64bit RISC-V pipeline architecture

Answers 1
What is the latency of each module in your design? (e.g. ALU, register file)
In pipeline latency is time between when instruction is issued and instructions is completed. And latency of each module is 2ns because each stage takes like 2 clock cycles.

Answer 2
Which path is the critical path of your cpu? And how can you decrease the latency of it? (2%)
Critical path in datapath is execution of LW instruction. Because this instruction uses all the modules in datapath. And to reduce the latency, optimizations in the datapath are carried out.

Answer 3
How to solve data hazard? (2%)
Data hazard happens when an instruction depends on the results of previous instruction and that result has not been ready yet. There are three ways of handling data hazard.
Forwarding
it is about adding some hardware, which sends result of ALU to the input of ALU so that for next instruction results will be ready
Reordering instruction
Data hazard can also be resolved if order of instructions is changed.
Adding stalls
Stalls are bubble instructions can be added in the pipeline so that when instruction executes, results of previous instructions are ready. But this method reduces the efficiency.

Answer 4
How to solve control hazard? (2%)
To handle control hazard, one way is to do **branch prediction**, if the prediction is correct, great. If prediction is wrong, efficiency is affected by the **branch penalty**.
The second method is to used **stalls** whenever branch instruction enters the pipeline. This also effects performance.

Answer 5
Describe 3 different workloads attributes, and which one can be improved tremendously by branch predictor? (2%)
Branch predictor is hardware circuit that predicts which way a branch instruction goes, before execution of that instruction. It plays a critical role in performance of pipeline architecture in terms of dealing with control hazards.

Answer 6
Is it always beneficial to insert multiple stage of pipeline in designs? How does it affect the latency? (2%)
Speed up in pipeline architecture depends on the number of stages of pipeline architecture has. More the stages, performance would be better. But it affects latency, because when instruction enters and leaves the pipeline after execution would be huge. So performance in terms of throughput is great but in terms of latency it is bad having more stages.

Detailed pipeline design can be referred below:

RISC-V Data path & Control Lines

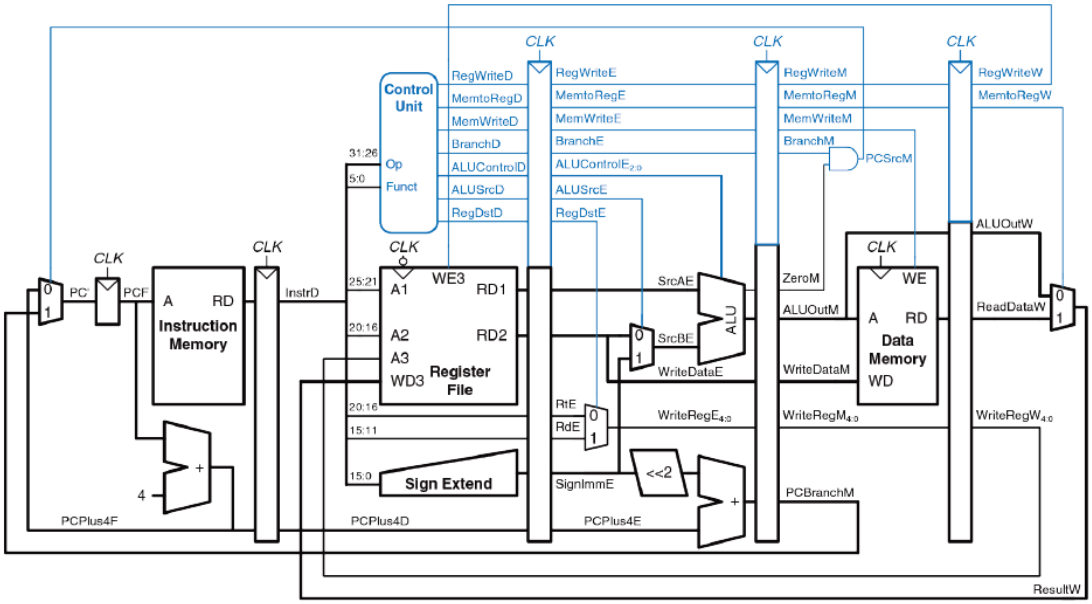


Figure 3.5: Complete Data path

1 Introduction & Background of RISC-V

1.1 Origin of RISC-V

In 2010 professor Krste Asanovic team started a three month project at UC

Berkeley was looking for the right ISA to use in their next project

I X86 is too complicated to implement in a processor for a research chip

II ARM (in 2010) did not have 64 bit support

1.2 RISC-V (Reduced Instruction Set Architecture)

I A high quality license-free royalty-free RISC ISA
II Standard maintained by the non-profit RISC-V foundation
III Suitable for all type of computing system
IV RISC-V is not a company or not a specific CPU implementation.

