RISC-V CPU core in VHDL

Report

Mario Ivanov
Muhammad Khan Lodhi
Kamran Rashidov
Orkhan Elchuev

October 10, 2022

Contents

1 Introduction

During this course, the team set a goal to design a CPU, and implement the logic behind it in VHDL. The main aim of the team was not to design a modern, state-of-the-art microprocessor, but to deepen the participants' understanding of digital design and programming in VHDL. After spending time learning about the available architectures and trends and modern processor's design the team decided to implement a 32-bit single-cycle RISC-V core, adhering to the integer architecture, also known as RV32I.

2 Architecture

The architecture for the CPU, apart from being RV32I, was chosen to be the Harvard architecture, or in other words, having separate modules for data memory and instruction memory. The reason the team decided on this architecture was based on the decision to implement a single-cycle CPU rather than a multi-cycle or pipe-lined one, which requires the memory to be read and written in the same cycle which makes reading and writing from the same memory impossible.

3 Components implementation

The team implemented the following components, listed in alphabetical order: Adder, Arithmetic Logic Unit (ALU), Data Memory, Decoder, Instruction Memory, Multiplexer, Register File, and Sign Extender.

3.1 Adder

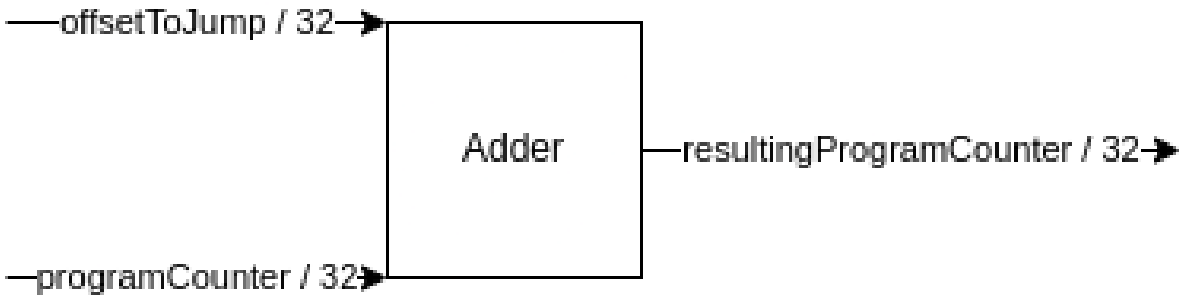


Figure 1: Adder block diagram

The adder adds the program counter register to the offset to jump to. Since it is used for fetching instructions only and the team does not use the compressed instruction extension, it allows us to use plus one as an increment which increments the actual counter by 32 bits. Since the addition is rather a lengthy operation, it is done using combinations logic, without the clock. This component could have been omitted if the design was chosen to be pipe-lined, since the ALU could have been used to increment the program register, usually R1 as specified by the specification, however not the case in this design.

3.2 Arithmetic Logic Unit

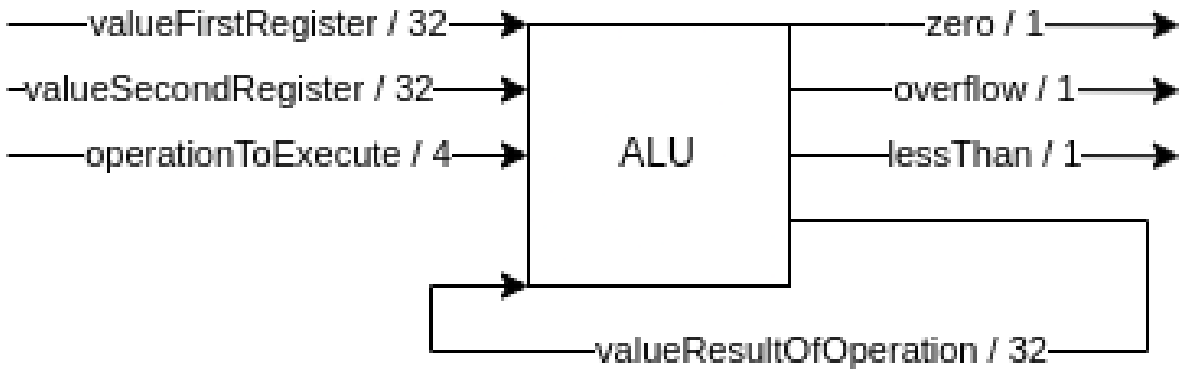


Figure 2: ALU block diagram

The ALU is used to carry the computations on the registers. It takes 2 registers and a 4-bit instruction to carry on said registers. The output is a 32-bit result of the operation as well a flag for zero, overflow or less than. Zero and less than are used for implementation of the branching instructions, the overflow however is not used since it's used to throw an exception, however, the core the team implemented does not cover the privileged instruction set, so action is taken on it. The circuit is implemented as sequential logic since the rising clock edge can not be used for the output updates on a change of inputs rather than on each clock cycle.

3.3 Data Memory

While the data memory is not part of the core itself the team implemented a small memory, with the idea that if the design was to be extended it can be used as cache memory inside the core. The data memory is written on a clock edge, which would suggest that there is a delay of a clock cycle between the execution and the write-back, however, it was the prevalent design in the reference literature. The data memory is implemented as an array of arbitrary size of 8-bit "std_logic_vector" to allow the implementation of loading and storing byte, half and word into the registers.

