Department of Electronics and Communications Engineering National Institute of Technology Rourkela, Odisha, India **PROJECT REPORT ON RISC-V SINGLE CYCLE CORE**  
  
  
**PROF.AYAS KANTA SWAIN**

SUBMITTED BY:-  
PRATEEK KUMAR DASH

Roll NO:-121EE0663

**CONTENTS**

* INTRODUCTION…………………………………………………………..3
* ALU DESIGN……………………………………………………………….5
* CONTROL UNIT……………………………………………………………6
* DATA PATH……………………………………………………………….11
* VERILOG MODULES……………………………………………………..17
* REFERENCES……………………………………………………………...24

**INTRODUCTION**

ABOUT RISC-V

The world of processor design has long been dominated by a few major players with proprietary instruction set architectures (ISAs). This closed ecosystem limited innovation and customization for specific applications. However, a new era has dawned with the rise of RISC-V, an open-source ISA poised to revolutionize processor design. This essay will explore the core principles of RISC-V design, its advantages, and the potential impact it holds for the future of computing.

At its heart, RISC-V follows the philosophy of Reduced Instruction Set Computing (RISC). Unlike its counterpart, Complex Instruction Set Computing (CISC), RISC-V utilizes a smaller set of simpler instructions. This streamlined approach allows for efficient decoding and execution, leading to faster processors with lower power consumption. Furthermore, the open-source nature of RISC-V fosters collaboration and innovation. Anyone can access, modify, and develop custom extensions to the base ISA, enabling the creation of processors tailored to specific needs. This flexibility is particularly valuable for niche applications like embedded systems and Internet-of-Things (IoT) devices, where power efficiency and low cost are paramount.

HOW TO DESIGN RISC-V PROCESSOR

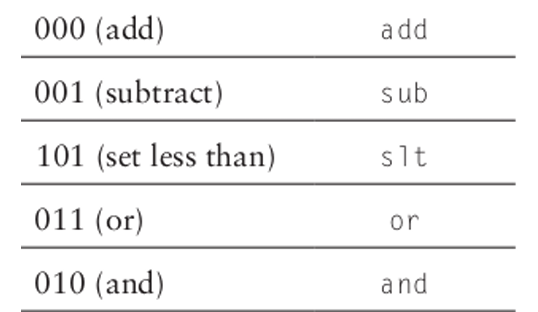
* Choose the RISC-V ISA configuration.
* Design the processor microarchitecture.
* Instruction Set Decoding logic.
* Design of ALU.
* Design Control Unit.
* Design of datapath.
* Design of memory hierarchy.
* Verification and Validation.

ADVANTAGES OF RISC-V

RISC-V processors offer several advantages over traditional architectures:

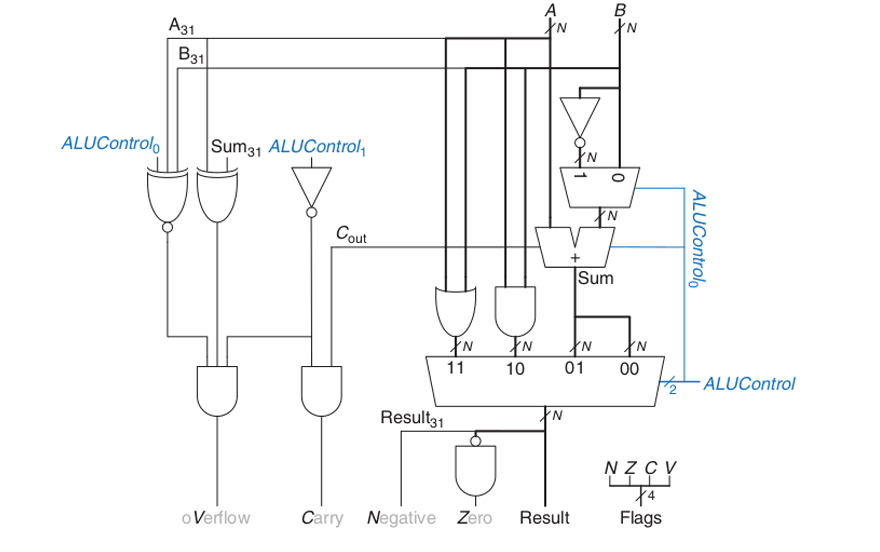
* **Open-source and royalty-free:** Unlike proprietary ISAs, RISC-V is open-source, allowing anyone to design, modify, and distribute RISC-V cores without paying licensing fees. This fosters innovation and lowers the barrier to entry for new players.
* **Flexibility and Customization:** The modular design of RISC-V with its base ISA and optional extensions enables customization for specific needs. You can choose the exact features required for your application, leading to optimized performance, power efficiency, and cost-effectiveness.
* **Simplicity and Efficiency:** RISC-V follows the RISC principles of a small set of simple instructions. This streamlined approach simplifies processor design, reduces complexity, and potentially leads to faster execution and lower power consumption.
* **Security:** The open nature of RISC-V allows for greater scrutiny and community-driven security enhancements. Additionally, specific extensions like RV32C can be incorporated for hardware-based security features.
* **Scalability:** RISC-V can be scaled to address a wide range of applications. The same ISA can be used for tiny embedded devices requiring minimal power to high-performance processors needing complex functionalities.
* **Vibrant Ecosystem:** The open-source nature fosters collaboration and a growing ecosystem of developers, researchers, and toolmakers. This collaborative environment accelerates innovation and development for both hardware and software aspects of RISC-V.
* **Future-proof:** With increasing focus on open-source hardware and diverse computing needs, RISC-V is well-positioned for future advancements. Its flexibility allows for adaptation to evolving technologies and application demands.

**THE ALU**

We will design an ALU that can perform a subset of the ALU operations of a full Processor ALU. In this exercise, we will develop an ALU that will take two 2-inputs, A and B and is able to execute the following instructions:

*FIG: ALU OPCODE*

The ALU will generate a 32-bit output that we will call ‘Result’ and an additional 1-bit flag ‘Zero’ that will be set to ‘logic-1’ if all the bits of ‘Result’ are 0. The different operations will be selected by a 3-bit control signal called ‘ALUControl’ according to the following table. For example, when the ‘ALUControl’ input is ‘011’, the function Result = A or B should be calculated. It is easy to see that there are many values of ‘ALUControl’ for which no operation has been defined. It is not very important what the circuit does when ‘ALUControl’ has these values, since the ‘Result’ will simply be ignored in these cases. You can use this to your advantage to simplify the circuit. Right now, the described operations may look random, but once we learn more about the Instruction set architecture, these choices will make more sense.



