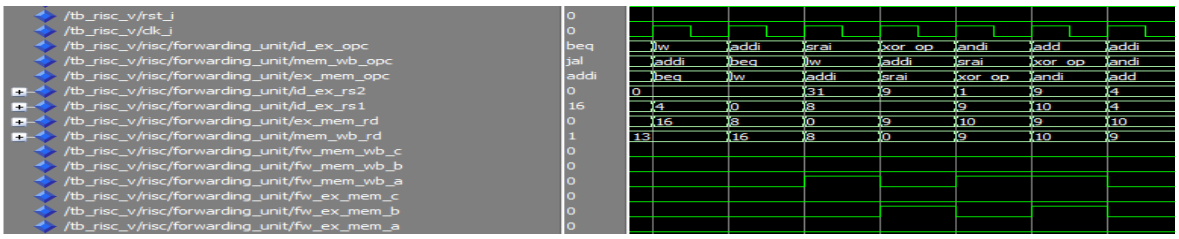

Figure 1.26: *Forwarding_Unit* behaviour

Furthermore, when the *jal* is executed, a nop is inserted after it in order to let the right program counter be fetched.
The last section, labeled "*done*", consists of storing the result in the right memory location and entering an infinite loop (label "*endc*").

1.3 Synthesis and Place&Route

The synthesis of the design has been carried out mainly using two *tcl* scripts, *syn_RISC.tcl* and *post-syn_RISC.tcl*. The first one allows analyzing all the *.vhd* files, elaborating the design, applying the clock and other constraints and compile using *compile_ultra*. The latter has been exploited in order to save the timing and area reports, as well as the netlist and the *.sdc* and the *.sdf* files.

An iterative approach has been applied in order to reach the maximum frequency. The synthesis process has been repeated until a null slack has been retrieved with the "$report_timing$" command. Once reached this condition, the place and route phase could begin. Unfortunately, the worst slack obtained in the post-route *.slk* timing report regarding the Setup was negative. Hence, this value has been added to the one of the previous clock in the *syn_RISC.tcl* file and synthesis has been run again. This process has been iterated until the worst slack in the *.slk* files was positive. Finally, the netlists generated by Design Compiler and Innovus have been simulated in order to check that they behaved like intended by the design. An image of the post-place&route processor is showed in Figure 1.27.