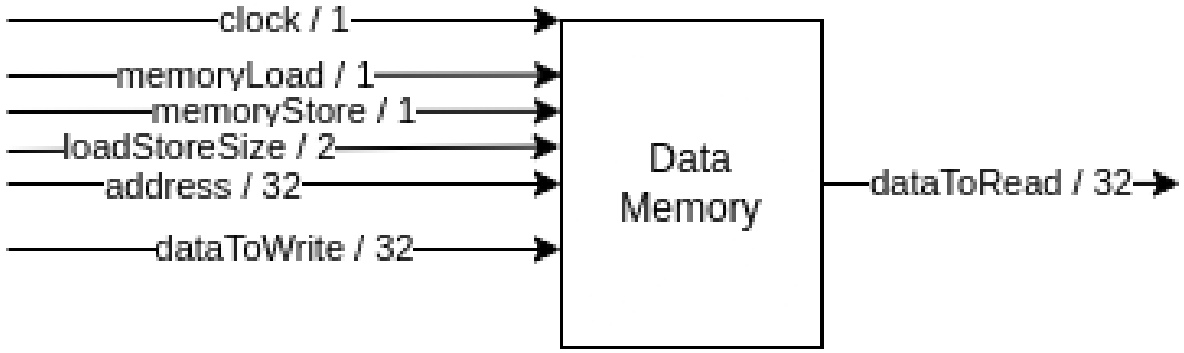


Figure 3: Data memory block diagram

3.4 Decoder

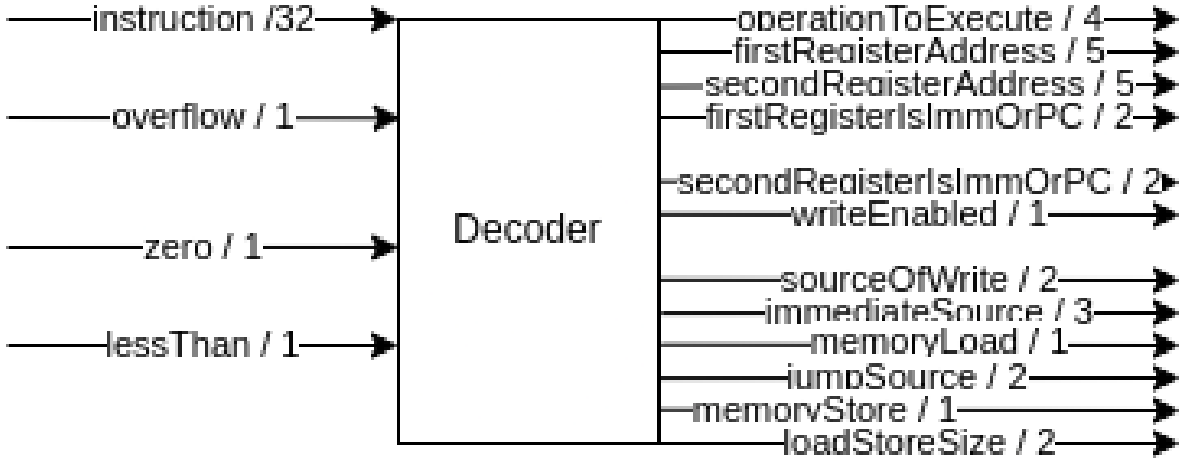


Figure 4: Decoder block diagram

The decoder is the circuit which controls the data path. It takes the whole instruction and decodes it to the flags needed to control the sources of the ALU and the operation to execute on them, the write ports of the memory and the register file as well as the decision to increment the program counter with a specified offset, i.e. jump. While the instruction set does not specify, how this memory is to be addressed the team made the decision, as mentioned before an increment of one to load the next 32 bits or jump instruction. This gives a much larger range of a jump compared with the one which requires an offset of 4 to jump a single instruction since the jump offset is encoded into 11-bit immediate.

3.5 Instruction Memory

The instruction memory, much like the data memory, reads and writes on a rising clock edge. It is implemented as an array of 32 bit “std_logic_vector”. Since the team does

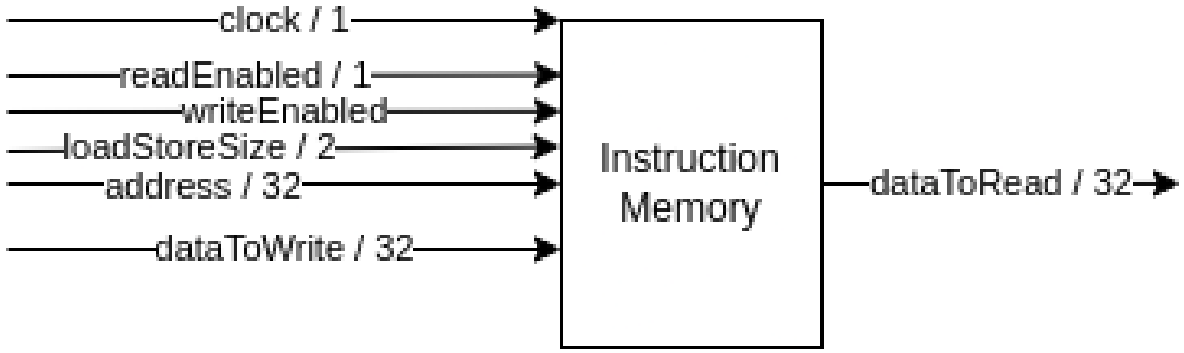


Figure 5: Instruction memory block diagram

not implement the compressed version of the RISC-V instruction set and this memory is used for instructions only the need to be able to load and store variable size is not needed, which as a plus, removes the need to check for misalignment of addresses.

3.6 Multiplexer

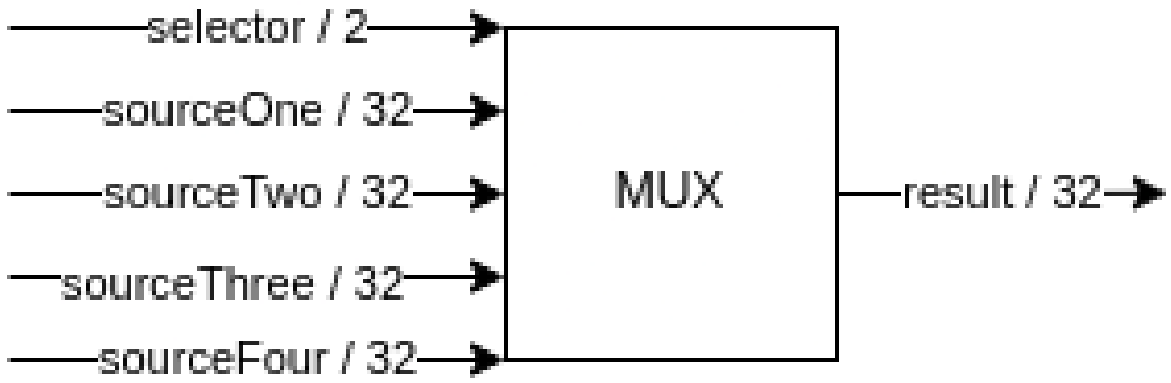


Figure 6: 4 to 1 multiplexer block diagram

The multiplexer was used for selecting the source of a jump for the program counter, the write port of the register file, and the write port of the data memory, although the team simplified the design by using the same selector as well as a multiplexer for both memory and register file with the only difference of the enable write flag, both sources of the ALU. While not all have the same number of ports the the decision to use only one type of multiplexer, namely 4-to-1 and give zero values to the unconnected ports with the idea that an extension of the CPU might require it.