1.3 Foundation of RISC-V

RISC-V foundation governs the RISC-V architecture and a non-profit

organization which was formed in 2015

1.3.1 Foundation Functions

Foundation has three primary functions which are:

- To direct the future development of ISA
- It is also to develop compliance tests and Suites for the architecture of RISC-V
- Promotion of architecture at industry events and marketing.

1.4RISC-V base Integer ISA

i RV32I : 32-Bits

- Less than 5 instructions needed

ii RV32E : 32-Bits Embedded

- Reduced register count from 32 to16 for tiny micro controller

iii RV64I : 64-Bits

- It is useful for Linux

iv RV128I : 128-Bits

- **Further-proof address space limitation**

Instruction Set Architecture of RISC-V

2.1 Instruction Formats

RISC-V is an open source, frozen and simple ISA. It provides open source license that don't require pay fee to use and equivalent to everyone having micro-architecture license. It provides the possibility of designing custom silicon i.e. Application Specified Integrated Circuits ASICS. Initially it was designed for educational research but now getting attention towards industry adoption because of its benefits

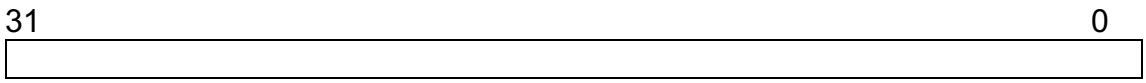
2.2 Stored-Program Concept

This theory was evolved by Jon Van Neumann. The stored program concept is the scheme that data and instructions in the form of binary are stored simultaneously in memory.

The program or data is fetched from memory one by one and then execute.

By convention, in RISC-V instructions one word is composed to 4 bytes which is equal to 32 bits.

Table 2.1: 32bit wide instructions



There are 6 types of instruction format which are:

- R type

In R-type instruction has 3 register, arithmetic/logical operation
Like add, sub, AND, OR

- I type

In I-format instruction with immediate, loads like addi, subi or lw

- S type

S-format has store instruction like sw, sb

- SB type

SB-format has branch instruction like beq, bne

- U type

U-format has instruction with upper immediate of 20-bits like LUI, AUIPC

- UJ type

UJ-format has jump instruction like JAL, JALR

2.3 R-Format Instructions

Table: 2.2 R FORMAT

R-Type	Funct7	rs2	rs1	Funct3	rd	opcode
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

In R-Type format,

I Opcode is 7 bits start from 0 to 6.it identify the operation

II Rd is the destination register result is stored in this register

III Funct3 is additional opcode bits which is 3 bits.It tells what operation is performed

IV Rs1 is source register 1 which is 5 bits.

V Rs2 is source register 2 which is also 5 bits

VI Funct7 contain 7 bits which is also additional opcode bits/immediate

Example of R-Format Encoding
Instruction: add, x6, x10, x6
Representation in decimal:

Table 2.3 Decimal representation

0	6	10	0	6	51

Representation in binary:

Table 2.4: Binary representation

0000000	00110	01010	000	00110	0110011

$(0000\ 0000\ 0110\ 0101\ 0000\ 0011\ 0011\ 0011)_2 = (\ 00650333\ )_{16}$
Some of the instructions are used in R-Type format which are:
ADD-instruction:

Table 2.5: ADD instruction

0000000	rs2	rs1	000	rd	0110011

SUB-instruction:

Table 2.6 SUB instruction

0100000	rs2	rs1	000	rd	0110011

SLL-instruction:

Table 2.7: SLL instruction

0000000	rs2	rs1	001	rd	0110011

SLT-instruction:

Table 2.8: SLT instruction

0000000	rs2	rs1	010	rd	0110011

SLTU-instruction:

Table 2.9: SLTU instruction

0000000	rs2	rs1	011	rd	0110011

OR-instruction:

Table 2.10 OR instruction

0000000	rs2	rs1	110	rd	0110011

AND-instruction:

Table 2.11 AND instruction

0000000	rs2	rs1	111	rd	0110011

XOR-instruction:

Table 2.12 XOR instruction

0000000	rs2	rs1	100	rd	0110011

In all these instructions funct7 + funct3 choose different operations

2.4 I-Format Instructions

Table 2.13: I FORMAT

I-type	immediate	rs1	Funct3	rd	opcode

12 bits 5 bits 3 bits 5 bits 7 bits

I In I-Type format we use 2 registers first is source register and second one is destination register.
II In I-Type format opcode is 7 bits which specially identify the instruction
III rs1 is source register or also called base address register which contain 5 bits to identify the register operand
IV Rd is the destination register result is stored in this register.
V Funct3 contain 3 bits
VI Immediate contain 12 bits it is constant oprand and also called offset base address.
VII Sign extension is used in I-type format because we have to convert 12 bits into 32 bits

