RISC-V (Single Cycle)

Name :	**Ehab Mostafa Farghly**
Track :	**DIGITAL IC Design**

I. Overview:

The RISC-V processor architecture is like a set of tools that people can use to design computer processors. These processors can be made to work well for different tasks and can be adjusted to fit specific needs. This architecture is open to anyone, meaning people can look at it, change it, and use it freely. It's designed to be easy to work with and can be customized to make processors that are both simple and powerful.

RISC-V offers various ways to make processors, including:

Single Cycle: In a single-cycle architecture, each instruction takes a single clock cycle to complete. While conceptually simple, it's not highly efficient for modern processors due to potentially long cycle times for complex instructions.

Multicycle: The multicycle architecture divides the instruction execution into multiple stages, such as instruction fetch, decode, execute, memory access, and write-back. Each stage takes a cycle, allowing for more efficient use of resources.

Pipelined: Pipelining breaks down instruction execution into a series of stages, with different instructions at various stages simultaneously. This enables higher throughput and improved performance as long as hazards (data dependencies, control dependencies) are managed effectively.

RISC-V (Single Cycle)

RISC-V offers various types of operations, including:

- R-type: like ADD,SUB,AND,SLT,SLL……
- I-type: like lw,addi……
- B-type:like beq,bnq,…
- S-type:like sw,…….
- J-type:Jalr,Jal,……