*FIG: ALU LOGIC DIAGRAM*

The three instructions add, sub, and slt are arithmetic operations, whereas the two remaining and, or are logical operations. Therefore, we have two separate groups of operations. Now let us look at the figure above and determine for which values of ALUControl we perform an operation from which group. It should be pretty clear that when ALUControl[1] is logic-1 we select a logic operation and when ALUControl[1] is logic-0 we have an arithmetic operation. This means that the output of either group can be selected by a 4-input multiplexer that is controlled by ALUControl[1:0].

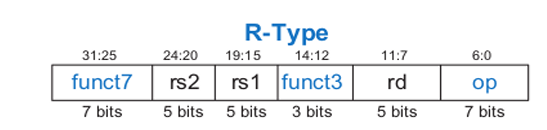
Note that in the above example, there are many values of ALUControl where the circuit would perform ‘strange’ operations (100 for example). This is not important because the circuit specification that was given to us said that these inputs were not relevant (this is probably because it can be guaranteed that these inputs will not appear during normal operation)

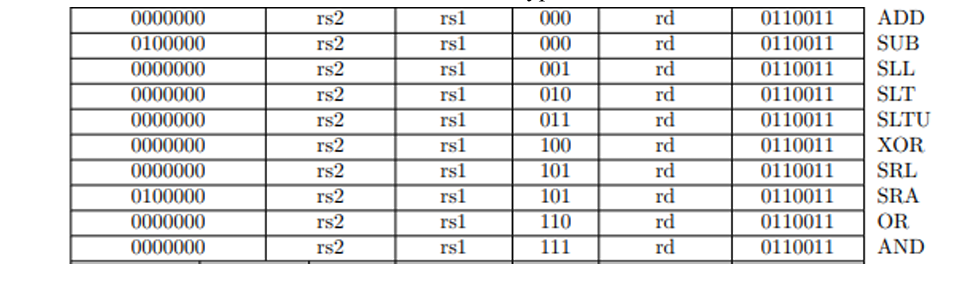
THE CONTROL UNIT

RISC-V uses 32-bit instructions. RISC-V consists of defining the following instruction formats: R-type, I-type, S-Type, B-Type. R-type instructions operate on three registers. I-type, S-type and B-type instructions operate on two registers and a 12-bit immediate.

**THE R-TYPE INSTRUCTION**

The name R-type is short for register-type. R-type instructions use three registers as operands: two as sources, and one as a destination. The figure below shows the R-type machine instruction format. The 32-bit instruction has six fields: op, rs1, rs2, rd, funct3, and funct7. Each field is five or seven bits, as indicated. The operation the instruction performs is encoded in the three fields highlighted in blue: op (also called opcode or operation code) and funct3 and funct7 (also called the function). All R-type instructions have an opcode of 33. The specific R-type operation is determined by the function fields. The operands are encoded in the three fields: rs1, rs2, and rd. The first two registers, rs1, and rs2 are the source registers; rd is the destination register.

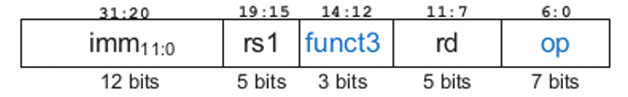
Instruction Format for some of the R-type instructions is shown below:

 *FIG: R-TYPE INSTRUCTION FORMAT*

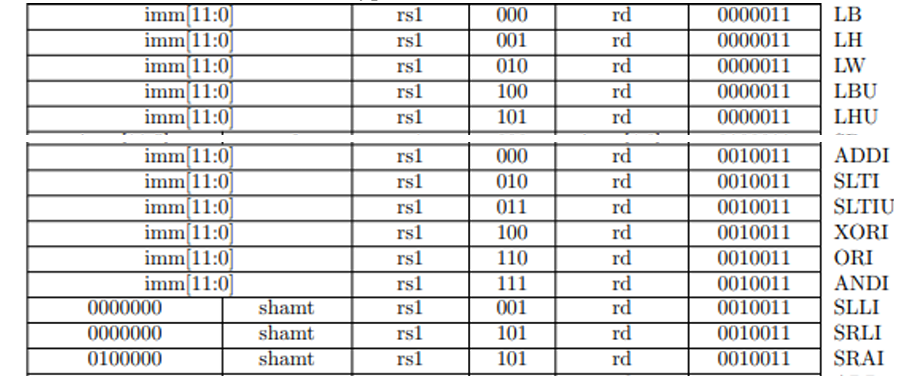
*FIG: SOME R-TYPE INSTRUCTIONS*

**THE I-TYPE INSTRUCTION**

The name I-type is short for immediate-type. I-type instructions use two register operands and one immediate operand. The figure below shows the I-type machine instruction format. The 32-bit instruction has five fields: op, rs1, rd, funct3, and imm. The first three fields, op, rs1, and rd, are like those of R-type instructions. The imm field holds the 12-bit immediate. The funct3 holds the operation to be performed. The operation is determined by the opcode and funct3, highlighted in blue. The operands are specified in the two fields rs1, and imm. rs1 and imm are always used as source operands. rd is used as a destination.



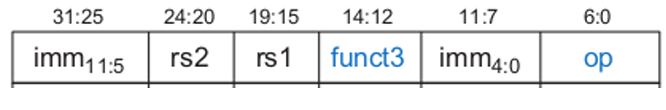
*FIG: I-TYPE INSTRUCTION FORMAT*

Instruction format for some of the I-type instructions are shown below:

*FIG: SOME I-TYPE INSTRUCTIONS*

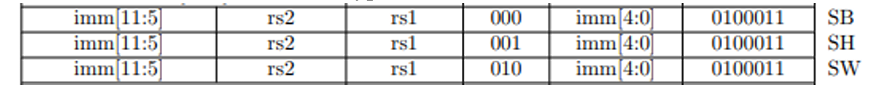
**THE S-TYPE INSTRUCTION**

The name S-type is short for store-type. S-type instructions use two register operands and one immediate operand. Figure below shows the S-type machine instruction format. The 32-bit instruction has five fields: op, rs1, rs2, funct3 and imm. The first three fields op, rs1, and rs2 are like those of R-type instructions. The imm fields hold the 12-bit immediate. The 12-bit immediate is split into two sets as shown below. The operation is determined by the opcode and funct3, highlighted in blue. The operands are specified in the three fields rs1, rs2 and imm.