Example of I-Format Encoding

Instruction: Addi x14, x5 ,-27

rd = x14;

r1 = x5;

Representation in binary:

Table 2.14: Binary representation

111111100100	00101	0	01110	0010011

Some of the instructions are used in I-Type format which are:

ADDI-instruction:

Table 2.15: ADDI instruction

Imm[11:0]	rs1	000	rd	0010011

SLTI-instruction:

Table 2.16: SLTI instruction

Imm[11:0]	rs1	010	rd	0010011

ORI-instruction:

Table 2.17: ORI instruction

Imm[11:0]	rs1	110	rd	0010011

ANDI-instruction:

Table 2.18: ANDI instruction

Imm[11:0]	rs1	111	rd	0010011

SLLI-instruction:

Table 2.19: SLLI instruction

0000000	shamt	rs1	001	rd	0010011

SRLI-instruction:

Table 2.20: SRLI instruction

0000000	shamt	rs1	101	rd	0010011

In SLLI instruction or SRLI instruction shamt is the immediate shift in this case instruction use low 5-bits value of immediate for shift

2.4.1 Load Instructions in I-type:

Table 2.21 Load instruction

31 0

Imm[11:0]	rs1	Funct3	rd	opcode
Offset[11:0]	Base reg	Funct3	Destination reg	LOAD

I Signed-immediate of 12-bits are add up to base-address in source register or base-register to form the memory-address

II And then value is taken from the memory to store in destination register.

Example of I-Format LOAD Encoding
Instruction: lw x13, 9(x3)
Representation in decimal:

Table 2.22: Decimal representation

9	3	2	13	3

Representation in binary:

Table 2.23: Binary representation

000000001001	00011	010	01101	000011

$(0000\ 0000\ 1001\ 0001\ 1010\ 0110\ 1000\ 0011)_2$
Some of the instructions are used in I-Type format for LOAD:
LB-instruction:

Table 2.24 LB instruction

Imm[11:0]	rs1	000	rd	0000011

LH-instruction:

Table 2.1: LH instruction

Imm[11:0]	rs1	001	rd	0000011

LW-instruction:

Table 2.2: LW instruction

Imm[11:0]	rs1	010	rd	0000011

LBU-instruction:

Table 2.3: LBU instruction

Imm[11:0]	rs1	100	rd	0000011

LHU-instruction:

Table 2.28: LHU instruction

Imm[11:0]	rs1	101	rd	0000011

I LBU stands for load-unsigned-bytes
II LH stands for Load-halfword that can load 2 bytes which is equal to 16-bits and then sign extension is used to fill rd register 32-bits.
III LW stands for load-word
IV LHU stands for load-unsigned-halfword that has 0 extended 16-bits which is used to fill rd register 32-bits

2.5 S-Format Instructions

Table 2.4: S FORMAT

Imm[11:5]	rs2	rs1	Funct3	Imm[4:0]	opcode

I Store is used to read registers
II rs1 register is used for base-address register
III rs2 register is used for source-operand register
IV Immediate is used as offset is added up to base-address
V There is no destination register
VI The design of RISC-V is to move immediate of lower 5-bits to destination register by holding the rs1 and rs2 in same position

Example of S-Format Encoding

Instruction: sw x13 10 (x3)

Table 2.5: S format encoding

31 0

Imm[11:5]	rs2	rs1	Funct3	Imm[4:0]	opcode

Representation in decimal:

Table 2.6: Decimal representation

0	13	3	2	10	35

Representation in binary:

Table 2.7: Binary representation

0000000	01101	00011	010	01010	0100011

Some of the instructions are used in S-Type format which are:

SB-instruction:

Table 2.8: SB instruction

Imm[11:5]	rs2	rs1	000	Imm[4:0]	0100011

SH-instruction:

Table 2.9: SH instruction

Imm[11:5]	rs2	rs1	001	Imm[4:0]	0100011

SW-instruction:

Table 2.10: SW instruction

Imm[11:5]	rs2	rs1	010	Imm[4:0]	0100011

2.6 SB-Format Instructions

I This format is used by branch instruction,Branch is an instruction that changes the program execution based on some condition.Branch instruction is an instruction that compare and check a condition and it has to decide whether the next PC would be PC+4 or it should be some other address

II If the branch is not taken after the branch instruction we have to execute the next instruction that is if the branch instruction is PC next instruction is PC+4 and this instruction is we have to execute

III PC = PC + 4 = Next-instruction

IV if branch is taken we have to move to other location new value of PC will be something else and it is called branch target or immediate of 12-bits

V PC = PC + (imm * 4)