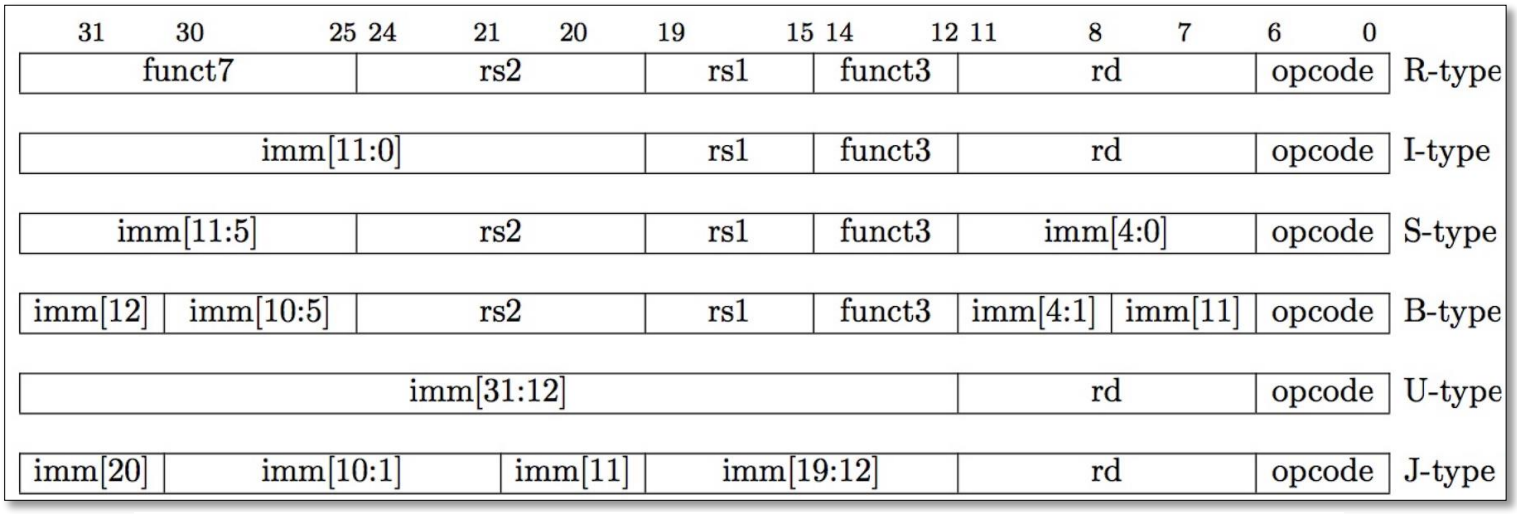


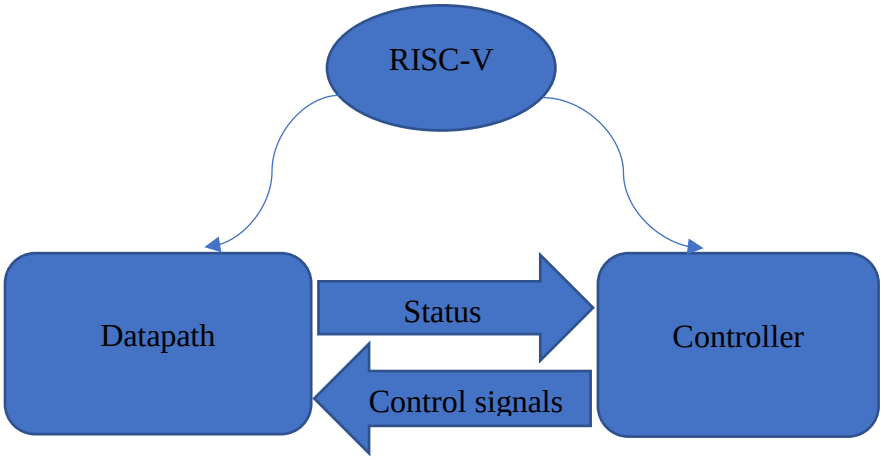
Figure 1 Different Types of operation on RISC-V

II. Architecture:

In this short paper, we will create and explain the RISC-V Single Cycle processor design in simple steps.

Our design consists of 2 basis Block:

- Datapath: The Datapath operates on words of data. It contains structures such as memories, registers, ALUs, and multiplexers. We are implementing the 32-bit RISC-V (RV32I) architecture, so we use a 32-bit Datapath.
- Controller: The control unit receives the current instruction from the Datapath and tells the Datapath how to execute that instruction. Specifically, the control unit produces multiplexer select, register enable, and memory write signals to control the operation of the Datapath.



Datapath Structure:

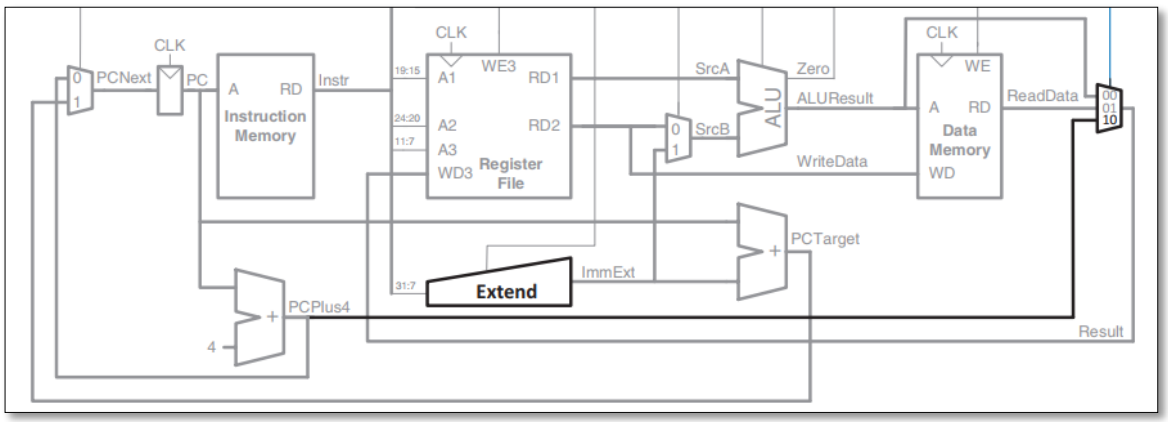


Figure 2 Datapath Structure

RISC-V (Single Cycle)

Datapath Architecture consist of:

I. **PC (Program Counter):** keeps track of the memory address of the current instruction being executed. that indicates where the processor is in the program's sequence of instructions.