*FIG: S-TYPE INSTRUCTION FORMAT*

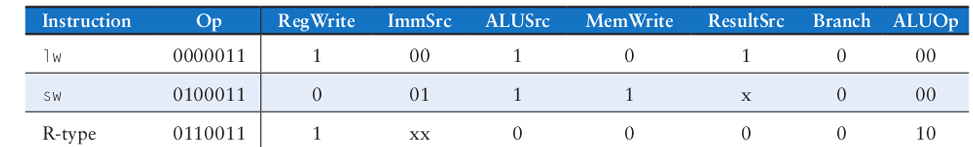
Instruction format for some of the S-type instructions are shown below:



The control unit computes the control signals based on the opcode and funct fields of the instruction, Instr[31:25], Instr[14:12] and Instr[6:0]. Most of the control information comes from the opcode, but for further operations function fields are used. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic.

**THE MAIN DECODER**

The table below is a truth table for the main decoder that summarizes the control signals as a function of the opcode. All R-type instructions use the same main decoder values; they differ only in the ALU decoder output. Recall that, for instructions that do not write to the register file (e.g., S-type), the ResultSrc control signals is don't care (X); the address and data to the register write port do not matter because RegWrite is not asserted. The logic for the decoder can be designed using your favorite techniques for combinational logic design.



**ALU DECODER**

The main decoder computes most of the outputs from the opcode. It also determines a 2-bit ALUOp signal. The ALU decoder uses this ALUOp signal in conjunction with the funct field and opcode bit to compute ALUControl. The meaning of the ALUOp signal is given in Table below.

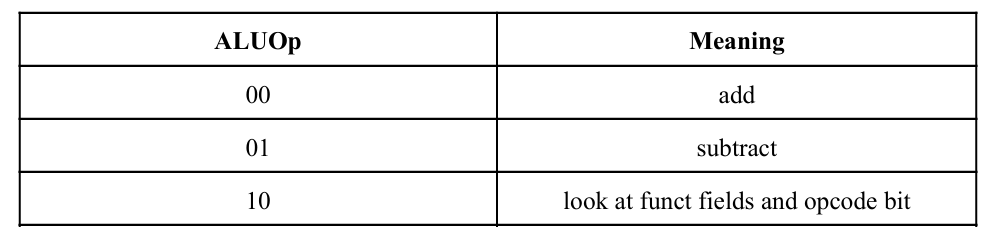
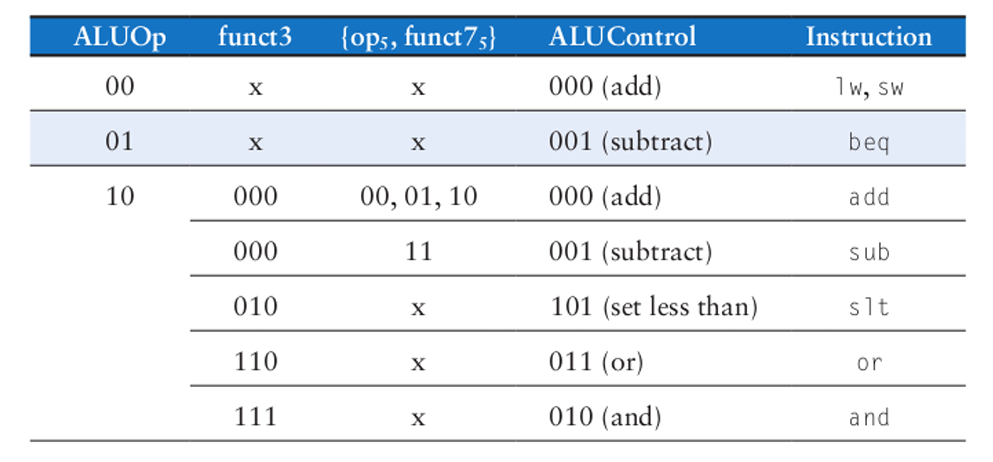


Table below is a truth table for the ALU decoder. The logic of ALUControl was covered in the above chapter. When ALUOp is 00 or 01, the ALU should add or subtract, respectively. When ALUOp is 10, the decoder examines the function fields and operand bit to determine the ALUControl. The control signals for each instruction were described as we built the datapath.



**DATA PATH**

A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter and registers). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Figure below shows a block diagram with the four state elements: **the program counter, register file, and instruction and data memories.**

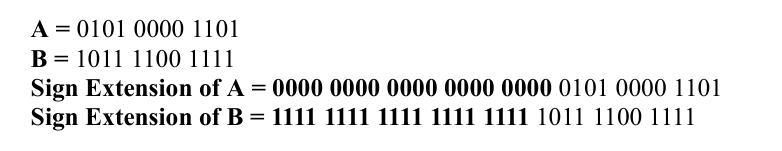
The first Datapath element we need is Instruction Memory. It is a memory unit to store the instructions of a program and supply instructions given an address. The instruction memory need only provide read access because the datapath does not write instructions. Since the instruction memory only reads, we treat it as combinational logic: the output at any time reflects the contents of the location specified by the address input, and no read control signal is needed. (We will need to write the instruction memory when we load the program; this is not hard to add, and we ignore it for simplicity.)

The second Datapath element we need is Program Counter (PC), which is a 32-bit register that holds the address of the current instruction. It is written at the end of every clock cycle and thus does not need a write control signal. We need is an Adder to increment the PC to the address of the next instruction. It has been permanently made an adder and cannot perform the other ALU functions.

The third element, data memory has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

The fourth Datapath element we need is Register file. The processor’s 32 general-purpose registers are stored in a structure called a register file. A register file is a collection of registers in which any register can be read or written by specifying the number of the register in the file. We need Alu to operate on the values read from the registers. R-format instructions have three register operands i.e. read two data words from the register file and write one data word into the register file for each instruction. For each data word to be read from the registers, we need an input to the register file that specifies the register number to be read and an output from the register file that will carry the value that has been read from the registers. To write a data word, we will need two inputs: one to specify the register number to be written and one to supply the data to be written into the register. we need a total of three inputs (two for register numbers and one for data) and two outputs (both for data). The register number inputs are 5 bits wide to specify one of 32 registers (32 = 2^5 ), whereas the data input and two data output buses are each 64 bits wide.

