VHDL design of a simple RISC-V architecture

Mirco Tollardo 217918

a.y. 2024-2025

Contents

1 Introduction **1**

2 Building an RV32I datapath **2**
Basic Instruction Fetch . . . 2
Basic description . . . 2
Simulation . . . 3
Basic Instruction Decode . . . 3
Instruction formats . . . 4
Decoding . . . 5
Register File . . . 6
Final concept for the ID and simulation . . . 6
Basic Instruction Execute . . . 7
Simulation . . . 8
Data Memory, Write Back and Structural Hazards . . . 9
Data Memory . . . 9
Write Back . . . 9
Simulation and final consideration . . . 10

3 Developing the Architecture **11**
Pipeline . . . 11
Solving data hazards . . . 12
Managing control hazards . . . 14
Simulation and Testing . . . 15

4 Conclusions **17**

5 Appendices **18**
Appendix A: Instruction Tables . . . 18
Appendix B: Code segments of unpipelined datapath . . . 19
Instruction Fetch [RV32I] . . . 19
Instruction Execute [RV32I] . . . 27
Data Memory [RV32I] . . . 32
WriteBack [RV32I] . . . 33
Unpipelined Datapath Simulation . . . 33
Appendix C: Code segments of the finalized datapath . . . 37
Final Instruction Fetch entity description . . . 37
Final Instruction Decode entity description . . . 37
Final Instruction Execute entity description . . . 39
Final Data Memory stage entity description . . . 40
IF/ID register . . . 41
ID/IE register . . . 42
IE/DM register . . . 43
DM/WB register . . . 44
Pipelined architecture (Structural definition) . . . 45
Appendix D: Testing Block for Nexys 4 . . . 51
Reading from register file and DM . . . 51

Introduction

This document is going to describe a basic RISC-V datapath written in VHDL, starting from a simple non-pipelined architecture just to implement more complex features and instructions with a bottom-up approach. When talking about RISC-V it is necessary to distinguish the two faces of the ISA:

- *Unprivileged ISA*: that defines the user-level instruction types, which mainly compose a program to be executed.
- *Privileged ISA*: which defines exception management and operation modes (supervisor, hypervisor and user mode), interrupt handling and multiple other functionalities for operating system support.

The project will focus on partially implementing the base user-level instructions, defined in the unpriviledged ISA, defining at each stage of the execution what information in an instruction are used and how they affect the state of the sequential network, helping the explanation with behavioral simulations, and building the project in a way that can be later expanded with other features. Thus, the first stage of this project will focus on implementing a datapath that executes some commands included in the User Instruction Set, in particular, for the first part of the project, the 32-bit Base Integer Instruction Set (RV32I). The report will be divided into chapters for each of the following 5 stages of the CPU:

- **Instruction Fetch**
- **Instruction Decode**
- **Instruction Execute**
- **Data Memory**
- **Writeback**

The completion of the first rudimental architecture, able to execute one instruction per clock cycle, has to be then optimized by introducting registers between each block, and thus achieving a *pipelined architecture*, adding more complexity overall yet incrementing the performance of the circuit.
This addition although does not come without problems, a pipelined architecture introduces some inconsistencies that need to be addressed in order to have the architecture to remain fully-functional. In particular:

- **Data Hazards**: Subsequent instructions that access the same registers, for both reading and writing, in different stages of the pipeline, need to have the most recent data available without having to wait for each previous instruction to complete its cycle of execution.
- **Control Hazards**: Pipelined architectures need to carefully manage jumps and, if needed, stop every other partial execution of any instruction that comes next, since there is no certainty that the flow of the program will continue in the same way as before the jump.

With everything fixed, the five stages can finally operate concurrently and thus actually benefitting from having a pipeline.
At the very end, having a synthetizable code does not mean that the code will follow the behavioral simulation, so the code will be implemented on a Nexys 4 DDR with an additional expansion for the core to interact with the 7-segments display and switches to actually see the content of the Data Memory and Register File, just to confirm that everything i working properly.
With this being said, the project will follow said procedure, while making available the code in Appendix B for the unpipelined datapath, Appendix C for the pipelined one, and Appendix D for the extra blocks for testing purposes.

Building an RV32I datapath

Basic Instruction Fetch

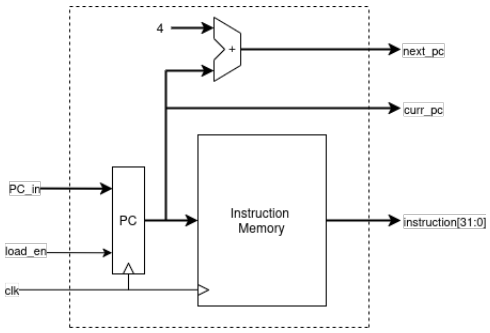


Figure 2.1: Basic instruction fetch block diagram

Basic description

This stage of the datapath fetches the instruction to be executed, which is stored in the Instruction Memory. This memory can be implemented using Vivado's Block Memory Generator IP, configured as a single-port RAM with a 32-bit read-write width. The memory size should be addressable using at least 12 bits, because the AUIPC instruction must be able to load absolute addresses that are shifted by that amount into the Program Counter (PC). The Program Counter is a register that holds the address of the current instruction and, during program execution, the latter is incremented at each rising edge of the clock; Since memory is byte-addressable and each instruction is 4 bytes long, the PC is incremented by 4 at each step. In this project, however, the Instruction Memory is simulated using a RAM with 32-bit words per address; While the architecture should still work with bytes, the increment is going to remain 4, but the two least significant bits of the address shall be unused from now on. To handle jumps or branches in the program, the PC may accept another value as input from a different stage along the datapath, allowing it to be loaded with a new address after such instructions are executed. In VHDL, a register can be synthesized by declaring a clock-dependent (and in this case also *load_en* dependent) *signal*, hence the PC will be implemented with this method. A VHDL entity that describes this stage can be defined as follows:

```
entity instr_fetch is
 port (
  clk        : in std_logic;
  pc_load_en : in std_logic;
  pc_in      : in std_logic_vector(11 downto 0);

  next_pc    : out std_logic_vector(11 downto 0);
  curr_pc    : out std_logic_vector(11 downto 0);
  instr      : out std_logic_vector(31 downto 0)
  );
end instr_fetch;
```

With the previously specified settings, the IP used for the Instruction Memory will generate a component that can be instantiated in the architecture as follows:

```
component instruction_memory
 port(
  clka  : in std_logic;
  wea   : in std_logic;
  addra : in std_logic_vector(9 downto 0);
  dina  : in std_logic_vector(31 downto 0);
  douta : out std_logic_vector(31 downto 0));
end component;
```

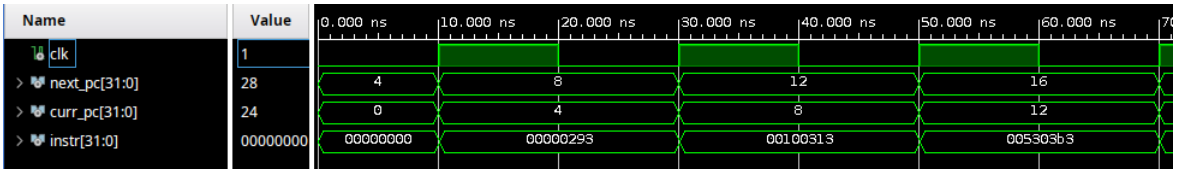
The inputs *wea* (i.e. the write-enabling input) and *dina* (data input) will be pulled low since there is no need for the moment to write inside the memory. Since the PC has to act as a pointer for the instruction, its output will be routed to the *addra* input, excluding the first two least significant bits in order to proceed 32 bits (the length of a standard instruction) at a time. With this being said, the only thing that is left to do is a clock-sensitive process that increments the PC or loads another address from the outside if the load-enable input of its register is pulled high. By implementing the register as a *signal* named *pc_reg*, the resulting VHDL script for the architecture is shown in Source Code 5.1.

Simulation

A testbench for this simulation will require the PC to be incremented by 4 so that every location in the memory is addressed, but in order to actually see a stream of instructions at the output a .coe file for the initialization of the instruction memory is required:

```
memory_initialization_radix=16;
memory_initialization_vector=
00000293,      --> addi x5, x0, 0
00100313,      --> addi x6, x0, 1
005303B3;      --> add x7, x6, x5
```

A .coe file requires the hexadecimal dump of the program, in this given example each instruction has its assembly equivalent on the right.
For the testbench, a simple clock can be simulated with a process while a small register can be simulated for storing *next_pc* and re-routing its value on *pc_in*, hence the testbench, shown in Code 5.2 gives the resulting behavior:



The results state that *curr_pc*, in fact, acts as a register, refreshed as the clock's rising edge approaches, and the flow of instructions are the same as the ones stated in the *.coe* file.

Basic Instruction Decode

After the instruction is returned from the fetching stage, it must be decoded in order to select the right action to be performed in the next step of the datapath. RISC-V ISA provides instructions for registers to operate with other registers, constants, Data Memory or the PC itself and there is the necessity to distinguish what parts of the instruction make up the operands of the execution and where the resulting information has to be stored. This action is done by the Instruction Decode, which selects what to use as operands through the multiplexers located at the upcoming stage, while also recognizing and subsequently giving more information about the instruction to the ALU. There are four core instruction formats that encode said informations:

- **R-type**: Register to register instructions. These instructions are expected to use two registers as sorce for data to be elaborated and then store the result of the operation in a destination register.

  ```
  add rd, rs1, rs2        ; rd = rs1 + rs2
  ```

  This instruction has to select the two source registers as operands in the ALU, hence two boolean outputs *a_sel* and *b_sel* will be pulled high to indicate to the ALU that the registers pointed by the values of rs1 (bits 19 down to 15 of the 32 bits instruction) and rs2 (bits 24 down to 20).

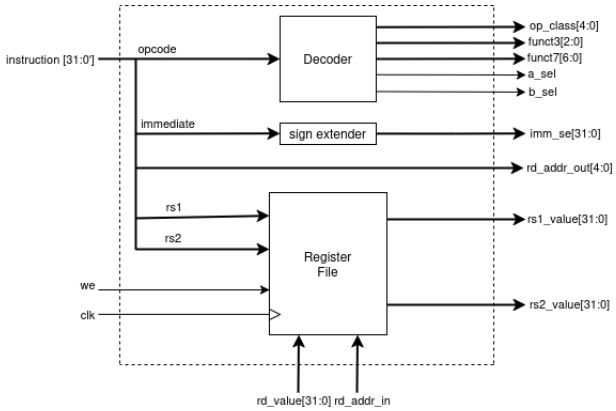


Figure 2.2: A block diagram for the instruction decoding stage

- **I-type**: These ones have a constant value, i.e. an immediate, encapsulated in the code, that becomes be part of the operation. Some examples are:

```
addi rd, rs1, imm          ; rd = rs1 + imm
lw rd, imm(rs1)            ; rd = Mem[rs1 + imm]
```

  The value stored in rs1 is encapsulated in bits 19 down to 15 and the immediate value from bit 31 to 20. *a_sel* has to be set high while *b_sel* is set to low.

- **S-type**: For store operations, where the content of the register rs2 is stored inside the memory address in the data memory pointed by rs2, possibly summed to an immediate value (little endian encoded, bits 31 to 25 for the 7 most significant bits and bits 11 to 7 for the least significant ones).

```
sw rs2, imm(rs1)           ; Mem[rs1 + imm] = rs2
```

  In this case *a_sel* is set to high while *b_sel* stays low in order for the ALU to sum rs1 and the immediate value.

- **U-type**: Loading immediates inside a register, like for example LUI (Load Upper Immediate, also known as the pseudo-instruction li). In this case *b_sel* is set to be low while *a_sel* can be either high or low, as the ALU can be programmed to ignore the first operand if necessary.

```
lui rd, imm                ; rd = imm
```

Instruction formats

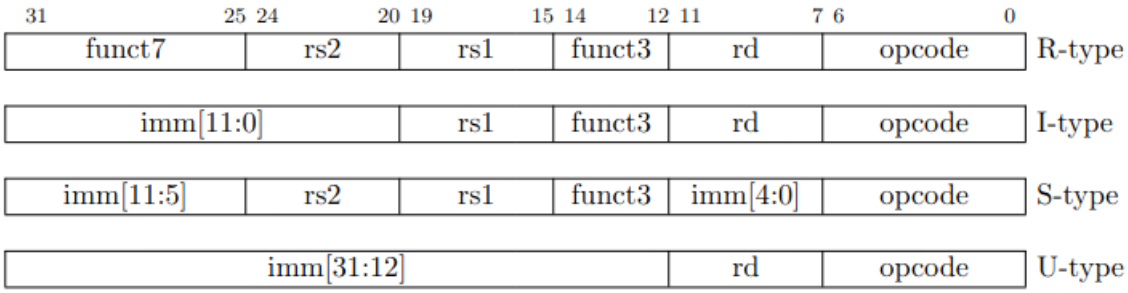


Figure 2.3: Base instruction format [1]

These instructions share the same positions for the operands and the destination registers, which simplifies the structure of the code (Figure 2.3). Instructions that operate with the PC and conditional instructions such as branches share similar structure with respectively U-type and S-type instructions, in the ISA they are defined as:

- **B-type** or SB-type: Just as in store operations, the rs1 and rs2 addresses are coded in the same position, with the difference that the immediate value has its bits coded in different positions (imm[12] on bit 31, imm[11] on bit 7, imm[10:5] from bit 30 to 25, imm[4:1] from bit 11 to 8). For this type of operation there is no need to change *a_sel* and *b_sel*, since, as it will be shown later, the logic unit just operates between registers.

- **J-type** or UJ-type: Jump operations sums an immediate to the PC and stores the next instruction's PC value. These are coded just as unsigned operations, with the difference being the immediate's bits having different positions (imm[20] on position 31, imm[19:12] from position 19 to 12, imm[11] on bit 20 and imm[10:1] from position 30 to 21).

With these definitions, the base ISA extends to this: Knowing that the next block can be programmed to use certain

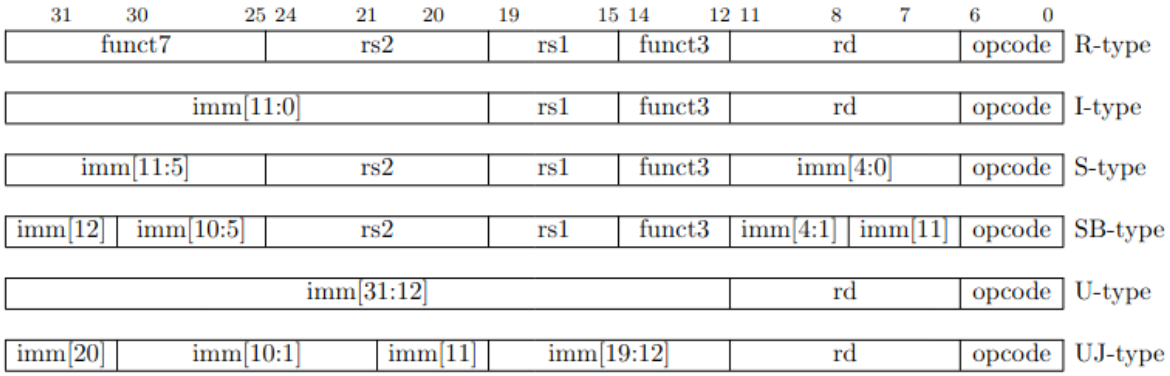


Figure 2.4: Full 32-bit instruction format [1]

parts of the instruction in a certain way, the VHDL code for the ID architecture can be simplified by giving out all the sections of the instruction and letting the Instruction Execute select what specific parts of it are needed.

Decoding

Reading opcodes for operand selection

The encoding type of an instruction is not sufficient to detect what operation is actually performed. For example, *LW* and *ADDI* are both I-type instructions but the latter has its result written somewhere in the Register File instead of the Data Memory. For this reason it is necessary to assess some information segments inside the instruction itself. By looking at Figure 2.4 it is noticeable that there are at most three elements that are descriptive of the content of the instruction and its behavior:

- **opcode**: Unique and present in every instruction, composed by the first 7 bits (starting from the LSB, considering a Little Endian encoding).
- **funct3**: Present in all instructions but U and J -types, made by bit 12 up to 14, necessary for the ALU to select the arithmetic or logic operation to be executed.
- **funct7**: Only for R-type instructions, in the RV32I ISA only one of the seven bits is used.

It's obvious that *funct3* and *funct7* do not give information about the identity of the instruction, it is the *opcode* that differentiates the operation and therefore the one that has to be taken into account. The Instruction Set manual [described by Waterman et al. (2016) [1] page 53 Table 9.1] gives the general opcode map that can help to discriminate instructions:

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

Figure 2.5: RISC-V base opcode map, RISC-V user-level ISA version 2.1 [1]

The opcode map gives the idea that a decoder can be described with a nested case statement, with an external case structure driven by the last two bits of the opcode, from which is possible to extrapolate what are the operands to be passed to the ALU or Logic Unit, being it the PC, two registers or an immediate and a register.

Instruction classification

The last and most delicate problem that arises when decoding instructions is that they operate in different ways even if they share the same encoding type, for example, ADDI (I-type) will write the result of the arithmetic operation in a destination register, while LW (I-type) will have as destination register a memory location pointed by the result of another arithmetic operation. Also, the interaction with the PC can widely differ based off the circumstances, hence there is the additional requirement to define in advance a classification method so that the Write Back stage can select what can has to be passed to the Register File and select the next value for the Program Counter. There are three possible cases in which the Write Back accesses the register file: Jumps (since they write back the return address), Loads and arithmetic/logic operations; In addition, the PC can have a new value loaded if, during a branch intruction, the Logic Unit returns a logic high, or during a jump instruction. Otherwise, PC+4 is always returned as next value.
The solution could be encoding these informations for the next stages as a 6 bit vector, with a one-hot encoding for Arithmetic-Logic Operations, Jump, Load and Store,since store operations can be added as an additional bit that can be used as a write-enable for the Data Memory.
Some instructions though cannot follow this pattern, for example, *Load Upper Immediate* (U-type) takes the 32 bit immediate, with the bits 31 downto 12 being the ones the user can manipulate and the remaining 11 downto 0 being filled with zeros, and loads it to a destination register; No operation other than returning a constant is being made yet it is not a load operation nor an arithmetic or logic operation. The same thing happens for AUIPC with the difference being that the resulting immediate is alse added to the PC.
To solve this problem the 6 vector could instead not be one-hot encoded and possibly include these instructions that, at first glance, could actually seem like a combination of multiple types, for example, if operations are visualized as "000010" and load operations as "000100", then LUI can be encoded as "000110". This could actually be a fine solution, opcode map Figure 2.5 might tell otherwise if the purpose of this project was its expansion to include floating-point operations and multiplications, yet they could actually be encoded as simple operations, letting the ALU manage them in function of *funct7* and *funct3*, while system instruction can be easily encoded as instruction class "000000" or similarly as other instruction if necessary.

