Computer Organization & Design (RISC-V)

– Hitesh Pranav

HOW TO DO COD?

Before jumping into the unit,
A quick disclaimer ⚠️

I have not seen a subject so interlinked.
Even if you start from the end and go to the first,
there is some interlinking to each topic

This is my attempt to cover all the topics.
Don't neglect the codes & implementation diagrams.

I hope you get the deeper level understanding for this subject

ALL THE BEST! ❤️

- Hitesh Pranav

U1 Language of the Computer

What is a processor ?

→ The brain of a computer
which fetches instructions from memory, decodes & executes them
it can also store the result, perform arithmetic, logic & control operations

What is microprocessor ?

→ A processor implemented on a single chip which contains CPU functions (ALU, registers, control)
ex: AMD Ryzen, Intel i7, RISC-V etc.,

What is microcontroller?

→ It is a complete mini computer on a chip which contains CPU, Memory, I/O Ports, Timers, ADC/DAC
ex: Arduino, ESP32, Raspberry pi etc.,

Von-Neumann Architecture ?

→ Single memory for both instructions & data
→ Single bus means CPU can do only one thing at a time
→ Simple design & cheap
→ Slower

Harvard Architecture ?

→ One memory for instructions & other for data
→ Has seperate buses for fetching instructions & read/write data
→ Faster (No bottleneck)
→ Complex & Costlier

Instruction Set Architecture (ISA)

→ Instructions are the words of a computer language & Instruction set is the vocabulary
→ Computers respond only to these instructions (small binary encoded commands). Not Python, C, Java
→ 2 Design philosophies for ISA : CISC & RISC

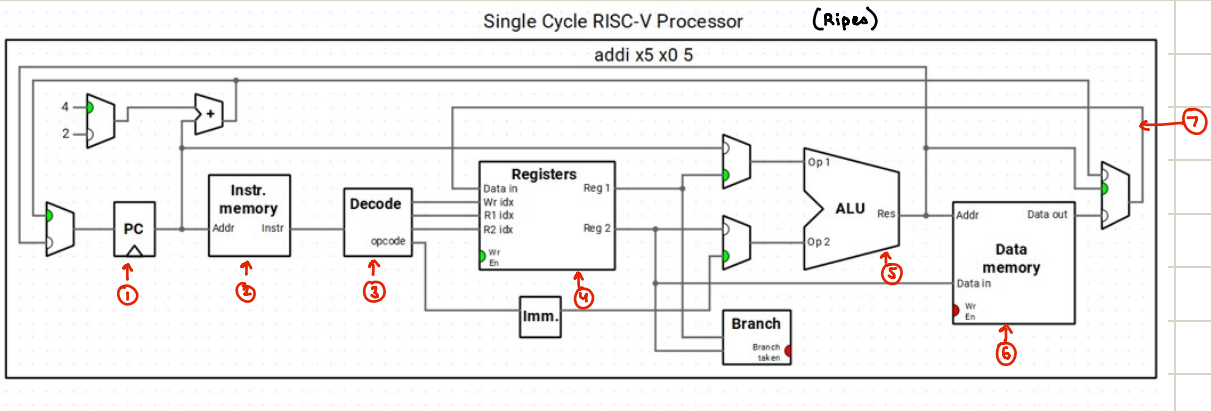
CISC (Complex Instruction Set Computer)

→ Consists of fewer instructions per program but each instruction is complex
→ Complex addressing modes
→ Instructions take multiple cycles to execute

RISC (Reduced Instruction Set Computer)

→ Makes instruction set simple & fast but many lines
→ Provides freedom to change it & add own things (open source)

→ In order to understand the topics, lets understand the RISC-V processor



PROCESSOR	Memory (Instructions)	Time Slot
	Instruction Fetch	1
	Instruction Decoding	2
	Execution	3
	Memory Access	4
	Write Back	5

① , ② → Fetch instructions for executions

③ , ④ → Decode the instructions & access registers

⑤ → ALU executes the function desired

⑥ → Accesses memory if required

⑦ → If necessary to write back, it will

Operations of Computer

→ Usually the instructions must be able to perform fundamental arithmetic operations

ex : `add a, b, c` ⇒ a = b+c

→ Similarly we have multiple operations

→ Note that we only use 3 variables. This for all operations!

So if you wanted to perform a = (b+c) - (d+e)

```
add f, b, c
add g, d, e
sub a, f, g
```

The reason for this is

* Simplicity favours regularity

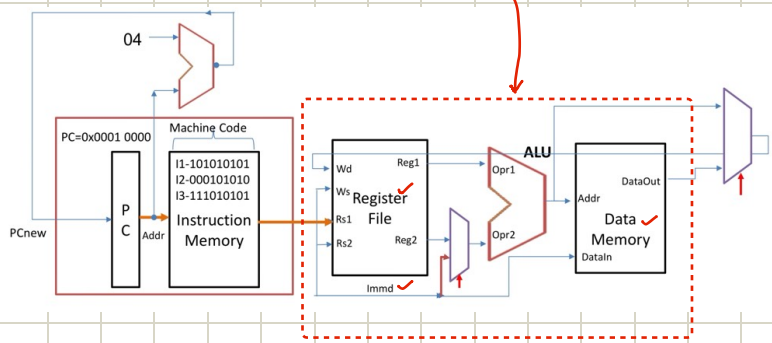
Category	Instruction	Example	Meaning	Comments
Arithmetic	Add	`add x5, x6, x7`	`x5 = x6 + x7`	Three register operands; add
	Subtract	`sub x5, x6, x7`	`x5 = x6 - x7`	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	`x5 = x6 + 20`	Used to add constants
Data transfer	Load word	`lw x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Word from memory to register
	Load word, unsigned	`lwu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	`Memory[x6 + 40] = x5`	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Halfword from memory to register
	Load halfword, unsigned	`lhu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	`Memory[x6 + 40] = x5`	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte from memory to register
	Load byte, unsigned	`lbu x5, 40(x6)`	`x5 = Memory[x6 + 40]`	Byte unsigned from memory to register
	Store byte	`sb x5, 40(x6)`	`Memory[x6 + 40] = x5`	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	`x5 = Memory[x6]`	Load; 1st half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	`Memory[x6] = x5; x7 = 0/1`	Store; 2nd half of atomic swap
	Load upper immediate	`lui x5, 0x12345`	`x5 = 0x12345000`	Loads 20-bit constant shifted left 12 bits
Logical	And	`and x5, x6, x7`	`x5 = x6 & x7`	Three reg. operands; bit-by-bit AND
	Inclusive or	`or x5, x6, x8`	`x5 = x6 | x8`	Three reg. operands; bit-by-bit OR
	Exclusive or	`xor x5, x6, x9`	`x5 = x6 ^ x9`	Three reg. operands; bit-by-bit XOR
	And immediate	`andi x5, x6, 20`	`x5 = x6 & 20`	Bit-by-bit AND reg. with constant
	Inclusive or immediate	`ori x5, x6, 20`	`x5 = x6 | 20`	Bit-by-bit OR reg. with constant
	Exclusive or immediate	`xori x5, x6, 20`	`x5 = x6 ^ 20`	Bit-by-bit XOR reg. with constant
Shift	Shift left logical	`sll x5, x6, x7`	`x5 = x6 << x7`	Shift left by register
	Shift right logical	`srl x5, x6, x7`	`x5 = x6 >> x7`	Shift right by register
	Shift right arithmetic	`sra x5, x6, x7`	`x5 = x6 >> x7`	Arithmetic shift right by register
	Shift left logical immediate	`slli x5, x6, 3`	`x5 = x6 << 3`	Shift left by immediate
	Shift right logical immediate	`srli x5, x6, 3`	`x5 = x6 >> 3`	Shift right by immediate
	Shift right arithmetic immediate	`srai x5, x6, 3`	`x5 = x6 >> 3`	Arithmetic shift right by immediate
Conditional branch	Branch if equal	`beq x5, x6, 100`	`if (x5 == x6) go to PC+100`	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	`if (x5 != x6) go to PC+100`	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	`if (x5 < x6) go to PC+100`	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	`if (x5 >= x6) go to PC+100`	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	`x1 = PC+4; go to PC+100`	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	`x1 = PC+4; go to x5+100`	Procedure return; indirect call

→ All instructions

Operands of the Computer Hardware

↳ There are 3 locations in Hardware

Registers | Memory | Immediate



We are focused on the 3 locations that are "✓"

Registers

→ RISC-V operations use registers which are special locations in the hardware

→ They are limited in number but faster than memory

Because Smaller is Faster ★

Incase there were large no. of registers, this would increase clock cycle time

→ In RISC-V, each register has specific use case

* In RISC-V There are always 32 Register ONLY
In RV-64 ⇒ 64 bit registers
In RV-32 ⇒ 32 bit registers

	Name	Register Number	Usage
★	zero	x0	Constant value 0
	ra	x1	Return address
★	sp	x2	Stack pointer
	gp	x3	Global pointer
	tp	x4	Thread pointer
★	t0-2	x5-7	Temporaries
★	s0/fp	x8	Saved register / Frame pointer
★	s1	x9	Saved register
	a0-1	x10-11	Function arguments / return values
	a2-7	x12-17	Function arguments
★	s2-11	x18-27	Saved registers
	t3-6	x28-31	Temporaries

Zero register ⇒ Always zero

Saved registers ⇒ Hold variables

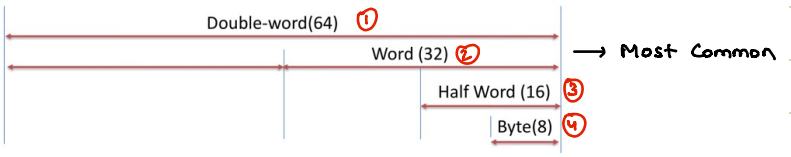
Temporaries ⇒ Hold intermediate values

Now lets write a = (b+c) - (d+e) with registers,

```
add  x5, x20, x21   // t0 = b+c
add  x6, x22, x23   // t1 = d+e
sub  x19, x5, x6    // a = t0-t1
```

→ In RISC-V, we also have 4 datatypes based on bit size

0x0F	0x0E	0x0D	0x0C	0x0B	0x0A	0x09	0x08	← Address in Hexadecimal → bytes
15	14	13	12	11	10	9	8	← Address in decimal
63 56	55 48	47 40	39 32	31 24	23 16	15 8	7 0	→ bits



→ Most Common

double word = 64 bits = 8 bytes

word = 32 bits = 4 bytes

half word = 16 bits = 2 bytes

byte = 8 bits = 1 byte

Memory

→ Processor can hold only upto some data, but sometimes we need to store complex data structures like arrays, stacks etc.,

→ Memory is slow compared to registers but large

→ There are multiple ways to address memory but we need the most efficient way

One way is **word addressable memory**

↳ In this, each 32-bit data word has unique address

Data will be stored in successive 32 bit locations (multiples of 32)

So it will be stored like

```
0x80000004 → 0x12345678
0x80000003 → 0x76EBAE78
0x80000002 → 0x1256ACDB
0x80000001 → 0x9734A8B7
0x80000000 → 0x125654AD
```

But the problem with this is that it won't be used efficiently

i) Array elements occupy only least significant 8 bits

ii) Read & write can't be byte level

Instead we use **byte addressable memory**

↳ In this each byte has its own address

So its stored like

```
0x80000004 → 0x12
0x80000003 → 0x76
0x80000002 → 0x12
0x80000001 → 0x97
0x80000000 → 0x12
```

This is used in RISC-V

Data type	Size in bytes	Physical Address
Double word	8	PA = BA + Index x 8
Word	4	PA = BA + Index x 4
Half word	2	PA = BA + Index x 2
Byte	1	PA = BA + Index x 1

General Formula
PA = BA + Index * Size of data (in bytes)

How exactly does RISC-V access such large data structures?