We also have other element, Immediate Generation. The immediate generation unit (ImmGen) has a 32-bit instruction as input that selects a 12-bit field for load, store, and branch if equal that is sign extended into a 32-bit result appearing on the output. Sign-extended means to increase the size of a data item by replicating the high-order sign bit of the original data item in the high-order bits of the larger, destination data item.

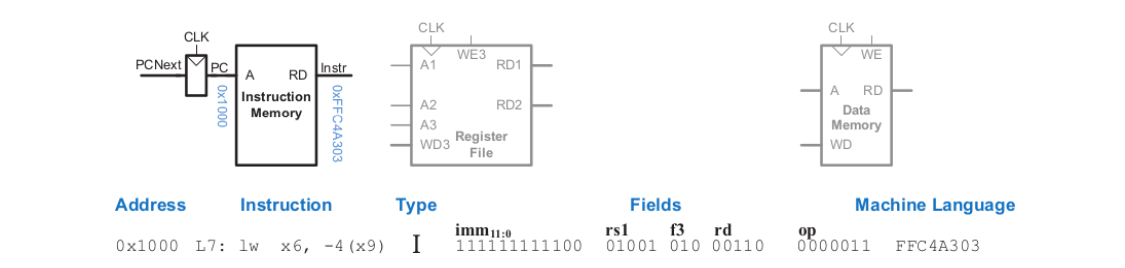


*FIG: Sign extension example*

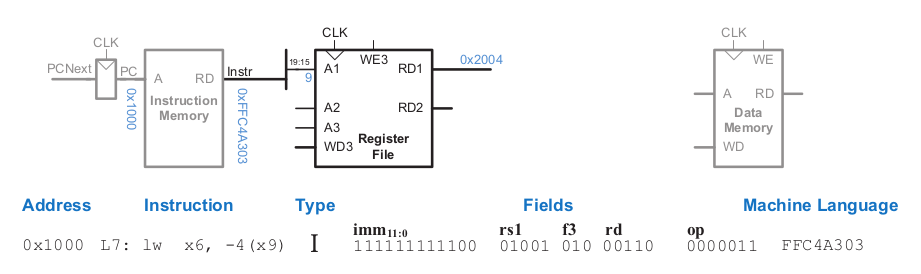
To execute any instruction, we must start by fetching the instruction from memory. To prepare for executing the next instruction, we must also increment the program counter so that it points at the next instruction, 4 bytes later. Figure below shows how to combine the three elements from Figure above to form a datapath that fetches instructions and increments the PC to obtain the address of the next sequential instruction.

**DATAPATH FOR LOAD WORD INSTRUCTION**

This section gradually develops the single-cycle datapath, adding one piece at a time to the state elements. The program counter (PC) register contains the address of the instruction to execute. The first step is to read the instruction from instruction memory. Figure below shows that the PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or fetches, the 32-bit instruction, labeled Instr.



For a load instruction, the next step is to read the source register containing the base address. This register is specified in the rs1 field of the instruction, Instr[19:15]. These bits of the instruction are connected to the address input of one of the register file read ports, A1, as shown in Figure below. The register file reads the register value onto RD1.



The load instruction also requires an offset. The offset is stored in the immediate field of the instruction, Instr[31:20]. Because the 12-bit immediate might be either positive or negative, it must be sign-extended to 32 bits, as shown in Figure below.

A diagram of a computer

Description automatically generated

The 32-bit sign-extended value is called ImmExt. The sign extension simply copies the sign bit (most significant bit) of a short input into all of the upper bits of the longer output. Specifically, ImmExt[15:0] = Instr[31:20] and ImmExt[31:16] = Instr[31].

The processor must add the base address to the offset to find the address to read from memory. Figure below introduces an ALU to perform this addition. The ALU receives two operands, SrcA and SrcB. SrcA comes from the register file, and SrcB comes from the sign-extended immediate. The ALU can perform many operations. The 3-bit ALUControl signal specifies the operation. The ALU generates a 32-bit ALUResult and a Zero flag, which indicates whether ALUResult == 0. For a load instruction, the ALUControl signal should be set to 000 to add the base address and offset.

A diagram of a machine

Description automatically generated

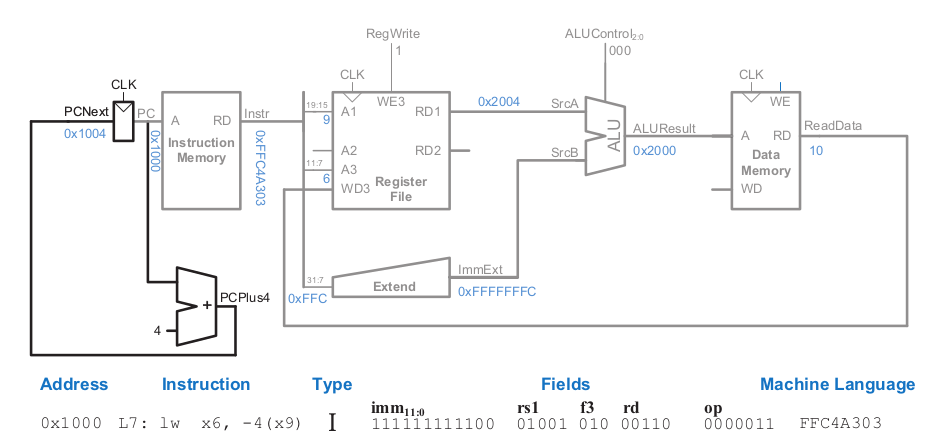
ALUResult is sent to the data memory as the address for the load instruction. The data is read from the data memory onto the ReadData bus, then written back to the destination register in the register file at the end of the cycle, as shown in Figure below. Port 3 of the register file is the write port.

A diagram of a machine

Description automatically generated

The destination register for the load instruction is specified in the rd field, Instr[11:7], which is connected to the port 3 address input, A3, of the register file. The ReadData bus is connected to the port 3 write data input, WD3, of the register file. A control signal called RegWrite is connected to the port 3 write enable input, WE3, and is asserted during a load instruction so that the data value is written into the register file.

The write takes place on the rising edge of the clock at the end of the cycle. While the instruction is being executed, the processor must compute the address of the next instruction, PCNext. Because instructions are 32 bits = 4 bytes, the next instruction is at PC + 4. Figure below shows that datapath uses another adder to increment the PC by 4. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the load instruction.



