RISCV Processor

Presented by:

- Assem Amen
- Shahd Mohamed

Table Of Contents

Table Of Figures

Introduction

This documentation outlines our data path configuration for a single-cycle RISC-V processor implementation. Within this document, we provide a comprehensive explanation of the instructions supported by our code, considering that our presented data path includes the essential hardware components necessary to handle various RISC-V instructions. Additionally, at the conclusion of the document, you will find a detailed description of our project's design, which was created using the VIVADO tool.

A RISC-V single-cycle processor is a computer processor designed based on the RISC-V architecture, which stands for Reduced Instruction Set Computing. The RISC-V architecture is an open standard instruction set architecture (ISA) that is becoming increasingly popular due to its simplicity and versatility.

In a single-cycle processor design, each instruction is executed in a single clock cycle. This means that the cycle time has to be enough to complete all the instruction stages such as fetch, decode, execute, and write back, they're all completed within a single clock cycle.

While single-cycle processors may not be the most efficient in terms of performance, they serve as an excellent starting point for understanding computer architecture and processor design principles. The key characteristic of a RISC-V processor is the modular ISA it has. This modularity makes the hardware design much simpler as we will illustrate.

Datapath And Control Unit

In figure (1) we find the hardware implementation of a single cycle RISC-V processor. The key to the implementation simplicity comes from the modularity of the ISA.

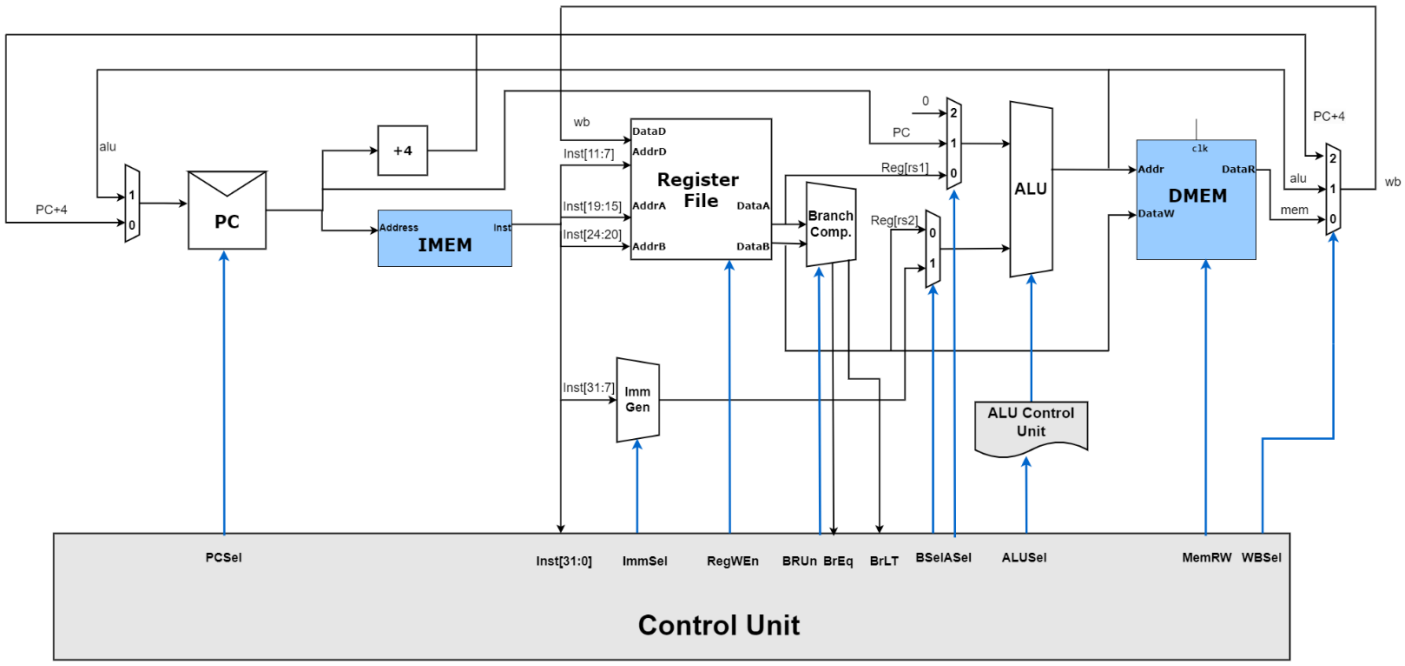

Figure 1:RISCV datapath and control unit.