↳ RISC-V supports **Load and Store** architecture

↳ This seperates instructions into 2 types → Operate on memory
→ Operate on data in register file
→ Both locations don't mingle instructions

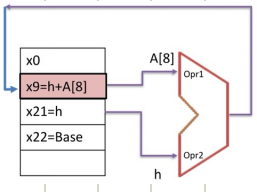
So RISC-V must include instructions that transfer data b/w memory & registers which are called data transfer instructions

↳ ex : lb, lh, lw, ld, sb, sh, sw, sd

For example, Let h be associated with x21 & base address of array A is x22.

Then A[12] = h + A[8] in C ⟹

```
lw  x9, 32(x22)   // x9 = A[8]
add x9, x21, x9   // x9 = h + A[8]
sw  x9, 48(x22)   // A[12] = x9
```



Address	Data (32bits)
2032	A[12]
2028	A[11]
2024	A[10]
2020	A[9]
2032	A[8]
2028	A[7]
2024	A[6]
2020	A[5]
2016	A[4]
2012	A[3]
2008	A[2]
2004	A[1]
2000	A[0]

x9 stored here →
x22 base address →

Constants / Immediates

→ Not everytime we need a variable. Sometimes we use a constant in operations

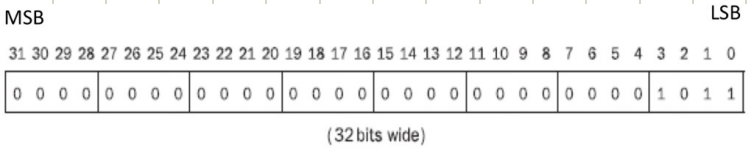
Now to load a constant into memory, we can use

```
addi  x22, x22, 4   // x22 = x22 + 4
```

↳ most popular instruction in most RISC-V programs

Signed and Unsigned Numbers

→ Take a 32 bit (word) number



In unsigned format, we can represent the range 0 to $2^{32} - 1 = 4294967295$ numbers

Now for signed format,

31	30 0
Sign	Magnitude

↳ Range is from -2^{31} to $(2^{31} - 1)$

↳ −2147483648 to 2147483647

How to find negative numbers?

So we use something called 2's complement

It follows the logic $x + \bar{x} + 1 = 0$

Lets try some examples

i) Represent −2 using 8, 16, 32, 64 bit number system

- signed byte ⇒ 0xFE
- signed half word ⇒ 0xFFFE
- signed word ⇒ 0xFFFFFFFE
- signed double word ⇒ 0xFFFFFFFFFFFFFFFE

ii) Represent 127 using 8, 16, 32, 64 bit number system

- signed byte ⇒ 0x7F
- signed half word ⇒ 0x007F
- signed word ⇒ 0x0000007F
- signed double word ⇒ 0x000000000000007F

Sign Extension

→ Sometimes we must represent a number that is represented in n bits, in more than n bits

→ We use a shortcut, to take MSB (sign bit) & replicate it to fill the new bits

ex: +7 in 4 bits is 0111

in 8 bits is 0000 0111

−7 in 4 bits is 1001

in 8 bits is 1111 1001

→ Since we can't represent the number in same way, RISC-V uses 2 types of byte loads:

`lbu` → treats number as unsigned number & zero-extends to fill leftmost bits of register

`lb` → works with signed integers

Representation of Instructions

→ Computer hardware only understands 1's & 0's & giving something like add x5, x6, x7 is meaningless.
So in order to make hardware understand, we use instruction formats.
Each instruction has different format based on data it takes in & gives out

	Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
R-Type	Register/register	funct7							rs2					rs1					funct3			rd					opcode							→ add, sub, sll, xor, srl, sra, or, and
I-Type	Immediate	imm[11:0]												rs1					funct3			rd					opcode							→ lb, lh, lw, ld, lbu, lhu, addi, slli, xori, ori, andi, jalr
U-Type	Upper Immediate	imm[31:12]																				rd					opcode							→ lui, auipc
S-Type	Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode							→ sb, sh, sw, sd
B-Type	Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]			[11]		opcode							→ beq, bne, blt, bge, bltu, bgeu
J-Type	Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode							→ jal, auipc

- opcode (7 bit): partially specifies which of the 6 types of *instruction formats*
- funct7 + funct3 (10 bit): combined with opcode, these two fields describe what operation to perform
- rs1 (5 bit): specifies register containing first operand
- rs2 (5 bit): specifies second register operand
- rd (5 bit):: Destination register specifies register which will receive result of computation

Lets try few examples to understand each format

i) R-Type : add x8, x9, x10 // x8 = x9 + x10

Now writing in its format ⇒ funct7 0000000 | rs2 01010 | rs1 01001 | funct3 000 | rd 01000 | opcode 0110011 ⇒ 00000000101001001000010000110011
0 0 A 4 8 4 3 3
0x00A48433

ii) I-Type : addi x11, x12, 9 // x11 = x12 + 9

Now writing in its format ⇒ imm[11:0] 000000001001 | rs1 01100 | funct3 000 | rd 01011 | opcode 0010011 ⇒

iii) S-Type : sw x9, 120(x10) // x9 ⇒ x10 + 120

Now writing in its format ⇒ imm[11:5] 0000011 | rs2 01001 | rs1 01010 | funct3 010 | imm[4:0] 11000 | opcode 0100011

Format	Instruction	Opcode	Funct3	Funct6/7
R-type	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I-type	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S-type	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
SB-type	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U-type	lui	0110111	n.a.	n.a.
UJ-type	jal	1101111	n.a.	n.a.

↙ Refer for opcodes, funct3 & funct7

Logical Operations

Logical operations	C operators	Java operators	RISC-V instructions
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Shift right arithmetic	>>	>>	sra, srai
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	xori

→ Can use for multiplication by 2
→ Can use for division by 2

Lets try an example

Q. The following source code will, for example, move bits [7:4] of the value in x6 to the 4 least significant bits [3:0] of x5 nullifying all its other bits:

A.

Instruction	Comment
addi x6, x0, 0x123	x6=0x00000123=0000 0000 0000 0000 0000 0001 **0010** 0011b
slli x7, x6, 24	x7=0x23000000=0010 0011 0000 0000 0000 0000 0000 0000b
srli x5, x7, 28	x5=0x00000002=0000 0000 0000 0000 0000 0000 0000 0010b

First assign to variable
Then clear everything from [31:8]
Then bring [31:28] to [3:0], Thus nullifying other bits

Q. Code for $((888/8)-(123*4))^*2$ in x5

A.

```
addi  x6, x0, 888
srli  x6, x6, 3
addi  x7, x0, 123
slli  x7, x7, 2
sub   x5, x6, x7
slli  x5, x5, 1
```

Q For the following C statement, write the corresponding RISC-V assembly code. Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively.
B[8] = A[i–j];

A.

```
sub x30, x28, x29   # compute i-j  (x28 – x29)
slli x30, x30, 2    # offset = Index * 4 = x30<<2 = (i-j) * 4
add x3, x30, x10    # physical address of A[i-j] = Base Address + 4 x [i-j]
lw x30, 0(x3)       # load A[i-j]
sw x30, 32(x11)     # store in the content of x30 into B[8] =B[BA+4x8] = B[x11 + 32]
```

Q. Translate the following C code to RISC-V. Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively. Assume that the elements of the arrays A and B are 4-byte size:
B[8] = A[i] + A[j];

A.

```
slli x28, x28, 2      # x28 = i*4 ( Address of ith element)
add x10,x10,x28       # x10 = &A[i] – Effective Address (Physical Address)
lw x28, 0(x10)        # x28 = A[i]
slli x29, x29, 2      # x29 = j*4
add x12,x11,x29       # x12 = &B[j] – Effective Address (Physical Address)
lw x29,0(x12)         # x29 = B[j]
add x29, x28, x29     # x29 = A[i]+B[j]
sw x29, 32(x11)       # B[8] = B[i]+B[j]
```

Q. Translate the following **RV64I code to C**. Assume that the variables f, g, h, i, and j are assigned to registers x5, x6, x7, x28, and x29, respectively. Assume that the base address of the **arrays A and B are in registers x10 and x11**, respectively.

A.

```
# f = 2*(& A)

addi x30, x10, 8        #  x30 = & A[1] ; Address of A[1]
addi x31, x10, 0        #  x31 = &A ; Base Address of A
sw x31, 0(x30)          #  A[1] = &A ; A[1] = Base address of A
lw x30, 0(x30)          #  x30 = A[1] = & A =Base address of A
add x5, x30, x31        #  f = &A + &A = 2*(&A)
```

Q. Assume that registers x5 and x6 hold the values x80000000 and 0xD0000000, respectively.
a) What is the value of x30 for the following assembly code?
add x30, x5, x6
b) Is the result in x30 the desired result, or has there been overflow?
c) For the contents of registers x5 and x6 as specified above, what is the value of x30 for the following assembly code?
sub x30, x5, x6
d) Is the result in x30 the desired result, or has there been overflow?
e) For the contents of registers x5 and x6 as specified above, what is the value of x30 for the following assembly code?
add x30, x5, x6
add x30, x30, x5
f) Is the result in x30 the desired result, or has there been overflow?

A.
a) 0x50000000
b) overflow
c) 0xB0000000
d) no overflow
e) 0xD0000000
f) overflow

Instructions for making decisions

→ RISC-V supports decision making instructions similar to if-statement with go-to

→ beq branch = =

bne branch ! =

blt branch <

bge branch >=

bltu branch < (Unsigned)

bgeu branch >= (Unsigned)

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

→ Usually for branches, we use PC relative addressing

↳ normally it increments by 4 (Jump from this line of code to next)

BUT when branching we tell PC to go to some other address

We can only branch forward or backward by range of $\pm 2^{n}$ units (unit = 2 bytes / halfword)

So around 4 KB around current instruction

We don't use bytes as unit (we don't branch by odd no. of bytes) because we would branch in middle of instruction

in G Extension, instructions are 4 bytes

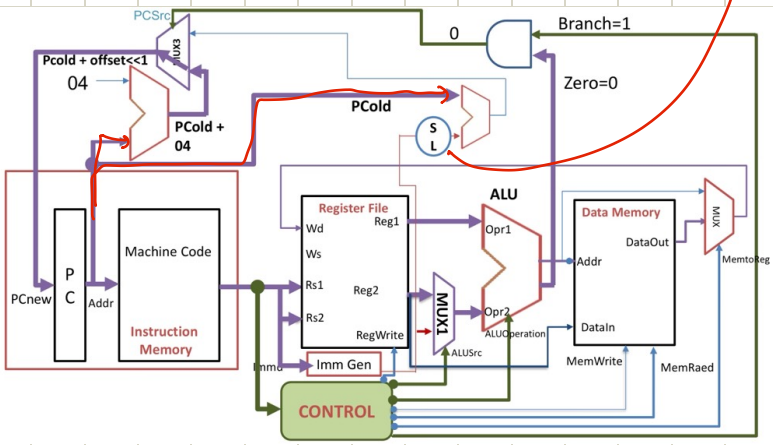
in C Extension, instructions are 2 bytes

Branch	[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	[11]	opcode

↳ If we observe instruction format as well, imm only from [12 : 1], 0th element is not required because it HAS TO BE 0 (even no.)

→ SL : Shift Left, It will shift by 1 bit

How does processor behave?



Based on whether to branch, the control signal will send if it satisfies the condition to branch

Both $PC_{old}+4$ & PC_{old} + offset are sent to MUX for decision making & sent back to Program Counter as PC_{New}

Q. Here is a traditional loop in C:

```
while (save[i] == k)
i += 1;
```

Assume that **i and k correspond to registers x22 and x24** and the **base of the array save is in x25.** What is the RISC-V assembly code corresponding to this C code?

A.

Address	Instruction
0x8000	loop: slli x10, x22, 2 // x10 = x22 * 4 = i * 4
0x8004	add x10,x10,x25
0x8008	lbu x9, 0(x10) // x9=save[i]
0x800C	bne x9,x24,exit
0x8010	addi x22,x22,1
0x8014	beq x0,x0,loop
0x8018	exit

Q. // C[i] = A[i] +B[i]

A.

```
.data
A: .word   0x00000005,0x00000008,0x000000A0,0x000000001
B: .word   0x80,0x0f,0x01,0x01
C: .word   0,0,0,0
.text
la x18,A   # loads the location address of the specified SYMBOL
la x19,B
la x20,C

addi x24,x0,4

back: lw x21,0(x18)
lw x22,0(x19)

add x23,x21,x22
sw x23,0(x20)

addi x18,x18,4
addi x19,x19,4
addi x20,x20,4

addi x24,x24,-1
bne x24,x0,back
```

U2 Instructions & Data Flow Model

Wide immediate

→ We have noticed that from I-Type instruction format that it supports only 12-bit immediate values But 12 bits sometimes it may be too small.
So we use an instruction called lui (load upper immediate)

→ lui loads a 20 bit (other than the normal 12 bit in I-Type) & then fills lower 12 with zeros

Q. Write an assembly code to initialize 0x003D0500 in register X19

A.
```
lui X19,0x003D0  // Loads upper 20 bits (5 bytes) & x19 = 0x003D0000
addi X19,X19,0X500 // Then we must fill the lower 12 bits & x19 = 0x003D0500
```

x19	0x0000 0000	0000 0000 0000 0000 0000	0000 0000 0000 → Before execution
x19	0x003D0000	0000 0000 0011 1101 0000	0000 0000 0000 → After lui
imm12	0x500	0000 0000 0000 0000 0000	0101 0000 0000 → Add immediate
x19	0x003D0000	0000 0000 0011 1101 0000	0101 0000 0000 → Final result

This has no issue when MSB of immediate is 0
It has an issue when MSB of immediate is 1 because addi 12-bit immediate is sign extended

Q. Write a assembly code to initialize 0xDEADBEEF in register X19

A.
```
lui X19,0xDEADB
addi X19,X19,0XEEF
```

x19	0x0000 0000	0000 0000 0000 0000 0000	0000 0000 0000
x19	0xDEADB000	1101 1110 1010 1101 1011	0000 0000 0000
imm12	0xFFFFFEEF	1111 1111 1111 1111 1111	1110 1110 1111
x19	0xDEADAEEF	1101 1110 1010 1101 1010	1110 1110 1111
x19	0xDEADAEEF - It is Not a expected Initialization. What has to be done to initialize x19 with the correct result?		

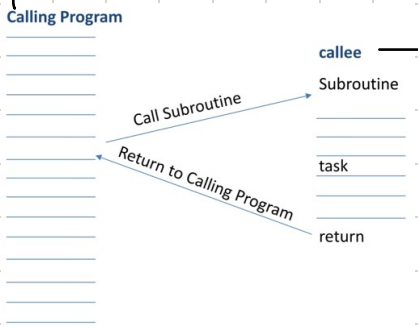
So to prevent this, we only have to increment by 1 before initializing
instead of 0x DEAD B we need to use 0x DEAD C

↳ Note while loading upper immediate, it is 20 bits ONLY [20:1]

Supporting Procedures in Computer Hardware

→ Procedures are stored subroutines that perform specific tasks based on parameters provided

↳ Initiates procedure & provides necessary parameters



↳ Executes series of stored instructions based on parameters provided by caller & returns control to caller

Parameters act as an interface b/w procedure & rest of program data, since they can pass values & return results

To do this,

Procedures' Procedure:

i) Acquires resources

ii) Performs task

iii) Covers his/her tracks

iv) Returns to point of origin with desired result

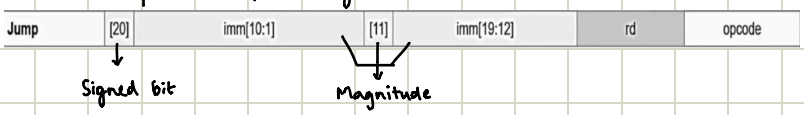
```
jal  rd, imm       # rd = pc+4; pc += imm
jalr rd, rs1, imm  # rd = pc+4; pc = rs1+imm
```

jal : jump and link

jalr : jump and link register

jal → direct jumping branches to an address based on $PC_{new} = PC_{old}$ + (signed Offset) and simultaneously saves the address of the following instruction in destination address rd

(signed Offset) → $+/- 2^{20}$



jalr → indirect jumping branches to an address based on $PC_{new} = PC_{old}$ + (Signed Immediate [11:0]) and simultaneously saves the address of the following instruction in destination address rd

Immediate [11:0] → $+/- 2^{11}$

11^{th} Bit Sign

[10:0] Magnitude

auipc builds 32-bit addresses relative to PC. It is used for PC relative addressing of symbols

We use auipc over lui because we get an address relative to PC, lui loads fixed absolute address

Lets see how jal, jalr, auipc are used

```
jal x1, 0x800
```
→ Jumps forward by 2048 bytes & saves the current PC value +4 into x1

↳ x1 = PC+4

& PC = PC + 0x800

```
jalr x0, 0(x1)
```
→ Jumps back to address in x1 & x0 = PC+4 ↳ This is pointless when we just need to go back

Now if we wanted to jump to addresses more than 20 bits

```
auipc x10, 0x7FFFF
jalr  x1, x10, 0x010
```

→ x10 = PC + (0x7FFFF << 12) = PC + 0x7FFFF000

→ jump to x10 + 0x010 = 0x7FFF010 & x1 = PC+4

Q. **Write a program to initialize x10 with**
a) 0x7FFFF718
b) 0x7FFFF818
Assume PCpresent =0x00000008

A. a) PC = 0x00000008
Target = 0x7FFFF718
Relative = 0x7FFFF710

```
auipc  x10, 0x7FFFF
addi   x10, x10, 0x710
```

b) PC = 0x00000008
Target = 0x7FFFF818
Relative = 0x7FFFF810

```
auipc  x10, 0x7FFFF
addi   x10, x10, 0x810
```

Q. **Write a program to initialize PC with absolute value**
a)0x7FFFF718
b)0x7FFFF818
Assume PCpresent =0x00000008. Use lui and jalr Instruction

A. a) Target = 0x7FFFF718
Upper 20 = 0x7FFFF
Lower 12 = 0x718

```
lui  x10, 0x7FFFF
jalr x0, x10, 0x718
```

b) Target = 0x7FFFF818
Upper 20 = 0x7FFFF
Lower 12 = 0x818

```
lui  x10, 0x7FFFF
jalr x0, x10, 0x818
```

Q. **Assume base address of symbol num3 in data memory to be accessed is 0x1000 00AC and PCpreset = 0x0000 0008. Write program to initialize base address of num3 in x11 using auipc and addi**

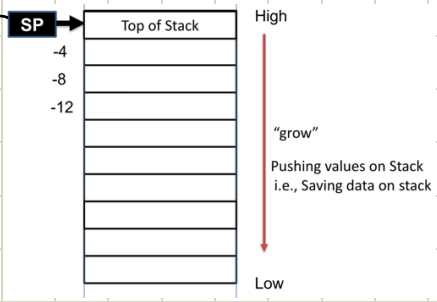
A. PC = 0x00000008
Target = 0x100000AC
Relative = 0x100000A4

```
auipc  x11, 0x10000
addi   x11, x11, 0x0A4
```

→ Now another issue we face is, requirement of more registers
Since we cover our tracks after procedure is completed, we must restore the values that were originally used before executing the procedure.
↳ we need to spill registers into memory

→ x10 - x17 are arguement registers
we need to use more registers means we use a data structure known as stack
↳ Last in First out Queue

Stack pointer holds the most recently allocated address in a stack



In RISC-V, SP is x2 register
→ SP is adjusted by 1 word for each register that is saved
(Push - saving data on stack
Pop - Removing data on stack)

↳ Stack grows from higher addresses to lower addresses
This conventions means you push values onto stack by decrementing SP by 4

Leaf Procedure ?

→ A leaf procedure is a function that doesn't call any other functions (it is the leaf of call tree, meaning it doesn't branch further) → 

Procedure

i) It must save all registers used by procedure

ii) Body of procedure performs task

iii) value of the registers used by procedure must be saved in parameter register

iv) Restore value of registers that were stored in stack by popping them from stack

v) Return from procedure

```c
#include <stdio.h>
int main(){
    int y;
    ...
    y = diffofsums(2,3,4,5);
    ...
}

int diffofsums(int f, int g, int h, int i){
    int result;
    result = (f+g)-(h+i);
    return result;
}
```

```
main :
    addi x10,x0,2
    addi x11,x0,3
    addi x12,x0,4
    addi x13,x0,5
    jal x1,diffofsums
    add x18,x10,x0

diffofsums :
    addi sp,sp,-12  //make space on stack for 3 registers
    sw x5,8(sp)
    sw x6,4(sp)  //x5 and x6 are temporaries
    sw x20,0(sp)
    add x5,x10,x11
    add x6,x12,x13
    sub x20,x5,x6
    addi x10,x20,0  //return value in argument register
    lw x20,0(sp)   // returning back initial values of registers
    lw x6,4(sp)
    lw x5,8(sp)
    addi sp,sp,12  //deallocate stack space
    jalr x0,0(x1)  //return to caller
```

→ Makes space in Stack

→ Puts back into register & clear Stack

→ Few more examples

```
.data
ary: .word 11,22,33,44,55,66,77,88,99
res1: .string 0xFF
res2: .string 0x00

.text
la x4,ary
la x5,res1
la x6,res2
li x12,0
li x5,9
li x10,88
back: lw x11,0(x4)
      jal x1,check
      addi x4,x4,4
      addi x5,x5,-1
      bne x5,x0,back
check: beq x11,x10,found
       lw x12,0(x6)
       jalr x0,0(x1)
found: lw x12,0(x5)
       jalr x0,0(x1)
```

→ Code is searching an array of 9 numbers for 88:

If found, load res1 (0xFF) into x12

If not, load res2 (0x00) into x12

→ The procedure strcpy copies string y to string x using the null byte termination convention of C:

```c
void strcpy (char x[], char y[])
{
    size_t i;
    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
        i += 1;
}
```

What is the RISC-V assembly code?

→

```
strcpy:  addi sp, sp, -4      #// adjust stack for 1 more item
         sw x19, 0(sp)        #// save x19 as x19 is used for i
         add x19, x0, x0      # // i = 0+0

L1:      add x5, x19, x11     # // EA address of y[i] in x5
         lbu x6, 0(x5)        # // x6 = y[i]
         add x7, x19, x10     #// address of x[i] in x7
         sb x6, 0(x7)         #// x[i] = y[i]

         beq x6, x0, L2

         addi x19, x19, 1     # // i = i + 1
         jal x0, L1   # // go to L1

L2:      lw x19, 0(sp)         #// restore old x19
         addi sp, sp, 4        # // pop 1 word off stack
         jalr x0, 0(x1)        #// return
```

Nested Procedures?

→ Procedures that invoke other procedures.
Recursive procedure invoke clones of themselves

Lets see an example before understanding the problem & finding solution.

```c
#include <stdio.h>

int fact(int n) {
    if (n <= 0) {
        return 1;               // base case
    } else {
        return n * fact(n - 1); // recursive case
    }
}

int main() {
    int n = 5;
    int result = fact(n);
    printf("Factorial of %d = %d\n", n, result);
    return 0;
}
```

→

```
.data
                        # store the base address (0x12000000)
n:      .word 5         # factorial input (5)
result: .word 0         # space for storing factorial result

.text
.globl main
main:
    la   x8, base       # x8 = &base
    lw   x9, 0(x8)      # x9 = 0x12000000 (base address)
    lw   x10, 4(x8)     # x10 = n (5)
    jal  x1, fact       # call factorial

    sw   x10, 8(x8)     # store factorial result at result
    j exit

fact:
    addi sp, sp, -8
    sw   x1, 4(sp)
    sw   x10, 0(sp)

    addi x5, x10, -1
    bge  x5, x0, L1

    addi x10, x0, 1
    addi sp, sp, 8
    jr   x1

L1: addi x10, x10, -1
    jal  x1, fact

    addi x6, x10, 0
    lw   x10, 0(sp)
    lw   x1, 4(sp)
    addi sp, sp, 8
    mul  x10, x10, x6
    jr   x1

exit: nop
```

Problem with Recursive / Nested Procedures

→ When we use nested procedure calls or recursion, multiple calls may exist at same time

→ x1, a0-a7, t0-t6, s0-s11 are all shared hardware resources
If the values aren't stored somewhere safe, next call overwrites them & data is lost

To solve this :

i) Before making call, save any arguement registers / temporary registers it still needs after call

ii) Upon being called, adjust the stack pointer to make room in stack & store them there

iii) Upon returning, Callee restores saved registers, any registers it had saved & reset sp.

The registers which are preserved & not preserved after procedure call

Preserved	Not preserved
Saved registers: x8-x9, x18-x27	Temporary registers: x5-x7, x28-x31
Stack pointer register: x2(sp)	Argument/result registers: x10-x17
Frame pointer: x8(fp)	
Return address: x1(ra)	
Stack above the stack pointer	Stack below the stack pointer

Last few base instructions

→ Comparision instructions ⇒

Instruction	Name	Format	Description
Add upper immediate to PC	auipc	U	Add 20-bit upper immediate to PC; write sum to register
Set if less than	slt	R	Compare registers; write Boolean result to register
Set if less than, unsigned	sltu	R	Compare registers; write Boolean result to register
Set if less than, immediate	slti	I	Compare registers; write Boolean result to register
Set if less than immediate, unsigned	sltiu	I	Compare registers; write Boolean result to register

→ Standard Extensions – The ISA of RISC-V is small so extra features are packaged as extensions,

- M → Multipy (mul, div, rem)
- A → Atomic (Lr.w, sc.w)
- F → Single-precision floating point
- D → Double-precision floating point
- C → Compressed (use 16 bit versions of addi, lw, jalr etc)

Mnemonic	Description	Insn. Count
I	Base architecture	51
M	Integer multiply/divide	13
A	Atomic operations	22
F	Single-precision floating point	30
D	Double-precision floating point	32
C	Compressed instructions	36

Now instead of looking at snippets, Lets C 😜 some codes

Swap

C Code :

```
void swap(int v[], size_t k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Assembly Code :

```
swap:
    slli    x6, x11, 2    // reg x6 = k * 4
    add     x6, x10, x6   // reg x6 = v + (k * 4)
    lw      x5, 0(x6)     // reg x5 (temp) = v[k]
    lw      x7, 4(x6)     // reg x7 = v[k + 1]
    sw      x7, 0(x6)     // v[k] = reg x7
    sw      x5, 4(x6)     // v[k+1] = reg x5 (temp)
    jalr    x0, 0(x1)     // return to calling routine
```

Sort

C Code :

```
void sort (int v[], size_t int n)
{
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v,j);
        }
    }
}
```

Assembly Code :

Saving registers

```
sort:   addi sp, sp, -20        # make room on stack for 5 registers
        sw x1, 16(sp)           # save return address on stack
        sw x22, 12(sp)          # save x22 on stack
        sw x21, 8(sp)           # save x21 on stack
        sw x20, 4(sp)           # save x20 on stack
        sw x19, 0(sp)           # save x19 on stack
```

Procedure body

Move parameters		addi x21, x10, 0 # copy parameter x10 into x21 addi x22, x11, 0 # copy parameter x11 into x22
Outer loop		addi x19,x0, 0 # i = 0 for1tst:bge x19, x22, exit1 # go to exit1 if i >= n

Inner loop		addi x20, x19, -1 # j = i - 1
		for2tst:blt x20, x0, exit2 # go to exit2 if j < 0
		slli x5, x20, 2 # x5 = j * 4
		add x5, x21, x5 # x5 = v + (j * 4
		lw x6, 0(x5) # x6 = v[j]
		lw x7, 4(x5) # x7 = v[j + 1]
		ble x6, x7, exit2 # go to exit2 if x6 < x7

Pass parameters and call		addi x10, x21, 0 # first swap parameter is v
		addi x11, x20, 0 # second swap parameter is j
		jal x1, swap # call swap

Inner loop		addi x20, x20, -1 j for2tst jal, x0 for2tst # go to for2tst
Outer loop	exit2:	addi x19, x19, 1 # i += 1 jal, x0 for1tst # go to for1tst

Restoring registers

```
exit1:  lw x19, 0(sp)           # restore x19 from stack
        lw x20, 4(sp)           # restore x20 from stack
        lw x21, 8(sp)           # restore x21 from stack
        lw x22, 12(sp)          # restore x22 from stack
        lw x1, 16(sp)           # restore return address from stack
        addi sp, sp, 20         # restore stack pointer
```

Procedure return

```
        jalr x0, 0(x1)          # return to calling routine
```

Array vs Pointers

→ Lets take a task to clear an array & do it using array and pointer method

1. Array Version (`clear1`)

C code:

```c
void clear1(int array[], int size) {
    for (int i = 0; i < size; i++)
        array[i] = 0;
}
```

RISC-V Assembly:

```asm
# Registers:
# x10 = array (base address)
# x11 = size
# x5  = i (loop counter)
# x6  = temporary for byte offset
# x7  = address of array[i]

    addi x5, x0, 0       # i = 0
loop1:
    slli x6, x5, 2       # x6 = i * 4
    add  x7, x10, x6     # x7 = address of array[i]
    sw   x0, 0(x7)       # array[i] = 0
    addi x5, x5, 1       # i++
    blt  x5, x11, loop1  # if i < size, repeat
```

✅ Each iteration recalculates the element address (`i*4 + base`).

→ Recalculates address every iteration

2. Pointer Version (`clear2`)

C code:

```c
void clear2(int array[], int size) {
    for (int *p = array; p < &array[size]; p++)
        *p = 0;
}
```

RISC-V Assembly (optimized version):

```asm
# Registers:
# x10 = array (base address)
# x11 = size
# x5  = pointer p
# x6  = temporary for end offset
# x7  = end address

    addi x5, x10, 0      # p = &array[0]
    slli x6, x11, 2      # x6 = size * 4 (byte size)
    add  x7, x10, x6     # x7 = address just after last element
loop2:
    sw   x0, 0(x5)       # *p = 0
    addi x5, x5, 4       # p++
    bltu x5, x7, loop2   # if p < end address, repeat
```

✅ Loop increments the pointer directly. The end address is precomputed outside the loop → fewer instructions.

→ Increments pointer directly (Fewer instructions per loop)

Q. In the pointer version of dealing with array _____ the loop, thereby reducing the instructions executed per iteration from five to three.

a) moves the scaling shift and the array-bound addition inside.
b) moves the scaling shift and the array bound addition outside. ✓
c) Remain the same in
d) None of these

Q. Pointers are variables that _____ and can avoid indexing complexity

a) store the address of another variable. ✓
b) store the value of another variable.
c) store address of pointer variable
d) None of these

Q. Consider the following C statement

```
int array [] = {6,10,20,5,4}
int *p=array;
```

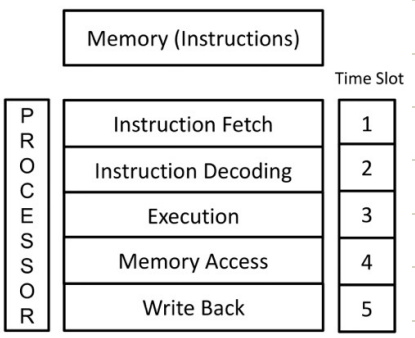
If array [0] is stored in memory location 0x10000004 what will be the content of pointer variable p?

a) p=array [0]
b) P=0x10000004. ✓
c) P=array [5]
d) None of these

→ p points to a[0]
but contains &a[0]

Processor

→



→ We know that these 5 steps are important but not all instructions use all of them. But the first 2 are common & rest are used based on instruction

The last 3 can be executed using ALU operations

For example,

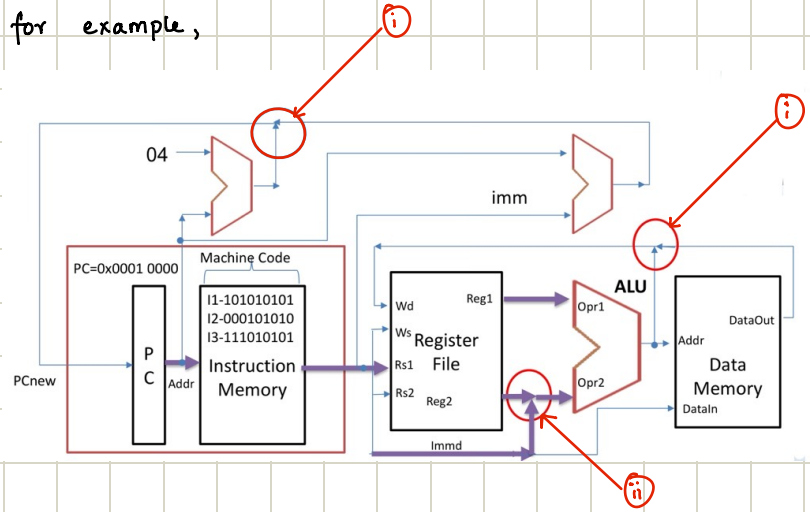
Store → access memory, write data

Load → access memory, read data from load & load to destination register

Arithmetic logical → Write data from ALU to register

Branch → Need to change to next instruction address based on comparision

→ Now for example,

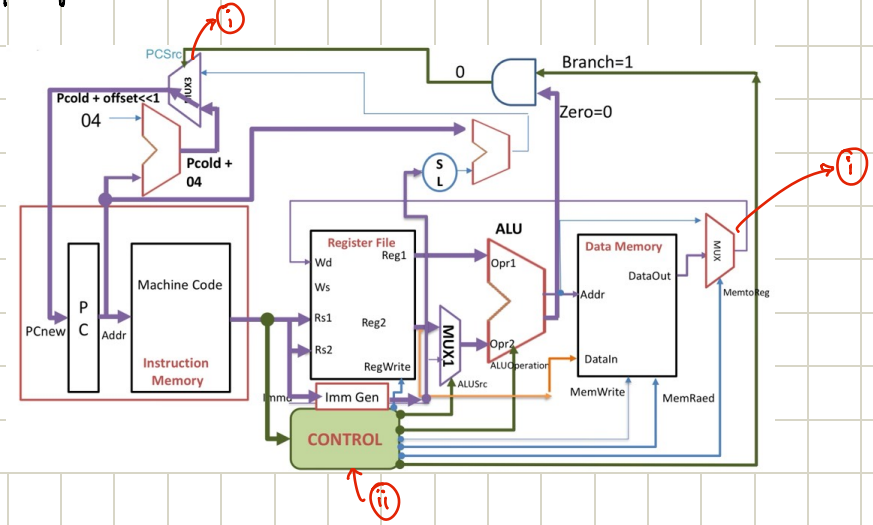


There are 2 problems.

i) Data coming from 2 different units going to a common point can't just be joined by joining wires. We must a MUX so it decides which signal to use

ii) Several units must be controlled depending on type of instruction, So we use Control Signal

Now fixing these 2 issues, here is a better abstract!



There are 7 control signals

i) RegWrite

ii) MemRead

iii) MemWrite

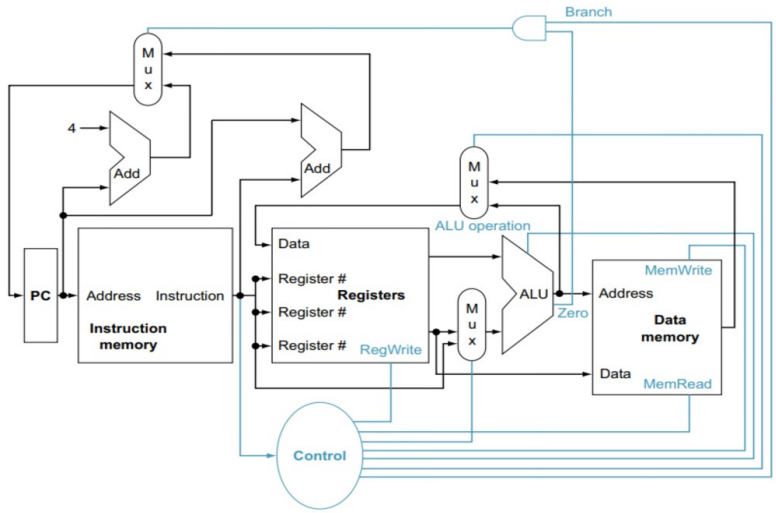
iv) ALU Operation

v) Branch

vi) ALUSrc

vii) MemtoReg

Here lets take a few example instructions & their control signals



Instruction	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch
R-Type	0	0	1	0	0	0
lw	1	1	1	1	0	0
sw	1	X	0	0	1	0
Beq	0	X	0	0	0	1

Data path Elements

→ We have 2 types of datapath elements

Combinational

→ Elements that operate on data values meaning o/p depends only on current input

ex: ALU, MUX, Immediate Generator

Sequential

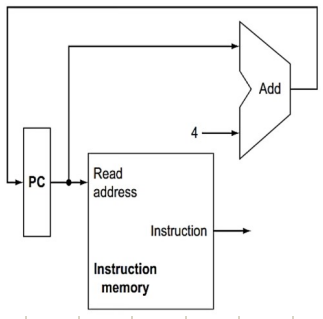
→ State elements contain state (has some memory) because they hold same state even after power off and restart.

ex: Instruction memory, Data Memory, Register File, PC

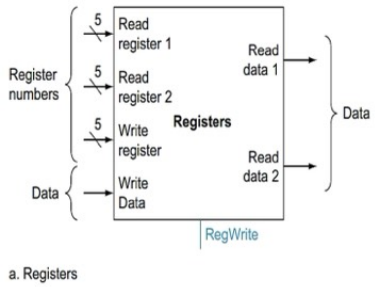
Why Clock?

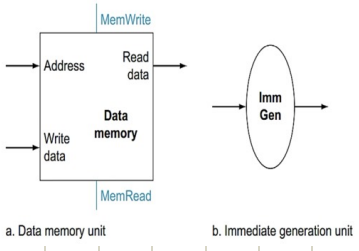
→ Clocking methodology defines when signal can be read or written, making it easy to predict

Datapath



Instruction Fetching & PC incrementing





Memory Access

Instruction implementation & Execution

Instruction	Use of ALU
Load/Store Instructions	Calculate address using 12-bit offset. Use ALU, but sign-extend offset
Branch Instruction	Compare operands Use ALU, subtract and check Zero output.
	Calculate target address **Sign-extend displacement Shift left 1 place** (halfword displacement) and **Add to PC** value
R-type	Function depends on op-code

Multiple Level of Control

→ Using multiple levels of control can reduce the size of main control unit

→ Using several smaller control units may also reduce latency of control unit

Latency is a critical factor to determine clock cycle time

For example, `lw x1, 0(x2)`

High level control detects instruction as load & signals ALU to do addition, memory read enable, write back needed.

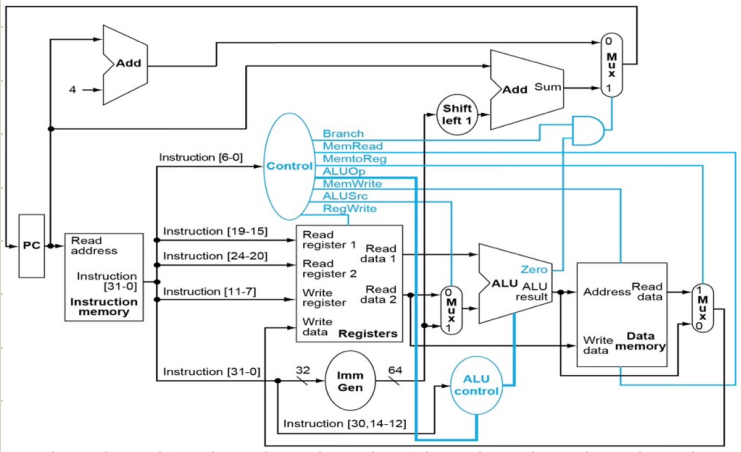
Low level ALU control enables memory read & passes result to register file

→ ALU operation

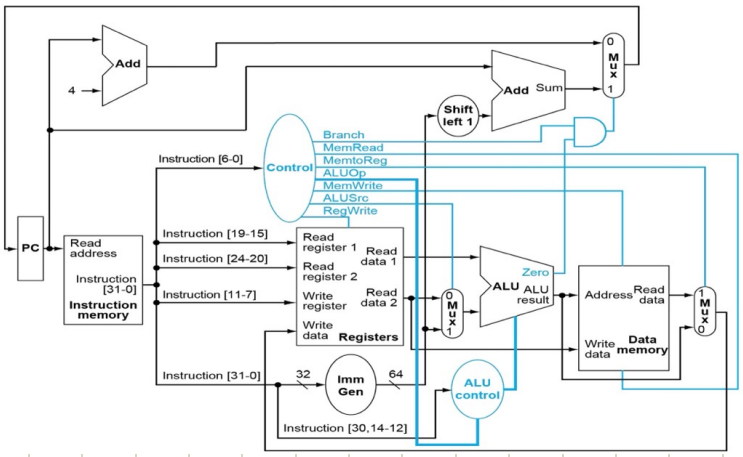
Operation	Desired Opern	ALUoperation	
Load/Store	Addition	00	10
Branch	Subtraction	01	10
Addition	Addition	00	10
Subtraction	Subtraction	01	10
ANDing	AND	00	00
ORing	OR	00	01

Arithmetic unit is used only when ALUop = 10_2

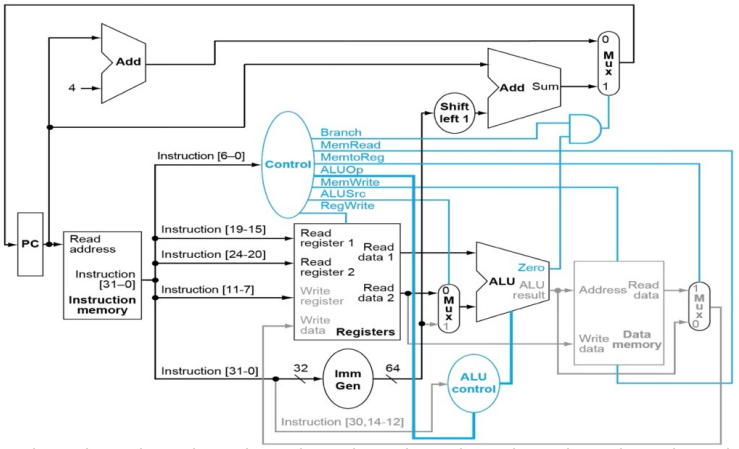
Datapath with the control unit (Multilevel Control Unit)



R-type Instruction



Branch on Equal Instruction



Load Instruction

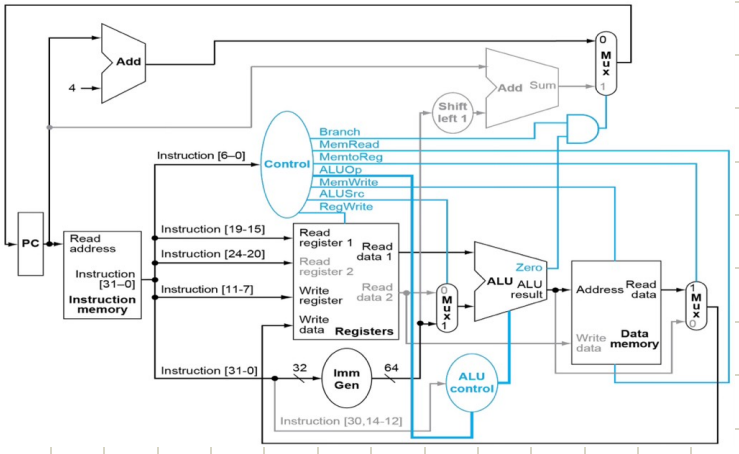


FIGURE 4.22 The setting of the control lines is completely determined by the opcode fields of the instruction. The first

Input or output	Signal name	R-format	lw	sw	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Control line status for subset of Instructions

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

→ Performance Issues

i) Longest delay determines clock period

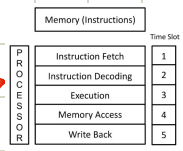
ii) Varying period per instruction isn't feasible (cuz only 1 clock)

iii) Instead of slowing down all instructions, we can optimize common instructions using PIPELINING!

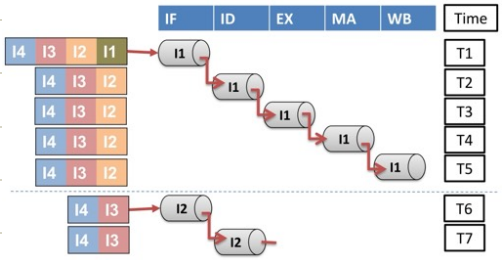
Single Cycle Performance

→ Instructions may involve stages like IF, ID, Ex, MA, WB

→ Every instruction uses 1 clock cycle, so we accomodate accordingly



In case of Non-Pipelining



After 1 instruction is complete and passed through

Q. Assume IF (T_1) = 200n, ID (T_2) = 100n, EX (T_3) = 200n, MA (T_4) = 200n, WB (T_5) = 100n

Find T_{CLK} when we use lw, sw, R-Type & beq. Also find T_{total}

A. For lw ⇒ $T_1 + T_2 + T_3 + T_4 + T_5$ = 800n

sw ⇒ $T_1 + T_2 + T_3 + T_4$ = 700n

R ⇒ $T_1 + T_2 + T_3 + T_5$ = 600n

beq ⇒ $T_1 + T_2 + T_3$ = 500n

T_{CLK} = time taken by slowest instruction = 800n

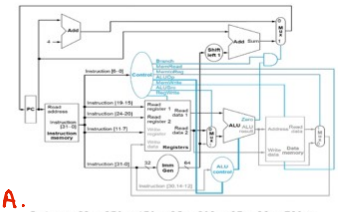
T_{total} = N × T_{CLK} = 4 × 800 = 3200n

Q. Assume that the logic blocks used to implement a processor's Single cycle datapath have the following latencies:

I-Mem / D-Mem	Register File	Mux	ALU	Adder	Single gate	Register Read	Register Setup	Sign extend	Control
250ps	150ps	25ps	200ps	150ps	5ps	30ps	20ps	50ps	50ps

"Register read" is the time needed after the rising clock edge for the new register value to appear on the output. This value applies to the PC only. "Register setup" is the amount of time a register's data input must be stable before the rising edge of the clock. This value applies to both the PC and Register File.

a) What is the latency of an R-type instruction (i.e., how long must the clock period be to ensure that this instruction works correctly)?
b) What is the latency of lw?
c) What is the latency of sw?
d) What is the latency of beq?
e) What is the latency of an arithmetic, logical, or shift I-type (non-load) instruction?
f) What is the minimum clock period for this CPU?



A.

R-type: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

ld: 30 + 250 + 150 + 25 + 200 + 250 + 25 + 20 = 950ps

sd: 30 + 250 + 150 + 200 + 25 + 250 = 905

beq: 30 + 250 + 150 + 25 + 200 + 5 + 25 + 20 = 705

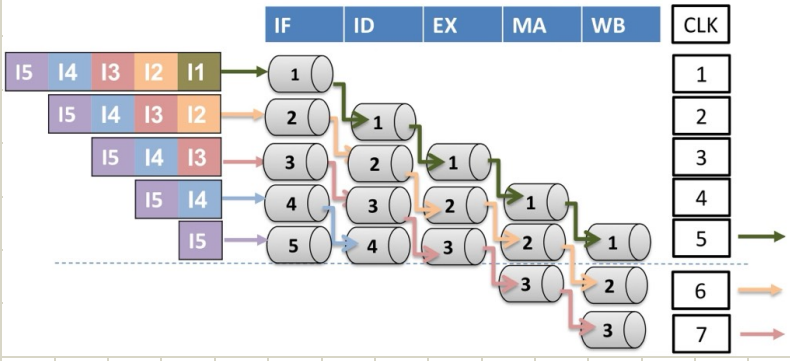
I-type: 30 + 250 + 150 + 25 + 200 + 25 + 20 = 700ps

950ps

Pipelining

→ Now if these stages are independantly performing the task in a sequence, it is called pipelining

→ Each pipeline stage takes single clock cycle & clock cycle should be long enough to accommodate the slowest operation.



⇒ Now T_{CLK} = Time taken by slowest stage

= T_{max} { T_1, T_2, T_3, T_4, T_5 }

From first example, T_{CLK} = 200n

→ Time b/w instructions $_{pipelined}$ = Time b/w instructions $_{non\text{-}pipelined}$ / No. of pipestages → Pipelining speedup

Pipelining Speedup

Under ideal condition for large number of Instructions, **Speed up of pipelining is approximately equal to number of pipeline stages.**

Stages in the pipeline are imperfectly balanced.

Pipelining involves some overhead. Therefore time per instruction in pipeline will exceed the minimum possible and **speed-up will be less than the number of pipeline stages.**

Pipelining improves the performance by increasing the throughput, in contrast to decreasing the execution time of an Instruction.

Control Hazards

→ Happens when CPU doesn't know the correct next instruction to fetch because outcome of branch instruction isn't resolved

→ While processor decides branch outcome, it may fetch wrong instruction into pipeline, which can be flushed (trashed), wasting cycles.

→ Usually branch outcome is only known in EX or MEM stage but by then PC would have fetched next instruction (PC + 4)

```
beq x1, x0, L1     # if (x1 == 0) jump to L1
and x12, x2, x5
or  x13, x6, x2
add x14, x2, x2
L1: lw x4, 100(x7)
```

Here beq enters pipeline, & until EX stage we can't decide if branch is taken, meanwhile if and, or, add are executed and later we know branch is taken then those 3 instructions are flushed, wasting cycles

→ Now instead of just waiting for branch outcome, we can predict the outcome & start fetching along the path.

If correct prediction → No flushing
If wrong prediction → flush & recover
→ This improves average performance if prediction is accurate

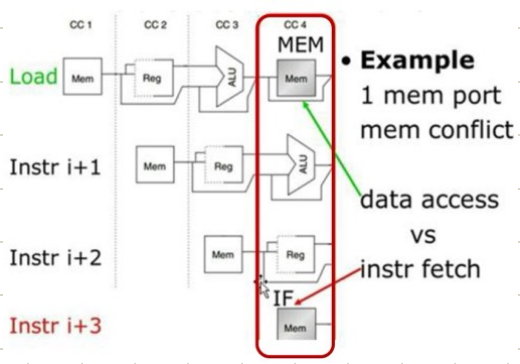
→ One approach is to look up address of instruction to see if conditional branch was taken the last time this instruction was executed & if so, begins fetching from same place ⇒ Dynamic Branch Prediction

Structural Hazard

→ Hardware can't support combination of instructions that we want to execute in same clock cycle

→ It uses one memory for both data & instructions & there will be a conflict

→ To solve this,
Use seperate memories, or memories with multiple ports or pipeline stalls
↳ least efficient



1. Assume a 5-stage pipelined architecture with Von Neuman bus architecture (common memory for code and data). What is the type of hazard it leads to?
   a. Data hazards
   b. Control hazards
   c. ✓ Structural hazards
   d. None
2. In the 5-stage pipelined architecture, the PC gets updated based on the outcome PCSrc control signal. Assume a branch Instruction is in the MEM stage and the outcome of the branch is TRUE. What type of hazard does it lead to & how many clocks would get wasted to resolve the hazard?
   a. Data Hazard, 3clk
   b. Structural Hazard, 3clk
   c. Control Hazard, 1clk
   d. ✓ Control Hazard, 3clk
3. Which are the techniques used to handle the control hazard in a 5-stage pipelined architecture?
   a. Branch predictor
   b. Flushing the instructions that have entered the pipeline after the branch instruction
   c. Flushing the instructions that have entered the pipeline before the branch instruction
   d. ✓ Both a & b
4. Assume the comparator & adder which were present in the execution stage are moved to the decoder stage. This modification leads to the outcome of the branch being known in the decoder stage, what will be the number of clks that will be wasted in flushing the instructions if the outcome of the branch is true?
   a. 3 clks
   b. ✓ 2 clks
   c. 1 clks
   d. 0 clks

1. Consider the following program, which is getting executed on a 5-stage pipelined architecture

Address	Instructions
36:	sub x10, x4, x8
40:	beq x1, x3, 16
44:	and x12, x2, x5
48:	or x13, x2, x6
52:	add x14, x4, x2
56:	sub x15, x6, x7
	...
	...
68	add x5, x5, x7
72:	ld x4, 50(x7)
74	

   a) In the Program given, if the branch on equal is in the MEM access stage, then which instructions are in the WB, EX, ID, and IF stages?
   b) If the outcome of the branch is true, what will the PC value be?
   c) If the outcome of the branch is true which addressing mode is used to calculate PCnew?
   d) If the outcome of the branch is true, which Instructions in the pipeline are to be flushed to handle control hazards, and how many clocks are getting wasted?

a) WB → 36

EX → 44

ID → 48

IF → 52

b) If beq at 40,

Target PC = PC + imm

= 40 + 16

= 56

c) PC Relative Addressing

d) While branch was at MEM, 44, 48 & 52 were fetched

So 3 instructions flushed

3 cycles wasted

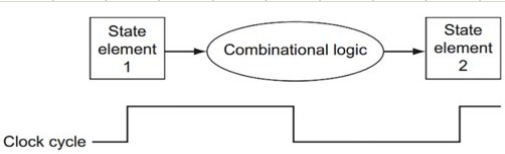
Pipelined Datapath and Control

→ Clocking Methodology

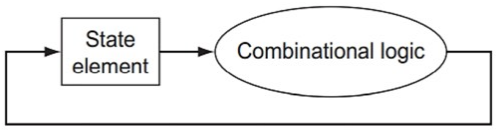
→ It defines when signals when a signal can be read or written

→ It is designed to make hardware predictable

→ On each clock edge (usually rising edge) sequential elements can capture data & combinational logic b/w registers perform computation during the clock cycle



↳ b/w clock edges



↳ Input from state elements, output to state element

→ Datapath

→ Usually instructions & data move from left to right through 5 stages as they get executed

→ But there are 2 exceptions :

i) Write back (Places result back to register file) → Leads to Data Hazards

ii) Selection of next value to PC (Choosing b/w incrementation or branched value) → Leads to Control Hazards

→ Data flowing backwards don't affect current instruction.
only affects later instructions

→ Pipelined & Non-Pipelined Datapaths

→ In non-pipelined datapath, information retained on buses & used by successive stages throughout execution stage which results them not being free to execute next information until current instruction completes execution.

→ Whereas in pipelined datapath, a stage should be made free for next instruction once current instruction passes through it.

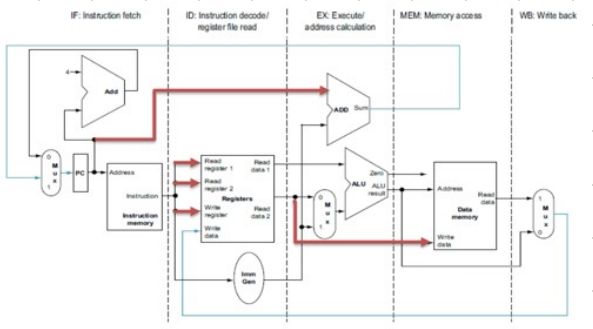
→ All instructions must update some state in the processor

i) The register file - R type, lw

ii) PC - beq, every instruction

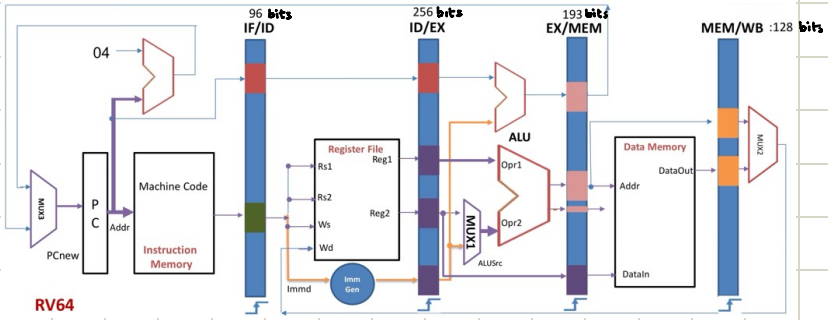
iii) Memory - sw

→ The computer can't know which type of instruction is being fetched, hence, it must prepare for any instruction, passing potentially needed information down the pipeline



→ The PC can be assumed as a pipeline register (Feeds IF stage of pipeline) but not treated as one because it is part of visible architectural state & contents must be stored when exception occurs (Contents of pipeline registers can be discarded / flushed)

→ Registers must be wide enough to store all the data corresponding to lines that go through them



ex: IF/ID register must be 96 bits as it holds 32 bit instruction & 64 bit PC address

Other 3 pipeline registers hold 256, 193, 128 bits respectively

→ Instruction Fetch Stage

→ Address in PC is sent to instruction memory & fetches machine code of instruction to be executed

→ IF/ID pipeline register holds Instruction Fetch code & PC also saved if needed later (like beq)

→ Instruction Decode Stage

→ This stage requires address of rs1, rs2/imm which is supplied by IF/ID register

→ The immediate data is passed through sign extension unit to provide sign extended value

→ ID/Ex pipeline register holds all 3 values (rs1, rs2, sign extended imm) & again transfer everything that may be needed by instruction during later clock cycle (Contents of IF/ID moved to ID/EX registers)

→ Execution Stage

→ This stage takes operand 1 & operand 2 required by ALU to perform operation based on instruction

→ Operand 1 & 2 can be contents of 2 registers (or) 1 register & sign extended immediate present in ID/EX

→ EX/MEM pipeline register holds output of ALU (ALUout & zero) & everything that is needed by an instruction during a later clock cycle

→ Memory Access Stage

→ This stage is only for load & store instructions.

→ Store : takes address & data to be stored in data memory from EX/ID register & completes information

Load : Reading data memory using the address from EX/MEM pipeline register

→ MEM/WB pipeline register used to hold data out from data memory in case of load & ALU out in case of Arithmetic / Logical operation

→ Write Back

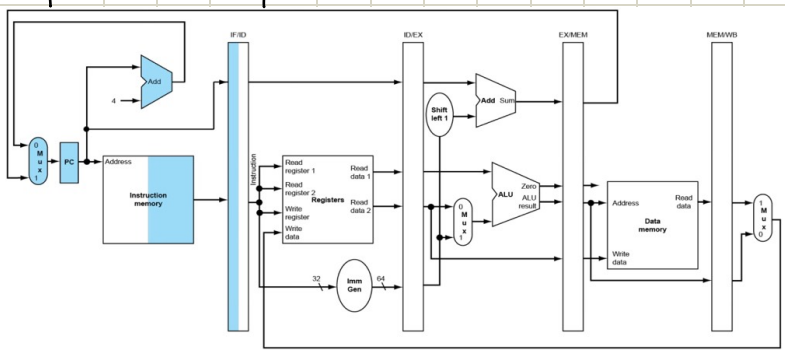
→ This stage reads data from MEM/WB pipeline register & writes it into a register in register file by moving data from right to left

→ The walk-through of the pipelined datapath shows that any information needed in a later pipe stage must be passed to that stage via pipeline register

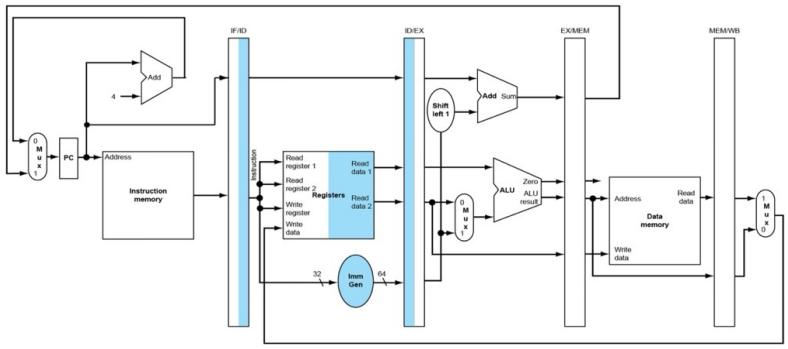
→ Now, for load instruction, it undergoes 5 stages of pipelined execution

Note:
→ Highlight right half for registers (or) memory when they are being read
→ Highlight left half for registers (or) memory when they are being written

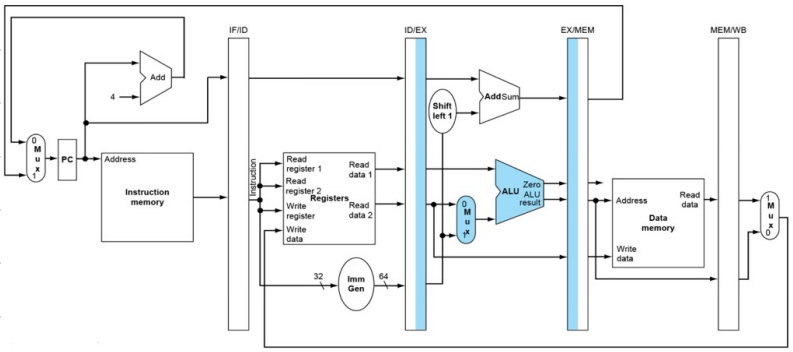
1) IF Stage



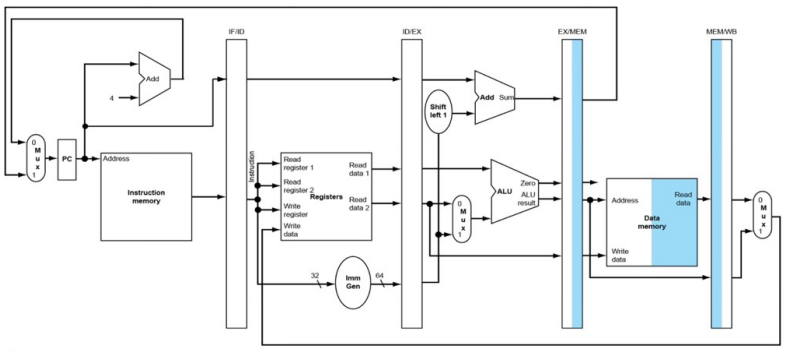
2) ID Stage



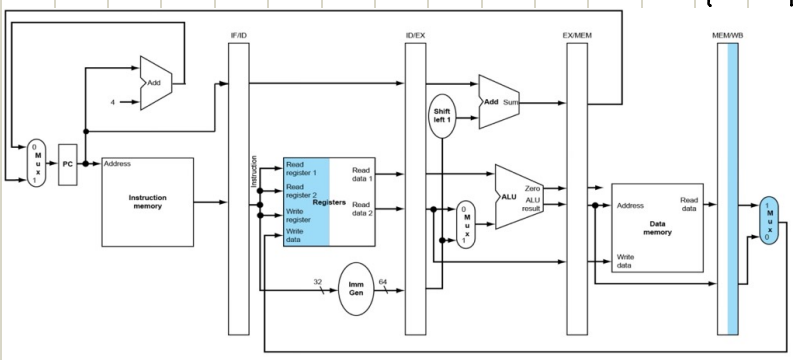
3) EX Stage



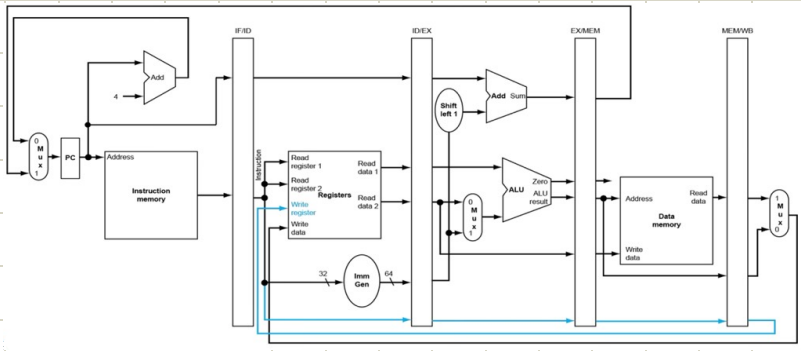
4) MEM Stage



5) WB Stage

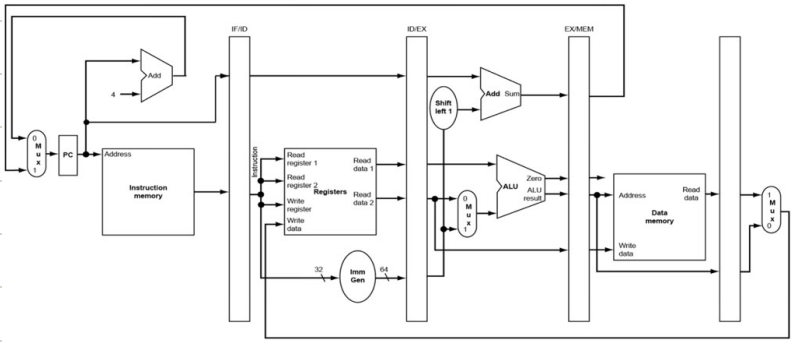


(In load, data is read from MEM/WB pipeline register & written into register file in middle of datapath)



← With write back, this is the corrected Datapath for load

→ Now if we have taken this for Store instruction, we would stop with MEM stage because nothing is changed in MEM/WB pipeline register, so it would look like this, no highlighted parts

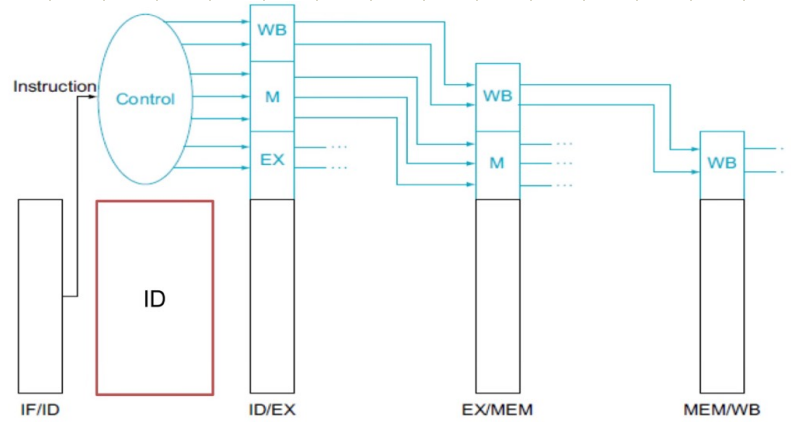


→ Functions of 6 control signals

	Signal	Effect when deasserted	Effect when asserted
1)	RegWrite	–	Register on Write Register input is written with the value on Write data i/p
2)	ALUSrc	2nd ALU operand comes from 2nd register file o/p (Read data 2)	2nd ALU operand is sign extended, 12 bits of the instruction
3)	PCSrc	PC is replaced by o/p of the adder that computes PC+4	PC is replaced by o/p of adder that computes branch target
4)	MemRead	–	Data memory contents designated by the address i/p put on Read data i/p
5)	MemWrite	–	Data memory contents designated by the address i/p replaced by Write data i/p
6)	MemtoReg	Value fed to Register Write data i/p from ALU	Value fed to Register Write data i/p from data memory

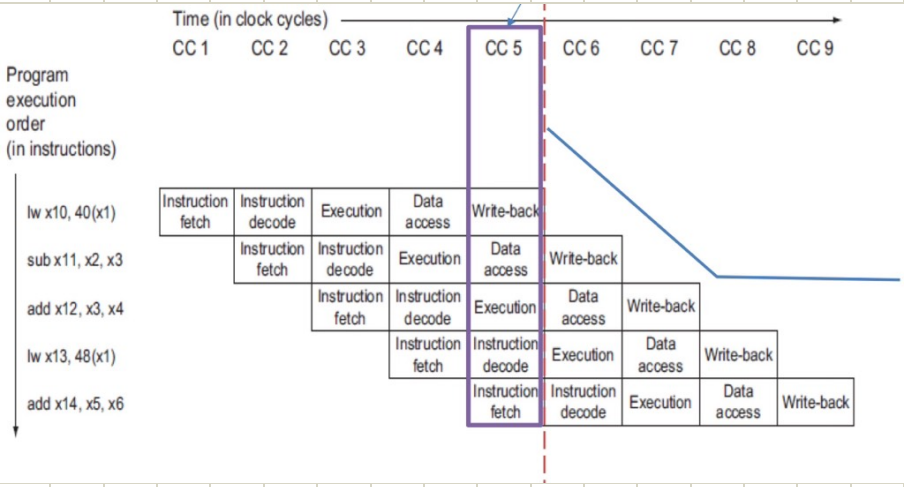
→ Classification of control lines for final 3 stages

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

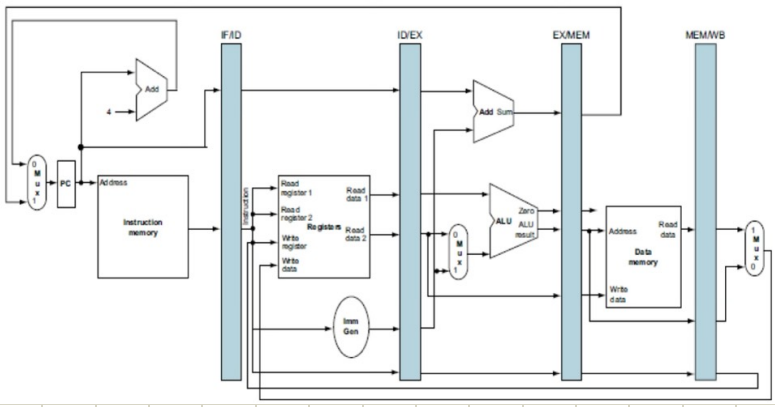


→ Take the following snippet as example

```
lw x10, 40(x1)
sub x11, x2, x3
add x12, x3, x4
lw x13, 48(x1)
add x14, x5, x6
```



→ A single-clock cycle diagram represents vertical slice of one clock cycle through a set of multiple clock cycle diagrams



→ Single-Clock-Cycle diagram corresponding to clock cycle no. 5 of the pipeline