**DATAPATH FOR STORE WORD INSTRUCTION**

In this section, we will extend the data path to Store instructions. Like the load instruction, the store instruction reads a base address from port 1 of the register file and sign-extends an immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by the datapath. The store instruction also reads a second register from the register file and writes it to the data memory. The figure below shows the new connections for this function. The register is specified in the rs2 field, Instr[24:20]. These bits of the instruction are connected to the second register file read port, A2.

The register value is read onto the RD2 port. It is connected to the write data port of the data memory. The write enable port of the data memory, WE are controlled by MemWrite. For a store instruction, MemWrite = 1, to write the data to memory; ALUControl = 000, to add the base address and offset; and RegWrite = 0, because nothing should be written to the register file. Note that data is still read from the address given to the data memory, but that this ReadData is ignored because RegWrite = 0.

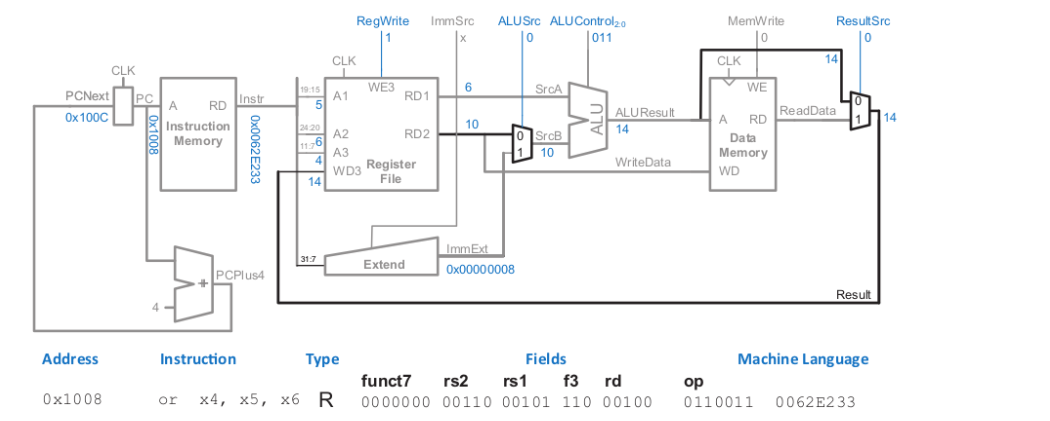
A diagram of a computer

Description automatically generated

**DATAPATH FOR R TYPE INSTRUCTION**

In this section we will extend the data path to handle R-type instructions add, sub, and, or, and slt. All of these instructions read two registers from the register file, perform some ALU operation on them, and write the result back to a third register in the register file. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware, using different ALUControl signals.

Figure below shows the enhanced datapath handling R-type instructions. The register file reads two registers. The ALU performs an operation on these two registers. In previous labs we saw that the ALU always received its SrcB operand from the sign-extended immediate (ImmExt). Now, we add a multiplexer to choose SrcB from either the register file RD2 port or ImmExt. The multiplexer is controlled by a new signal, ALUSrc. ALUSrc is 0 for R-type instructions to choose SrcB from the register file; it is 1 for Load and store to choose ImmExt.

****

This principle of enhancing the datapath’s capabilities by adding a multiplexer to choose inputs from several possibilities is extremely useful. Indeed, we will apply it twice more to complete the handling of R-type instructions. Previously, the register file always got its written data from the data memory. However, R-type instructions write the ALUResult to the register file. Therefore, we add another multiplexer to choose between ReadData and ALUResult. We call its output Result. This multiplexer is controlled by another new signal, ResultSrc. ResultSrc is 0 for R-type instructions to choose Result from the ALUResult; it is 1 for Load to choose ReadData. We don’t care about the value of ResultSrc for store, because store does not write to the register file.