The Instruction Set Architecture (ISA):

Within the expansive RISC-V Instruction Set Architecture (ISA), we've taken a deliberate approach by carefully choosing a range of instructions that span various formats, including computational, control flow, and memory access. This selective approach will guide us as we design a custom Datapath and precisely specify the control signals needed to execute our chosen instructions efficiently.

The instruction set in RISC-V is organized into distinct instruction formats, with each format comprising individual "fields." These fields are essentially separate unsigned integers, each serving as a dedicated container for conveying precise information about the intended operation to be executed. The 6 instruction formats are shown in FIG1:

1. R-Format: instructions using 3 register inputs.
2. I-Format: instructions with immediates, loads.
3. S-Format: store instructions.
4. SB-Format: branch instructions.
5. U-Format: instructions with upper immediate.

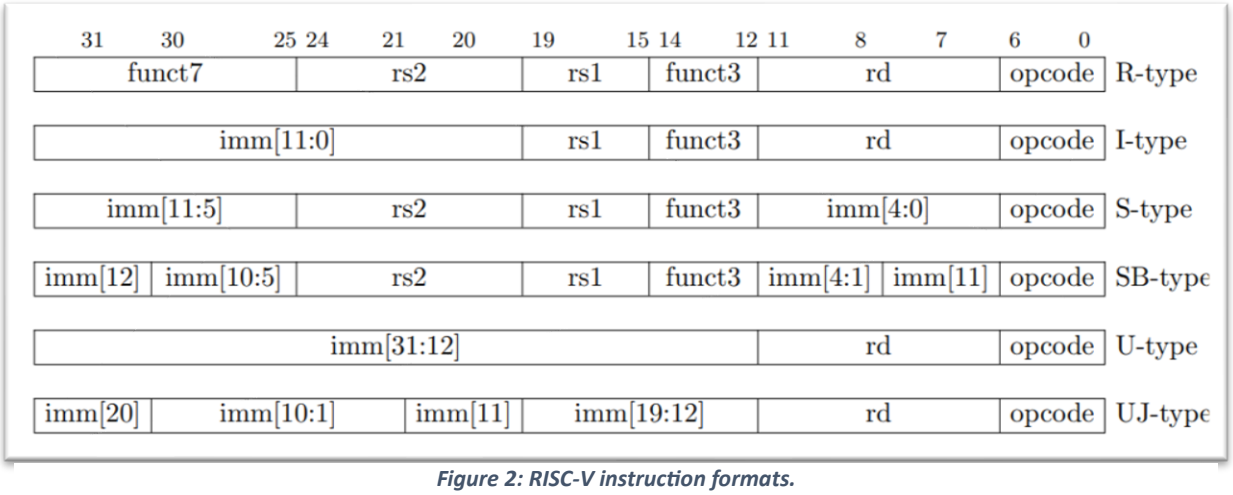

Figure 2: RISC-V instruction formats.

6. UJ-Format: the jump instruction.

- Now, let's individually highlight each instruction format.