VI However in RISC-V branch target is calculated by :

VII PC = PC+ (imm * 2)

VIII Actually we are doing PC = PC + (imm * 4) but multiply by 2 is done only to support 2 byte instruction in our RV32I there are only 4-bytes instruction but in RV32E this particular ISA using 2-bytes or 16-bits

IX Branch instruction should contain opcode it should know that what 2 operands i have to compare ,It should have branch target these are the 3 parts that are stored in branch instruction

X There is no destination register

XI For branch target specific we are taking the immediate and filling it from [12:1] and the 0^{th} bit remain vacant then we fill it with zero and remaining bits are sign-extended.there is left shift by 1-bit

XII Format of SB-instruction is :

Table 2.11: SB-FORMAT encoding

Imm[12]	Imm[10:5]	rs 2	rs1	func3	Imm[4:1]	Imm[11]	opcode

There are six types of branch instruction which are:

BEQ-instruction:

Table 2.12: BEQ instruction

Imm[12|10:5]	rs2	rs1	000	Imm[4:1|11]	1100011

BNE-instruction:

Table 2.13: BNE instruction

Imm[12|10:5]	rs2	rs1	001	Imm[4:1|11]	1100011

BLT-instruction:

Table 2.14: BLT instruction

Imm[12|10:5]	rs2	rs1	100	Imm[4:1|11]	1100011

BGE-instruction:

Table 2.15: BGE instruction

Imm[12|10:5]	rs2	rs1	101	Imm[4:1|11]	1100011

BLTU-instruction:

Table 2.16: BLTU instruction

Imm[12|10:5]	rs2	rs1	110	Imm[4:1|11]	1100011

BGEU-instruction:

Table 2.17: BGEU instruction

Imm[12|10:5]	rs2	rs1	111	Imm[4:1|11]	1100011

2.7 U-Format Instructions

Uformat is upper-immediate instruction

Table 2.18: U FORMAT

31 0

Imm[31:12]	rd	opcode

I U-format contain 7-bits opcode
II There is only one register which is destination register of 5-bits
III There is immediate of 20-bits that is upper 20-bits of 30-bits instruction
IV In Upper-immediate format 20-bits are filled are extended in such a way to 32-bits the size that the most significant 20-bits will be taken from the instruction and the least significant 12-bits will be embedded with zeros
V In U-format there are 2 types of instruction first is LUI and second is AUIPC.
VI U-format is used for instruction which is load-upper-immediate
VII And also used for Add-Upper-Immediate to program counter(PC)

2.7.1 Load-Upper-Immediate

This instruction loads the immediate value into a register

Example

Instruction: LUI x3, 0xABCDE

Table 2.19: LUI instruction

Imm[31:12]	rd	opcode
0xABCDE	X3	LUI

The size of register is 32-bits however from these 32-bits we are loading 20-bits and these 20-bits are basically immediate and this will be extended to 32-bits and this is extended in such a way most significant 20-bits will be ABCDE and the least significant 12-bits are 000 so these 32-bits will be loaded in x3

Table 2.20: LUI instruction encoding

x3

A	B	C	D	E	0	0	0

2.7.2 Add-Upper-Immediate-to-PC

- It adds an immediate value that is extended to 32-bits to program counter.we can write as:

Instruction:

AUIPC x0, 0xABCD1

I X0 is basically we want a register here the register has no value in AUIPC. Register has no functionality, whatever operation we will do it will always be zero, so it is of no use in particular instruction.

II Instruction take this value extend it to 32-bits it will extended as ABCD1000 and it will add this value and we can say the offset to PC.so our new PC will be:

III PC = ABCD1000 + PC

IV The use of this instruction is sometimes we want that our PC. We want to jump or go to some location which belong to some other function. Basically our PC get updated as

V PC = PC+4

VI Because our instruction is 4 bytes which is 32-bits that's why we are doing plus 4 every time. However this is sequential execution that means instruction are executed one after the other

VII However in some scenario we want to jump from PC to some other location which is far off in the program and in this specific case we have to update the PC with a new value and new value will be added some offset to this PC and then our program will jump to that particular location.

VIII So Add-Upper-immediate will be required when we want to jump to a far off location in the program.

