Computer Architecture (CS F342)

Design, Analysis, Execution and Optimization of Instructions

Reduced Instruction Set Computer (RISC): MIPS

Multicycle Datapath and Control unit

Book: COD by Patterson and Hennessy from Chapter **4 A**

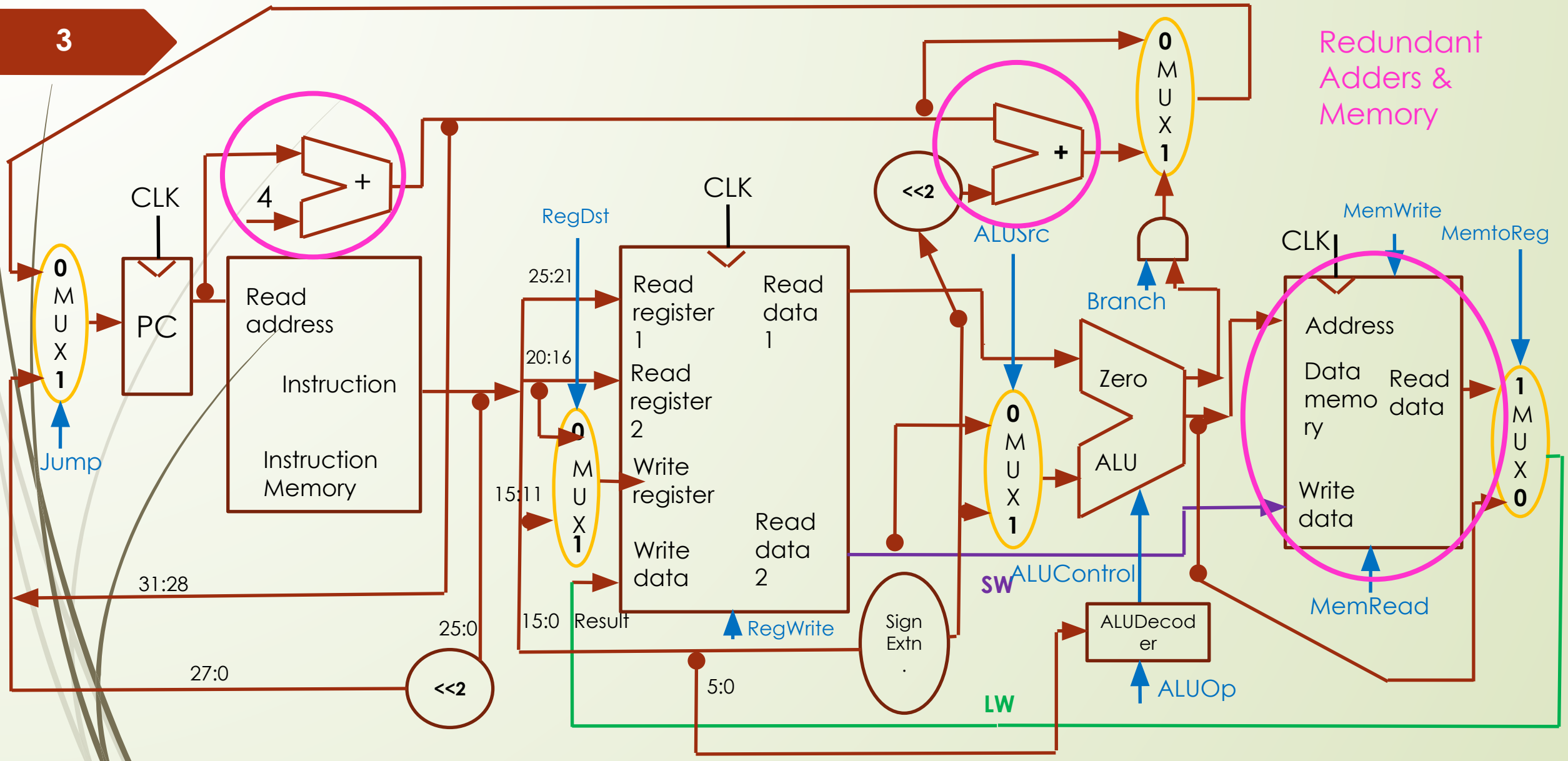
Fetch-and-Execute Algorithm for MIPS Microprocessor (32 bit)

```
#define OPCODE 0b11111100000000000000000000000000
#define RS     0b00000011111000000000000000000000
#define RT     0b00000000000111110000000000000000
#define RD     0b00000000000000001111100000000000
#define SHIFT  0b00000000000000000000011111000000
#define OFFSET 0b00000000000000001111111111111111

int PC, IMM[1024], DMM[1024], RF[32], ALUControl; bool ZERO;
Load(IMM, DMM);
Set PC with address 1st instruction which is stored in IMM;

  while (1){
    switch((IMM[PC] & OPCODE) >>26){
      case R-type:
            Set ALUControl; //ADD, SUB, AND, OR, etc
             RF[(IMM[PC] & RD)>>11] = ALU(RF[(IMM[PC] & RS)>>21], RF[(IMM[PC] & RT) >>16]); PC = PC + 4;
      case SW-type:
            Set ALUControl = 0b0010;
           DMM[ALU((IMM[PC] & RS)>>21, (IMM[PC] & OFFSET) ] = RF[(IMM[PC] & RT)>>16]; PC = PC + 4;
       case LW-type:
            Set ALUControl = 0b0010;
           RF[IMM[PC] & RT]= DMM[ALU((IMM[PC] & RS) >>21, IMM[PC] & OFFSET) ]; PC = PC + 4;
        case B-type:
            Set ALUControl = 0b0110;
            ALU(RF[(IMM[PC] & RS)>>21], RF[(IMM[PC] & RT) >>16]);
           IF (ZERO ==1) PC = (PC + 4) + ((IMM[PC] & OFFSET) <<2); ELSE PC = PC + 4;
    }
  }
```

```
ALU(Src1, Src2){
      switch (ALUControl){
          case B-type: ZERO = (Src1- Src2) == 0 ? 1: 0;
                       break;
          case ADD: return (Src1 + Src2); break;
            …
      }
}
```

Need conversion for 16 bit offset to 32 bits

Single-cycle datapath

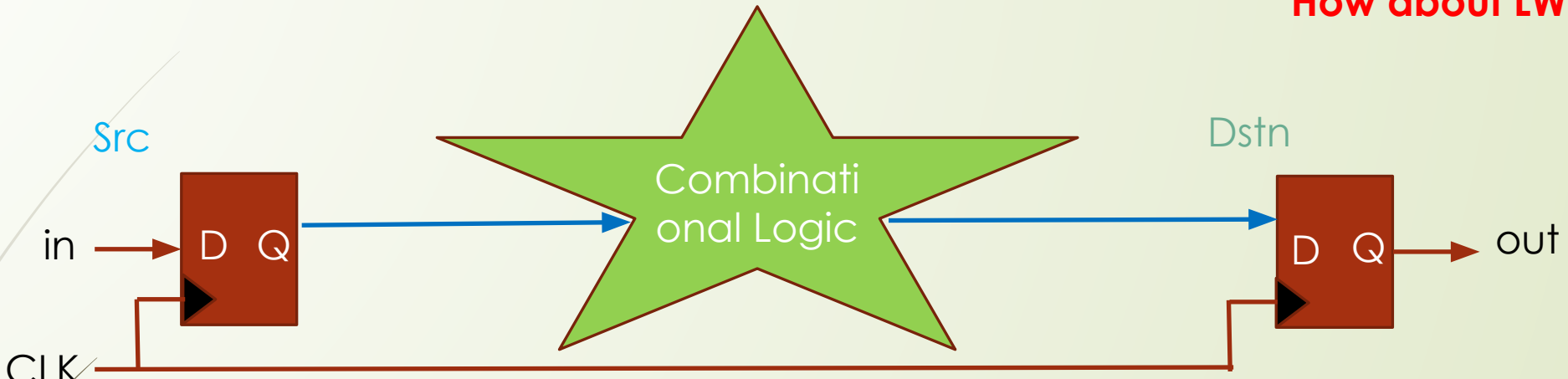


Performance analysis

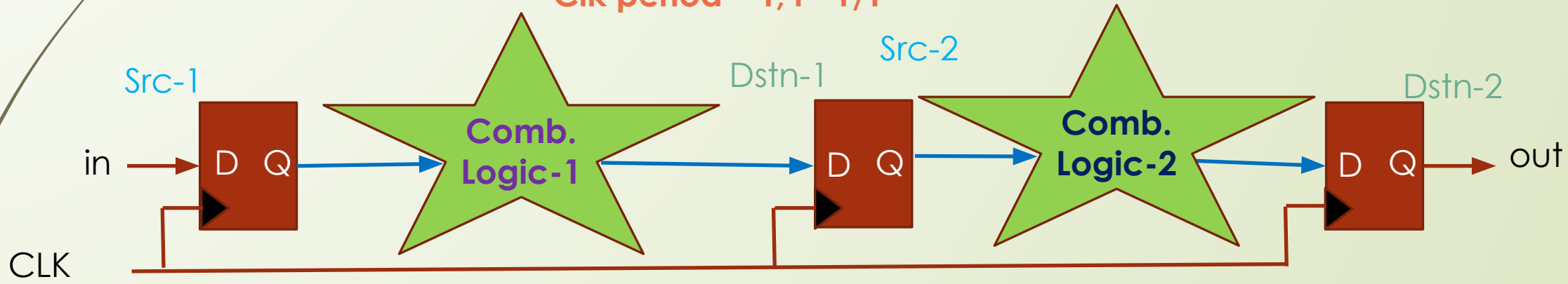
- The execution time of a program is a metric
  - $Execution\ Time = (\#instructions)\left(\frac{Cycles}{instruction}\right)\left(\frac{Seconds}{cycle}\right)$
- #instructions or length of a program depends on ISA
- How did we measure the clock period?
- Which instruction was dominating the clock period?
- **How can we reduce the clock period or improve the frequency (f=1/T)?**

Improvement on clock period

How about LW?