A. R-type

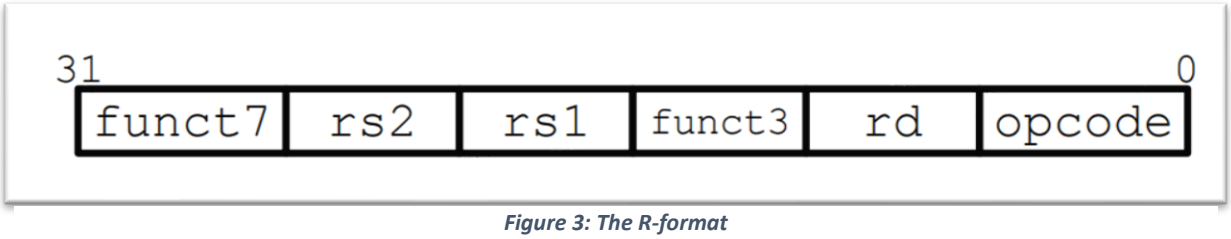

Figure 3: The R-format

- opcode (7): partially specifies operation.
- funct7&funct3 (10): combined with opcode, these two fields describe what operation to perform.
- rs2 (5): 2nd operand (second source register)
- rd (5): "destination register" — receives the result of computation.
- R: represents the register file

➢ The instructions supported and their description:

Mnemonic	Name	Description
add	Add	R[rd] = R[rs1] + R[rs2]
sub	Subtract	R[rd] = R[rs1] - R[rs2]
sll	Shift left	R[rd] = R[rs1] << R[rs2]
xor	XOR	R[rd] = R[rs1] ^ R[rs2]
srl	Shift right	R[rd] = R[rs1] >> R[rs2]
sra	Shift arithmetic right	R[rd] = R[rs1] >>> R[rs2]
or	OR	R[rd] = R[rs1] | R[rs2]
and	AND	R[rd] = R[rs1] & R[rs2]

B. I-type

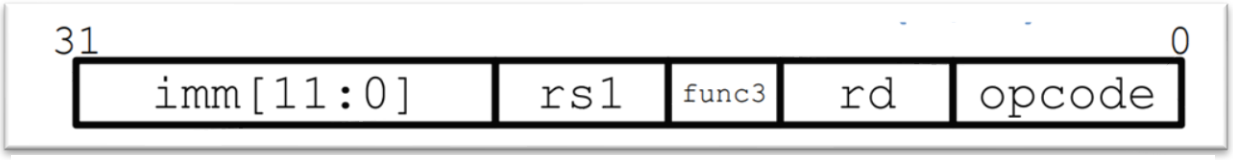

Figure 4: The general I-format

0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

Figure 5: The I-format at shift immediate

- Opcode (7) and func (3) both specify the operation.
- immediate (12): 12-bit number –All computations done in words, so 12-bit immediate must be extended to 32 bits –always sign-extended to 32-bits before use in an arithmetic operation.
- Shift amount is limited to 5bits in all shift operations whether it's R or I type instructions as it's logical to only shift an amount ranging from 0 to 31.
- M: represents the data memory

➢ **The instructions supported and their description:**

Mnemonic		Name	Description
addi		Add	R[rd] = R[rs1] + imm
slli		Shift left	R[rd] = R[rs1] << imm
xori		XOR	R[rd] = R[rs1] ^ imm
srli		Shift right	R[rd] = R[rs1] >> imm
srai	Shift arithmetic right	R[rd] = R[rs1] >>> imm	
ori	OR	R[rd] = R[rs1] | imm	
andi	AND	R[rd] = R[rs1] & imm	
lw	Load Word	R[rd] = M[R[rs1] + imm]	
jalr	Jump and link register	R[rd] = PC + 4, PC = R[rs1] + imm	

C. S-type

- To simplify the hardware, the shared fields between multiple instructions must be kept in the same bit indices. That's why the immediate value is written in this format.