II. **Instruction Memory:** that holds all the instructions for a program. Each instruction tells the processor what specific task to perform. The Instruction Memory is responsible for storing these instructions in a specific order.

III. **Register File:** It serves as a rapid-access storage space for temporary data, allowing the processor to swiftly perform calculations and operations without constantly fetching data from slower main memory.

IV. **Data Memory:** Acts as a storage hub for larger and longer-term data storage in a processor's operation. It enables the efficient exchange of data between the processor and memory.

V. **ALU:** Used in executing arithmetic and logic instructions. It takes data from the Register File, performs the required operation, and provides the result to be stored back in registers or used for further computations. processor and memory.

VI. **Sign Extension:** extends a smaller binary number to a larger size, we apply it in immediate and sign extension differ respect to type operation.

VII. **MUXS**: Used in some locations

- Instruction Fetch: Mux determine whether the next instruction comes from the Instruction Memory, based on control signals. This choice is essential for fetching the correct instruction at the right time.
- ALU Operand Selection: Mux decide which data should be sent to the ALU for computation, depending on the operation being performed.
- Write-Back to Register: After an operation is completed, mux determines whether the result should be written back to a register or memory location.

VIII. **Adders**: Used in some locations

- PC (Program Counter): increment PC by 4 to take next instruction.
- PC(Target): determine if there is Jump operation to give which instruction should be executed.

Table 1 EXTENSION Table

ImmSrc	ImmExt	Type	Description
00	{{20{*Instr*[31]}}, *Instr*[31:20]}	I	12-bit signed immediate
01	{{20{*Instr*[31]}}, *Instr*[31:25], *Instr*[11:7]}	S	12-bit signed immediate
10	{{20{*Instr*[31]}}, *Instr*[7], *Instr*[30:25], *Instr*[11:8], 1'b0}	B	13-bit signed immediate
11	{{12{*Instr*[31]}}, *Instr*[19:12], *Instr*[20], *Instr*[30:21], 1'b0}	J	21-bit signed immediate

Control Unit Structure:

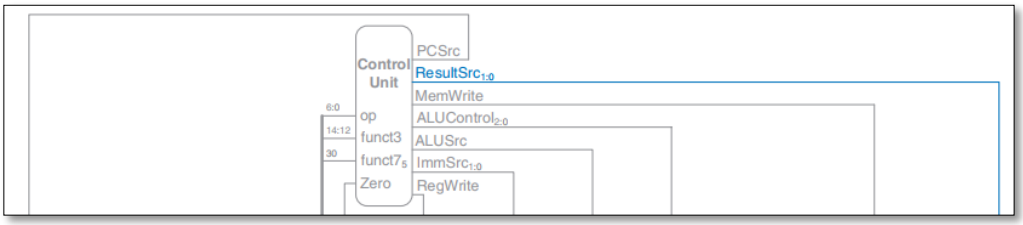


Figure 4 Control Unit Block diagram

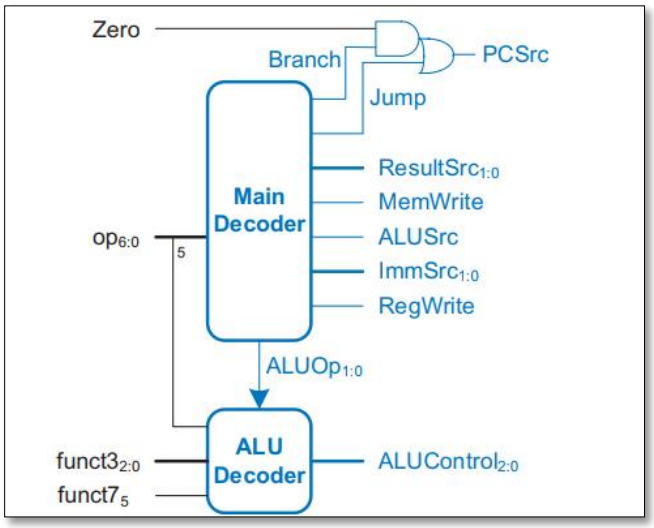


Figure 3 Control Unit Structure

Control Unit role:

I. it takes the instruction from Datapath and put it in some of decoders to determine the control signals. I will explain more and show the table of each decoder.