2.8 UJ-Format Instructions

I UJ-format is similar to upper-immediate format

II One instruction of UJ-format is Jal for general jumps like function calls we don't use branch instruction instead we may be jumping to some far off location using the function calls. For function calls we are using Jal instruction

III In UJ-format opcode is of 7-bits and destination register of 5-bits and then we have an immediate

IV However in store format the immediate was 12:31 bits so if we have 32-bits immediate so the most significant bits from 12:31 were stored in instruction

V In UJ-format the opcode of Jal is 1101111

VI In our UJ-format instruction we are not storing the most significant basically if our immediate is 32-bits so the first 20-bits from 0:19 come from the instruction itself.however instead of having from 0:19 we are keeping it 1:20 and the least significant bit is filled with zero.it is the same as branch instruction because for jump instruction also we have to multiply it by 2 to find out the branch target at the address so we have to shift left by 1-bitand

the least significant bit is filled with zero and from 1:20 bits comes in the 20-bit instruction ad 20-bits are like this:

Table 2.21: UJ instruction encoding

Imm[20|10 : 1|11|19 : 12]	rd	opcode

2.8.1 Functionality of Jal:

- Functionality of Jal is : it changes PC as PC + offset and the offset is the immediate bits [20:1] and 0^{th} bit will be zero and the remaining bits will be sign extended.zero is taken because it is multiplied by 2 basically is is multiplied by 4 because every instruction has 4 bytes however we did multiply by 2 in order to support 2-byte instruction as well, so Jal instruction form the PC as PC+offset and x5 will be PC+4 and PC will be PC+offset

Jal x5, imm[20:1]
PC = PC + offset
Jal saves PC+4 in destination register
x5 = PC+4
PC = PC+offset

- If we write the instruction like :

Jal x0, imm[20:1]
It store in x0, PC +4 but we know that whatever operation we do x0 will always have the value 0, so it is not storing anything it will simply discard the return address and the new PC will be PC + offset. This is required in the case the return address is not required that means we are making the function call, we are not returning from the function call
For this kind of instruction in which we simply neglect the return address the pseudo instruction for the same J and then the offset this pseudo instruction is internally converted into Jal

2.8.2 Jalr-Instruction:

I This Jalr instruction belong to I-format
II Opcode for this instruction is 1100111.immediate will be 12-bits from [11:0]there is no multiplication of 2,that means no shifting of bits
III There is a destination register and a source register also

Table 2.22: Jalr instruction

31				0
Imm[11:0]	rs1	Funct3	rd	opcode
Offset	base	0	dest	Jalr

Jalr rd, rs1, imm
rd = PC + 4
PC = [rs1] + imm

I It writes PC+4 in destination register and PC+4 is the return address
II And it sets PC equal to the contents of rs1+imm and the immediate is the sign extended to 32-bits so it works similarly as Jal.but there is no logical 1-bit shift
III Jalr is mostly used to implement the return instruction and return instruction basically takes the return address and store that return address in PC.

3 RISC-V Architecture

3.1 RISC-V

RISC-V is an open source Instruction Set Architecture, based on Reduced Instruction Set Computer. It provides open source license that don't require pay fee to use and equivalent to everyone having micro-architecture license. There is a lot of implementation details are online available without any commercial string attached. The authors present two reasons of designing this new Instruction Set Architecture. First, all the prevalent Instruction set are proprietary we can't use them without license. Secondly, it has Frozen ISA and also base instruction frozen.

3.2 Components of Computer

- **CPU**
  CPU stands for Central Processing Unit, it is active part of the computer what does all the operations/actions including Decision making and data manipulating.
  It contains:
  - Control
  - Arithmetic Logic Unit (Data path)
  - Registers (Data path)
- **Memory**
- **Peripheral Devices**
  - Input & Output Devices
- **System Bus**
  - Data bus
  - Control bus
  - Address bus

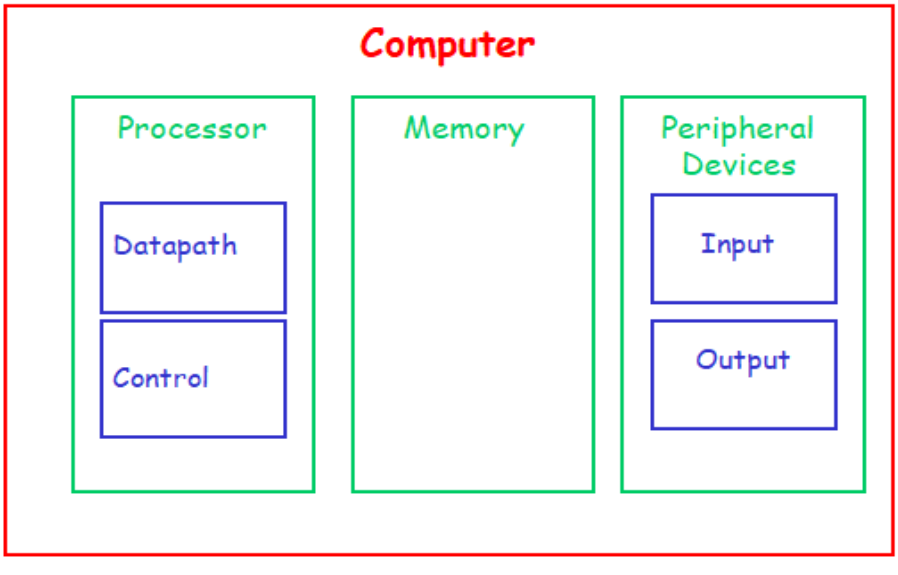


Figure 3.1: Computer components

3.3 Data path & Control

- Data path is the portion of the processor contains hardware to perform all the require operations .Its aim to design Data-path is to support data transfer that require by the instructions.
- Control is also a portion of processor (also hardware) that tells the Data-path what to do or what needs to be done.