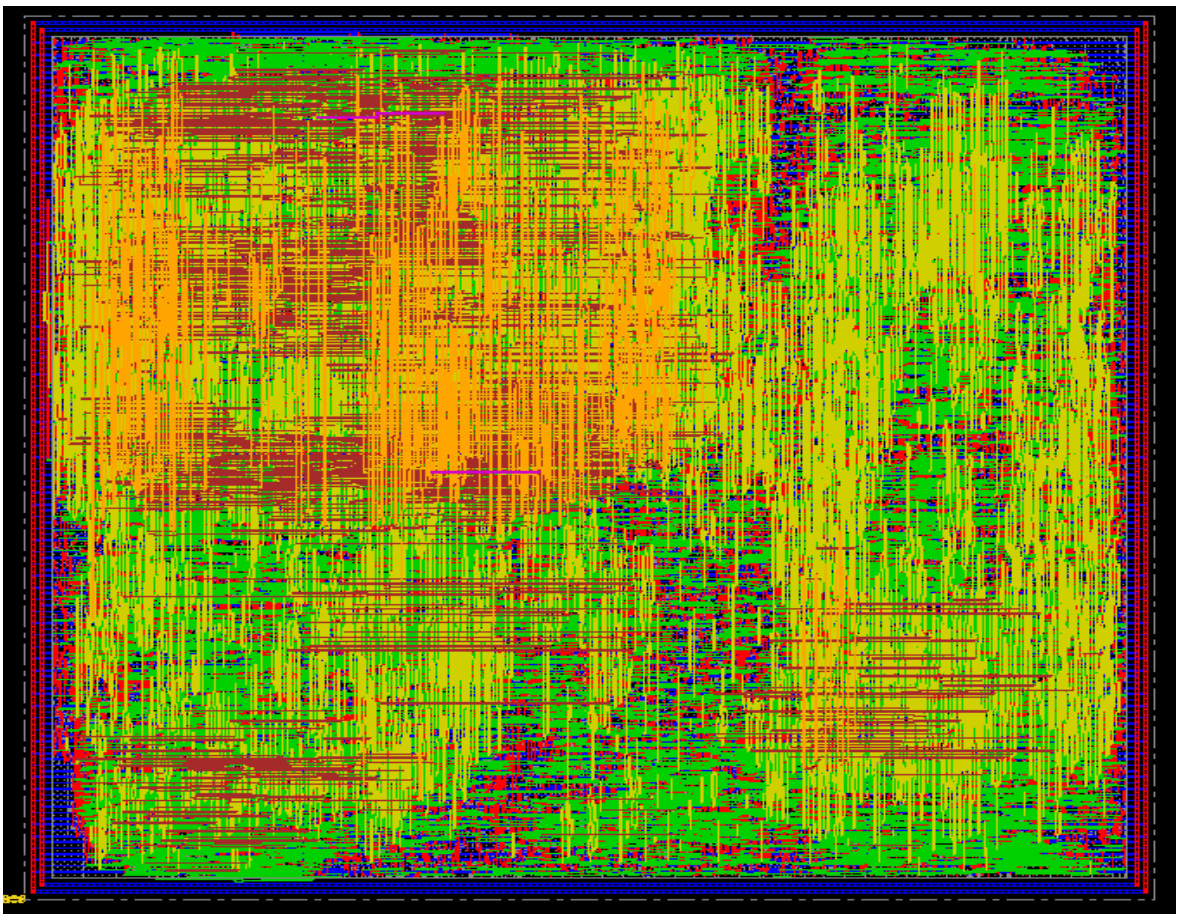

Figure 1.27: post-place&route RISCV-lite processor

1.3.1 Timing and Area

The final maximum frequency is 574.7MHz, with a clock period of 1.74ns. From the output of the *report_timing* command on Design Compiler, saved in *timing_report_RISC_V.txt*, it can be noticed that the critical path is the one involving the branch target buffer, in particular the one from its input address to the output target PC, which is delivered in input to the multiplexer selecting the proper program counter to be given to the PC register. Moreover, both *.slk* files generated by Innovus in the post-route phase have been checked in order to verify that no negative slack is present. From the *.slk* file obtained setting *Analysis Type* to *Setup*, it can be observed that the worst slacks correspond to *INSTRUCTION_IN_ID_reg* and *BRANCH_TARGET_BUFFER_BTB_reg*. On the other hand, from the *.slk* file obtained setting *Analysis Type* to *Hold*, it can be noticed that the worst slacks correspond to and *BRANCH_TARGET_BUFFER_BTB_TWO_BIT_PREDICTOR_reg_reg*, *INSTRUCTION_IN_EX_reg* and *PC_IN_ID_reg*.

As regards the occupied area, the output of the *report_area* command on Design Compiler has been saved in the file *area_report_RISC_V.txt*. The reported total cell area is 23841.31µm^2, similar to the value retrieved in the file *RISC_V.gatecount* generated by Innovus, which is 24074.9µm^2.

CHAPTER 2

Modified RISC-V-lite Design

2.1 VHDL Model

The RISC-V-lite design previously developed has been modified in order to accomodate a new instruction, called *absv*, and a new functional unit aimed at calculating the absolute value. Hence, a proper encoding has been selected for *absv* and a dedicated component called *Absolute_Value* has been included into the *ALU*.

The format chosen for the new instruction is the following:

absv RD, RS1

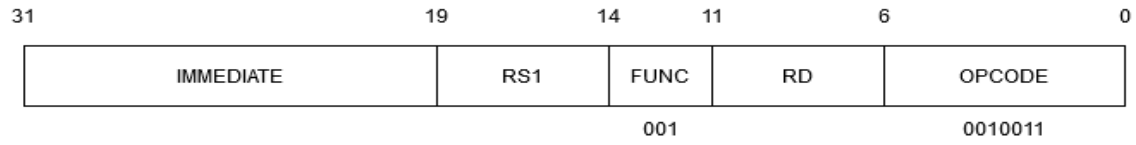

Figure 2.1: ABSV Encoding

Since the *IMMEDIATE* field is not used, it has been decided to set it to all zeros. As regards the functional unit, it simply performs one out of these two possible operations on the input *op1*:

- it forwards *op1* to *output* if it is already positive;
- it complements *op1* and adds '1' to it if it is negative.