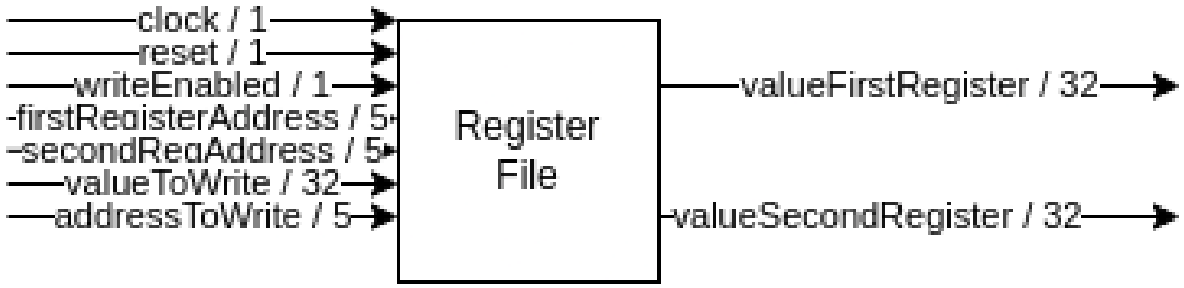


Figure 7: Register file block diagram

3.7 Register File

The register file consists of 32 registers of XLEN size. In this particular project, this is 32-bit registers. The component is implemented as an array of “"std_logic_vector"” and is addressable by a 5-bit address. The register file consists of two read ports and one write port, as well as an address port for each of them. The register file is a combinational circuit which executes only on the rising edge of the clock. It consists of two processes one for writing to the register and one to read from the register file. As specified by the ISA register 0 is a hard-coded zero. Writing to it doesn't result in an error however, no change the register is made.

3.8 Sign Extender

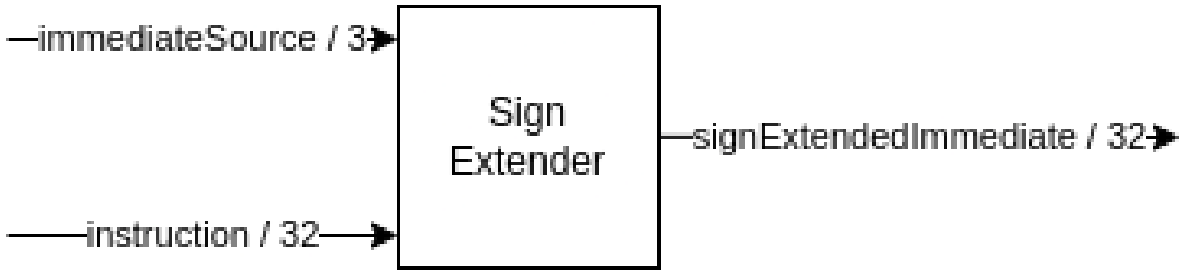


Figure 8: Sign Extender block diagram

The sign extender aims to build a 32-bit sign-extended immediate from an 11-bit one. Depending on the type of instruction the position of the bits building the immediate varies. Apart from that function, the team used the circuit for the instruction load upper immediate which does not sign extend, but rather takes the immediate and concatenates zeroes to the back. Jump and link register is also partially implemented with the help of the sign extender although sign extension is not done. The instruction is used to add an arbitrary offset to the program counter unconditionally by providing a 20-bit immediate value which needs to be built to 32-bit before being added to the program counter.

3.9 Putting it all together

After having the components they can generally be split into 2 parts data path and control. In this project, the control is done entirely by the decoder and the data path is the connection of all the rest of the components. As previously specified the team implemented the integer part of the instruction set. The general idea is described below.

3.9.1 LUI and AUIPC

For both "load upper immediate" and "add upper immediate to PC" the sign extender was overloaded with building the value from the 20 bit immediate. Both instructions set the most significant bits so the sign extender is not the most intuitive component to do this task however it is best suited for it since its output is already routed to the PC and the ports of the ALU and register file. After building the immediate the source of either the ALU is selected for "LUI" or the source in front of the adder producing the PC is set.

3.9.2 JAL and JALR

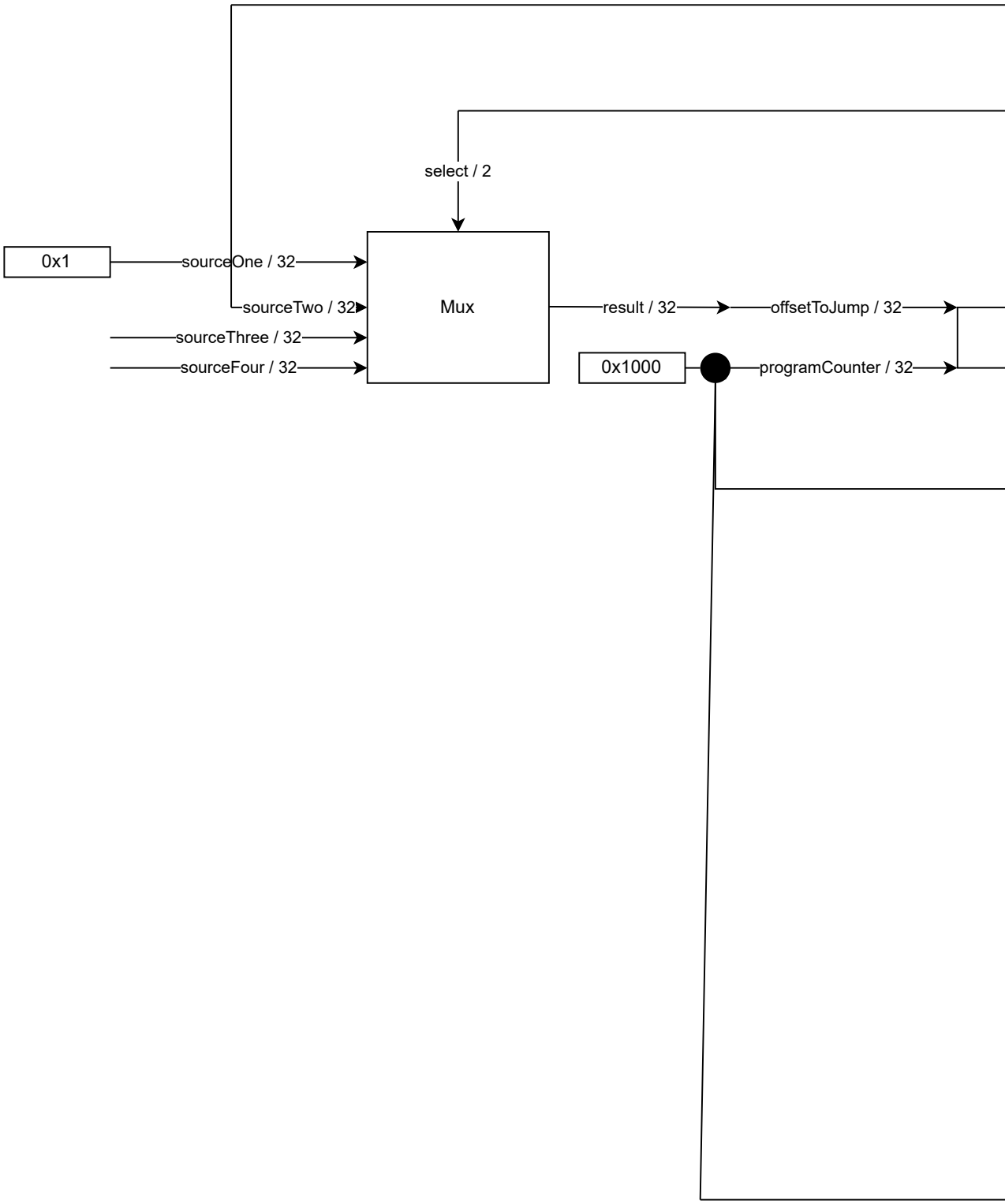
The instructions "jump and link" and "jump and link register" are both used for unconditional jumps in the program. JAL stores the address of the instruction following the jump into the destination register, or in other words the immediate is sign extended and added into the ALU with the current PC after that the control selects the address of the register file to be the destination register specified, enables write to register file and selects the source of write to be the ALU result. For the JALR instruction, the path is the same only instead of the PC the offset is added to the specified source register. Both instructions need to be paired with the ones above otherwise the PC will not actually be incremented. The instructions are supposed to raise an exception if the addresses specified are misaligned, however, this has not been implemented since in the current design the instruction memory is 4-byte addressable so misalignment is not possible.