Register File

The Register File (Source Code 5.5) is the last block needed to complete the stage. In these architectures is made by 32 register, each one with a length of 32 bits which greatly simplifies its imlementation in VHDL by declaring it as an array of logic vectors. The register as VHDL entity can be defined as a three port RAM that allows read and write operations in the same clock cycle without the need of a correct timing, as it would happen in dual-port RAMs; Write operations will be performed at the rising edge of the clock if a write-enabling input is set to logic high, while read operation can be asynchronous to the clock. In addition, every write operation to the x0 register will be ignored as by RISC-V specification.

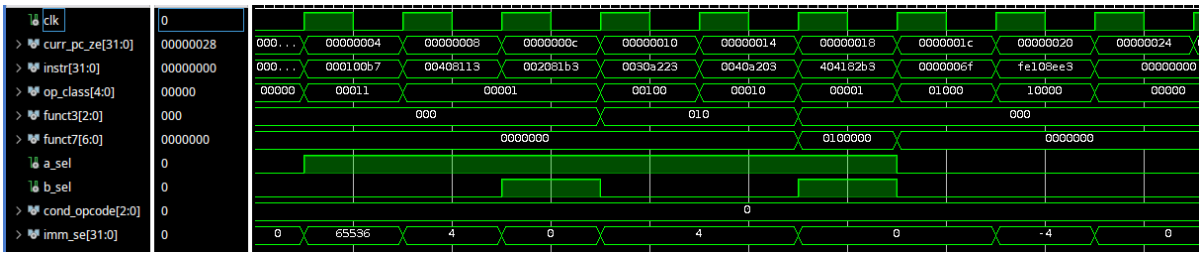
Final concept for the ID and simulation

With the current initialization files for the IM, only a small amount of instructions are going to be tested, for this reason the new file should contain each existing tipology, possibly testing the ones that use *funct7* while also implementing jumps and branches, possibly with a loop so that the next stages can be easily tested for inconsistencies or error that may stop the code from running.The new initialization could become:

```
memory_initialization_radix = 16;
memory_initialization_vector =
000100b7,   --> lui x1, 16
00408113,   --> addi x2, x1, 4
002081b3,   --> add x3, x1, x2
0030a223,   --> sw x3, 4(x1)
0040a203,   --> lw x4, 4(x1)
404182b3,   --> sub x5, x3, x4
0000006f,   --> jal x0, 0
fe108ee3;   --> beq x1, x1, -8
```

The finalized concept of the Instruction decode, which follows the block diagram at Figure 2.2, introduces the Register File and a specialized block for decoding, taking as inputs all the IF's outputs (including pc_out and next_pc for sign extension). The most complex part of this stage, the decoder, is implemented as specified earlier, with two nested case statements, with the outer statement using the two most significant bits of the opcode. A snippet is provided at Source Code 5.4. By using the same testbench as for the IF and istantiating the ID (Notes

Code 5.6), by routing the selected instruction and the PC's values, the code is expected to fetch the instruction 0x000100b7, decode it and classify it as an operation with an immediate as second operand, hence with *a_sel*=1, *b_sel*=0 and *op_class*=000110, in fact:



One particular oservation that could be made is that the Decoder, at any point where the IM returns an empty instruction, i.e. with its hexadecimal value at 0, is going to report it as a LOAD instruction (*op_class* = "000100") if the first two bits of the instruction are not read. This id due to the fact that if the first two bits (that should always be "11" in RV32) are ignored, the decoder will select the firt case as true, thus performing a load operation. For this reason, even if they can be ignored, the caase statement will take them into account to avoid problems with empty IM addresses.

Basic Instruction Execute

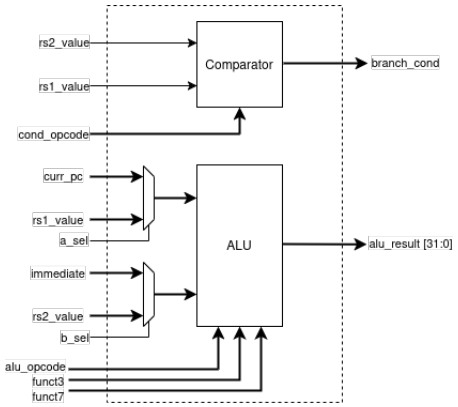


Figure 2.6: A block diagram for the execution stage

With the information previously decoded from the instruction, the execute stage is responsible for performing the required computation based on the instruction type. This involves selecting the correct operation and applying it to the values stored in the source registers. In a VHDL implementation, the execution stage can be designed as the union of two main blocks:

- **Comparator**: This block evaluates a condition between two 32-bit inputs and returns a boolean result based on the comparison criterion defined by the funct3 field. It is specifically used for branch-type instructions, determining whether a jump in the control flow should occur depending on the result of the comparison.
- **Arithmetic-Logic Unit**: This block performs all arithmetic and logical operations, including shift operations. When implementing the ALU, it is important to account for both signed and unsigned computations. To handle this correctly, the inputs to each multiplexer are initially cast as unsigned, ensuring consistency with immediate values, which are inherently unsigned, and then cast appropriately based on the operation being performed.

Starting with the ALU, the most complex of them, the first step before implementing its VHDL description is to clearly identify the specific requirements to fetch for each core instruction. This involves selecting the operation the necessary fields to determine the correct arithmetic or logical function and select the correct operators, whether they are represented by registers (including the PC) or immediates. As said earlier, some of them can be distinguished for both the presence of *funct3* (bit 14 to 12) and *funct7*, when dealing with R-type encoding, or just *funct3*. As specified in the base RV32I manual, the univocal codes for the basic core instructions are listed in table 5.1. It is

noticeable that, uniquely in RV32I architectures, the actual purpose of *funct7* is to switch between addition and substraction, since they are encoded as "000", or between logic and arithmetic right shift (i.e SRA and SRL as shown in the table). A simple solution could be editing the behavior of the decoder to onlytake the sixth bit of *funct7* into account and have all subtractions encoded as "010", as it would happen for SLT, and similarly for SRL and SRA. This solution would remove the necessity for *funct7* and shrink the necessary opcode for the ALU to 3 bits. However, that portion of the instruction, seemingly unnecessary for RV32I, comes useful when implementing floating-point instructions and multiplications, hence the most flexible solution, however complex, is to make the decoder return *funct7* and have the ALU taking it into account when switch between operations. With this in mind now the decoder can be edited to return *funct7* when treating opcodes mapped as OP, and some instances of OP-IMM when treating shifts.With this being said, it is decided that the ALU can switch operation by taking into account, in order of importance: the instruction class, *funct3* and, to maintain flexibility for future modifications, *funct7*. If the ALU can be defined as a big case statement as for the decoder, Table 5.1 may help to simplify the code, in fact some observations can be already made:

1. Many instructions expect the ALU to perform an addition, thus if the architecture of the block is written as a case statement, *the addition can be performed if no other case applies*.

2. All load and store operations require *funct3* to be used later during WriteBack stage because they may operate with less than 32 bits, thus it can be useful to *pass it down to the latest stages of the datapath*.

3. Shift operations using immediates are encoded as I-type but they just need five bits indicated as *shamt* (shift amount), for shifting more than 32-bits would not make any sense. While this does not affect SLLI and SRLI, the presence of *funct7* rises a problem with SRAI. *Shifts with immediates need to filter part of the second operand*.

The code for the ALU will thus check the operation class, funct3 and funct7 in the latter sequence. Source code 5.8 shows a VHDL implementation that follows said constraints.
Branches only require a comparison and just need a single signal to indicate whether or not a jump can happen. From the RV32I instruction table, the remaining core instructions to implement are indicated in Table 5.1 as BRANCH class instrutions. As it has been for the ALU, the table leads us to a case statement, much simpler than before due to just having to look at *funct3*; A much more self-explanatory architecture can be seen in Source Code 5.9.
Finally, the IE (Shown as Source Code 5.7, much easier to visualize in Figure 2.6) can be "assembled" by routing inputs and outputs of each block, and by also coding multiplexers for the operand selection:

```
alu_mux_a   <= rs1 when a_sel = '1' else curr_pc;
alu_mux_b   <= rs2 when b_sel = '1' else imm_se;
```

Simulation

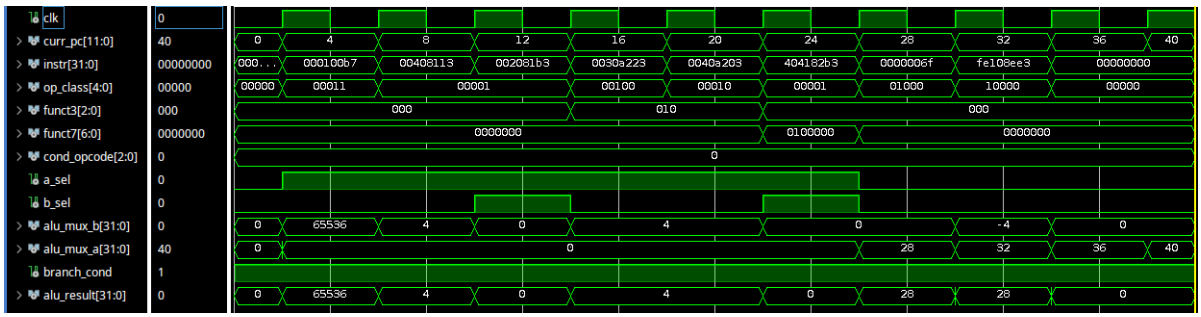


Figure 2.7: Complete IE simulation

The testbench, following the step-by-step assembly of the whole datapath proposed in the ID section, can be built using a copy of the earlier code with the IE as an addition (Source Code 5.10).
Now when running a behavioral simulation the following events are expected to happen to be sure that the design is working properly: the first instruction is going to be encoded with *op_class* = "000110", while setting *a_sel* and pulling down *b_sel*, leaving *alu_result* = 65536, because LUI operation imply that the immediate operand is shifted by 16 bits to the left, thus multiplying it by 2^{15}; The second one should sum the previous result with 4 and the folowing addition should return a 0, since the accessed register cannot yet be written; The following load, store and jump instructions should be performing additions, easy to notice from the ALU's output, which in case of SW and LW, should just return the immediate 4 summed to the registers that are still limited to store just zeroes,

while jumps and branches sum the current PC's value to their orresponding immediate value (in Figure 2.7).The subtraction is easily noticed by the fact that it represent the only one that changes *funct7* and the last one should have the comparator set its only output:

Data Memory, Write Back and Structural Hazards

Data Memory

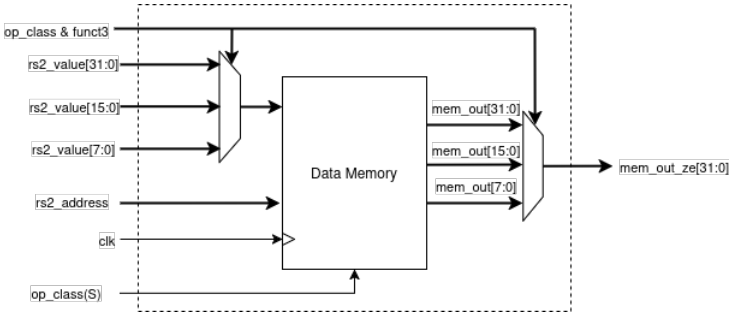


Figure 2.8: DM interface block diagram

The final pieces that are left to be designed are the one that manage the Data Memory, the one which decides when to write and where, and the stage which, depending on the operation type, selects what to return as new value for the PC and what to store in the destination register, i.e the Write Back.
The Data Memory will instantiated as a 16KB single port RAM and separated from the rest, forming a 5-stage datapath when it is going to be pipelined. In 4-stages architecture it would be instead included in the same block, reducing the overall size of the processor, reducing power consumption, due to the presence of one less register, but at the cost of a reduced throughput.
The Table 5.1 indicates that there are many ways to perform L/S instruction and not only 32-bit vectors are handled. In fact some of them could manage just bytes or half-words, leaving some other options to be managed when executing these instructions. This discrimination cannot be performed by the ALU nor the Decoder, but it is easy to implement as interface for the Data Memory. To be more specific, another case statement can be deployed to select filter the input and output of and from the memory, leaving the discrimination to both *op_class*, to see if an L/S operation is being performed, and *funct3* to classify if the latter is working with bytes, half-words or words.

Write Back

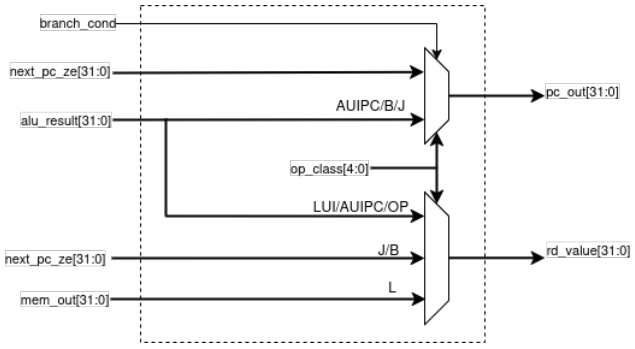


Figure 2.9: Write Back block diagram

The final step of execution, where the destination register and the program counter are written, needs a selector to decide what has to be overwritten and where; In the case of the PC two options can be identified: PC + 4 or the result from the ALU, which happens in case of a jump, a true condition from a branch or AUIPC. The destination register instead can be: the output of the ALU for classes OP, LUI, UIPC, PC + 4 for jumps and branches (when a logic high is returned, otherwise a high impedance output can be returned) or the output from the data memory. Source Code 5.12 shows again a case statement enclosed in an input-sensitive process, that is, a simple combinatory network.

The reading and writing of the register file in the same clock cycle might seem impossible, since *rs1* would have to be selected before the execution happens, meaning it

Simulation and final consideration

With all the code prepared, the final simulation of the unpipelined datapath can be performed; Leaving the load-enable of the PC high, for there is no need to stall the execution for now, it is noticeable that the overall behavior corresponds to what is expected from it:

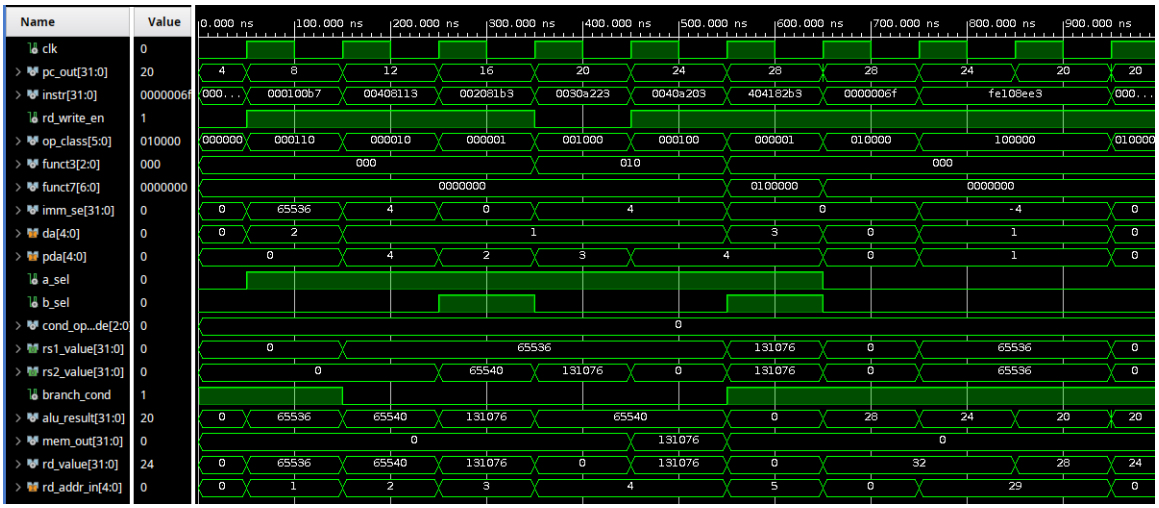


Figure 2.10: Complete simulation of the unpipelined datapath

Jumps and branches seem to be able to actively change the PC, in fact, the branch returns the value of PC where the sub instruction is located, meaning that if the simulation were to run infinitely, the program would loop between PC=28 and PC=32.
Also it is noticeable that L/S operations can effectively interact with the memory, although not each one of them has been tested, and other instruction can successfully read and write from the register file.