Clk period = T, f =1/T



T =Max{ (src-1, dstn-1), (src-2, dstn-2) }

Clk period = T/2, f = 2/T

This is not the pipeline structure

Combinational Logic = Combinational Logic-1 U Combinational Logic-2

Placement of Registers/ Partition the logic

- How does one place the registers in the datapath?
- T = Max{ (src-1, dstn-1), (src-2, dstn-2), …}
- What Max{ } is telling us?
- Each partition can take almost equal time for computation

Problems of Single-cycle Datapath Design

- Single-cycle design works well but inefficient design
- Clock length (worst-case delay) is same for all instructions
- It is not a **balanced** design
- CPI is 1
- Use more resources:
  - Adder
  - Memory
- Necessity of balanced datapath design technique by focusing on common-case design & analysis principle

Balanced datapath design: Multi-cycle approach

- Instruction execution can be broken down to smaller steps
- Simple instruction can complete the execution earlier than the complex instructions
- One can design the multi-cycle datapath as similar in single-cycle
  - Connecting architectural elements with **the storage** using combinational logic
  - Next, design the controller

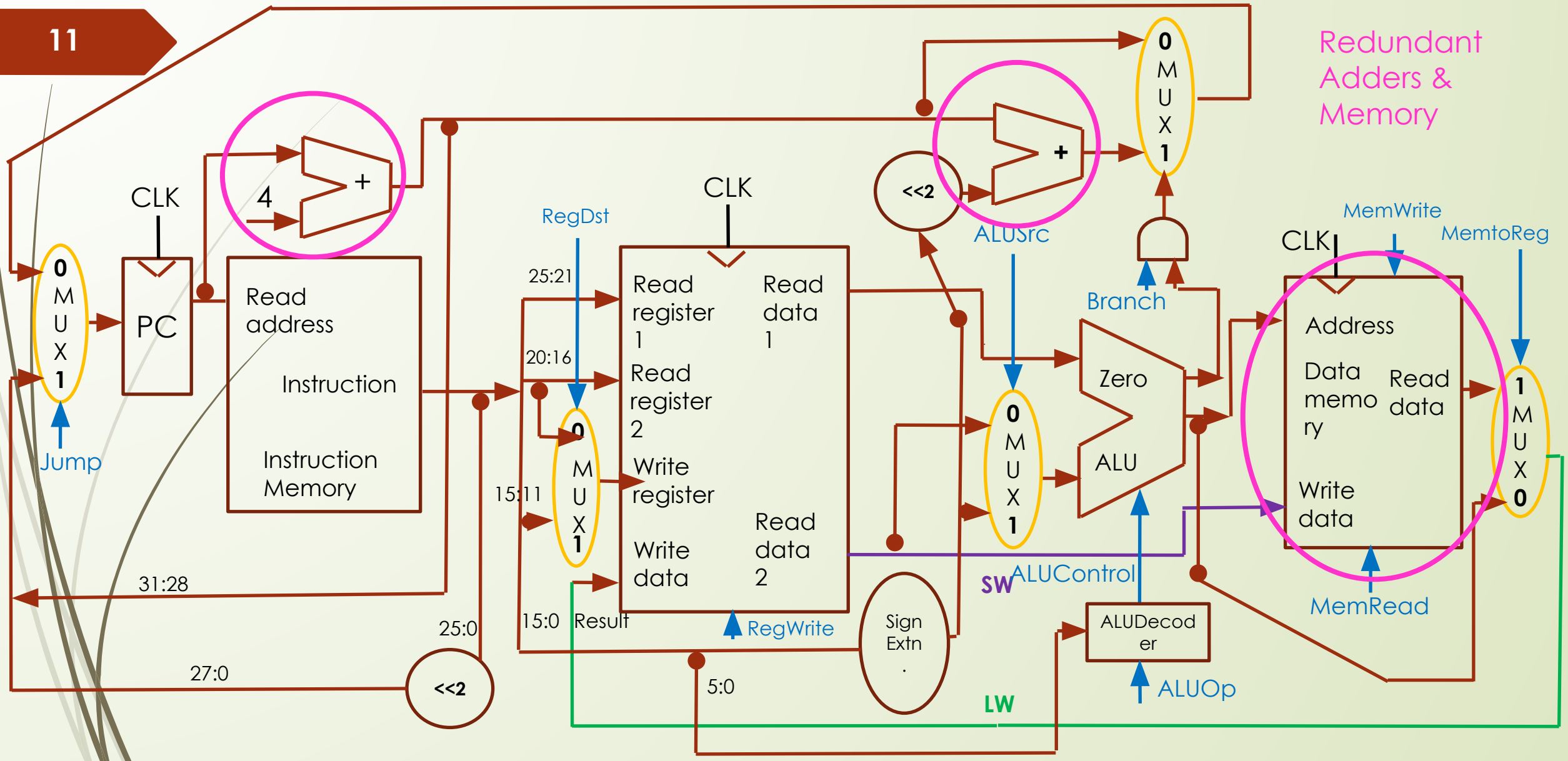
Balanced datapath design: Multi-cycle approach

- The key difference is
  - Controller produces different signals on different steps/states
  - A finite state machine (FSM) approach

Balanced datapath design: Multi-cycle approach

- Combined the instruction and data memory
- Remove the redundant adders
- Incorporate the **non-architectural state/storage** elements
  - Not visible to the programmer

Single-cycle datapath



Balanced datapath design: Multi-cycle approach

- Remove the redundant adders & memory
- Where do we place this operations?
- How does one control such operations?

Fetch-and-Execute Algorithm for MIPS Microprocessor (32 bit)

```
#define OPCODE 0b11111100000000000000000000000000
#define RS     0b00000011111000000000000000000000
#define RT     0b00000000000111110000000000000000
#define RD     0b00000000000000001111100000000000
#define SHIFT  0b00000000000000000000011111000000
#define OFFSET 0b00000000000000001111111111111111

int PC, IMM[1024], DMM[1024], RF[32], ALUControl; bool ZERO;
Load(IMM, DMM);
Set PC with address 1st instruction which is stored in IMM;

  while (1){
    switch((IMM[PC] & OPCODE) >>26){
      case R-type:
            Set ALUControl; //ADD, SUB, AND, OR, etc
            RF[(IMM[PC] & RD)>>11] = ALU(RF[(IMM[PC] & RS)>>21], RF[(IMM[PC] & RT) >>16]); PC = PC + 4;
      case SW-type:
            Set ALUControl = 0b0010;
            DMM[ALU((IMM[PC] & RS)>>21, (IMM[PC] & OFFSET) ] = RF[(IMM[PC] & RT)>>16]; PC = PC + 4;
       case LW-type:
            Set ALUControl = 0b0010;
            RF[IMM[PC] & RT]= DMM[ALU((IMM[PC] & RS) >>21, IMM[PC] & OFFSET) ]; PC = PC + 4;
        case B-type:
            Set ALUControl = 0b0110;
            ALU(RF[(IMM[PC] & RS)>>21], RF[(IMM[PC] & RT) >>16]);
            IF (ZERO ==1) PC = (PC + 4) + ((IMM[PC] & OFFSET) <<2); ELSE PC = PC + 4;
    }
  }
```

```
ALU(Src1, Src2){
      switch (ALUControl){
          case B-type: ZERO = (Src1- Src2) == 0 ? 1: 0;
                       break;
          case ADD: return (Src1 + Src2); break;
            …
      }
}
```

Need conversion for 16 bit offset to 32 bits

Key points: Resource Sharing leads to Multi-Cycle Approach

- Datapath components are shared
  - Functional units
  - Memory
  - Interconnect
- Balanced datapath design

Book: COD by Patterson and Hennessy from Chapter **4 A**

Fetch-and-Execute Algorithm for MIPS Microprocessor (32 bit)

```
#define OPCODE 0b11111100000000000000000000000000
#define RS     0b00000011111000000000000000000000
#define RT     0b00000000000111110000000000000000
#define RD     0b00000000000000001111100000000000
#define SHIFT  0b00000000000000000000011111000000
#define OFFSET 0b00000000000000001111111111111111

int PC, MM[1024],  RF[32], ALUControl; bool ZERO;
int IR, A, B, ALUOut, MDR; //non-architectural elements
Load(MM);
Set PC with address 1st instruction which is stored in IMM;
   while (1){ IR = MM[PC]; ALUControl = 0b0010;PC = ALU(PC, 4);
      A = RF[(IR & RS)>>21]; B = RF[(IR & RT) >>16];
      switch((IR & OPCODE) >>26){
         case R-type:
                Set ALUControl; //ADD, SUB, AND, OR, etc
                ALUOut = ALU(A, B); RF[(IR & RD)>>11] = ALUOut;
         case LW-type:
                ALUControl = 0b0010;
                ALUOut = ALU(A, (IR & OFFSET)); MDR = MM[ALUOut];   RF[(IR & RT)>>16]= MDR;
         case SW-type:
                ALUControl = 0b0010;
                ALUOut = ALU(A, (IR & OFFSET)); MM[ALUOut] = B;
          case B-type: ALUControl = 0b0010;
                ALUOut = ALU(PC, (IR & OFFSET) <<2);
                ALUControl = 0b0110; ALU(A, B);
                 IF (ZERO ==1) {PC = ALUOut;}
      }
   }
```

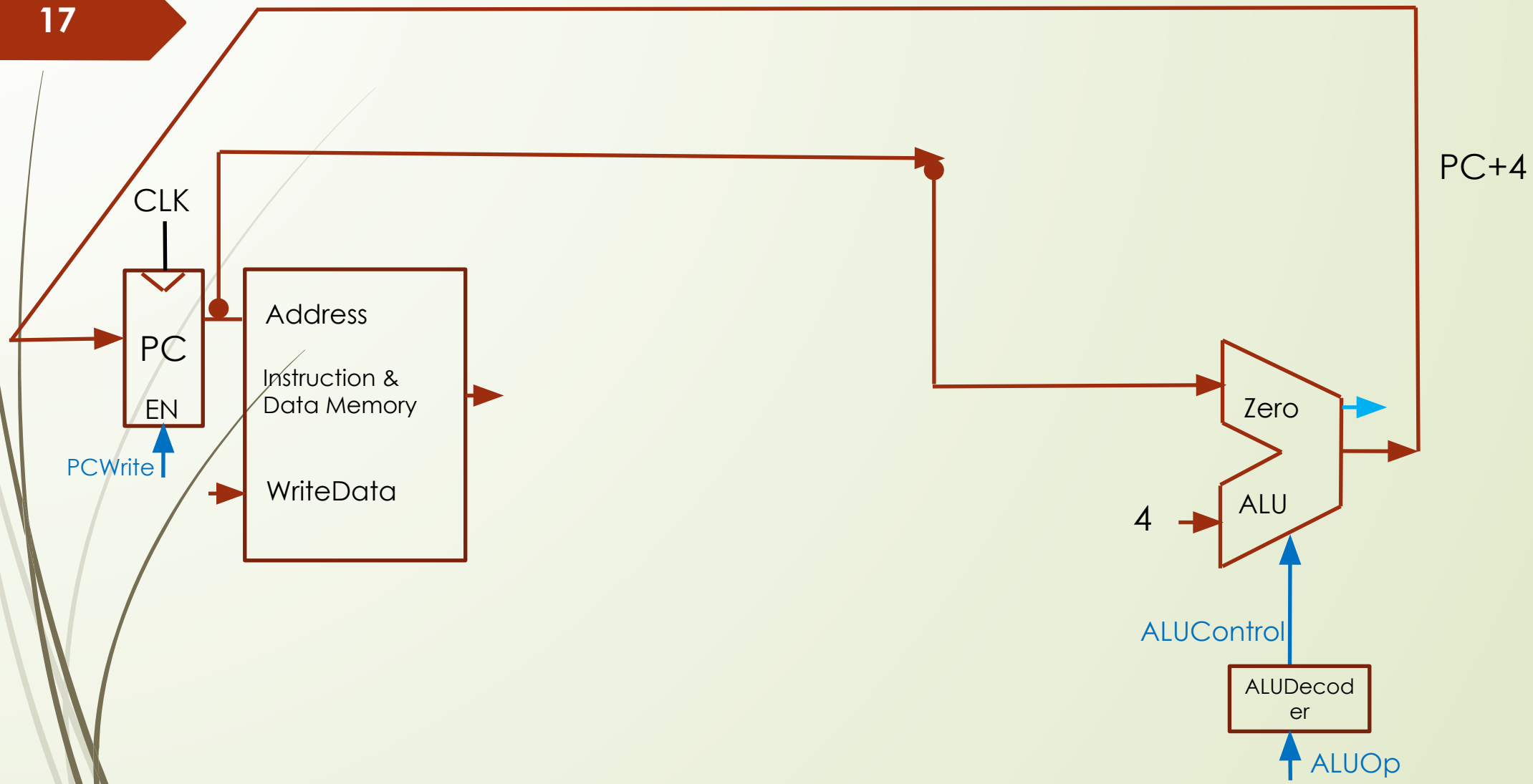
```
ALU(Src1, Src2){
        switch (ALUControl){
           case B-type: ZERO = (Src1- Src2) == 0 ? 1: 0; break;
           case ADD: return (Src1 + Src2);
                …
        }
}
```