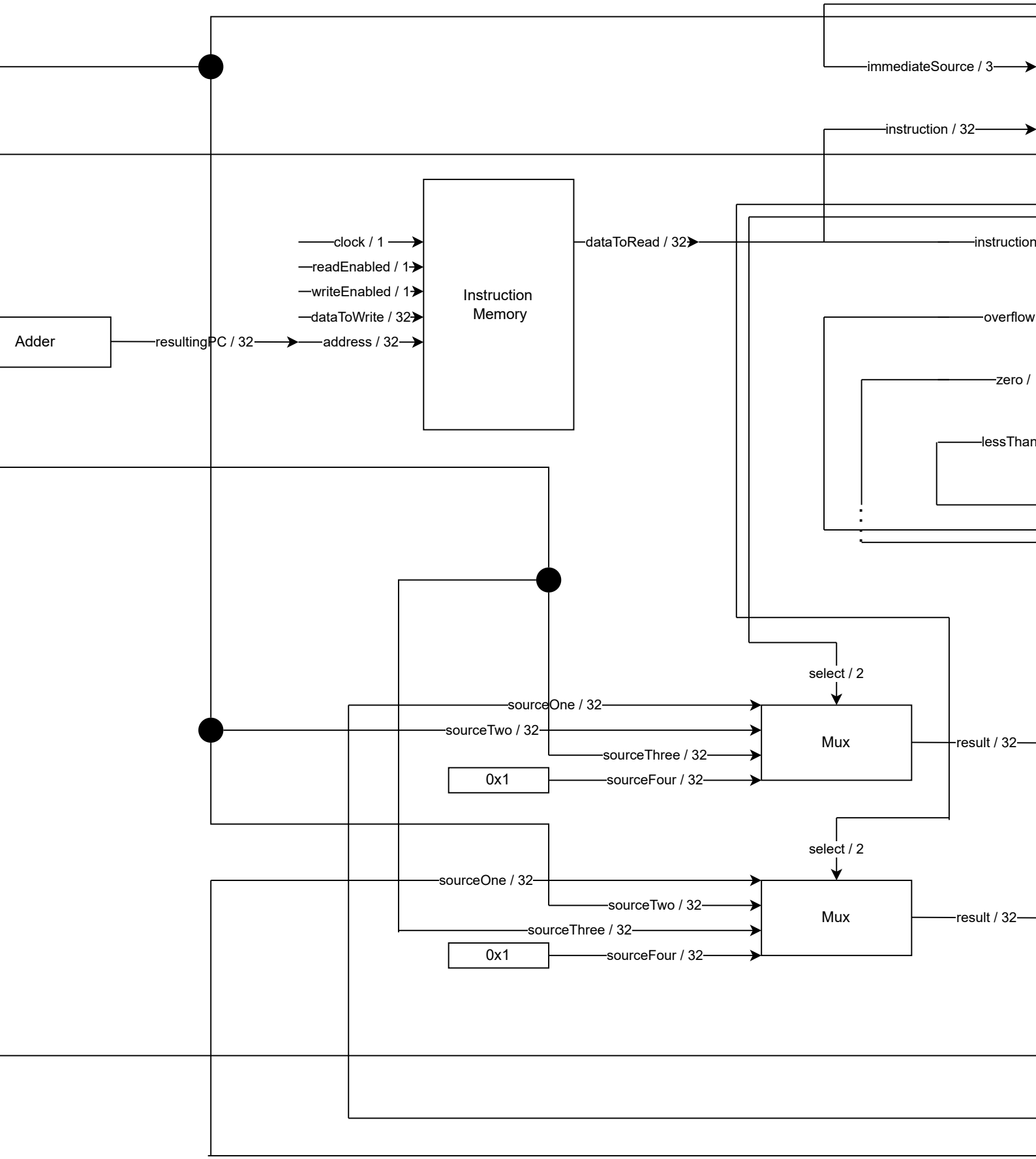
3.9.3 Branching and ALU operations

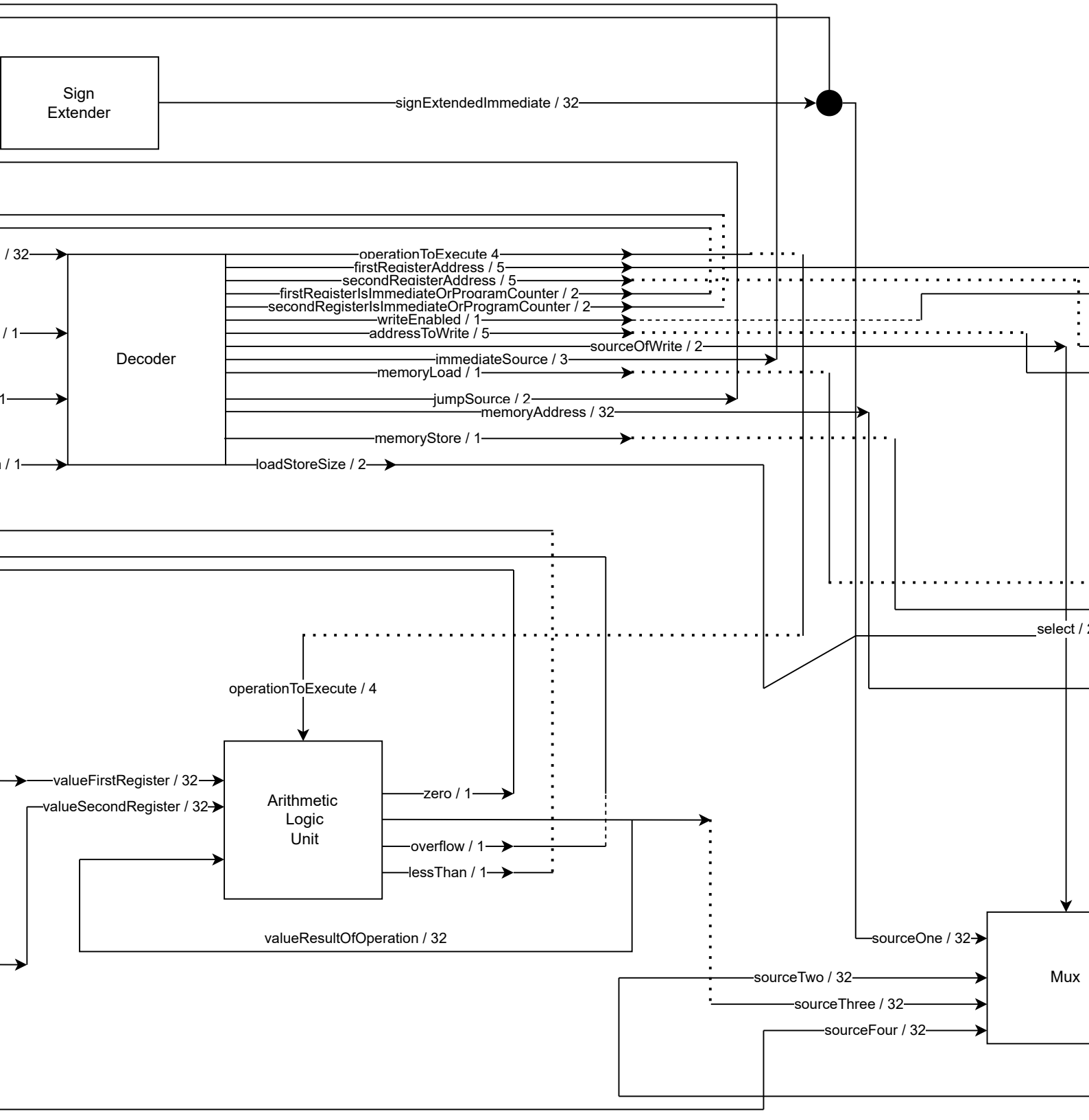
All branch instructions are mapped to an operation of the ALU. For each operation, two registers are compared after which the control unit sets the source of a jump for the PC added. The operations of the ALU are implemented for half of the instructions not taking into account whether the addition is done on two registers or the sign is extended immediately. For immediate instructions, the litteral is first sign extended followed by loading it into the second register and carrying the ALU operation. Those operations are added, subtract, set less than, set less than unsigned, xor, or, and, shift left logical, shift right logical, and shift right arithmetic.

