Name: Abdullah Al Mamun
ID: 21301580

Chapter - 03 Overflow and Multiplication

- Operations on integers → Addition, Substraction
  → Multiplication, Division
  → Dealing with overflow.

- Floating point real numbers → Representation
  → Operation.

• Overflow:

Sometimes, while adding or substracting, there may be wrong result. As we know, we have registers to do our arithmatic operation and each register can hold upto 32 bit. So we have 2 binary numbers and there may be cases when we will not be able show the result using 32 bit register. That's when overflow occur.

P.T.O

Example:

let's think of a 4 bit architecture.

```
  0111  (7)
  0101  (8)
  ----------
  1111
```

↳ MSB 1 → so, its sign is negative. But the addition resultant of 2 positive numbers cannot be negative

↳ Wrong Answer.

But if we represented the number in 5 bit → 01111 the result would have been Correct.

- Overflow occurs if result out of range.

Case 01: Adding (+) and (−) opperands,

↳ No Overflow

Case 02: Adding two (+) operands

↳ if result sign is 1 → Overflow

Adding two (−) operands

↳ if result sign is 0 → Overflow.

Overflow when Addition

Range for nbit signed number

-2^{n-1} to $2^{n-1}+1$

- No overflow: when adding one positive and one negative binary number.

  ↳ Because there is no chance of going the number out of range.

Example 01:

Suppose, we are considering a 5 bit signed number

$$-2^{5-1} \text{ to } 2^{5-1}-1$$

$$= (-16 \text{ to } +15)$$

```
  01011 — (11)
  11111 — (-1) → Got by doing 2's compliment.
+
 -------------
  01010 (10)
```

```
  00001
  ↓↓↓↓↓
  11110 → (1's)
     +1
 -------
  11111
```

↳ 2's compliment

↳ Correct Answer

↳ No overflow.

• **Overflow**: when adding two positive binary numbers.

↳ Overflow if sign bit result is 1.

$$
\begin{array}{rl}
 & 01011 \ (11) \\
+ & 01111 \ (15) \\
\hline
 & 11010
\end{array}
$$

→ Sign bit 1, so overflow occured.

2's complement to Decimal Conversion:

$$11010 = -2^4 + 2^3 + 2^1$$

$$= -16 + 8 + 2 = -6$$

Overflow: When adding two negative binary numbers.

↳ Overflow if sign bit of result is 0.

$$
\begin{array}{rl}
 & 10101 \ (-11) \\
 & 10001 \ (-15) \\
(+) & \\
\hline
(1) & 00110 \ (6)
\end{array}
$$

But the result should have been (-26)

→ Sign bit is 0, so overflow occured.

$$1.00110 = -2^5 + 2^2 + 2^1$$

$$= -26.$$

Integer Substraction

- Add negation of second operand.

Example:

```
 7 → 00000000   0000 0111
-6 → 00000000   0000 0110
     ↓↓↓↓↓↓↓↓   ↓
     1111 1111  1111 1001
                        +1
     ----------------------
     11111111   1111 1010
     ----------------------
(+1) 00000000   0000 0001
```

- Overflow if result out of range.

Case 01: Substracting two (+) or two (−) operands
↳ No overflow.

Case 02: Substracting (+) from (−) operand.
↳ Overflow if result sign is 0.

Case 03: Substracting (−) from (+) operand.
↳ Overflow if result sign is 1.

Overflow when substraction

Case 01: When substracting a negative number from a positive number. → Overflow if sign bit of result is 1.

00011 (3)

10001 (-15) → 3-(-15) = 18

So, to avoid any problem, we can just take 2's compliment of (-15) to turn it into a positive number.

01110
+1
01111 → (15)

∴ 00011
01111
10010 → (18)

Case 02: When substracting a positive number from a negative number. → Overflow if the sign bit of result is 0.

-8 11000 -8-(10)

+) -10 10110

~~00110~~

(14) 01110

So if take 2's compliment of the second number it solves it.

Dealing with Overflow

- Some specific language like C ignore overflow. because it uses addu, addui, subu instructions.
  → uses unsigned convention for arithmatic operation.
- On overflow, invoke exception handler.

Steps:

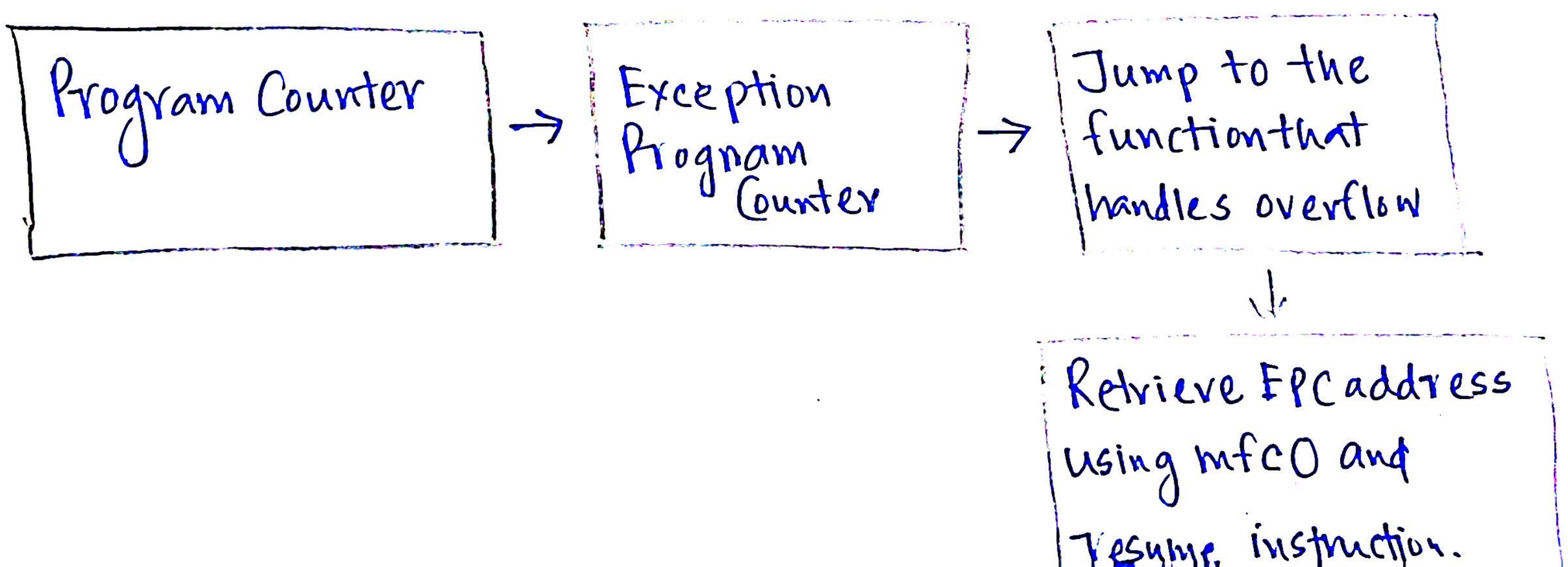
i) So, there is an overflow in an operation and ~~P&~~ compiler was able to detect it. So, currently PC has the address of that instruction.

ii) Then it saves the PC in exception program counter (EPC) register.

iii) Jump to predefined handler address. (Function that handles overflow)

→ iv) Retrieves EPC address using mfc0 (move from co-processor reg) thus return after corrective action.

Flow Chart:



Long Multiplication Approach:

→ lengthy and complex approach

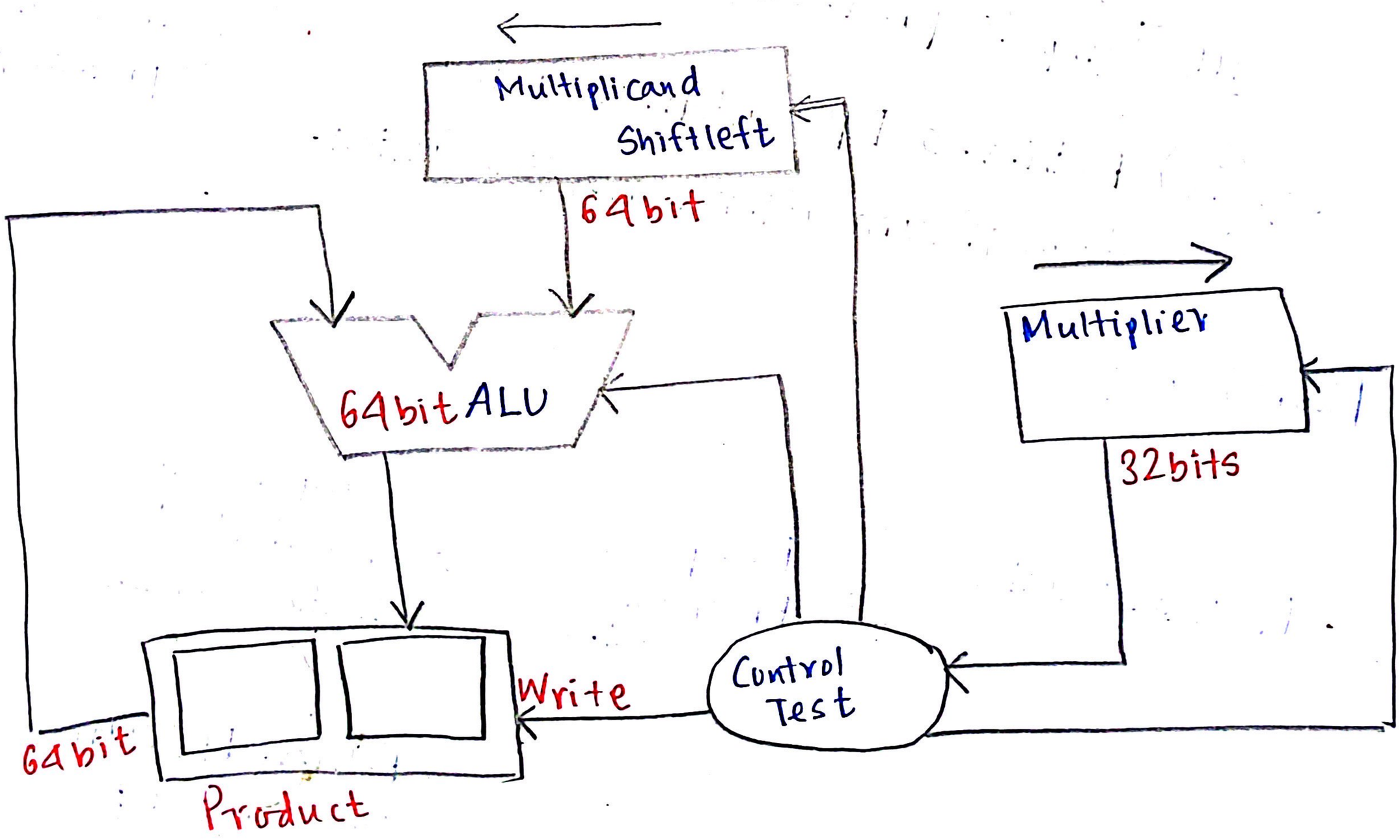
→ We have a circuit for doing multiplication

→ Multiplication has different approach.

→ Primitive approach : long multiplication.

> Highest length of product:
> length of multiplicand +
> length of multiplier

```
multiplicand:  1000
multiplier  : X1001
              ----
              1000
             00000
            0000
           1000
           ---------
           1001000 = 7 bit
```



Note: Here, product register is seen to be a 64 bit register. But in MIPS architecture, we don't have any 64 bit register. So, Product register actually consists of 2 register → one contains 32 bit LSB and other contains 32 bit MSB.

- Multiplicand is also 64 bit. So, it also contains 2 registers but why?

→ Here, in multiplicand's registers, one contains 32 bit word and other contains 32 bit 0. It is 64 bit because there is a connection with ALU here. And for mathematical operation, two things cannot be in different length. ~~So~~, As there is a relationship including ALU ~~using~~ between Multiplicand and product, so if the product is 64 bit and multiplicand is 32 bit. ALU won't be able to differentiate between the ~~sub~~ summision.

Example

For 4 bit system

```
   1000  → Multiplicand → 4 bit
 × 1001  → Multiplier
 ------
   1000
  0000
 0000
1000
 -------
1001000  → length
```

Multiplicand : 00001000 → its in 64 bit

Multiplier : 1001

Product :

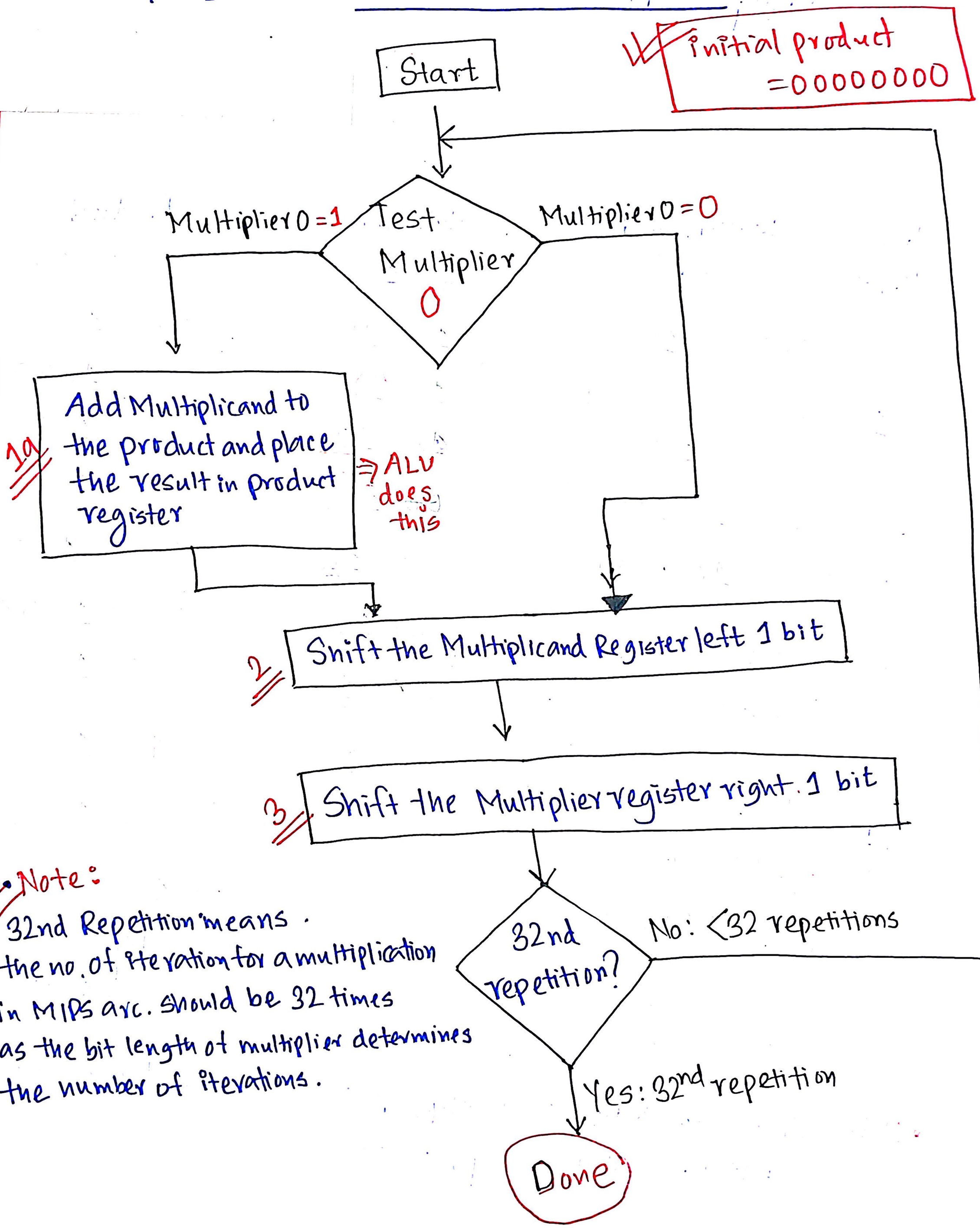
initial → 00000000

final → 1001000

Multiplicand always left 1 bit

Multiplier always right 1 bit

Multiplication Approach



Note:

32nd Repetition means the no. of iteration for a multiplication in MIPS arc. should be 32 times as the bit length of multiplier determines the number of iterations.

Example (Two Positive Value)

Multiplicand → 1000 (8) Multiplier → 1001 (9)

* Number of iterations = number of bits in multiplier

No. of Iterations	Multiplicand 00001000	Multiplier 1001	Product 00000000 (initial)
1	1a 00001000	1001	00001000 (New)
	2 00010000	1001	00001000
	3 00010000	0010	00001000
2	2 00100000	0100	00001000
	3 00100000	0010	00001000
3	2 01000000	0010	00001000
	3 01000000	0001	00001000
4	1a 01000000	0001	01001000 (New)
	2 10000000	0001	01001000
	3 10000000	0000	01001000 Final Result

• **Note:** When we make the multiplier 0000 after doing right shifts, we will get the product.

Flow Chart:

1. Test Multiplier 0 : Test Multiplier 0 has the last digit of multiplier.
   register.

Note: When we will be given one positive and one negative number, we will treat both of the numbers as positive and follow the flow chart and find out the product. But, as the result should be negative → we will take 2's Compliment as output of the product.

P.T.O

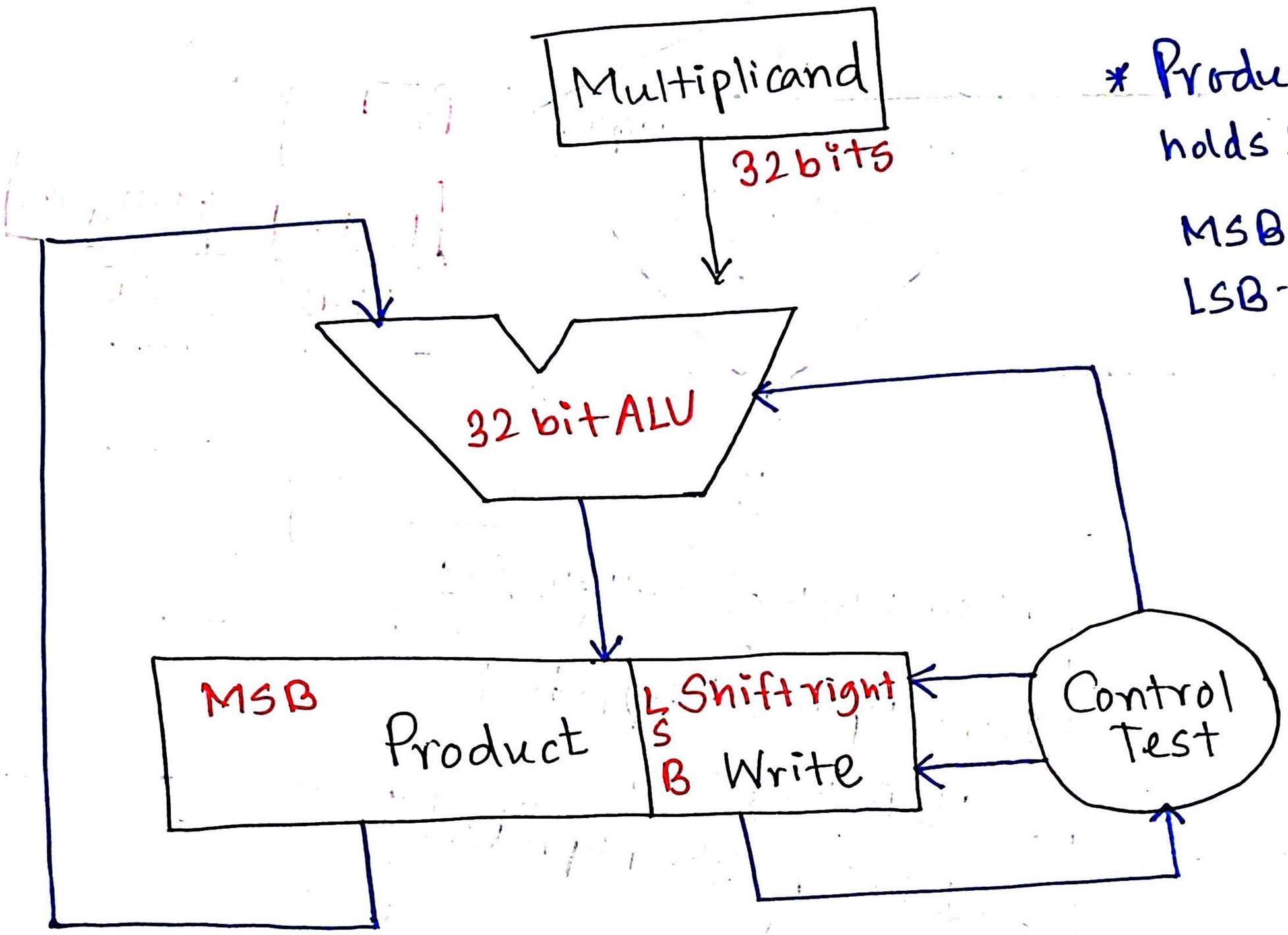
Example 02

Multiplicand = 01001 (& 9) Number of iterations required = 5

Multiplier = 01010 (10)

No. of Iteration	Multiplicand	Multiplier	Product
	00000001001	01010	00000000000
1	2 00000010010	01010	00000000000
	00000010010	3 00101	00000000000
2	2 00001000100	3 00010	1a 00000010010
3	2 00010010000	3 00001	00000010010
4	2 00100100000	3 00000	1a 00001011010
5	0100100000	00000	00001011010

• Optimized Multiplier for 32 bits



* Product register holds 2 register.
  MSB - 32 bit
  LSB - 32 bit

* Multiplier is in 32 bit of LSB 32 bit of Product Register.
* Here, ~~we~~ 32 bit ALU will perform addition with only 32 bit MSB of product.