**VERILOG MODULES**

ALU WITH FLAGS

A screenshot of a computer program

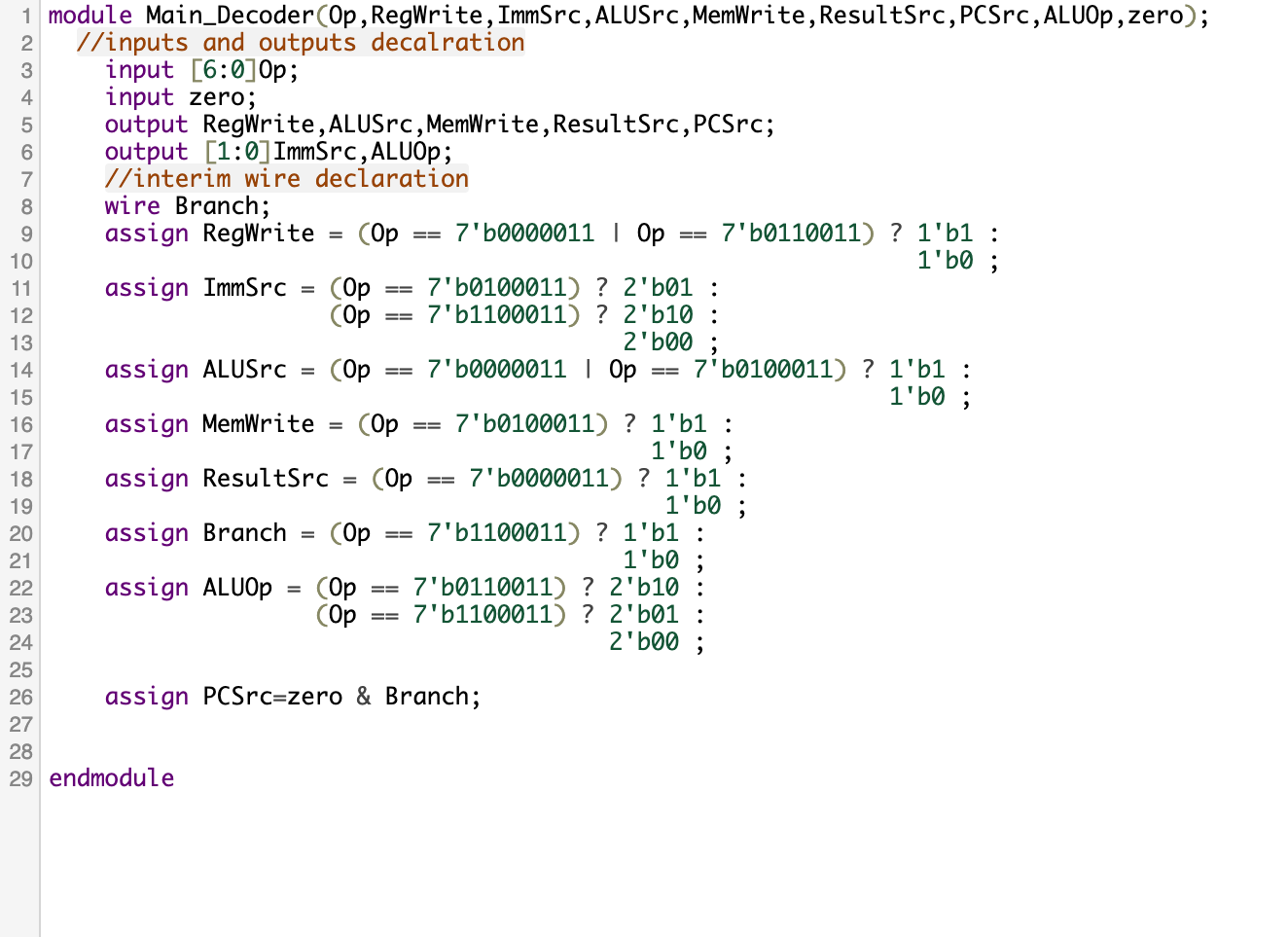
Description automatically generated

ALU DECODER

A screenshot of a computer program

Description automatically generated

MAIN DECODER

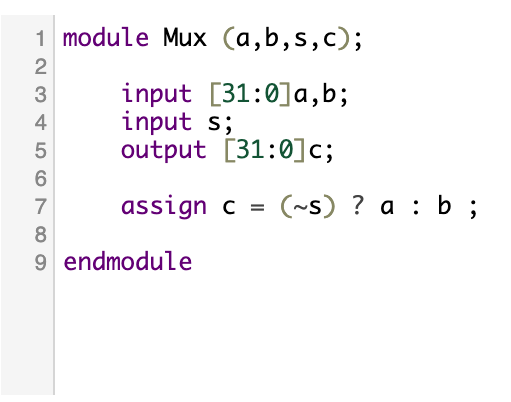


CONTROL UNIT TOP

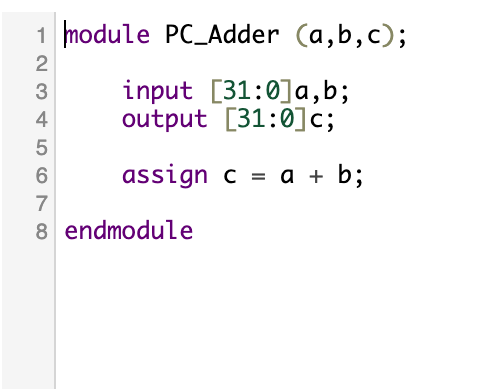
A screenshot of a computer program

Description automatically generated

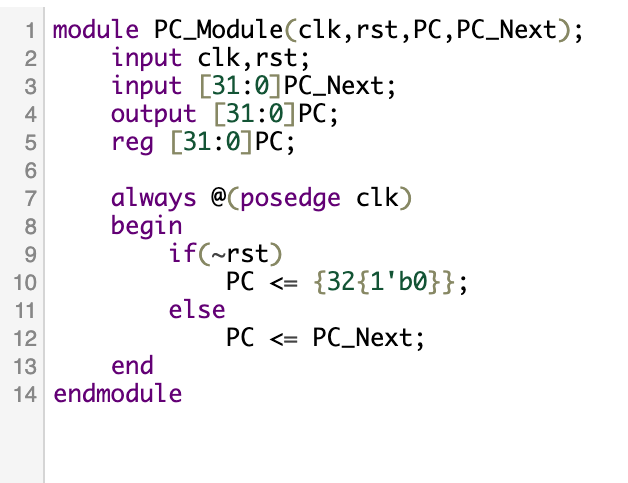
MUX



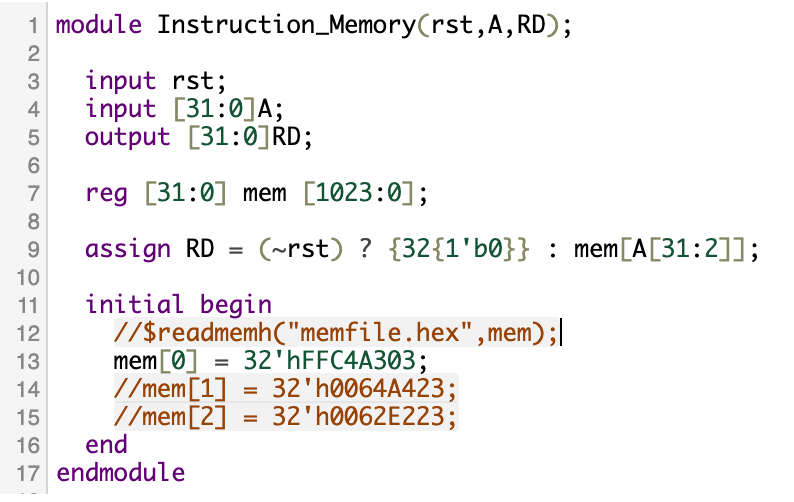
PC ADDER



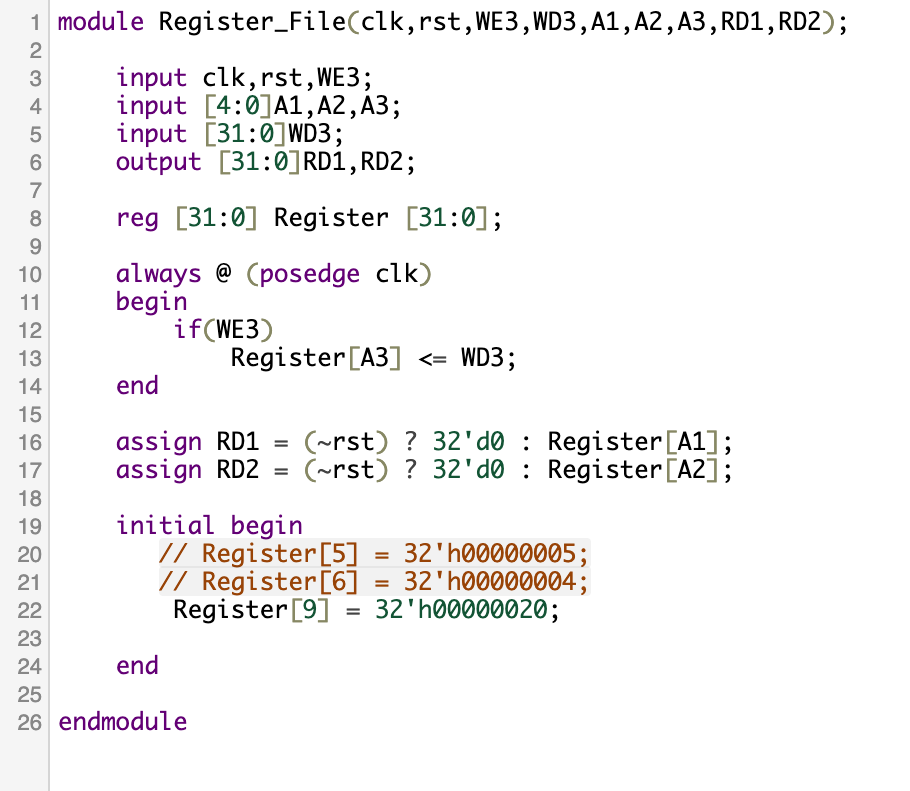
PC(PROGRAM COUNTER)