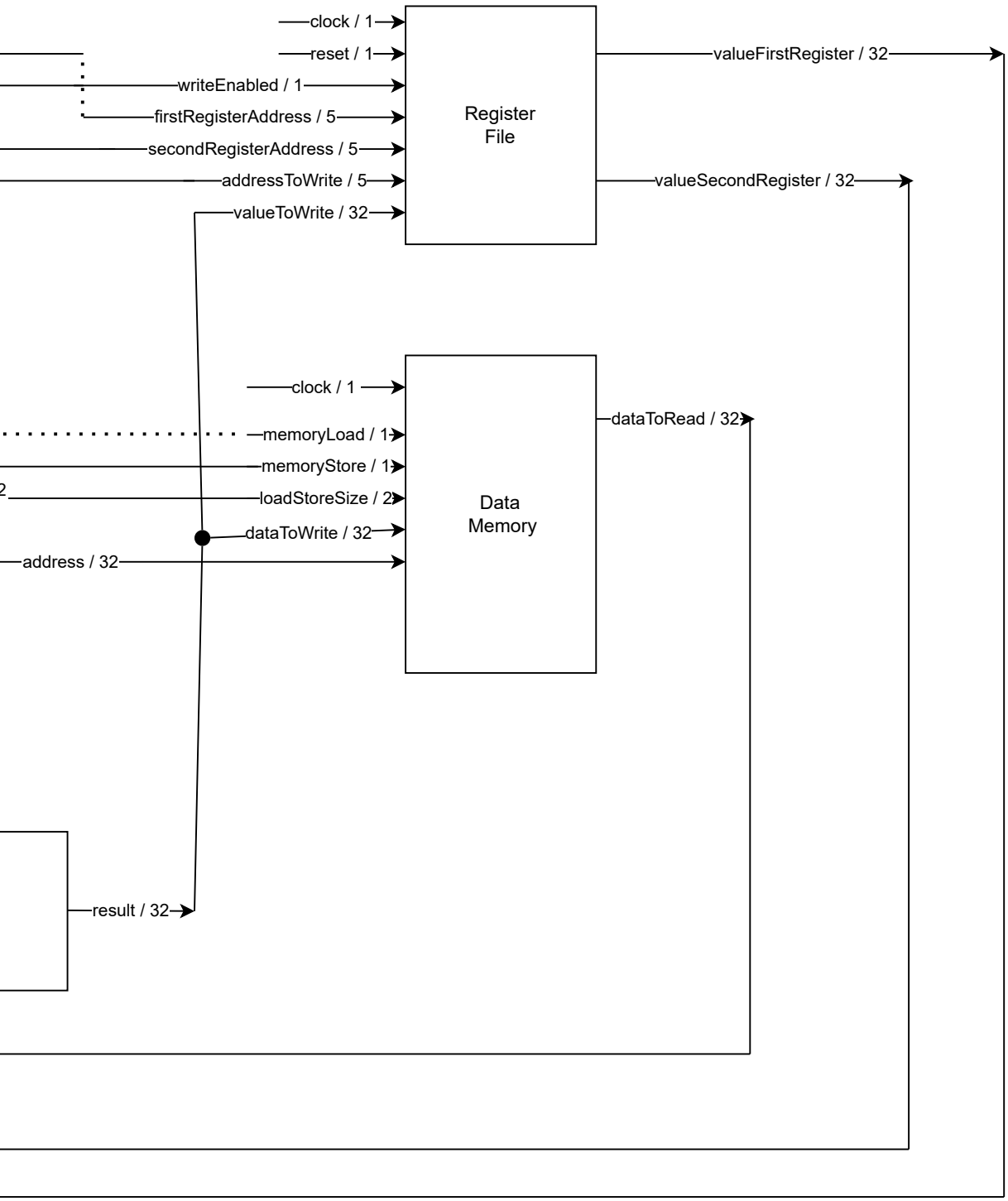
3.9.4 Load-Store operations

The load-store operations are implemented taking into account only the data memory since the instruction memory doesn't allow, by design, loading a different size than 4 bytes. The signedness of the loaded and stored data is ignored since they are all stored into 2's complement and this is left for the compiler. "Unimplemented instructions" The ISA specifies 5 instructions which have not been implemented, namely FENCE, FENCE.TSO, PAUSE, RECALL, BREAK. The instructions are not implemented since they deal with mitigating memory inconsistencies in out of order, pipeline processors and context switching, for processors who implement the privileged calls as well.


select / 2
0x1
sourceOne / 32
sourceTwo / 32
sourceThree / 32
sourceFour / 32
Mux
result / 32
offsetToJump / 32
0x1000
programCounter / 32


immediateSource / 3
instruction / 32
clock / 1
readEnabled / 1
writeEnabled / 1
dataToWrite / 32
address / 32
Instruction Memory
dataToRead / 32
instruction
Adder
resultingPC / 32
overflow
zero /
lessThan
select / 2
sourceOne / 32
sourceTwo / 32
sourceThree / 32
0x1
sourceFour / 32
Mux
result / 32
select / 2
sourceOne / 32
sourceTwo / 32
sourceThree / 32
0x1
sourceFour / 32
Mux
result / 32


Sign
Extender
signExtendedImmediate / 32
Decoder
operationToExecute 4
firstRegisterAddress / 5
secondRegisterAddress / 5
firstRegisterIsImmediateOrProgramCounter / 2
secondRegisterIsImmediateOrProgramCounter / 2
writeEnabled / 1
addressToWrite / 5
sourceOfWrite / 2
immediateSource / 3
memoryLoad / 1
jumpSource / 2
memoryAddress / 32
memoryStore / 1
loadStoreSize / 2
operationToExecute / 4
valueFirstRegister / 32
valueSecondRegister / 32
Arithmetic
Logic
Unit
zero / 1
overflow / 1
lessThan / 1
valueResultOfOperation / 32
sourceOne / 32
sourceTwo / 32
sourceThree / 32
sourceFour / 32
Mux


clock / 1
reset / 1
writeEnabled / 1
firstRegisterAddress / 5
secondRegisterAddress / 5
addressToWrite / 5
valueToWrite / 32
Register
File
valueFirstRegister / 32
valueSecondRegister / 32
clock / 1
memoryLoad / 1
memoryStore / 1
loadStoreSize / 2
dataToWrite / 32
address / 32
Data
Memory
dataToRead / 32
result / 32