Need conversion for 16 bit offset to 32 bits

Multi-Cycle Approach

Single memory unit, **Fetch stage** and datapath for reusing the ALU

Single memory unit, Fetch stage and datapath for reusing the ALU

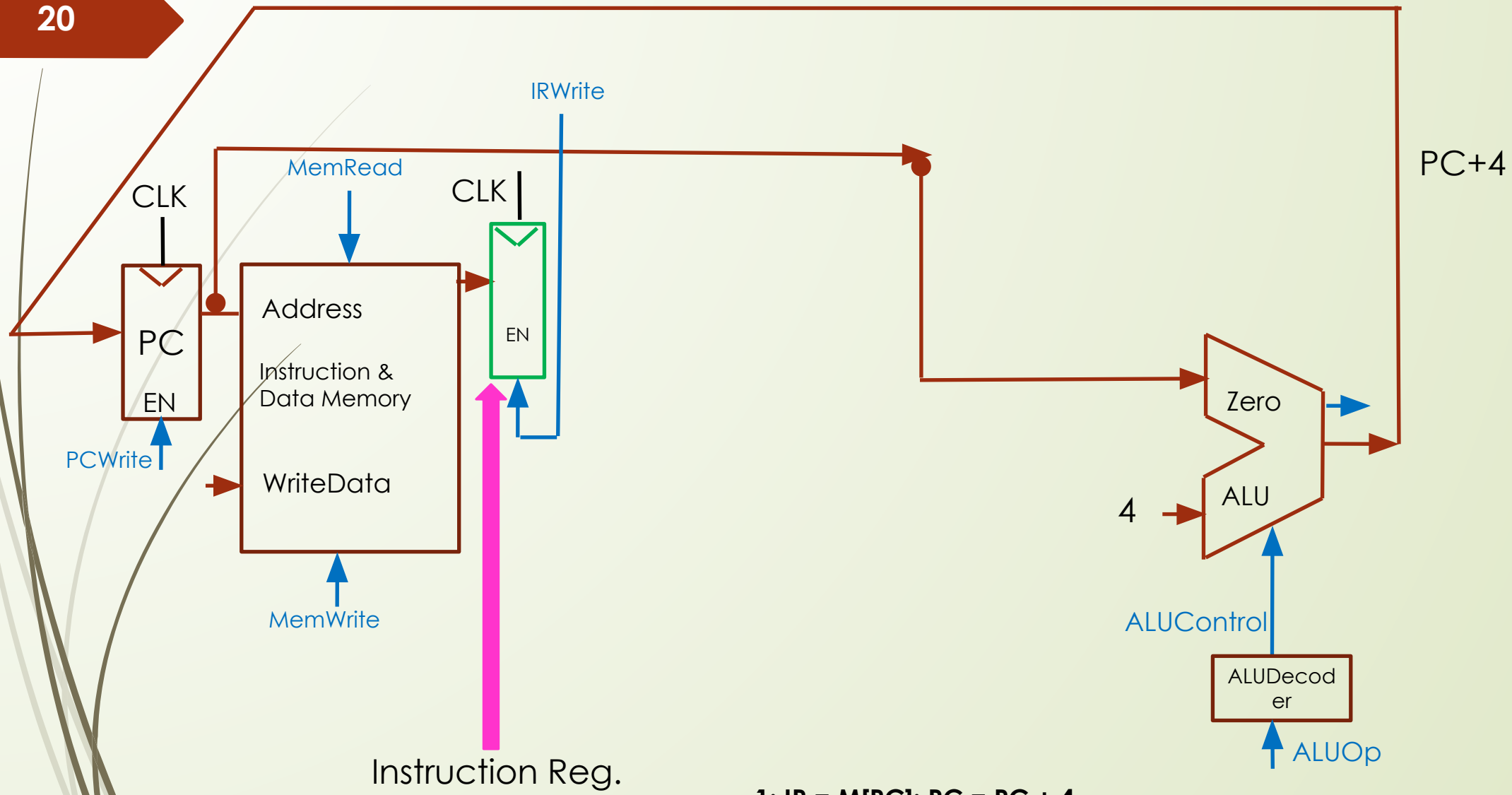


Balanced datapath design: Multi-cycle approach

- Combined the instruction and data memory
- Remove the redundant adders
- Incorporate the **non-architectural state/storage** elements

Fetch stage and **Non-architectural** elements Instruction register

Fetch stage and **Non-architectural** elements Instruction register



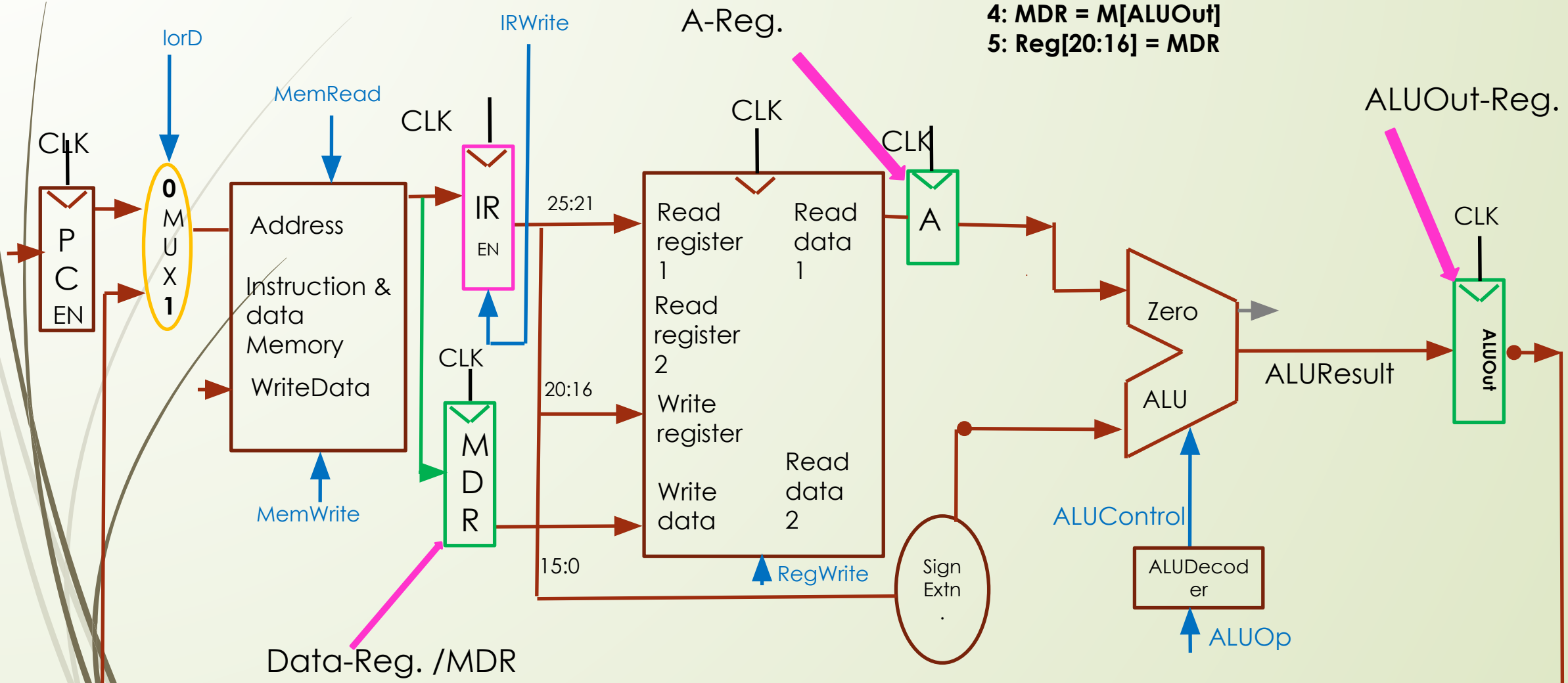
1: IR = M[PC]; PC = PC + 4

lw-instr. and **non-architectural elements**: A, Data and ALUOut-register

1: IR = M[PC]; PC = PC + 4
2: A = Reg[25:21];
3: ALUOut = A + SignExtn(Imm)
4: MDR = M[ALUOut]
5: Reg[20:16] = MDR

lw-instr. and **non-architectural elements**: A, Data and ALUOut-register

1: IR = M[PC]; PC = PC + 4
2: A = Reg[25:21];
3: ALUOut = A + SignExtn(Imm)
4: MDR = M[ALUOut]
5: Reg[20:16] = MDR

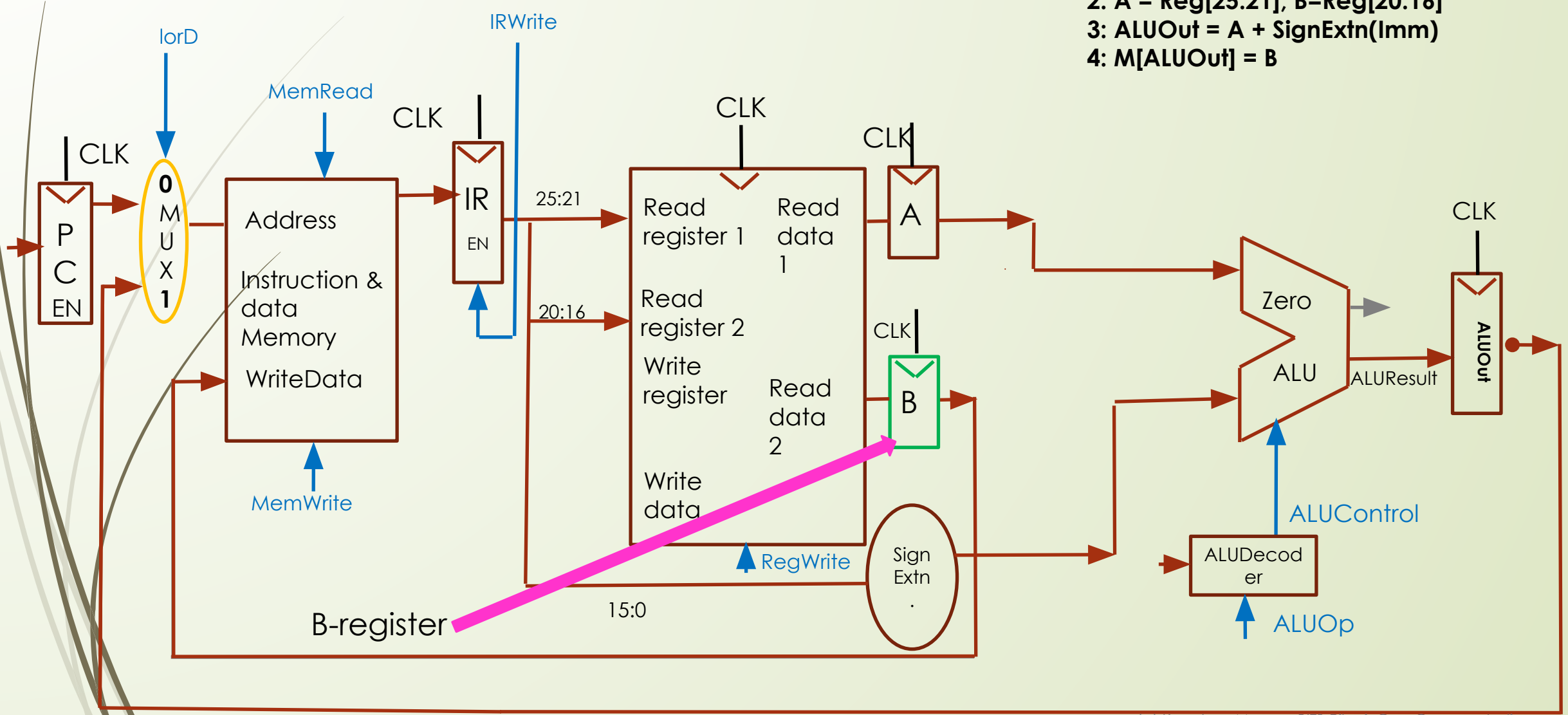


sw-instr and non-architectural elements: B-register

1: IR = M[PC]; PC = PC + 4
2: A = Reg[25:21]; B=Reg[20:16]
3: ALUOut = A + SignExtn(Imm)
4: M[ALUOut] = B

sw-instr and non-architectural elements: B-register

1: IR = M[PC]; PC = PC + 4
2: A = Reg[25:21]; B=Reg[20:16]
3: ALUOut = A + SignExtn(Imm)
4: M[ALUOut] = B