INSTRUCTION MEMORY



REGISTER FILE

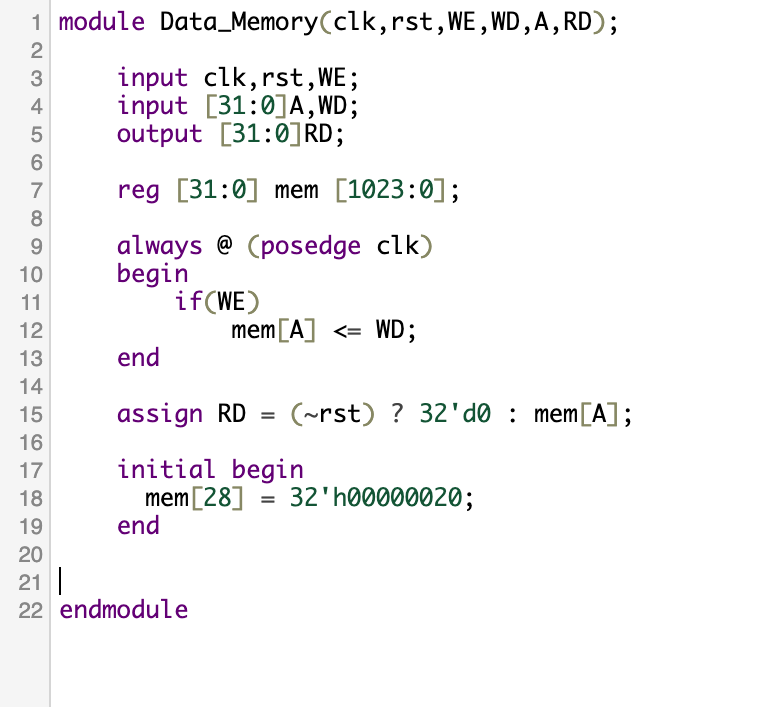


SIGN EXTEND

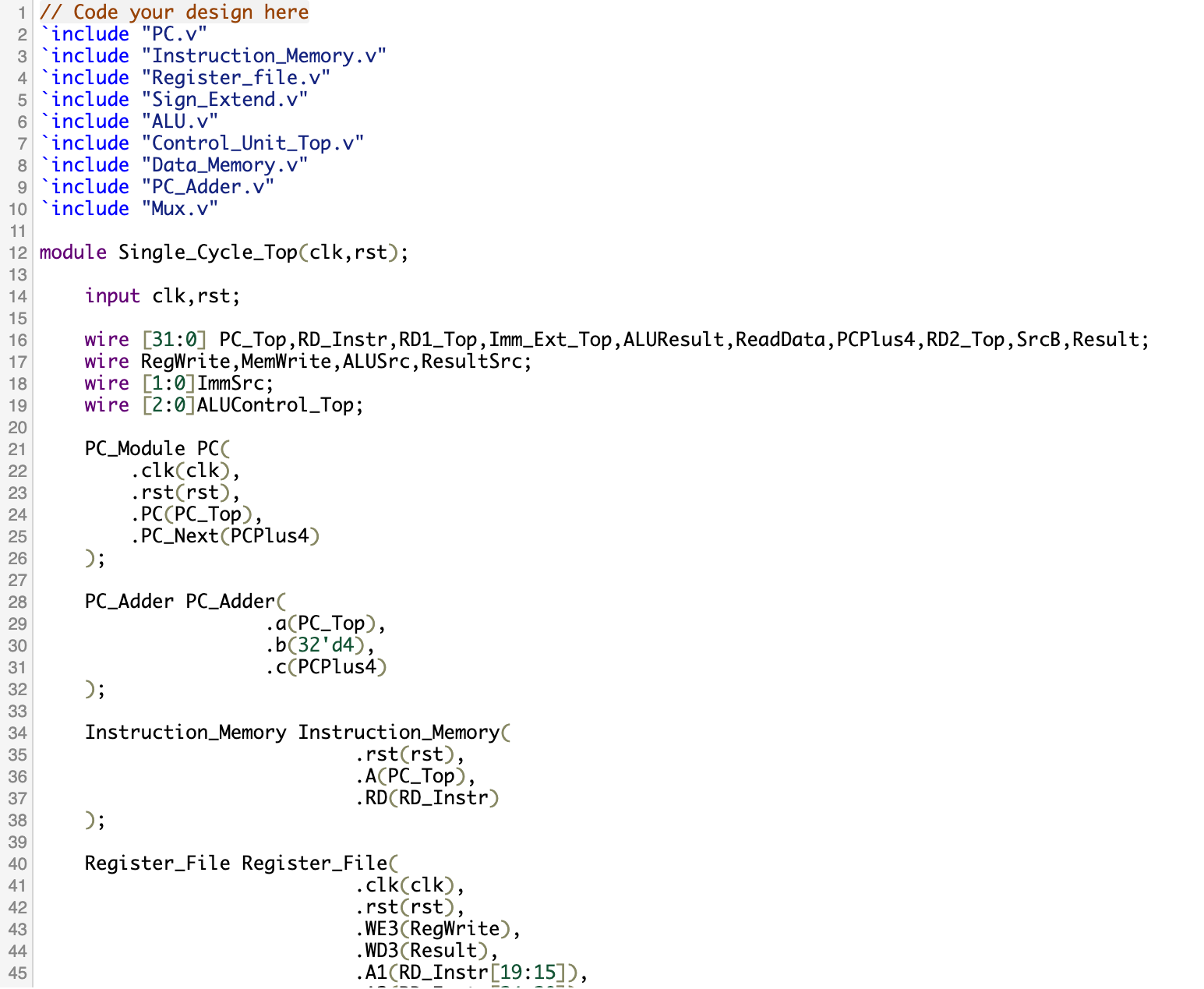
A computer code with numbers and symbols

Description automatically generated with medium confidence

DATA MEMORY



SINGLE CYCLE TOP



A screenshot of a computer program

Description automatically generated

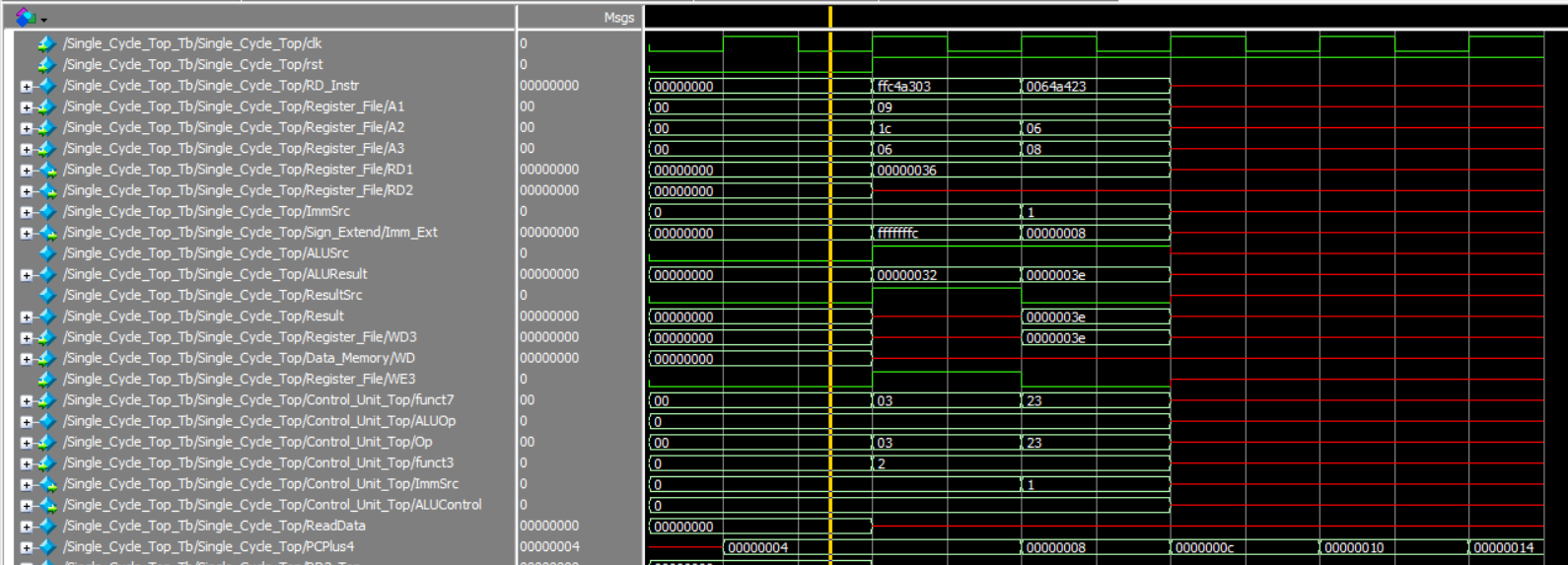
A screenshot of a computer code

Description automatically generated

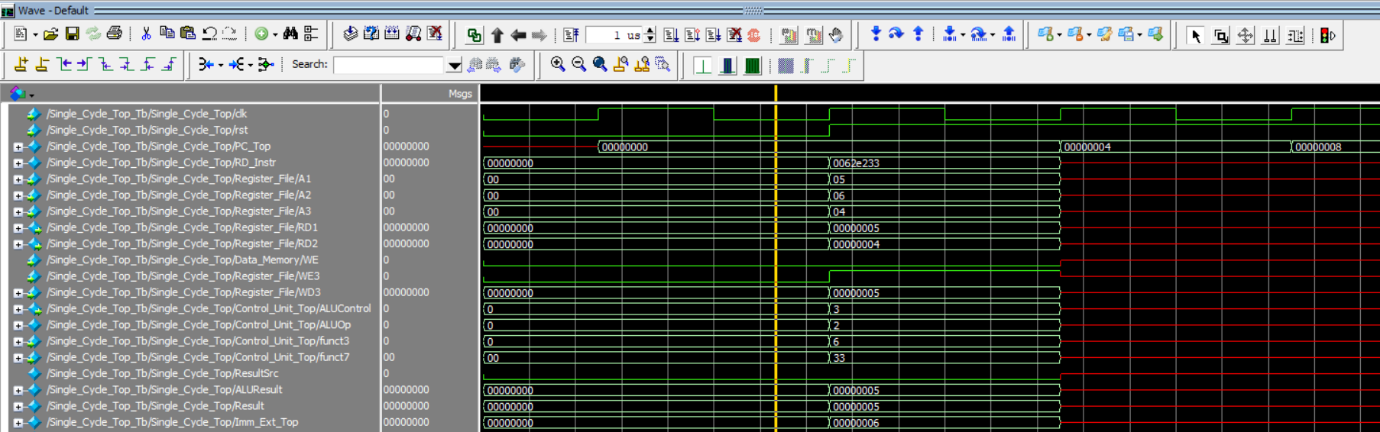
TESTBENCH

A screenshot of a computer program

Description automatically generated



*FIG: SIMULATION RESULT FOR LOAD WORD AND STORE WORD*



*FIG: SIMULATION RESULT FOR R-TYPE INSTRUCTION*

**REFERENCES**

* Computer Architecture: A Quantitative Approach-Book by David A Patterson and John L. Hennessy
* Digital Design and Computer Architecture, RISC-V Edition

Book by David Harris (Author), Sarah L. Harris (Author)

* [Lecture videos](https://www.youtube.com/watch?v=lrN-uBKooRY&list=PLhA3DoZr6boVQy9Pz-aPZLH-rA6DvUidB) by Sarah and David Harris