Developing the Architecture

Pipeline

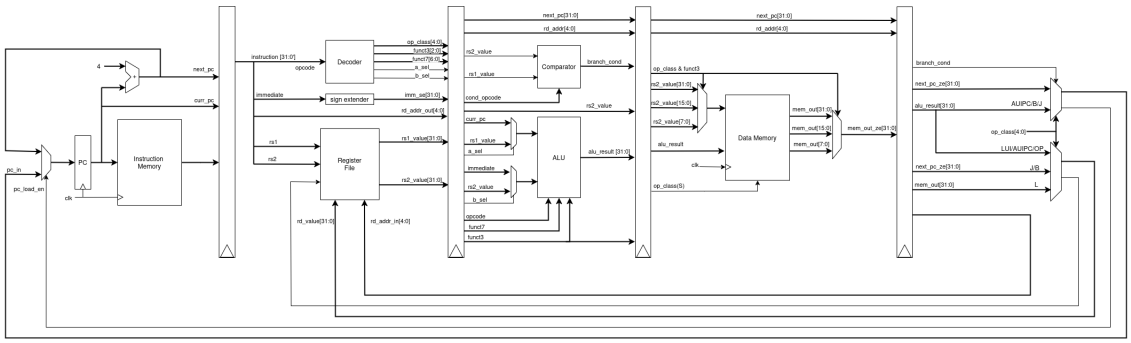


Figure 3.1: An example of pipelined RISC-V architecture

As mentioned earlier, having the processes sensitive to the rising edge of the clock, and having it update output signals after the end is the VHDL equivalent to registers, for example, supposing an input *pc_in* and an output *pc_out*:

```
architecture Behavioral of Example is
    signal pc_reg : std_logic_vector(11 downto 0);
begin
process(clk)
    begin
        if rising_edge(clk) then
            pc_reg <= pc_in;
        end if;
    end process;
    pc_out  <= pc_reg;
end Behavioral;
```

A process in VHDL executes a routine whenever any signal in the sensitivity list changes, meaning that whatever the routine is, its output is not available until the sensitive inputs change. In the example above *pc_reg* changes only when a rising edge of the clock approaches, thus keeping the output still for the whole clock cycle; The same concept can be applied to all the blocks that compose the architecture, by having all processes clock-sensitive, some kind of pipeline can be naturally implemented.
This method, however convenient, does not take into account the signals that are used in multiple process, for example, if *funct3* is passed directly to the DM stage, with *op_class* commanded by a different instruction, the DM could wrongly load or store the result of the original operation.
Both for this reason and for not having to change the whole code and thus maintaining clarity over the project's code, the inter-stage registers will be programmed as separate blocks in VHDL. Before starting to code, it is imperative to identify which signals are the ones that need to be passed down and the ones that are not useful anymore. From simple analysys of the code, Table 3.1 lists all the outputs expected at the boundary of each stage, indicated as the union of the names of the two stages that the boundary shares.

Once that each register is described, not timing requirements for setting the load enable for both destination register and Program Counter is needed, since the change in the IF's input can be updated every clock cycle without any risk of glitching due to the simulation not taking into account the delay of each stage. Before setting up the Structural definition of the datapath, including registers and major blocks, the IF stage must be edited to increment the PC before the first register, otherwise PC+4 would instead be returned by the WB, 4 clock cycle after the

IF/ID	ID/IE	IE/DM	DM/WB	WB
curr_pc	curr_pc	curr_pc	next_pc	pc_out
next_pc	next_pc	next_pc	mem_out	rd_value
instr	op_class	op_class	branch_cond	pc_load_en
	funct3	funct3	alu_result	mem_we
	funct7	alu_result	rd_addr	rd_addr
	a_sel	branch_cond	op_class	
	b_sel	rs2_value		
	imm_se	rd_addr		
	rs1_value			
	rs2_value			
	rd_addr			
	cond_opcode			

Table 3.1: List of input signals for each register

instruction is first withdrawed from the IM. With this being said, WB will return a value for the PC only during jumps and branches, while the IF will select between PC+4 and the returning value from WB.

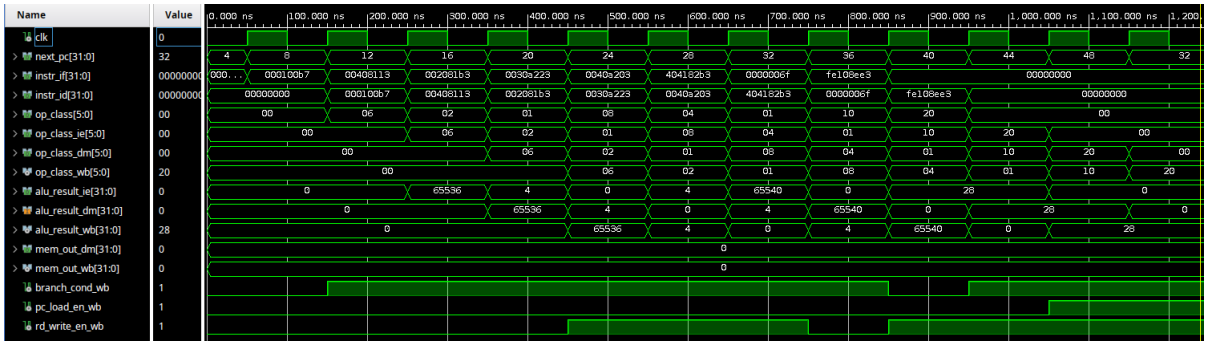


Figure 3.2: Datapath simulation with pipeline

Now the processor can execute a part of the instruction in a clock cycle, allowing for higher clock speeds, yet this improvement creates some bugs, one of which can be directly observed from Figure 3.2; In fact, right after the second instruction reaches the execution stage, the *addi* retrieves a 0 instead of 65535 as content of x1, as happens in Figure 2.10 where the instruction returns 65540 as a result. Also, in case of jumps and branches, the correct value for the Program Counter is updated after one clock cycles while the previous instructions still get executed. In addition, since x1 is updated only after 5 clock cycles, the store and load instrucction will operate with different addresses. While the architecture is now completed, it misses some important features that solve these types of hazards and therefore ensure correct program execution; Thus it becomes essential to implement mechanisms that can handle such hazards to preserve instruction accuracy and consistency while maintaining the performance of a pipelined architecture.

Solving data hazards

Pipelined architectures change the core system in order to execute instructions partially and with each segment being executed in sequence. This allows for faster clock and a higher throughput, but the fact that each instruction is initiated, in its execution, just a clock cycle later than the previous one introduces a crippling problem regarding the addressing of the same register in consecutive commands, as it happened in the previous simulation (Figure 3.2). This problem can be solved easily by either stalling the pipeline or by passing down the result of the ALU to another stage when needed, also called operand forwarding. By comparing the destination register of the previous instructions, both in the DM and WB stages, and the operands of the next one, the datapath can be programmed to not increment the PC, thus stalling the pipeline for some clock cycles; Although simple, the solution is wasteful and can lead to slowdowns during very intensive arithmetic operations. Operand Forwarding could instead have the execution flow just as before and still solve the problem; By adopting the same comparison method, the multiplexers at the ALU's inputs can select the previous result when necessary, and when the same register is addressed multiple times before it can be written, the most recent data can be used. To successfully implement operand forwarding inside the IE, there is the need to analyze the previous destination register's address that has been passed down to

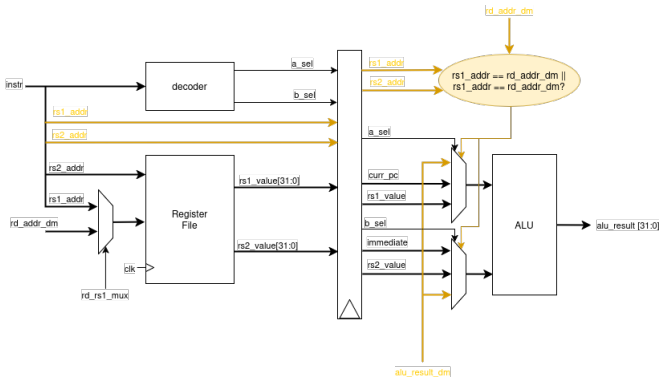


Figure 3.3: Concept for operand forwarding for solving data hazards

both DM and WB, as said earlier, then confront both of the addresses with the address of each one of the operands and finally overwrite them with the most recent ALU's output. A VHDL description could be given below:

```
rs1_fw          <= alu_result_wb when (rd_addr_wb = rs1_addr_in)
                    else alu_result_dm when (rd_addr_dm = rs1_addr_in)
                    else rs1_value_in;
rs2_fw          <= alu_result_wb when (rd_addr_wb = rs2_addr_in)
                    else alu_result_dm when (rd_addr_dm = rs2_addr_in)
                    else rs2_value_in;
first_operand   <= rs1_fw when a_sel = '1' else curr_pc;
second_operand  <= rs2_fw when b_sel = '1' else imm_se;
```

However this code is incomplete, including the fact that the forwarded value for rs2 has to reach the data memory for the right value to be stored (which can already be dealt with by having *rs2_fw* as output from the IE), the register file will always be written after 3 clock cycles from the execution stage, meaning that there is another stage that must be covered, which is the ID. In this case an additional line can be added in order for the output of the register file to be overwritten if the register that has to be accessed has not been written yet:

```
reg : register_file
port map(
    clk         => clk,
    da          => instr(11 downto 7) when op_class = "000110"  else instr(19 downto 15),
    pda         => instr(24 downto 20),
    dina        => rd_addr_in,
    din         => rd_value,
    we          => rd_write_en,
    rso         => rs1_value_reg,
    prso        => rs1_value_reg);

rd_addr <= instr(11 downto 7);
rs1_addr<= instr(19 downto 15);
rs2_addr<= instr(24 downto 20);

rs1_value_out <= rd_value when (instr(19 downto 15) = rd_addr_wb and rd_write_en = '1')
                 else rs1_value_reg;
rs2_value_out <= rd_value when (instr(24 downto 20) = rd_addr_wb and rd_write_en = '1')
                 else rs1_value_reg;
```

With the forwarding logic now ready, the simulation will show that the processor may be able to change the value of the operand with the latest data from other stages, this can be proven by having the store and load operation reading and writing from the same register as happens in the unpipelined datapath simulation. In Figure 3.4 the withdrawed register values are shown in yellow, the forwarded values are highligted in orange. The final value used for each operand is shown in red and the results are exactly what expected, with the SW instruction having the ALU's result pointing at the address number 1, since the upper 19 bits of rs2 (containing 65540) are ignored (hence the resulting address should 4) with the first two bits as well, and with LW being able to address the same location in the memory, whose value is shown in the signal at the bottom to be in fact 131076.

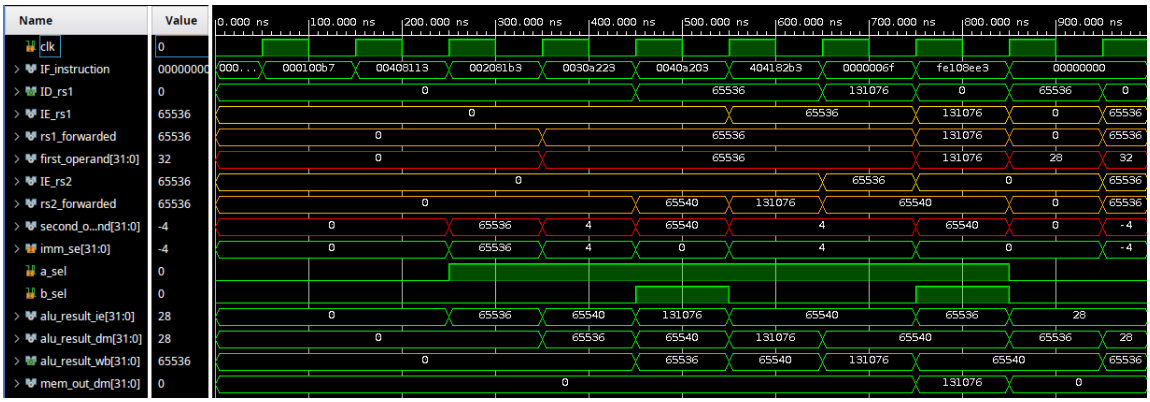


Figure 3.4: Datapath simulation with forwarding logic

Managing control hazards

Managing jumps and branches in a pipelined architecture is not trivial but it is a necessary step to take to achieve a functional processor. Overlooking this feature could lead to incorrect values written in some registers or in the data cache while a jump instruction is being executed. To make an example, the following instruction can be analyzed step by step:

```
jal x5, 8           ; PC += 8, x5 = PC + 4
add x1, x2, x3      ; x1 = x2 + x3
sub x4, x1, x3      ; x4 = x1 - x3
```

As the first instruction is decoded, the addition is already being fetched and while the PC gets updated just after 4 clock cycles, both the addition and subtraction have been almost fully executed, leading to the content of x4 being updated to the value stored in x2, two cycles after the jump has completed its elaboration and updated the PC, leading to a double execution of the subtraction that also uses a wrong value for x1. One solution for this would imply that, when the jump is first decoded, the PC has to be kept constant until the jump is fully executed. In addition, all data given by each stage after the jump is first decoded has to be reset (flushed) in order to avoid any unnecessary elaboration. To do this, every register in the pipeline could have a reset input that discards all input values and keeps all outputs to zero; With the current outputs available from the other stages, this is easily doable by having the latter being pulled high when *op_class* = "010000" (unconditional jump) at any stage or when a branch has its condition true, that is, when *branch_cond* = 1. By looking at the actual code, the first modification that needs to be made is adding a new input to the IF entity that can keep the PC constant until a new value is loaded. The snippet below shows how the PC can interact with the stated modification:

```
process (clk)
begin
    if rising_edge(clk) then
        if pc_stall = '1' then
            pc_reg  <= pc_reg;
        elsif pc_load_en = '1' then
            pc_reg  <= unsigned(pc_in);
        else
            pc_reg <= pc_reg + 4;
        end if;
    end if;
end process;
```

The second crucial modification that has to be introduced is a way to have all the inter-stage registers, in particular the ones before the current elaboration stage of the jump, flushed. Since *op_class* and *branch_cond* are propagated to every stage after decoding, the sole presence of an operation class associated to an unconditional jump, or a true condition for a branch, at any stage should be a sufficient condition for the register to be flushed. For each one of them, Table 3.2 shows which condition for the flush can be used:

IF/ID	branch_cond_(ie,dm) = 1 OR op_class_(ie,dm) = "010000"
ID/IE	branch_cond_(ie,dm,wb) = 1 OR op_class_(ie,dm,wb) = "010000"
IE/DM	branch_cond_(dm,wb) = 1 OR op_class_(dm,wb) = "010000"
DM/WB	branch_cond_(wb) = 1 OR op_class_(wb) = "010000"

Table 3.2: Flush condition for each inter-stage register

One thing to notice is that the IF/ID stage is not being flushed when a jump instruction reaches WB, this is because at the rising edge of the clock, after the jump reaches the writeback, the PC value is updated and so does the inputs at the register. Having IF/ID flushed in that moment would mean that the PC is going to be updated to the value at the output of DM/WB, which was previously flushed and would return a 0 as next address. Another observation that needs to be made before implementing the logic is that aas of now, branch_cond can be high even if there is no branch instruction being executed, for that reason, an additional modification has to be made to the original code of the IE stage:

```
architecture Structural of instr_exec_pipeline is

    -- other signal declarations

    signal branch_cond_buf  : std_logic;

    -- ALU component declaration

    component comparator is
        port (
            first_operand   : in std_logic_vector(31 downto 0);
            second_operand  : in std_logic_vector(31 downto 0);
            cond_opcode     : in std_logic_vector(2 downto 0);

            branch_cond     : out std_logic);
    end component;
begin
    comp : comparator
    port map(
        first_operand   => rs1_value_in,
        second_operand  => rs2_value_in,
        cond_opcode     => cond_opcode,
        branch_cond     => branch_cond_buf);

    -- forwarding logic and ALU instantiation

    branch_cond     <= branch_cond_buf when op_class(5) else '0';
end Structural;
```

Source Code 5.18 shows in the IF/ID register the implementation of a flush input, while Source Code 5.22 shows the statements that drive that input in each register.

Simulation and Testing

With all the basic features needed for the architecture to work properly, there is the need to test if jumps can work properly, therefore a new *.coe* file can be loaded to the IM:

```
    memory_initialization_radix=16;
    memory_initialization_vector=

    000100b7,   --> lui x1, 16
    00408113,   --> addi x2, x1, 4
    002081b3,   --> add x3, x1, x2
    0030a223,   --> sw x3, 4(x1)
    00309423,   --> sh x3, 8(x1)
    00308623,   --> sb x3, 12(x1)
    0040a283,   --> lw x5, 4(x1)
    00409303,   --> lh x6, 4(x1)
    00408383,   --> lb x7, 4(x1)
    404282b3,   --> sub x5, x5, x4
    00100113,   --> addi x2, x0, 1
    00209293,   --> slli x5, x1, 2
    0020d293,   --> srli x5, x1, 2
    0030a033,   --> slt x0, x1, x3
    00302113,   --> slti x2, x0, 3
```

```
fe108ce3,   --> beq x1, x1, -8
0000006f;   --> jal x0, 0
```

By executing this program a loop should be expected to initiate just before the jump, having the program go back.

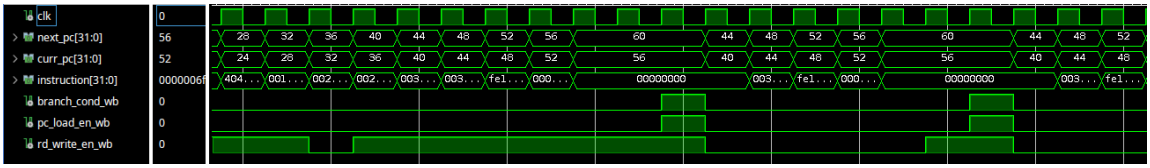


Figure 3.5: CPU stalling any increment of the PC until the branch is fully executed

With everything simulated, the architecture can now be tested. The chosen board for this project is a Nexys4 DDR, by setting up the constraint file, the simplest way to check the content of the memory to show the correct execution of the program would be using the included switches to select the register and the 7-segment displays to show its content. Source Code 5.23 shows a simple driver for the displays with a function that converts a four bits vector to a hexadecimal value, so that the content of each address in memory or register can be fully shown on 8 ciphers. The block itself though needs access to DM and register file, thus some modifications need to be made; For the first one, an easy way is to re-declare the DM as a dual-port RAM and access the data from port B; As for the Register File, just one additional line can be added to the original code:

```
entity register_file is
    Port (

        -- signals that connect to the ID stage

        addr_sel    : in std_logic_vector(13 downto 0);
        reg_out     : out std_logic_vector(31 downto 0)
    );
end register_file;

architecture Behavioral of register_file is
    -- ...
begin
    -- clock sensitive process definition for register file

    rso     <= registers(to_integer(unsigned(da)));
    prso    <= registers(to_integer(unsigned(pda)));

    -- checking registers from data reader
    reg_out <= registers(to_integer(unsigned(addr_sel(4 downto 0))));
end Behavioral;
```

The final test requires to check if data corresponds to the table below:

Location	Address	Value
DM	4	$131076_{10} = 20004_{16}$
	8	0004_{16}
	12	04_{16}
RF	x1	$65536_{10} = 10000_{16}$
	x2	$FFFFFFFF_{16}$
	x3	$131076_{10} = 20004_{16}$
	x4	0 (should never be written)
	x5	$16384_{10} = 4000_{16}$
	x6	0004_{16}
	x7	04_{16}

Table 3.3: Final test for the implementation on Nexys 4 DDR

Conclusions

With a fully functional, syntetizable CPU, it has been shown how a simple RISC-V core can be designed in VHDL. However, this project, as of now, does not include a decent interaction between L1 cache (both I-cache and D-cache) and an external L2 cache or RAM; Also, only the RV32I is implemented (partially) and for any other expansion that can stem from the full ISA there is the need to extend the code in both the decoder and the ALU, although it should be much simpler to do so from this point on.
Adding features with a working base to start from should be easy, and the project could take another step further in the future; One good feature would be adding the capability for the processor to execute multiplications and divisions between integers, easy enough since VHDL introduces the synthetizable operators '*' and '/'.
One thing that won't be easy, though, is implementing floating-point operations, since it would require the introduction to a data type that follow the IEEE 754 standard for floating point numbers. The only way to avoid the problem would be having them encoded in VHDL using the data type **real**, which is not synthetizable, although it would make a good option for HDL designs that won't go further from simulation.