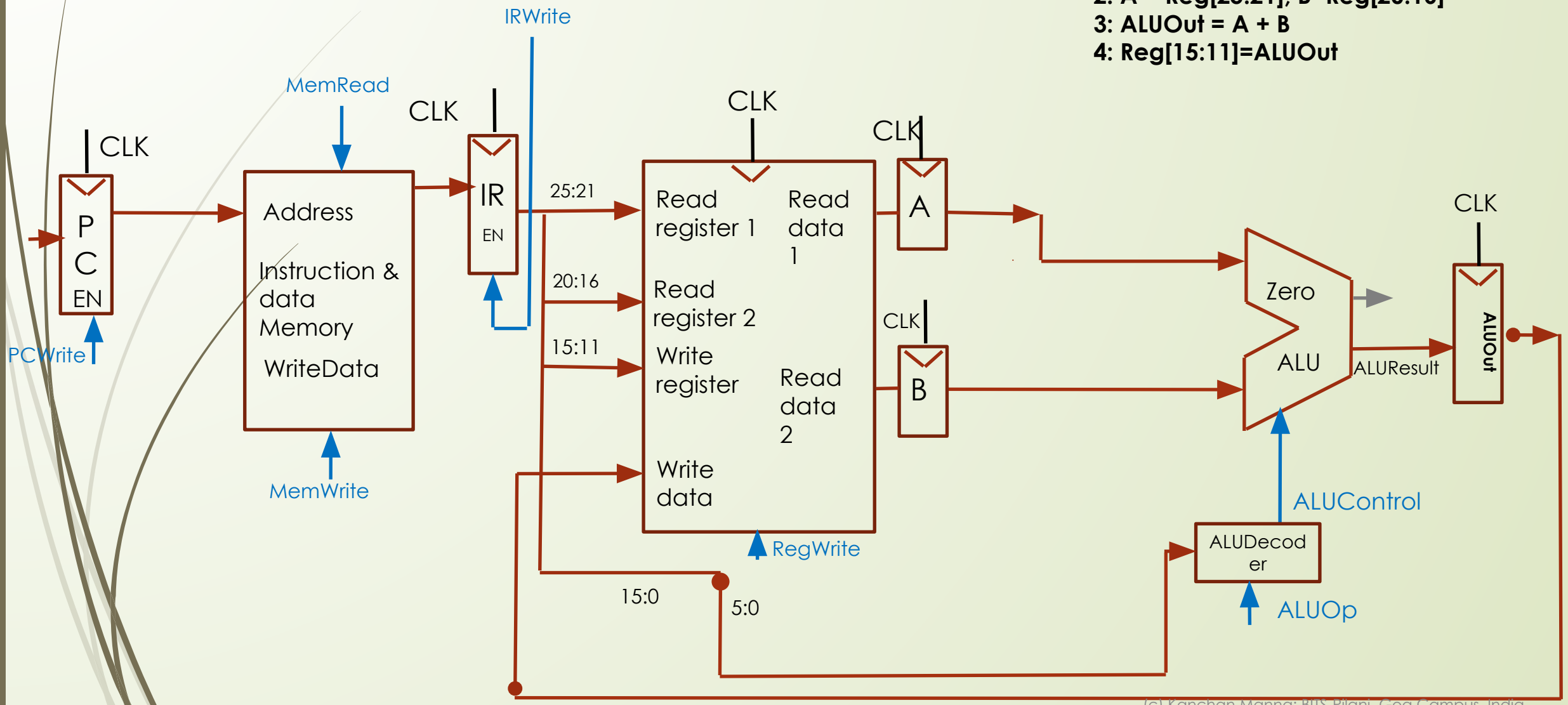


R-type instruction

1: IR = M[PC]; PC = PC + 4
2: A = Reg[25:21]; B=Reg[20:16]
3: ALUOut = A + B
4: Reg[15:11]=ALUOut

R-type instruction

1: IR = M[PC]; PC = PC + 4
2: A = Reg[25:21]; B=Reg[20:16]
3: ALUOut = A + B
4: Reg[15:11]=ALUOut



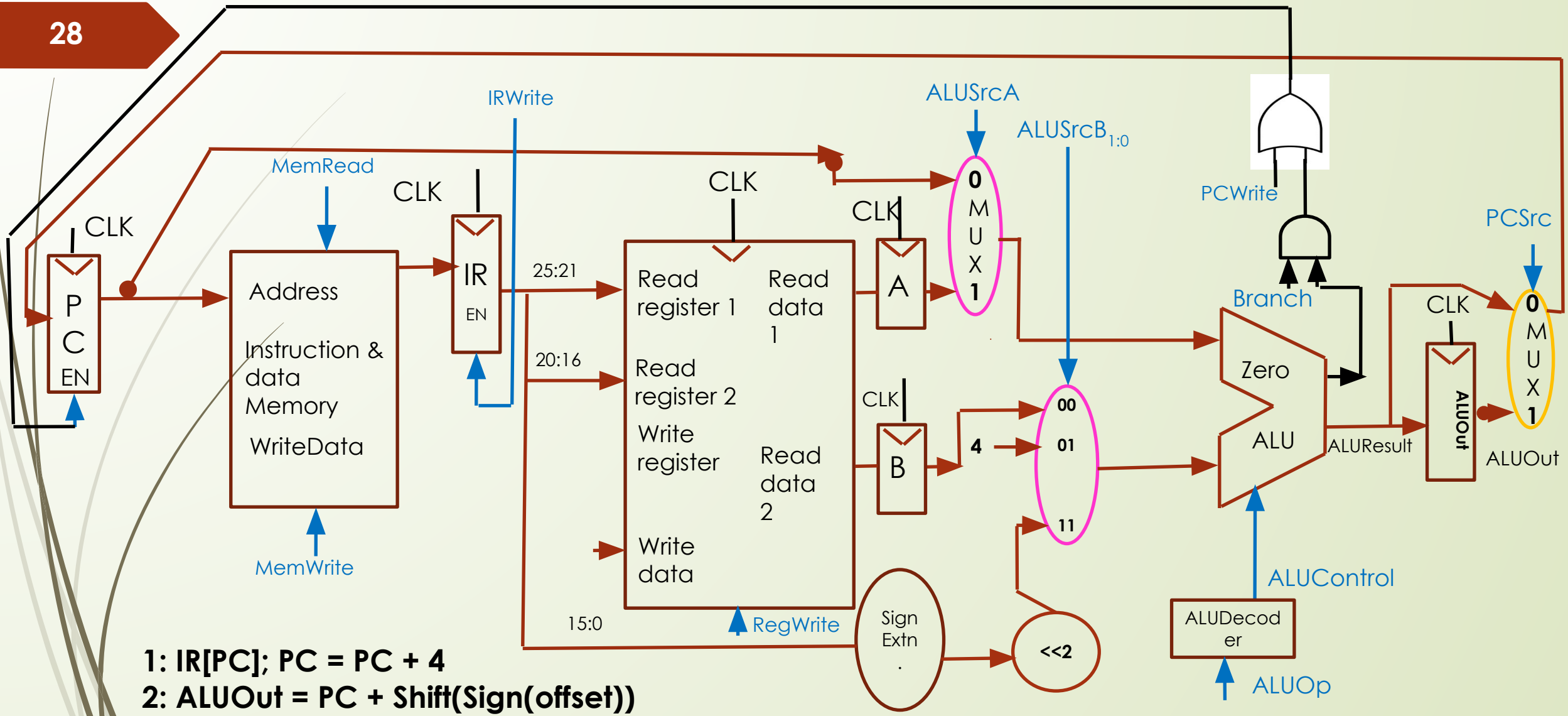
Fetch stage and B(R)-type instruction

BEQ R1, R2, offset; if true, PC =PC+4+offset
else PC = PC+4

1: IR[PC]; PC = PC + 4
2: ALUOut = PC + Shift(Sign(offset))
3: PC = ALUOut if Zero == 1 //A-B

Fetch stage and B(R)-type instruction

BEQ R1, R2, offset; if true, PC =PC+4+offset
else PC = PC+4



1: IR[PC]; PC = PC + 4
2: ALUOut = PC + Shift(Sign(offset))
3: PC = ALUOut if Zero == 1 //A-B

