Integrated Systems Architecture

Politecnico di Torino

Department of Electronics and Telecommunications

Laboratory 3: RISC-V special project

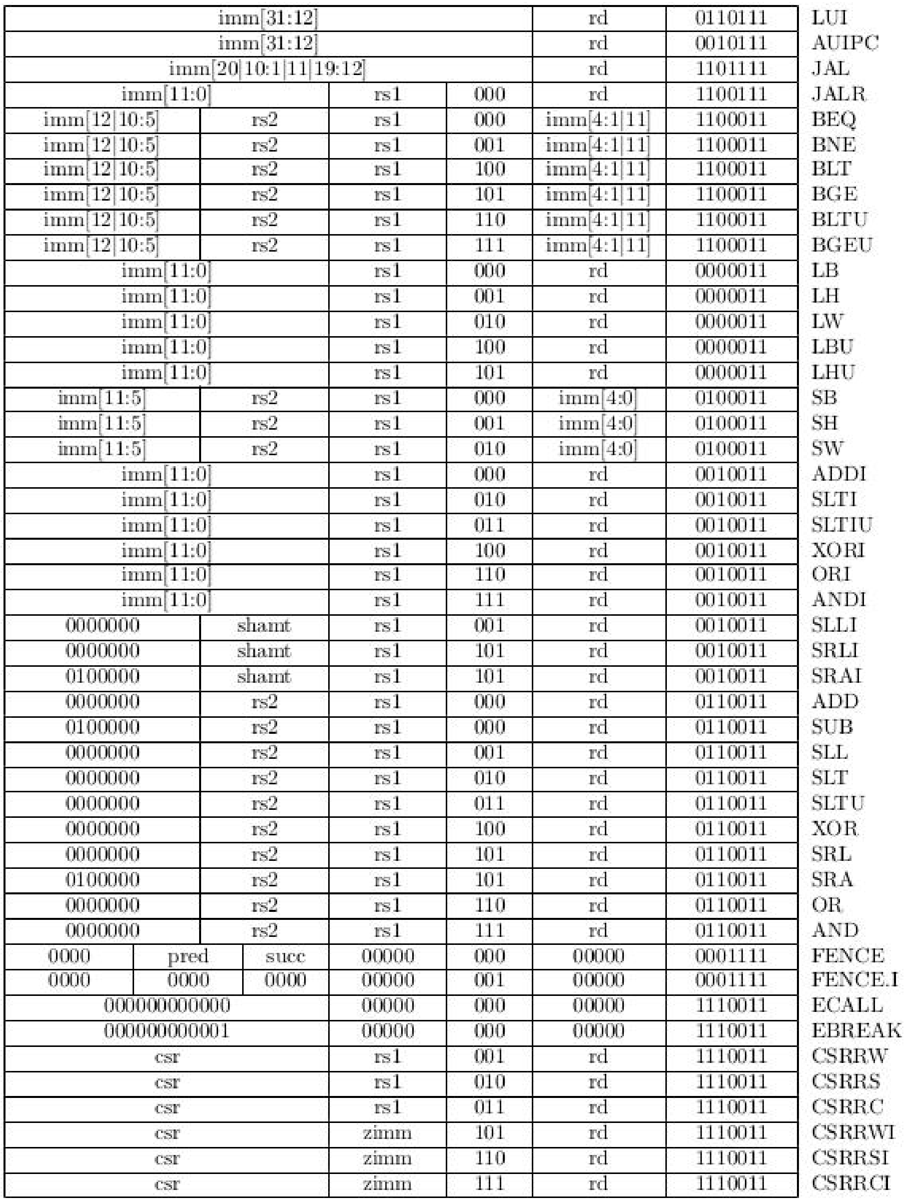
Authors:
Marco Andorno (247222)
Michele Caon (253027)
Alessio Colucci (251197)
Matteo Perotti (251453)

March 20, 2019

Contents

1 Introduction

The aim of this laboratory session is to design a simple RISC-V core implementing the RV32I instruction set in SystemVerilog, which is the basic 32-bit integer-only set of instructions, without multiply and divide. The complete set is shown in figure 1.



imm[31:12]					rd	0110111	LUI
imm[31:12]					rd	0010111	AUIPC
imm[20|10:1|11|19:12]					rd	1101111	JAL
imm[11:0]			rs1	000	rd	1100111	JALR
imm[12|10:5]		rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]		rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]		rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]		rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]		rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]		rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]			rs1	000	rd	0000011	LB
imm[11:0]			rs1	001	rd	0000011	LH
imm[11:0]			rs1	010	rd	0000011	LW
imm[11:0]			rs1	100	rd	0000011	LBU
imm[11:0]			rs1	101	rd	0000011	LHU
imm[11:5]		rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]		rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]		rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]			rs1	000	rd	0010011	ADDI
imm[11:0]			rs1	010	rd	0010011	SLTI
imm[11:0]			rs1	011	rd	0010011	SLTIU
imm[11:0]			rs1	100	rd	0010011	XORI
imm[11:0]			rs1	110	rd	0010011	ORI
imm[11:0]			rs1	111	rd	0010011	ANDI
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		rs2	rs1	000	rd	0110011	ADD
0100000		rs2	rs1	000	rd	0110011	SUB
0000000		rs2	rs1	001	rd	0110011	SLL
0000000		rs2	rs1	010	rd	0110011	SLT
0000000		rs2	rs1	011	rd	0110011	SLTU
0000000		rs2	rs1	100	rd	0110011	XOR
0000000		rs2	rs1	101	rd	0110011	SRL
0100000		rs2	rs1	101	rd	0110011	SRA
0000000		rs2	rs1	110	rd	0110011	OR
0000000		rs2	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Figure 1: RV32I instructions

Actually, not all the instructions were implemented as the support for an operating system and exception handling is beyond the scope of this experience.

The different formats of the instructions are shown in figure 2, which is useful reference in the rest of this report.

1.1 Pipeline structure

From an architectural point of view, the design is based on the classic RISC 5-stage pipeline, divided as follows:

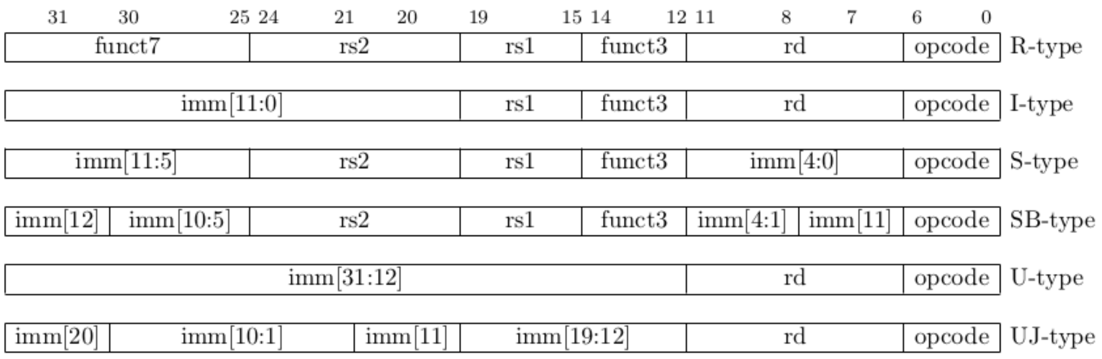

31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | SB-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | UJ-type

Figure 2.3: RISC-V base instruction formats showing immediate variants.

Figure 2: Instruction formats

1. Instruction fetch (IF): the new instruction if read from the instruction memory, pointed by the current Program Counter (PC).
2. Instruction decode (ID): operands are read from the register file (RF), the control unit generates control signals for the following stages, immediate fields are extended on 32 bits and the ALU controls are decoded.
3. Execute (EX): the ALU performs the required operation and the new PC is computed in case of branch or jump.
4. Memory access (DMEM): the data memory is read or written for instructions that require so (load and stores), otherwise data just bypasses this stage.
5. Write back (WB): either the ALU result or the data memory output is written to the destination register, when required.

Pipeline registers separating each stage take the name of the two stages they are in between (e.g. ID/EX). Actually, assuming a synchronous memory interface, two of these registers are in part bypassed by the memory timing, in order to keep the number of stages at five. For more details on this matter, refer to section 5.1.

The following sections describe the different datapath and control blocks and the testing methodology.

2 Datapath

2.1 Register file

The RISC-V register file is composed of 32 registers, each 32-bit wide (for RV32I), called `x0` to `x31`, where `x0` is a special register hardwired to the value 0, which can turn useful for some instructions. Figure 3 shows the top level diagram of the register file structure.

Writes to the register file are, of course, synchronous and happen on the positive edge of the clock. For a correct write operation, the destination register must be selected using the `writeAddr` port, the input data must be placed on the `dataIn` port and the signal `regWrite` must be asserted. Internally, the register file will enable only the selected register using a decoder.

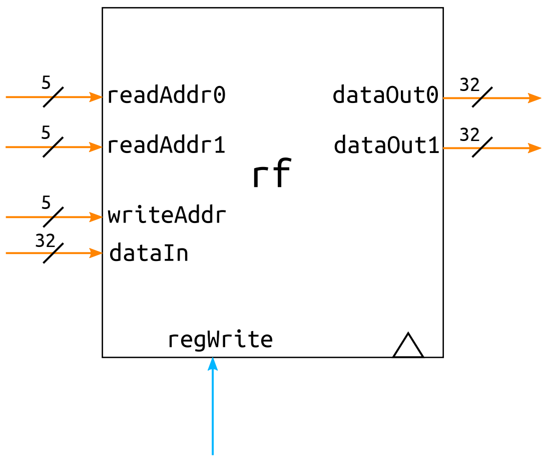


Figure 3: Register file

Reads are instead combinational and can occur on two different registers at a time, thanks to two different read ports. To select the correct output value, a 32-to-1 multiplexer is used on each read port. However, in order to avoid data hazards during the write back stage, the register file also implements bypassing of input data directly to the output if the same register is read and written during the same clock cycle. Figure 4 shows this read selection process (no multiplexer is used to select 0 in case the register being read is `x0` as we can suppose it is hardwired directly at its output).

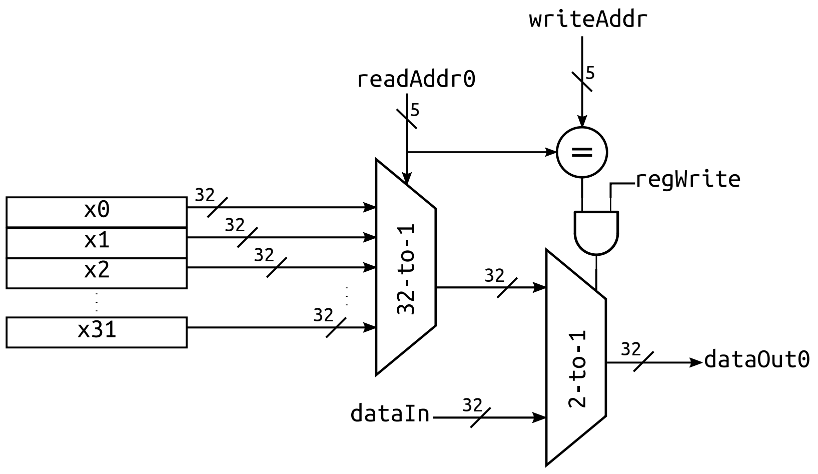


Figure 4: Read operation in the register file

To better illustrate the behavior of the register file operations, their timing diagram is shown in 5.