Figure 6: The S-format

➢ **The instruction supported and its description:**

Mnemonic	Name	Description
sw	Store word	M[imm + R[rs1]] = R[rs2]

D. SB-type

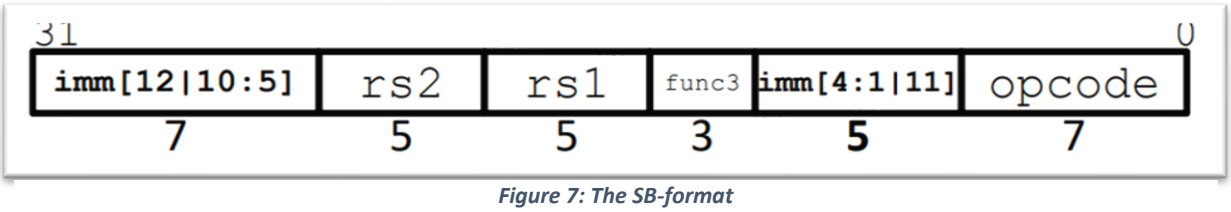

Figure 7: The SB-format

- Branches typically used for loops (if-else, while, for)
- immediate is number of instructions to move either forward (+) or backwards (–)
- 32-bit Instructions are "word-aligned": Address is always a multiple of 4 (in bytes), we won't be covering the half-word alignment in this design as it's a feature and not from the base ISA.
- All SB-format conditional statements chosen are signed comparisons as it's definitely most common.

➢ **The instructions supported and their description:**

Mnemonic	Name	Description
beq	Branch if equal	If(R[rs1] == R[rs2]) PC = PC + {imm,1'b0}
bne	Branch if not equal	If(R[rs1] != R[rs2]) PC = PC + {imm,1'b0}
blt	Branch less than	If(R[rs1] < R[rs2]) PC = PC + {imm,1'b0}
bge	Branch greater than or equal	If(R[rs1] >= R[rs2]) PC = PC + {imm,1'b0}

E. U-type

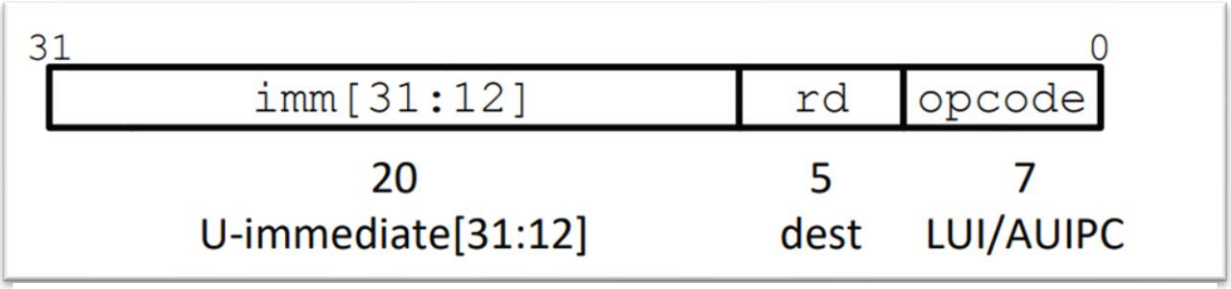

Figure 8: The U-format

- Has 20-bit immediate in upper 20 bits of 32- bit instruction word
- One destination register, rd
- Used for two instructions – LUI – Load Upper Immediate – AUIPC – Add Upper Immediate to PC. We have supported only LUI in the design and the reason for this choice is related to LUI and is explained in the next 3 points.
- LUI writes the upper 20 bits of the destination with the immediate value and clears the lower 12 bits.
- Together with an addi to set low 12 bits, can create any 32-bit value in a register using two instructions (lui/addi).
- HINT : Check the instruction description in the table first before reading the following.
  Using LUI + Addi + JALR instructions combined we can set a certain value in a register in the register file then put it in the PC using this sequence :
  LUI x10 imm[31:12] R[x10] = {imm[31:12], 12'h000}
  addi x10 x10 imm[11:0] R[x10] = R[x10] + imm[11:0]
  **x10 registers now contains a 32bit value
  jalr x9 x10 12'h000 x9 = PC + 4, PC = R[x10] + 12'h000
  **PC contains the required value.
- Generally, the instructions we chose to support are enough to basically carry out most of the operations in the ISA.