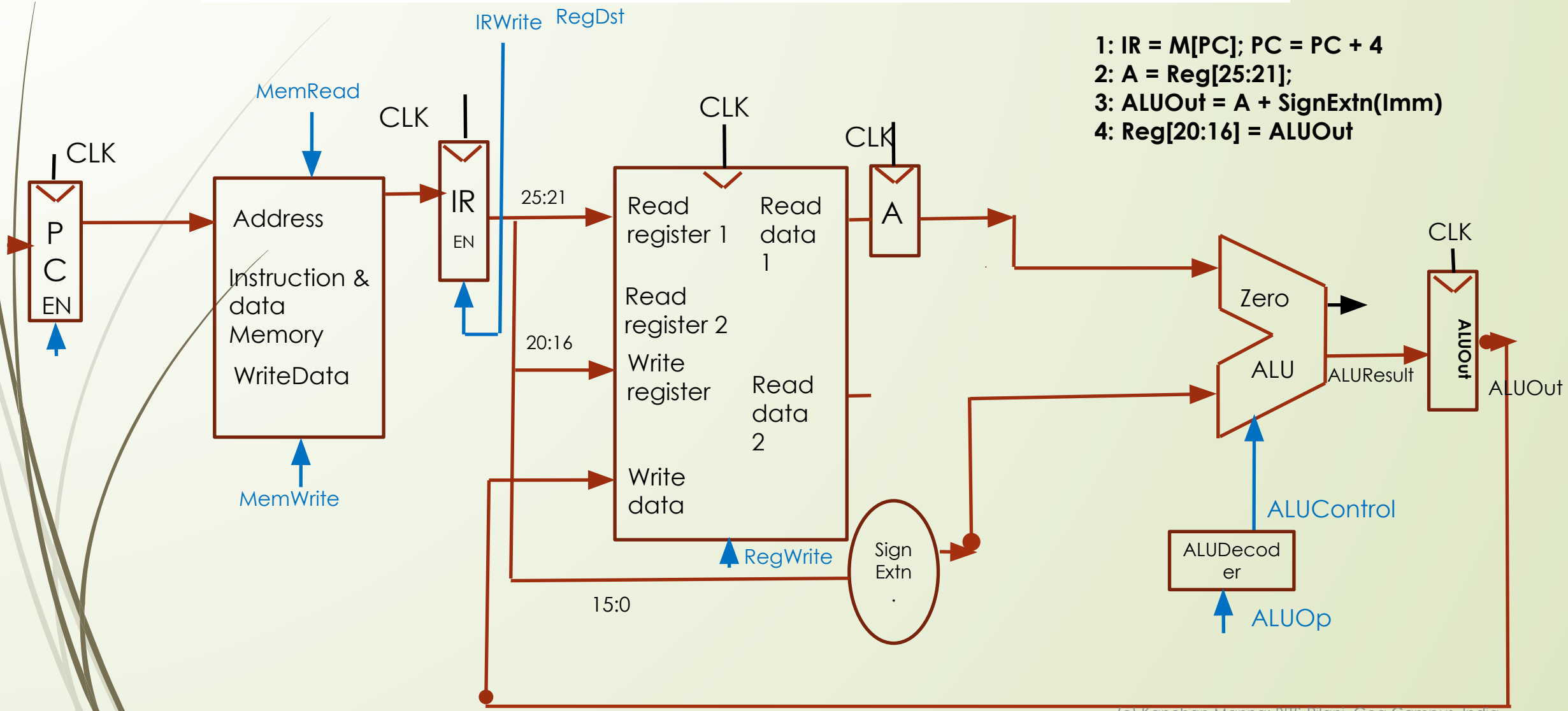
I-type instruction: ADDI

op	rs	rd	Immediate		
6-bits(31-26)	5-bits(25-21)	5-bits(20-16)	5-bits(15-11)	5-bits(10-6)	6-bits (5-0)

1: IR = M[PC]; PC = PC + 4
2: A = Reg[25:21];
3: ALUOut = A + SignExtn(Imm)
4: Reg[20:16] = ALUOut

I-type instruction: ADDI

op	rs	rd	Immediate		
6-bits(31-26)	5-bits(25-21)	5-bits(20-16)	5-bits(15-11)	5-bits(10-6)	6-bits (5-0)



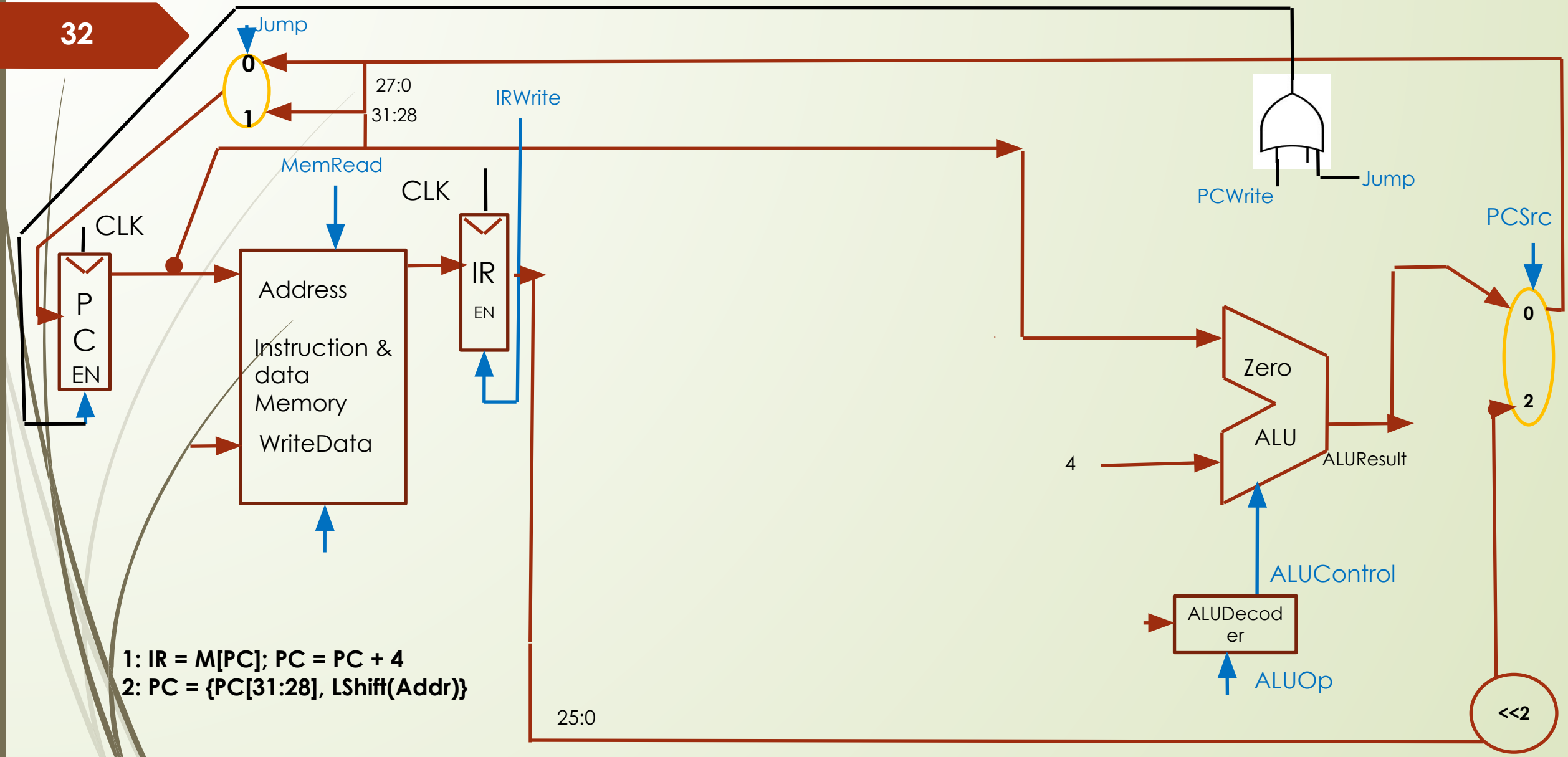
Fetch stage and Jump instruction

PC □ PC+4[31:28] addrs[27:0]

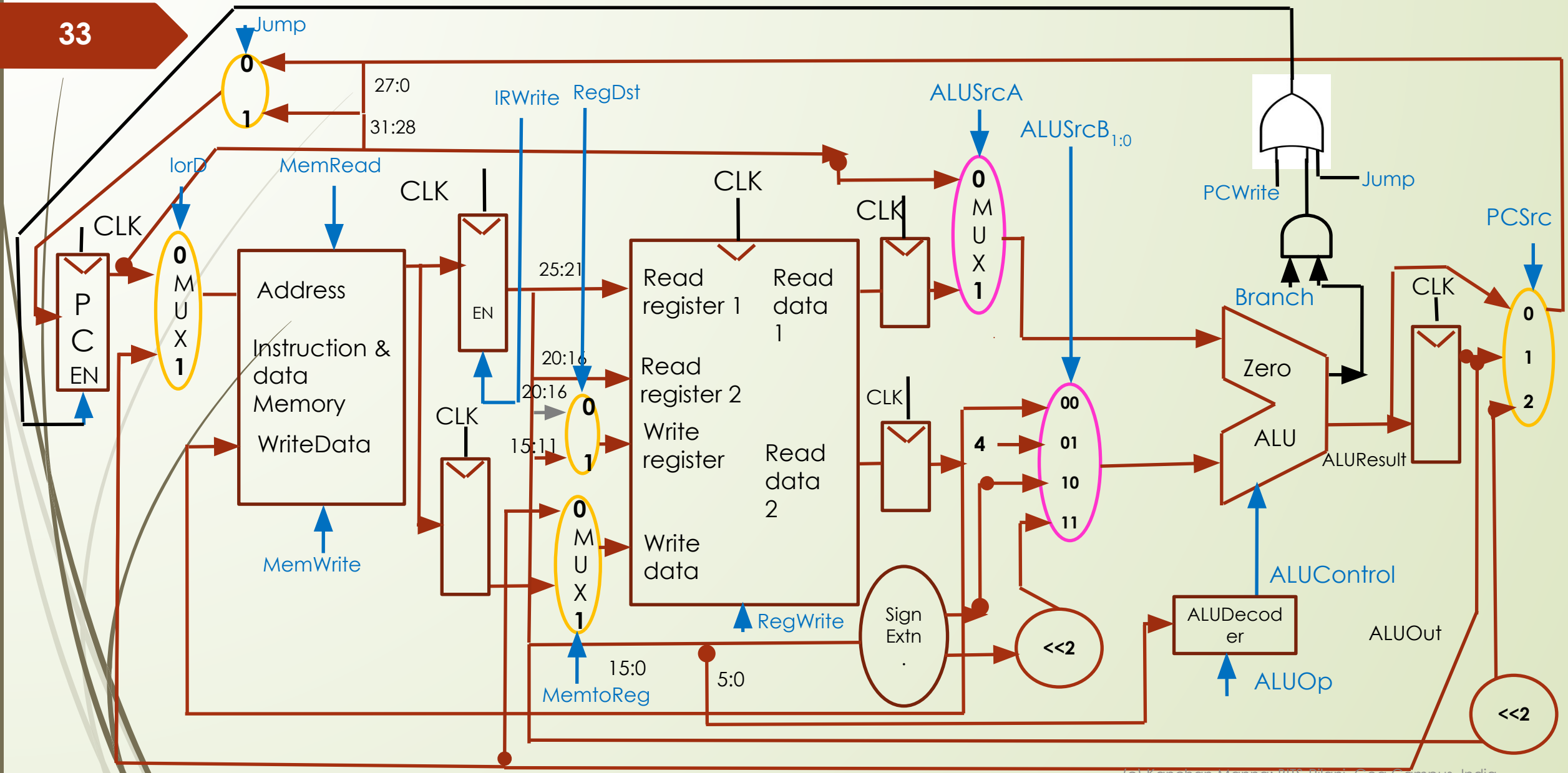
1: IR = M[PC]; PC = PC + 4
2: PC = {PC[31:28], LShift(Addr)}

Fetch stage and Jump instruction

PC □ PC+4[31:28] addrs[27:0]



Combined Datapath of all types of instructions



Control signals

- IorD
- Jump
- Memwrite
- MemRead
- IRWrite
- RegDst
- MemtoReg
- RegWrite
- ALUSrcA
- ALUSrcR1:0
- PCWrite
- Branch
- PCSrc
- ALUOp
- ALUControl