4 Testing

4.1 Adder

Here is the test case representation of implemented Adder behaviour. We have 32 Bit program counter "programCounter" with an initial value of "00100000000000000000000000000000" and we have a 32 bit offset value. The offset is represented by variable "offsetToJump". We add the value of the offset to the program counter after every 20 nanoseconds.
For e.g. First value of offset is "00000000000000000000000000000001". The expected result then is offset value + program counter value, which equates to 00100000000000000000000000000001 in Binary representation. All the expected values are represented on the Figure 9 below.

```
signal offsetToJump : STD_LOGIC_VECTOR (31 downto 0);
signal programCounter: STD_LOGIC_VECTOR (31 downto 0) := "00100000000000000000000000000000";
signal resultingProgramCounter: STD_LOGIC_VECTOR(31 downto 0);
begin                                                -- Offset to jump                    Initial program counter           Expected result
offsetToJump <= "00000000000000000000000000000001" after 20 ns,  -- 00000000000000000000000000000001 + 00100000000000000000000000000000 = 00100000000000000000000000000001
                "00000000000000000000000000000011" after 40 ns,  -- 00000000000000000000000000000011 + 00100000000000000000000000000001 = 00100000000000000000000000000100
                "00000000000000000000000000000101" after 60 ns,  -- 00000000000000000000000000000101 + 00100000000000000000000000000100 = 00100000000000000000000000001001
                "00000000000000000000000000001001" after 80 ns,  -- 00000000000000000000000000001001 + 00100000000000000000000000001001 = 00100000000000000000000000010010
                "00000000000000000000000000010001" after 100 ns, -- 00000000000000000000000000010001 + 00100000000000000000000000010010 = 00100000000000000000000000100011
                "00000000000000000000000000100001" after 110 ns, -- 00000000000000000000000000100001 + 00100000000000000000000000100011 = 00100000000000000000000001000100
                "00000000000000000000000001000001" after 130 ns, -- 00000000000000000000000001000001 + 00100000000000000000000001000100 = 00100000000000000000000010000101
                "00000000000000000000000010000001" after 150 ns, -- 00000000000000000000000010000001 + 00100000000000000000000010000101 = 00100000000000000000000100000110
                "00000000000000000000000100000001" after 170 ns, -- 00000000000000000000000100000001 + 00100000000000000000000100000110 = 00100000000000000000001000000111
                "00000000000000000000001000000001" after 190 ns, -- 00000000000000000000001000000001 + 00100000000000000000001000000111 = 00100000000000000000010000001000
                "00000000000000000000010000000001" after 210 ns, -- 00000000000000000000010000000001 + 00100000000000000000010000001000 = 00100000000000000000100000001001
                "00000000000000000000100000000001" after 230 ns, -- 00000000000000000000100000000001 + 00100000000000000000100000001001 = 00100000000000000001000000001010
                "00000000000000000001000000000001" after 250 ns, -- 00000000000000000001000000000001 + 00100000000000000001000000001010 = 00100000000000000010000000001011
                "00000000000000000010000000000001" after 270 ns, -- 00000000000000000010000000000001 + 00100000000000000010000000001011 = 00100000000000000100000000001100
                "00000000000000000100000000000001" after 290 ns, -- 00000000000000000100000000000001 + 00100000000000000100000000001100 = 00100000000000001000000000001101
                "00000000000000001000000000000001" after 310 ns, -- 00000000000000001000000000000001 + 00100000000000001000000000001101 = 00100000000000010000000000001110
                "00000000000000010000000000000001" after 330 ns, -- 00000000000000010000000000000001 + 00100000000000010000000000001110 = 00100000000000100000000000001111
                "00000000000000100000000000000001" after 350 ns, -- 00000000000000100000000000000001 + 00100000000000100000000000001111 = 00100000000001000000000000010000
                "00000000000001000000000000000001" after 370 ns, -- 00000000000001000000000000000001 + 00100000000001000000000000010000 = 00100000000010000000000000010001
                "00000000000010000000000000000001" after 390 ns, -- 00000000000010000000000000000001 + 00100000000010000000000000010001 = 00100000000100000000000000010010
                "00000000000100000000000000000001" after 410 ns, -- 00000000000100000000000000000001 + 00100000000100000000000000010010 = 00100000001000000000000000010011
                "00000000001000000000000000000001" after 430 ns, -- 00000000001000000000000000000001 + 00100000001000000000000000010011 = 00100000010000000000000000010100
                "00000000010000000000000000000001" after 450 ns, -- 00000000010000000000000000000001 + 00100000010000000000000000010100 = 00100000100000000000000000010101