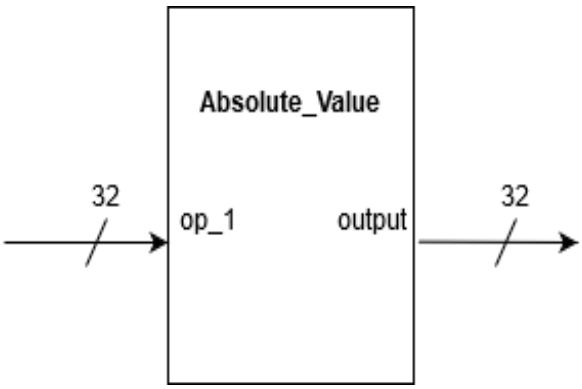

Figure 2.2: Absolute_Value

In order to support the insertion of the new instruction, some other components have been modified accordingly:

- ALU: it includes the new dedicated component and allows its selection according to the input control signals;
- *Control_Unit*: it takes into account the *absv* instruction by generating a new dedicated control word and proper control signals for the ALU;
- *Instruction_Encoder*: it takes *absv* into account generating the right instruction encoding with respect to the instruction type declared in *my_package*.

2.2 Design Simulation

In order to simulate the new design, the same test program previously used has been exploited. However, the newly added instruction allows simplifying the code, eliminating the set of operations needed to compute the absolute value, thus reducing the code size. The instructions that have been removed are the following:

```
srai x9,x8,31
xor x10,x8,x9
andi x9,x9,0x1
add x10,x10,x9
```

They have been replace with:

```
absv x10, x8
```

The new code becomes:

```
beq x16,x0,done
lw x8,0(x4)
absv x10, x8
addi x4,x4,0x4
addi x16,x16,-1
slt x11,x10,x13
beq x11,x0,loop
add x13,x10,x0
jal loop
```

As it can be noticed, a stall is always present after the *lw* because of the fact that the source register of the *absv* is equal to the destination register of the *lw*. As regards branch behaviour, it is equal to the one explained in the previous chapter for the first RISCV-V-lite design. Figure 2.3, 2.4 and 2.5 shows the *Forwarding_Unit*, *Hazard_Unit* and *Branch_Forwarding_Unit* behaviour in the design simulation.

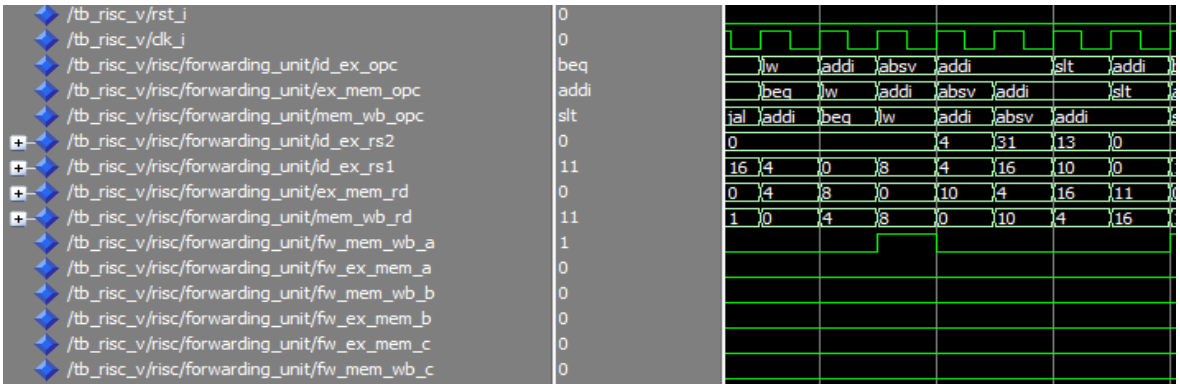

Figure 2.3: Forwarding_Unit behaviour

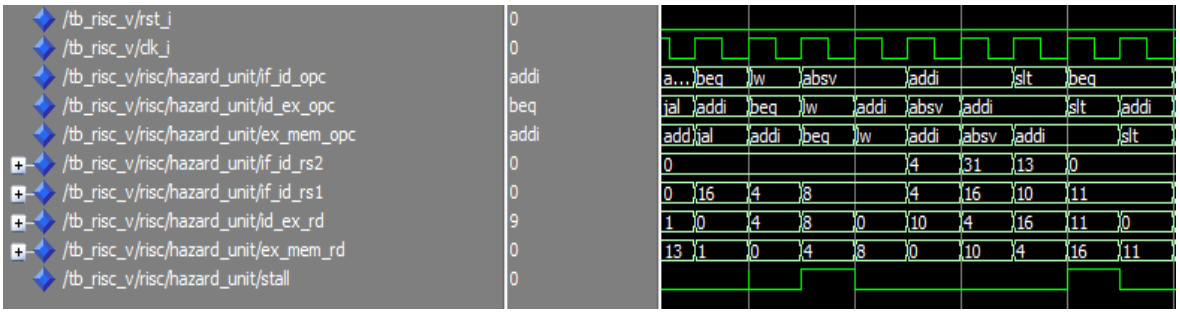

Figure 2.4: Hazard_Unit behaviour

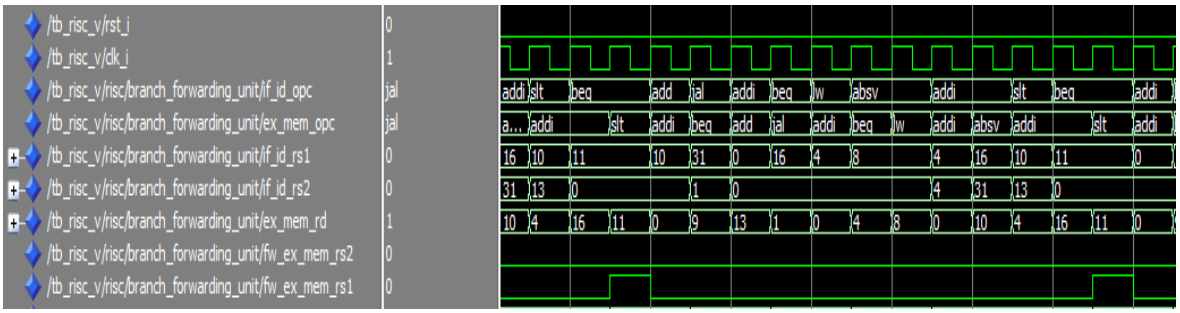

Figure 2.5: *Branch_Forwarding_Unit behaviour*

2.3 Synthesis and Place&Route

As with the previous RISC-V-lite design, the synthesis has been carried out using two *tcl* scripts, *syn_RISC.tcl* and *postsyn_RISC.tcl*. The first one allows analyzing all the *.vhd* files, elaborating the design, applying the clock and other constraints and, finally, compile. The latter has been exploited in order to save the timing and area reports, as well as the netlist and the *.sdc* and the *.sdf* files.

As described before, an iterative approach has been applied in order to reach the maximum frequency. The synthesis process has been repeated until a null slack has been retrieved with the "$report_timing$" command. Once reached this condition, the place and route phase could begin. The worst slack obtained in the post-route *.slk* timing report regarding the Setup was negative, as in the previous design case. Hence, this value has been added to the one of the previous clock in the *syn_RISC.tcl* file and synthesis has been run again. This process has been iterated until the worst slack in the *.slk* files was positive. Finally, the netlists generated by Design Compiler and Innovus have been simulated in order to check that they behaved like intended by the design. An image of the post-place&route processor is showed in Figure 2.6.

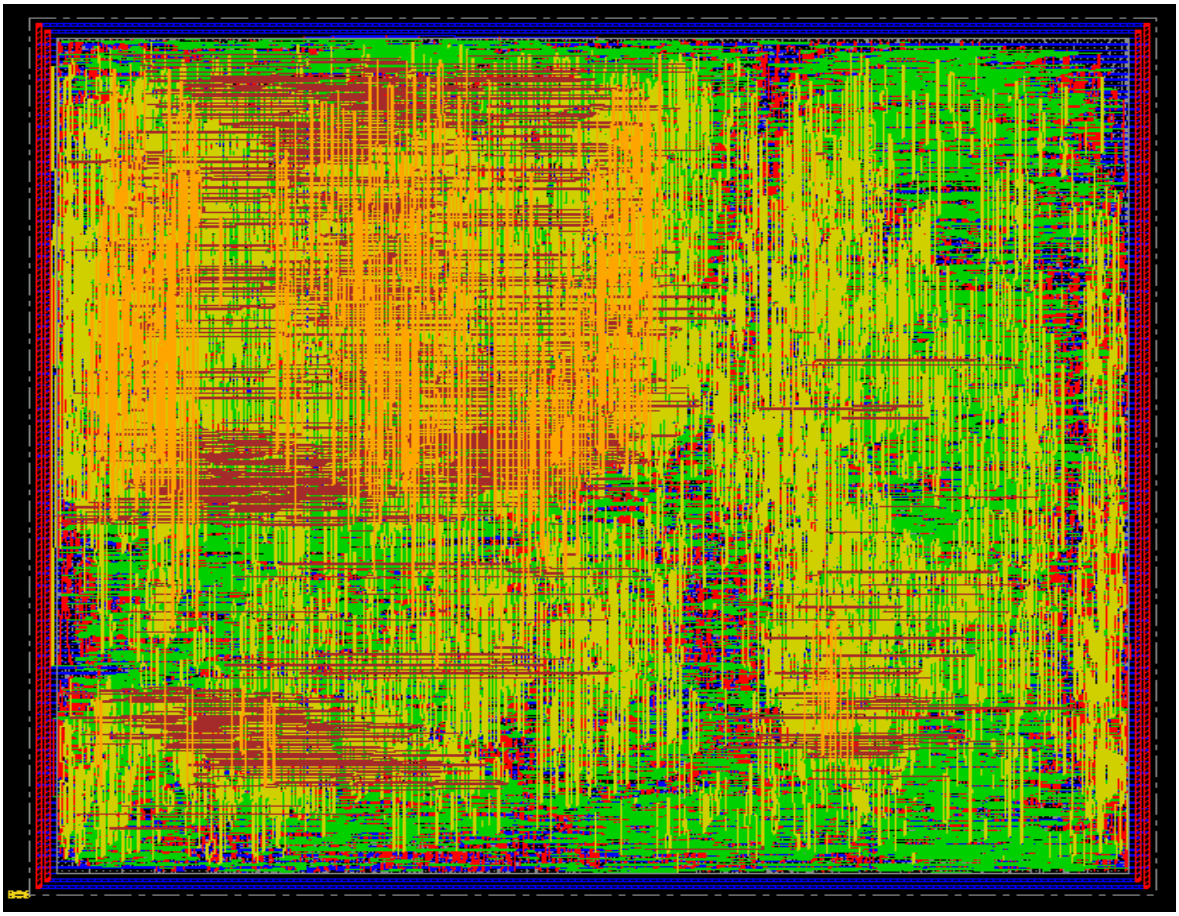

Figure 2.6: post-place&route modified RISCV-lite processor

2.3.1 Timing and Area

The final maximum frequency is 555.5MHz, with a clock period of 1.80ns. From the output of the *report_timing* command on Design Compiler, saved in *timing_report_RISC_V_ABSV.txt*, it can be noticed that the critical path is the one involving the component *Branch_Target_Buffer* as in the previous design. However, in this case, the startpoint is the reset signal and the endpoint consists in a memory element instantiated in order to implement the branch history table. Moreover, both *.slk* files generated by Innovus in the post-route phase have been checked in order to verify that no negative slack is present. From the *.slk* file obtained setting *Analysis Type* to *Setup*, it can be observed that the worst slacks correspond to *OUTALU_IN_MEM_reg* and *BRANCH_TARGET_BUFFER_BTB_reg*. On

the other hand, from the *.slk* file obtained setting *Analysis Type* to *Hold*, it can be noticed that the worst slacks correspond to *INSTRUCTION_IN_EX_reg*, *PC_IN_ID_reg* (similarly to the first design case) and *CONTROL_UNIT_ALU_INST_reg*.

As regards the occupied area, the output of the *report_area* command on Design Compiler has been saved in the file *area_report_RISC_V_ABSV.txt*. The reported total cell area is 23944.5µm^2, similar to the value retrieved in the *.gateCount* file generated by Innovus, which is 24266.1µm^2. As expected, both values are greater than the two retrieved in the previous design.