CO502 - Part 1 (Planning)

Group 1

Table of Contents

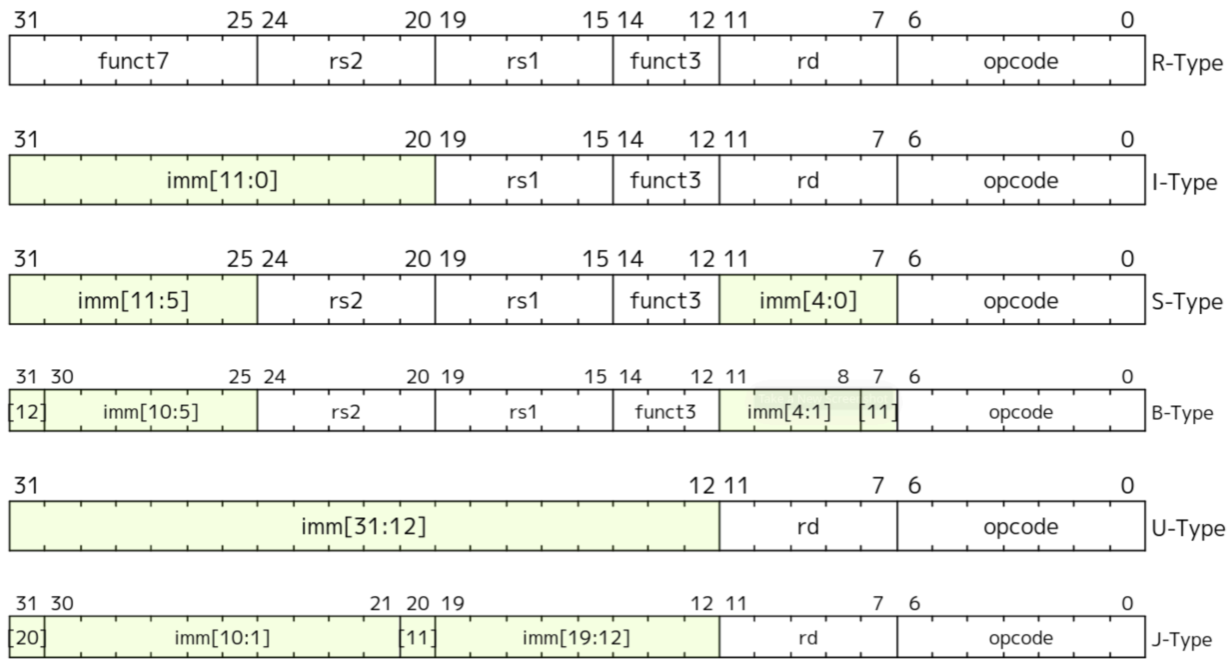
Activity 1

There are four core instruction formats:

- R-type - Arithmetic and load operations. All operations read the rs1 and rs2 registers as source operands and write the result into register rd. The funct7 and funct3 fields select the type of operation.
- I-type - Arithmetic, load and system operations. Reads one 12-bit immediate operand and one register operand (rs1) and writes to rd
- S-type - Store instructions transfer a value between the registers and memory.
- U-type - Upper immediate instructions. Loads a 20-bit immediate value into a register.

Further, two variants of the instruction formats exist based on the handling of immediate:

- J-type - Unconditional jumps and writes return address to rd
- B-type - Conditional branching based on the comparison between values in rs1 and rs2.


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

<u>Instruction Encoding Formats</u>

Instructions and Opcodes

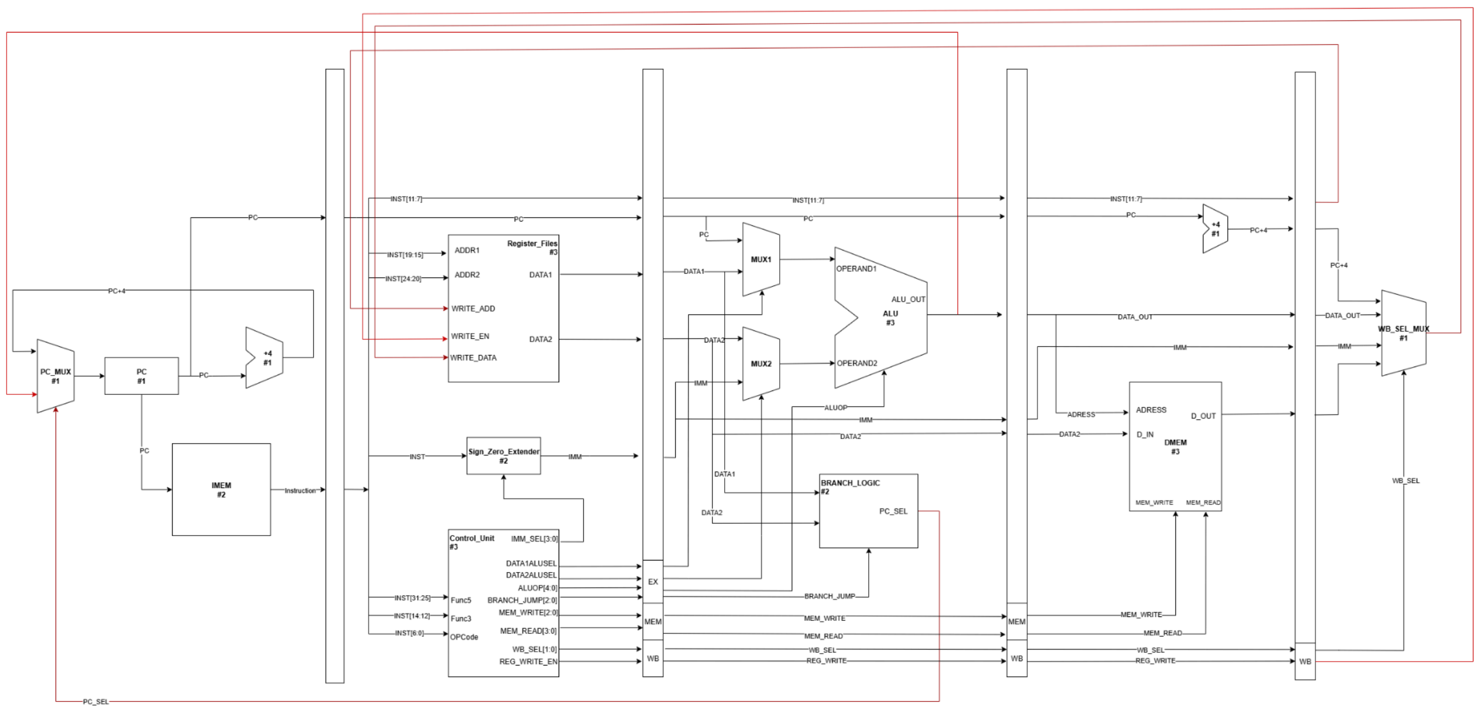
Instruction	Description	Opcode	funct3	funct7	Notes
R-Type					
ADD	rd = rs1 + rs2	0110011	0x0	0x00	
SUB	rd = rs1 - rs2	0110011	0x0	0x20	
XOR	rd = rs1 ^ rs2	0110011	0x4	0x00	
OR	rd = rs1 | rs2	0110011	0x6	0x00	
AND	rd = rs1 & rs2	0110011	0x7	0x00	
SLL	rd = rs1 << rs2	0110011	0x1	0x00	
SRL	rd = rs1 >> rs2	0110011	0x5	0x00	
SRA	rd = rs1 >> rs2	0110011	0x5	0x20	Sign extend
SLT	rd = (rs1 < rs2) ? 1:0	0110011	0x2	0x00	
SLTU	rd = (rs1 > rs2) ? 1:0	0110011	0x3	0x00	
MUL	rd = (rs1 × rs2)[31:0]	0110011	0x0	0x01	
MULH	rd = (rs1 × rs2)[63:32]	0110011	0x1	0x01	
MULHSU	rd = (rs1 × rs2)[63:32]	0110011	0x2	0x01	
MULHU	rd = (rs1 × rs2)[63:32]	0110011	0x3	0x01	
DIV	rd = rs1 ÷ rs2	0110011	0x4	0x01	
DIVU	rd = rs1 ÷ rs2 (unsigned)	0110011	0x5	0x01	

REM	rd = rs1 % rs2	0110011	0x6	0x01	
REMU	rd = rs1 % rs2 (unsigned)	0110011	0x7	0x01	
I-Type					
ADDI	rd = rs1 + imm	0010011	0x0		Imm value is sign-extended
XORI	rd = rs1 ^ imm	0010011	0x4		Imm value is sign-extended
ORI	rd = rs1 | imm	0010011	0x6		Imm value is sign-extended
ANDI	rd = rs1 & imm	0010011	0x7		Imm value is sign-extended
SLLI	rd = rs1 << imm[4:0]	0010011	0x1	imm[5:11]=0x00	
SRLI	rd = rs1 >> imm[4:0]	0010011	0x5	imm[5:11]=0x00	
SRAI	rd = rs1 >> imm[4:0]	0010011	0x5	imm[5:11]=0x20	
SLTI	rd = (rs1 < imm)?1:0	0010011	0x2		Imm value is sign-extended
SLTIU	rd = (rs1 > imm)?1:0	0010011	0x3		Imm value is zero-extended
LB	rd = mem[rs1+imm][7:0]	0000011	0x0		
LH	rd = mem[rs1+imm][15:0]	0000011	0x1		
LW	rd = mem[rs1+imm][31:0]	0000011	0x2		
LBU	rd = mem[rs1+imm][7:0]	0000011	0x4		zero-extends

LHU	rd = mem[rs1+imm][15:0]	0000011	0x5		zero-extends
JALR	rd = PC+4; PC = rs1 + imm	1100111			
ECALL	Transfer control to OS	1110011	0x0	Imm = 0x0	
EBREAK	Transfer control to debugger	1110011	0x0	Imm = 0x1	
S-Type					
SB	mem[rs1+imm][7:0] = rs2[7:0]	0100011	0x0		
SH	mem[rs1+imm][15:0] = rs2[15:0]	0100011	0x1		
SW	mem[rs1+imm][31:0] = rs2[31:0]	0100011	0x2		
SBU		0100011	0x4		
SHU		0100011	0x5		
U-Type					
LUI	rd = imm << 12	0110111	-		Load Upper Imm
AUIPC	rd = PC + (imm << 12)	0010111	-		Add Upper Imm to PC
B-Type					
BEQ	if(rs1 == rs2) PC += imm	1100011	0x0		
BNE	if(rs1 != rs2) PC += imm	1100011	0x1		
BLT	if(rs1 < rs2)	1100011	0x4		

	PC += imm				
BGE	if(rs1 >= rs2) PC += imm	1100011	0x5		
BLTU	if(rs1 < rs2) PC += imm	1100011	0x6		zero-extends
BGEU	if(rs1 >= rs2) PC += imm	1100011	0x7		zero-extends
J-Type					
JAL	rd = PC+4; PC += imm	1101111	-		
FENCE		0001111	0x0		
FENCE.I		0001111	0x1		

Activity 2

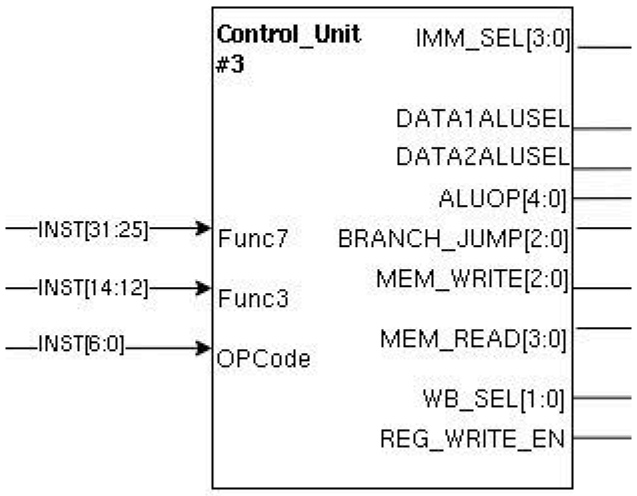


Pipeline Diagram with Datapath and Control

Hardware Units and Signals

Control Unit

The control unit interprets instruction fields and generates control signals to coordinate data flow and operations within the processor. The table below summarizes key control signals and decoding fields used for instruction execution in a RISC-V pipeline.

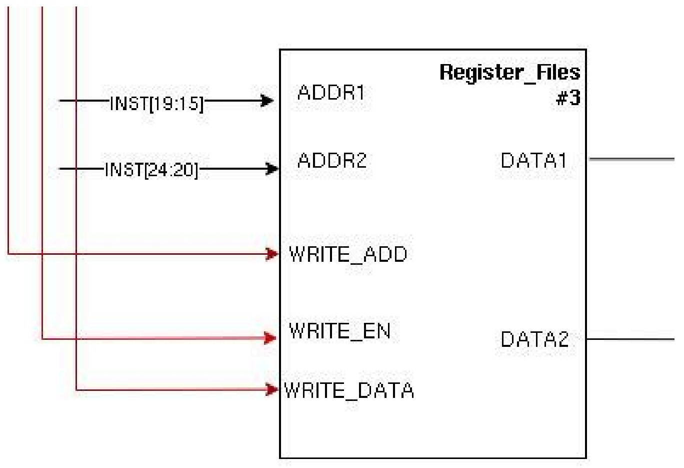


Control Signals and Instruction Decoding Fields

Signal / Field	Description
OPCODE	Specifies the main operation to be performed; used for primary instruction decoding.
FUNC7	Further decodes R-type instructions to determine the exact ALU operation.
FUNC3	Distinguishes between multiple operations under the same opcode.
IMM_SEL	Selects the format of the immediate value (I-type, S-type, B-type, etc.).
DATA1_ALU_SEL	Selects the source for the first ALU operand (typically from register or PC).
DATA2_ALU_SEL	Selects the source for the second ALU operand (register, immediate, etc.).
ALU_OP	Specifies the operation that the ALU should perform (add, sub, and, or, etc.).
BRANCH_JUMP	Indicates if the current instruction is a branch or jump; also determines the branch type (e.g., BEQ, BNE, JAL).
MEM_WRITE	Enables writing data to memory.
MEM_READ	Enables reading data from memory.
WB_SEL	Selects the correct source (ALU, memory, or immediate) to write back to the register file.
REG_WRITE_EN	Enables writing to the register file (i.e., allows the destination register to be updated).

Register File

The register file provides fast storage for operands and results during instruction execution. The following signals are used to control register selection, reading, and writing.



Register File Control Signals and Ports

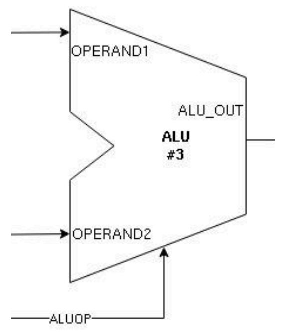
Signal / Port	Description
ADDR1	Selects the address of the first source register for a read operation.
ADDR2	Selects the address of the second source register for a read operation.
WRITE_ADD	Specifies the address of the destination register for a write operation.
WRITE_EN	Enables writing data to the selected destination register.
WRITE_DATA	The data value to be written into the destination register.
DATA1	The data read from the register specified by ADDR1.
DATA2	The data read from the register specified by ADDR2.

ALU

The Arithmetic Logic Unit (ALU) performs arithmetic and logical operations on input operands based on control signals. The table below outlines the key signals involved in ALU operation.

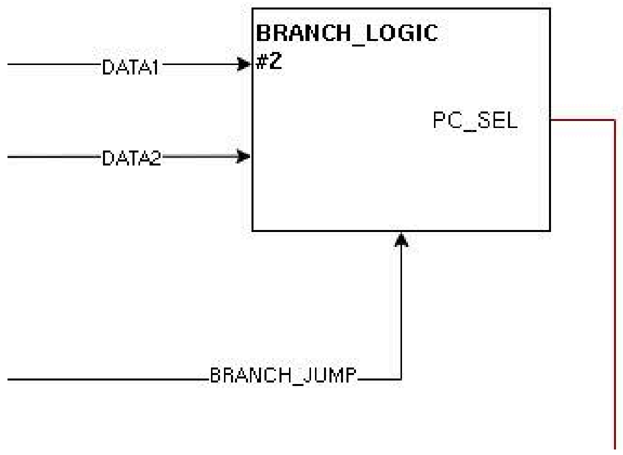
Arithmetic Logic Unit (ALU) Signals

Signal	Description
OPERAND1	The first input value to the ALU, typically from a register or program counter.
OPERAND2	The second input value to the ALU, from a register or an immediate value.
ALUOP	Control signal that determines the specific operation to be performed (e.g., add, sub, and, or).
ALU_OUT	The output result of the ALU operation.



Branch Logic

The branch logic unit determines the branch taken or not taken and the next program counter (PC) value based on branch or jump conditions. It evaluates comparisons and signals the control unit to alter instruction flow when necessary.

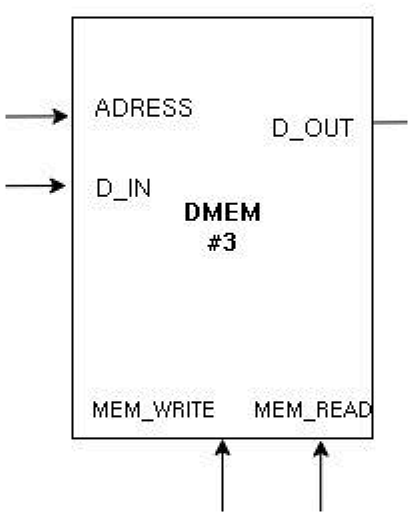


Branch Logic Control Signals

Signal	Description
BRANCH_JUMP	Indicates whether the current instruction is a branch or jump, and specifies the branch type (e.g., BEQ, JAL).
DATA1	First source register value used for branch condition evaluation.
DATA2	Second source register value used for branch condition evaluation.
PC_SEL	Control signal that selects whether to use the sequential PC or a computed branch/jump target.

Data Memory

The data memory module handles load and store operations during program execution. The control and data signals below manage memory access and data transfer between the processor and memory.



Data Memory Interface Signals

Signal	Description
MEM_READ	Enables a memory read operation.
MEM_WRITE	Enables a memory write operation.
ADDRESS	Specifies the memory address for the read or write operation.
D_IN	Data to be written to memory (for store instructions).
D_OUT	Data read from memory (for load instructions).

Activity 3

Unit Testing

Isolate single modules and test them using a Verilog simulator (Icarus Verilog and GTKWave) and test benches. The following factors are considered when writing the test benches:

1. Testing edge cases such as zero inputs, maximum values
2. Test units in isolation

Unit	Expected Tests
Control Unit	Proper generation of control signals at correct

	timing
Register File	Proper address decoding, proper reset
ALU	Mathematical accuracy of operations, edge cases
Branch Logic	Correct PC update decisions
Data Memory	Proper address decoding, proper reset

Formal Verification

Use of a framework such as RISCV-Formal (https://github.com/YosysHQ/riscv-formal) to formally verify the correctness of the core. The following outlines the high-level steps that need to be taken:

1. Wrap the core with the RISC-V Formal Interface (RVFI), which outputs architectural state changes (registers, PC, memory) and instruction details.
2. Use the genchecks.py script from riscv-formal to automatically generate verification properties for the core. These checks will verify that the core behaves correctly instruction-by-instruction and over sequences.
3. Run formal verification and the correctness of the core.