                "00000000100000000000000000000001" after 470 ns, -- 00000000100000000000000000000001 + 00100000100000000000000000010101 = 00100001000000000000000000010110
                "00000001000000000000000000000001" after 490 ns, -- 00000001000000000000000000000001 + 00100001000000000000000000010110 = 00100010000000000000000000010111
                "00000010000000000000000000000001" after 510 ns, -- 00000010000000000000000000000001 + 00100010000000000000000000010111 = 00100100000000000000000000011000
                "00000100000000000000000000000001" after 530 ns, -- 00000100000000000000000000000001 + 00100100000000000000000000011000 = 00101000000000000000000000011001
                "00001000000000000000000000000001" after 550 ns, -- 00001000000000000000000000000001 + 00101000000000000000000000011001 = 00110000000000000000000000011010
                "00010000000000000000000000000001" after 570 ns, -- 00010000000000000000000000000001 + 00110000000000000000000000011010 = 01000000000000000000000000011011
                "00100000000000000000000000000001" after 590 ns, -- 00100000000000000000000000000001 + 01000000000000000000000000011011 = 01100000000000000000000000011100
                "01000000000000000000000000000001" after 610 ns; -- 01000000000000000000000000000001 + 01100000000000000000000000011100 = 10100000000000000000000000011101
```

Figure 9: Test Cases

4.2 Arithmetic Logic Unit

The example testbench prints out the text shown in the table. The time values printed are the accumulated simulation times based on the nanosecond delays that were specified in the stimulus file.

Testing the ALU				
wait time	FirstRegister	SecondRegister	operation	Result
Testing the inputs with add operation				
5ns	00000001	00000010	0010	00000011
10ns	00000010	00000100	0010	00000110
15ns	00000100	00001000	0010	00001100
20ns	00001000	00010000	0010	00011000
25ns	00010000	00100000	0010	00110000
30ns	00100000	01000000	0010	01100000
Changing the operation to subtract				
35ns	11100000	01100000	0110	10000000
40ns	01110000	00110000	0110	01000000
45ns	00111000	00011000	0110	00100000
50ns	00011100	00001100	0110	00010000
55ns	00001110	00000110	0110	00001000
60ns	00000111	00000011	0110	00000100
Changing the operation to AND				
65ns	11111111	11111111	0000	11111111
70ns	11111111	00000000	0000	00000000
75ns	00000000	11111111	0000	00000000
Changing the operation to OR				
80ns	11111111	11111111	0001	11111111
81ns	11111111	00000000	0001	11111111
82ns	00000000	11111111	0001	11111111
83ns	00000000	00000000	0001	00000000
Changing the operation to NOR				
84ns	11111111	11111111	1100	00000000
85ns	11111111	00000000	1100	00000000
86ns	00000000	11111111	1100	00000000
87ns	00000000	00000000	1100	11111111

4.3 RV32I-CORE

The example testbench prints out the text shown in the table. The time values printed are the accumulated simulation times based on the nanosecond delays that were specified in the stimulus file.

Testing the Core						
wait time	rs2	rs1	rd	imm	instruction	pc
Testing the inputs with LOAD BYTE						
5ns	*	0x2	*	0x002	0x00104081	0000+1
6ns	0x1	*	*	0x001	0x00108103	0001+1
Changing the instruction to ADD						
10ns	0x1	0x2	0x3	*	0x00820133	0002+1
Changing the instruction to ADDI						
15ns	*	0x2	0x8	0x006	0x00620113	0003+1
Changing the instruction to subtract						
20ns	0x1	0x2	0x1	*	0x40820133	0004+1
Changing the instruction to LOAD rs1						
21ns	*	0x3	*	0x003	0x00308103	0005+1
Changing the instruction to AND rs2 rs1						
25ns	0x1	0x3	0x1	*	0x00827133	0006+1
Changing the instruction to OR rs2 rs1						
30ns	0x1	0x3	0x3	*	0x00826133	0007+1
Changing the instruction to XOR rs2 rs1						
35ns	0x1	0x3	0x1	*	0x00824133	0008+1
Changing the instruction to AUIPC						
40ns	*	*	0xA	0x1	0x00001117	0009+1
Changing the instruction to BGEU						
45ns	0x1	0x3	*	0000000010	0x00413963	0x000C
Changing the instruction to JAL						
50ns	*	*	0xD	0000000010	0x0000216F	0x000E

4.4 Decorder

The I-type format, one of the most common, is typically used for instructions operating on one register and one immediate value and placing the resulting value in a register. For instance, take assembly instruction ADDI x5, x6, 7; it instructs the processor to add the value currently contained in register x6 and the immediate value 7 and place the result in register x5. According to the specifications, ADDI has a 7-bit opcode field equal to 0010011 and a 3-bit funct3 field equal to 000. The combination of these two fields is what determines that the operation to perform is ADDI.
Now, if the source register is x6, then the 5-bit rs1 field is equal to 00110. Similarly, if the destination register is x5, then the 5-bit rd field is equal to 00101. Finally, if the immediate is 7, then the 12-bit imm field is equal to 000000000111. Putting it together, the 32-bit binary representation of assembly instruction ADDI x5, x6, 7 is
0b00000000011100110000001010010011 (which is 0x00730293 in hexadecimal).