3.4 Pipeline Stages

i. Stage 01: Fetch

ii. Stage 02: Decode

iii. Stage 03: Execute

iv. Stage 04: Memory

v. Stage 05: Write Back

3.4.1 Stage 01: Instruction Fetch

- **Functionality of Instruction memory**
    - The 32 bit instruction must firstly be fetch from the instruction memory.
    - It reads Program counter and send it to instruction memory, and fetch instruction from instruction memory.
    - Increment in program counter.

      PC=PC+4, to point to the next instruction.
- **Instruction Fetch-Elements**

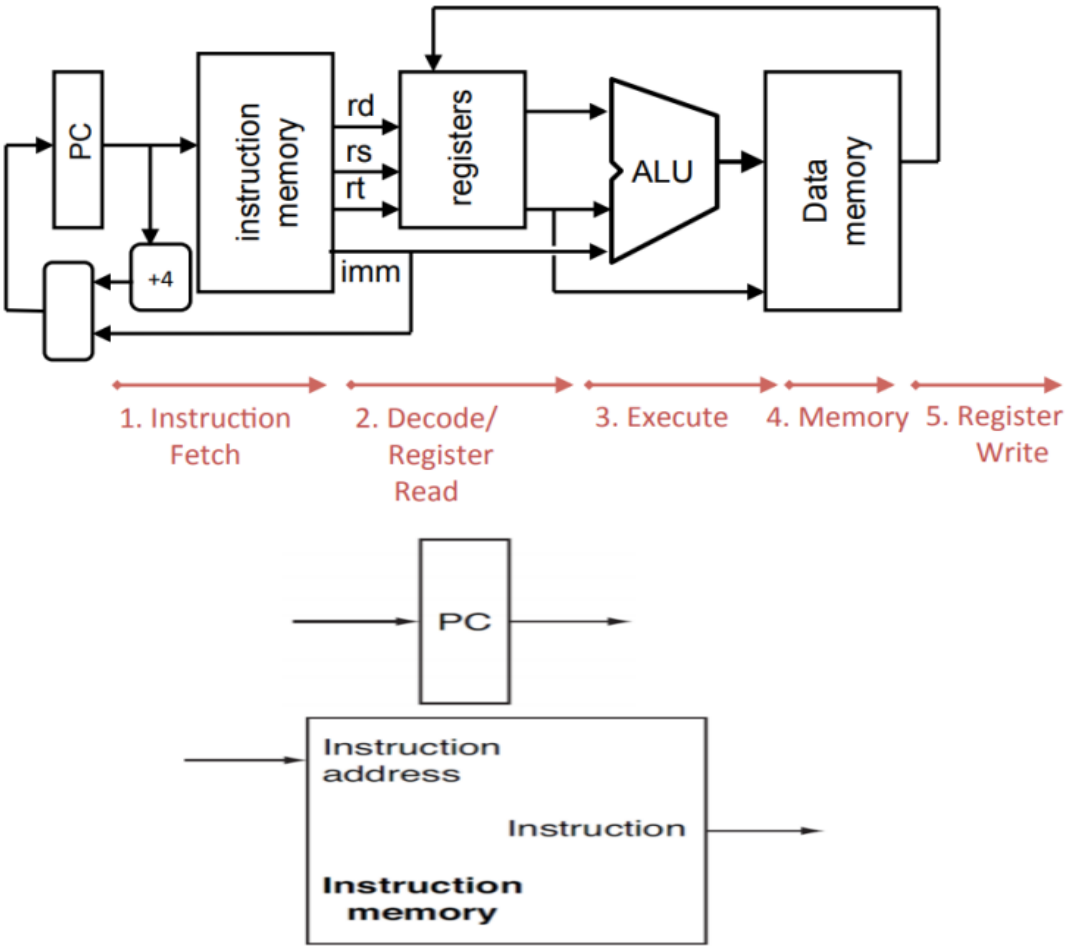


Figure 3.3: Instruction fetch (a)