Appendices

Appendix A: Instruction Tables

class	funct7	funct3	instruction	operation
OP	0000000	000	ADD	addition
	0100000	000	SUB	subtraction
	0000000	001	SLL	left logic shift
	0000000	010	SLT	set if less than
	0000000	011	SLTU	set if less than unsigned
	0000000	100	XOR	bit-wise EXOR
	0000000	101	SRL	right logic shift
	0100000	101	SRA	right arithmetic shift
	0000000	110	OR	bit-wise OR
	0000000	111	AND	bit-wise AND
OP-IMM	-	000	ADDI	addition
	0000000	001	SLLI	left logic shift
	-	010	SLTI	set if less than
	-	011	SLTIU	set if less than unsigned
	-	100	XORI	bit-wise EXOR
	0000000	101	SRLI	right logic shift
	0100000	101	SRAI	right arithmetic shift
	-	110	ORI	bit-wise OR
	-	111	ANDI	bit-wise AND
STORE	-	000	SB	addition
	-	001	SH	addition
	-	010	SW	addition
LOAD	-	000	LB	addition
	-	001	LH	addition
	-	010	LW	addition
	-	100	LBU	addition
	-	101	LHU	addition
JALR	-	000	JALR	addition
JAL	-	-	JAL	addition
LUI	-	-	LUI (pseudo instruction LI)	addition
AUIPC	-	-	AUIPC	addition
BRANCH	-	000	BEQ	branch if equal
	-	001	BNE	branch if not equal
	-	100	BLT	branch if lower than
	-	101	BGE	branch if greater or equal
	-	110	BLTU	branch if lower than unsigned
	-	111	BGEU	branch if greater or equal unsigned

Table 5.1: RV32I core instructions

Appendix B: Code segments of unpipelined datapath

Instruction Fetch [RV32I]

Base architecture

Source Code 5.1: Architecure of the Instruction Fetch

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


entity instr_fetch is
    port (
        clk         : in std_logic;
        pc_load_en  : in std_logic;
        pc_in       : in std_logic_vector(31 downto 0);

        next_pc     : out std_logic_vector(31 downto 0);
        curr_pc     : out std_logic_vector(31 downto 0);
        instr       : out std_logic_vector(31 downto 0));
end instr_fetch;

architecture Structural of instr_fetch is
    signal pc_reg  : unsigned(31 downto 0) := (others => '0');

    component instruction_memory
    port(
        clka        : in std_logic;
        wea         : in std_logic;
        addra       : in std_logic_vector(13 downto 2);
        dina        : in std_logic_vector(31 downto 0);
        douta       : out std_logic_vector(31 downto 0));
    end component;
begin
    instr_mem: instruction_memory
    port map(
        clka        => clk,
        wea         => '0',
        dina        => (others => '0'),
        addra       => std_logic_vector(pc_reg(13 downto 2)),
        douta       => instr);

    process (clk)
    begin
        if rising_edge(clk) then
            if pc_load_en = '1' then
                pc_reg  <= unsigned(pc_in);
            end if;
        end if;
    end process;
    next_pc     <= std_logic_vector(pc_reg + 4);
    curr_pc     <= std_logic_vector(pc_reg);
end Structural;
```

Testbench

Source Code 5.2: first Instruction Fetch testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity IF_testbench is
end IF_testbench;

architecture Behavioral of IF_testbench is
    constant clk_period     : time := 20 ns;
    signal pc_in            : std_logic_vector(31 downto 0) := (others => '0');
    signal pc_load_en       : std_logic := '1';
    signal next_pc          : std_logic_vector(31 downto 0) := (others => '0');
    signal curr_pc          : std_logic_vector(31 downto 0) := (others => '0');
    signal instr            : std_logic_vector(31 downto 0) := (others => '0');
    signal clk              : std_logic := '0';

    component instr_fetch
        port (
            clk         : in std_logic;
            pc_load_en  : in std_logic;
            pc_in       : in std_logic_vector(31 downto 0);
            next_pc     : out std_logic_vector(31 downto 0);
            curr_pc     : out std_logic_vector(31 downto 0);
            instr       : out std_logic_vector(31 downto 0));
    end component;
begin
    if_inst : instr_fetch
        port map (
            clk         => clk,
            pc_load_en  => pc_load_en,
            pc_in       => pc_in,
            next_pc     => next_pc,
            curr_pc     => curr_pc,
            instr       => instr);

    process
    begin
        clk <= '0';
        wait for clk_period / 2;
        clk <= '1';
        wait for clk_period / 2;
    end process;

    pc_in       <= next_pc;
end Behavioral;
```

Instruction Decode [RV32I]

Source Code 5.3: Architecure of the Instruction Fetch

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity instr_decode is
    Port (
        clk         : in std_logic;
        instr       : in std_logic_vector(31 downto 0);

        -- Inputs from mem writeback

        rd_write_en : in std_logic;
        rd_value    : in std_logic_vector(31 downto 0);
        rd_addr_in  : in std_logic_vector(4 downto 0);

        -- Decoded instruction informations

        op_class    : out std_logic_vector(5 downto 0);
        funct3      : out std_logic_vector(2 downto 0);
        funct7      : out std_logic_vector(6 downto 0);
        a_sel       : out std_logic;
        b_sel       : out std_logic;
        cond_opcode : out std_logic_vector(2 downto 0);
        rd_addr_out : out std_logic_vector(4 downto 0);

        -- Data to be elaborated

        rs1_value   : out std_logic_vector(31 downto 0);
        rs2_value   : out std_logic_vector(31 downto 0);
        imm_se      : out std_logic_vector(31 downto 0)
    );
end instr_decode;

architecture Structural of instr_decode is
    signal rd_rs1_mux       : std_logic_vector(4 downto 0) := (others => '0');
    signal we               : std_logic := '0';

    component register_file is
    port (
        clk         : in std_logic;
        da          : in std_logic_vector(4 downto 0);
        pda         : in std_logic_vector(4 downto 0);
        dina        : in std_logic_vector(4 downto 0);
        din         : in std_logic_vector(31 downto 0);
        we          : in std_logic;

        rso         : out std_logic_vector(31 downto 0);
        prso        : out std_logic_vector(31 downto 0));
    end component;

    component decoder is
    port(
        instr       : in std_logic_vector(31 downto 0);
        op_class    : out std_logic_vector(5 downto 0);
        funct3      : out std_logic_vector(2 downto 0);
        funct7      : out std_logic_vector(6 downto 0);
        a_sel       : out std_logic;
        b_sel       : out std_logic;
        cond_opcode : out std_logic_vector(2 downto 0);
        imm_se      : out std_logic_vector(31 downto 0));
    end component;
begin
    reg : register_file
    port map(
        clk         => clk,
        da          => instr(19 downto 15),
        pda         => instr(24 downto 20),
        dina        => rd_addr_in,
        din         => rd_value,
        we          => rd_write_en,
        rso         => rs1_value,
        prso        => rs2_value);
```

```
    dec : decoder
    port map(
        instr       => instr,
        op_class    => op_class,
        funct3      => funct3,
        funct7      => funct7,
        a_sel       => a_sel,
        b_sel       => b_sel,
        cond_opcode => cond_opcode,
        imm_se      => imm_se);

    rd_addr_out <= instr(11 downto 7);
end Structural;
```

Decoder

Source Code 5.4: Decoder fully implementing RV32I instructions

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity decoder is
    Port (
        instr       : in std_logic_vector(31 downto 0);
        op_class    : out std_logic_vector(5 downto 0);
        funct3      : out std_logic_vector(2 downto 0);
        funct7      : out std_logic_vector(6 downto 0);
        a_sel       : out std_logic;
        b_sel       : out std_logic;
        cond_opcode : out std_logic_vector(2 downto 0);
        imm_se      : out std_logic_vector(31 downto 0)
     );
end decoder;  -- make so that the decoder gives out the addresses of the first and second operand fot the LUI instruction doe

architecture Behavioral of decoder is
begin
process(instr)
begin
case instr(6 downto 5) is
    when "00" =>
        case instr(4 downto 0) is
            when "000" & "11"  => -- LOAD [I-type]
                op_class    <=  "000100";
                imm_se      <=  std_logic_vector(resize(signed(instr(31 downto 20)), imm_se'length));-- Sign extension
                a_sel       <=  '1'; -- Select register as first operand
                b_sel       <=  '0'; -- Select immediate as second operand
                funct3      <=  instr(14 downto 12);
                funct7      <=  (others => '0');
                cond_opcode <=  (others => '0');
            when "100" & "11"  => -- OP-IMM [I-type]
                op_class    <=  "000010";
                imm_se      <=  std_logic_vector(resize(signed(instr(31 downto 20)), imm_se'length));
                a_sel       <=  '1';
                b_sel       <=  '0';
                funct3      <=  instr(14 downto 12);
                funct7      <=  instr(31 downto 25); -- shifts have funct7
                cond_opcode <=  (others => '0');
            when "101" & "11"  => -- AUIPC [U-type] can be encoded as a combination jump/operation
                op_class    <=  "010010";
                imm_se      <=  instr(31) & instr(30 downto 20)
                                & instr(19 downto 12) & (11 downto 0 => '0');
                a_sel       <=  '0'; -- Select Program counter as first operand
                b_sel       <=  '0'; -- Select immediate as second operand
                funct3      <=  (others => '0');
                funct7      <=  (others => '0');
                cond_opcode <=  (others => '0');
            when others =>
                op_class    <=  (others => '0');
                imm_se      <=  (others => '0');
                a_sel       <=  '0';
                b_sel       <=  '0';
                funct3      <=  (others => '0');
                funct7      <=  (others => '0');
                cond_opcode <=  (others => '0');
        end case;
```

```
when "01" =>
    case instr(4 downto 0) is
        when "000" & "11"  => -- STORE [S-type]
            op_class    <=  "001000";
            imm_se      <=  std_logic_vector(resize(signed(instr(31 downto 25)
                            & instr(11 downto 7)), imm_se'length));
            a_sel       <=  '1';
            b_sel       <=  '0';
            funct3      <=  instr(14 downto 12);
            funct7      <=  (others => '0');
            cond_opcode <=  (others => '0');
        when "100" & "11"  => -- OP [R-type]
            op_class    <=  "000001";
            imm_se      <=  (others => '0');
            a_sel       <=  '1';
            b_sel       <=  '1';
            funct3      <=  instr(14 downto 12);
            funct7      <=  instr(31 downto 25);
            cond_opcode <=  (others => '0');
        when "101" & "11"  => -- LUI [U-type] encoded as a combination load/operation
            op_class    <=  "000110";
            imm_se      <=  instr(31) & instr(30 downto 20)
                            & instr(19 downto 12) & (11 downto 0 => '0');
            a_sel       <=  '1';
            b_sel       <=  '0';
            funct3      <=  (others => '0');
            funct7      <=  (others => '0');
            cond_opcode <=  (others => '0');
        when others =>
            op_class    <=  (others => '0');
            imm_se      <=  (others => '0');
            a_sel       <=  '0';
            b_sel       <=  '0';
            funct3      <=  (others => '0');
            funct7      <=  (others => '0');
            cond_opcode <=  (others => '0');
    end case;
when "11" =>
    case instr(4 downto 0) is
        when "000" & "11"  => -- BRANCH [B-type]
            op_class    <=  "100000";
            imm_se      <=  std_logic_vector(resize(signed(instr(31) & instr(7) &
                            instr(30 downto 25) & instr(11 downto 8) & '0'), imm_se'length));
            a_sel       <=  '0';
            b_sel       <=  '0';
            funct3      <=  (others => '0');
            funct7      <=  (others => '0');
            cond_opcode <=  instr(14 downto 12);
        when "001" & "11"  => -- JALR [I-type]
            op_class    <=  "010000";
            imm_se      <=  std_logic_vector(resize(signed(instr(31 downto 20)), imm_se'length));
            a_sel       <=  '0';
            b_sel       <=  '0';
            funct3      <=  instr(14 downto 12);
            funct7      <=  (others => '0');
            cond_opcode <=  (others => '0');
        when "011" & "11"  => -- JAL [J-type]
            op_class    <=  "010000";
            imm_se      <=  std_logic_vector(resize(signed(instr(31) & instr(19 downto 12)
                            & instr(20) & instr(30 downto 21) & "0"), imm_se'length));
            a_sel       <=  '0';
            b_sel       <=  '0';
            funct3      <=  (others => '0');
            funct7      <=  (others => '0');
            cond_opcode <=  (others => '0');
        when others =>
            op_class    <=  (others => '0');
            imm_se      <=  (others => '0');
            a_sel       <=  '0';
            b_sel       <=  '0';
            funct3      <=  (others => '0');
            funct7      <=  (others => '0');
            cond_opcode <=  (others => '0');
    end case;
when others =>
    op_class    <=  (others => '0');
    imm_se      <=  (others => '0');
```

```
        a_sel       <=  '0';
        b_sel       <=  '0';
        funct3      <=  (others => '0');
        funct7      <=  (others => '0');
        cond_opcode <=  (others => '0');
end case;
end process;
end Behavioral;
```

Register File

Source Code 5.5: Simple 32 bits Register File

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity register_file is
    Port (
        clk         : in std_logic;
        da          : in std_logic_vector(4 downto 0);
        pda         : in std_logic_vector(4 downto 0);
        dina        : in std_logic_vector(4 downto 0);
        din         : in std_logic_vector(31 downto 0);
        we          : in std_logic;

        rso         : out std_logic_vector(31 downto 0);
        prso        : out std_logic_vector(31 downto 0);

        -- checking out registers from outside

        addr_sel    : in std_logic_vector(13 downto 0);
        reg_out     : out std_logic_vector(31 downto 0)
    );
end register_file;

architecture Behavioral of register_file is
    type reg_file is array(31 downto 0) of std_logic_vector(31 downto 0);
    signal registers: reg_file  := (others => (others => '0'));
    signal rso_buf  : std_logic_vector(31 downto 0) := (others => '0');
    signal prso_buf : std_logic_vector(31 downto 0) := (others => '0');
begin
    process(clk)
        variable registers_buf  : reg_file  := (others => (others => '0'));
    begin
        if rising_edge(clk) then
            if we = '1' and dina /= "00000" then
                registers_buf(to_integer(unsigned(dina))) := din;
            end if;
            registers   <= registers_buf;
        end if;
    end process;

    rso     <= registers(to_integer(unsigned(da)));
    prso    <= registers(to_integer(unsigned(pda)));

    -- checking registers from data reader
    reg_out <= registers(to_integer(unsigned(addr_sel(4 downto 0))));
end Behavioral;
```

Testbench

Source Code 5.6: Instruction Decode Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ID_testbench is
end ID_testbench;

architecture Behavioral of ID_testbench is
constant clk_period     : time := 100 ns;
signal pc_in            : std_logic_vector(31 downto 0) := (others => '0');
signal pc_load_en       : std_logic := '1';
```

```
signal next_pc          : std_logic_vector(31 downto 0) := (others => '0');
signal next_pc_reg      : std_logic_vector(31 downto 0) := (others => '0');
signal curr_pc          : std_logic_vector(31 downto 0) := (others => '0');
signal instr            : std_logic_vector(31 downto 0) := (others => '0');
signal clk              : std_logic := '0';

-- ID signals

signal rd_write_en      : std_logic := '0';
signal rd_value         : std_logic_vector(31 downto 0) := (others => '0');
signal rd_addr_in       : std_logic_vector(4 downto 0) := (others => '0');
signal rd_addr_out      : std_logic_vector(4 downto 0) := (others => '0');
signal op_class         : std_logic_vector(5 downto 0) := (others => '0');
signal funct3           : std_logic_vector(2 downto 0) := (others => '0');
signal funct7           : std_logic_vector(6 downto 0) := (others => '0');
signal a_sel            : std_logic := '0';
signal b_sel            : std_logic := '0';
signal cond_opcode      : std_logic_vector(2 downto 0) := (others => '0');
signal rs1_value        : std_logic_vector(31 downto 0) := (others => '0');
signal rs2_value        : std_logic_vector(31 downto 0) := (others => '0');
signal imm_se           : std_logic_vector(31 downto 0) := (others => '0');

component instr_fetch
    port (
        clk         : in std_logic;
        pc_load_en  : in std_logic;
        pc_in       : in std_logic_vector(31 downto 0);
        next_pc     : out std_logic_vector(31 downto 0);
        curr_pc     : out std_logic_vector(31 downto 0);
        instr       : out std_logic_vector(31 downto 0));
end component;

component instr_decode
    port (
        clk         : in std_logic;
        instr       : in std_logic_vector(31 downto 0);
        rd_write_en : in std_logic;
        rd_value    : in std_logic_vector(31 downto 0);
        rd_addr_in  : in std_logic_vector(4 downto 0);
        op_class    : out std_logic_vector(5 downto 0);
        funct3      : out std_logic_vector(2 downto 0);
        funct7      : out std_logic_vector(6 downto 0);
        a_sel       : out std_logic;
        b_sel       : out std_logic;
        cond_opcode : out std_logic_vector(2 downto 0);
        rd_addr_out : out std_logic_vector(4 downto 0);
        rs1_value   : out std_logic_vector(31 downto 0);
        rs2_value   : out std_logic_vector(31 downto 0);
        imm_se      : out std_logic_vector(31 downto 0));
end component;
begin
    if_inst : instr_fetch
        port map (
            clk         => clk,
            pc_load_en  => pc_load_en,
            pc_in       => pc_in,
            next_pc     => next_pc,
            curr_pc     => curr_pc,
            instr       => instr);
    id_inst : instr_decode
        port map (
            clk         => clk,
            instr       => instr,
            rd_write_en => rd_write_en,
            rd_value    => rd_value,
            rd_addr_in  => rd_addr_in,
            op_class    => op_class,
            funct3      => funct3,
            funct7      => funct7,
            a_sel       => a_sel,
            b_sel       => b_sel,
            cond_opcode => cond_opcode,
            rd_addr_out => rd_addr_out,
            rs1_value   => rs1_value,
            rs2_value   => rs2_value,
            imm_se      => imm_se);
    process
```

```
    begin
        clk <= '0';
        wait for clk_period / 2;
        clk <= '1';
        wait for clk_period / 2;
    end process;
    pc_in       <= next_pc;
end Behavioral;
```

Instruction Execute [RV32I]

Architecture

Source Code 5.7: Architecture of Instruction Execute

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity instr_exec is
    Port (
        a_sel           : in std_logic;
        b_sel           : in std_logic;
        rs1_value       : in std_logic_vector(31 downto 0);
        rs2_value       : in std_logic_vector(31 downto 0);
        imm_se          : in std_logic_vector(31 downto 0);
        curr_pc      : in std_logic_vector(31 downto 0);
        cond_opcode     : in std_logic_vector(2 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
        funct7          : in std_logic_vector(6 downto 0);
        op_class        : in std_logic_vector(5 downto 0);

        branch_cond     : out std_logic;
        alu_result      : out std_logic_vector(31 downto 0)
     );
end instr_exec;

architecture Structural of instr_exec is
    signal alu_mux_a    : std_logic_vector(31 downto 0);
    signal alu_mux_b    : std_logic_vector(31 downto 0);

    component ALU is
        port (
            first_operand   : in std_logic_vector(31 downto 0);
            second_operand  : in std_logic_vector(31 downto 0);
            funct3          : in std_logic_vector(2 downto 0);
            funct7          : in std_logic_vector(6 downto 0);
            op_class        : in std_logic_vector(5 downto 0);

            alu_result      : out std_logic_vector(31 downto 0));
    end component;

    component comparator is
        port (
            first_operand   : in std_logic_vector(31 downto 0);
            second_operand  : in std_logic_vector(31 downto 0);
            cond_opcode     : in std_logic_vector(2 downto 0);

            branch_cond     : out std_logic);
    end component;
begin
    comp : comparator
    port map(
        first_operand   => rs1_value,
        second_operand  => rs2_value,
        cond_opcode     => cond_opcode,
        branch_cond     => branch_cond);

    alu_1 : ALU
    port map(
        first_operand   => alu_mux_a,
        second_operand  => alu_mux_b,
        funct3          => funct3,
        funct7          => funct7,
        op_class        => op_class,

        alu_result      => alu_result);

    alu_mux_a   <= rs1_value when a_sel = '1' else curr_pc;
    alu_mux_b   <= rs2_value when b_sel = '1' else imm_se;
end Structural;
```

ALU

Source Code 5.8: ALU

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ALU is
    Port (
        first_operand   : in std_logic_vector(31 downto 0);
        second_operand  : in std_logic_vector(31 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
        funct7          : in std_logic_vector(6 downto 0);
        op_class        : in std_logic_vector(5 downto 0);

        alu_result      : out std_logic_vector(31 downto 0)
    );
end ALU;

architecture Behavioral of ALU is
begin
process(op_class, funct3, funct7, first_operand, second_operand)
begin
    case op_class is
        when "000010" | "000001" => -- OP and OP-IMM
            case funct3 is
                when "001" => -- SLL,SLLI
                    alu_result <= std_logic_vector(shift_left(unsigned(first_operand),
                                        to_integer(unsigned(second_operand))));
                when "010" => -- SLT,SLTI
                    alu_result <=
                        (0 => '1', others => '1')
                    when (unsigned(first_operand) < unsigned(second_operand)) else
                        (others => '0');
                when "011" => -- SLTU,SLTIU
                    alu_result <=
                        (0 => '1', others => '1')
                    when (unsigned(first_operand) < unsigned(second_operand)) else
                        (others => '0');
                when "100" => -- XOR, XORI
                    alu_result <= first_operand xor second_operand;
                when "101" => -- right shifts
                    alu_result     <=
                    std_logic_vector(shift_right(unsigned(first_operand),
                                        to_integer(unsigned(second_operand(4 downto 0)))))
                    when funct7(5) = '1' else
                    std_logic_vector(shift_right(signed(first_operand),
                                        to_integer(unsigned(second_operand(4 downto 0)))));
                when "110" => -- OR,ORI
                    alu_result      <= first_operand or second_operand;
                when "111" => -- AND, ANDI
                    alu_result      <= first_operand and second_operand;
                when others =>
                    if funct7(5) = '1' and op_class = "000001" then
                        alu_result <=
                            std_logic_vector(signed(first_operand) - signed(second_operand));
                    else
                        alu_result <=
                            std_logic_vector(signed(first_operand) + signed(second_operand));
                    end if;
            end case;
        when "000100" => -- LOAD
            alu_result  <= std_logic_vector(signed(first_operand) + signed(second_operand));
        when "001000" => -- STORE
            alu_result  <= std_logic_vector(signed(first_operand) + signed(second_operand));
        when "000110" => -- LUI
            alu_result  <= second_operand;
        when "010010" => -- AUIPC
            alu_result  <= std_logic_vector(signed(first_operand) + signed(second_operand));
        when "010000" => -- JUMP
            alu_result  <= std_logic_vector(signed(first_operand) + signed(second_operand));
        when "100000" => -- BRANCH
            alu_result  <= std_logic_vector(signed(first_operand) + signed(second_operand));
        when others =>
            alu_result  <= (others => '0');
    end case;
end process;
end Behavioral;
```

Comparator

Source Code 5.9: Comparator

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity comparator is
    port (
        first_operand   : in std_logic_vector(31 downto 0);
        second_operand  : in std_logic_vector(31 downto 0);
        cond_opcode     : in std_logic_vector(2 downto 0);

        branch_cond     : out std_logic
    );
end comparator;

architecture Behavioral of comparator is
begin
    process(cond_opcode, first_operand, second_operand)
    begin
        branch_cond     <= '0';
        case cond_opcode is
            when "000"  =>          -- EQ
                if first_operand = second_operand then
                    branch_cond <= '1';
                end if;
            when "001"  =>          -- NEQ
                if not(first_operand = second_operand) then
                    branch_cond <= '1';
                end if;
            when "100"  =>          -- LT lower than
                if signed(first_operand) < signed(second_operand) then
                    branch_cond <= '1';
                end if;
            when "101"  =>          -- GE greater or equal
                if signed(first_operand) >= signed(second_operand) then
                    branch_cond <= '1';
                end if;
            when "110"  =>          -- LT unsigned
                if unsigned(first_operand) < unsigned(second_operand) then
                    branch_cond <= '1';
                end if;
            when "111"  =>          -- GE unsigned
                if unsigned(first_operand) >= unsigned(second_operand) then
                    branch_cond <= '1';
                end if;
            when others =>
                branch_cond <= '0';
        end case;
    end process;
end Behavioral;
```

Testbench

Source Code 5.10: Instruction Execute Testbench

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity IE_testbench is
end IE_testbench;

architecture Behavioral of IE_testbench is
constant clk_period     : time := 100 ns;
signal pc_in            : std_logic_vector(31 downto 0) := (others => '0');
signal next_pc          : std_logic_vector(31 downto 0) := (others => '0');
signal next_pc_reg      : std_logic_vector(31 downto 0) := (others => '0');
signal curr_pc          : std_logic_vector(31 downto 0) := (others => '0');
signal instr            : std_logic_vector(31 downto 0) := (others => '0');
signal clk              : std_logic := '0';
signal pc_load_en       : std_logic := '1';

signal rd_write_en      : std_logic := '0';
signal rd_value         : std_logic_vector(31 downto 0) := (others => '0');
signal rd_addr_in       : std_logic_vector(4 downto 0) := (others => '0');
signal rd_addr_out      : std_logic_vector(4 downto 0) := (others => '0');
signal op_class         : std_logic_vector(5 downto 0) := (others => '0');
signal funct3           : std_logic_vector(2 downto 0) := (others => '0');
signal funct7           : std_logic_vector(6 downto 0) := (others => '0');
signal a_sel            : std_logic := '0';
signal b_sel            : std_logic := '0';
signal cond_opcode      : std_logic_vector(2 downto 0) := (others => '0');
signal rs1_value        : std_logic_vector(31 downto 0) := (others => '0');
signal rs2_value        : std_logic_vector(31 downto 0) := (others => '0');
signal imm_se           : std_logic_vector(31 downto 0) := (others => '0');

signal branch_cond      : std_logic := '0';
signal alu_result       : std_logic_vector(31 downto 0) := (others => '0');

component instr_fetch
    port (
        clk             : in std_logic;
        pc_load_en      : in std_logic;
        pc_in           : in std_logic_vector(31 downto 0);
        next_pc         : out std_logic_vector(31 downto 0);
        curr_pc         : out std_logic_vector(31 downto 0);
        instr           : out std_logic_vector(31 downto 0));
end component;

component instr_decode
    port (
        clk             : in std_logic;
        instr           : in std_logic_vector(31 downto 0);
        rd_write_en     : in std_logic;
        rd_value        : in std_logic_vector(31 downto 0);
        rd_addr_in      : in std_logic_vector(4 downto 0);
        op_class        : out std_logic_vector(5 downto 0);
        funct3          : out std_logic_vector(2 downto 0);
        funct7          : out std_logic_vector(6 downto 0);
        a_sel           : out std_logic;
        b_sel           : out std_logic;
        cond_opcode     : out std_logic_vector(2 downto 0);
        rd_addr_out     : out std_logic_vector(4 downto 0);
        rs1_value       : out std_logic_vector(31 downto 0);
        rs2_value       : out std_logic_vector(31 downto 0);
        imm_se          : out std_logic_vector(31 downto 0));
end component;

component instr_exec
    port(
        a_sel           : in std_logic;
        b_sel           : in std_logic;
        rs1_value       : in std_logic_vector(31 downto 0);
        rs2_value       : in std_logic_vector(31 downto 0);
        imm_se          : in std_logic_vector(31 downto 0);
        curr_pc         : in std_logic_vector(31 downto 0);
        cond_opcode     : in std_logic_vector(2 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
```

```
        funct7          : in std_logic_vector(6 downto 0);
        op_class        : in std_logic_vector(5 downto 0);
        branch_cond     : out std_logic;
        alu_result      : out std_logic_vector(31 downto 0));
end component;
begin
    if_inst : instr_fetch
        port map (
            clk         => clk,
            pc_load_en  => pc_load_en,
            pc_in       => pc_in,
            next_pc     => next_pc,
            curr_pc     => curr_pc,
            instr       => instr);
    id_inst : instr_decode
        port map (
            clk         => clk,
            instr       => instr,
            rd_write_en => rd_write_en,
            rd_value    => rd_value,
            rd_addr_in  => rd_addr_in,
            op_class    => op_class,
            funct3      => funct3,
            funct7      => funct7,
            a_sel       => a_sel,
            b_sel       => b_sel,
            cond_opcode => cond_opcode,
            rd_addr_out => rd_addr_out,
            rs1_value   => rs1_value,
            rs2_value   => rs2_value,
            imm_se      => imm_se);
    ie_inst : instr_exec
        port map(
            a_sel       => a_sel,
            b_sel       => b_sel,
            rs1_value   => rs1_value,
            rs2_value   => rs2_value,
            imm_se      => imm_se,
            curr_pc  => curr_pc,
            cond_opcode => cond_opcode,
            funct3      => funct3,
            funct7      => funct7,
            op_class    => op_class,
            branch_cond => branch_cond,
            alu_result  => alu_result);

    process
    begin
        clk <= '0';
        wait for clk_period / 2;
        clk <= '1';
        wait for clk_period / 2;
    end process;

    pc_in       <= next_pc;
end Behavioral;
```

Data Memory [RV32I]

Source Code 5.11: Data Memory stage

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity data_memory is
    Port (
        clk             : in std_logic;
        op_class        : in std_logic_vector(5 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
        rs2_value       : in std_logic_vector(31 downto 0);
        alu_result      : in std_logic_vector(31 downto 0);

        mem_out         : out std_logic_vector(31 downto 0));
end data_memory;

architecture Behavioral of data_memory is
    signal mem_out_raw  : std_logic_vector(31 downto 0) := (others => '0');
    signal write_enable : std_logic_vector(3 downto 0)  := (others => '0');

    component data_mem is
        port(
            clka        : in std_logic;
            wea         : in std_logic_vector(3 downto 0);
            dina        : in std_logic_vector(31 downto 0);
            addra       : in std_logic_vector(11 downto 0);
            douta       : out std_logic_vector(31 downto 0));
    end component;
begin
    dm : data_mem
    port map(
        clka    => clk,
        wea     => write_enable, -- enable write only if it's a store instruction
        dina    => rs2_value,
        addra   => alu_result(13 downto 2),
        douta   => mem_out_raw);

    process(funct3, op_class, rs2_value)
    begin
        case funct3 & op_class is
            when "000" & "000100" => -- LB
                write_enable<= "0000";
                mem_out     <= std_logic_vector(resize(
                                signed(mem_out_raw(7 downto 0)), mem_out'length));
            when "000" & "001000" => --SB
                write_enable<= "0001";
                mem_out     <= (others => '0');
            when "001" & "000100" => -- LH
                write_enable<= "0000";
                mem_out     <= std_logic_vector(resize(
                                signed(mem_out_raw(15 downto 0)), mem_out'length));
            when "001" & "001000" => -- SH,
                write_enable<= "0011";
                mem_out     <= (others => '0');
            when "010" & "000100" => -- LW
                write_enable<= "0000";
                mem_out     <= mem_out_raw;
            when "010" & "001000" => -- SW
                write_enable<= "1111";
                mem_out     <= (others => '0');
            when "100" & "000100" => -- LBU
                write_enable<= "0000";
                mem_out     <= std_logic_vector(resize(
                                unsigned(mem_out_raw(7 downto 0)), mem_out'length));
            when "101" & "000100" => -- LHU
                write_enable<= "0000";
                mem_out     <= std_logic_vector(resize(
                                unsigned(mem_out_raw(15 downto 0)), mem_out'length));
            when others =>
                write_enable<= "0000";
                mem_out     <= (others => '0');
        end case;
    end process;
end Behavioral;
```

WriteBack [RV32I]

Source Code 5.12: Write Back stage

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity write_back is
    port (
        branch_cond     : in std_logic;
        op_class        : in std_logic_vector(5 downto 0);
        mem_out         : in std_logic_vector(31 downto 0);
        next_pc         : in std_logic_vector(31 downto 0);
        alu_result      : in std_logic_vector(31 downto 0);

        rd_write_en     : out std_logic;
        pc_out          : out std_logic_vector(31 downto 0);
        rd_value        : out std_logic_vector(31 downto 0)
    );
end write_back;

architecture Behavioral of write_back is
begin
    process(op_class,mem_out,alu_result,next_pc)
    begin
        case op_class is
            when "000010" | "000001" =>         -- OP - OP-IMM
                rd_value    <= alu_result;
                pc_out      <= next_pc;
                rd_write_en <= '1';
            when "000110" =>          -- LUI
                rd_value    <= alu_result;
                pc_out      <= next_pc;
                rd_write_en <= '1';
            when "000100" =>          -- LOAD
                rd_value    <= mem_out;
                pc_out      <= next_pc;
                rd_write_en <= '1';
            when "010000" =>          -- JUMP
                rd_value    <= next_pc;
                pc_out      <= std_logic_vector(signed(alu_result));
                rd_write_en <= '1';
            when "100000" =>          -- BRANCH
                rd_value    <= next_pc when branch_cond = '1' else (others => '0');
                pc_out      <= alu_result when branch_cond = '1' else next_pc;
                rd_write_en <= '1';
            when "010010" =>          -- AUIPC
                rd_value    <= std_logic_vector(signed(alu_result)+signed(next_pc));
                pc_out      <= alu_result;
                rd_write_en <= '1';
            when others  =>
                rd_value    <= (others => '0');
                rd_write_en <= '0';
                pc_out      <= next_pc;
        end case;
    end process;
end Behavioral;
```

Unpipelined Datapath Simulation

Source Code 5.13: Testbench with all components of the RV32I datapath

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity WB_testbench is
end WB_testbench;

architecture Behavioral of WB_testbench is
constant clk_period     : time := 100 ns;
```

```
signal pc_in            : std_logic_vector(31 downto 0) := (others => '0');
signal next_pc          : std_logic_vector(31 downto 0) := (others => '0');
signal curr_pc          : std_logic_vector(31 downto 0) := (others => '0');
signal instr            : std_logic_vector(31 downto 0) := (others => '0');
signal clk              : std_logic := '0';
signal pc_load_en       : std_logic := '1';

signal rd_write_en      : std_logic := '0';
signal rd_value         : std_logic_vector(31 downto 0) := (others => '0');
signal rd_addr_in       : std_logic_vector(4 downto 0) := (others => '0');
signal rd_addr_out      : std_logic_vector(4 downto 0) := (others => '0');
signal op_class         : std_logic_vector(5 downto 0) := (others => '0');
signal funct3           : std_logic_vector(2 downto 0) := (others => '0');
signal funct7           : std_logic_vector(6 downto 0) := (others => '0');
signal a_sel            : std_logic := '0';
signal b_sel            : std_logic := '0';
signal cond_opcode      : std_logic_vector(2 downto 0) := (others => '0');
signal rs1_value        : std_logic_vector(31 downto 0) := (others => '0');
signal rs2_value        : std_logic_vector(31 downto 0) := (others => '0');
signal imm_se           : std_logic_vector(31 downto 0) := (others => '0');

signal branch_cond      : std_logic := '0';
signal alu_result       : std_logic_vector(31 downto 0) := (others => '0');

signal mem_out          : std_logic_vector(31 downto 0) := (others => '0');
signal pc_out           : std_logic_vector(31 downto 0) := (others => '0');

component instr_fetch
    port (
        clk             : in std_logic;
        pc_load_en      : in std_logic;
        pc_in           : in std_logic_vector(31 downto 0);
        next_pc         : out std_logic_vector(31 downto 0);
        curr_pc         : out std_logic_vector(31 downto 0);
        instr           : out std_logic_vector(31 downto 0));
end component;

component instr_decode
    port (
        clk             : in std_logic;
        instr           : in std_logic_vector(31 downto 0);
        rd_write_en     : in std_logic;
        rd_value        : in std_logic_vector(31 downto 0);
        rd_addr_in      : in std_logic_vector(4 downto 0);
        op_class        : out std_logic_vector(5 downto 0);
        funct3          : out std_logic_vector(2 downto 0);
        funct7          : out std_logic_vector(6 downto 0);
        a_sel           : out std_logic;
        b_sel           : out std_logic;
        cond_opcode     : out std_logic_vector(2 downto 0);
        rd_addr_out     : out std_logic_vector(4 downto 0);
        rs1_value       : out std_logic_vector(31 downto 0);
        rs2_value       : out std_logic_vector(31 downto 0);
        imm_se          : out std_logic_vector(31 downto 0));
end component;

component instr_exec
    port(
        a_sel           : in std_logic;
        b_sel           : in std_logic;
        rs1_value       : in std_logic_vector(31 downto 0);
        rs2_value       : in std_logic_vector(31 downto 0);
        imm_se          : in std_logic_vector(31 downto 0);
        curr_pc         : in std_logic_vector(31 downto 0);
        cond_opcode     : in std_logic_vector(2 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
        funct7          : in std_logic_vector(6 downto 0);
        op_class        : in std_logic_vector(5 downto 0);
        branch_cond     : out std_logic;
        alu_result      : out std_logic_vector(31 downto 0));
end component;

component data_memory is
    Port (
        clk             : in std_logic;
        op_class        : in std_logic_vector(5 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
```

```
        rs2_value       : in std_logic_vector(31 downto 0);
        alu_result      : in std_logic_vector(31 downto 0);
        mem_out         : out std_logic_vector(31 downto 0));
end component;

component write_back is
    port (
        branch_cond     : in std_logic;
        op_class        : in std_logic_vector(5 downto 0);
        mem_out         : in std_logic_vector(31 downto 0);
        next_pc         : in std_logic_vector(31 downto 0);
        alu_result      : in std_logic_vector(31 downto 0);
        rd_write_en     : out std_logic;
        pc_out          : out std_logic_vector(31 downto 0);
        rd_value        : out std_logic_vector(31 downto 0));
end component;

begin
    if_inst : instr_fetch
        port map (
            clk             => clk,
            pc_load_en      => pc_load_en,
            pc_in           => pc_in,
            next_pc         => next_pc,
            curr_pc         => curr_pc,
            instr           => instr);
    id_inst : instr_decode
        port map (
            clk             => clk,
            instr           => instr,
            rd_write_en     => rd_write_en,
            rd_value        => rd_value,
            rd_addr_in      => rd_addr_in,
            op_class        => op_class,
            funct3          => funct3,
            funct7          => funct7,
            a_sel           => a_sel,
            b_sel           => b_sel,
            cond_opcode     => cond_opcode,
            rd_addr_out     => rd_addr_out,
            rs1_value       => rs1_value,
            rs2_value       => rs2_value,
            imm_se          => imm_se);
    ie_inst : instr_exec
        port map(
            a_sel           => a_sel,
            b_sel           => b_sel,
            rs1_value       => rs1_value,
            rs2_value       => rs2_value,
            imm_se          => imm_se,
            curr_pc         => curr_pc,
            cond_opcode     => cond_opcode,
            funct3          => funct3,
            funct7          => funct7,
            op_class        => op_class,
            branch_cond     => branch_cond,
            alu_result      => alu_result);
    mem : data_memory
        port map(
            clk             => clk,
            op_class        => op_class,
            funct3          => funct3,
            rs2_value       => rs2_value,
            alu_result      => alu_result,
            mem_out         => mem_out);
    wb : write_back
        port map(
            branch_cond     => branch_cond,
            op_class        => op_class,
            mem_out         => mem_out,
            next_pc         => next_pc,
            alu_result      => alu_result,
            rd_write_en     => rd_write_en,
            pc_out          => pc_out,
            rd_value        => rd_value);
    process
    begin
```

```
        clk <= '0';
        wait for clk_period / 2;
        clk <= '1';
        wait for clk_period / 2;
    end process;

    rd_addr_in  <= rd_addr_out;
    pc_in       <= pc_out(31 downto 0);
end Behavioral;
```

Appendix C: Code segments of the finalized datapath

Final Instruction Fetch entity description

Source Code 5.14: Instruction Fetch implemented for a pipelined architecture

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity instr_fetch_pipeline is
    port (
        clk         : in std_logic;
        pc_load_en  : in std_logic;
        pc_in       : in std_logic_vector(31 downto 0);
        pc_stall    : in std_logic;

        next_pc     : out std_logic_vector(31 downto 0);
        curr_pc     : out std_logic_vector(31 downto 0);
        instr       : out std_logic_vector(31 downto 0));
end instr_fetch_pipeline;

architecture Structural of instr_fetch_pipeline is
    signal pc_reg  : unsigned(31 downto 0) := (others => '0');

    component instruction_memory
    port(
        clka        : in std_logic;
        wea         : in std_logic;
        addra       : in std_logic_vector(11 downto 0);
        dina        : in std_logic_vector(31 downto 0);
        douta       : out std_logic_vector(31 downto 0));
    end component;
begin
    instr_mem: instruction_memory
    port map(
        clka        => clk,
        wea         => '0',
        dina        => (others => '0'),
        addra       => std_logic_vector(pc_reg(13 downto 2)),
        douta       => instr);

    process (clk)
    begin
        if rising_edge(clk) then
            if pc_stall = '1' then
                pc_reg <= pc_reg;
            elsif pc_load_en = '1' then
                pc_reg <= unsigned(pc_in);
            else
                pc_reg <= pc_reg + 4;
            end if;
        end if;
    end process;
    next_pc     <= std_logic_vector(pc_reg + 4);
    curr_pc     <= std_logic_vector(pc_reg);
end Structural;
```

Final Instruction Decode entity description

Source Code 5.15: Instruction Decode implemented for a pipelined architecture

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity instr_decode_pipeline is
    Port (
        clk         : in std_logic;
        instr       : in std_logic_vector(31 downto 0);

        -- Inputs from mem writeback

        rd_write_en : in std_logic;
        rd_value    : in std_logic_vector(31 downto 0);
```

```
        rd_addr_in  : in std_logic_vector(4 downto 0);

        -- Decoded instruction informations

        op_class    : out std_logic_vector(5 downto 0);
        funct3      : out std_logic_vector(2 downto 0);
        funct7      : out std_logic_vector(6 downto 0);
        a_sel       : out std_logic;
        b_sel       : out std_logic;
        cond_opcode : out std_logic_vector(2 downto 0);
        rd_addr_out : out std_logic_vector(4 downto 0);
        rs1_addr_out: out std_logic_vector(4 downto 0);
        rs2_addr_out: out std_logic_vector(4 downto 0);

        -- Data to be elaborated

        rs1_value   : out std_logic_vector(31 downto 0);
        rs2_value   : out std_logic_vector(31 downto 0);
        imm_se      : out std_logic_vector(31 downto 0);

        -- connections to memory reader
        addr_sel    : in std_logic_vector(13 downto 0);
        reg_out     : out std_logic_vector(31 downto 0)
    );
end instr_decode_pipeline;

architecture Structural of instr_decode_pipeline is
    signal rd_rs1_mux       : std_logic_vector(4 downto 0) := (others => '0');
    signal rs1_value_reg    : std_logic_vector(31 downto 0) := (others => '0');
    signal rs2_value_reg    : std_logic_vector(31 downto 0) := (others => '0');

    component register_file is
    port (
        clk         : in std_logic;
        da          : in std_logic_vector(4 downto 0);
        pda         : in std_logic_vector(4 downto 0);
        dina        : in std_logic_vector(4 downto 0);
        din         : in std_logic_vector(31 downto 0);
        we          : in std_logic;

        rso         : out std_logic_vector(31 downto 0);
        prso        : out std_logic_vector(31 downto 0);

        addr_sel    : in std_logic_vector(13 downto 0);
        reg_out     : out std_logic_vector(31 downto 0)
    );
    end component;

    component decoder is
    port(
        instr       : in std_logic_vector(31 downto 0);
        op_class    : out std_logic_vector(5 downto 0);
        funct3      : out std_logic_vector(2 downto 0);
        funct7      : out std_logic_vector(6 downto 0);
        a_sel       : out std_logic;
        b_sel       : out std_logic;
        cond_opcode : out std_logic_vector(2 downto 0);
        imm_se      : out std_logic_vector(31 downto 0));
    end component;
begin
    reg : register_file
    port map(
        clk         => clk,
        da          => instr(11 downto 7) when op_class = "000110"  else
                       instr(19 downto 15),
        pda         => instr(24 downto 20),
        dina        => rd_addr_in,
        din         => rd_value,
        we          => rd_write_en,
        rso         => rs1_value_reg,
        prso        => rs2_value_reg,
        addr_sel    => addr_sel,
        reg_out     => reg_out
        );
    dec : decoder
    port map(
        instr       => instr,
```

```
        op_class    => op_class,
        funct3      => funct3,
        funct7      => funct7,
        a_sel       => a_sel,
        b_sel       => b_sel,
        cond_opcode => cond_opcode,
        imm_se      => imm_se);

    rd_addr_out <= instr(11 downto 7);
    rs1_addr_out<= instr(19 downto 15);
    rs2_addr_out<= instr(24 downto 20);

    rs1_value <= rd_value when (instr(19 downto 15) = rd_addr_in and rd_write_en = '1')
                 else rs1_value_reg;
    rs2_value <= rd_value when (instr(24 downto 20) = rd_addr_in and rd_write_en = '1')
                 else rs1_value_reg;
end Structural;
```

Final Instruction Execute entity description

Source Code 5.16: Instruction Execute implemented for a pipelined architecture

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity instr_exec_pipeline is
    Port (
        a_sel           : in std_logic;
        b_sel           : in std_logic;
        rs1_value_in    : in std_logic_vector(31 downto 0);
        rs2_value_in    : in std_logic_vector(31 downto 0);
        imm_se          : in std_logic_vector(31 downto 0);
        curr_pc         : in std_logic_vector(31 downto 0);
        cond_opcode     : in std_logic_vector(2 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
        funct7          : in std_logic_vector(6 downto 0);
        op_class        : in std_logic_vector(5 downto 0);

        -- operand forwarding inputs
        rs1_addr_in     : in std_logic_vector(4 downto 0);
        rs2_addr_in     : in std_logic_vector(4 downto 0);
        rd_addr_dm      : in std_logic_vector(4 downto 0);
        alu_result_dm   : in std_logic_vector(31 downto 0);
        rd_addr_wb      : in std_logic_vector(4 downto 0);
        alu_result_wb   : in std_logic_vector(31 downto 0);

        branch_cond     : out std_logic;
        rs2_value_out   : out std_logic_vector(31 downto 0);
        alu_result      : out std_logic_vector(31 downto 0)
     );
end instr_exec_pipeline;

architecture Structural of instr_exec_pipeline is
    signal first_operand    : std_logic_vector(31 downto 0) := (others => '0');
    signal second_operand   : std_logic_vector(31 downto 0) := (others => '0');
    signal rs1_fw           : std_logic_vector(31 downto 0) := (others => '0');
    signal rs2_fw           : std_logic_vector(31 downto 0) := (others => '0');
    signal branch_cond_buf  : std_logic;

    component ALU is
        port (
            first_operand   : in std_logic_vector(31 downto 0);
            second_operand  : in std_logic_vector(31 downto 0);
            funct3          : in std_logic_vector(2 downto 0);
            funct7          : in std_logic_vector(6 downto 0);
            op_class        : in std_logic_vector(5 downto 0);

            alu_result      : out std_logic_vector(31 downto 0));
    end component;

    component comparator is
        port (
            first_operand   : in std_logic_vector(31 downto 0);
            second_operand  : in std_logic_vector(31 downto 0);
            cond_opcode     : in std_logic_vector(2 downto 0);
```

```
            branch_cond     : out std_logic);
    end component;
begin
    comp : comparator
    port map(
        first_operand   => rs1_value_in,
        second_operand  => rs2_value_in,
        cond_opcode     => cond_opcode,
        branch_cond     => branch_cond_buf);

    alu_1 : ALU
    port map(
        first_operand   => first_operand,
        second_operand  => second_operand,
        funct3          => funct3,
        funct7          => funct7,
        op_class        => op_class,

        alu_result      => alu_result);

    rs1_fw          <= alu_result_wb when (rd_addr_wb = rs1_addr_in)
                       else alu_result_dm when (rd_addr_dm = rs1_addr_in)
                       else rs1_value_in;
    rs2_fw          <= alu_result_wb when (rd_addr_wb = rs2_addr_in)
                       else alu_result_dm when (rd_addr_dm = rs2_addr_in)
                       else rs2_value_in;
    first_operand   <= rs1_fw when a_sel = '1' else curr_pc;
    second_operand  <= rs2_fw when b_sel = '1' else imm_se;
    rs2_value_out   <= rs2_fw;

    branch_cond     <= branch_cond_buf when op_class(5) else '0';
end Structural;
```

Final Data Memory stage entity description

Source Code 5.17: Data Memory implemented for a pipelined architecture

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity data_memory_pipeline is
    Port (
        clk             : in std_logic;
        op_class        : in std_logic_vector(5 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
        rs2_value       : in std_logic_vector(31 downto 0);
        alu_result      : in std_logic_vector(31 downto 0);

        read_addr       : in std_logic_vector(13 downto 0);

        read_data_out   : out std_logic_vector(31 downto 0);
        mem_out         : out std_logic_vector(31 downto 0));
end data_memory_pipeline;

architecture Behavioral of data_memory_pipeline is
    signal mem_out_raw  : std_logic_vector(31 downto 0) := (others => '0');
    signal write_enable : std_logic_vector(3 downto 0)  := (others => '0');

    component data_mem is
        port(
            clka        : in std_logic;
            wea         : in std_logic_vector(3 downto 0);
            dina        : in std_logic_vector(31 downto 0);
            addra       : in std_logic_vector(11 downto 0);
            douta       : out std_logic_vector(31 downto 0);

            clkb        : in std_logic;
            web         : in std_logic_vector(3 downto 0);
            dinb        : in std_logic_vector(31 downto 0);
            addrb       : in std_logic_vector(11 downto 0);
```

```
            doutb       : out std_logic_vector(31 downto 0));
    end component;
begin
    dm : data_mem
    port map(
        clka    => clk,
        wea     => write_enable, -- enable write only if it's a store instruction
        dina    => rs2_value,
        addra   => alu_result(13 downto 2),
        douta   => mem_out_raw,

        clkb    => clk,
        web     => (others => '0'),
        dinb    => (others => '0'),
        addrb   => read_addr(13 downto 2),
        doutb   => read_data_out
        );

    process(all)
    begin
        case funct3 & op_class is
            when "000" & "000100" => -- LB
                write_enable<= "0000";
                mem_out     <= std_logic_vector(resize(
                                signed(mem_out_raw(7 downto 0)), mem_out'length));
            when "000" & "001000" => -- SB
                write_enable<= "0001";
                mem_out     <= (others => '0');
            when "001" & "000100" => -- LH
                write_enable<= "0000";
                mem_out     <= std_logic_vector(resize(
                                signed(mem_out_raw(15 downto 0)), mem_out'length));
            when "001" & "001000" => -- SH,
                write_enable<= "0011";
                mem_out     <= (others => '0');
            when "010" & "000100" => -- LW
                write_enable<= "0000";
                mem_out     <= mem_out_raw;
            when "010" & "001000" => -- SW
                write_enable<= "1111";
                mem_out     <= (others => '0');
            when "100" & "000100" => -- LBU
                write_enable<= "0000";
                mem_out     <= std_logic_vector(resize(
                                unsigned(mem_out_raw(7 downto 0)), mem_out'length));
            when "101" & "000100" => -- LHU
                write_enable<= "0000";
                mem_out     <= std_logic_vector(resize(
                                unsigned(mem_out_raw(15 downto 0)), mem_out'length));
            when others =>
                write_enable<= "0000";
                mem_out     <= (others => '0');
        end case;
    end process;
end Behavioral;
```

IF/ID register

Source Code 5.18: IF/ID register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity IF_ID is
    Port (
        clk         : in std_logic;
        flush       : in std_logic;
        curr_pc_if  : in std_logic_vector(31 downto 0);
        next_pc_if  : in std_logic_vector(31 downto 0);
        instr_if    : in std_logic_vector(31 downto 0);

        curr_pc_id  : out std_logic_vector(31 downto 0);
```

```
        next_pc_id  : out std_logic_vector(31 downto 0);
        instr_id    : out std_logic_vector(31 downto 0)
    );
end IF_ID;

architecture Behavioral of IF_ID is
    signal curr_pc_reg  : std_logic_vector(31 downto 0) := (others => '0');
    signal next_pc_reg  : std_logic_vector(31 downto 0) := (others => '0');
    signal instr_reg    : std_logic_vector(31 downto 0) := (others => '0');
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if flush = '1' then
                curr_pc_reg <= (others => '0');
                next_pc_reg <= (others => '0');
                instr_reg   <= (others => '0');
            else
                curr_pc_reg <= curr_pc_if;
                next_pc_reg <= next_pc_if;
                instr_reg   <= instr_if;
            end if;
        end if;
    end process;

    curr_pc_id  <= curr_pc_reg;
    next_pc_id  <= next_pc_reg;
    instr_id    <= instr_reg;
end Behavioral;
```

ID/IE register

Source Code 5.19: ID/IE register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity ID_IE is
    Port (
        clk             : in std_logic;
        flush           : in std_logic;
        curr_pc_id      : in std_logic_vector(31 downto 0);
        next_pc_id      : in std_logic_vector(31 downto 0);
        op_class_id     : in std_logic_vector(5 downto 0);
        funct3_id       : in std_logic_vector(2 downto 0);
        funct7_id       : in std_logic_vector(6 downto 0);
        a_sel_id        : in std_logic;
        b_sel_id        : in std_logic;
        imm_se_id       : in std_logic_vector(31 downto 0);
        rs1_value_id    : in std_logic_vector(31 downto 0);
        rs2_value_id    : in std_logic_vector(31 downto 0);
        rd_addr_id      : in std_logic_vector(4 downto 0);
        rs1_addr_id     : in std_logic_vector(4 downto 0);
        rs2_addr_id     : in std_logic_vector(4 downto 0);
        cond_opcode_id  : in std_logic_vector(2 downto 0);

        curr_pc_ie      : out std_logic_vector(31 downto 0);
        next_pc_ie      : out std_logic_vector(31 downto 0);
        op_class_ie     : out std_logic_vector(5 downto 0);
        funct3_ie       : out std_logic_vector(2 downto 0);
        funct7_ie       : out std_logic_vector(6 downto 0);
        a_sel_ie        : out std_logic;
        b_sel_ie        : out std_logic;
        imm_se_ie       : out std_logic_vector(31 downto 0);
        rs1_value_ie    : out std_logic_vector(31 downto 0);
        rs2_value_ie    : out std_logic_vector(31 downto 0);
        rd_addr_ie      : out std_logic_vector(4 downto 0);
        rs1_addr_ie     : out std_logic_vector(4 downto 0);
        rs2_addr_ie     : out std_logic_vector(4 downto 0);
        cond_opcode_ie  : out std_logic_vector(2 downto 0)
    );
end ID_IE;

architecture Behavioral of ID_IE is
signal curr_pc_reg      : std_logic_vector(31 downto 0) := (others => '0');
```

```
signal next_pc_reg      : std_logic_vector(31 downto 0) := (others => '0');
signal op_class_reg     : std_logic_vector(5 downto 0) := (others => '0');
signal funct3_reg       : std_logic_vector(2 downto 0) := (others => '0');
signal funct7_reg       : std_logic_vector(6 downto 0) := (others => '0');
signal a_sel_reg        : std_logic := '0';
signal b_sel_reg        : std_logic := '0';
signal imm_se_reg       : std_logic_vector(31 downto 0) := (others => '0');
signal rs1_value_reg    : std_logic_vector(31 downto 0) := (others => '0');
signal rs2_value_reg    : std_logic_vector(31 downto 0) := (others => '0');
signal rd_addr_reg      : std_logic_vector(4 downto 0) := (others => '0');
signal rs1_addr_reg     : std_logic_vector(4 downto 0) := (others => '0');
signal rs2_addr_reg     : std_logic_vector(4 downto 0) := (others => '0');
signal cond_opcode_reg  : std_logic_vector(2 downto 0) := (others => '0');
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if flush = '1' then
                curr_pc_reg         <= (others => '0');
                next_pc_reg         <= (others => '0');
                op_class_reg        <= (others => '0');
                funct3_reg          <= (others => '0');
                funct7_reg          <= (others => '0');
                a_sel_reg           <= '0';
                b_sel_reg           <= '0';
                imm_se_reg          <= (others => '0');
                rs1_value_reg       <= (others => '0');
                rs2_value_reg       <= (others => '0');
                rd_addr_reg         <= (others => '0');
                rs1_addr_reg        <= (others => '0');
                rs2_addr_reg        <= (others => '0');
                cond_opcode_reg     <= (others => '0');
            else
                curr_pc_reg         <= curr_pc_id;
                next_pc_reg         <= next_pc_id;
                op_class_reg        <= op_class_id;
                funct3_reg          <= funct3_id;
                funct7_reg          <= funct7_id;
                a_sel_reg           <= a_sel_id;
                b_sel_reg           <= b_sel_id;
                imm_se_reg          <= imm_se_id;
                rs1_value_reg       <= rs1_value_id;
                rs2_value_reg       <= rs2_value_id;
                rd_addr_reg         <= rd_addr_id;
                rs1_addr_reg        <= rs1_addr_id;
                rs2_addr_reg        <= rs2_addr_id;
                cond_opcode_reg     <= cond_opcode_id;
            end if;
        end if;
    end process;

    curr_pc_ie          <= curr_pc_reg;
    next_pc_ie          <= next_pc_reg;
    op_class_ie         <= op_class_reg;
    funct3_ie           <= funct3_reg;
    funct7_ie           <= funct7_reg;
    a_sel_ie            <= a_sel_reg;
    b_sel_ie            <= b_sel_reg;
    imm_se_ie           <= imm_se_reg;
    rs1_value_ie        <= rs1_value_reg;
    rs2_value_ie        <= rs2_value_reg;
    rd_addr_ie          <= rd_addr_reg;
    rs1_addr_ie         <= rs1_addr_reg;
    rs2_addr_ie         <= rs2_addr_reg;
    cond_opcode_ie      <= cond_opcode_reg;
end Behavioral;
```

IE/DM register

Source Code 5.20: IE/DM register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
entity IE_DM is
    Port (
```

```
        clk             : in std_logic;
        flush           : in std_logic;
        curr_pc_ie      : in std_logic_vector(31 downto 0);
        next_pc_ie      : in std_logic_vector(31 downto 0);
        op_class_ie     : in std_logic_vector(5 downto 0);
        funct3_ie       : in std_logic_vector(2 downto 0);
        alu_result_ie   : in std_logic_vector(31 downto 0);
        rs2_value_ie    : in std_logic_vector(31 downto 0);
        rd_addr_ie      : in std_logic_vector(4 downto 0);
        branch_cond_ie  : in std_logic;

        curr_pc_dm      : out std_logic_vector(31 downto 0);
        next_pc_dm      : out std_logic_vector(31 downto 0);
        op_class_dm     : out std_logic_vector(5 downto 0);
        funct3_dm       : out std_logic_vector(2 downto 0);
        alu_result_dm   : out std_logic_vector(31 downto 0);
        rs2_value_dm    : out std_logic_vector(31 downto 0);
        rd_addr_dm      : out std_logic_vector(4 downto 0);
        branch_cond_dm  : out std_logic
    );
end IE_DM;

architecture Behavioral of IE_DM is
    signal curr_pc_reg      : std_logic_vector(31 downto 0) := (others => '0');
    signal next_pc_reg      : std_logic_vector(31 downto 0) := (others => '0');
    signal op_class_reg     : std_logic_vector(5 downto 0)  := (others => '0');
    signal funct3_reg       : std_logic_vector(2 downto 0)  := (others => '0');
    signal alu_result_reg   : std_logic_vector(31 downto 0) := (others => '0');
    signal rs2_value_reg    : std_logic_vector(31 downto 0) := (others => '0');
    signal rd_addr_reg      : std_logic_vector(4 downto 0)  := (others => '0');
    signal branch_cond_reg  : std_logic := '0';
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if flush = '1' then
                curr_pc_reg         <= (others => '0');
                next_pc_reg         <= (others => '0');
                op_class_reg        <= (others => '0');
                funct3_reg          <= (others => '0');
                alu_result_reg      <= (others => '0');
                rs2_value_reg       <= (others => '0');
                rd_addr_reg         <= (others => '0');
                branch_cond_reg     <= '0';
            else
                curr_pc_reg         <= curr_pc_ie;
                next_pc_reg         <= next_pc_ie;
                op_class_reg        <= op_class_ie;
                funct3_reg          <= funct3_ie;
                alu_result_reg      <= alu_result_ie;
                rs2_value_reg       <= rs2_value_ie;
                rd_addr_reg         <= rd_addr_ie;
                branch_cond_reg     <= branch_cond_ie;
            end if;

        end if;
    end process;
    curr_pc_dm          <= curr_pc_reg;
    next_pc_dm          <= next_pc_reg;
    op_class_dm         <= op_class_reg;
    funct3_dm           <= funct3_reg;
    alu_result_dm       <= alu_result_reg;
    rs2_value_dm        <= rs2_value_reg;
    rd_addr_dm          <= rd_addr_reg;
    branch_cond_dm      <= branch_cond_reg;
end Behavioral;
```

DM/WB register

Source Code 5.21: DM/WB register

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity DM_WB is
```

```
    Port (
        clk             : in std_logic;
        next_pc_dm      : in std_logic_vector(31 downto 0);
        mem_out_dm      : in std_logic_vector(31 downto 0);
        alu_result_dm   : in std_logic_vector(31 downto 0);
        rd_addr_dm      : in std_logic_vector(4 downto 0);
        branch_cond_dm  : in std_logic;
        op_class_dm     : in std_logic_vector(5 downto 0);

        next_pc_wb      : out std_logic_vector(31 downto 0);
        mem_out_wb      : out std_logic_vector(31 downto 0);
        alu_result_wb   : out std_logic_vector(31 downto 0);
        rd_addr_wb      : out std_logic_vector(4 downto 0);
        branch_cond_wb  : out std_logic;
        op_class_wb     : out std_logic_vector(5 downto 0)
    );
end DM_WB;

architecture Behavioral of DM_WB is
signal next_pc_reg      : std_logic_vector(31 downto 0) := (others => '0');
signal mem_out_reg      : std_logic_vector(31 downto 0) := (others => '0');
signal alu_result_reg   : std_logic_vector(31 downto 0) := (others => '0');
signal rd_addr_reg      : std_logic_vector(4 downto 0) := (others => '0');
signal branch_cond_reg  : std_logic := '0';
signal op_class_reg     : std_logic_vector(5 downto 0):= (others => '0');
begin
    process(clk)
    begin
        if rising_edge(clk) then
            next_pc_reg     <= next_pc_dm;
            mem_out_reg     <= mem_out_dm;
            alu_result_reg  <= alu_result_dm;
            rd_addr_reg     <= rd_addr_dm;
            branch_cond_reg <= branch_cond_dm;
            op_class_reg    <= op_class_dm;
        end if;
    end process;
    next_pc_wb   <= next_pc_reg;
    mem_out_wb      <= mem_out_reg;
    alu_result_wb   <= alu_result_reg;
    rd_addr_wb      <= rd_addr_reg;
    branch_cond_wb  <= branch_cond_reg;
    op_class_wb     <= op_class_reg;
end Behavioral;
```

Pipelined architecture (Structural definition)

Source Code 5.22: Structural definition of a pipelined datapath

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity datapath is
    port(
        clk         : in std_logic;
        sw_i        : in std_logic_vector(15 downto 0);
        btnc_i      : in std_logic;
        disp_seg_o  : out std_logic_vector(7 downto 0);
        disp_an_o   : out std_logic_vector(7 downto 0);
        rgb2_green_o: out std_logic;
        rgb2_red_o  : out std_logic
    );
end datapath;

architecture Structural of datapath is

signal next_pc_if       : std_logic_vector(31 downto 0) := (others => '0');
signal curr_pc_if       : std_logic_vector(31 downto 0) := (others => '0');
signal instr_if         : std_logic_vector(31 downto 0) := (others => '0');
signal instr_id         : std_logic_vector(31 downto 0) := (others => '0');

signal rd_write_en_wb   : std_logic := '0';
signal rd_value_wb      : std_logic_vector(31 downto 0) := (others => '0');
signal rd_addr_in_wb    : std_logic_vector(4 downto 0) := (others => '0');
```

```
signal flush_if_id      : std_logic := '0';
signal pc_stall         : std_logic := '0';

signal rd_addr_id       : std_logic_vector(4 downto 0) := (others => '0');
signal rs1_addr_id      : std_logic_vector(4 downto 0) := (others => '0');
signal rs2_addr_id      : std_logic_vector(4 downto 0) := (others => '0');
signal next_pc_id       : std_logic_vector(31 downto 0) := (others => '0');
signal curr_pc_id       : std_logic_vector(31 downto 0) := (others => '0');
signal op_class_id      : std_logic_vector(5 downto 0) := (others => '0');
signal funct3_id        : std_logic_vector(2 downto 0) := (others => '0');
signal funct7_id        : std_logic_vector(6 downto 0) := (others => '0');
signal a_sel_id         : std_logic := '0';
signal b_sel_id         : std_logic := '0';
signal cond_opcode_id   : std_logic_vector(2 downto 0) := (others => '0');
signal rs1_value_id     : std_logic_vector(31 downto 0) := (others => '0');
signal rs2_value_id     : std_logic_vector(31 downto 0) := (others => '0');
signal imm_se_id        : std_logic_vector(31 downto 0) := (others => '0');
signal rd_addr_ie       : std_logic_vector(4 downto 0) := (others => '0');
signal rs1_addr_ie      : std_logic_vector(4 downto 0) := (others => '0');
signal rs2_addr_ie      : std_logic_vector(4 downto 0) := (others => '0');
signal next_pc_ie       : std_logic_vector(31 downto 0) := (others => '0');
signal curr_pc_ie       : std_logic_vector(31 downto 0) := (others => '0');
signal op_class_ie      : std_logic_vector(5 downto 0) := (others => '0');
signal funct3_ie        : std_logic_vector(2 downto 0) := (others => '0');
signal funct7_ie        : std_logic_vector(6 downto 0) := (others => '0');
signal a_sel_ie         : std_logic := '0';
signal b_sel_ie         : std_logic := '0';
signal cond_opcode_ie   : std_logic_vector(2 downto 0) := (others => '0');
signal rs1_value_ie     : std_logic_vector(31 downto 0) := (others => '0');
signal rs2_value_ie     : std_logic_vector(31 downto 0) := (others => '0');
signal imm_se_ie        : std_logic_vector(31 downto 0) := (others => '0');

signal branch_cond_ie   : std_logic := '0';
signal alu_result_ie    : std_logic_vector(31 downto 0) := (others => '0');
signal rs2_value_out    : std_logic_vector(31 downto 0) := (others => '0');
signal branch_cond_dm   : std_logic := '0';
signal alu_result_dm    : std_logic_vector(31 downto 0) := (others => '0');
signal next_pc_dm       : std_logic_vector(31 downto 0) := (others => '0');
signal curr_pc_dm       : std_logic_vector(31 downto 0) := (others => '0');
signal op_class_dm      : std_logic_vector(5 downto 0) := (others => '0');
signal rs2_value_dm     : std_logic_vector(31 downto 0) := (others => '0');
signal funct3_dm        : std_logic_vector(2 downto 0) := (others => '0');
signal rd_addr_dm       : std_logic_vector(4 downto 0) := (others => '0');

signal mem_out_dm       : std_logic_vector(31 downto 0) := (others => '0');
signal next_pc_wb       : std_logic_vector(31 downto 0) := (others => '0');
signal mem_out_wb       : std_logic_vector(31 downto 0) := (others => '0');
signal alu_result_wb    : std_logic_vector(31 downto 0) := (others => '0');
signal rd_addr_wb       : std_logic_vector(4 downto 0) := (others => '0');
signal branch_cond_wb   : std_logic := '0';
signal op_class_wb      : std_logic_vector(5 downto 0) := (others => '0');

signal pc_out_wb        : std_logic_vector(31 downto 0) := (others => '0');
signal pc_load_en_wb    : std_logic := '0';

signal data_read_dm_sig : std_logic_vector(31 downto 0) := (others => '0');
signal data_read_reg_sig: std_logic_vector(31 downto 0) := (others => '0');
signal addr_sel_sig     : std_logic_vector(13 downto 0) := (others => '0');


component instr_fetch_pipeline
    port (
        clk             : in std_logic;
        pc_load_en      : in std_logic;
        pc_in           : in std_logic_vector(31 downto 0);
        pc_stall        : in std_logic;
        next_pc         : out std_logic_vector(31 downto 0);
        curr_pc         : out std_logic_vector(31 downto 0);
        instr           : out std_logic_vector(31 downto 0));
end component;

component IF_ID
    port (
        clk             : in std_logic;
        flush           : in std_logic;
        curr_pc_if      : in std_logic_vector(31 downto 0);
        next_pc_if      : in std_logic_vector(31 downto 0);
```

```
        instr_if        : in std_logic_vector(31 downto 0);
        curr_pc_id      : out std_logic_vector(31 downto 0);
        next_pc_id      : out std_logic_vector(31 downto 0);
        instr_id        : out std_logic_vector(31 downto 0)
    );
end component;

component instr_decode_pipeline
    port (
        clk             : in std_logic;
        instr           : in std_logic_vector(31 downto 0);
        rd_write_en     : in std_logic;
        rd_value        : in std_logic_vector(31 downto 0);
        rd_addr_in      : in std_logic_vector(4 downto 0);
        op_class        : out std_logic_vector(5 downto 0);
        funct3          : out std_logic_vector(2 downto 0);
        funct7          : out std_logic_vector(6 downto 0);
        a_sel           : out std_logic;
        b_sel           : out std_logic;
        cond_opcode     : out std_logic_vector(2 downto 0);
        rd_addr_out     : out std_logic_vector(4 downto 0);
        rs1_addr_out    : out std_logic_vector(4 downto 0);
        rs2_addr_out    : out std_logic_vector(4 downto 0);
        rs1_value       : out std_logic_vector(31 downto 0);
        rs2_value       : out std_logic_vector(31 downto 0);
        imm_se          : out std_logic_vector(31 downto 0);
        addr_sel        : in std_logic_vector(13 downto 0);
        reg_out         : out std_logic_vector(31 downto 0));
end component;

component ID_IE
    Port (
        clk             : in std_logic;
        flush           : in std_logic;
        curr_pc_id      : in std_logic_vector(31 downto 0);
        next_pc_id      : in std_logic_vector(31 downto 0);
        op_class_id     : in std_logic_vector(5 downto 0);
        funct3_id       : in std_logic_vector(2 downto 0);
        funct7_id       : in std_logic_vector(6 downto 0);
        a_sel_id        : in std_logic;
        b_sel_id        : in std_logic;
        imm_se_id       : in std_logic_vector(31 downto 0);
        rs1_value_id    : in std_logic_vector(31 downto 0);
        rs2_value_id    : in std_logic_vector(31 downto 0);
        rd_addr_id      : in std_logic_vector(4 downto 0);
        rs1_addr_id     : in std_logic_vector(4 downto 0);
        rs2_addr_id     : in std_logic_vector(4 downto 0);
        cond_opcode_id  : in std_logic_vector(2 downto 0);

        curr_pc_ie      : out std_logic_vector(31 downto 0);
        next_pc_ie      : out std_logic_vector(31 downto 0);
        op_class_ie     : out std_logic_vector(5 downto 0);
        funct3_ie       : out std_logic_vector(2 downto 0);
        funct7_ie       : out std_logic_vector(6 downto 0);
        a_sel_ie        : out std_logic;
        b_sel_ie        : out std_logic;
        imm_se_ie       : out std_logic_vector(31 downto 0);
        rs1_value_ie    : out std_logic_vector(31 downto 0);
        rs2_value_ie    : out std_logic_vector(31 downto 0);
        rd_addr_ie      : out std_logic_vector(4 downto 0);
        rs1_addr_ie     : out std_logic_vector(4 downto 0);
        rs2_addr_ie     : out std_logic_vector(4 downto 0);
        cond_opcode_ie  : out std_logic_vector(2 downto 0));
end component;

component instr_exec_pipeline
    port(
        a_sel           : in std_logic;
        b_sel           : in std_logic;
        rs1_value_in    : in std_logic_vector(31 downto 0);
        rs2_value_in    : in std_logic_vector(31 downto 0);
        imm_se          : in std_logic_vector(31 downto 0);
        curr_pc         : in std_logic_vector(31 downto 0);
        cond_opcode     : in std_logic_vector(2 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
        funct7          : in std_logic_vector(6 downto 0);
        op_class        : in std_logic_vector(5 downto 0);
```

```
        rs1_addr_in     : in std_logic_vector(4 downto 0);
        rs2_addr_in     : in std_logic_vector(4 downto 0);
        rd_addr_dm      : in std_logic_vector(4 downto 0);
        alu_result_dm   : in std_logic_vector(31 downto 0);
        rd_addr_wb      : in std_logic_vector(4 downto 0);
        alu_result_wb   : in std_logic_vector(31 downto 0);
        branch_cond     : out std_logic;
        rs2_value_out   : out std_logic_vector(31 downto 0);
        alu_result      : out std_logic_vector(31 downto 0));
end component;

component IE_DM
    Port (
        clk             : in std_logic;
        flush           : in std_logic;
        curr_pc_ie      : in std_logic_vector(31 downto 0);
        next_pc_ie      : in std_logic_vector(31 downto 0);
        op_class_ie     : in std_logic_vector(5 downto 0);
        funct3_ie       : in std_logic_vector(2 downto 0);
        alu_result_ie   : in std_logic_vector(31 downto 0);
        rs2_value_ie    : in std_logic_vector(31 downto 0);
        rd_addr_ie      : in std_logic_vector(4 downto 0);
        branch_cond_ie  : in std_logic;
        curr_pc_dm      : out std_logic_vector(31 downto 0);
        next_pc_dm      : out std_logic_vector(31 downto 0);
        op_class_dm     : out std_logic_vector(5 downto 0);
        funct3_dm       : out std_logic_vector(2 downto 0);
        alu_result_dm   : out std_logic_vector(31 downto 0);
        rs2_value_dm    : out std_logic_vector(31 downto 0);
        rd_addr_dm      : out std_logic_vector(4 downto 0);
        branch_cond_dm  : out std_logic);
end component;

component data_memory_pipeline is
    Port (
        clk             : in std_logic;
        op_class        : in std_logic_vector(5 downto 0);
        funct3          : in std_logic_vector(2 downto 0);
        rs2_value       : in std_logic_vector(31 downto 0);
        alu_result      : in std_logic_vector(31 downto 0);
        read_addr       : in std_logic_vector(13 downto 0);
        read_data_out   : out std_logic_vector(31 downto 0);
        mem_out         : out std_logic_vector(31 downto 0));
end component;

component DM_WB is
    Port (
        clk             : in std_logic;
        next_pc_dm      : in std_logic_vector(31 downto 0);
        mem_out_dm      : in std_logic_vector(31 downto 0);
        alu_result_dm   : in std_logic_vector(31 downto 0);
        rd_addr_dm      : in std_logic_vector(4 downto 0);
        branch_cond_dm  : in std_logic;
        op_class_dm     : in std_logic_vector(5 downto 0);
        next_pc_wb      : out std_logic_vector(31 downto 0);
        mem_out_wb      : out std_logic_vector(31 downto 0);
        alu_result_wb   : out std_logic_vector(31 downto 0);
        rd_addr_wb      : out std_logic_vector(4 downto 0);
        branch_cond_wb  : out std_logic;
        op_class_wb     : out std_logic_vector(5 downto 0));
end component;

component write_back_pipeline is
    port (
        branch_cond     : in std_logic;
        op_class        : in std_logic_vector(5 downto 0);
        mem_out         : in std_logic_vector(31 downto 0);
        next_pc         : in std_logic_vector(31 downto 0);
        alu_result      : in std_logic_vector(31 downto 0);
        rd_write_en     : out std_logic;
        pc_load_en      : out std_logic;
        pc_out          : out std_logic_vector(31 downto 0);
        rd_value        : out std_logic_vector(31 downto 0));
end component;

component memory_reader is
    port (
```

```
        clk             : in std_logic;
        data_in_dm      : in std_logic_vector(31 downto 0);
        data_in_reg     : in std_logic_vector(31 downto 0);
        sw_i            : in std_logic_vector(15 downto 0);
        sw_reg_mem      : in std_logic;

        disp_seg        : out std_logic_vector(7 downto 0);
        an              : out std_logic_vector(7 downto 0);
        led_green       : out std_logic;
        led_red         : out std_logic;
        addr_sel        : out std_logic_vector(13 downto 0)
    );
end component;


begin
    if_inst : instr_fetch_pipeline
        port map (
            clk             => clk,
            pc_load_en      => pc_load_en_wb,
            pc_in           => pc_out_wb,
            pc_stall        => (branch_cond_ie or branch_cond_dm)
                                or (op_class_ie(4) or op_class_dm(4)),
            next_pc         => next_pc_if,
            curr_pc         => curr_pc_if,
            instr           => instr_if);
    reg_if_id : IF_ID
        port map(
            clk             => clk,
            flush           => (branch_cond_ie or branch_cond_dm)
                                or (op_class_ie(4) or op_class_dm(4)),
            curr_pc_if      => curr_pc_if,
            next_pc_if      => next_pc_if,
            instr_if        => instr_if,
            curr_pc_id      => curr_pc_id,
            next_pc_id      => next_pc_id,
            instr_id        => instr_id);
    id_inst : instr_decode_pipeline
        port map (
            clk             => clk,
            instr           => instr_id,
            rd_write_en     => rd_write_en_wb,
            rd_value        => rd_value_wb,
            rd_addr_in      => rd_addr_wb,
            op_class        => op_class_id,
            funct3          => funct3_id,
            funct7          => funct7_id,
            a_sel           => a_sel_id,
            b_sel           => b_sel_id,
            cond_opcode     => cond_opcode_id,
            rd_addr_out     => rd_addr_id,
            rs1_addr_out    => rs1_addr_id,
            rs2_addr_out    => rs2_addr_id,
            rs1_value       => rs1_value_id,
            rs2_value       => rs2_value_id,
            imm_se          => imm_se_id,
            addr_sel        => addr_sel_sig,
            reg_out         => data_read_reg_sig);
    reg_id_ie: ID_IE
        port map(
            clk             => clk,
            flush           => (branch_cond_ie or branch_cond_dm or branch_cond_wb)
                                or (op_class_ie(4) or op_class_dm(4) or op_class_wb(4)),
            curr_pc_id      => curr_pc_id,
            next_pc_id      => next_pc_id,
            op_class_id     => op_class_id,
            funct3_id       => funct3_id,
            funct7_id       => funct7_id,
            a_sel_id        => a_sel_id,
            b_sel_id        => b_sel_id,
            imm_se_id       => imm_se_id,
            rs1_value_id    => rs1_value_id,
            rs2_value_id    => rs2_value_id,
            rd_addr_id      => rd_addr_id,
            rs1_addr_id     => rs1_addr_id,
            rs2_addr_id     => rs2_addr_id,
            cond_opcode_id  => cond_opcode_id,
```

```
            curr_pc_ie      => curr_pc_ie,
            next_pc_ie      => next_pc_ie,
            op_class_ie     => op_class_ie,
            funct3_ie       => funct3_ie,
            funct7_ie       => funct7_ie,
            a_sel_ie        => a_sel_ie,
            b_sel_ie        => b_sel_ie,
            imm_se_ie       => imm_se_ie,
            rs1_value_ie    => rs1_value_ie,
            rs2_value_ie    => rs2_value_ie,
            rd_addr_ie      => rd_addr_ie,
            rs1_addr_ie     => rs1_addr_ie,
            rs2_addr_ie     => rs2_addr_ie,
            cond_opcode_ie  => cond_opcode_ie);
    ie_inst : instr_exec_pipeline
        port map(
            a_sel           => a_sel_ie,
            b_sel           => b_sel_ie,
            rs1_value_in    => rs1_value_ie,
            rs2_value_in    => rs2_value_ie,
            imm_se          => imm_se_ie,
            curr_pc         => curr_pc_ie,
            cond_opcode     => cond_opcode_ie,
            funct3          => funct3_ie,
            funct7          => funct7_ie,
            op_class        => op_class_ie,
            rs1_addr_in     => rs1_addr_ie,
            rs2_addr_in     => rs2_addr_ie,
            rd_addr_dm      => rd_addr_dm,
            alu_result_dm   => alu_result_dm,
            rd_addr_wb      => rd_addr_wb,
            alu_result_wb   => alu_result_wb,
            branch_cond     => branch_cond_ie,
            rs2_value_out   => rs2_value_out,
            alu_result      => alu_result_ie);
    reg_ie_dm: IE_DM
        port map(
            clk             => clk,
            flush           => branch_cond_dm or branch_cond_wb or
                                op_class_dm(4) or op_class_wb(4),
            curr_pc_ie      => curr_pc_ie,
            next_pc_ie      => next_pc_ie,
            op_class_ie     => op_class_ie,
            funct3_ie       => funct3_ie,
            alu_result_ie   => alu_result_ie,
            rs2_value_ie    => rs2_value_out,
            rd_addr_ie      => rd_addr_ie,
            branch_cond_ie  => branch_cond_ie,
            curr_pc_dm      => curr_pc_dm,
            next_pc_dm      => next_pc_dm,
            op_class_dm     => op_class_dm,
            funct3_dm       => funct3_dm,
            alu_result_dm   => alu_result_dm,
            rs2_value_dm    => rs2_value_dm,
            rd_addr_dm      => rd_addr_dm,
            branch_cond_dm  => branch_cond_dm);
    mem : data_memory_pipeline
        port map(
            clk             => clk,
            op_class        => op_class_dm,
            funct3          => funct3_dm,
            rs2_value       => rs2_value_dm,
            alu_result      => alu_result_dm,
            read_addr       => addr_sel_sig,
            read_data_out   => data_read_dm_sig,
            mem_out         => mem_out_dm);
    reg_dm_wb: DM_WB
        port map(
            clk             => clk,
            next_pc_dm      => next_pc_dm,
            mem_out_dm      => mem_out_dm,
            alu_result_dm   => alu_result_dm,
            rd_addr_dm      => rd_addr_dm,
            branch_cond_dm  => branch_cond_dm,
            op_class_dm     => op_class_dm,
            next_pc_wb      => next_pc_wb,
            mem_out_wb      => mem_out_wb,
```

```
            alu_result_wb   => alu_result_wb,
            rd_addr_wb      => rd_addr_wb,
            branch_cond_wb  => branch_cond_wb,
            op_class_wb     => op_class_wb);
    wb : write_back_pipeline
        port map(
            branch_cond     => branch_cond_wb,
            op_class        => op_class_wb,
            mem_out         => mem_out_wb,
            next_pc         => next_pc_wb,
            alu_result      => alu_result_wb,
            rd_write_en     => rd_write_en_wb,
            pc_load_en      => pc_load_en_wb,
            pc_out          => pc_out_wb,
            rd_value        => rd_value_wb);
     mem_read : memory_reader
        port map(
            clk             => clk,
            data_in_dm      => data_read_dm_sig,
            data_in_reg     => data_read_reg_sig,
            sw_i            => sw_i,
            sw_reg_mem      => btnc_i,
            disp_seg        => disp_seg_o,

            an              => disp_an_o,
            led_green       => rgb2_green_o,
            led_red         => rgb2_red_o,
            addr_sel        => addr_sel_sig
            );
end Structural;
```

Appendix D: Testing Block for Nexys 4

Reading from register file and DM

Source Code 5.23: Architecture of a register and DM reader with a 7-segments display driver for Nexys 4 DDR

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity memory_reader is
    Port (
        clk         : in std_logic;
        data_in_dm  : in std_logic_vector(31 downto 0);
        data_in_reg : in std_logic_vector(31 downto 0);
        sw_i        : in std_logic_vector(15 downto 0);
        sw_reg_mem  : in std_logic;

        disp_seg    : out std_logic_vector(7 downto 0);
        an          : out std_logic_vector(7 downto 0);
        led_green   : out std_logic;
        led_red     : out std_logic;
        addr_sel    : out std_logic_vector(13 downto 0)
    );
end memory_reader;

architecture Behavioral of memory_reader is
signal clk_div          : std_logic := '0';
signal mode_reg_mem     : std_logic := '0';
signal data_in_dm_buf   : std_logic_vector(31 downto 0) := (others => '0');
signal data_in_reg_buf  : std_logic_vector(31 downto 0) := (others => '0');
signal active_digit     : integer range 0 to 7 := 0;
signal led_green_buf    : std_logic := '0';
signal led_red_buf      : std_logic := '0';

function convert_bcd( data_in: std_logic_vector(3 downto 0)) return std_logic_vector is
variable disp_seg : std_logic_vector(7 downto 0);
begin
    case data_in is
        when "0000" => disp_seg := "11000000"; -- "0"
        when "0001" => disp_seg := "11111001"; -- "1"
        when "0010" => disp_seg := "10100100"; -- "2"
        when "0011" => disp_seg := "10110000"; -- "3"
        when "0100" => disp_seg := "10011001"; -- "4"
```

```
        when "0101" => disp_seg := "10010010"; -- "5"
        when "0110" => disp_seg := "10000010"; -- "6"
        when "0111" => disp_seg := "11111000"; -- "7"
        when "1000" => disp_seg := "10000000"; -- "8"
        when "1001" => disp_seg := "10010000"; -- "9"
        when "1010" => disp_seg := "10001000"; -- a
        when "1011" => disp_seg := "10000011"; -- b
        when "1100" => disp_seg := "11000110"; -- C
        when "1101" => disp_seg := "10100001"; -- d
        when "1110" => disp_seg := "10000110"; -- E
        when "1111" => disp_seg := "10001110"; -- F
    end case;
    return disp_seg;
end function;

begin
    process(clk)
        variable counter    : integer range 0 to 4999 := 0;
    begin
        if rising_edge(clk) then
            counter := counter + 1;
            if counter = 4999 then
                counter             := 0;
                if active_digit < 7 then
                    active_digit    <= active_digit + 1;
                else
                    active_digit    <= 0;
                end if;
            end if;
        end if;
    end process;

    process(clk)
    variable counter : integer range 0 to 4999 := 0;
    begin
        if rising_edge(clk) then
            if counter = 4999 then
                data_in_dm_buf      <= data_in_dm;
                data_in_reg_buf     <= data_in_reg;
                counter := 0;
            end if;
            counter := counter + 1;
        end if;
    end process;

    process(active_digit)
    begin
        if mode_reg_mem = '1' then
            led_green_buf   <= '1';
            led_red_buf     <= '0';
        else
            led_green_buf   <= '0';
            led_red_buf     <= '1';
       end if;

        case active_digit is
            when 0 =>
                disp_seg <= convert_bcd(data_in_dm_buf(3 downto 0) when mode_reg_mem = '0'
                            else data_in_reg_buf(3 downto 0));
                an       <= "11111110";
            when 1 =>
                disp_seg <= convert_bcd(data_in_dm_buf(7 downto 4) when mode_reg_mem = '0'
                            else data_in_reg_buf(7 downto 4));
                an       <= "11111101";
            when 2 =>
                disp_seg <= convert_bcd(data_in_dm_buf(11 downto 8) when mode_reg_mem = '0'
                            else data_in_reg_buf(11 downto 8));
                an       <= "11111011";
            when 3 =>
                disp_seg <= convert_bcd(data_in_dm_buf(15 downto 12) when mode_reg_mem = '0'
                            else data_in_reg_buf(15 downto 12));
                an       <= "11110111";
            when 4 =>
                disp_seg <= convert_bcd(data_in_dm_buf(19 downto 16) when mode_reg_mem = '0'
                            else data_in_reg_buf(19 downto 16));
                an       <= "11101111";
            when 5 =>
```

```
                disp_seg <= convert_bcd(data_in_dm_buf(23 downto 20) when mode_reg_mem = '0'
                            else data_in_reg_buf(23 downto 20));
                an       <= "11011111";
            when 6 =>
                disp_seg <= convert_bcd(data_in_dm_buf(27 downto 24) when mode_reg_mem = '0'
                            else data_in_reg_buf(27 downto 24));
                an       <= "10111111";
            when 7 =>
                disp_seg <= convert_bcd(data_in_dm_buf(31 downto 28) when mode_reg_mem = '0'
                            else data_in_reg_buf(31 downto 28));
                an       <= "01111111";
            when others =>
        end case;
    end process;

    addr_sel    <=  sw_i(13) & sw_i(12) & sw_i(11) & sw_i(10) &
                    sw_i(9) & sw_i(8) & sw_i(7) & sw_i(6) & sw_i(5) &
                    sw_i(4) & sw_i(3) & sw_i(2) & sw_i(1) & sw_i(0);

    mode_reg_mem <= sw_i(15);

    led_green   <= led_green_buf;
    led_red     <= led_red_buf;

end Behavioral;
```

Bibliography

[1] Andrew Waterman et al. The risc-v instruction set manual, volume i: User-level isa, version 2.1. EECS Department Technical Report UCB/EECS-2016-118, University of California at Berkeley, 2016.