II. Control unit consist of two main Decoders

- ALU Decoder: it determines ALUControl which specify which operation will run in ALU Block, Table 2 show the decoder operations.
- Main Decoder: it determines rest of control signals which determine rest of operations, Table 3 show the main decoder operations.

Table 2 ALU Decoder

ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000 (add)	`lw, sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt`
	110	x	011 (or)	`or`
	111	x	010 (and)	`and`

Table 3 Main Decoder

Instruction	Opcode	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp	Jump
`lw`	0000011	1	00	1	0	**01**	0	00	**0**
`sw`	0100011	0	01	1	1	**xx**	0	00	**0**
R-type	0110011	1	xx	0	0	**00**	0	10	**0**
`beq`	1100011	0	10	0	0	**xx**	1	01	**0**
I-type ALU	0010011	1	00	1	0	**00**	0	10	**0**
`jal`	**1101111**	**1**	**11**	**x**	**0**	**10**	**0**	**xx**	**1**

III. Simulation:

- Assembly code.

```
#       RISC-V Assembly         Description                 Address   Machine Code
main:   addi x2, x0, 5          # x2 = 5                    0         00500113
        addi x3, x0, 12         # x3 = 12                   4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4       10        0041F2B3
        add  x5, x5, x4         # x5 = 4 + 7 = 11           14        004282B3
        beq  x5, x7, end        # shouldn't be taken        18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0         1C        0041A233
        beq  x4, x0, around     # should be taken           20        00020463
        addi x5, x0, 0          # shouldn't execute         24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5) = 1          28        0023A233
        add  x7, x4, x5         # x7 = (1 + 11) = 12        2C        005203B3
        sub  x7, x7, x2         # x7 = (12 - 5) = 7         30        402383B3
        sw   x7, 84(x3)         # [96] = 7                  34        0471AA23
        lw   x2, 96(x0)         # x2 = [96] = 7             38        06002103
        add  x9, x2, x5         # x9 = (7 + 11) = 18        3C        005104B3
        jal  x3, end            # jump to end, x3 = 0x44    40        008001EF
        addi x2, x0, 1          # shouldn't execute         44        00100113
end:    add  x2, x2, x9         # x2 = (7 + 18) = 25        48        00910133
        sw   x2, 0x20(x3)       # [100] = 25                4C        0221A023
done:   beq  x2, x2, done       # infinite loop             50        00210063
```

Figure 5 assembly code and its machine code (Taken from pdf of Harris for RISC-v)

- Results:

  -

```
main:   addi x2, x0, 5          # x2 = 5                    0         00500113
        addi x3, x0, 12         # x3 = 12                   4         00C00193
        addi x7, x3, -9         # x7 = (12 - 9) = 3         8         FF718393
        or   x4, x7, x2         # x4 = (3 OR 5) = 7         C         0023E233
        and  x5, x3, x4         # x5 = (12 AND 7) = 4       10        0041F2B3
```

00000000	00000000000000000000000000000000
00000001	00000000000000000000000000000000
00000002	00000000000000000000000000000101
00000003	00000000000000000000000000001100
00000004	00000000000000000000000000000111
00000005	00000000000000000000000000000100
00000006	00000000000000000000000000000000
00000007	00000000000000000000000000000011

  -

```
        beq  x5, x7, end        # shouldn't be taken        18        02728863
        slt  x4, x3, x4         # x4 = (12 < 7) = 0         1C        0041A233
        beq  x4, x0, around     # should be taken           20        00020463
        addi x5, x0, 0          # shouldn't execute         24        00000293
around: slt  x4, x7, x2         # x4 = (3 < 5) = 1          28        0023A233
        add  x7, x4, x5         # x7 = (1 + 11) = 12        2C        005203B3
```

- At the end of all instructions