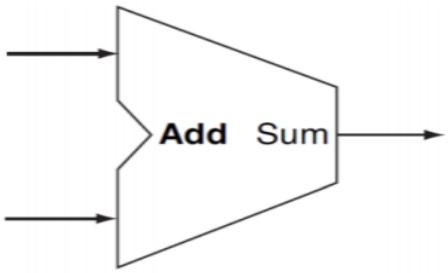


Figure 3.4: ALU

- **Instruction Fetch**

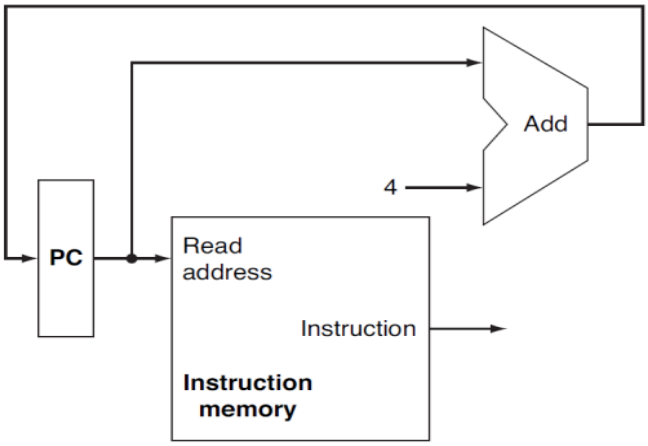


Figure 3.5: Instruction fetch (b)

3.4.2 Stage 02: Instruction Decode

- **Functionality of Register File**
  - It decodes all the necessary data, gather data from the fields.
  - Read Data from the registers.
  - It read 2 registers for add.
  - It read only one registers for addi.
  - For jal, no need necessary.

- **Register File**

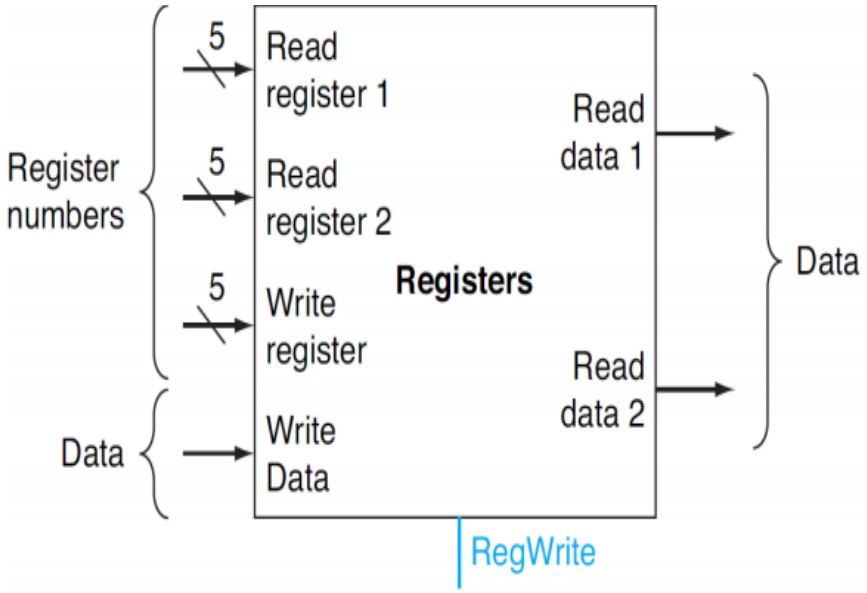


Figure 3.6: Register file

Input ports: Register Address, register write, control signal.
Output ports: Register Data

3.4.3 Stage 03: ALU (Arithmetic-Logic Unit)

- **Functionality of ALU**
  - The real work of most instructions is done in ALU, Arithmetic logic operations like
    - Addition, subtraction
    - Logical operation, shifting
    - Comparison (set less than)
  - Loads and stores in memory
    The address which is needed to be access from memory is the value in $t1 PLUS the value 120, so this will be done in this stage.
  - Conditional branch
    Comparison is done in this stage and provides only one solution.