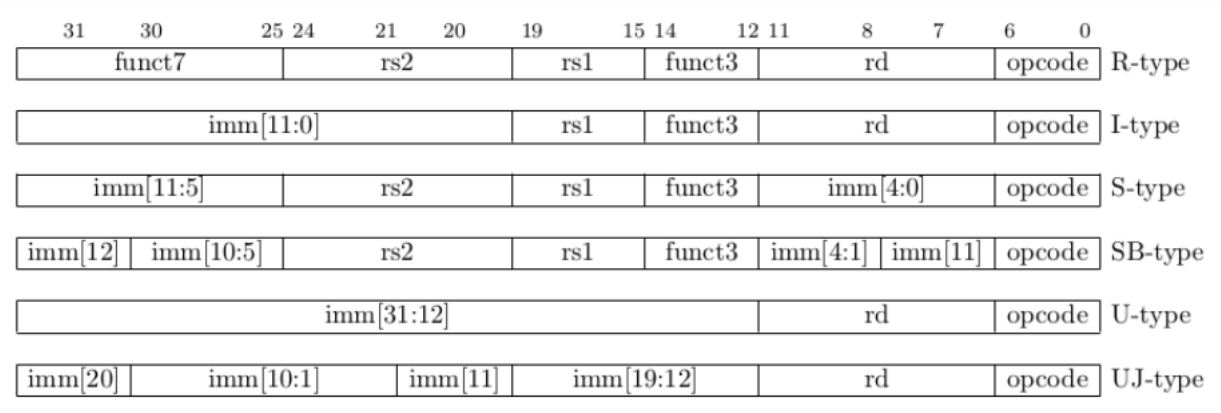


Figure 10: Instruction Format

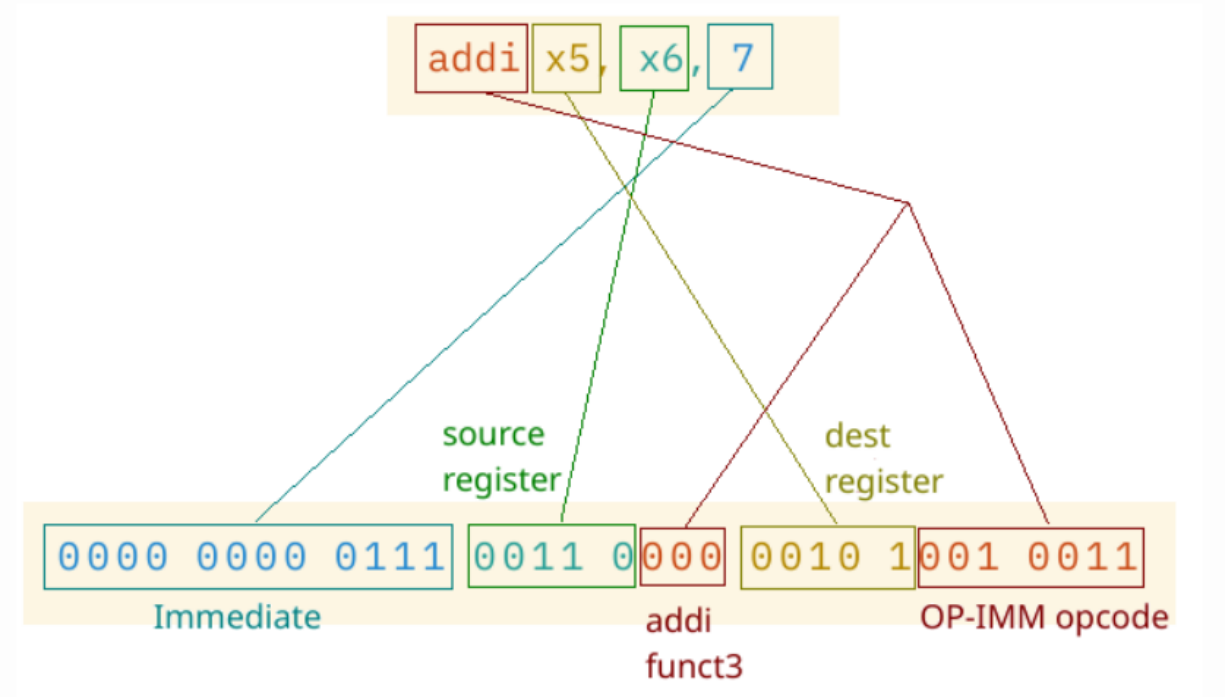


Figure 11: Decoder Algorithm

```
-- Test Cases
instruction  <= x"00820133" after 5 ns, -- decodes into ADD the values of rs1 & rs2 and then write it on rd
                x"00620113" after 10 ns,-- decodes into ADDI the values of rs1 & imm and then write it on rd
                x"40820133" after 15 ns,-- decodes into SUBTRACT rs2 from rs1 and then write it on rd
                x"00827133" after 20 ns,-- decodes into AND rs1 & rs2 and then write it on rd
                x"00826133" after 25 ns,-- decodes into OR rs1 & rs2 and then write it on rd
                x"00824133" after 30 ns,-- decodes into XOR rs1 & rs2 and then write it on rd
                x"00001117" after 35 ns;-- decodes into ADD pc & imm and then write it on rd
```

Figure 12: Test Cases

References

[1] SiFive Inc. Andrew Waterman1 Krste Asanović1. "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA". In: (2021). DOI: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf.

[2] Sarah L. Harris and David Harris. *Digital Design and Computer Architecture, RISC-V Edition*. Morgan Kaufmann, 2021. ISBN: 9780128200650.

[3] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. 5th ed. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.