Multiplicand : 32 bit
Product : MSB + LSB = 64 bit
↳ Multiplier 32 bit
ALU : 32 bit

Optimized Multiplier Approach

if

iteration <= multiplier bit length

Product

Product	Multiplier
MSB	LSB

if last bit of multiplier = 1

1. multiplicand + product MSB half bits
2. 1 bit right shift product

if last bit of multiplier = 0

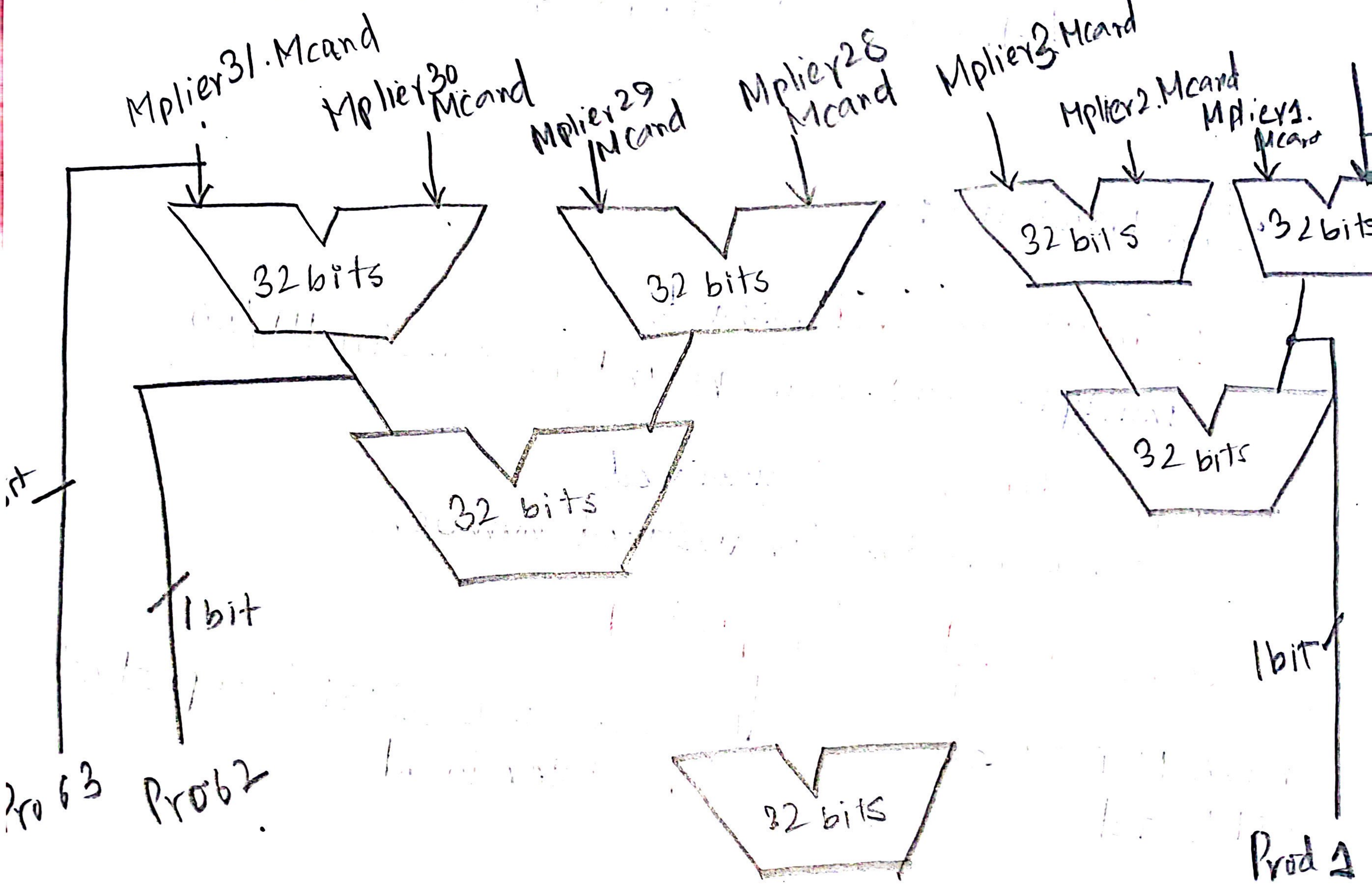
1 bit right shift product

	Multiplicand	Product
	1000	0000 1001 Multiplier bits
1	1000	10001001 01000100
2	1000	00100010
3	1000	00010001
4	1000	10010001
	1000	01001000 Final Answer

	Multiplicand 01001	Product 00000 $\underbrace{001010}_{\text{Multiplier}}$
1	01001	0000000101
2	01001	0100100101 0010010010
3	01001	0001001001
4	01001	0101101001 0010110100
5	01001	0001011010 Final Result

Faster Multiplier

- Uses multiple adders
- Cost/Performance trade off.
- Can be pipelined
- Several multiplication performed in parallel.



Registers for MIPS Multiplication

- Two 32 bit registers for product.
  - HI : most - significant 32 bits MSB
  - LO : least - significant 32 bits LSB

• Instructions :-

1. mult rs, rt (signed)
   multiplies numbers in rs and rt stores in HI/LO

2. multu rs, rt (unsigned)
   Same work just for unsigned numbers

3. mfhi rd / mflo rd
   - mfhi rd → moves MSB 32 bit to rd
   - mflo rd → moves LSB 32bit from product register to rd

4. mul rd, rs, rt
   This is a pseudo instruction. It consists two instruction.

1. mult rs, rt
2. mflo rd

Floating Point Representation

- Representation for non-integral numbers.
  Including very small and large numbers.

• floating point Standard

- Defined by IEEE Standard 754-1985.
- It has two representation.
  - → Single Precision 32 bit
  - → Double Precision 64 bit

$\pm 1.xxxxx \times 2^{y}$

• Normalization

Two Conditions:

1. Only One and Non Zero number before decimal point.
2. Only One and Non-Zero number before binary point

Example

5.64×10^{33} → Normalized

$109.64 \times 10^{33} = \underbrace{1.0964 \times 10^{35}}_{\text{Normalized}}$

↓ Not Normalized

$1011.1101 \times 2^{33} \longrightarrow 1.01111101 \times 2^{33+3}$

$= 1.01111101 \times 2^{36}$

Note: If we move the decimal or binary point to left, we will add the number of bits to the exponent. and if we move to the right, we will Substract that ~~b~~ from the exponent.

IEEE Floating Point Convension :-

Single Precision : 32 bit

Sign Bit	Exponent	Fraction/Mantissa
1 bit	8 bit	23 bit

0 = positive
1 = negative

- **Exponent:**

Exponent will be kept as unsigned 8 bit binary number.

$\therefore$ 8 bit unsigned binary range = 0 to $2^8 - 1$ = 0 to 255

But 00000000 and 11111111 is reserved. So

range of biased exponent = 1 to 254.

$$\boxed{\text{if biased exponent} = n \text{ bits}, \text{ bias} = 2^{n-1} - 1}$$

- For Single Precision,

Biased Exponent =
Actual exponent of binary number
+ Bias(127)

Range for Exponent $= 2^{-126} - 2^{127}$

Example

1.11101×2^{35}

Biased exponent $= 35 + 127 = 162$

1.11101×2^{-8}

Biased exponent $= -8 + 127 = 119$

- Decimal to Floating Point Conversion Flow Chart

1. Decimal to Binary

↓

2. Normalize Binary Number

↓

3. Find out Biased Exponent

↓

4. Find out sign bit & Fraction

↓

5. Write the whole Convention

Result always in Hex.

Example

Convert 50.6749 to 32 bit IEEE-754 Floating Point Representation.

Ans:

Step 01: Decimal $\longrightarrow$ Binary

$(50.6749)_{10} = (110010)_2$

$(50)_{10} \longrightarrow (110010)_2$

$(.6749)_{10} = (1010110011)_2$

$(50.6749)_{10} = (110010.1010110011)_2$

Step 02: Normalize the Binary Number

$(110010.1010110011) \times 2^0$

Normalize

$(1.10010101011 0011) \times 2^5$

.6749
× 2
MSB 1.3498
× 2
0.6996
× 2
1.3992
× 2
0.7984
× 2
1.5968
× 2
1.1936
× 2
0.3872
× 2
0.7744
× 2
1.5488
× 2
LSB 1.0976

Step 03: Find Out the Biased Exponent.

Exponent = 5

$\therefore$ Biased Exponent $= 5 + 127 = 132.$

$= (10000100)_2$

Step 04: Find out Sign Bit and Fraction.

Sign Bit = 0

Fraction $= (10010101011 0011)_2 00000000$

Step 05: Representation

0	10000100	10010101011 0011 00000000

$= 0x424AB300$

Double Precision (64 bit)

Sign Bit	Exponent (Biased)	Fraction/ Mantissa
1 bit	11 bit	52 bit

- **Sign Bit :** 0 → Positive
  1 → Negative.

- **Exponent :**
  11 bit unsigned binary Range = 0 to $2^{11}-1$
  = 0 to 2047

  Exponents 00000000 to 11111111 reserved,
  Range for Biased Exponent = 1 - 2046

For Exponent being 11 bit, Bias = $2^{(11-1)}-1 = 1023$

For Double Precision,

Biased Exponent = Actual Exponent of the Binary Number + 1023

Example:

Convert -0.232 to 12 bit, exponent is 4 bit.

Soln:

Given number -0.232.

Binary Number

$(0.232)_{10} = (0.00111011101)$

$= 1.1101101 \times 2^{-3}$

Exponent = 4 bit

Sign bit = 1 bit

Fraction = (12 - 5) = 7 bit

$\therefore$ Binary number $= 1.1101101 \times 2^{-3}$

Biased Exponent range $2^{4-1} - 1 = 7$

$\therefore$ Exponent $= -3 + 7 = 4 = 0100$

1	0100	1101101
1	4	7

$.232 \times 2$

0.464×2

0.928×2

1.856×2

1.712×2

1.424×2

0.848×2

1.696×2

1.392×2

0.784×2

1.568

Example :

Convert 1.232×10^2 to 15 bit, Where Exponent is 5

Sign Bit = 1 bit

Exponent = 5 bit

Fraction = $15 - (5+1) = 9$ bit

Given number $1.232 \times 10^2 = 123.2 \times 10$

$(123)_{10} = (1111011)_2$

$(.2)_{10} = (.000110)_2$

$(123.2)_{10} = (1111011.000110)_2$

$= 1.1110110001 10 \times 2^6$

$$\begin{array}{r} 0.2 \\ \times 2 \\ \hline 0.4 \\ \times 2 \\ \hline 0.8 \\ \times 2 \\ \hline 1.6 \\ \times 2 \\ \hline 1.2 \\ \times 2 \\ \hline 0.4 \end{array}$$

Exponent = 6

Biased Exponent $= 6 + (2^{5-1} - 1) = (21)_{10}$

$= (10101)_2$

Fraction = 111011000

∴ Representation =

0	10101	111011000

= 2BD8

Floating Point to Decimal Conversion.

0xF2400120

32 bit

Sign bit → 1

Exponent → 8

Fraction → 23

Step 01: Hex to Binary.

$$\underbrace{1}_{\text{Sign } (-)}\ \underbrace{1110010}_{\text{Exponent}}\ \underbrace{0100\ 0000\ 0000\ 0001\ 0010\ 0000}_{\text{Fraction}}$$

Biased Exponent $= (1110010)_2 = 228$

Exponent $= 228 - 127 = 101$

Fraction $= 10000000000001001000000$

$= 2^{-1} + 2^{-15} + 2^{-18}$

$= 50.50003433 23$

Decimal Value $= (-1)^{\text{sign bit}} \times (1 + \text{Fraction}) \times 2^{\text{Exponent}}$

$= -1.50003433 23 \times 2^{101}$

$= -3.8830388 43 \times 10^{30}$

Upto 5 decimal point rounding $= -3.80304 \times 10^{30}$

without rounding $= -3.80303 \times 10^{30}$

Floating Point to Decimal Conversion

Step 01: Figure out sign bit, Exponent and Fraction bits

Step 02: Hexadecimal to Binary conversion

Step 03: Set the binary number into format.

Step 04: Find out exponent and fraction.

$$\text{Exponent} = \text{Biased exponent} - (2^{n-1} - 1)$$

Step 05: Find decimal Value.

$$\text{Decimal Value} = (-1)^{\text{signbit}} \times (1 + \text{Fraction}) \times 2^{\text{exponent}}$$

** Round up result if necessary.

Floating point Addition and Substraction

Rule: Match the lower exponent with higher exponent.

- $35.23142 + 0.00053$

$X = 35.23142$

Binary 100010.00111101111

Normalized $= 1.00010001110111 \times 2^5$

$Y = 0.00053$

$= 0.00000000001000101 1$

$= 1.00010111 \times 2^{-11}$

$= 0.00000000000000000 10001011 \times 2^5$

$X + Y = 1.00010001110111 \times 2^5 +$

$0.00000000000000001000101 1 \times 2^5$

$= 1.00010001110111110001011 \times 2^5$

$= 100011.00111101111100010 11$

$= 35.2342224121$

Subtraction:

35.23142 − 0.00053

$= (1.00011001110 1.111 \times (2^5)) - (0.00000000000000 10001011 \times 2^5)$

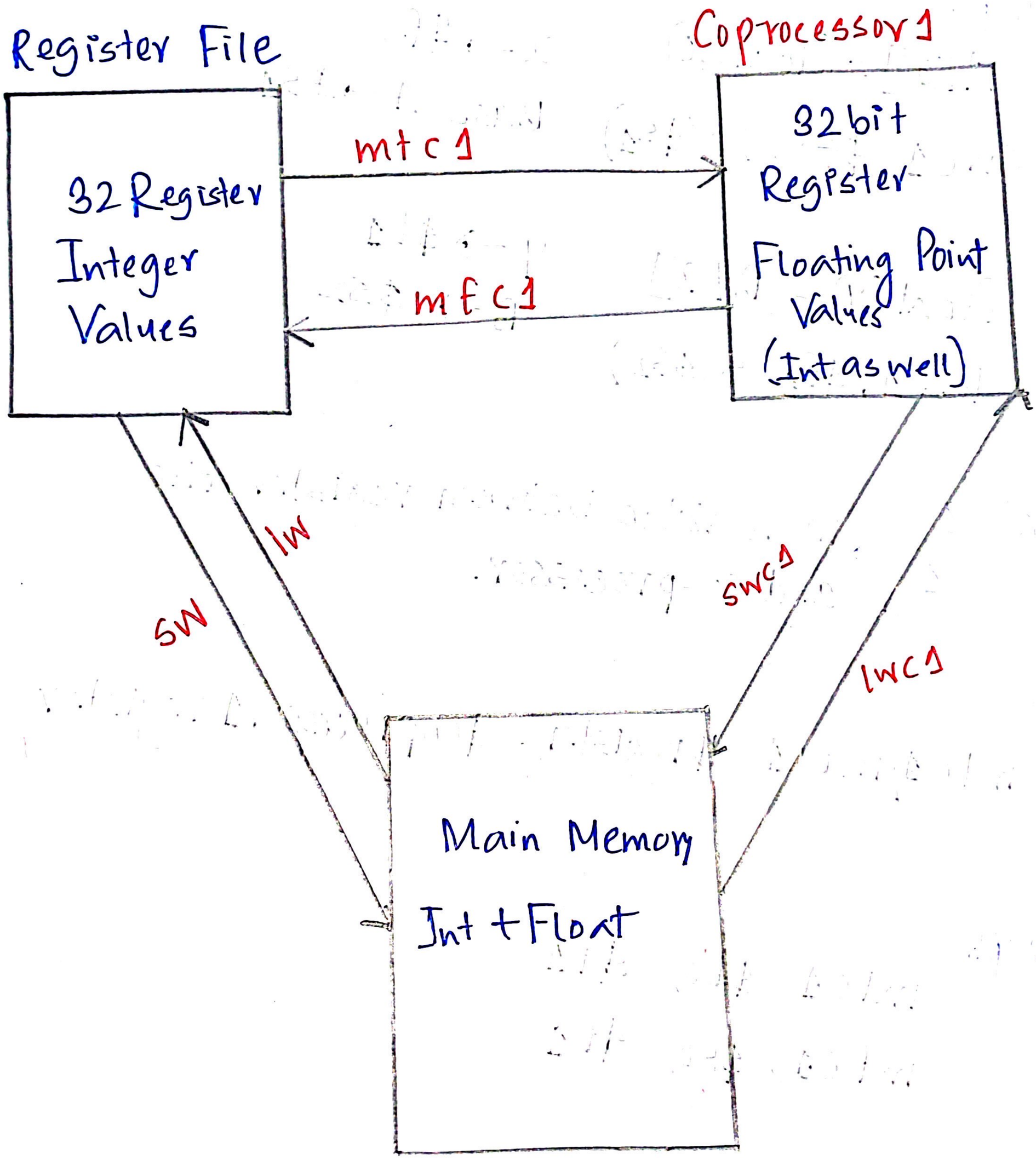
Multiplication:

Floating Point Registers

Coprocessor register $f0 - $f31

swc1 → storeword c1

lwc1 →



* lwc1
* swc1

→ used to ~~int~~ create interaction between main memory and coprocessor 1

Example

A[5] = float(x) x → $f0

swc1, $f0, 20($s1) Base A → $s1

float(y) = B[2] y → $f1

lwc1, $f1, 8($s2) B → $s2

* mtc1
* mfc1

→ move value between register file and co-processor.

Convension:

mtc1/mfc1 $register, $coprocessor1 register.

Example

mtc1 $t0, $f1

mfc1, $s0, $f1

MIPS Instruction for floating Point Register

** For Single Precision

add.s $f0, $f1, $f2 $f0 = $f1 + $f2

sub.s $f0, $f1, $f2 $f0 = $f1 - $f2

lwc1 $f0, 0($s1)

swc1 $f0, 0($s1)

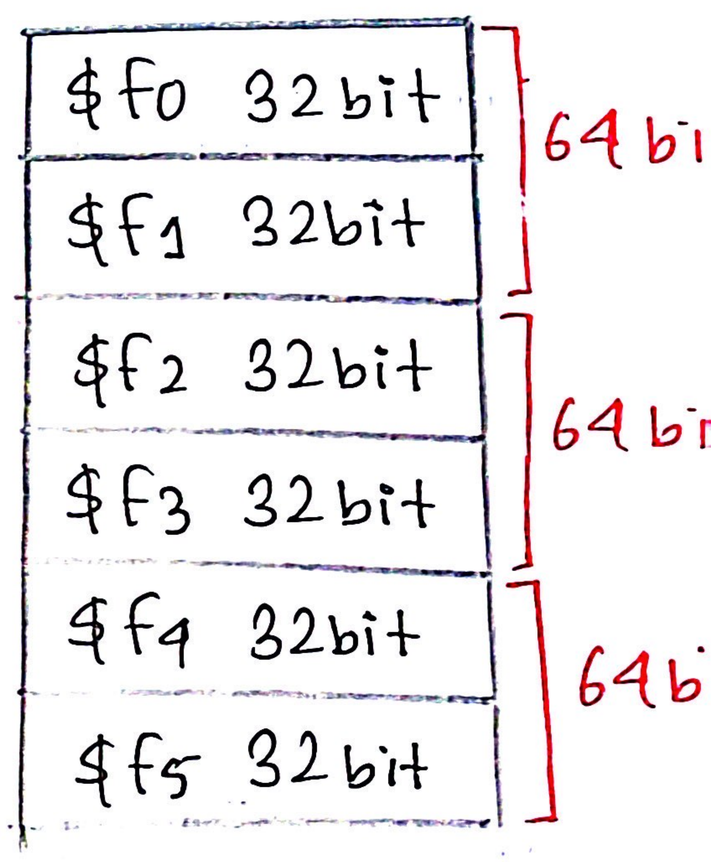
** For Double Precision

add.d $f0, $f2, $f4

sub.d $f0, $f2, $f4

ldc1 $f0, 0($s1)

swc1 $f2, 0($s1)



** To store or have 64 bit data, we need to use two consecutive registers.

$f0, $f1 = 64 bit

** All About Float

Move Value from Register

mtc1 $register $coprocessor1 register — move value from register file to coprocessor 1

mfc1 $register $coprocessor1 register — move value from coprocessor 1 to register 1.

Convert Float to Integer (32 bit)

cvt.w.s $cp1 $cp2 → Occurs in Coprocessor 1.

$cp1 → destination, $cp2 → source

Convert Integer to Float (32bit Single Precision)

cvt.s.w $cp1 $cp2 → Convert value of $cp2 from integer to float, stores in $ $cp1

cvt.s.d $cp1 $cp2 → Convert value of $cp2 ~~into~~ from double precision to single

cvt.d.s $cp1 $cp2 → Convert value from single to double precision.

Example

float(x) = int(y) + float(z)

Int(y) = float(x) - (float)z

x → $f1
y → $s1
z → $f2

mtc1 $s1, $f3 → Moving value as integer from regfile to coprocessor

cvt.s.w $f3, $f3 → Convert Value from int to float

add.s $f1, $f3, $f2

sub.s $f4, $f1, $f2

cvt.w.s $f4, $f4 → Convert to int from float

mfc1 $s1, $f4

* When a value is moved from integer to float register, it is moved as integer.
* We have to convert it to float (~~swc1~~) (cvt s.w) to do float op.

Example

$X = A[4] + B[2] + (B[3]) - 5X)$ Array A stores floating point Values

Base A → $s0

B stores Integer

Base B → $s1

X → $s2

```
lwc1 $f1, 16($s0)
lwc1 $f2, 8($s1)  → integer load
cvt.s.w $f2, $f2
add.s $f1, $f1, $f2   A[4] + B[2]

lw $t0, 12$s1  — B[3]
sll $t1, $s2, 2
add $t1, $t1, $s2   5x
sub $t1, $t0, $t1
mtc1 $t1, $f2  → moving as integer
cvt.w.s $f2, $f2

add.s $f3, $f1, $f2
cvt.w.s $f3, $f3
mfc1 $s2, $f3
```

Notes:

1) When we need to do float operation, but given a integer value, then
first move the integer value to cp1 register.
then convertit into float.

Example

$A = D[6] - 22 + float(C)$

$D[A] = 7 * A - float(B) + (float)\,C$

A → $s0
B → $f1
C → $f2
D Base → $s1

```
lw $t0, 24($s1)   - D[6]
addi $t0, $t0, -22   D[6] - 22
mtc1 $t0, $f3
cvt.s.w $f3, $f3
add.s $f4, $f3, $f2
cvt.w.s $f4, $f4
mfc1 $s0, $f4
sll $t0, $s0, 3
sub $t0, $t0, $s0
```

Chapter 04

Datapath Introduction

- **CPU performance Factors**

  1) Instruction Count → ISA and Compiler.
  2) CPI and Cycle Time → CPU Hardware.

- **MIPS Implementation:**

  1) A Simplified Version → Single Cycle Data Path (slow)
  2) Pipelined Version → Optimized.

Memory Reference : lw, sw
Arithmatic / Logical : add, sub, and, or, slt
Control Transfer : beq, j

These simple subsets show a lot of aspects.

P.T.O

• Instruction Execution

3 steps.

Step 01: Fetch Instruction. PC → Instruction Memory.

Step 02: Read Registers. Register number → register file.

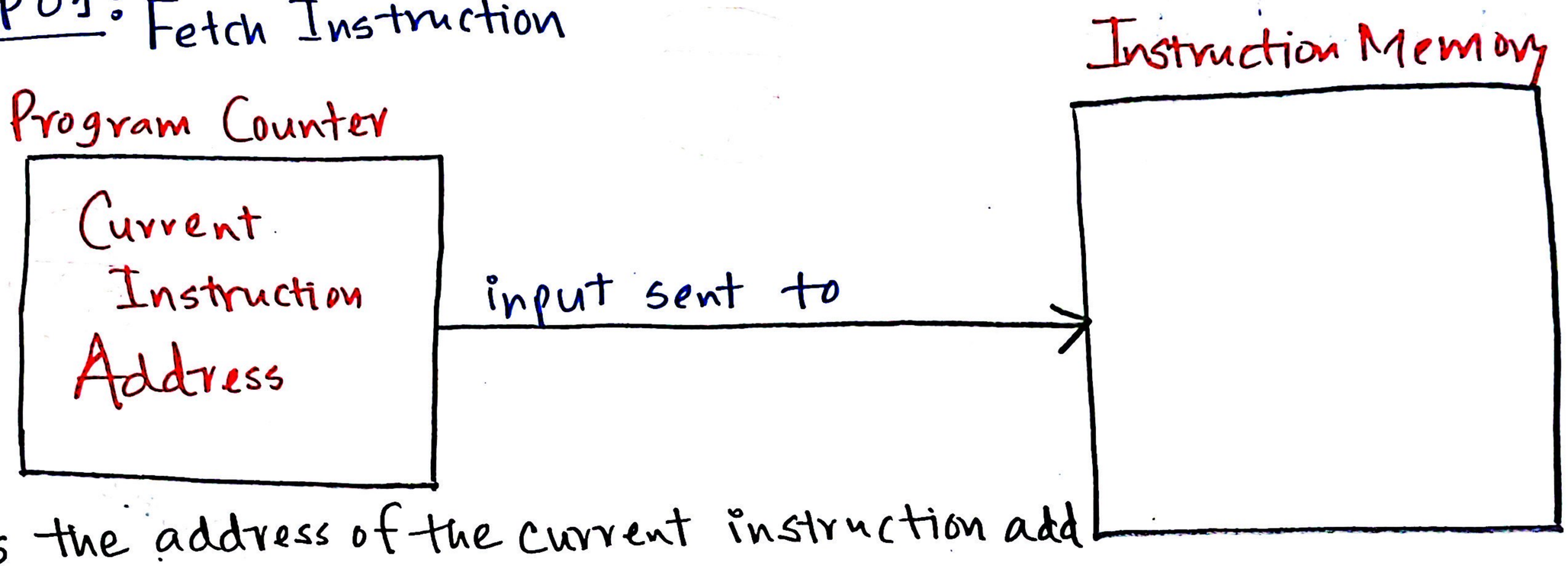
Step 01 and step 02 are identical in every instruction.

Step 03: This step depends on instruction class.

- → Arithmatic & logical operations execution.
- → Memory Address Calculation for load/store.
- → Branch for Comparison.

~~Step 04~~: Access data memory for load/store.

Step 01: Fetch Instruction



As the address of the current instruction add is sent to Instruction memory, the instruction gets activated and gets ready for execution.
So, when the instruction gets executed, it adds +4 to the ~~sends the dat~~ previous instruction and gets the address saved of the new instruction.

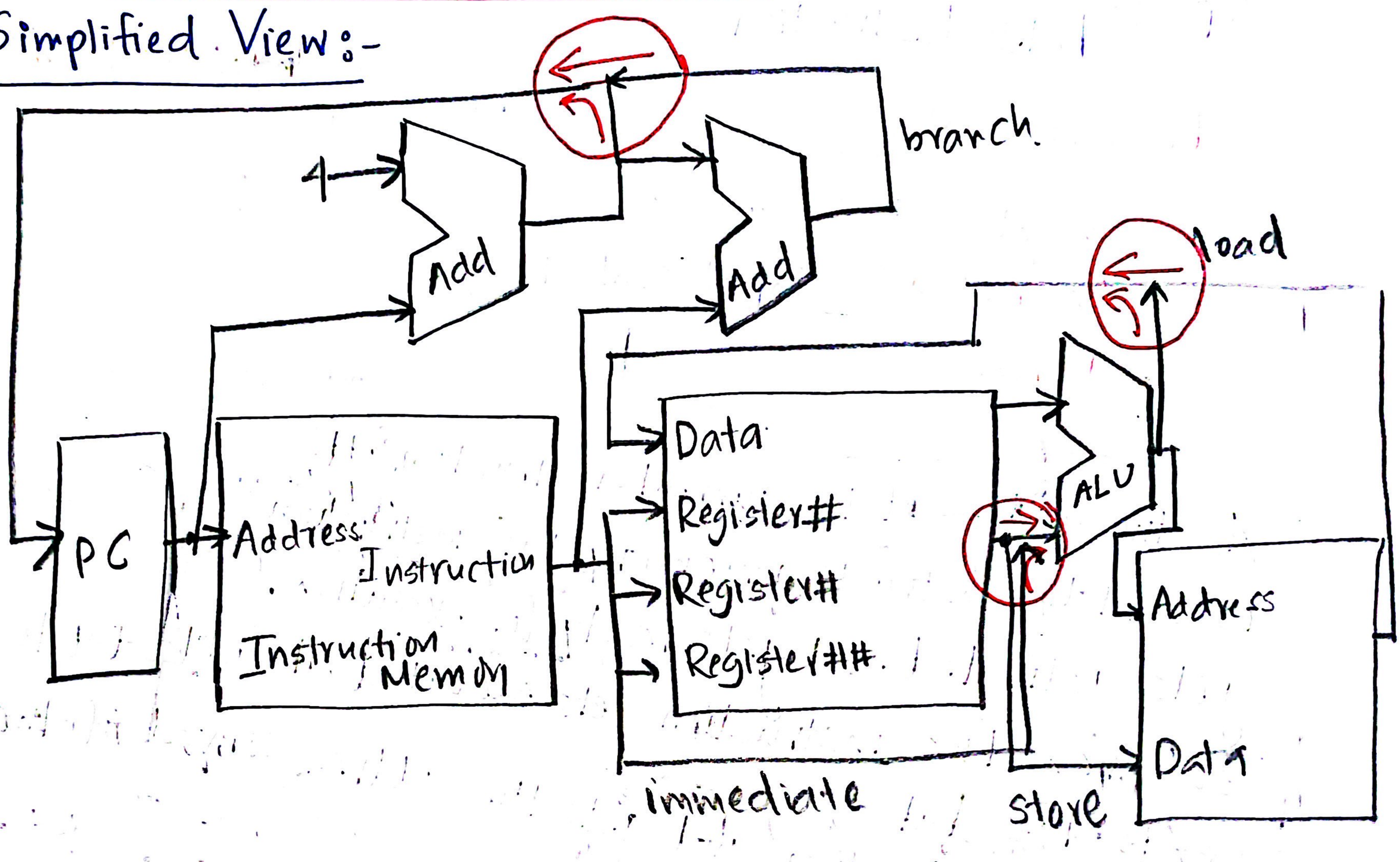
Decode Instruction : add $8, $17, $18

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

Instruction decoding is done by Control Unit.

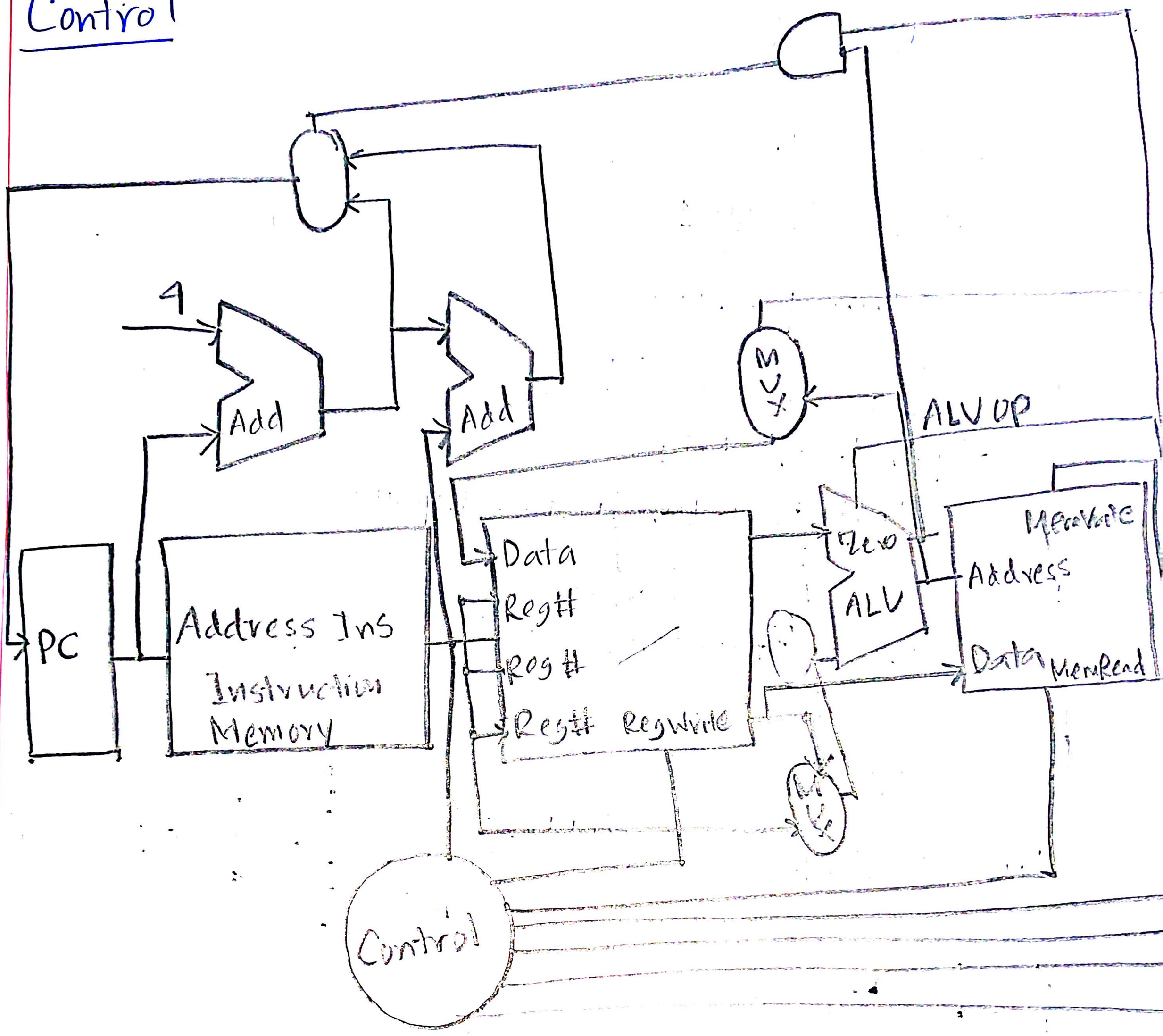
rd (5 bit) → decoder → selects write register.
rs (5 bit) → MUX → reads register
rt (5 bit) → MUX → reads register.

Simplified View:-



here, for the marked red ways, we have to add mux to direct the way for the datapath.

Control



RegWrite = 1 → Write ~~Store~~ in Register

MemWrite - 1 (SW)

Mem Read - 1 (Load)

Logic Basic Design

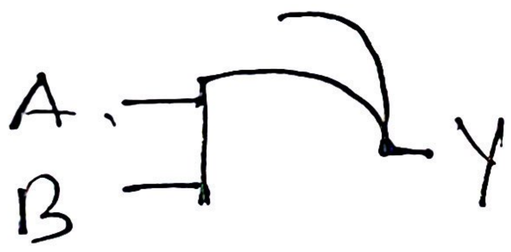
Low Voltage = 0
High Voltage = 1
One wire per bit.

* Combinational Element.
  → Operate on data.
  → Output is a function of input.

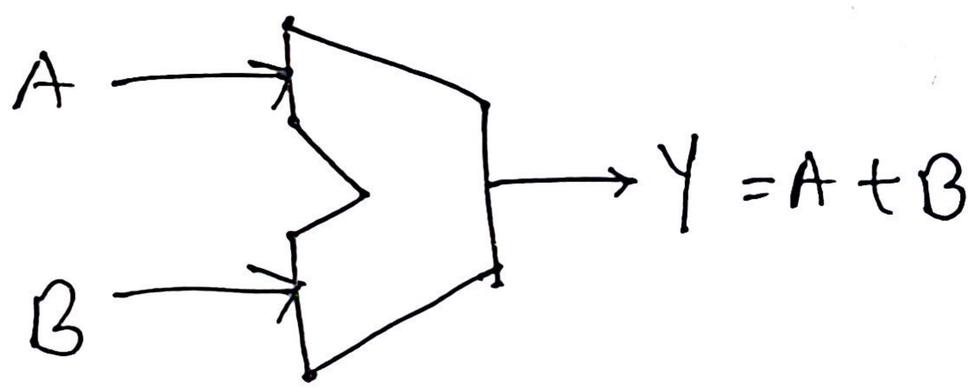
* State elements.
  → Store information.

Combinational Elements:

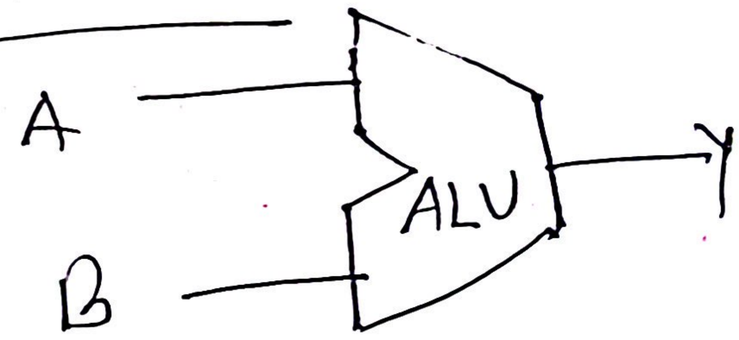
AND: $Y = A \cdot B$



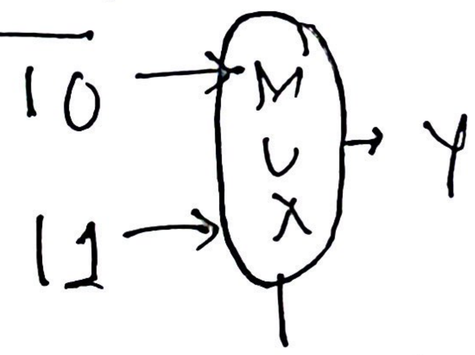
Adder:



ALU:



Multiplexer:



$Y = S ? I1 : I0$

Sequential Element

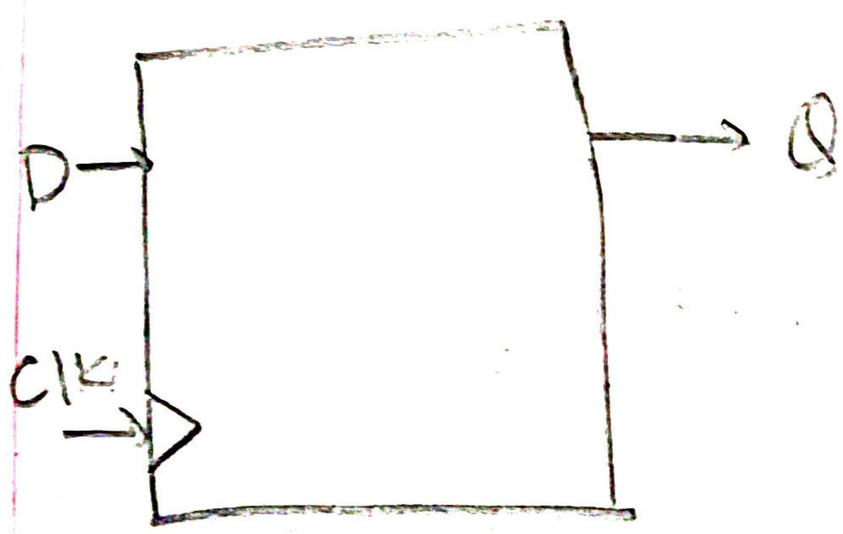
Register: Stores data in a Circuit.

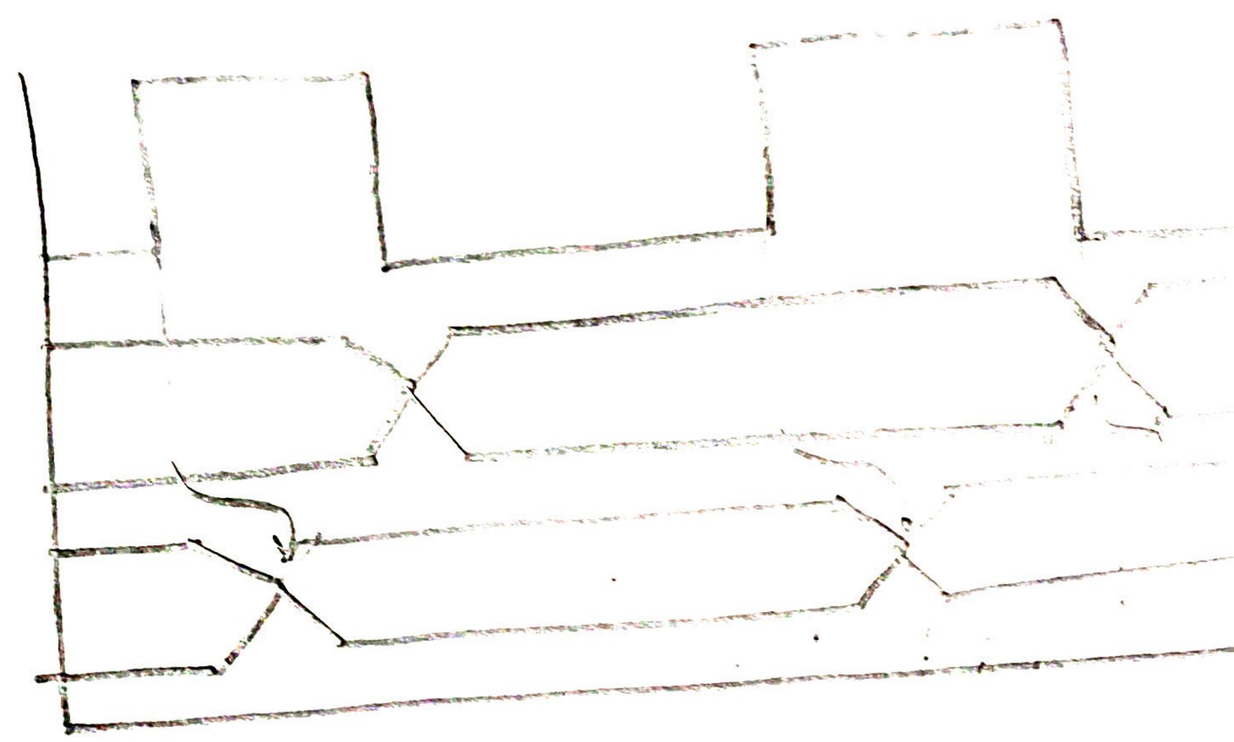
→ uses a clock signal to determine when to update the stored value.

→ Edge triggered.

Rising edge: update when clk 0 → 1

Falling edge: update when clk 1 → 0





Register with write Control:

Only updates on clock edge when Write Control Input is 1.

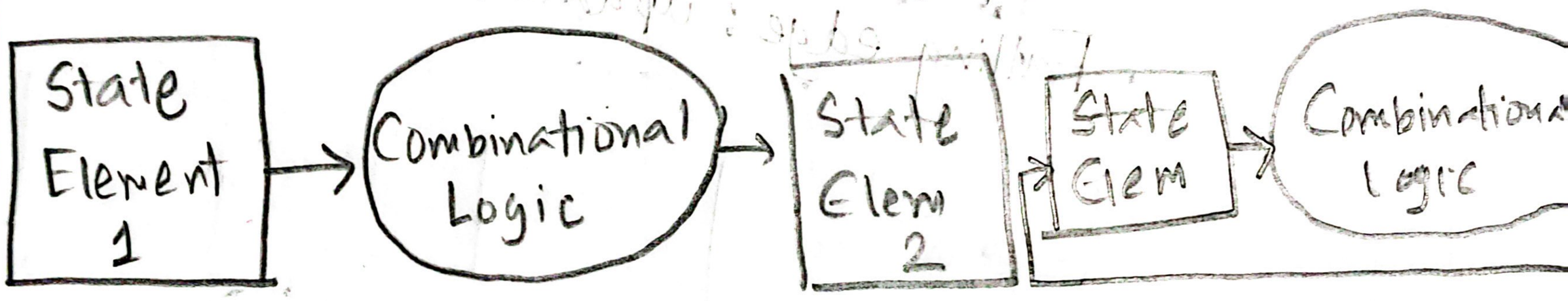
Used when stored Value is required later.

Clock Methodology:

Edge triggered clocking

↳ Combinational logic transforms data during clock cycles between clock edges.

** Longest delay determines clock period.



Total time = 10 + 10 + 10 + 10 = 40 ns

add - 5ns

mul - 8ns

sub - 5ns

lw - 10ns → longest delay = clock period = 10

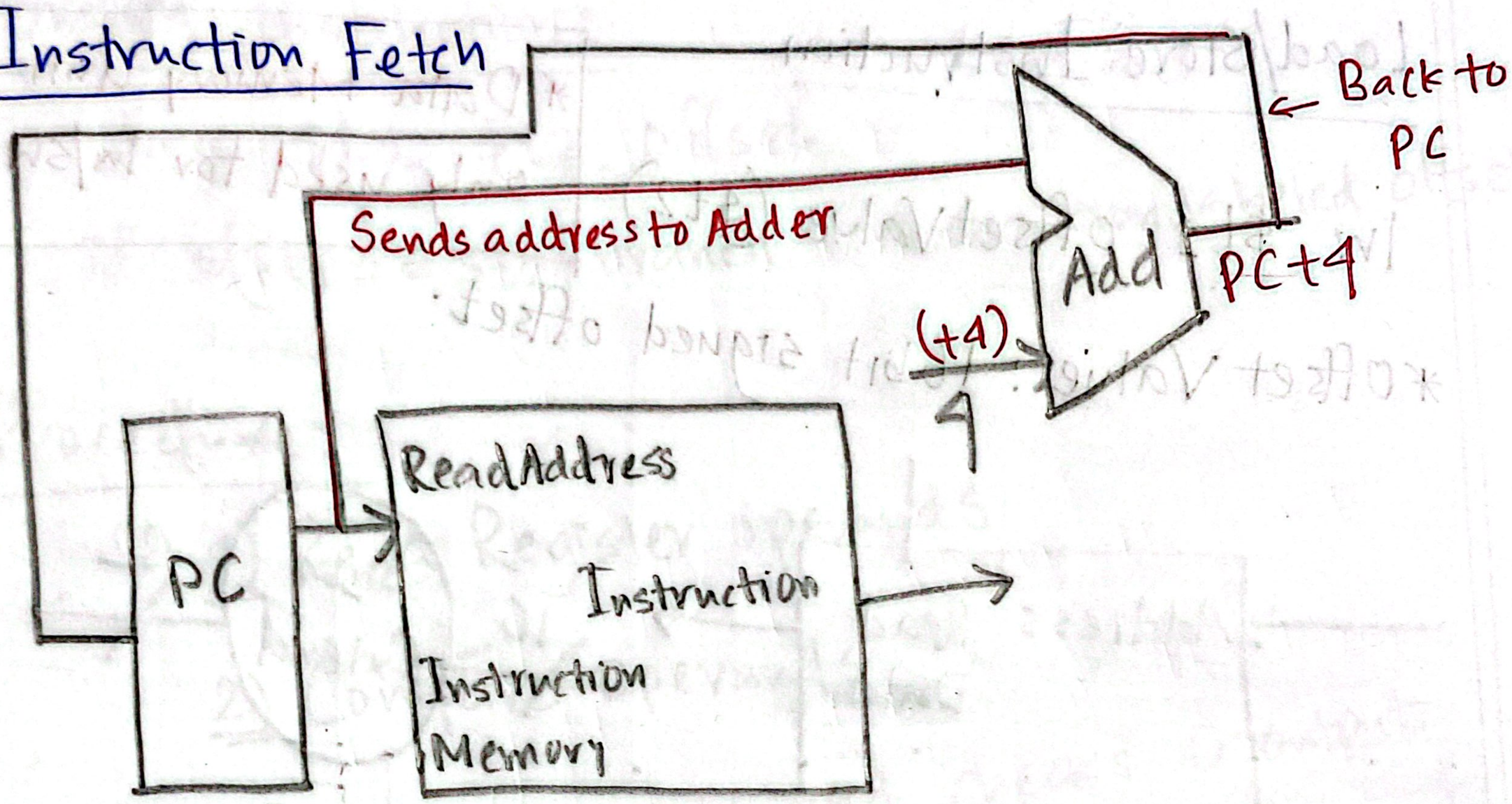
Actual = 5 + 8 + 5 + 10 = 28 ns

∴ Time waste = (40 - 28) = 12s

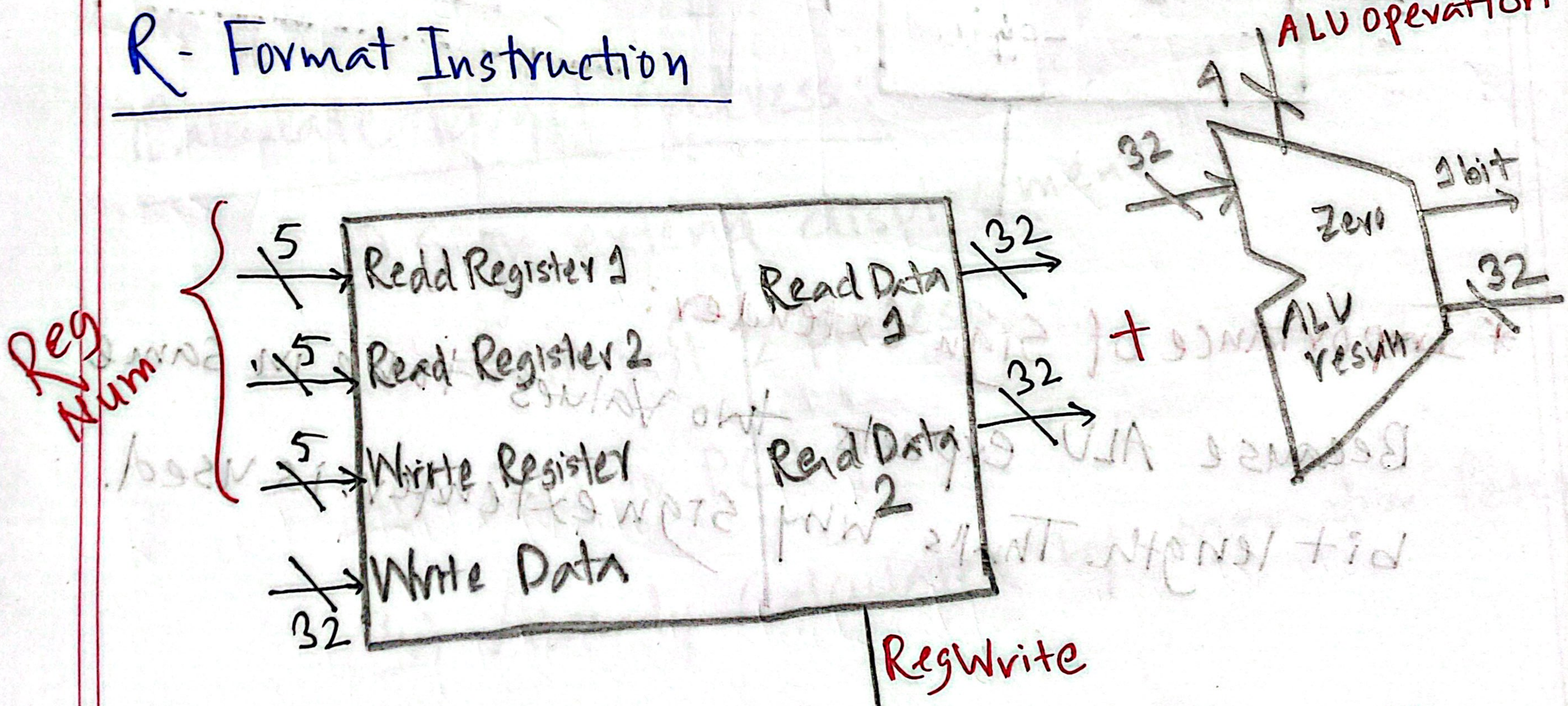
* For single cycle datapath, each instruction requires different clock pulse. And the instruction which takes longest time will be the clock period.

Building Datapath

Instruction Fetch



R- Format Instruction

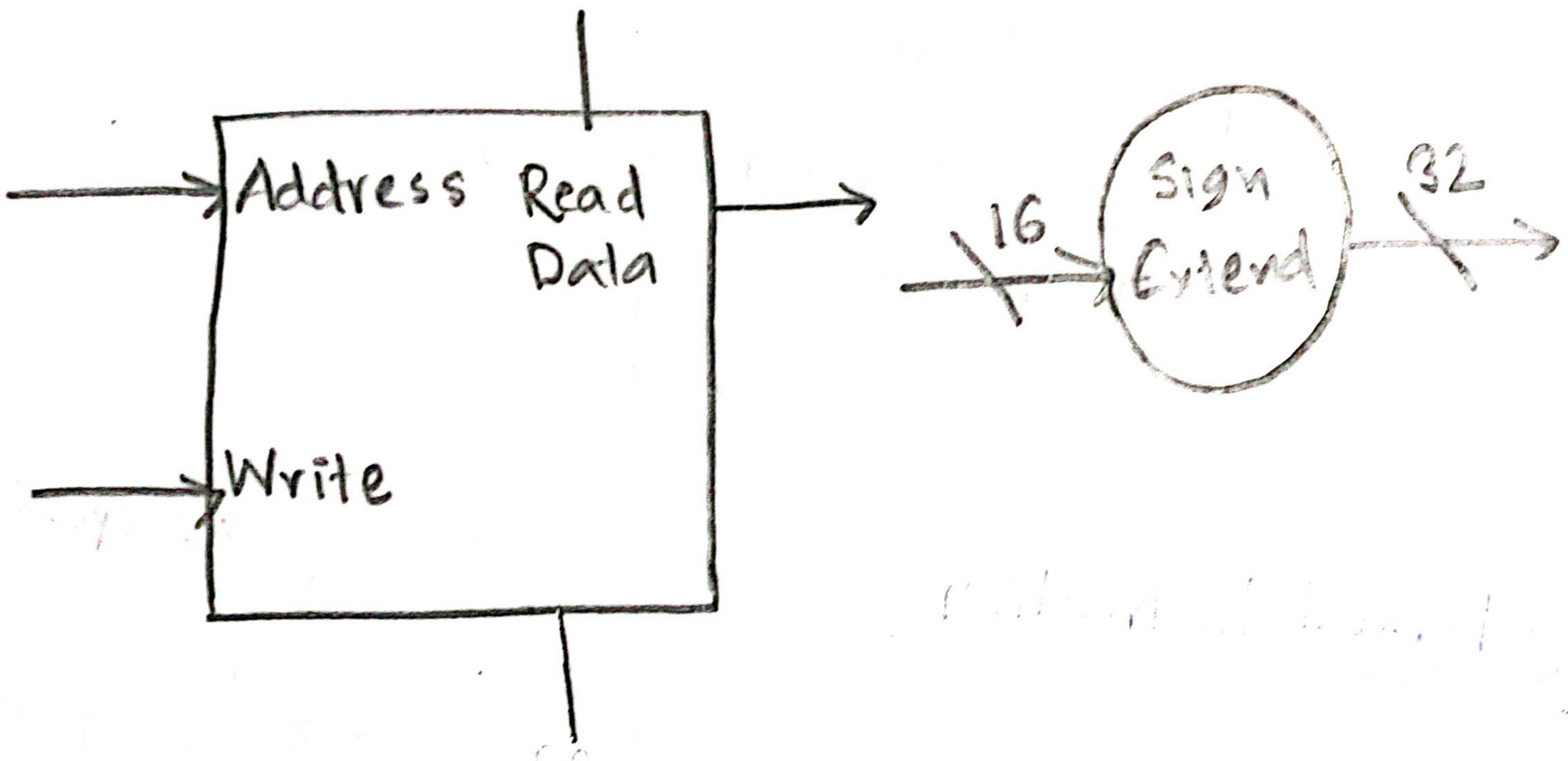


Load/Store Instruction

lw $t1, offsetValue ($t2)

* Data Memory Unit is only used for lw/sw

*offset Value : 16bit signed offset.



*Importance of Sign Extender:

Because ALU expects two values to be in same bit length. That's why sign extender is used.

** For example.

lw $8, 20($5)

Destination Source Address to fetch.

$5 + 20
↳ But it is in 16bit. So

to do the addition, we use sign extender

Branch Instruction:

beq $t1, $t2, offset

if ($t1 == $t2) branch to instruction labeled offset.

Procedure:

1) Read Register operands

2) Compare Operands.

3) ALU substracts and checks 0 output

Calculate target Address:

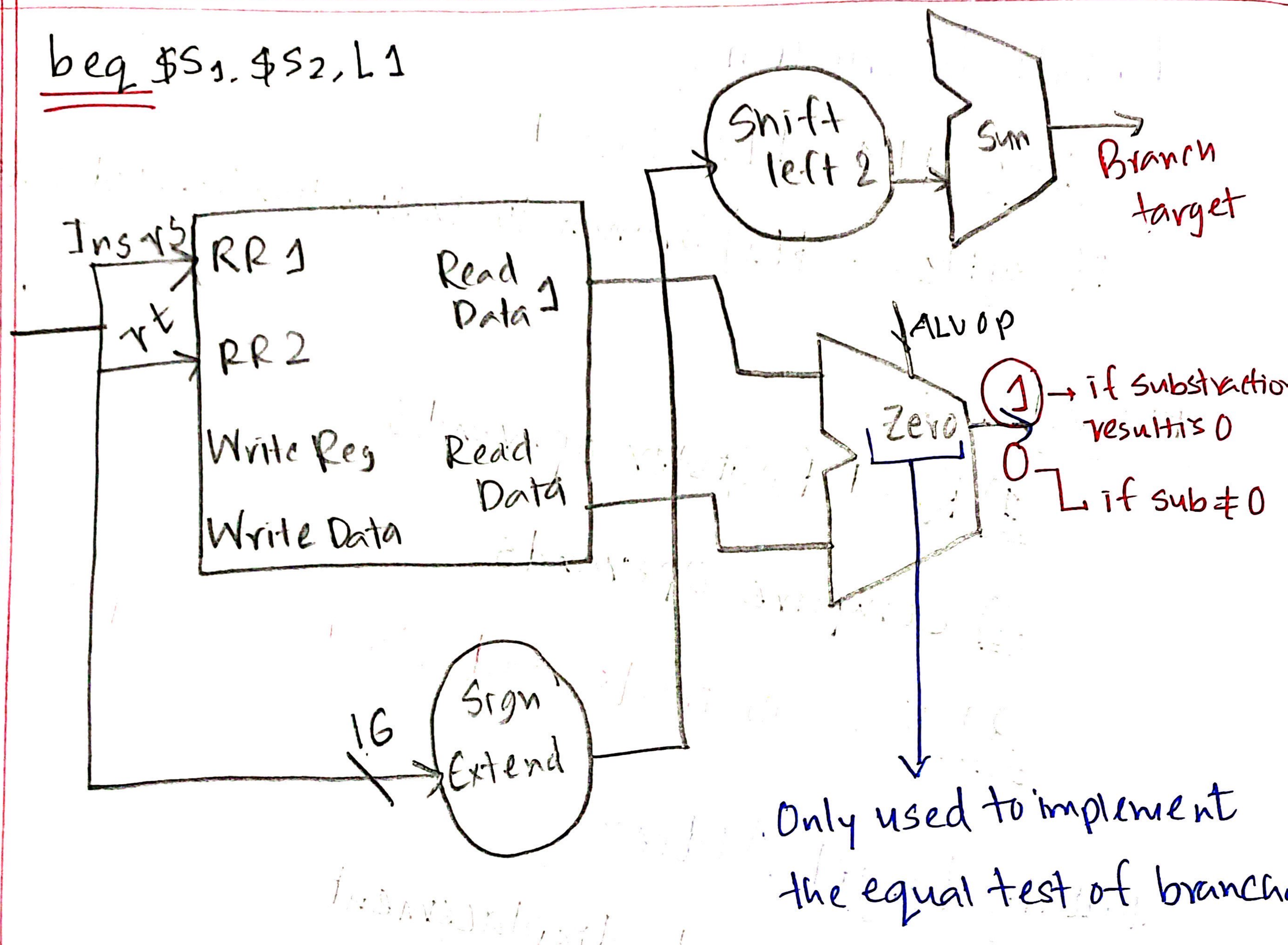
i) sign extend displacement

ii) Shift left 2 places.

iii) Add to PC+4

iv) Already Calculated by instruction fetch.

beq $S1, $S2, L1



* if the zero wire in ALU passes 1, the Calculated address replaces the PC counter Current Address.

Composing the elements

First cut datapath does an instruction in one clock cycle.

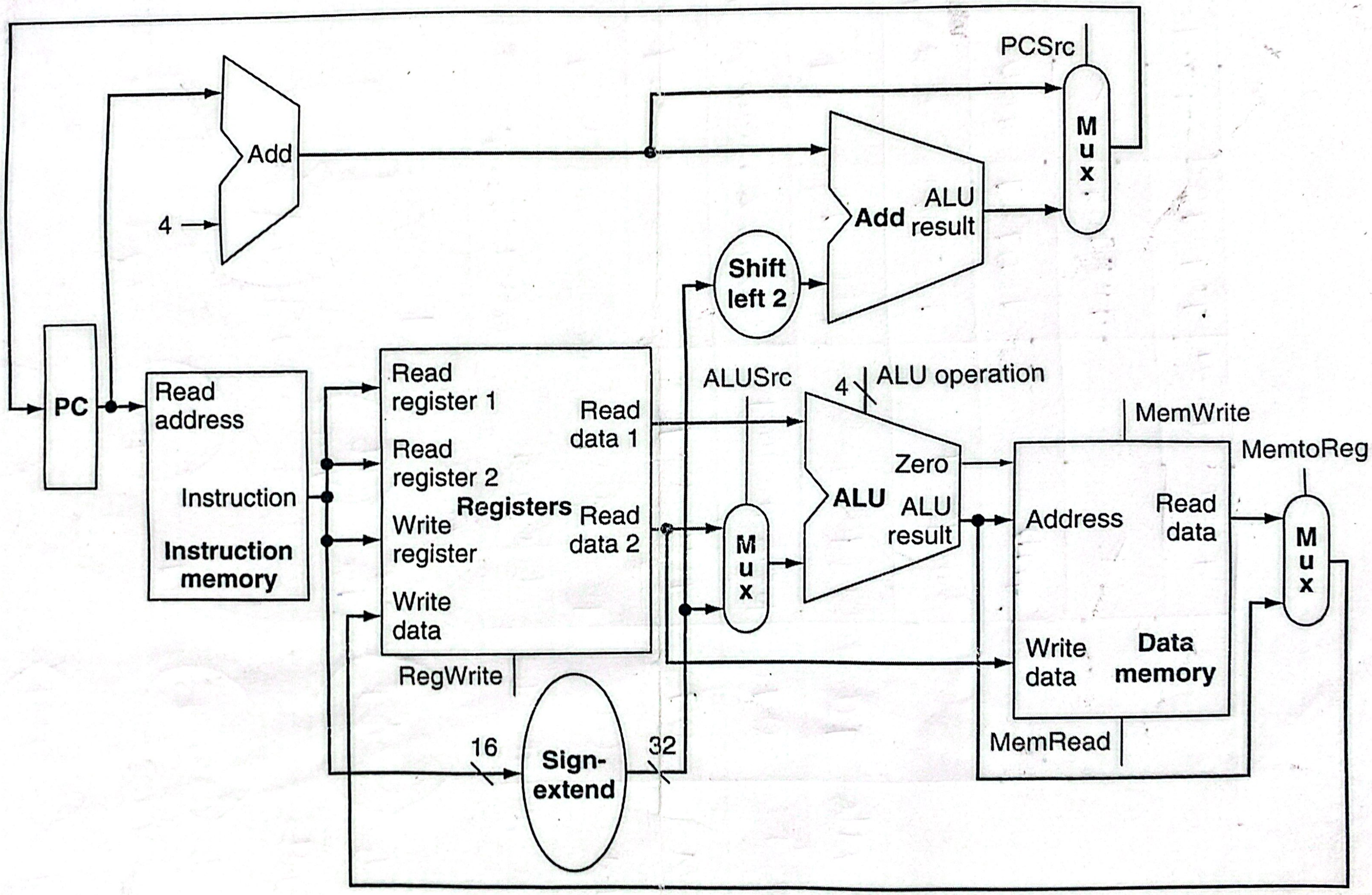
Hense, we need separate instruction and data memories.

That's why we use

Multiplexers & Control signal.

P.T.O

Full Datapath



ALU Control

*** Load/Store/Addi : F = add**

Reason:

* For load, we calculate memory address.

lw $8, 20($7). | sw $8, 20($7)

→ to calculate address of the source, we add $7 value and offset (20) to calculate address.

* Addi : constant value added.

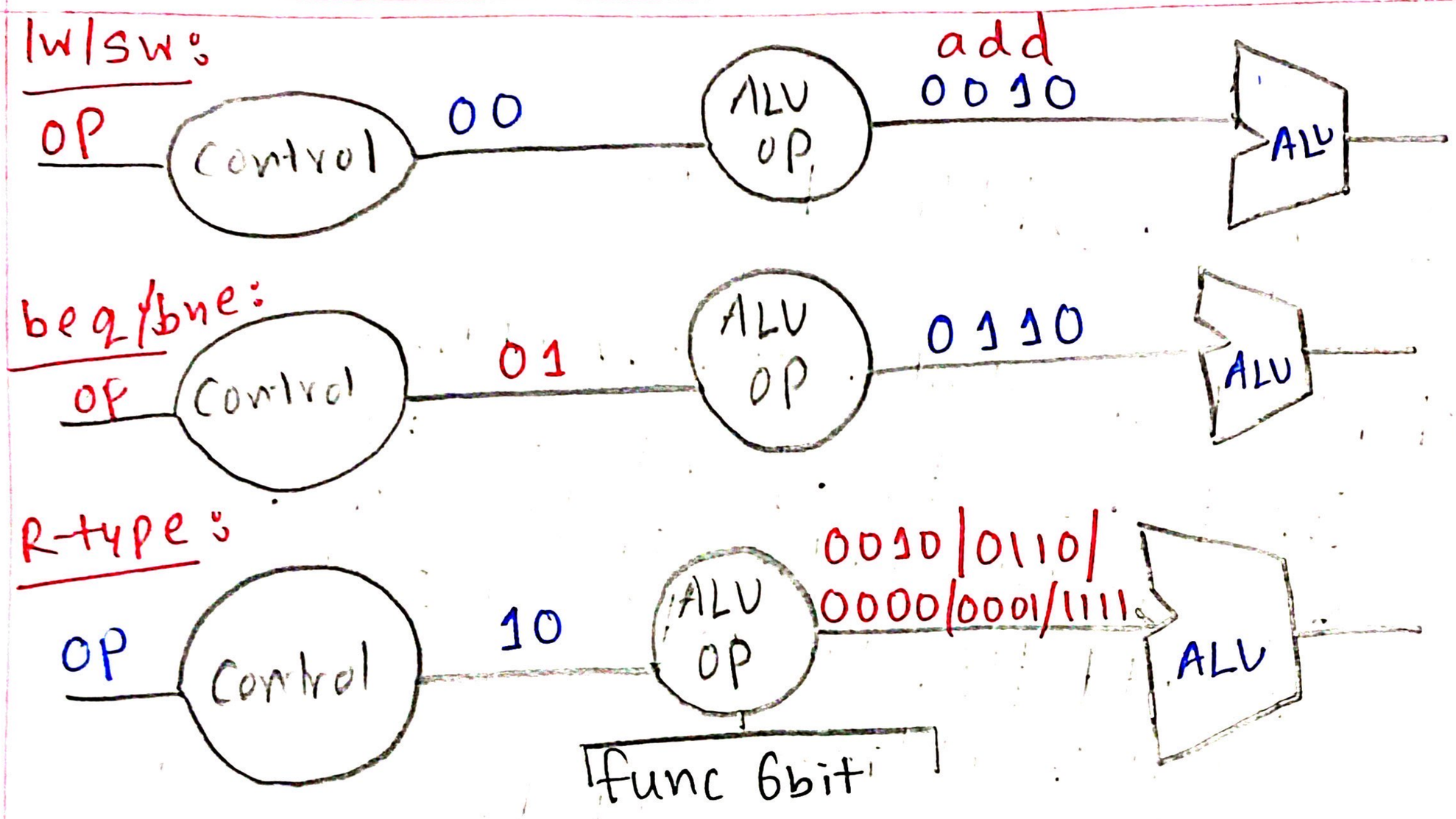
* Branch F = substract.

Reason:

bne, beq : we substract values from of two registers to check equality or inequality.

* R type Ins : F depends on funct field.

ALU Control	Function
0000	AND
0001	OR
0010	add
0110	Substract
0111	Set-on-less-than
1100	NOR

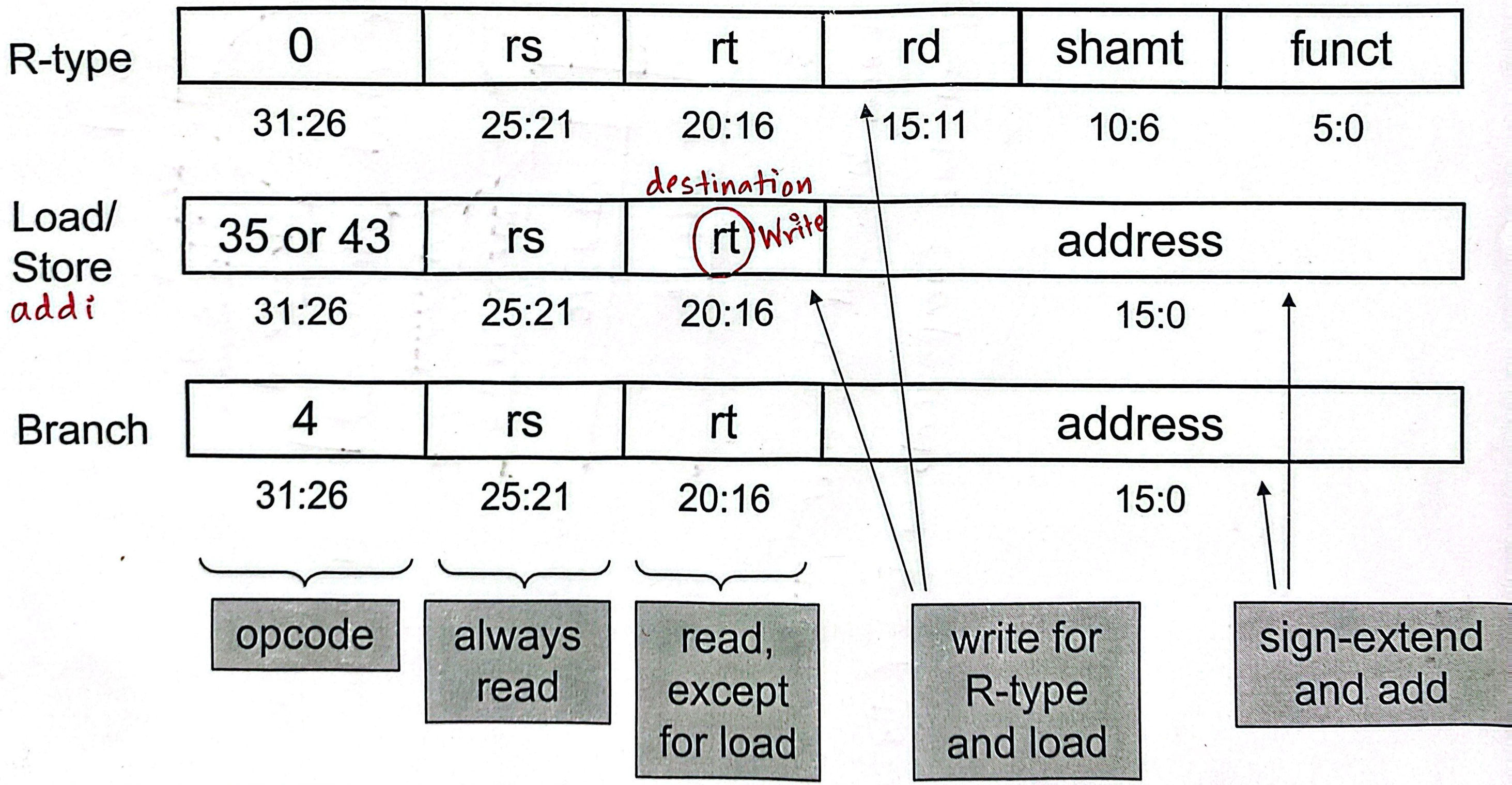


OPcode	ALUOP	Operation	funct	ALUfunction	ALU Control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	substract	0110
R-type	10	add	100000	add	0010
		substract	100010	substract	0110
		AND	100100	AND	0000
		OR	100101	OR	0001
		set-on-less-than	101010	set-on-less-than	0111

The Main Control Unit

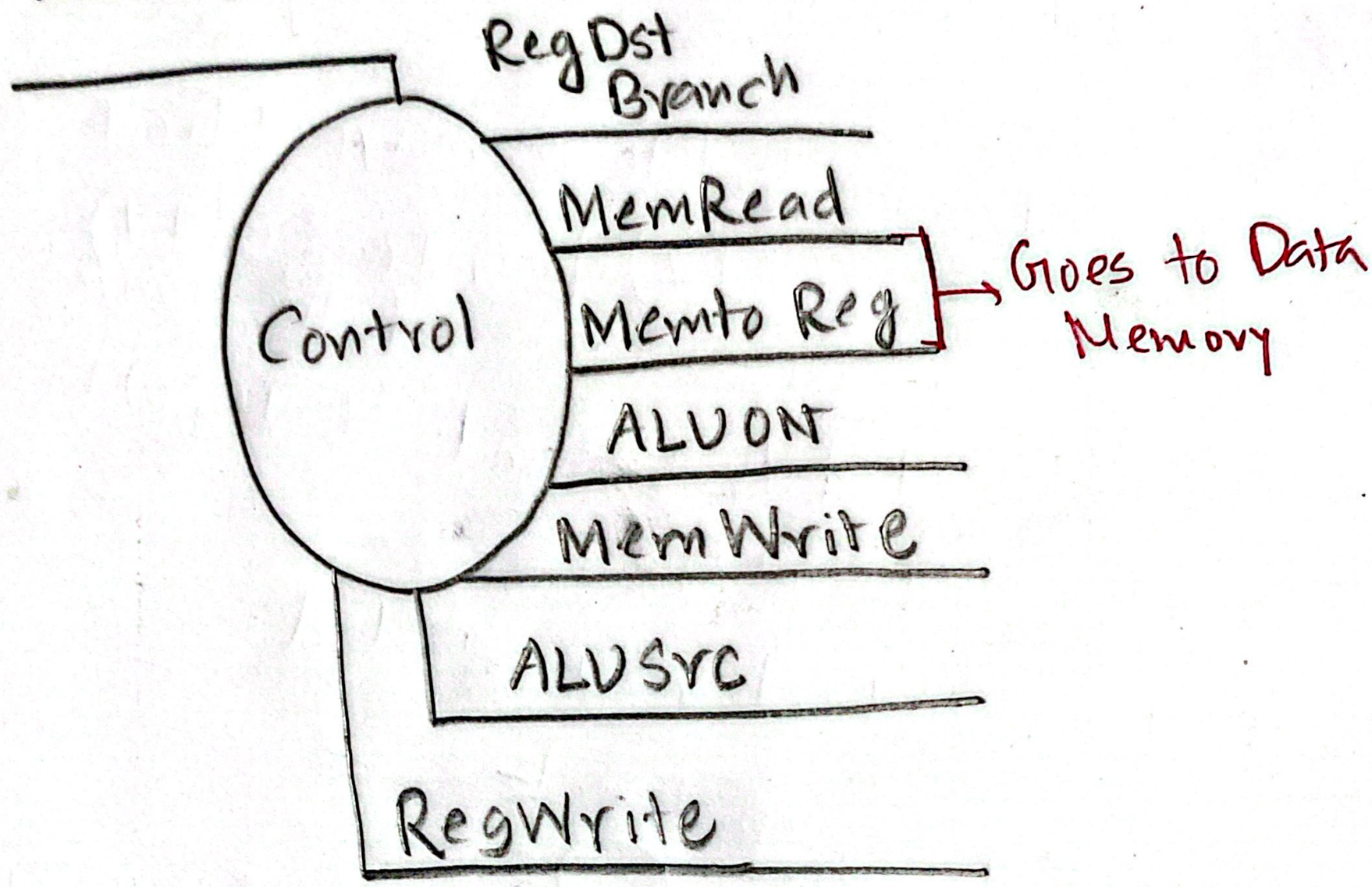
For sw (rs, rt) → Read
Branch (rs, rt) Read

- Control signals derived from instructio



Control

Wires from Control



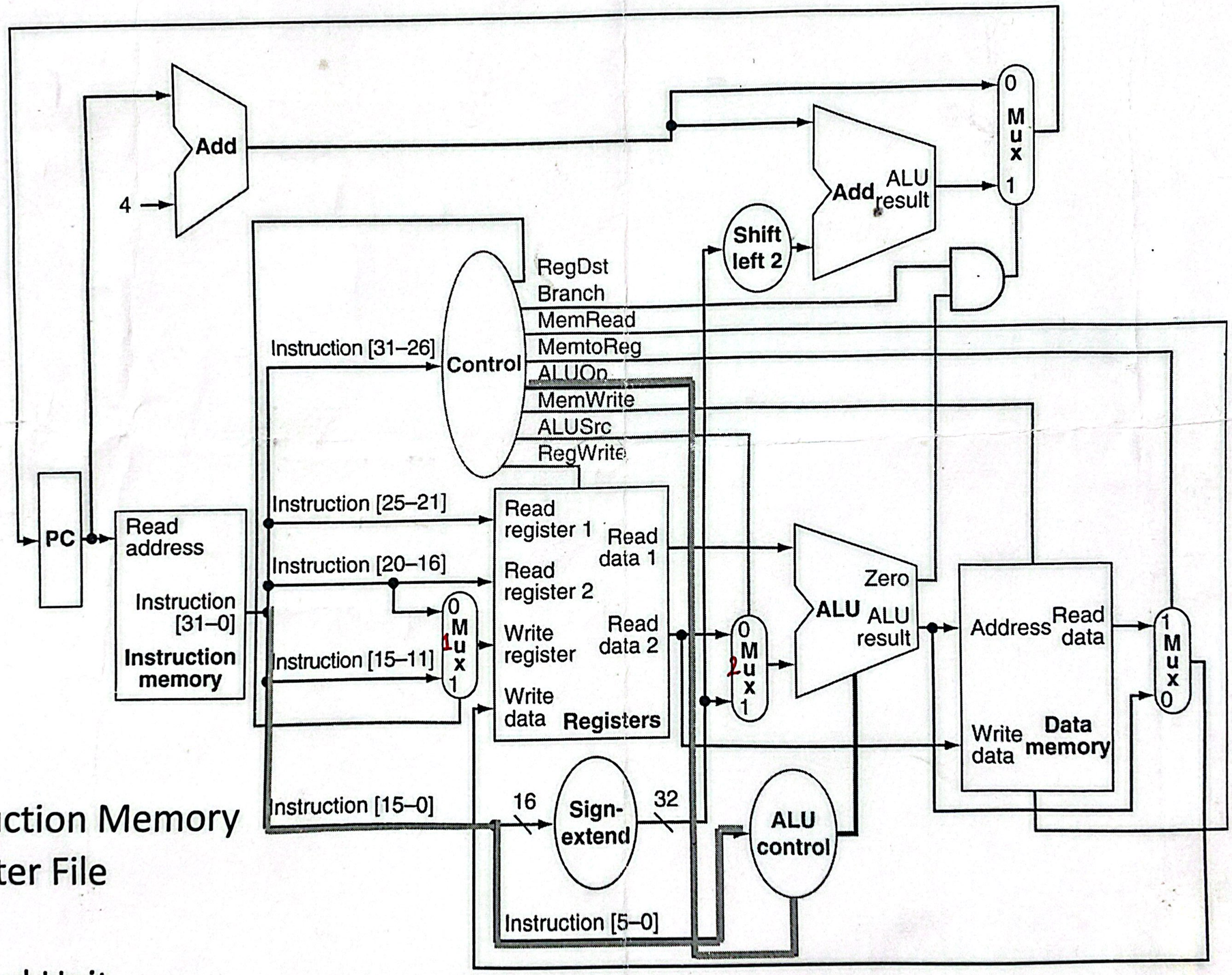
** We need to know how all the wires are connected to all the components in Datapath.

Datapath With Control

MUX 1:

MUX 1 = 0, Selects Rt for Write Register

MUX 1 = 1, Selects Rd for Write Register



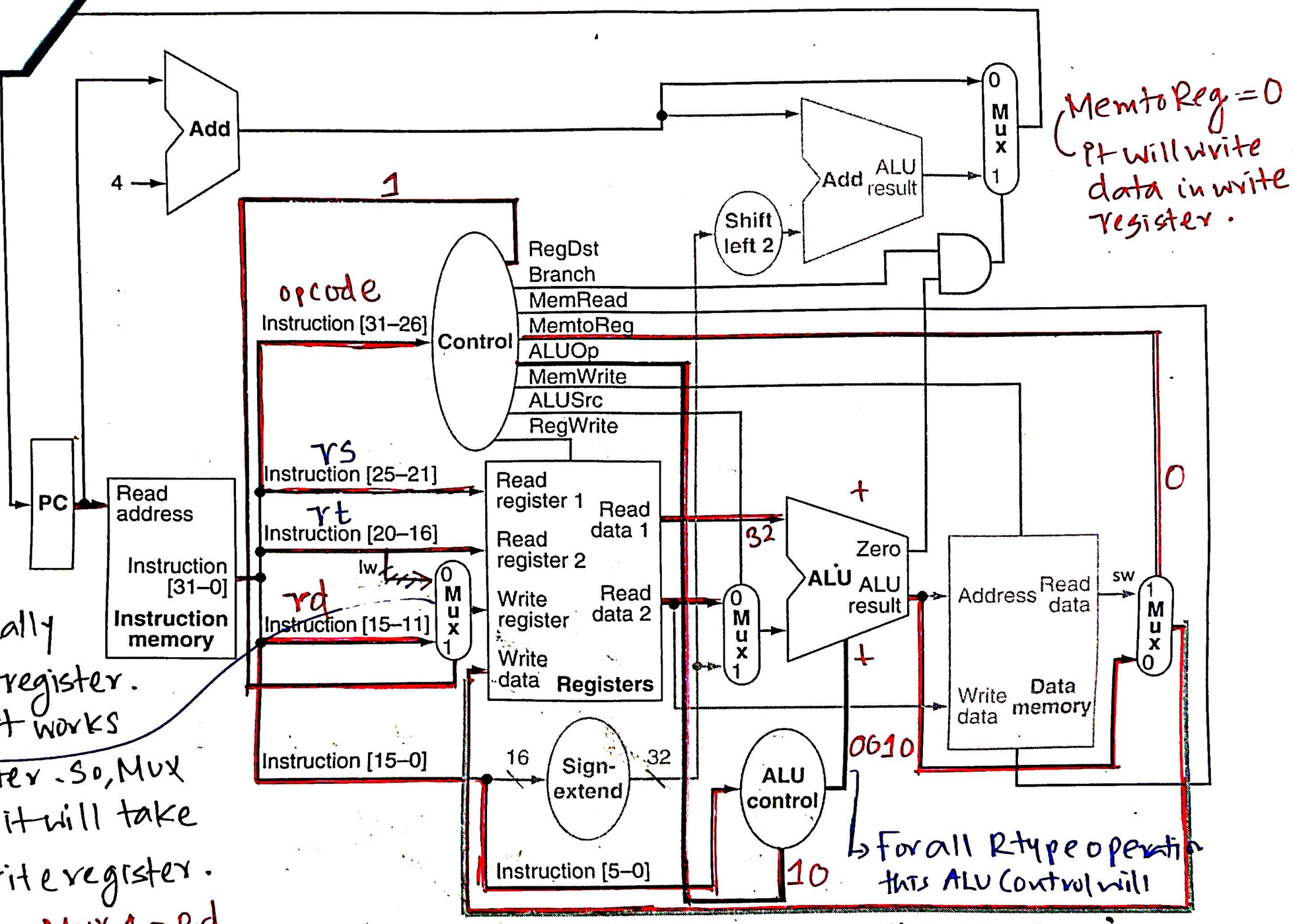
1. PC
2. Instruction Memory
3. Register File
4. ALU
5. Control Unit
6. Data Memory (lw/sw)

RegDstBranch=0 (rt write)
ALU Control =0010
ALUop = 00
ALUSrc = 1
MemRead = 1
MemtoReg = 1

...ction

add $8, $17, $18

000000	10001	10010	$18	00000	100000
opcode	rs	rt	rd	shamt	funct



MemtoReg = 0
It will write data in write register.

This MUX actually enables write register. as for lw/sw, rt works as write register. So, Mux will decide if it will take rt/rd for write register.

MUX 0 = rt, MUX 1 = Rd

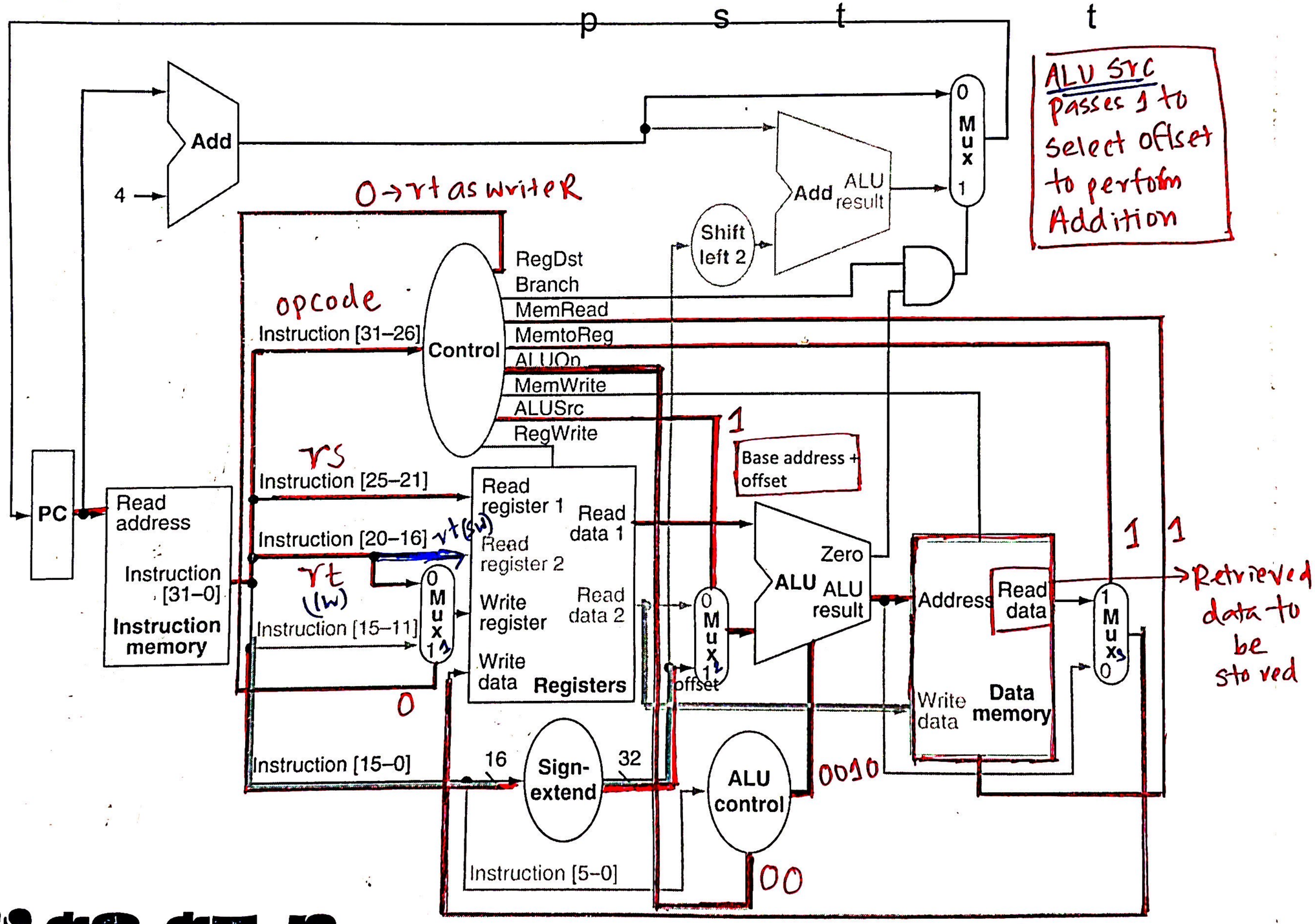
→ For all Rtype operation this ALU Control will change only

Load Instruction

for lw/sw
- ALU does (Add) 00

lw $8, 20($7)

op	$7	$8	20
op	rs	rt	offset



addi $8,$7, 3

Sign Extend used so that ALU can perform addition between equal bits.

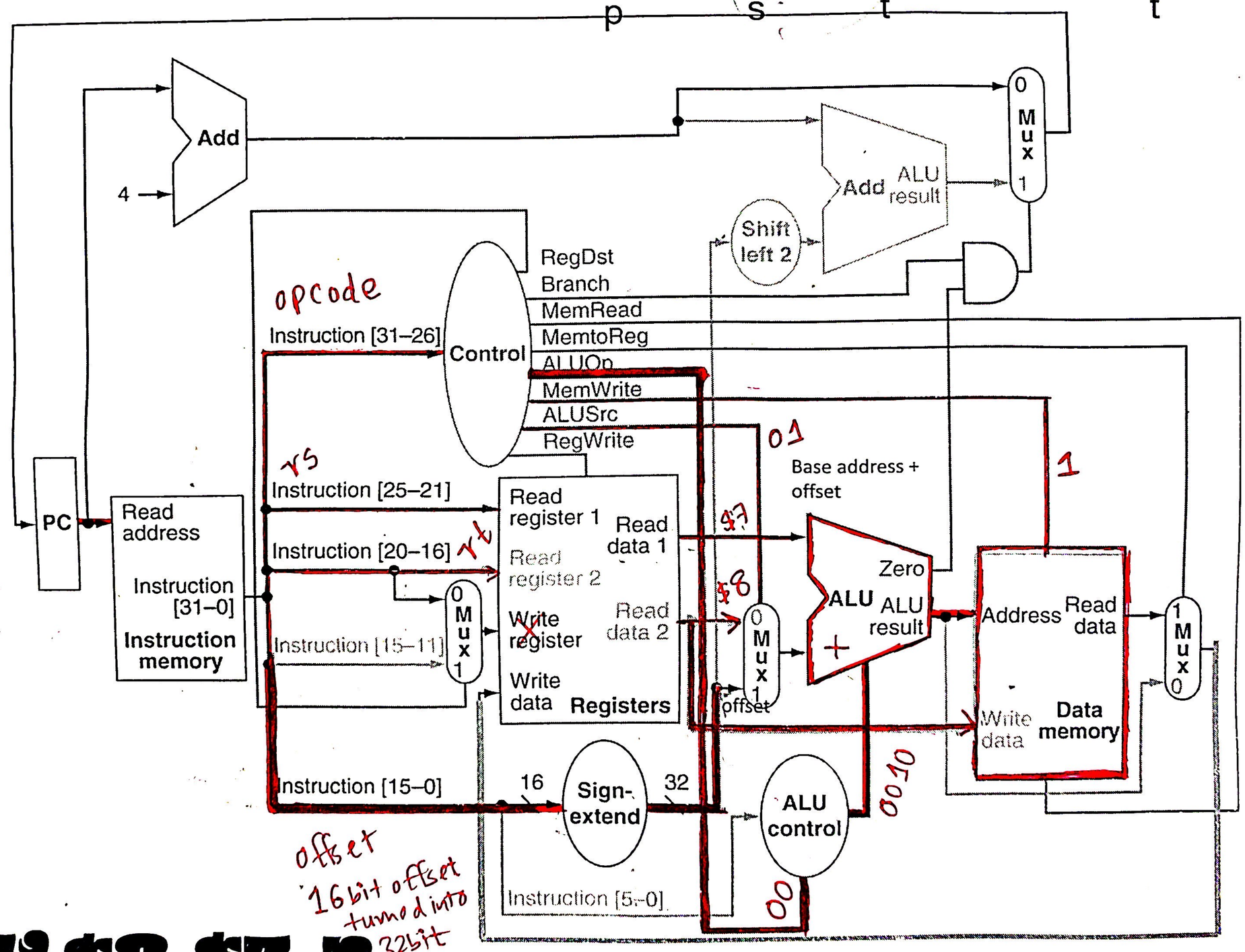
for sw, rt is read.

Load Instruction

sw

lw $8, 20($7)

op	$7	$8	20
op	rs	rt	offset



offset
16bit offset turned into 32bit

addi $8,$7, 3

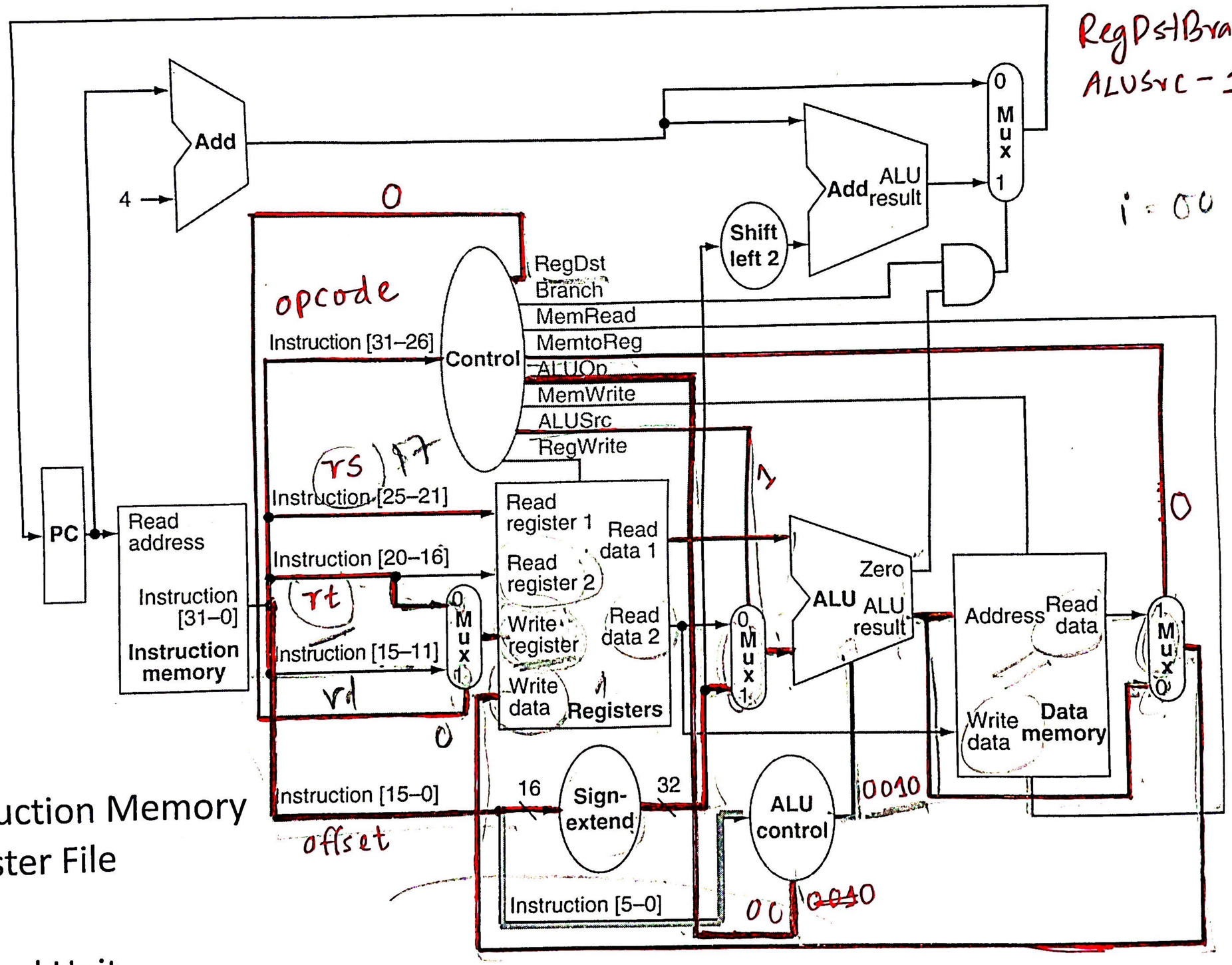
Datapath With Control

addi $8, $7, 3

ALUOP - 00
ALU Control → 0010
RegDstBranch → 0
ALUSrc - 1

i = 00

lw sw



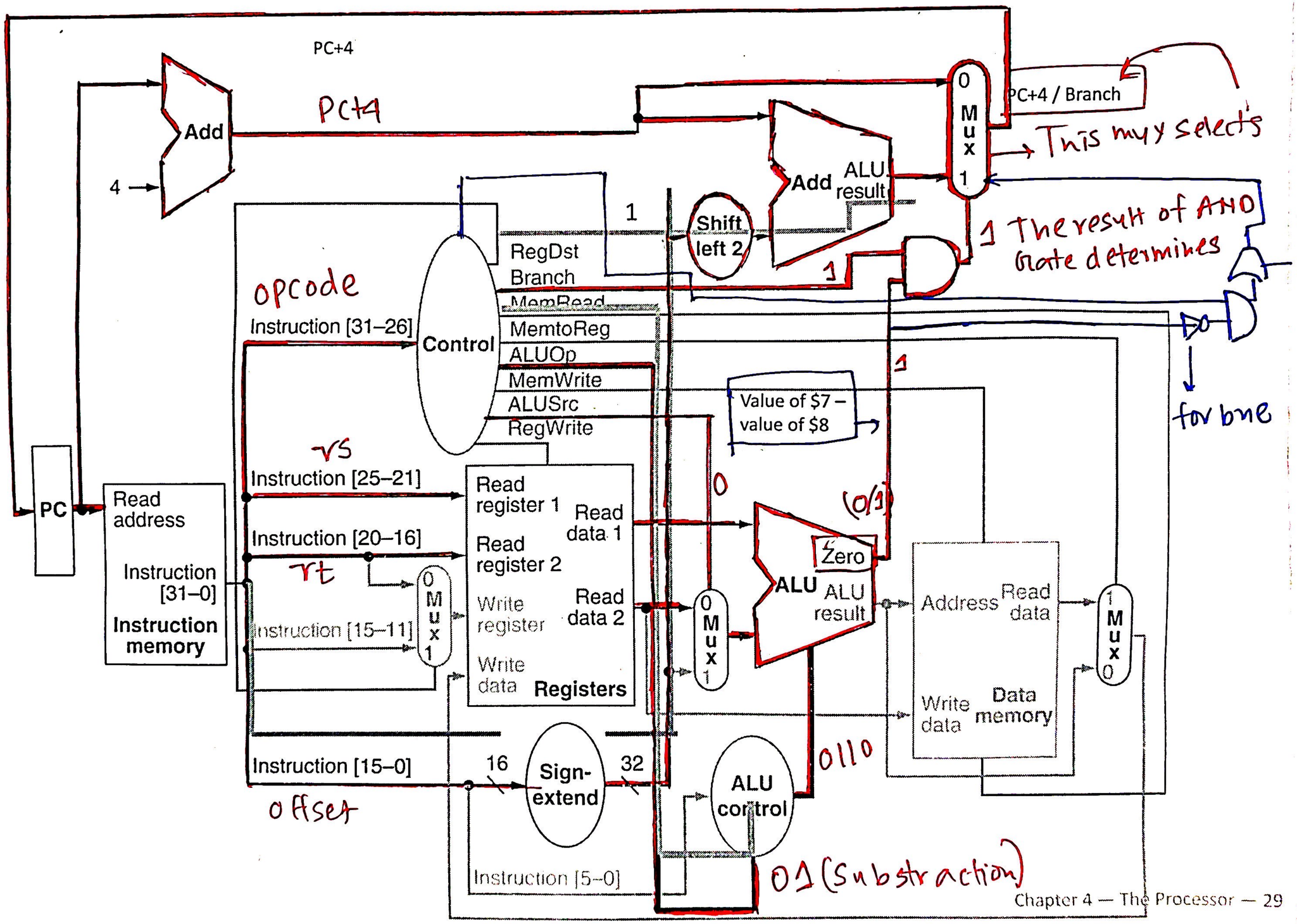
1. PC
2. Instruction Memory
3. Register File
4. ALU
5. Control Unit

Branch-on-Equal Instruction

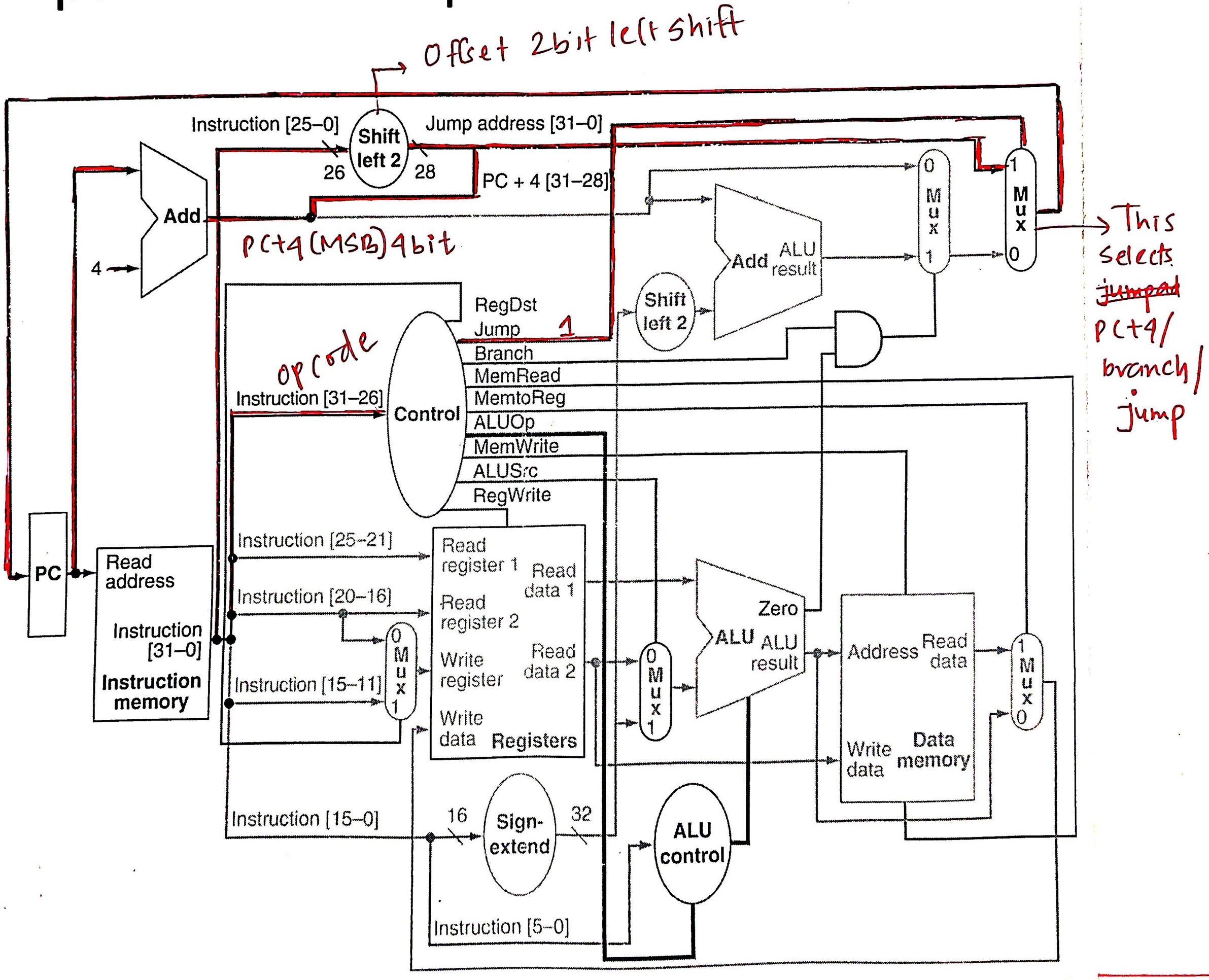
beq $7, $8, L1

Op	$7	$8	2
op	rs	rt	offset

ALU Zerowire = 0 → result not equal
1 → result equal



Datapath With Jumps Added



Performance Issue

*Longest delay → clock period.

SCD → every instruction gets one individual clock cycle.

add - 5 ns

mul - 8 ns

sub - 5 ns

lw - 10 ns → longest delay → clock period

∴ Total time = 10 + 10 + 10 + 10 = 40

But Actual time = 5 + 8 + 5 + 10
= 28 ns.

∴ Time Waste = (40 − 28) = 12 ns

*Critical path → load Instruction.

Instruction memory → Register file → ALU →

data memory → Register File.

MIPS Pipeline

Five stages.

IF: Instruction Fetch from memory [Uses Instruction Memory]

ID: Instruction Decode and Register Read. [Uses Register

Ex: Execution Operation. [Uses ALU]

MEM: Access Memory Operand [Uses Data Memory]

WB: Write back result to register.

Even if not every stage is used, we have to allocate time for every stage. For example, ~~Ad~~ R-type ins doesn't use MEM stage. So, for the stage, it will sit idle.

Pipeline Performance

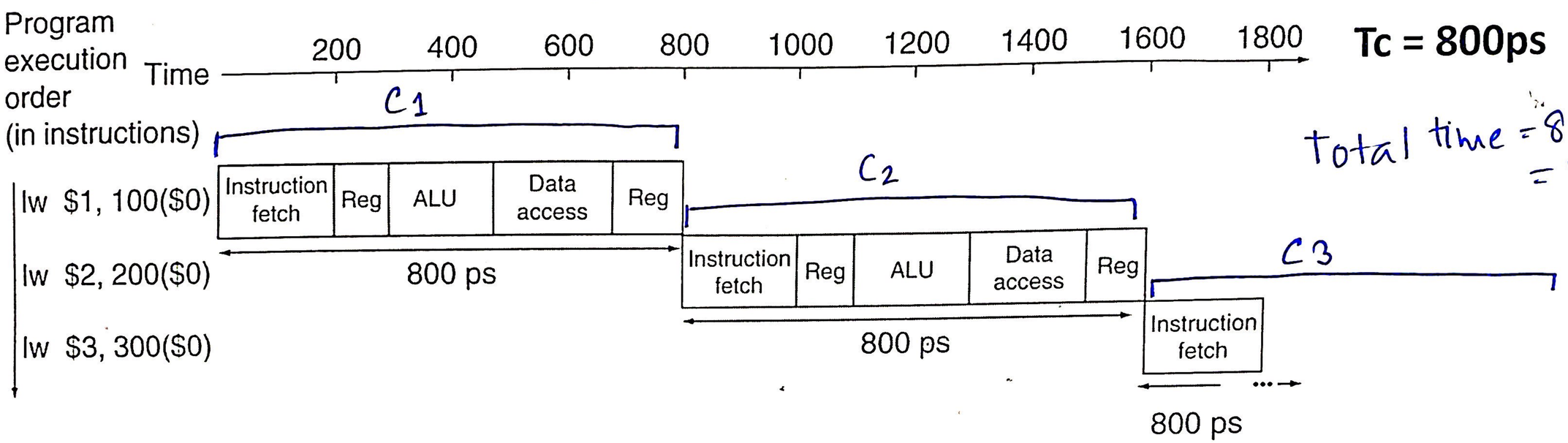
* Stage time = Clock period.
* In one clock cycle multiple stage are conducted.

- Assume time for stages is
  - 100ps for register read or write
  - 200ps for other stages
- Compare pipelined datapath with single-cycle datapath

Instr	Instr fetch (IR)	Register read (ID)	ALU op (EX)	Memory access (MEM)	Register write (WB)	Total time
lw	200ps	100 ps	200ps	200ps	100 ps	800ps
sw	200ps	100 ps	200ps	200ps		700ps
R-format	200ps	100 ps	200ps		100 ps	600ps
beq	200ps	100 ps	200ps			500ps

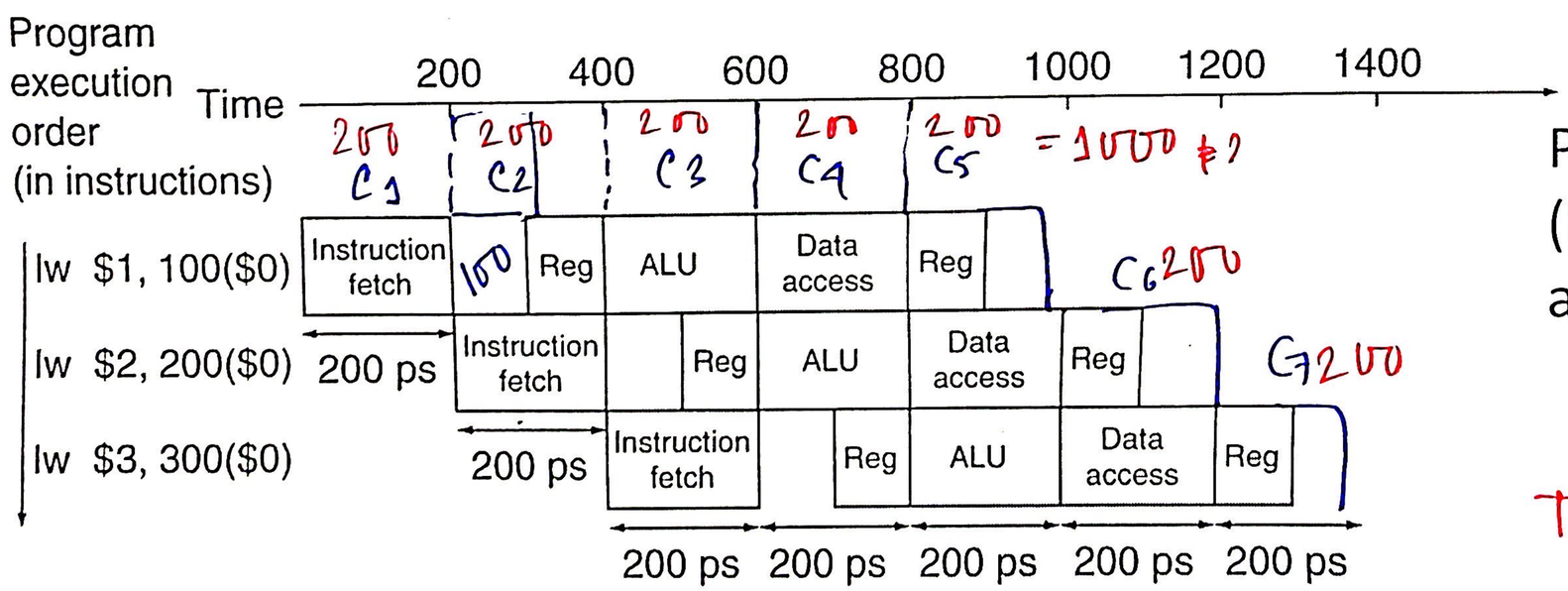
Pipeline Performance

Single Cycle Clock Cycle Tc = (Highest) Duration to complete an instruction



Tc = 800ps

Total time = 800x3 = 2400s



Pipelined Clock Cycle Tc = (Highest) Duration to complete a stage

Tc = 200ps (per stage)

Total = 1000 + 200 + 200 = 1400

Pipelining increases throughput

Pipeline Speedup

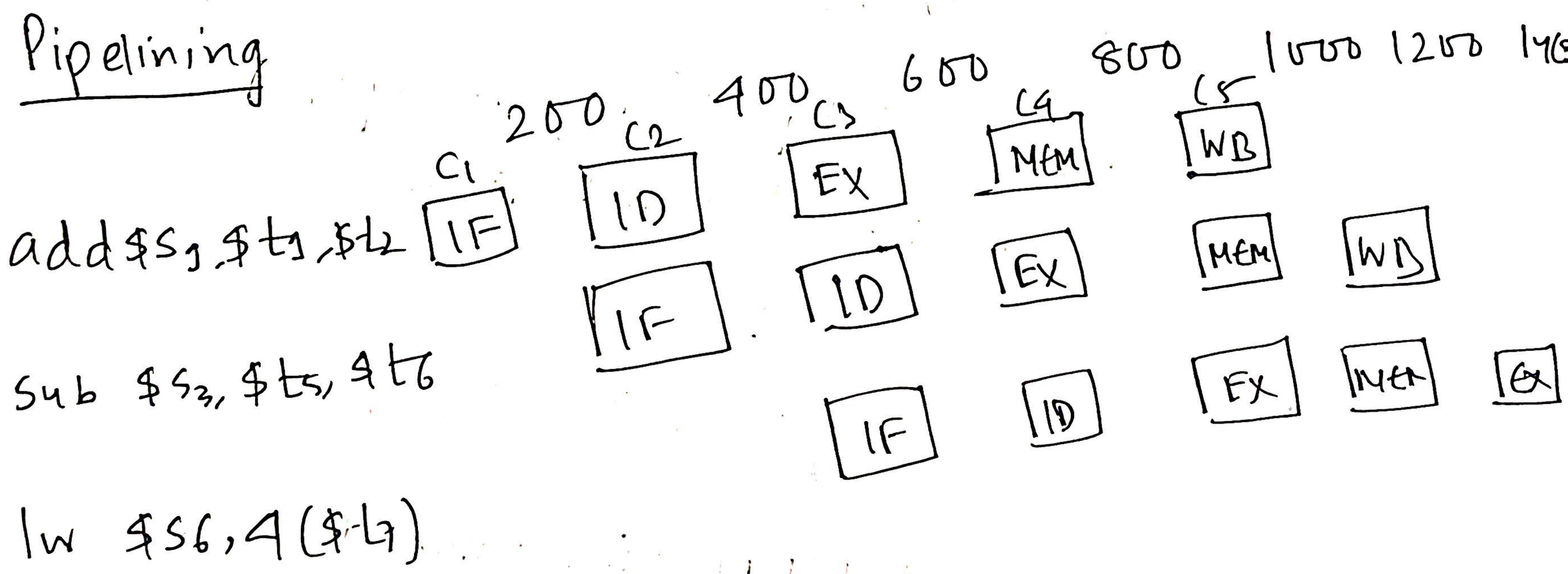
* If all stages are balanced:
  ∴ all the stages will take same amount of time.

* $\text{Time between instructions}_{\text{pipelined}} = \dfrac{\text{Time between instructions}_{\text{non-pipelined}}}{\text{Number of stages.}}$

If not balanced speedup is less.

* Not possible in real life.

Pipelining



Hazard

- → Structure hazard
- → Data Hazard
- → Control Hazard

** Structure Hazard: A required resource is busy

→ Conflict for use of a resource.

→ Load/store requires data access.

Von Neumann

→ Instruction fetch would have to stall for the cycle.

bubble

→ Pipeline datapaths require separate instructions | data memories.

** We have to see that if two instructions are in same clock cycle trying to use same hardware.

** here, IF and MEM using same (data + ins) memory.

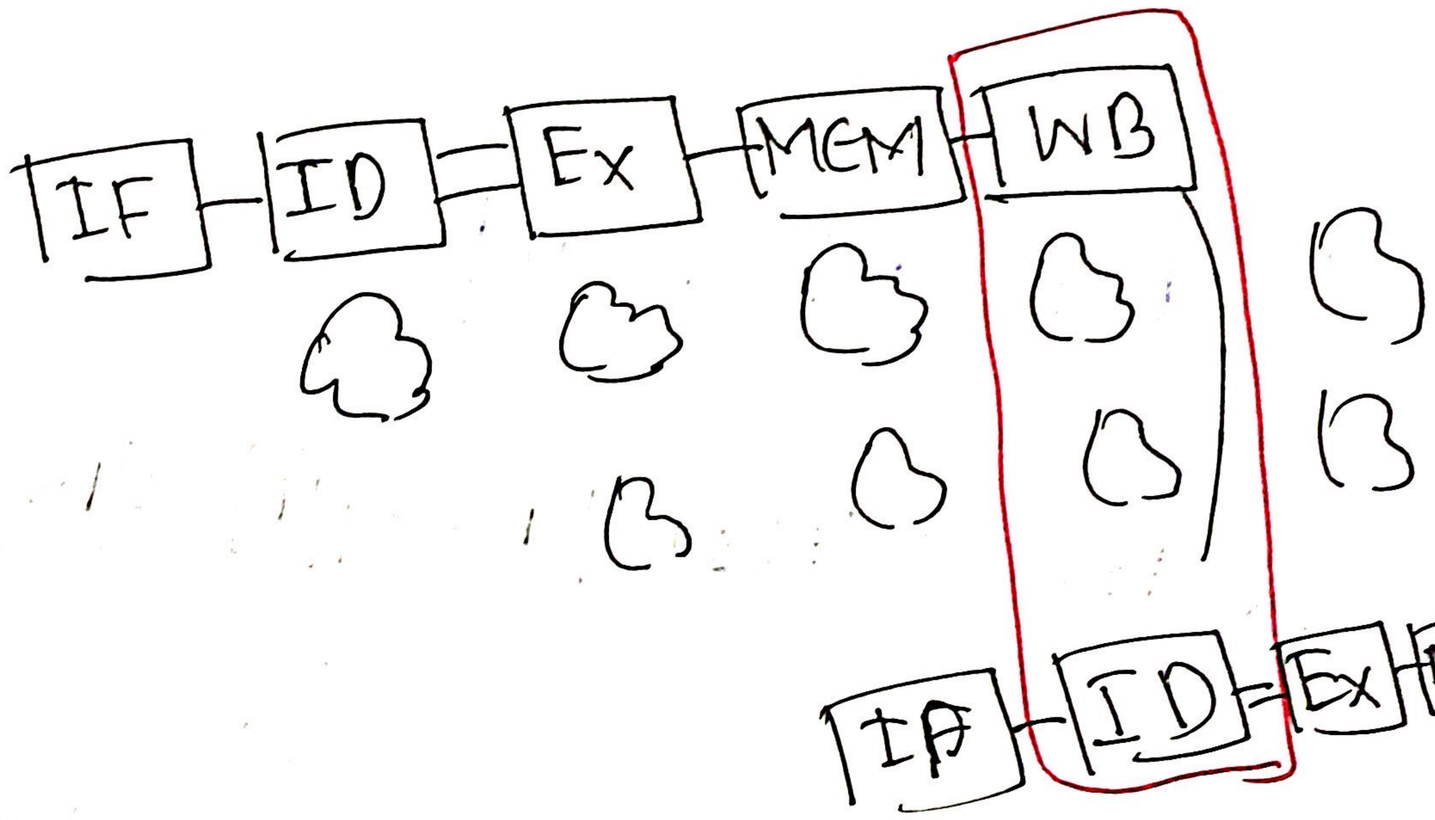
Data Hazards

An instruction depends on completion of data access by a previous instruction.

add $s0, $t0, $t1
sub $t2, $s0, $t3

here, we align WB and ID.

add $s0, $t0, $t1



sub $t2, $s0, $t3

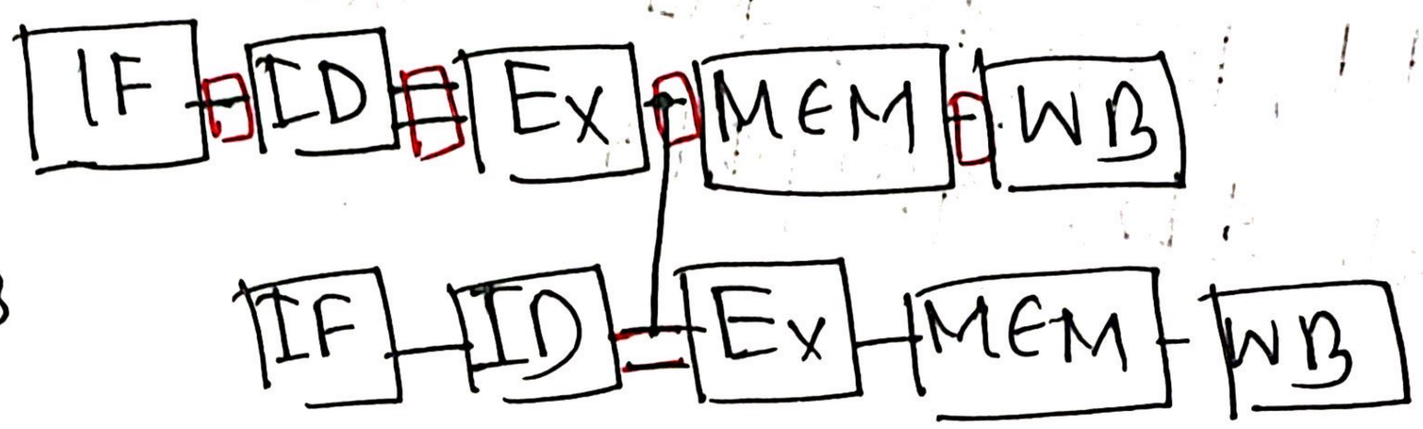
Forwarding

Don't wait for result to be stored in a register.

Requires extra Connections in datapath (registers)

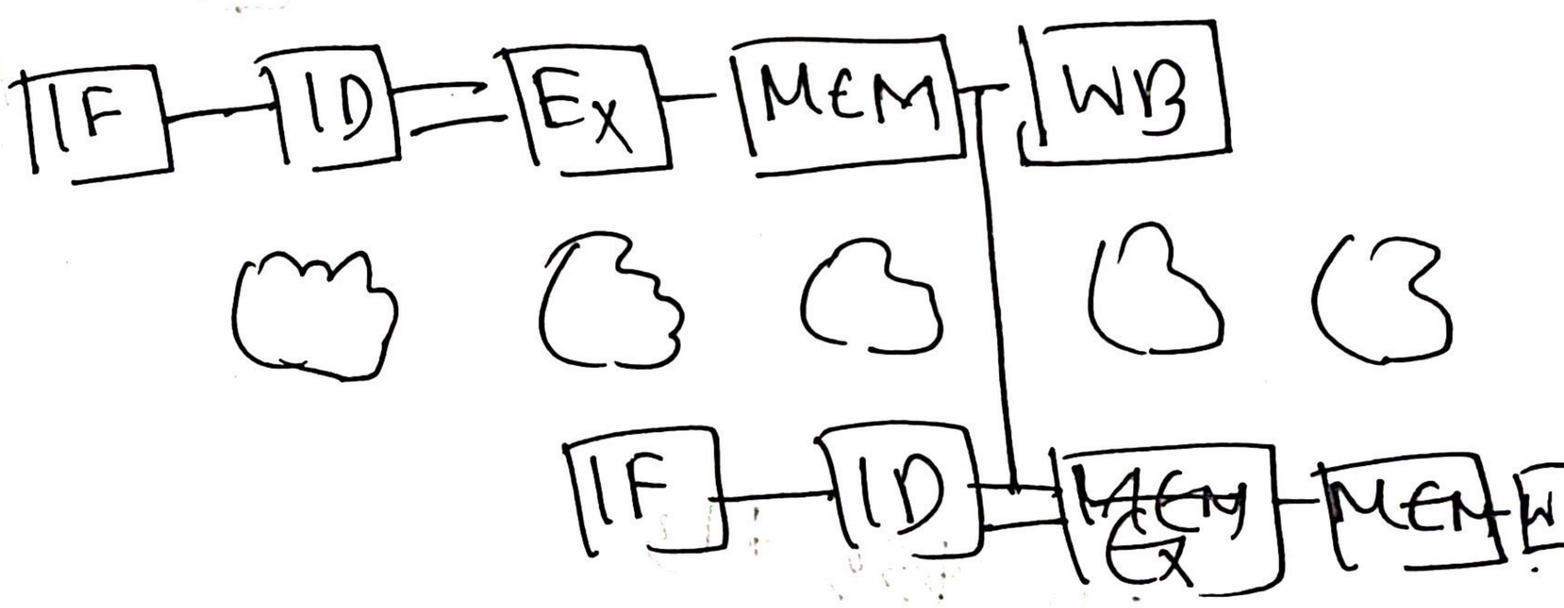
add $s0, $t0, $t1

sub $t2, $s0, $t3



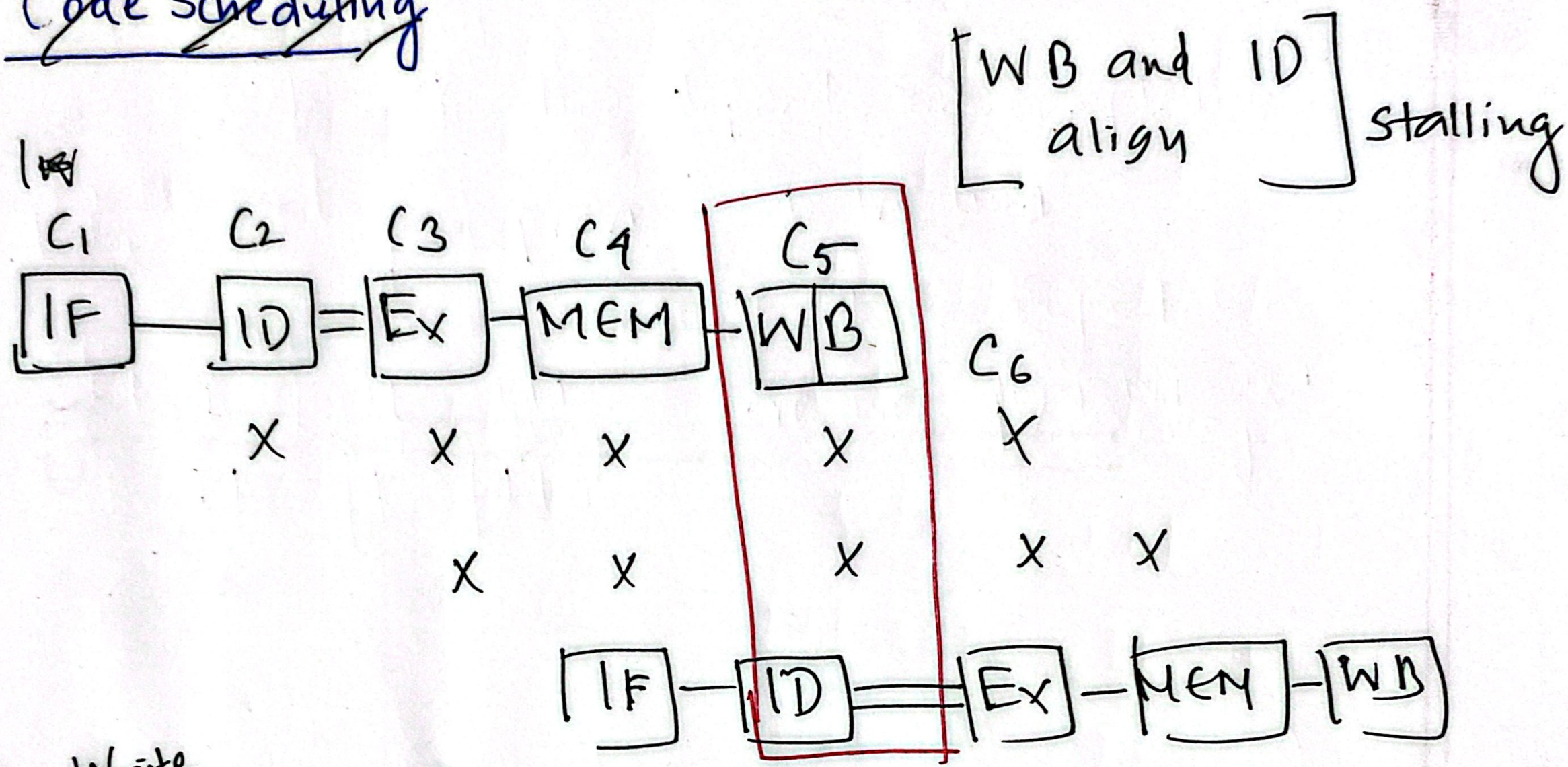
load use data Hazard

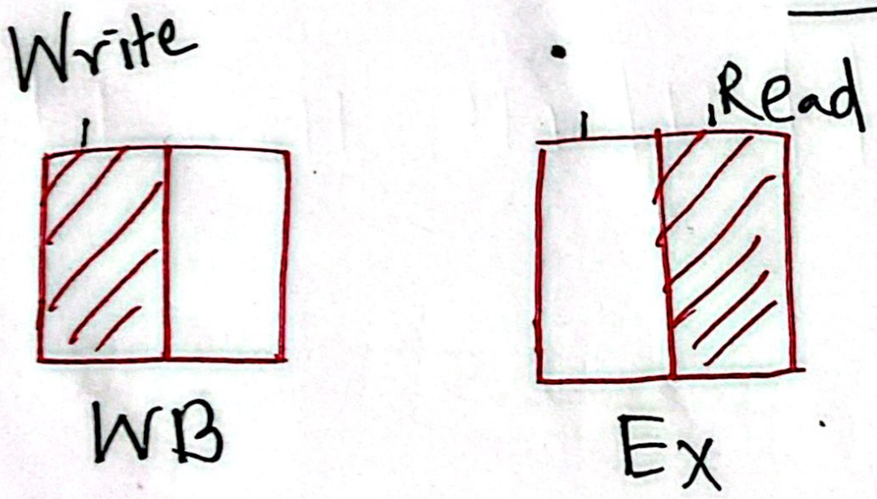
lw $s0, 20 ($t1)



Note: For data hazard, we must know that hazard occurs when the destination register's value is changed and that value is used in next another step.

Code Scheduling

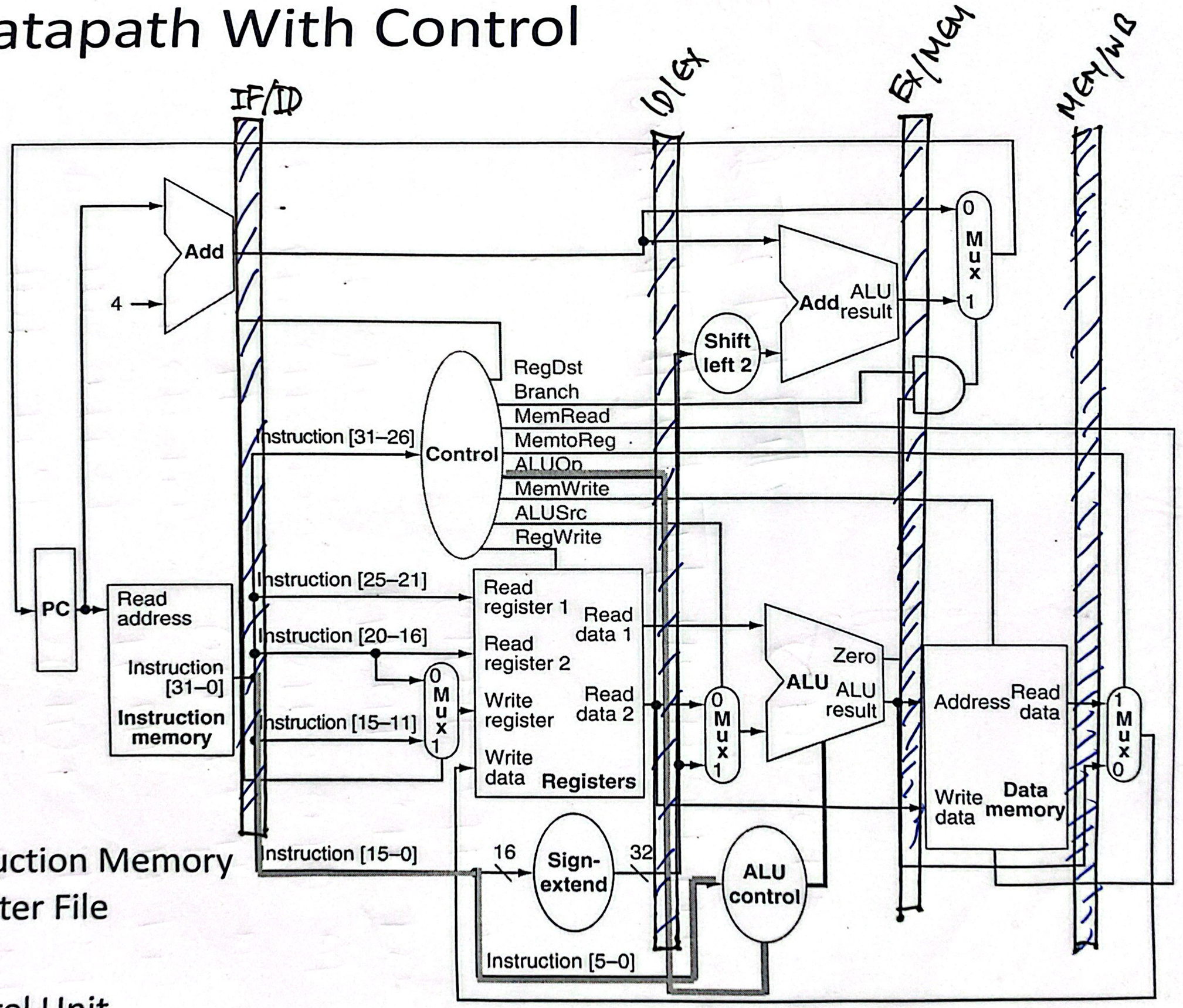




But if here every ~~st~~ ins takes 200ps. So, first WB half (here, 200/2 = 100) will write. and in ID, last half decodes data. So, the extra half are needed to be aligned so that, we can get data faster.

** If, the ~~data~~ clock cycle for all is 200. but WB and ID take 150. Then this would not work. So, we would not be able to align WB and ID as the first half take 150 but the other half 50 will ~~take~~ not be able to do the other half.

Datapath With Control



1. PC
2. Instruction Memory
3. Register File
4. ALU
5. Control Unit
6. Data Memory (lw/sw)

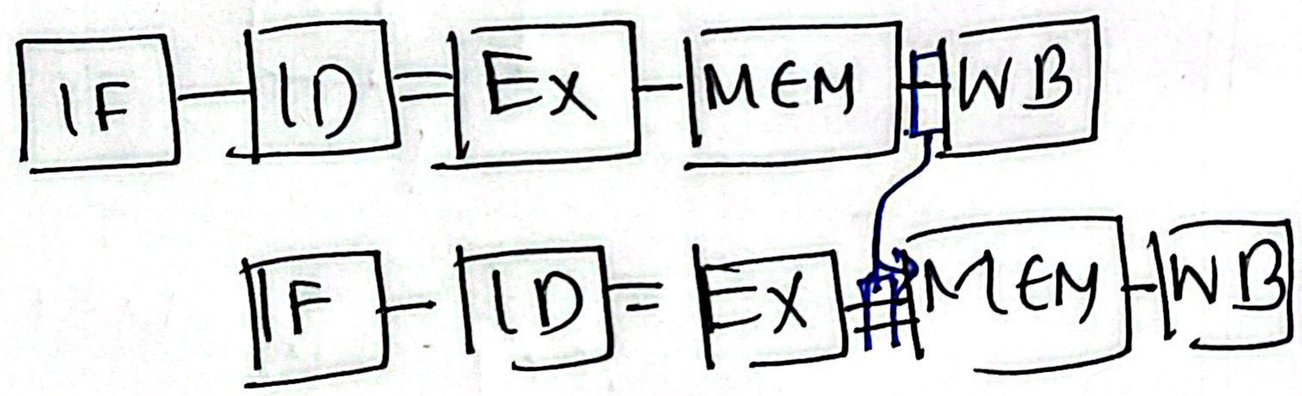
Code Scheduling

```
lw $t1, 0($t0)   5
lw $t2, 4($t0)   6
add $t3, $t1, $t2
sw $t3, 12($t0)
lw $t4, 8($t0)
add $t5, $t1, $t4
sw $t5, 16($t0)
```

After Code scheduling

```
lw $t1, 0($t0)    5
lw $t2, 4($t0)    6
lw $t4, 8($t0)    7
add $t3, $t1, $t2 8
sw $t3, 12($t0)   9
add $t5, $t1, $t4 10
sw $t5, 16($t0)   11
```

```
add $s0, $t0, $t1
sw $s0, 0($t2)
```



Pipelining Hazard.

	rd	rs	rt	
add	$5,	$6,	$7	IF – ID – Ex – MEM – WB
sub	$2,	$5,	$7	IF – ID – Ex – MEM – WB

Hazard Check:

ID/Ex.RegRs = Ex/Mem.RegRd
ID/Ex.RegRt = Ex/Mem.RegRd.

Consecutive

add	$5,	$6,	$7	IF – ID – Ex – MEM – WB
sub	$2,	$5,	$7	IF – ID – Ex – MEM – WB
add	$3,	$5,	$9	IF – ID – Ex – MEM – WB

ID.Ex.Reg.Rs / Rt = Mem.WB.Reg.Rd

Chapter 03

Question Solve

Spring 2023

1(a)

$X = 0.321$

$Y = 0.532$

$X = 0.321$

$= 0.01010010001011 \times 2^{0}$

$= 1.01001000101100000000000000000000 \times 2^{-2}$

Normalized

Biased Exponent

$-2+127 = 125$

$Y = 0.532$

$= 0.10001000011100$

$= 1.00010000011000000000000000000000 \times 2^{-1}$

Biased Exponent $= -1+127 = 126$

$X * Y$

$= (1.01001000101100 \ldots\ldots 0) \times 2^{125} \times (1.00010000110 \ldots\ldots 0) \times 2^{126}$

$= (1.01001000101100 \times 1.00010000110) \times 2^{251}$

$= (1.01011101101101001000001) \times 2^{251}$

$= 58 \cdot 0.170772$

0.532
×2
1.064
×2
0.128
×2
0.256
×2
0.512
×2
1.024
×2
0.048
×2
0.096
×2
0.192
×2
0.384
×2
0.768
×2
1.536
×2
1.072
×2
0.144
×2
0.288
×2
0.576
×2
1.152
×2
0.304
×2
0.608
×2
1.216
×2
0.432
×2
0.864
×2
1.728
×2
1.456
×2
0.912
×2
1.824
×2
1.648
×2
1.296
×2
0.592
×2
1.184
×2
0.368
×2
0.736
×2
1.472
×2
0.944
×2
1.888
×2
1.776
×2
1.552
×2
1.104
×2
0.208
×2
0.416
×2
0.832
×2
1.664
×2
1.328
×2
0.656
×2
1.312
×2
0.624
×2
1.248
×2
0.496
×2
0.992
×2
1.984
×2
1.968
×2
1.936
×2
1.872
×2
1.744
×2
1.488
×2
0.976
×2
1.952
×2
1.904
×2
1.808
×2
1.616
×2
1.232
×2
0.464
×2
0.928
×2
1.856
×2
1.712
×2
1.424
×2
0.848
×2
1.696
×2
1.392
×2
0.784
×2
1.568
×2
1.136
×2
0.272
×2
0.544
×2
1.088
×2
0.176
×2
0.352
×2
0.704
×2
1.408
×2
0.816
×2
1.632
×2
1.264

```
   1.01001000 1011
 x 1.0101000001 10
 ------------------
   1010100010 1011
 1010100010 1011x
                xx
               xxx
              xxxx
             xxxxx
            xxxxxx
           xxxxxxx
          1
   1010010001011xx xxxxxx
 1010010001011xxxxxxxxxxxx
 -------------------------
 1.01011101101100100001
```

There is no overflow or underflow as the exponent is in range and the output is normalized

b) -39.412

20 bit

Sign bit = 1

Exponent = 5 bit

Fraction $= 20-(5+1) = 14$ bit.

39.412

$= (100111.011010010 11)\times 2^0$

$= (1.0011101101 0010)\times 2^5$

Biased Exponent $= 2^{n-1}-1$

$= 2^{5-1}-1 = 15$

$\therefore$ Exponent $= (15+5) = 20 =$

1	10100	00111011010010

0D0DOED2

.412
×2
0.824
×2
1.648
×2
1.296
×2
0.592
×2
1.184
×2
0.368
×2
0.736
×2
1.472
×2
0.944
×2
1.888
×2
1.770

(c)

(a) True

(b) False

Q(b)

```
lw $t0, 20($s0)   A[5]
lw $t1, 24($s0)
beq $t0, $t1, Exit
lw $t0, 12($s0)
lw $t1, 8($s0)
sub $t2, $t0, $t1
addi $t0, $t2, -10
sw $t0, 20($2 $s2)

Exit:
```

if(A[5] != A[6])

f[5] = A[3] - A[2] - 10

A → $s0

f → $s4

for(i = 0 i<n , i++)

A[i] += 1

i → $1

n → $s2

~~i = 0~~

```
add $s1, $zero, $zero
Loop:
  slt $t0, $s1, $s2
  beq $t0, $zero, Exit
  sll $t0, $s1, 2      i×4
  add $t0, $s0, $t0    A+i×B
  lw $t1, 0($t0)       address
  ~~addi $t2, 5~~
  ~~sw $t2 0($t1)~~
  addi $t0, $t1, 1     A[i]+1
  sw $t0, 0($t0)
```

Fall 22

1a $X = 12AB000$

$Z = 444A0000$

$Y = 234B0000$

$X = 12AB000$

$= \underbrace{0}_{1}\underbrace{00000010}_{8}\underbrace{10101011000000000000}_{23}$

~~$X = 1.10101011000000000$~~

$X = 1.01010101011000000000000000 \times 2^{2}$

$Y = 234B0000$

$= \underbrace{0}_{1}\underbrace{01000110}_{8}\underbrace{1001011000000000000000}_{23}$

$Y = 1.10010110000000000000000000 \times 2^{70}$

exponent $70-127=-57$

$Z = 444A000$

$= \underbrace{0}_{1}\underbrace{10001000}_{8}\underbrace{10010100000000000000000}_{23}$

$Z = 1.10010100000000000000000000 \times 2^{136}$ exponent = ~~127~~

$136-127=9$

$X = 12AB0000$

$= \underbrace{0}_{1}\underbrace{00010010}_{8}\underbrace{10101011000000000000000}_{23}$

Ex $1.10101011000000000000000 \times 2^{18}$

Exponent $= 18 + 127 =$
-109

Biased Exponent

$X + Y + Z$

$= (1.10101011 0 \ldots 0 \times 2^{109} + 1.1001011 \times 2^{-57}$

$1.10010100 \ldots 0 \times 2^{9}) +$

-109

9

$= 0.(0 \ldots 0)11010101 10 \times 2^{9}$ $\quad 100$

$+ 0.(0 \ldots 0)11001011 \times 2^{9}$

$+ 1.10010100 \times 2^{9}$

$=$

1b

int(X) = float(Y) + $\boxed{\text{int}\ 3 * Z}$

float(Y) = int(X) + A[4]

X → $s0

Z → $s1

A → $t5

Y → $f0

```
sll $t0, $s1, 2
sub $t0, $t0, $s1
mtc1 $f1, $t0
cvt.s.w $f1, 0($f1)
add.s $f2, $f0, $f1      float(Y) + float(int 3*Z).
cvt.w.s $f2, 0($f2)
mfc1 $s0, $f2
lw $t0, 16($t5)
add $t1, $t0, $s0
mtc1 $f1, $t1
cvt.s.w $f0, 0($f1)
```

1 d

A → $S0
E → $S4
i → $S5
N → $S6
X → $S7

X = A[B[C[D[E[i]]]]]

X = [ ] + zero

add $S5, $zero, $zero

loop
~~beq~~
slt $t0 $S5, $S6
beq $t0, $zero, Exit
sll $t0, $S5, 2
add $t1, $t0, $S4
lw $t0, 0($t1) E[i]
sll $t1, $t0, 2
add $t1, $t1, $S3
lw $t2, 0($t1) D[E[i]]
sll $t3, $t2, 2
add $t3, $t3, $S2
lw $t0, 0($t3)
sll $t0, $t0, 2
add $t2, $t1, $S1

lw $t1, 0($t2) B[C[D[E[i]]]]
sll $t0, $t1, 2
add $t1, $t0, $0
~~lw $~~
~~sw $t2, 0($t1)~~
lw $t0, 0($t1)
add $S7, $t0, $zero
addi $S5, $S5, 1
j loop
Exit:
C[D[E[i]]]

2(b)

```
lui $t1, 0xAA11
ori $t1, $t1, 0x2233
sw $t0, 0($t1)
```

lw $10, 40($11)

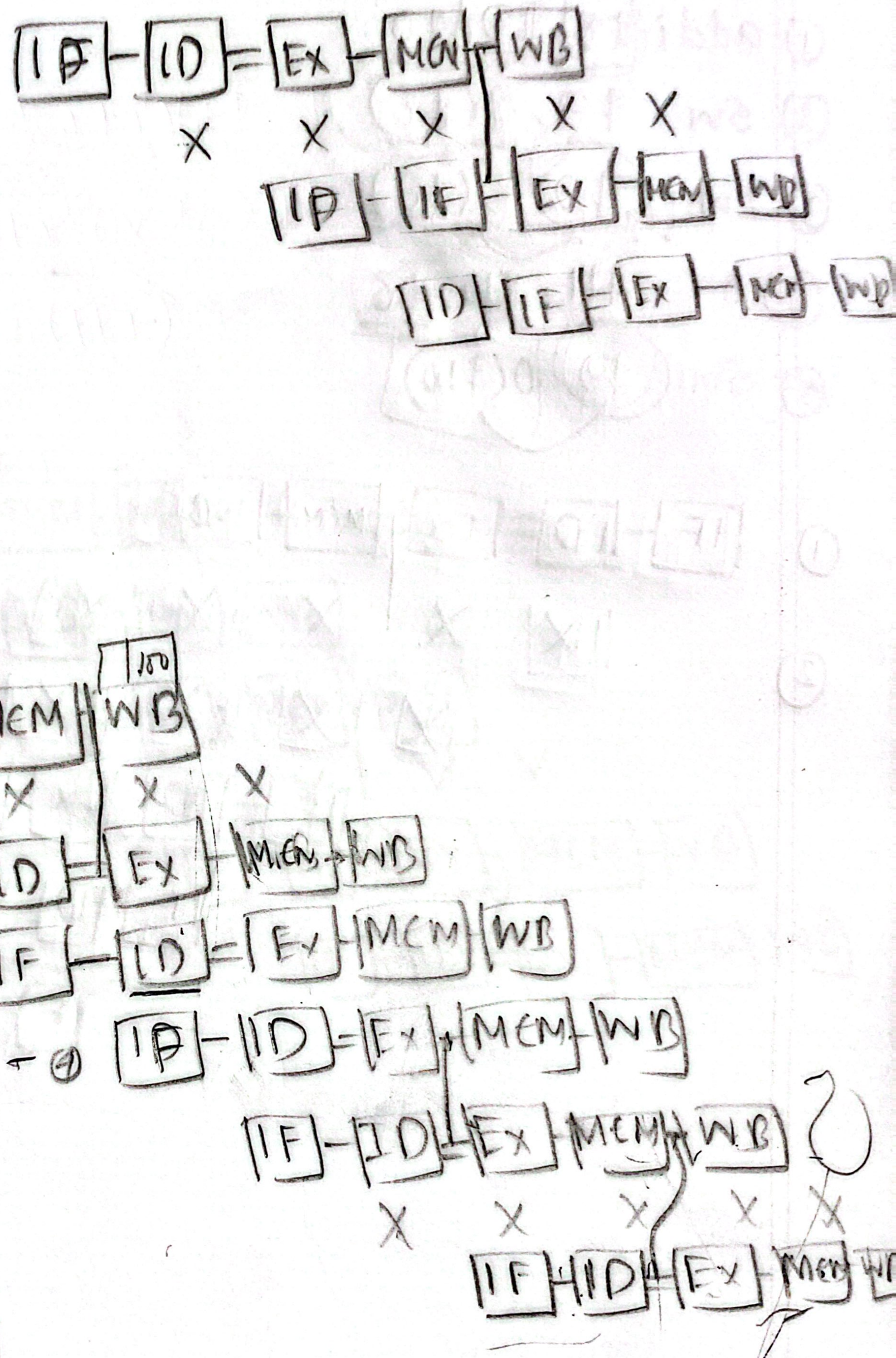
add $5, $10, $7

sub $9, $7, $1

sll $5, $5, 3

lw $13, 18($15)

sw $13, 32($5)



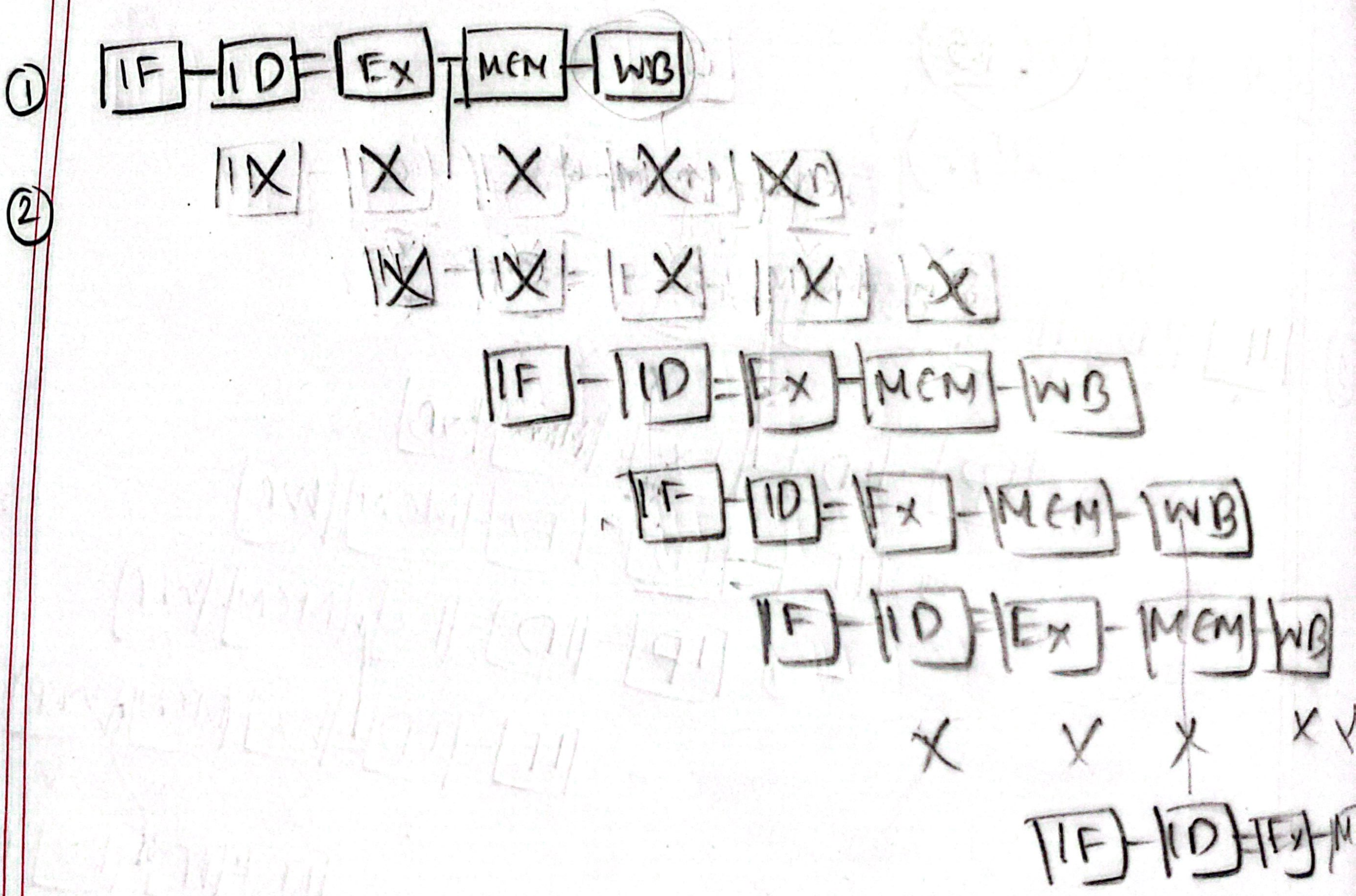
① addi $8, $9, 10

② sw $9, 1($8)

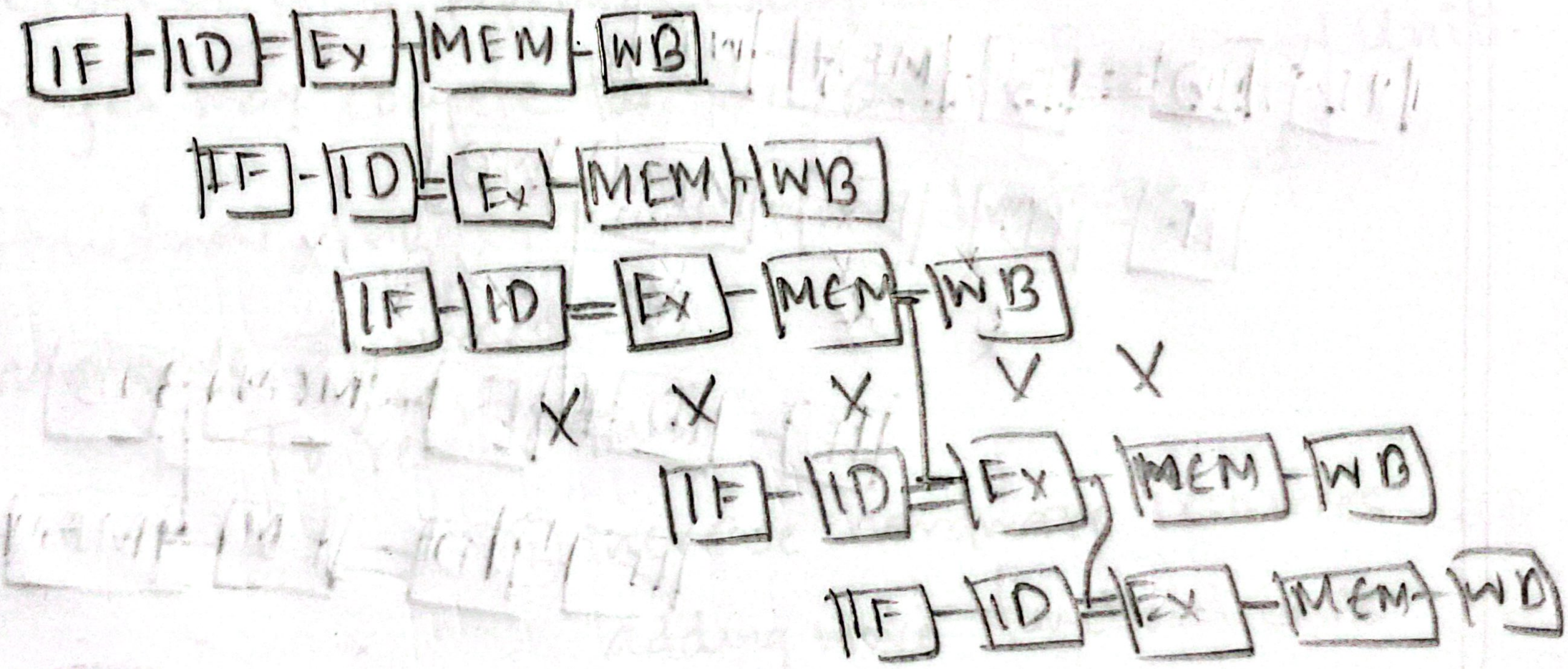
③ lw $9, 6($8)

④ or $11, $10, $8

⑤ sw $9, 0($10)



Sub $8, $9, $10
Sw $8, 4($9)
Lw $8, 20($11)
~~And~~
Addi $12, $8, 11
And $14, $12, $8



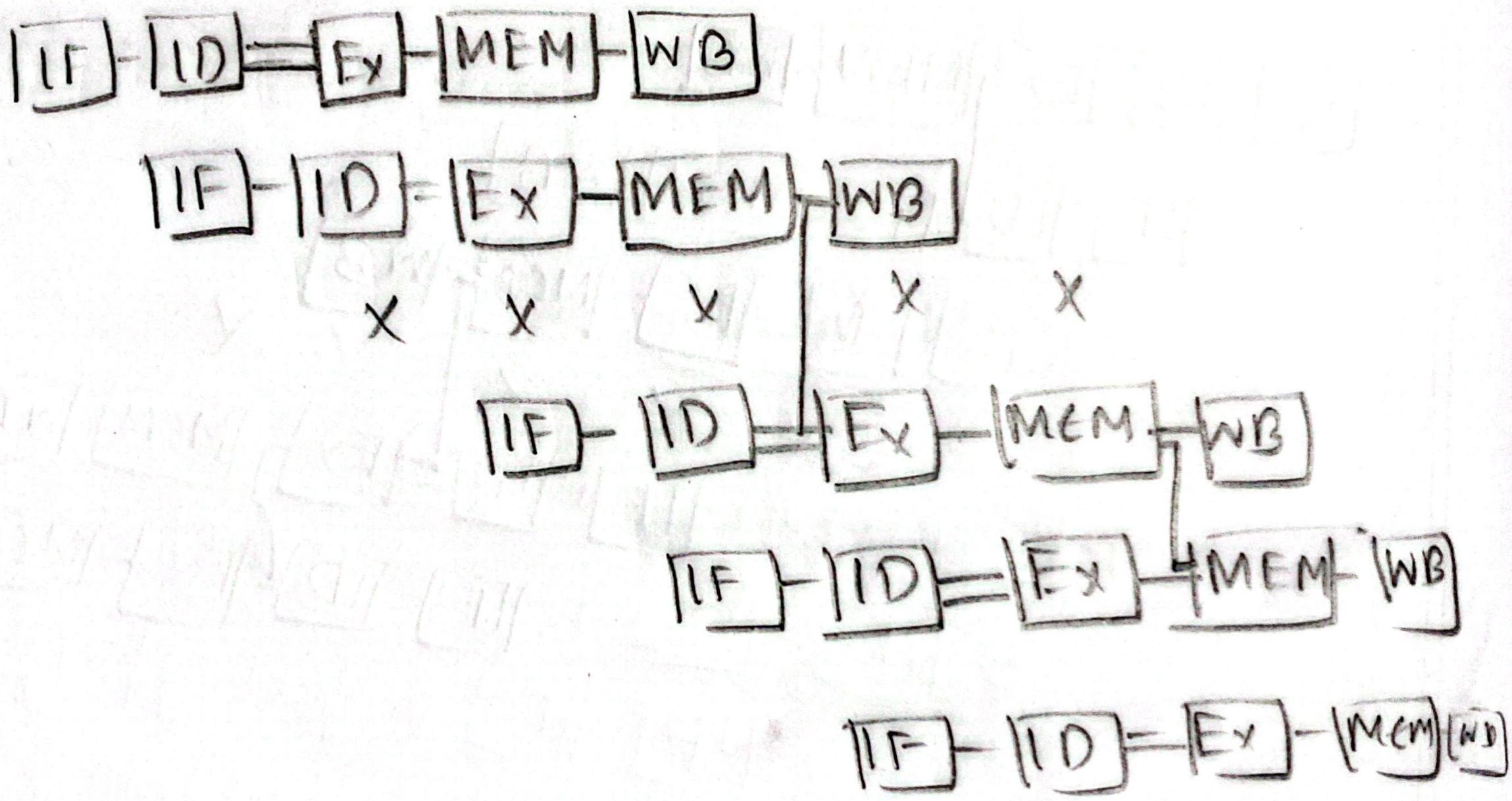
and $9, $8, $10

lw $8, 4($9)

and $11, $8, $12

sw $11, $8(9)

Addi $12, $14, 6



(1) Why structural hazards are easier to overcome:

Structural hazards mean that there aren't enough hardware resources as it sometimes occurs that there is a conflict of using a particular hardware resources in a pipeline. Example: multiple pipeline stages may compete for the same functional unit. Structural hazard occur from limitation of hardware resources. So,

To remove this.

(1) increase hardware resources adding more functional units

(10)