- **ALU Operation-Elements**

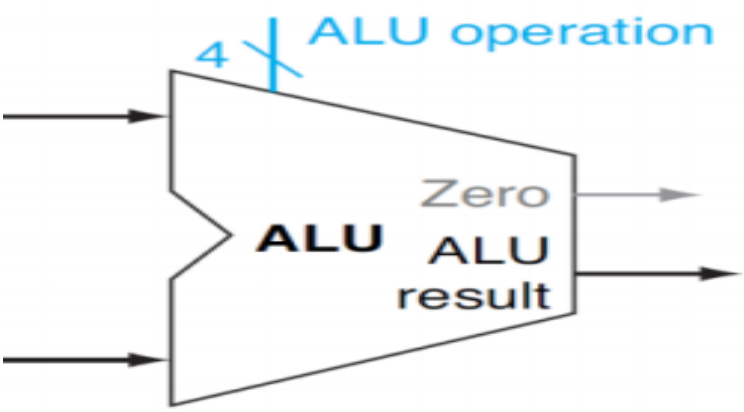


Figure 3. 1: ALU PINOUT

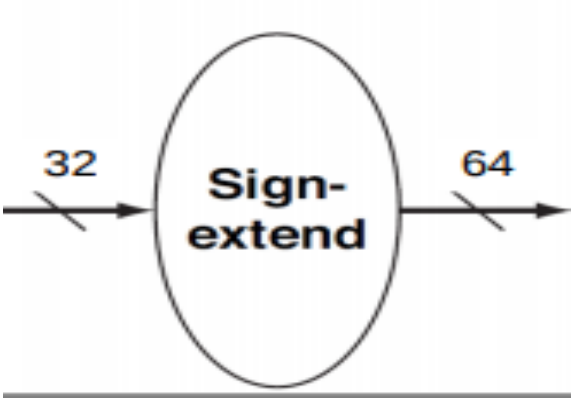


Figure 3.2: Sign extension

For addi, design will change and sign-extension will be used.
Input ports: Register Data, 2nd Register data/immediate value, control signal.
Output ports: ALU, zero

- **ALU Operation-Data path**

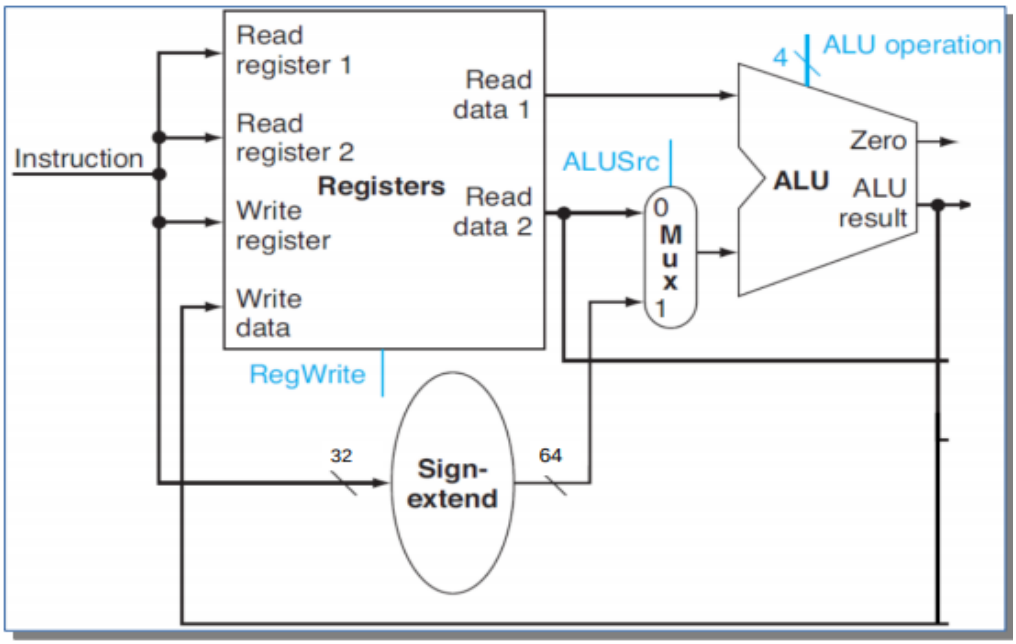


Figure 3.3: Execution stage

3.4.4 Stage 04: Memory Access

- **Functionality of Data Memory**
  - Only for load word and store word, other remains idle during this stage.
  - **Load instructions** are used to access data from memory to registers. Store instructions are used to send data from registers to memory.
- **Data Memory**

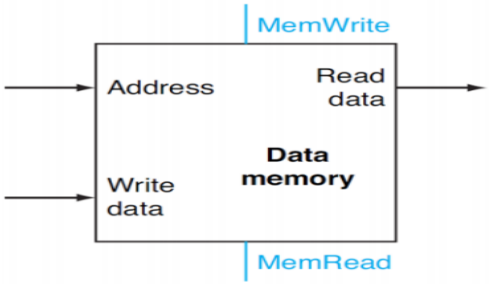


Figure 3.4: Data memory

Input ports: Register Address, write data, control signals.

Output ports: Register Data

3.4.5 Stage 05: Register Write

- Results of computation instructions are written on registers. Examples are: Arithmetic instructions, logical instructions.
- No need to write data in register at the end, for stores, branches, jumps. They remain idle during this fifth stage.

RISC-V Instruction Format

Table 3.1: Formats

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode