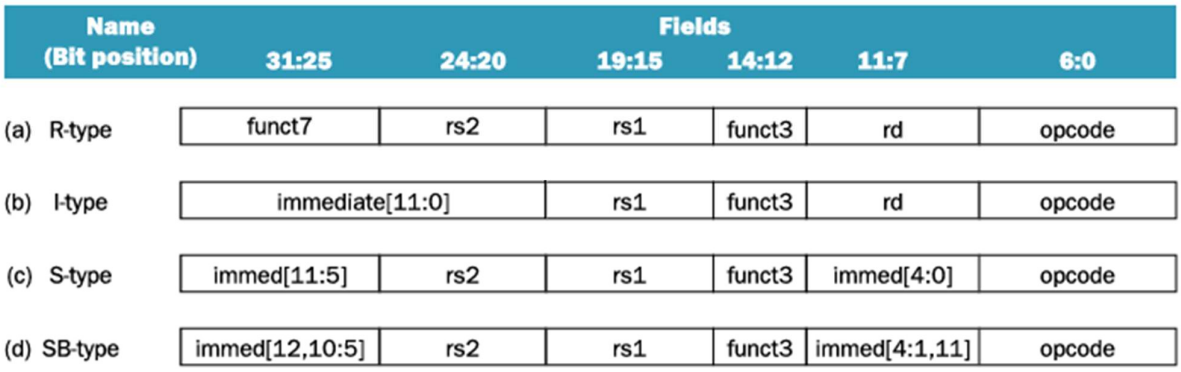
SINGLE CYCLE RISCV PROCESSOR _RV32I

Overview

This project implements a single-cycle RISC-V processor in Verilog. The processor executes one instruction per clock cycle, supporting a subset of the RISC-V instruction set (R-type, I-type, Load, Store, Branch, LUI, and Jump instructions). The design is modular, with each functional block separated for clarity and reusability.

RISC-V Instruction Formats



RISC-V instructions are 32 bits wide and come in several formats, each suited for different types of operations. The main formats relevant to your processor are:

FORMAT	FIELDS	PURPOSE
R-TYPE	funct7, rs2, rs1, funct3, rd, opcode	Register-register arithmetic/logic (e.g., add, sub)
I-TYPE	immediate[11:0], rs1, funct3, rd, opcode	Immediate arithmetic, loads, some jumps (e.g., addi, lw)
S-TYPE	immediate[11:5], rs2, rs1, funct3, immediate[4:0], opcode	Stores (e.g., sw)
SB-TYPE	immediate[12,10:5], rs2, rs1, funct3, immediate[4:1,11], opcode	Conditional branches (e.g., beq)
U-TYPE	immediate[31:12], rd, opcode	Upper immediate (e.g., lui)
UJ-TYPE	immediate[20,10:1,11,19:12], rd, opcode	Unconditional jumps (e.g., jal)

- **R-type**: Used for arithmetic and logical operations between two registers.
- **I-type**: Used for operations with immediates, loads, and some jumps.
- **S-type**: Used for store instructions, where data from a register is stored to memory.
- **SB-type**: Used for branch instructions, where the immediate encodes the branch offset.
- **U-type**: Used for instructions like LUI (Load Upper Immediate).
- **UJ-type**: Used for jump instructions like JAL (Jump and Link)

The Imm_Gen module decodes the immediate field based on the opcode, handling the different formats.

Name (Field size)	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Control Signals and Their Roles

The control signals orchestrate how data flows through the datapath and what operations are performed. The table in your image summarizes the main control signals for each instruction type:

Signal	Description
ALUSrc	Selects if the ALU's second operand is a register or an immediate
MemtoReg	Selects if data to write back to register comes from memory or ALU
RegWrite	Enables writing to the register file
MemRead	Enables reading from data memory
MemWrite	Enables writing to data memory
Branch	Signals if the instruction is a branch
ALUOp1/0	Used by ALU Control to determine the ALU operation

Instruction	ALUSrc	Memto-Reg	Reg-Write	Mem-Read	Mem-Write	Branch	ALUOp1	ALUOp0
R-format	0	0	1	0	0	0	1	0
lw	1	1	1	1	0	0	0	0
sw	1	X	0	0	1	0	0	0
beq	0	X	0	0	0	1	0	1

How these are generated:
Your Control module takes the opcode as input and outputs the appropriate control signals for each instruction type. For example:

- **R-type**: RegWrite=1, ALUSrc=0, MemRead=0, MemWrite=0, MemtoReg=0, Branch=0
- **lw**: RegWrite=1, ALUSrc=1, MemRead=1, MemWrite=0, MemtoReg=1, Branch=0
- **sw**: RegWrite=0, ALUSrc=1, MemRead=0, MemWrite=1, MemtoReg=X, Branch=0
- **beq**: RegWrite=0, ALUSrc=0, MemRead=0, MemWrite=0, MemtoReg=X, Branch=1

These control signals are routed to the relevant modules (ALU, register file, memory, multiplexers) to implement the correct behavior for each instructio

Connection to Your Verilog Code

- The **Control** module (see your code) generates these signals based on the opcode
- The **ALU_Control** module uses ALUOp1, ALUOp0, funct3, and funct7 to select the correct ALU operation
- The **Imm_Gen** module decodes the immediate field as per the instruction format
- The **datapath** modules use these signals to execute instructions as per the RISC-V specification.

Setting Opcodes in the Control Module

The instruction set table defines how each RISC-V instruction is structured, specifying the positions of opcode, register fields, function codes, and immediate values for every instruction type.
Always refer to this table when:

- Decoding which bits to use for immediate value extraction in the Imm_Gen module.
- Determining the opcode in the Control module to set the correct control signals for each instruction type.

This table ensures your processor correctly interprets and executes every supported instruction by standardizing field positions and opcode values

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
fm | pred | succ		rs1	000	rd	0001111	FENCE
1000 | 0011 | 0011		00000	000	00000	0001111	FENCE.TSO
0000 | 0001 | 0000		00000	000	00000	0001111	PAUSE
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

ISA

https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/instr-table.html

Top-Level Architecture

The Top module integrates all major components of the processor:

- **Program Counter (PC)**
- **Instruction Memory**
- **Control Unit**
- **Register File**
- **Immediate Generator**
- **ALU and ALU Control**
- **Data Memory**
- **Branch and PC Logic**
- **Multiplexers (muxes)**
-

The processor fetches instructions, decodes them, executes ALU operations, accesses memory, and writes results back to registers—all within a single clock cycle

Module Descriptions

1. **Program Counter (program_counter)**
   - **Purpose:** Holds the address of the current instruction.
   - **Operation:** On reset, sets PC to 0. Otherwise, updates PC to next_pc on every clock edge
2. **PC Adder (PC_Adder)**
   - **Purpose:** Calculates pc + 4 to point to the next sequential instruction
3. **Instruction Memory (instruction_memory)**
   - **Purpose:** Stores and provides instructions based on the current PC.
   - **Initialization:** Preloaded with sample RISC-V instructions for testing1.
   - **Addressing:** Instructions are word-aligned (read_address >> 2).
4. **Control Unit (Control)**
   - **Purpose:** Decodes the opcode and generates control signals for datapath modules (e.g., RegWrite, MemRead, MemWrite, ALUSrc, etc.)
   - **Supported Types:** R-type, I-type, Load, Store, Branch, Jump, LUI.
5. **Register File (registers)**
   - **Purpose:** Contains 32 general-purpose registers.
   - **Operation:** Supports two read ports and one write port. Register x0 is hardwired to zero.

6. **Immediate Generator (Imm_Gen)**
    - **Purpose:** Extracts and sign-extends immediate values for various instruction formats (I, S, B, U, J types)
7. **Multiplexers (mux2x1_32bit)**
    - **Purpose:** Select between two 32-bit inputs based on a control signal.
    - **Uses:** ALU input selection, write-back data selection, and PC source selection
8. **Branch Adder (Branch_Adder)**
    - **Purpose:** Computes the target address for branch instructions (PC + 4 + offset)
9. **ALU (ALU)**
    - **Purpose:** Performs arithmetic and logical operations as determined by the ALU control signal.
    - **Supported Operations:** ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT, SLTU, LUI
    - **Zero Flag:** Set if the result is zero (used for branch decisions).
10. **ALU Control (ALU_Control)**
    - **Purpose:** Generates a 4-bit control signal for the ALU based on instruction function fields and control signals
11. **Data Memory (data_memory)**
    - **Purpose:** Implements a 32-word data memory for load and store instructions.
    - **Operation:** Supports synchronous writes and asynchronous reads

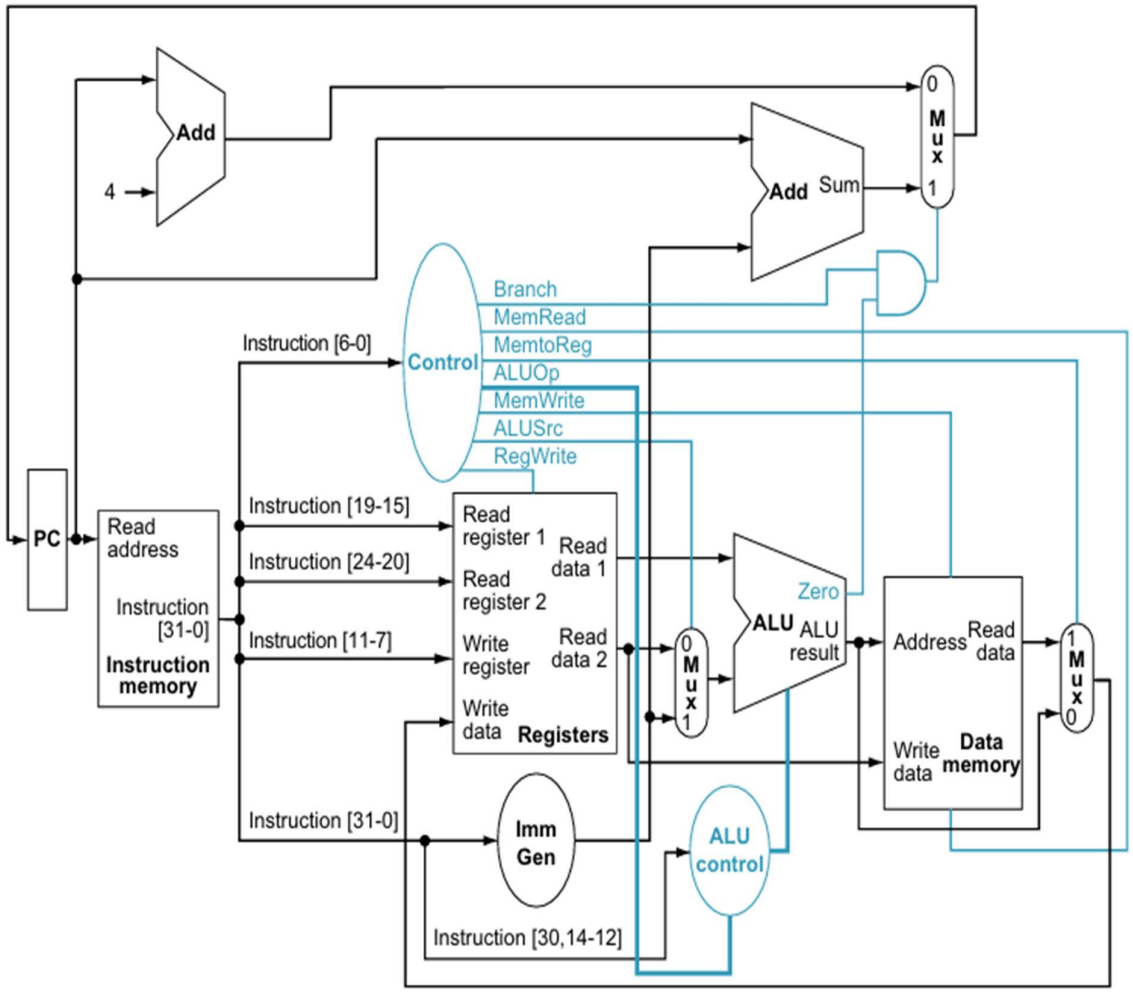
Data Path and Control Flow

1. **Instruction Fetch:**
    - PC provides address to instruction memory; instruction is fetched.
2. **Instruction Decode:**
    - Control unit decodes opcode.
    - Register file reads source operands.
    - Immediate generator extracts immediate if needed.
3. **Execution:**
    - ALU performs computation (arithmetic, logic, address calculation).
    - ALU control determines operation based on instruction type.
4. **Memory Access:**

- For load/store, data memory is accessed.

5. **Write Back:**
    - Result (from ALU or memory) is written to register file.
6. **PC Update:**
    - PC is updated to next sequential instruction or branch target.

ARCHITECTURE



Reference: **Pg : 314/1024**

https://theswissbay.ch/pdf/Books/Computer%20science/Computer%20Organization%20and%20Design-%20The%20HW_SW%20Inteface%205th%20edition%20-%20David%20A.%20Patterson%20%26%20John%20L.%20Hennessy.pdf

Wire Connections in the Top Module

Wire Name	Width	Source / Assignment	Connected To (Inputs of...)	Description / Purpose
clk	1	Top module input	program_counter, registers, data_memory	Clock signal for sequential elements
rst	1	Top module input	program_counter, registers, data_memory	Reset signal for sequential elements
pc	32	program_counter output	PC_Adder, instruction_memory, Branch_Adder	Current Program Counter value
next_pc	32	mux2x1_32bit (PC source mux) output	program_counter input	Next Program Counter value
pc_plus4	32	PC_Adder output	mux2x1_32bit (PC source mux) input0	PC + 4 (next sequential instruction address)
instruction	32	instruction_memory output	Control, Imm_Gen, ALU_Control, Top (for rs1, rs2, rd)	Current instruction fetched
rs1	5	Assigned from instruction[19:15]	registers (read_register1)	Source register 1 index
rs2	5	Assigned from instruction[24:20]	registers (read_register2)	Source register 2 index
rd	5	Assigned from instruction[11:7]	registers (write_register)	Destination register index
RegWrite	1	Control output	registers (reg_write)	Register write enable
MemRead	1	Control output	data_memory (MemRead)	Data memory read enable
MemWrite	1	Control output	data_memory (MemWrite)	Data memory write enable
MemToReg	1	Control output	mux2x1_32bit (result source mux) sel	Selects ALU result or memory data for register write-back
ALUSrc	1	Control output	mux2x1_32bit (ALU srcB mux) sel	Selects ALU input (reg or immediate)

Branch	1	Control output	mux2x1_32bit (PC source mux) sel	Indicates branch instruction
ALUOp0	1	Control output	ALU_Control input	ALU operation code (part 0)
ALUOp1	1	Control output	ALU_Control input	ALU operation code (part 1)
read_data1	32	registers output	ALU input (data1)	Data from source register 1
read_data2	32	registers output	mux2x1_32bit (ALU srcB mux) input0, data_memory (write_data)	Data from source register 2
imm_value	32	Imm_Gen output	mux2x1_32bit (ALU srcB mux) input1, Branch_Adder (offset)	Immediate value extracted from instruction
alu_srcB	32	mux2x1_32bit (ALU srcB mux) output	ALU input (data2)	ALU second operand (register or immediate)
alu_control	4	ALU_Control output	ALU input (Alu_control)	ALU operation select code
alu_result	32	ALU output	data_memory (address), mux2x1_32bit (result source mux) input0	ALU computation result
zero	1	ALU output	mux2x1_32bit (PC source mux) sel (with Branch)	ALU zero flag (used for branch decision)
branch_target	32	Branch_Adder output	mux2x1_32bit (PC source mux) input1	Target address for branch instructions
mem_read_data	32	data_memory output	mux2x1_32bit (result source mux) input1	Data read from memory (for loads)
write_data	32	mux2x1_32bit (result source mux) output	registers (write_data)	Data written to register file

How are instructions stored in memory?

In RISC-V, each instruction is at least 2 bytes (16 bits) long, but usually 4 bytes (32 bits).

All instructions start at addresses that are multiples of 2 or 4 (for example: 0, 4, 8, 12, ...).

This means:

- You can never have an instruction at address 1, 3, 5, etc.
- All valid instruction addresses **end with a 0 in binary**.

==========

RISC-V Example: beq x0, x1, +8

Step 1: Understanding the Offset

- You want to branch forward by **8 bytes**.
- In RISC-V, the immediate for branch instructions is in **multiples of 2 bytes**.
- So, the immediate value to encode is:

  $8 / 2 = 4$

Step 2: Encoding the Immediate

- The immediate field in the instruction will be **4** (binary: 0100).
- RISC-V branch encoding splits the immediate bits across the instruction:
  - imm = sign bit (here, 0 since +8 is positive)
  - imm[10:5] = 000000
  - imm[4:1] = 0100
  - imm = 0
- The **hardware appends one zero** (shifts left by 1) when calculating the actual byte offset.

Step 3: In Binary

- Immediate in instruction: 000000000100 (12 bits for +8 branch)
- When used, hardware shifts left by 1: 0000000001000 (13 bits, value = 8 bytes)

Step 4: PC Calculation

If the current PC is, say, 0x100:

- If x0 == x1, new PC = 0x100 + 8 = 0x108

Why Only One Zero?

- Because RISC-V instructions are aligned to 2 bytes, so only one zero is needed.

==========

MIPS32 Example: beq $zero, $at, +8

Step 1: Understanding the Offset

- You want to branch forward by **8 bytes**.
- In MIPS32, the immediate for branch instructions is in **multiples of 4 bytes**.
- So, the immediate value to encode is:

  $8 / 4 = 2$

Step 2: Encoding the Immediate

- The immediate field in the instruction will be **2** (binary: 0000 0000 0000 0010).
- MIPS hardware **appends two zeros** (shifts left by 2) when calculating the actual byte offset.

Step 3: In Binary

- Immediate in instruction: 0000 0000 0000 0010 (16 bits for +8 branch)
- When used, hardware shifts left by 2: 0000 0000 0000 1000 (value = 8 bytes)

Step 4: PC Calculation

If the current PC is, say, 0x100:

- If $zero == $at, new PC = 0x100 + 4 + 8 = 0x10C

  (MIPS always adds 4 to PC before applying the offset)

Why Two Zeros?

- Because MIPS32 instructions are aligned to 4 bytes, so two zeros are needed.