Mnemonic	Name	Description
lui	Load upper immediate	R[rd] = {im[31:12, 12'h000}

F. UJ_type

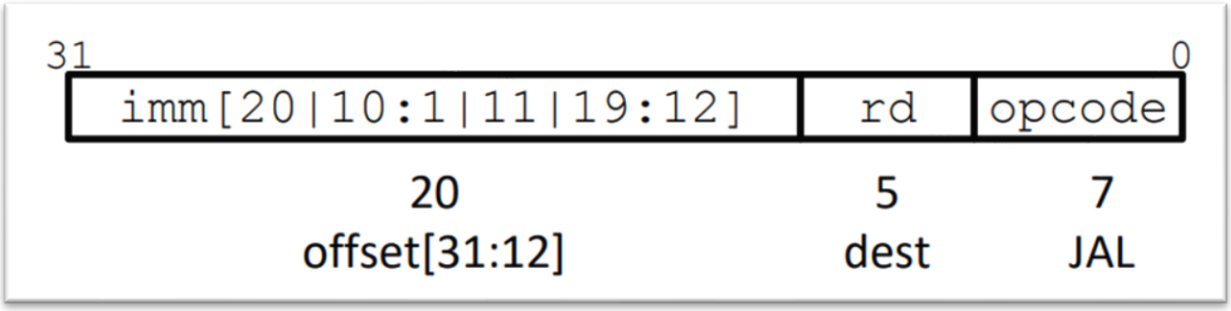

Figure 9: The UJ-format

- For general jumps (jal), we may jump to anywhere in code memory.
- JAL saves PC+4 in register rd (the return address).
- Set PC = PC + offset (PC-relative jump).
- ±2**18 32-bit instructions.
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost.

Mnemonic	Name	Description
jal	Jump and link	R[rd] = PC + 4, PC = {imm, 1'b0}

- Furthermore, it's worth noting that many of the unsupported instructions in the base ISA can still be executed through the instructions we've meticulously implemented. For those rare cases where direct execution is not feasible, subtle adjustments to the datapath can provide the necessary support. Our deliberate choice to create a semi-complicated model serves as a robust foundation for the RISC-V ISA.

Opcodes of supported instructions

Instruction	Opcode
R-type	001_1011
I-type	001_0011
Load	000_0011
S-type	010_0011
JALR	100_0111
SB-type	100_0011
U-type	011_0111
UJ_type	100_1111

Elaborate Design Implementation Using Vivado

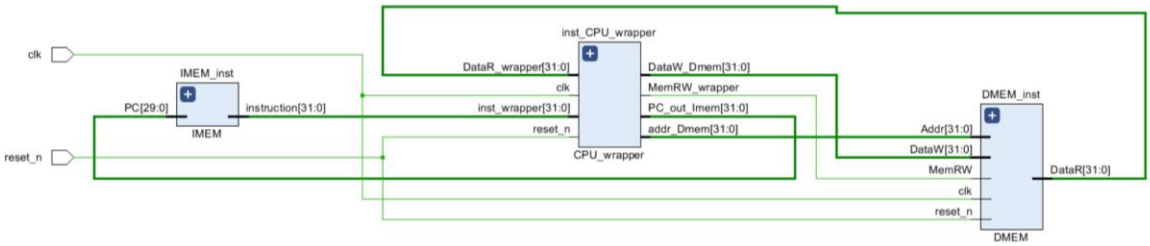

Figure 10: Vivado Elaborate Design

In FIG (9), we find Vivado elaborate design showing the signals connection between RISC-V processor containing datapath and control unit with the external data memory and instruction memory.

The reset signal is used to set the Program Counter (PC), Data Memory (DMEM) content, and the Register File to zero. This ensures a predictable and controlled startup condition for the processor, eliminating any previous values and allowing it to begin execution from a known state.

RISCV 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
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK

Figure 11: RISCV Instruction Set

In fig (11) we find the Instruction Set for RISCV processor. The table clearly shows the opcodes, func3 along with func7 value if required. Each instruction follows the instruction format it belongs to whether it is R-type, I-type, S-type, SB-type, U-type or UJ-type.