2.2 ALU

The ALU is in charge of performing all operations required by arithmetic and logic instructions, load and store, and branch comparison. Figure 6 shows its top level block diagram, which is simply composed of two inputs and one output on 32 bits, along with a 4-bit control signal to select the desired operation.

The complete list of operation that the ALU can perform is the following (in the order in which they are defined on the `ctl` input):

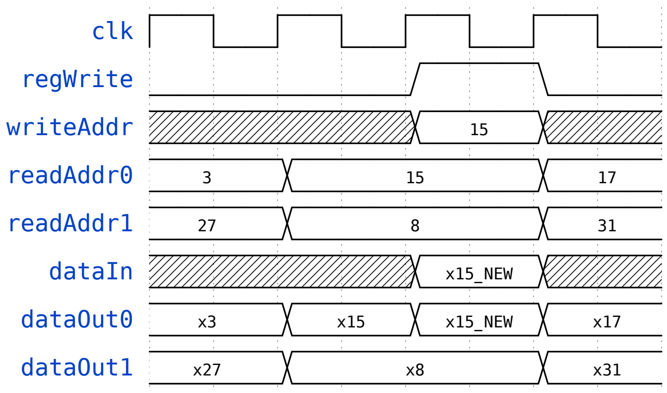


Figure 5: Register file timing

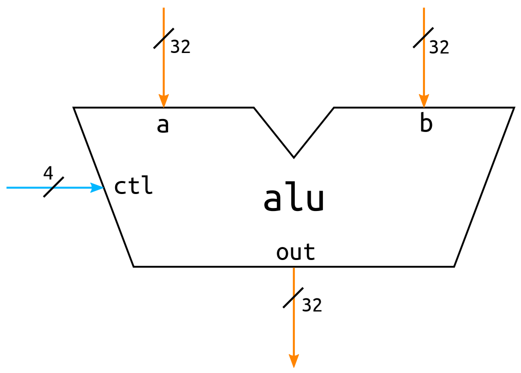


Figure 6: ALU

1. Add
2. Subtract
3. AND
4. OR
5. XOR
6. Left shift
7. Right shift
8. Right shift with sign extension
9. Set if equal
10. Set if not equal
11. Set if less than
12. Set if greater or equal than
13. Set if less than unsigned
14. Set if greater or equal than unsigned
15. AUIPC (add current PC and left-shifted immediate)

Note that all operations were described behaviorally as per specifications, in order to be as implementation independent as possible and open to every optimization that a synthesis tool can perform.

All 'set if *' operations set the output to the value 1 (`0x00000001`) if the condition is true, or 0 otherwise. This approach was chosen instead of using flags (such as Carry, Overflow, Negative and so on) to compute conditions as it was deemed simpler to implement and thorough enough, given that there would not have been other uses for the flags.

2.2.1 ALU decoder

The control input of the ALU is generated by a special decoder starting from the opcode, funct3 and funct7 fields of each instruction which requires an ALU operation, as described in figure 7. This block consists only of a series of conditional statements (that can easily be mapped to multiplexers) which select the correct control signal.

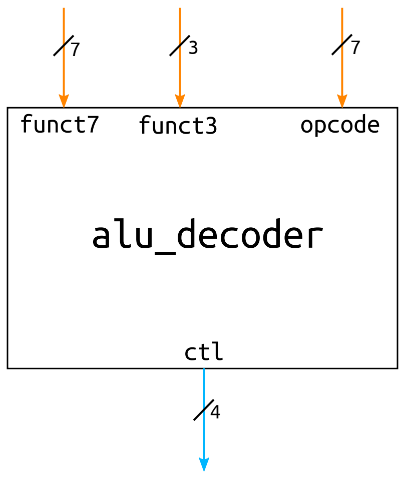


Figure 7: ALU decoder

Another approach would have been to split the control of the ALU into two decoding steps[1], but a preliminary analysis concluded that no practical advantage would be obtained this way. Moreover, only few bits of the input fields are required to make a definite decision on the control output, but in order to keep modularity and continuity in the design, the whole fields are given in the interface of the block.

2.3 Branch and Jump management

A general view of the unit is given in figure 8, along with the significant datapath blocks involved.

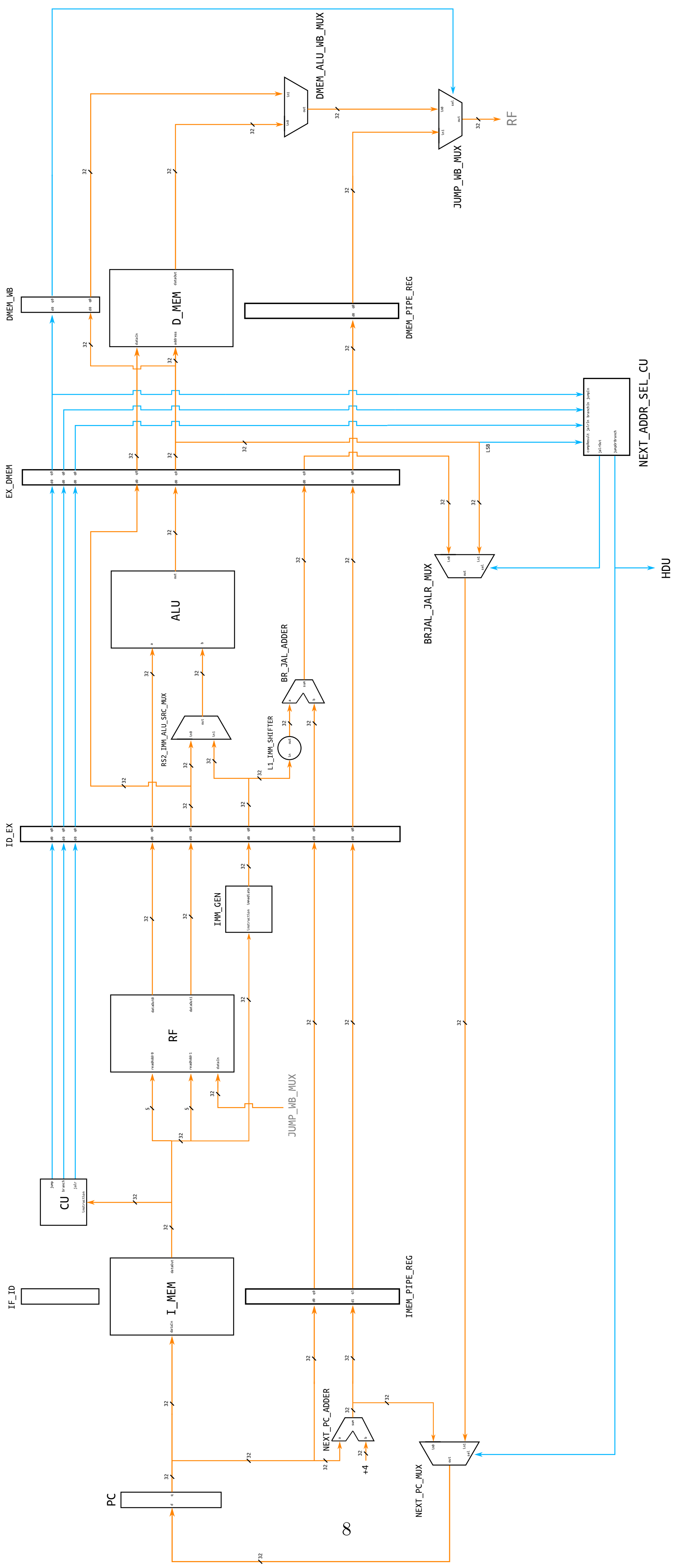
2.3.1 Types of instructions

There are two classes of instructions which can lead to a modification of the sequential flow of the program. In the RV32I ISA they are:

1. Branches

2. Jumps

[1]As suggested in Chapter 4 of D. Patterson, J. Hennessy, *Computer Organization and Design RISC-V Edition*, Morgan Kaufmann, 2017


PC
I_MEM
IF_ID
IMEM_PIPE_REG
NEXT_PC_ADDER
+4
NEXT_PC_MUX
CU
RF
JUMP_WB_MUX
IMM_GEN
ID_EX
RS2_IMM_ALU_SRC_MUX
L1_IMM_SHIFTER
BR_JAL_ADDER
ALU
EX_DMEM
BRJAL_JALR_MUX
HDU
LSB
NEXT_ADDR_SEL_CU
D_MEM
DMEM_WB
DMEM_PIPE_REG
DMEM_ALU_WB_MUX
JUMP_WB_MUX
RF

The former is a conditional change of the usual choice of the next address to be put in the PC, whereas the latter is unconditional. The condition, whenever present, is always based on the result of an ALU comparison.

Branch instructions exist in different flavours, depending on which comparison has to be performed between the content of two registers. Here follows a list of all of them:

1. BEQ (branch if equal)
2. BNE (branch if not equal)
3. BLT (branch if less than)
4. BGE (branch if greater or equal than)
5. BLTU (branch if less than unsigned)
6. BGEU (branch if greater or equal unsigned)

All these instructions belong to the B-type ones (figure 2). They have an immediate field split along the word, which indicates an effective immediate divided by two. Indeed, with a RISC-V standard architecture it is possible to address this way an halfword at most, but never the single byte. The effective immediate is calculated with a bit reorganization, a sign extension and a left shift of one position, to reach the final 32-bit width.

The instruction contains also the addresses of two registers, whose content will be compared by the ALU to decide whether to take the branch or not. To distinguish which type of comparison is needed, it is necessary to know the instruction field funct3.

Jump instructions can be of two types, each bringing to a different hardware path for the data:

1. JAL
2. JALR

JAL is a J-type instruction, whereas **JALR** is an I-type one. This difference is reflected on the hardware implementation, more on this later. A jump instruction is unconditional, but still need for an address computation. This operation is different for the two instructions: for JAL it is sufficient to use the same hardware used for branch address computation, whereas JALR requires the non shifted immediate to be added to the content of a register (the next address is not derived by the current one).

2.3.2 Instruction execution

Since no **BPU** (**B**ranch **P**rediction **U**nit) is present in the design, a "branch not taken" assumption is always made when the content of the PC is updated and the decoded instruction is a branch. The simplest way to manage a branch is to delay the decision until the execution stage, waiting for the ALU to do the comparison. The effective decision is then taken in the DMEM stage, not to exacerbate a path which can be critical by itself. Also the calculation of the next address, which involves the immediate and the program counter, is performed in the execution stage.

A possible improvement could be to anticipate the comparison and the next address calculation in the decoding stage, but to keep the design simple the first solution was chosen, as this would imply additional hardware. This is compliant with the calculation of the address for a JAL instruction. It is worth to mention that the absence of a condition to be verified is enough to simplify the anticipation of the address calculation and bring it in the decode stage. However this solution would increase the number of resources if the other branch/jump instructions are still executed in an another stage.

A JALR instruction behaves in a slightly different way: the address calculation is performed by the ALU, because the immediate is added to the content of a register.

A branch instruction has no side effects once it has been executed. On the contrary, a jump instruction leaves inside the pipeline the next instruction address to be saved in a destination register. This is not a issue though, because it is possible to see that even without forwarding units no data hazards can arise. If the pipeline was longer, maybe the forwarding unit would be the only thing to have the day saved (the design has it, though).

2.3.3 Effective calculation

The address calculation in case of branches/jumps is performed in the execution stage and it depends on the type of instruction:

- Branch/JAL: it is based on the "current" PC value (e.g. current for the instruction in that stage). The immediate is sign extended, one position left shifted and added to the PC value (percolated through the pipeline until there) by means of another adder. In the meantime, if the instruction is a JAL, the address of the next instruction goes on through the stages.
- JALR: it involves a sum between an immediate and the content of a register. The ALU performs this operation without shifting the immediate. When the result has to be used, the LSB is substituted with a zero. Even in this case, the address of the next instruction follows its path towards the write-back stage.

2.3.4 Next address selection CU

To control the multiplexers for the next address selection, there's the need for knowing:

1. Whether the instruction in the DMEM stage is a branch or a jump.
2. Which is between the two.
3. The result of the comparison.
4. If the instruction is a JALR.

The main CU generates two signals `branch` and `jump` which percolate along the pipeline, to allow the "Next address selection CU" to solve the first two points. The result of a comparison is simply the LSB of the ALU result. The main CU thus has to generate another signal `jalr` to indicate a JALR instruction.

The `jump` control signal is used also in the write-back stage, to select the right input for the register file. If a jump is performed, the data to be written in the destination register is the "next" address after the jump instruction.

In any case, the IMEM pipe register, together with IF/ID, ID/EX and EX/DMEM ones, have to be flushed. This brings to a performance loss of 4 instructions for each taken branch or executed jump.

For details about the `NEXT_ADDR_SEL_CU`, refer to figure 9 and 10.

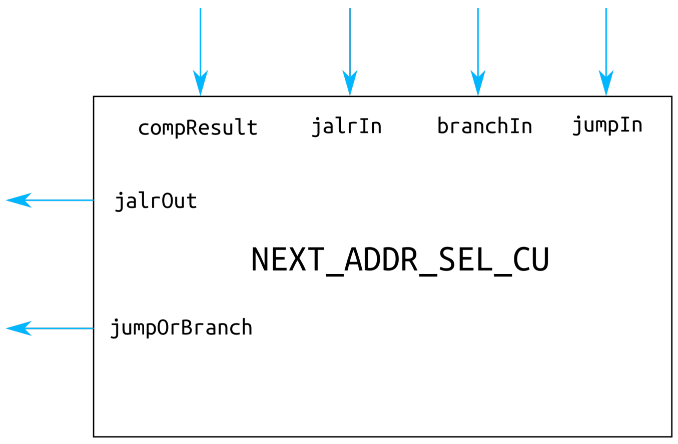


Figure 9: Next address selection CU, for jumps and branch management

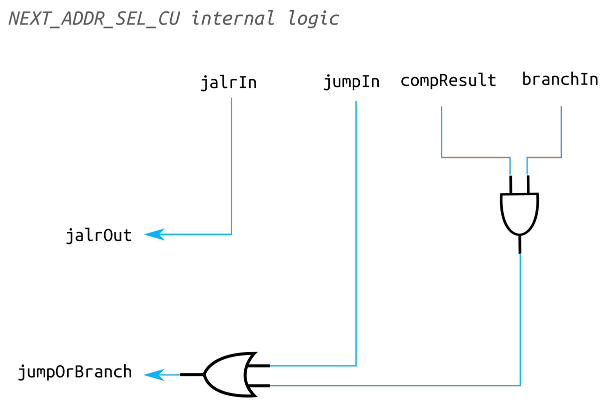


Figure 10: Next address selection CU internal logic

2.3.5 Next address generation

The next address is chosen by means of two multiplexers:

- `BRJAL_JALR_MUX` takes in input the result of the ALU with the LSB masked and the output of the additional adder of the execution stage. These two input come from the EX/DMEM pipe register.
- `NEXT_PC_MUX` takes in input the output of `BRJAL_JALR_MUX` and the current PC + 4

3 Control

3.1 Forwarding Unit (FWU)

Forwarding allows to avoid stalling the pipeline when a data hazard occurs between two subsequent instructions, if the needed data is already available in a following pipe stage. In this five stage pipeline results are written to the destination register

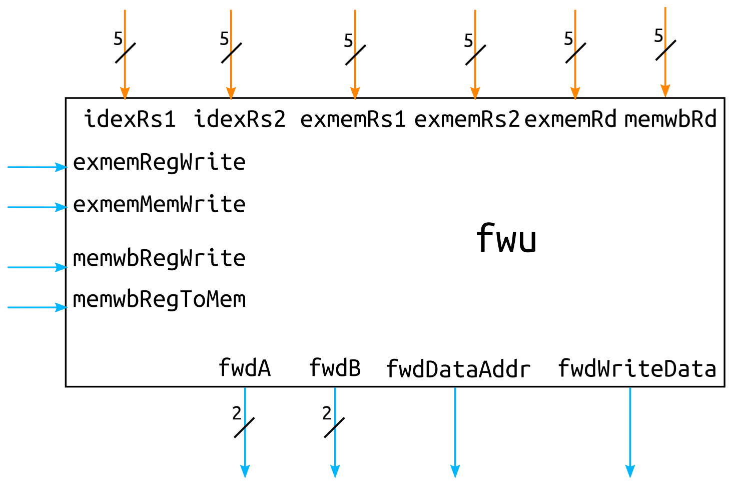


Figure 11: Forwarding Unit

during the write back stage, three clock cycles after the operand read in the decode stage. This means that, if an instruction modifies a certain register, only instructions starting from the third after the original would read the correct new data, or equivalently that up to two bubble should be inserted in case of a data hazard.

Forwarding simply bypasses data to the beginning of the execution stage (i.e. at the ALU inputs) if the required results are already present at the ALU output or in the following memory access stage.

To do so, the logic of the FWU (figure 11) performs some checks:

- Check that the earlier instruction in the pipe actually modifies some register (`regWrite` is asserted) and that the address does not point to register `x0`.
- Compare the destination register in the EX/MEM stage with both source register in the ID/EX stage and, if there is a match, drives the selection signal of the corresponding ALU input multiplexer (`RS{1,2}_ALU_FWD_MUX`) to select the previous ALU output (`fwdA/fwdB` = 10).
- Otherwise, compare the destination register in the MEM/WB stage with both source register in the ID/EX stage and, if there is a match, drives the selection signal of the corresponding ALU input multiplexer to select the result currently in the memory access stage (`fwdA/fwdB` = 01).

Note that, according to the list above, forwarding gives precedence to data present in the EX/MEM stage over the MEM/WB stage if the same register is present in both, as the former contains the latest result.

3.1.1 Load/store forwarding

The designed forwarding unit handles also the another special case of data hazard that occurs when a load is followed immediately by a store to the same memory location, such as in memory to memory copies.

In this case the FWU compares the destination register in MEM/WB stage with both the source registers in the EM/MEM stage. If the earlier instruction, in MEM/WB, is a load (`memwbMemToReg` is asserted) and the current instruction in EX/MEM is a store (`exmemMemWrite` is asserted):

- If the match is on `rs1` field, it means that the value just read from the memory is used as base address for the following store instruction. Therefore, `DMEM_ADDR_FWD_ADDER` sums the store instruction immediate field present in the EX/MEM pipe register to that value, and the result is selected by `DMEM_ADDR_FWD_MUX` as the data memory **address** input. The following two lines of assembly code show this case:

```
lw x5, 0(x0)
sw x6, 4(x5)
```

- Otherwise, if the match is on `rs2` field, the value just read from the memory has to be forwarded to the memory **data** input by `DMEM_DATA_FWD_MUX`. This second possibility is shown below:

```
lw x5, 0(x0)
sw x5, 4(x6)
```

This design allows the processor to cover every type of data hazard that can be solved by forwarding earlier results, as proven by the simulation results, leaving uncovered only the case of a load instruction followed by another instruction using that value that is not a store operation. In this case the processors stalls the pipeline inserting a `nop` instruction. This situation is handled by the Hazard Detection Unit (HDU, section 3.2). Figure 12 shows the FWU with all inputs on which decisions are taken and the three output signals controlling the related multiplexers in each pipeline stage of the core.

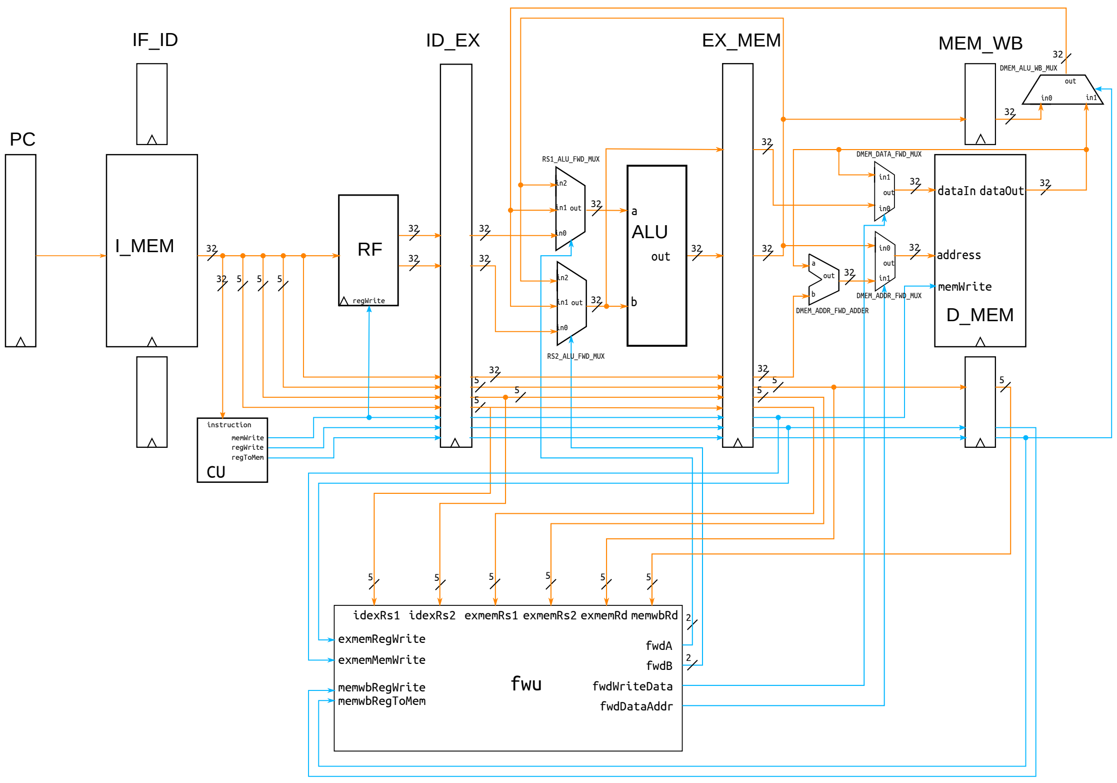


Figure 12: Forwarding Unit connection to the datapath of the core

3.2 Hazard Detection Unit (HDU)

When data cannot be forwarded, then the pipeline must be inevitably stalled by preventing the fetch of a new instruction and inserting a bubble. Specifically, this happens when a data hazard occurs between a load (`memRead` asserted in the decode stage) and another instruction using the value read from the memory, unless it is a store, for which forwarding accounts as explained in section 3.1.1. Moreover, when a branch is taken or an unconditional jump occurs, a similar action must be take and additionally the entire pipeline must be flushed, to get rid of invalid instructions already fetched decoded while waiting for the jump condition result or the destination address to be computed.

Both this occurrences are handled by the Hazard Detection Unit (figure 13), that according to the aforementioned checks, outputs three signals:

- `stall_n`: active low, is connected to the enable of the Program Counter and the IF/ID pipe register to prevent them from changing in the event of a stall.
- `flushIdEx`: to drive the multiplexer inserting the NOP in the ID/EX register, that will propagate to the rest of the pipeline, in case of a stall or a jump.
- `flushIfIdExMem`: to drive the multiplexer inserting the NOP in the IF/ID and EX/MEM registers in case of jump.

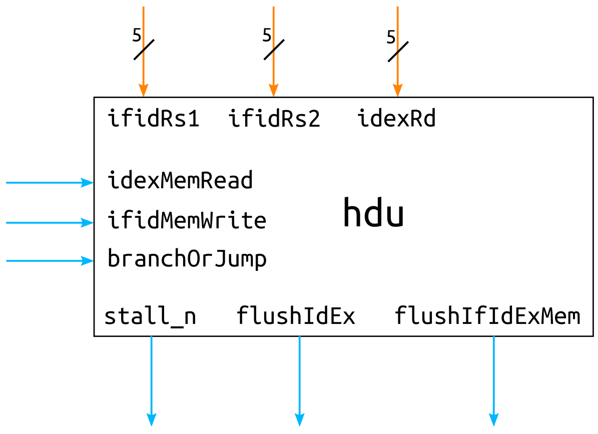


Figure 13: Hazard Detection Unit

A summary of the possible cases when this can occur is reported in table 1:

	`stall_N`	`flush_IdEx`	`flush_IfId_ExDmem`
No Hazard	0	0	0
Control Hazard	0	1	1
Data Hazard	1	1	0

Table 1: HDU output

3.2.1 Inserting a NOP

The NOP instruction is not present in the RISC-V ISA. It is possible to emulate it though, using an `ADDI x0 x0 0`. This instruction does nothing, because the register x0 is hardwired to value 0. The instruction NOP belongs to the set of pseudo-assembly instructions: they are translated in RISC-V language on the fly by the assembler, and they exist for programmers ease only.

For an effective NOP insertion in the IF/ID stage there is the need for a sequential control of the multiplexer which drive the source of the RF. There is no way of doing it before the pipe register, unless a NOP instruction is already present somewhere in memory.

Particular caution must be taken when the first instruction after the reset is de-asserted modifies the behaviour of the processor for the next cycles. This is the case of a `jal` instruction stored in the first instruction memory location. While the reset signal is asserted, this instruction continues to be fetched, since there's no way to reset the synchronous reset. The `jal` instruction enters the next pipeline stages as soon as the reset signal is de-asserted (even in the middle of a cycle, since it is asynchronous), and the CU issues a flush of the next fetched instructions, including the one that is fetched on the next cycle after the reset signal is de-asserted, that is of course the `jal` instruction itself. When this happens, the control signals used to compute and use the destination address of the jump are therefore overridden by the flushed issued before. The `IFID_FLUSH_FF` must therefore be reset to `1` instead of `0`, to propagate a `nop` instruction as long as the reset signal is asserted. This way, no real instruction enters the pipeline before the reset is released, allowing the first instruction in the memory to be properly executed, whatever it is.

Notice that another possible solution is to change the Control Unit from a simple combinational network to a Finite State Machine. However, to avoid state explosion and complex pipeline synchronization, a Mealy approach could have been used in place of a Moore one. This means that the output control signals would have been functions of the input ones (like in a combinational network) and the present state. Since the state change is a synchronous event during normal operation, the control signals would have been be issued only in states different from `RST`.

In figure 14 a high level datapath for hazard management is depicted. Moreover, a timing diagram to show how instructions are flushed is shown in figure 15. The three instructions which follow a jump or a taken branch are discarded.

3.3 Control Unit (CU)

The Control Unit is shown in figure 16. It is a combinational component, because all the synchronization is done in the pipeline registers. It basically reads the opcode and other eventual flags (funct3, funct7) to determine the format of the instruction used in the immediate generator, as well as other flags.

These flags are about memory commands (read, write, how much to read/write), register-file commands (write, address is generated in the remaining datapath), MUX commands (to decide the inputs to use for the ALU) and branch/jump commands.

There are two commands which are a bit less standardized, to tell the memory the dimension to access (byte, halfword, word) and to tell apart the different jump conditions like branch, jal, jalr.

It is implemented in a behavioral way, to avoid possible risks, and all the parameters are defined as constants, to be easily updated if needed.

3.4 Immediate Generator (ImmGen)

The Immediate Generator (show in figure 17) is needed to produce the correct immediate bit sequence from each different instruction formats. The type of instruction is inferred by the Control Unit, and the output is the reconstructed immediate. The immediate generation is done in the same cycle, since this is a combinational component.

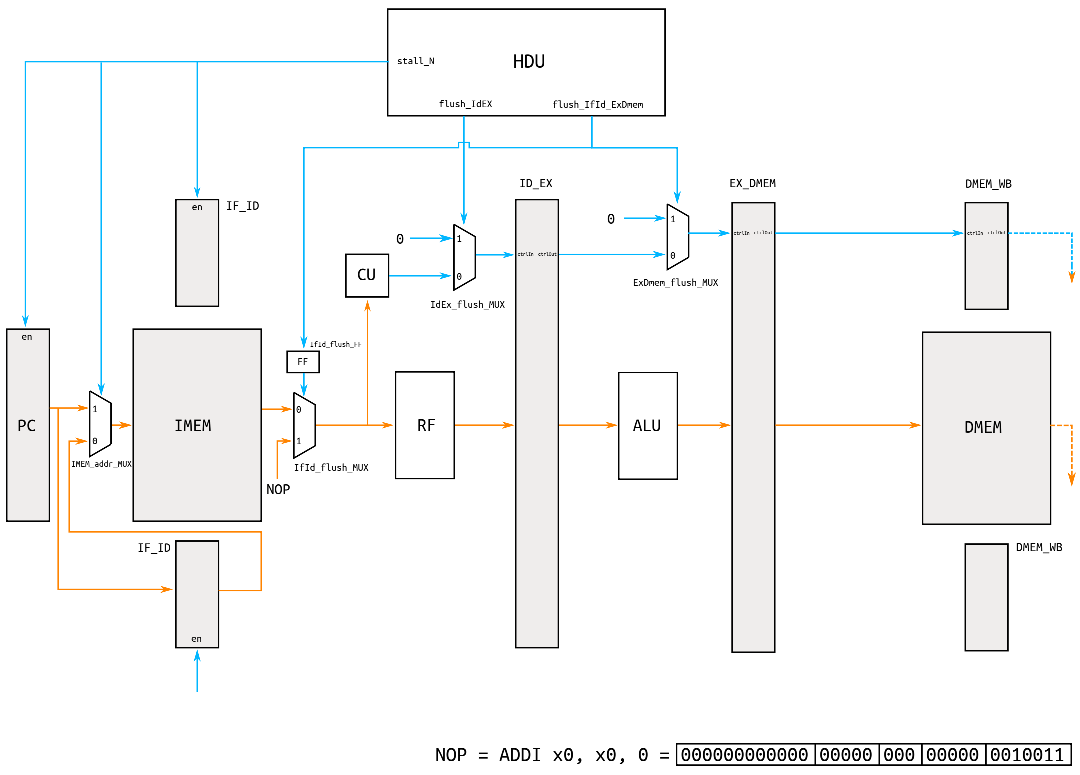


Figure 14: Hazard management HW

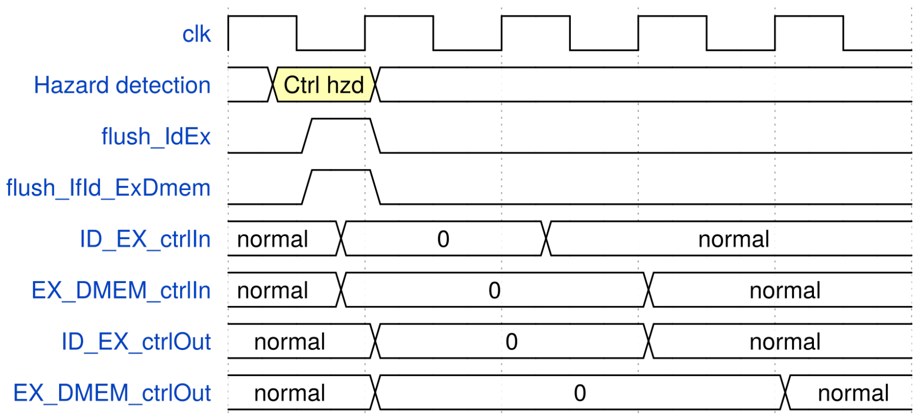


Figure 15: Hazard management timing

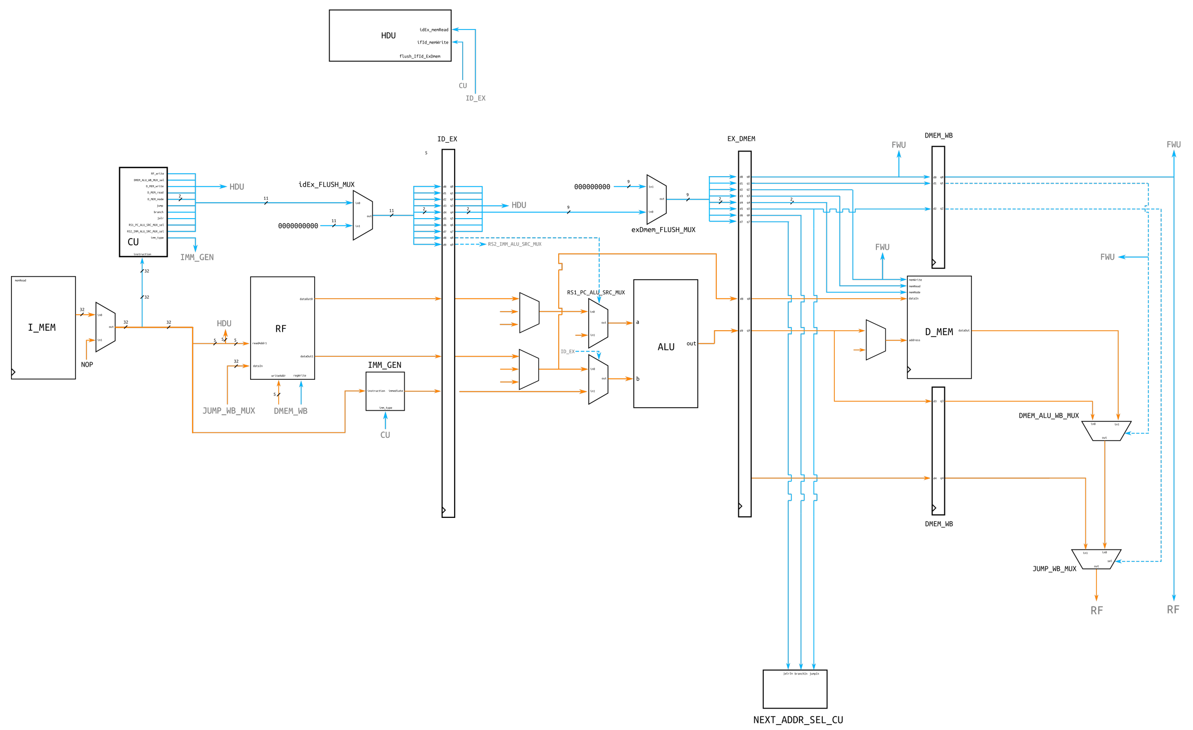


Figure 16: Control Unit and the units connected to it

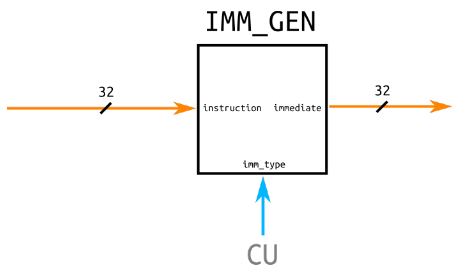


Figure 17: Immediate Generator

4 Main architecture

The main architecture is depicted in figure 18. One multiplexer was added in the execution stage to add the support for the AUIPC operation.

5 Simulation

5.1 Memory

Some kind of memory model is needed in order to perform a full simulation of the processor core. Given that the design of a full fledged memory subsystem is beyond the scope of this experience, we resorted to a simple behavioral model of a synchronous memory.

This model is only intended for simulation purposes and does not map a real memory chip on its own, but emulates the function of a more complex memory controller able to select individual bytes among a 32-bit word both in read and in write operations.

Figure 19 shows the interface of this block, where the `address` is left parametric, as it can differ between instructions and data memory. Note that compliant to the RISC-V byte addressing specification, each address represents a single byte, even if the data width is always 32 bits, which is the width of the data bus of the architecture. The data width for load and stores is selected by the `addrUnit` signal, according to the following encoding:

- 00: byte
- 01: halfword (16 bits)
- 10: word (32 bits)

Independently of the data width chosen, the correct output is always provided within a single clock cycle. It this behavior was to be replicated on a real byte-addressed memory chip, it would take (at most) four read operations and a clock four times faster.

Read and write operations are handled by the couple of control signals `memRead` and `memWrite`, of which only one should be asserted at each clock cycle to perform the desired action. Both signals active represent an forbidden condition and should be avoided by the whatever is in charge of controlling the memory.

Figure 20 shows the usual timing diagram of this fully synchronous memory, according to which both reads and writes take place at the next clock cycle after the proper control signals are asserted.

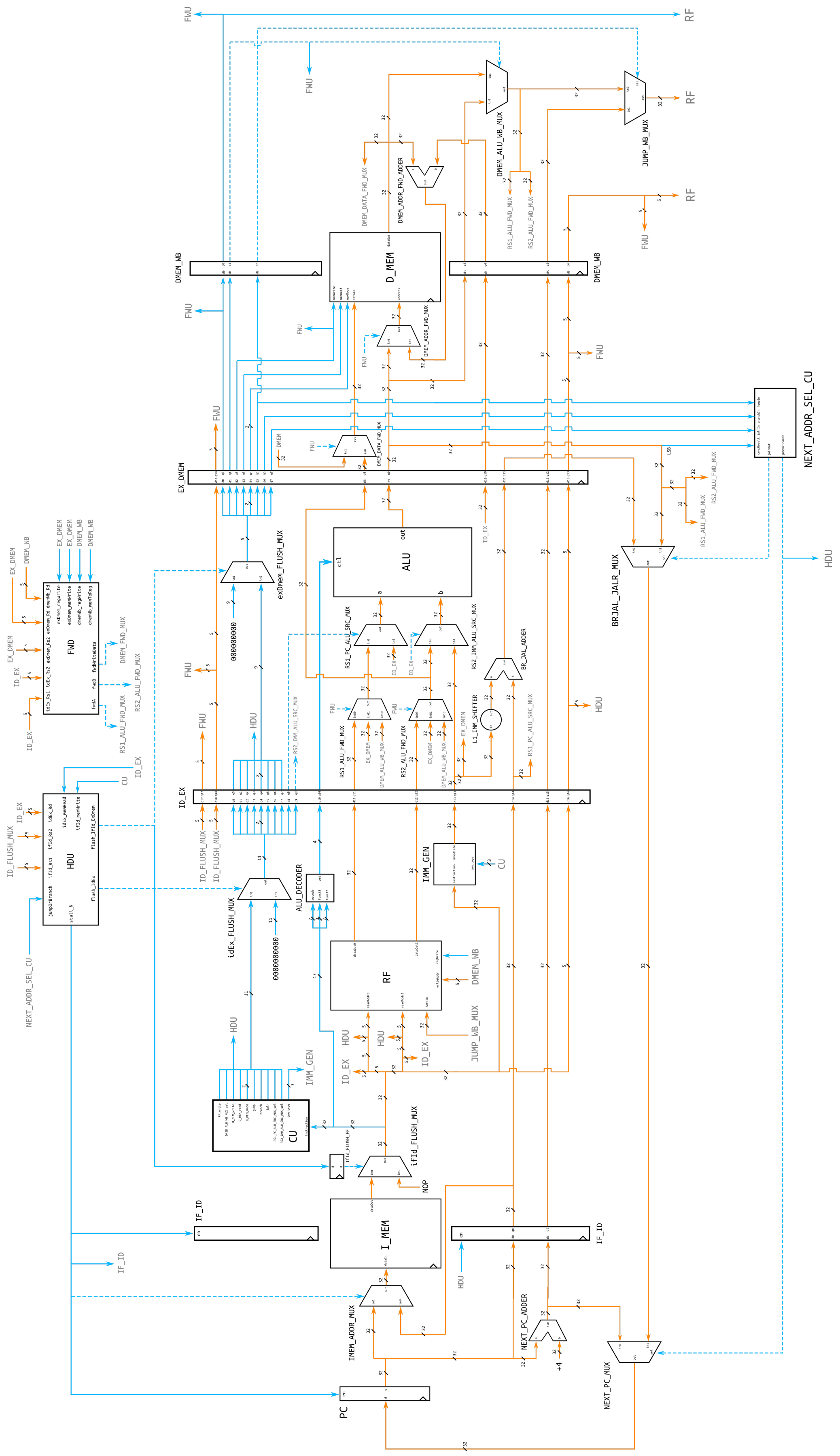


Figure 18: RV-MAGIC main architecture

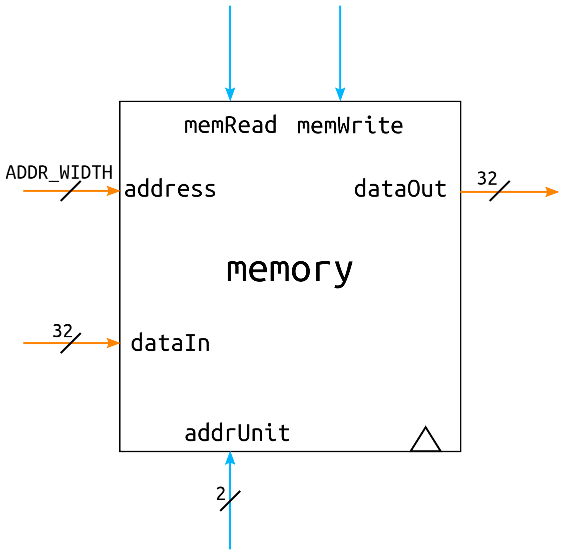


Figure 19: Memory

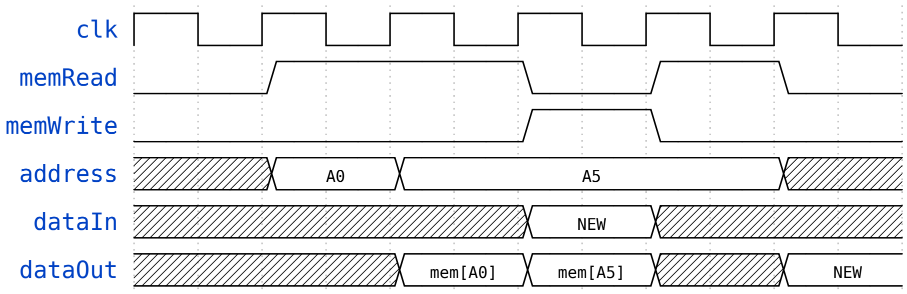


Figure 20: Memory timing diagram

5.2 Testing procedure

To test the whole processor, some patterns of instructions were developed. The main purposes were to check:

1. the correct behavior of each instruction in a flow without hazards
2. the correct management of all the hazards

To achieve this in an easier way, an assembler was developed to support a translation from the RISC-V assembly to the machine code. A detailed discussion about this tool is provided in section 5.3. Initially a search on the net was done to find a toolchain to convert high level languages like C in RV32I instructions, but a lot of issues arose and other strategies were chosen. The developing of an assembler is not critical, because of its static nature: it works like a decoder and it does not need to be "clever" like a compiler. Testing needs weren't critical and a string check processing was enough.

5.2.1 Testbench

The testbench was written in SystemVerilog coherently with all the previous work. The entity instantiates the **DUV** (**D**evice **U**nder **V**erification) together with the two memories. It also handles the clock and reset generation. It is worth to note that the addressable space of the memory was reduced because a complete 32-bit one was not feasible due to space problems: only a subset of the PC bits was bound to the address line of the storage devices.

The entire system is reset at the beginning of the simulation, and a parametric clock is fed to all the sequential elements. An `initial` statement inside the memory modules ensures the correct loading of the instructions/data inside them. To perform this task, the function `$readmemh` is used: it allows to read an ASCII text file in which are present hexadecimal data written in rows. Each row is assigned to a memory location.

With the aid of the ModelSim GUI, each signal was visually followed to check if the timing of the processor was respected.

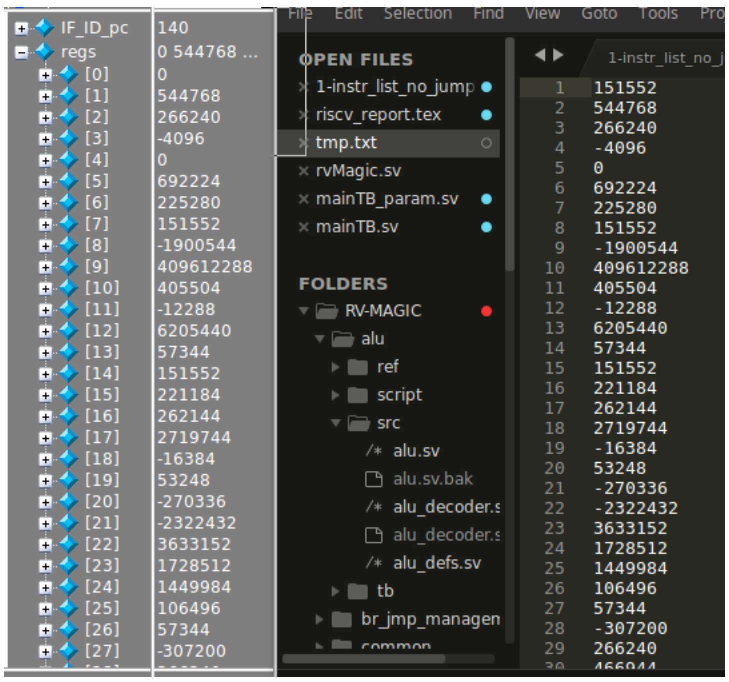
5.2.2 Single instructions test

First of all each instruction was individually tested to assure the correct behavior of the whole pipeline mechanism.

Register loads To test each register of the register file, as well as to make sure that register `x0` does not change, a series of consecutive `lui` instructions was used, trying with both positive and negative values. The `auipc` instruction was tested as well at this point.

```
lui x0, 37
lui x1, 133
lui x2, 65
lui x3, -1
lui x4, 0
...
auipc x2, 6466
```

As an example, figure 21 shows the completion of the register file loading test.



```
IF_ID_pc  140
regs      0 544768 ...
[0]       0
[1]       544768
[2]       266240
[3]       -4096
[4]       0
[5]       692224
[6]       225280
[7]       151552
[8]       -1900544
[9]       409612288
[10]      405504
[11]      -12288
[12]      6205440
[13]      57344
[14]      151552
[15]      221184
[16]      262144
[17]      2719744
[18]      -16384
[19]      53248
[20]      -270336
[21]      -2322432
[22]      3633152
[23]      1728512
[24]      1449984
[25]      106496
[26]      57344
[27]      -307200
```

```
151552
544768
266240
-4096
0
692224
225280
151552
-1900544
409612288
405504
-12288
6205440
57344
151552
221184
262144
2719744
-16384
53248
-270336
-2322432
3633152
1728512
1449984
106496
57344
-307200
266240
```

Figure 21: RF loading verification

Immediate ALU instructions Next, we tried all ALU operations involving immediate operands.

```
addi x4, x3, -56
andi x6, x5, 255
ori x8, x7, 255
xori x10, x9, 255
slli x12, x11, 3
srli x14, x13, 4
srai x16, x15, 8
slti x18, x17, 0
sltiu x20, x19, 5
```

Register ALU instructions Then, the same instructions were tried with both operands being read from the register file.

```
add x23, x21, x22
sub x26, x24, x25
and x29, x27, x28
or x1, x30, x31
xor x4, x2, x3
sll x7, x5, x6
srl x10, x8, x9
srai x13, x11, 12
slt x16, x14, x15
sltu x19, x17, x18
```

Load and stores Finally all load and stores were tested, on all the different data widths.

```
lb x21, 3(x20)
lh x23, 8(x22)
lbu x25, 34(x24)
lhu x27, 45(x26)
lw x29, 9(x28)

sb x31, 3(x30)
sh x2, 23(x1)
sw x4, 89(x3)
```

5.2.3 Branches and control hazard test

The next step was the test of all kinds of branch instructions, both taken and not taken, to verify at the same time the correct behavior of the HDU in case of taken branch and subsequent stall.

Unconditional jumps First of all, the two unconditional jump instructions were tested, namely `jal` and `jalr`. The initial `addi` of the second case was inserted only to have a known value in register `x3`.

```
jal x2, 2

addi x3, x0, 5
nop
nop
jalr x4, x3, 3
```

In this case, the important things to notice are the correct evolution of the program counter after the jump (along with the right generation of the address and LSB masking) and the correct generation of the CU signals `jumpOrBranch` and `jalr`, as well as `stall_n` from the HDU, to disable the PC and the IF/ID pipe register.

Branches A similar procedure was used to test branches.

```
addi x1, x0, 9
addi x2, x0, 9
nop
nop
beq x1, x2, 16
```

5.2.4 Data hazards and forwarding

The last set of instructions was aimed at testing the correct behavior of the HDU and FWU under all conditions.
The first case is the forwarding of the previous ALU result:

```
li x1, -44
addi x2, x0, 56
sub x3, x1, x2
```

Figure 22 shows the result of the simulation. Here, the first operand of the `sub` instruction, `x1`, is taken from the MEM/WB pipe register, since it was set two instruction earlier. Therefore, in the clock cycle indicated by the gold cursor, `FWU_fwdA`

`= 1`. The second operand, `x2`, is forwarded from the previous ALU result in the EX/MEM pipe register, so `FWU_fwdB = 2`. Both the sel signals are coloured in pale blue, while the outputs of these multiplexers are in purple. They are set correctly even if the corresponding register has not been written yet. As a result, the alu output of the third instruction in the EX stage (`ALU_out`) is correct: $-44 - 56 = -100$.

Next, forwarding can also occur from the ALU result of two cycles earlier:

```
addi x13, x1, 253
sub x7, x5, x6
xor x3, x13, x2
```

Finally, forwarding between load and store was tested:

```
addi x1, x0, 45
nop
nop
lw x2, 16(x1)
sw x2, 32(x1)
```

Again, there's no need to stall the pipeline since the value loaded from the data memory is forwarded to its data input before it is actually written to register `x2`. This kind of forwarding is particularly useful when a data in a memory location has to be moved to another location. This operation can be done in just two clock cycle. The wave postscript of this execution is reported in figure 23.

The hazard detection was tested with a load followed by another using instruction:

```
lw t0, 0(x0)
addi x1, t0, 1
```

This is the only case a data hazard cannot be resolved by forwarding, as mentioned in section 3.1.1. The pipeline must be stalled. This is shown in figure 24: the `stall_n` signal is asserted and the program counter (`I_MEM_addr`) is prevented from incrementing to the next instruction, so that the second one is fetched again in the next cycle, when the value loaded from the memory is available, even before it is actually written to register `t0` (`x5`, at the bottom of the wave figure), thanks to the forwarding logic instantiated into the register file. This example shows the perfect collaboration between the FWU and the HDU, and how they manage to speed up the processor together. Good guys they are.

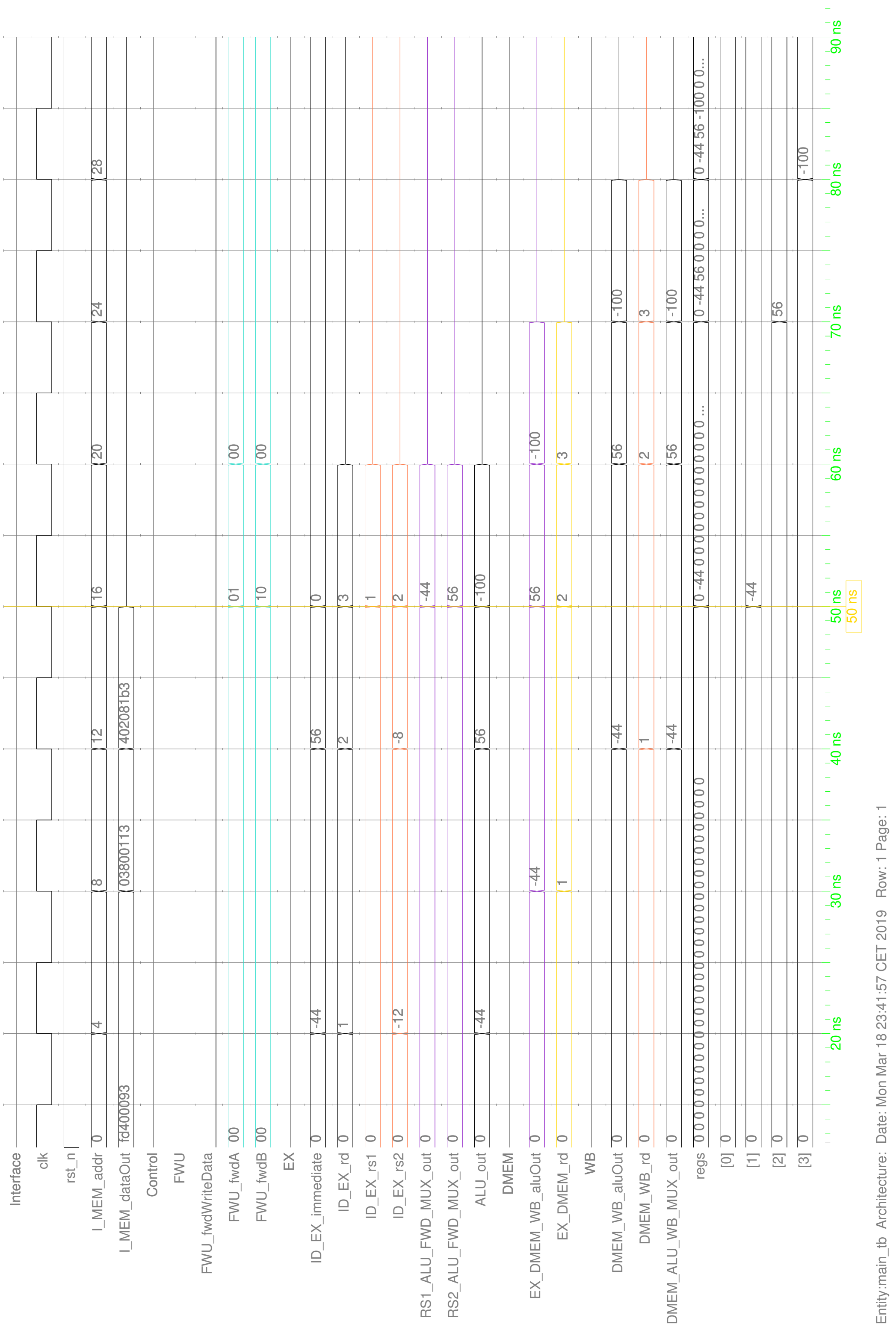


Figure 22: Forwarding: ALU instruction after ALU instruction

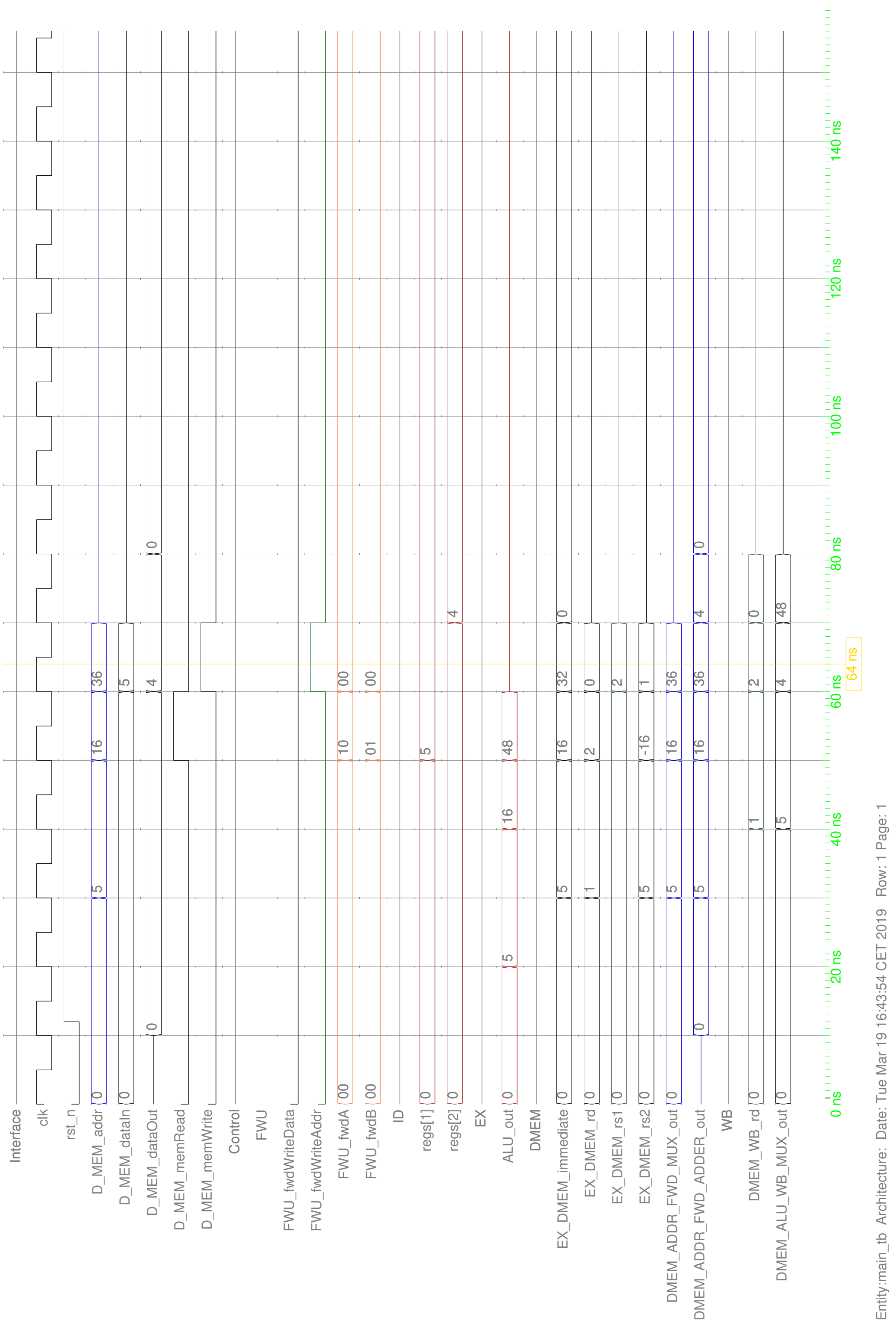


Figure 23: Forwarding: Store instruction after load instruction

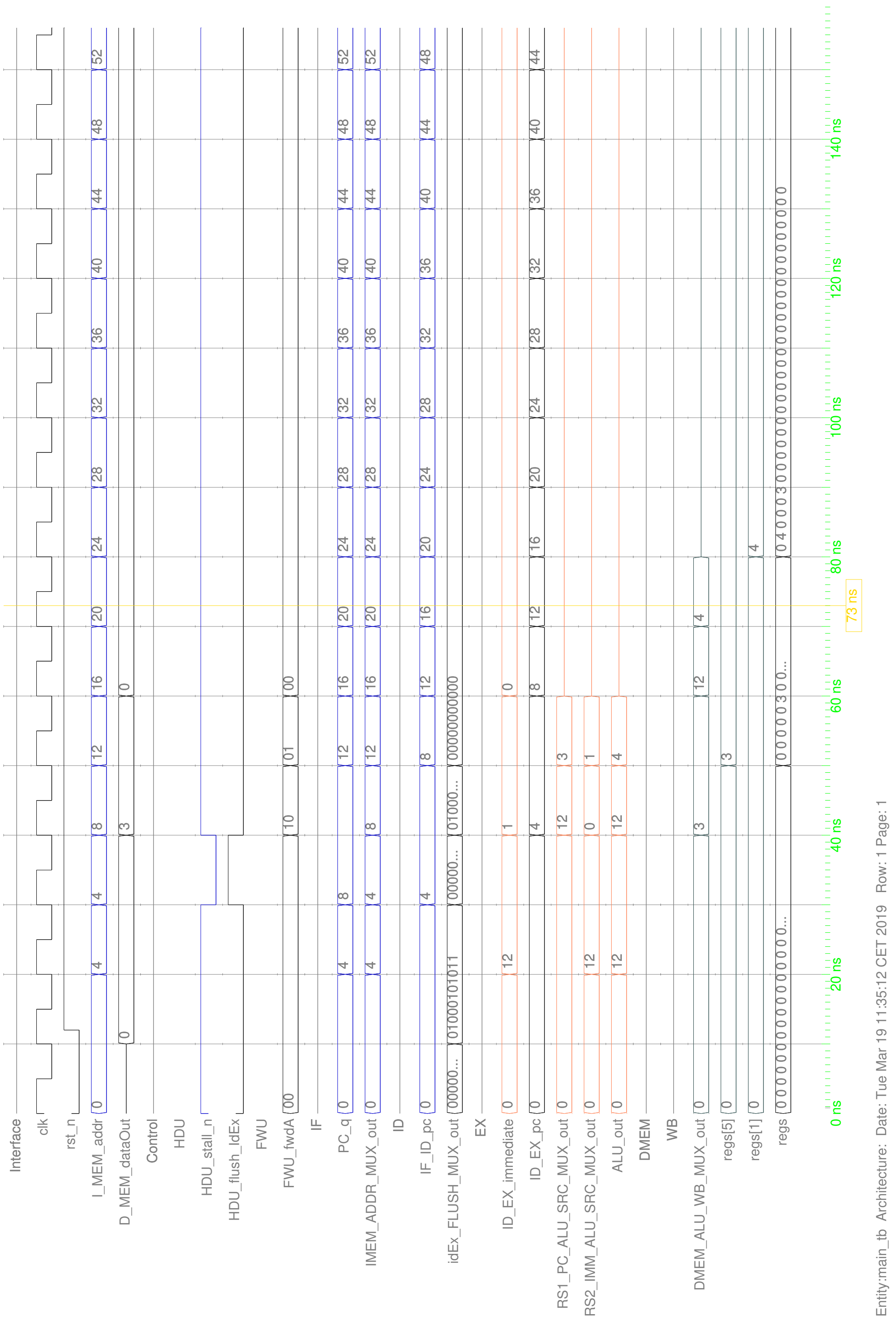
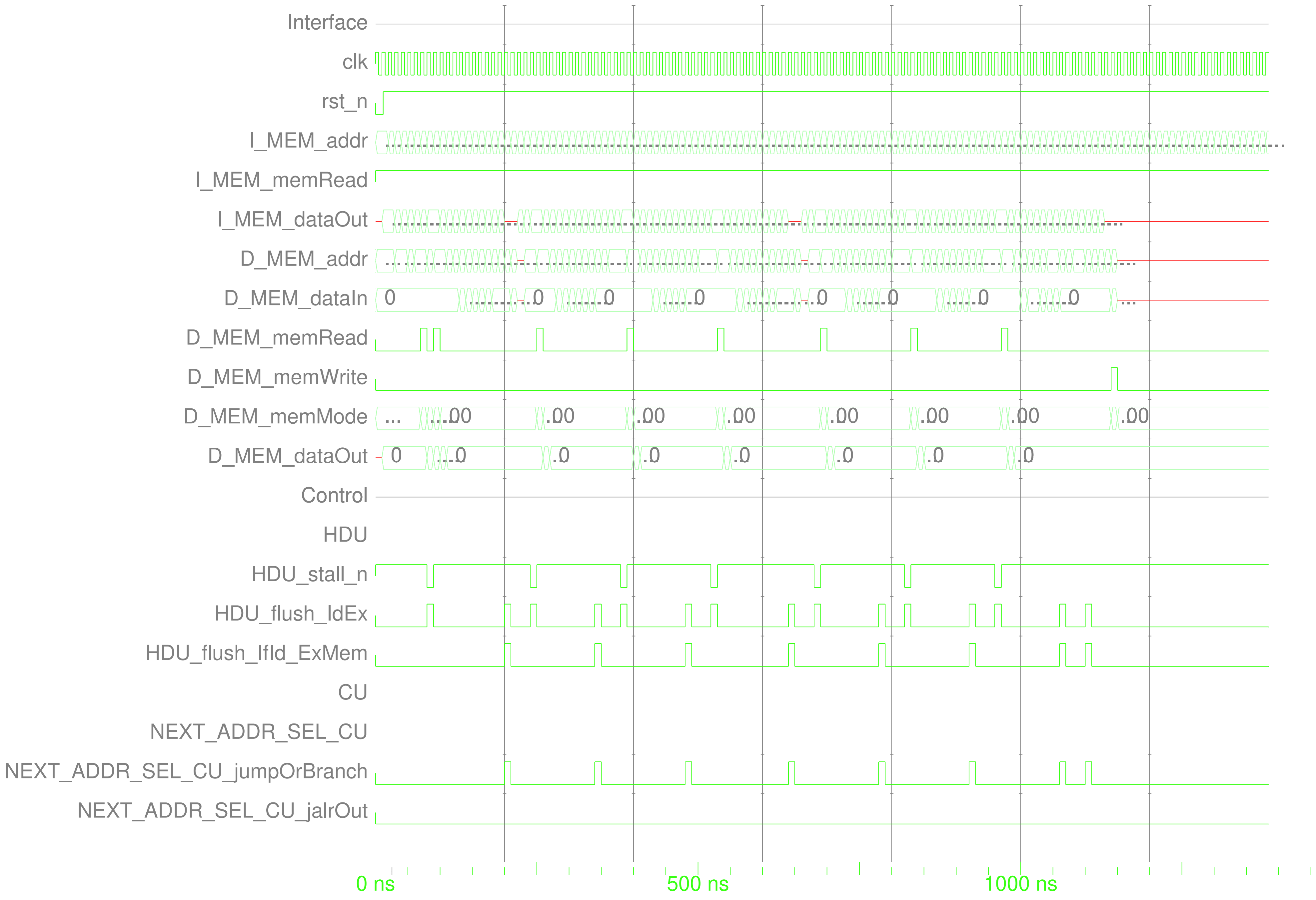


Figure 24: Forwarding: ALU instruction after load. The pipeline is stalled

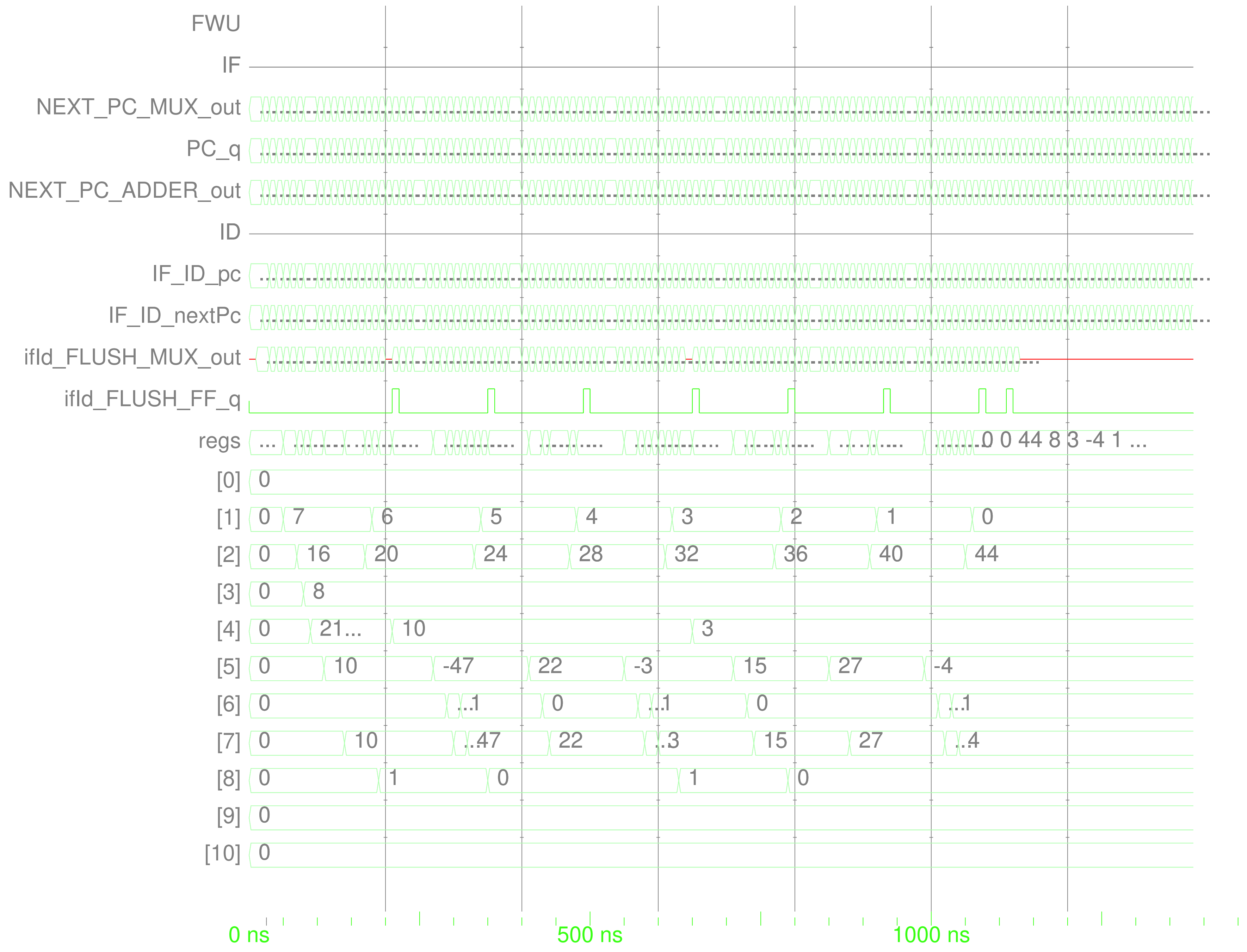
5.2.5 Complete program test

After having tested all single instructions and hazard scenario individually, a complete assembly program was tested, in order to verify the correctness of the design in a real application. The program in question (listed below) is the translation to RISC-V assembly of the test program provided for the laboratory experience on the MIPS processor, which computes the minimum absolute value among the elements of a vector.

```
#__start:
li x1, 7            # load x1 with 7 (number of elements)       0
li x2, 16           # put in x2 the address of v (16)           4
li x3, 8            # put in x3 the address of m (8)            8
lw x4, 0(x3)        # init x4 with max value (high)            12
#loop:
beq x1, x0, 48      # check all elements have been tested      16
lw x5, 0(x2)        # load new element in x5                   20
srai x6, x5, 31     # apply shift to get sign mask in x6       24
xor x7, x5, x6      # x7 = sign(x5)^x5 (invert x5 if negative)  28
andi x6, x6, 1      # x6 &= 1 (carry in)                       32
add x7, x7, x6      # x7 += x6 (add the carry in)              36
addi x2, x2, 4      # point to next element                    40
addi x1, x1, -1     # decrease x1 by 1                         44
slt x8, x7, x4      # x8 = (x7 < x4) ? 1 : 0                   48
beq x8, x0, -36     # next element                             52
add x4, x7, x0      # update min                               56
jal x0, -44         # next element                             60
#done:
sw x4, 0(x3)        # store the result                         64
```



Entity:main_tb Architecture: Date: Wed Mar 20 15:05:31 CET 2019   Row: 1 Page: 1



Entity:main_tb Architecture: Date: Wed Mar 20 15:05:31 CET 2019   Row: 1 Page: 2

5.3 Assembler

Since we decided to not support some of the RV32I instructions and we were not going to compile high level code, we decided to write our own custom assembler. So we came up with *com.py*. It is not a compiler, despite the name, nor a full RV32I assembler, and doesn't claim to be. The main reason for its name is the fact that it ends in *py*, and there you are the wordplay with its extension. Still, an assembler could be seen as a simple compiler for a low level language.

5.3.1 Features

Here is a little perspective on the main feature of *com.py*.

- Most of the instructions from RV32I are supported, excluding:
  - Environmental CALL (`ecall`) and BREAK (`ebreak`) instructions
  - Synch thread (`fence`) and synch instruction & data (`fence.i`) instructions
  - Status register manipulation instructions (`csr*`)
- Most pseudo-instruction are supported, excluded the ones that have to be translated to more than one instruction and those related to the unsupported base instructions.
- All the instructions and pseudo-instructions are processed and assembled making heavy use of python dictionaries. This greatly simplifies the code and most important keeps the assembler modular. Adding a new instruction is normally as easy as introducing a new `[key]: [value]` couple to the dictionaries.
- The output machine code can be provided both as ASCII binary file or ASCII hexadecimal file, so that they can be employed as source files for simulation.
- If requested, assembler can set-up a System Verilog file containing the simulation parameters, linking it to the assembled machine code. Then, a simulation script is launched in Modelsim.
- Registers can be addressed by their architectural names (`x[n]`) or by their ABI (Application Binary Interface) names (e.g. `ra` or `sp`).
- The offset field can be passed both as a standalone parameter (e.g `sw rs1, rs2, imm`) or as parenthesis prefix (e.g `sw rs2, imm(rs1)`).
- Comments are introduced by character `#` and are ignored.
- Very basic syntax error detection is provided.
- This is not meant to be a usage guide, so run `./com.py -h` for more help.

5.3.2 How it works

Figure 25 shows a qualitative flow chart of the assembler. After opening the input file, the script reads one line at a time and remove comments. Pseudo-instructions are then converted to base ones and processed, while the canonical format is retrieved:

```
instr_name param1, param2, [param3]
```

Then, all the instructions are processed the same way. The fields of the machine code that depends only on the type of instruction and the instruction name are processed first. After, the parameters like registers and immediate/offset fields are encoded based on the instruction type. Eventually, all the fields are printed on screen, joined together and appended to a ASCII binary output file with extension `.mc`. Since the assembler doesn't keep a copy of the entire code but processes one instruction at a time, memory usage is kept under control even when dealing with long pieces of assembly code.

As soon as an error is encountered processing one line (i.e. one instruction) the user is warned by a message on screen and the assembler exits with a specific error code. This makes it possible to use this script inside other bash or python scripts. Errors are handled by means of python exception handling `try:... except:...` construct, that interprets these exceptions based on the assembling context instead of throwing generic python interpreter errors to the user.

Here is an example of the output of the assembler when processing a piece of RISC-V assembly code that computes the Fibonacci sequence:

```
RV-MAGIC/common/src$ ./com.py ../../main/tb/assembly/fibo.asm y n

> ================================================================
>             2019-03-20 19:42:31 - Welcome to com.py
> ================================================================


> Assembling "../../main/tb/assembly/fibo.asm"...
>       1)    li x5, 7             =>   I-type: 000000000111  00000 000 00101 0010011
>       2)    li x6, 2             =>   I-type: 000000000010  00000 000 00110 0010011
>       3)    jal ra, 8            =>   J-type: 00000000100000000000    00001 1101111
>       4)    j 72                 =>   J-type: 00000100100000000000    00000 1101111
>       5)    bge x5, x6, 12       =>   B-type: 0000000 00110 00101 101 01100 1100011
>       6)    addi x7, x5, 0       =>   I-type: 000000000000  00101 000 00111 0010011
>       7)    jalr x0, 0(ra)       =>   I-type: 000000000000  00001 000 00000 1100111
>       8)    addi sp, sp, -12     =>   I-type: 111111110100  00010 000 00010 0010011
>       9)    sw x1, 0(sp)         =>   S-type: 0000000 00001 00010 010 00000 0100011
>       10)   sw x5, 4(sp)         =>   S-type: 0000000 00101 00010 010 00100 0100011
>       11)   addi x5, x5, -1      =>   I-type: 111111111111  00101 000 00101 0010011
>       12)   jal ra, -28          =>   J-type: 11111110010111111111    00001 1101111
>       13)   sw x7, 8(sp)         =>   S-type: 0000000 00111 00010 010 01000 0100011
>       14)   lw x5, 4(sp)         =>   I-type: 000000000100  00010 010 00101 0000011
>       15)   addi x5, x5, -2      =>   I-type: 111111111110  00101 000 00101 0010011
>       16)   jal ra, -44          =>   J-type: 11111101010111111111    00001 1101111
>       17)   lw x13, 8(sp)        =>   I-type: 000000001000  00010 010 01101 0000011
>       18)   add x7, x13, x7      =>   R-type: 0000000 00111 01101 000 00111 0110011
>       19)   lw ra, 0(sp)         =>   I-type: 000000000000  00010 010 00001 0000011
>       20)   addi sp, sp, 12      =>   I-type: 000000001100  00010 000 00010 0010011
>       21)   jalr x0, 0(ra)       =>   I-type: 000000000000  00001 000 00000 1100111

> BIN file successfully created:
      RV-MAGIC/main/tb/bin_mc/fibo.mc

> HEX file successfully created:
      RV-MAGIC/main/tb/hex_mc/fibo.riscv
```

Eventually, if no error was encountered, a hexadecimal file with extension `.riscv` is generated starting from the binary one. If requested, the script proceeds linking the assembled code in a System Verilog test-bench configutation file and launches the simulation in Modelsim, running a `.tcl` script with the necessary commands to compile and simulate the entire design. Since this configuration file can also be used for other purposes, *com.py* keeps a copy of the old version and restores it when Modelsim is quit.

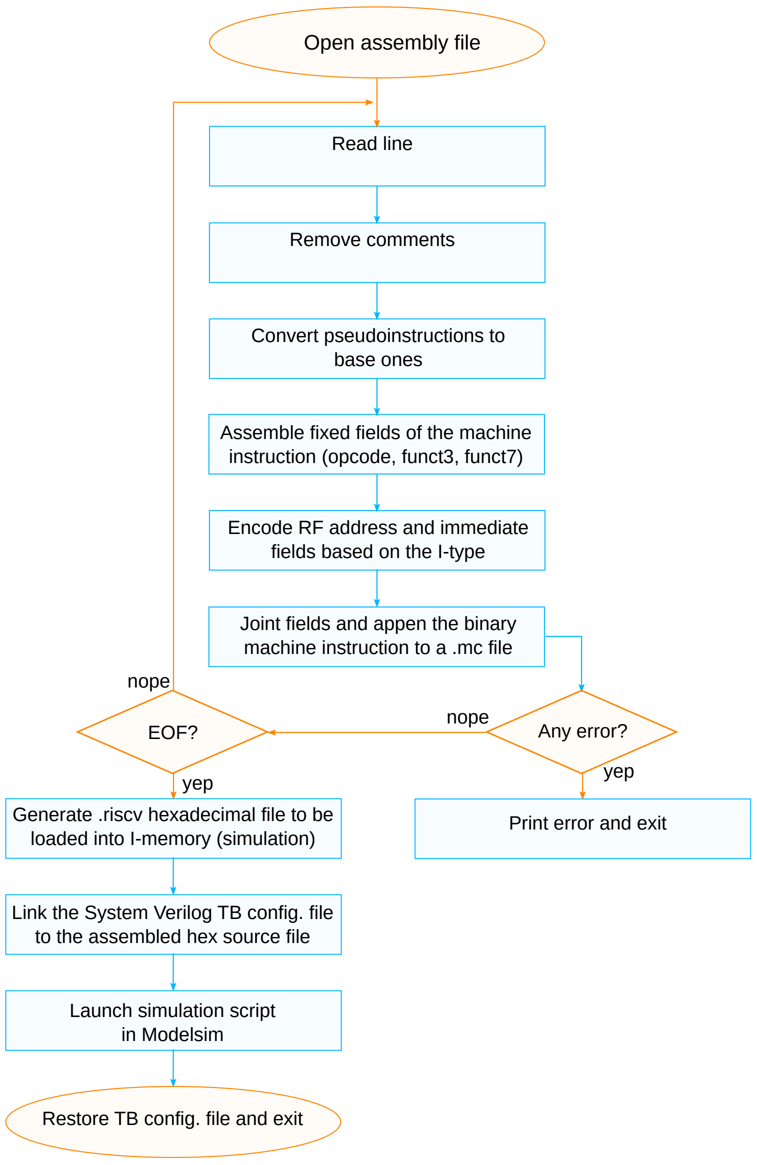

Open assembly file
Read line
Remove comments
Convert pseudoinstructions to base ones
Assemble fixed fields of the machine instruction (opcode, funct3, funct7)
Encode RF address and immediate fields based on the I-type
Joint fields and appen the binary machine instruction to a .mc file
nope
EOF?
nope
Any error?
yep
yep
Generate .riscv hexadecimal file to be loaded into I-memory (simulation)
Print error and exit
Link the System Verilog TB config. file to the assembled hex source file
Launch simulation script in Modelsim
Restore TB config. file and exit

Figure 25: *com.py* flow chart