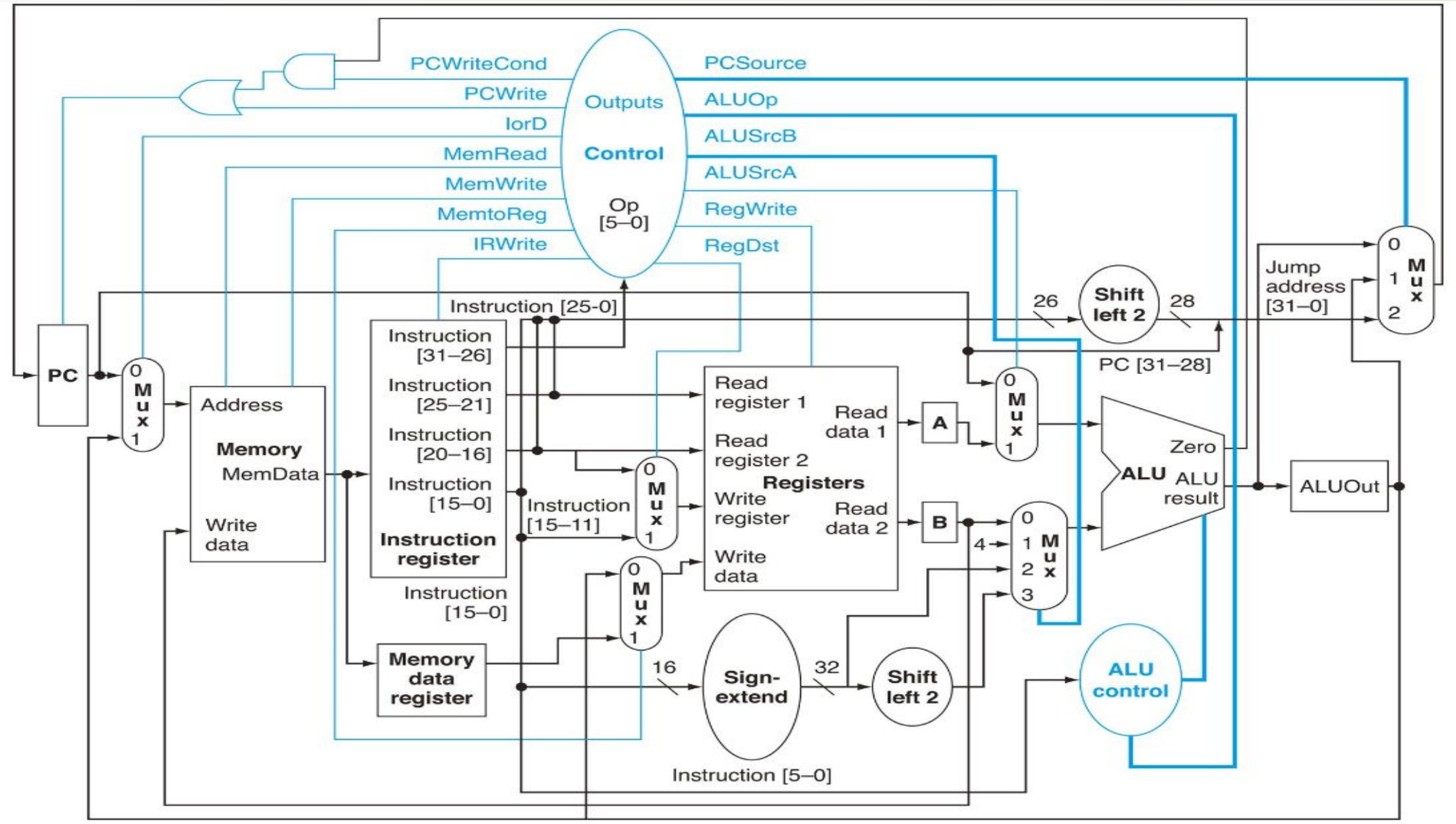
Truth Table?

15 Control signals

- States

How many bits are needed to represent a state?

Multicycle model



Summery of Steps taken to execute any instruction class

Step name	Action for R-type instructions	Action for memory reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/register fetch	A <= Reg [IR[25:21]] B <= Reg [IR[20:16]] ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend (IR[15:0])	if (A == B) PC <= ALUOut	PC <= {PC [31:28], (IR[25:0]],2'b00)}
Memory access or R-type completion	Reg [IR[15:11]] <= ALUOut	Load: MDR <= Memory[ALUOut] or Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Chapter 4 of COD by P&H

Fetch stage

Machine state	Operation	Control signals	Next state
T0	InsR □ M[PC]; PC □PC+4	IorD=0, IRWrite=1, MemRead=1 ALUSrcA=0, ALUSrcB=01, ALUOp=00, PCSrc=00, PCWrite=1, Jump=0	T1

Decode stage

Machine state	Operation	Control signals	Next state
T1	(PC+4) + Lshift(SigExtn(offset)) A = Reg[25:21] B = Reg[20:16]	ALUSrcA=0, $ALUSrcB_{1:0}$ = 11, ALUOp=00	x

Useful for BRZ instruction

LW type instruction

Machine state	Operation	Control signals	Next State
T2	A + sigEx(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00	T3
T3	MDR □M[A+sigEx(off)]	IorD=1 , MemRead=1	T4
T4	RF[dest] □MDR	RegDst=0, MemtoReg=1, RegWrite=1	T0

SW type instruction

Machine state	Operation	Control signals	Next state
T2	A + sigEx(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00	T5
T5	M[A+sigEx(offset)] □ B	IorD=1, MemWrite=1	T0

R-type instruction

Machine state	Operation	Control signals	Next state
T6	A Op B	ALUSrcA=1, $ALUSrcB_{1:0}$ = 00, ALUOp=00	T7
T7	RF[dstn] □ A Op B	RegDst=1, MemtoReg=0, RegWrite=1	T0

B-type instruction

Machine state	Operation	Control signals	Next State
T1	(PC+4) + SigExtn(offset) A = Reg[25:21] B = Reg[20:16]	ALUSrcA=0, $ALUSrcB_{1:0}$ = 11, ALUOp=00, PCSrc=01	T8
T8	A-B	ALUSrcA=1, $ALUSrcB_{1:0}$ = 00, ALUOp=01, Branch=1	T0

Decoding stage

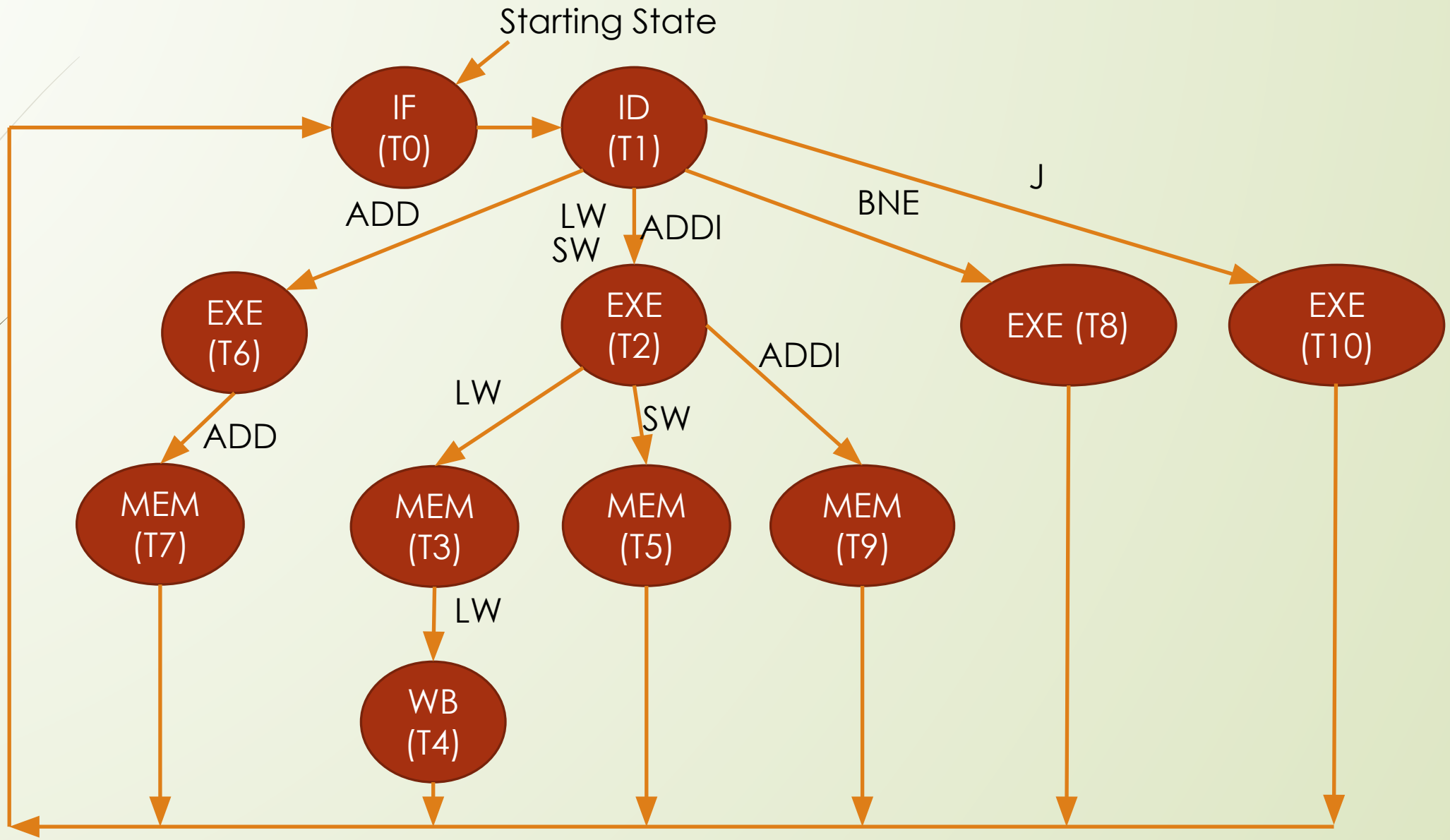
ADDI instruction

Machine state	Operation	Control signals	Next state
T2	A OP SigExtn(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00	T9
T9	RF[Destn] A OP SigExtn(offset)	RegDst=0, MemtoReg=0	T0

Jump instruction

Machine state	Operation	Control signals	Next state
T10	A OP SigExtn(offset)	Jump=1, PCSrc=10	T0

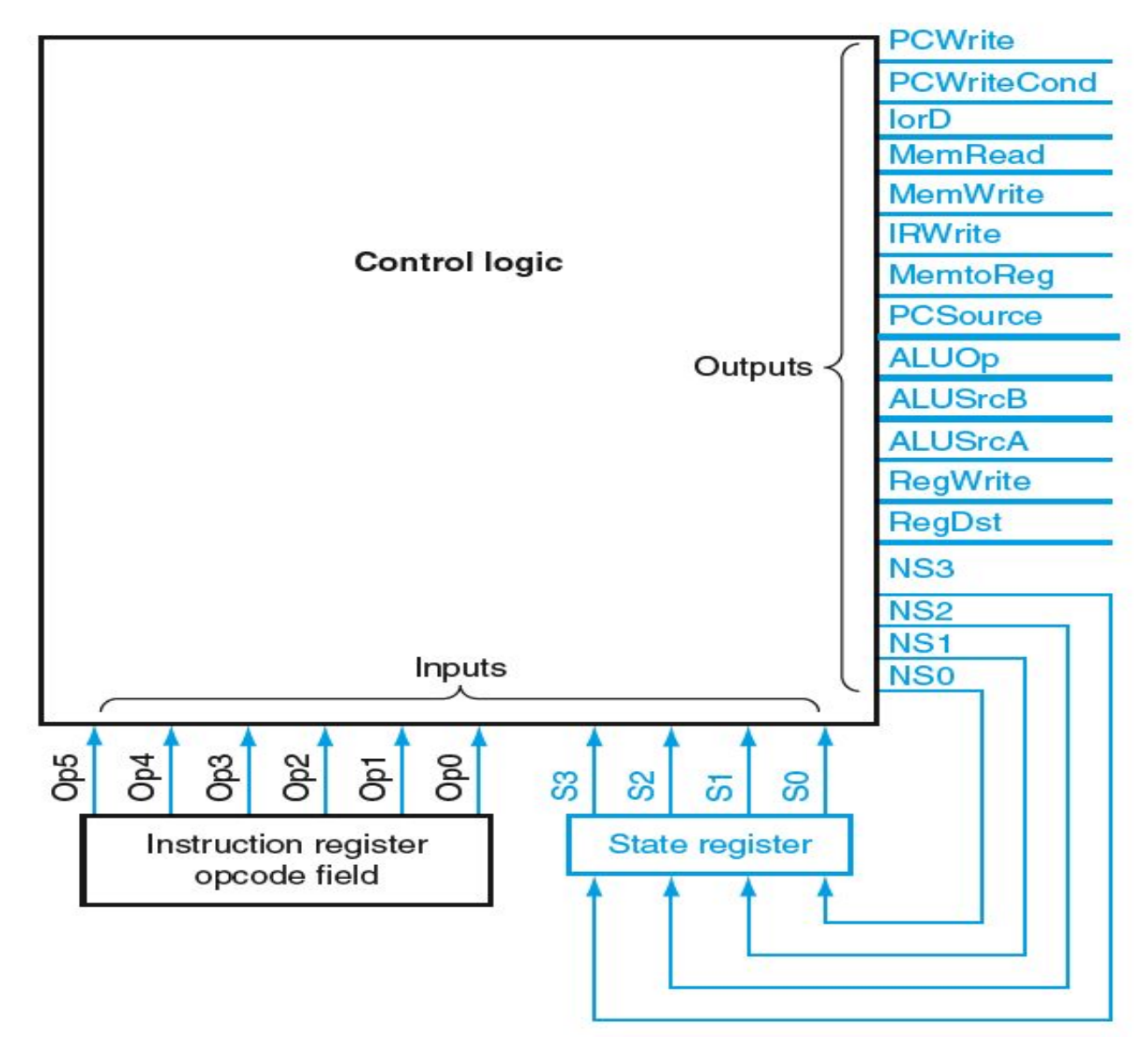
Controller States' information: Graphical Representation



Clock-cycle needed for the instructions

Instructions	Clock-cycle
LW	5
SW	4
R-type	4
BEQ	3
ADDI	4
J	3

(Finite state) Control Unit



Book-P&H – ch-appx-D

Fetch stage

Machine state (Inputs)	Operation	Control signals (Outputs)	Next state (Outputs)
T0	InsR □ M[PC]; PC □PC+4	IorD=0, IRWrite=1, MemRead=1 ALUSrcA=0, ALUSrcB=01, ALUOp=00, PCSrc=00, PCWrite=1, Jump=0	T1

Microprogram based control unit

Address	IorD	Jump	Memwrite	MemRead	IRWrite	RegDst	MemtoReg	RegWrite	ALUSrcA	ALUSrcB_1:0	PCWrite	Branch	PCSrc	ALUOp	NextAddress	Mode		States
0	0	0	0	1	1	x	x	0	0	01	1	0	00	00	1	0	Fetch	T0
1	x	0	0	0	0	x	x	0	0	11	0	0	00	00	xxxx	1	Decode	T1
2	x	0	0	0	0	x	x	0	1	10	0	0	00	00	3	0		T2
3	1	0	0	1	0	x	x	0	x	xx	0	0	00	xx	4	0	LW	T3
4	x	0	0	0	0	0	1	1	x	xx	0	0	00	xx	0	0		T4
5	x	0	0	0	0	x	x	0	1	10	0	0	00	00	6	0	SW	T2
6	1	0	1	0	0	x	x	0	x	xx	0	0	00	xx	0	0		T3
7	x	0	0	0	0	x	x	0	1	00	0	0	00	00	8	0	ADD	T2
8	x	0	0	0	0	1	0	1	x	xx	0	0	00	xx	0	0		T3
9	x	0	0	0	0	x	x	0	1	10	0	0	00	00	10	0	ADDI	T2
10	x	0	0	0	0	0	0	0	x	xx	0	0	00	xx	0	0		T3
11	x	1	0	0	0	x	x	0	x	xx	0	0	10	xx	0	0	J	T2
12	x	0	0	0	0	x	x	0	1	00	0	1	01	01	0	0	BEQ	T2

Can we reduce the bits used for NextAddress field?

Controller's Truth (state) Table

CF: Control Field: 15 Signals or 17 bits

NA: Next Address: 4 bit

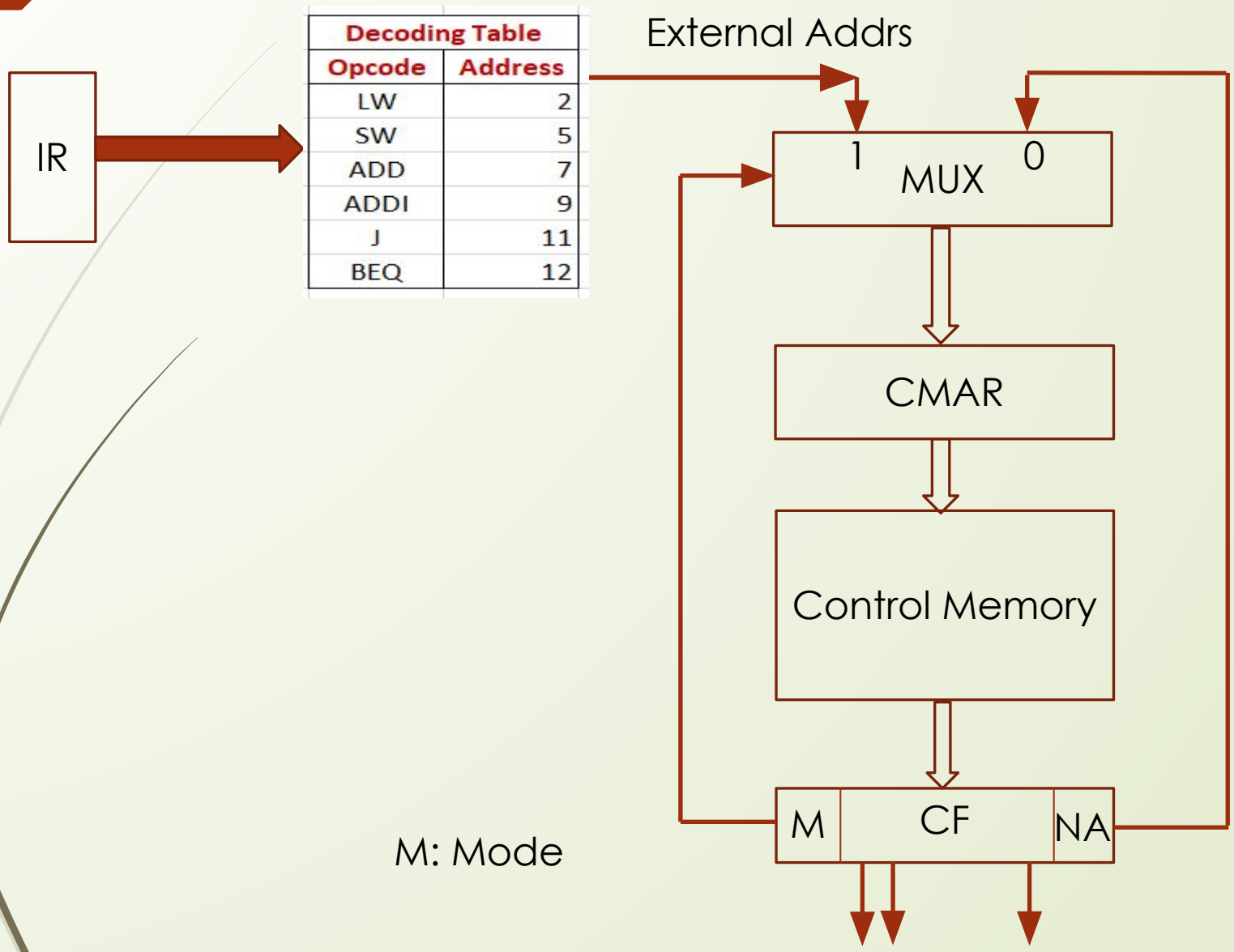
Decoding Table	
Opcode	**Address**
LW	2
SW	5
ADD	7
ADDI	9
J	11
BEQ	12

Book-Hamacher-ch5
P&H – ch-appx-C

For better understanding, we have represented the value in NextAddress and the Address as decimal.

Microprogrammed Control

- Organization of microprogrammed control unit



Microprogram sequencer: all units, except control memory.

CF: Control Field

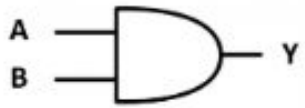
NA: Next Address

Book-Hamacher-ch5
P&H – ch-appx-C

Hardwired based control unit

Address	IorD	Jump	Memwrite	IRWrite	RegDst	MemtoReg	RegWrite	ALUSrcA	ALUSrcB_1:0	PCWrite	Branch	PCSrc	ALUOp	NextAddress	Mode		States
0	0	0	0	1	x	x	0	0	01	1	0	00	00	1	0	Fetch	T0
1	x	0	0	0	x	x	0	0	11	0	0	00	00	xxxx	1	Decode	T1
2	x	0	0	0	x	x	0	1	10	0	0	00	00	3	0		T2
3	1	0	0	0	x	x	0	x	xx	0	0	00	xx	4	0	LW	T3
4	x	0	0	0	0	1	1	x	xx	0	0	00	xx	0	0		T4
5	x	0	0	0	x	x	0	1	10	0	0	00	00	6	0		T2
6	1	0	1	0	x	x	0	x	xx	0	0	00	xx	0	0	SW	T3
7	x	0	0	0	x	x	0	1	00	0	0	00	00	8	0		T2
8	x	0	0	0	1	0	1	x	xx	0	0	00	xx	0	0	ADD	T3
9	x	0	0	0	x	x	0	1	10	0	0	00	00	10	0		T2
10	x	0	0	0	0	0	0	x	xx	0	0	00	xx	0	0	ADDI	T3
11	x	1	0	0	x	x	0	x	xx	0	0	10	xx	0	0	J	T2
12	x	0	0	0	x	x	0	1	00	0	1	01	01	0	0	BEQ	T2

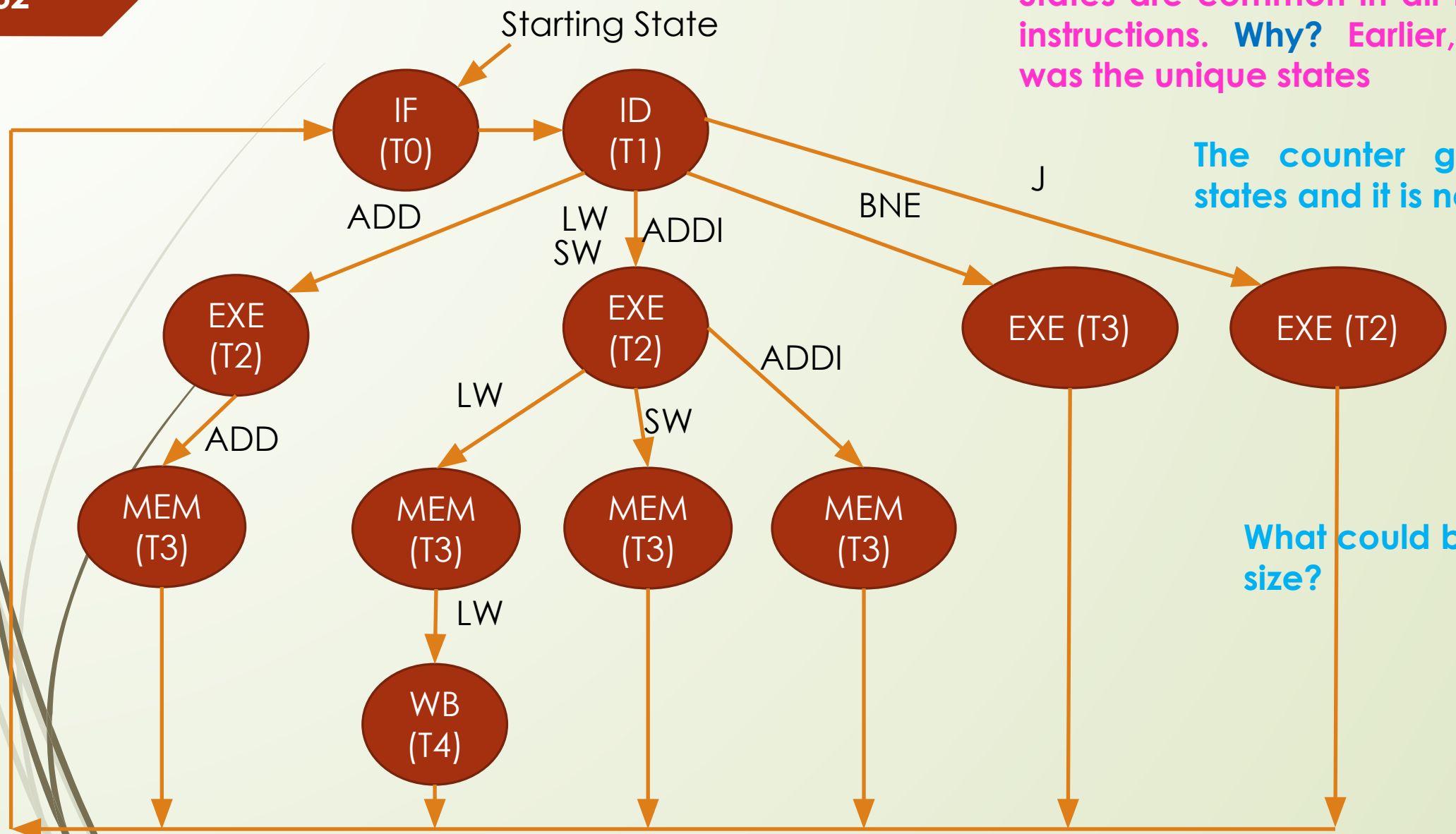
Inputs		Output
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



$Y = A.B$

Decoding Table	
Opcode	Address
LW	2
SW	5
ADD	7
ADDI	9
J	11
BEQ	12

Controller States' information for Hardwired sequential CU

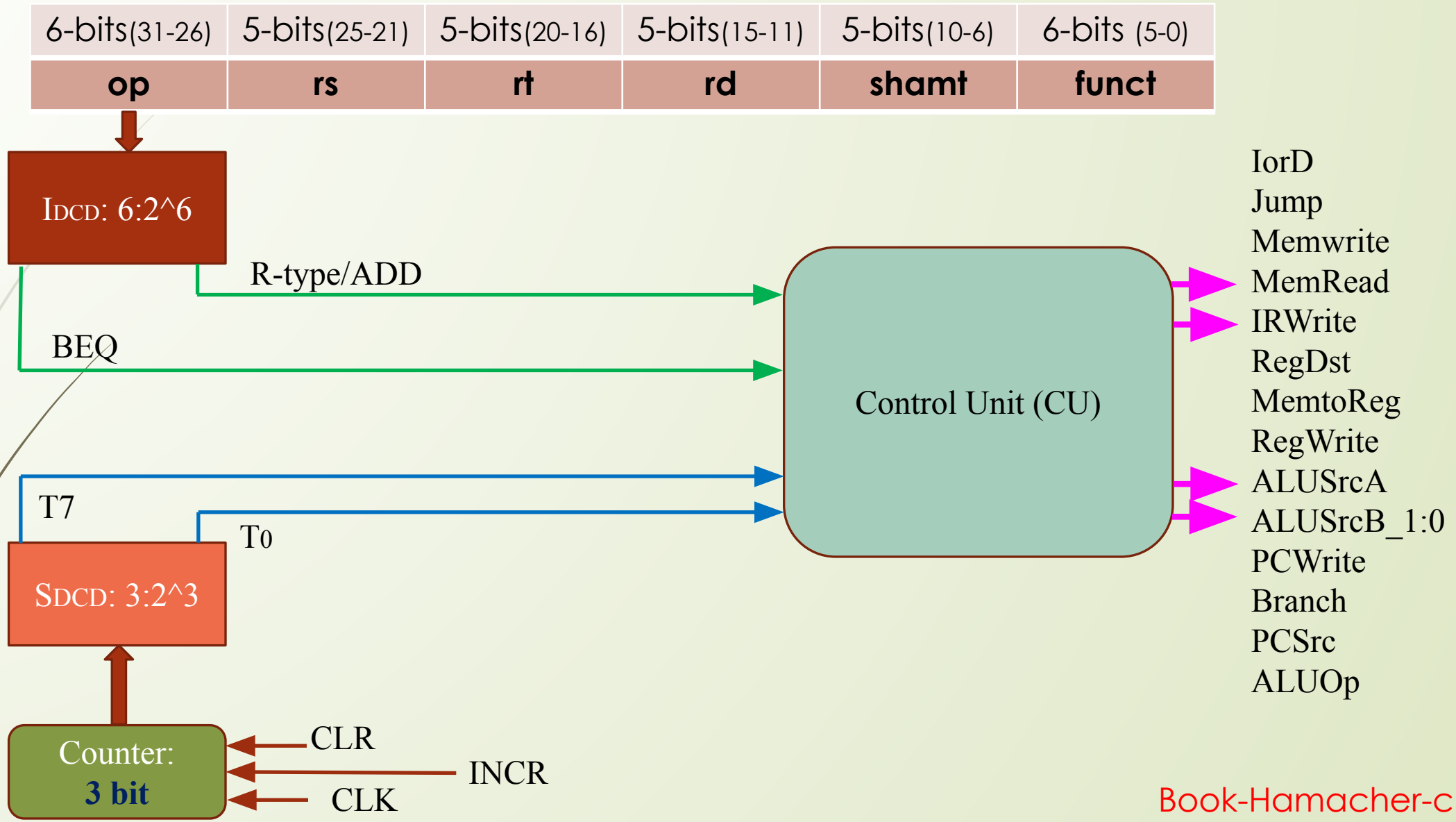


States are common in all the instructions. Why? Earlier, it was the unique states

The counter generates the states and it is not loadable.

What could be the counter size?

Hardwired based control unit



Book-Hamacher-ch5
P&H – ch-appx-C

Hardwired based control unit

IorD = T0’ + T3.LW + T3.SW +
Jump = T2.J
Memwrite = T3.SW
MemRead=T0’+
IRWrite =
RegDst =
MemtoReg =
RegWrite =
ALUSrcA =
ALUSrcB_1:0 =
PCWrite =
Branch =
PCSrc =
ALUOp = {T0’,T0’}+{T1’,T1’}+{(T2.LW)’, (T2.LW)’}
INCR = T0 + T1 + T2.LW + T3.LW + T2.SW
CLR = T4.LW + T3.SW +

Logical Expression of Control Signals

Two approaches

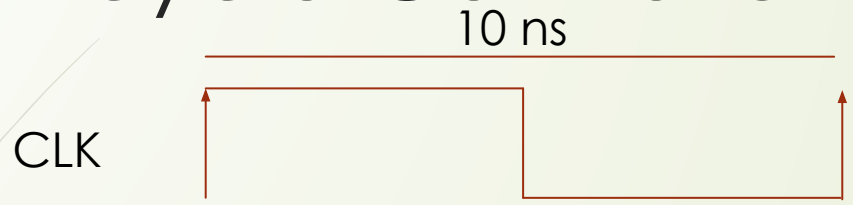
- All the ALU operations (ALUOp) similar as in Single-cycle approach
- Generate the control signals using:
  - Hardwired-based approach
  - Microprogrammed-based approach
- Inputs of the control unit similar as in Single-cycle approach

Comparison between Single and Multi-cycle datapath

- Single cycle
  - No non-architectural component
  - Every instruction takes one clock cycle (CPI=1)
- Multi-cycle
  - Non-architectural elements/register
    - Instruction, data (memory), A, B, ALU-output
  - Instructions take different unit of clock cycle
    - Average CPI (>1)

Comparison between Single and Multi-cycle Control unit

Single Cycle

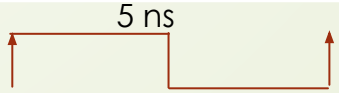


ALUOp	Meaning
00	add
01	subtract
10	Look at *funct* field
11	n/a

Instr.	Jump	RegDst	RegWrite	ALUSrc	Branch	ALUOp1	ALUOp0	MemRead	MemWrite	MemtoReg
R-type	0	1	1	0	0	1	0	0	0	0
lw	**0**	**0**	**1**	**1**	**0**	**0**	**0**	**1**	**0**	**1**
sw	0	x	0	1	0	0	0	0	1	x
addi	0	0	1	1	0	0	0	0	0	0
B-type	0	x	0	0	1	0	1	0	0	x
J-type	1	x	0	x	x	x	x	0	0	x

Multi-Cycle

LW's FSM



Machine state	Operation	Control signals	Next State
T2	A + sigEx(offset)	ALUSrcA=1, $ALUSrcB_{1:0}$ = 10, ALUOp=00	T3
T3	Data □M[A+sigEx(off)]	IorD=1	T4
T4	RF[dest] □Data	RegDst=0, MemtoReg=1, RegWrite=1	T0

Performance Analysis

- The program consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Determine the average CPI for this program.

Performance Analysis

- The program consists of approximately 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions. Determine the average CPI for this program.
- The average CPI is the sum over each instruction of the CPI for that instruction multiplied by the fraction of the time that instruction is used
- **Average CPI** = (0.11 + 0.02)(3) + (0.52 + 0.10)(4) + (0.25)(5) = 4.12
- For Single-cycle approach, Avg. CPI is 1

Performance Analysis

- Each cycle involved **one** ALU operation, memory access, or register file access
- Each instruction is using only one stage at any time
- Assumptions:
  - the register file is faster than the memory and
  - writing memory is faster than reading memory
- Datapath has two possible critical paths:
- $T_c = t_{clk2q} + t_{mux} + \max\{t_{mux} + t_{ALU}, t_{mem}\} + t_{setup}$

Begin of each stage

Delay elements

Setup time for each stage

t_{clk2q}= clock-to-Q D-ff

Performance Analysis

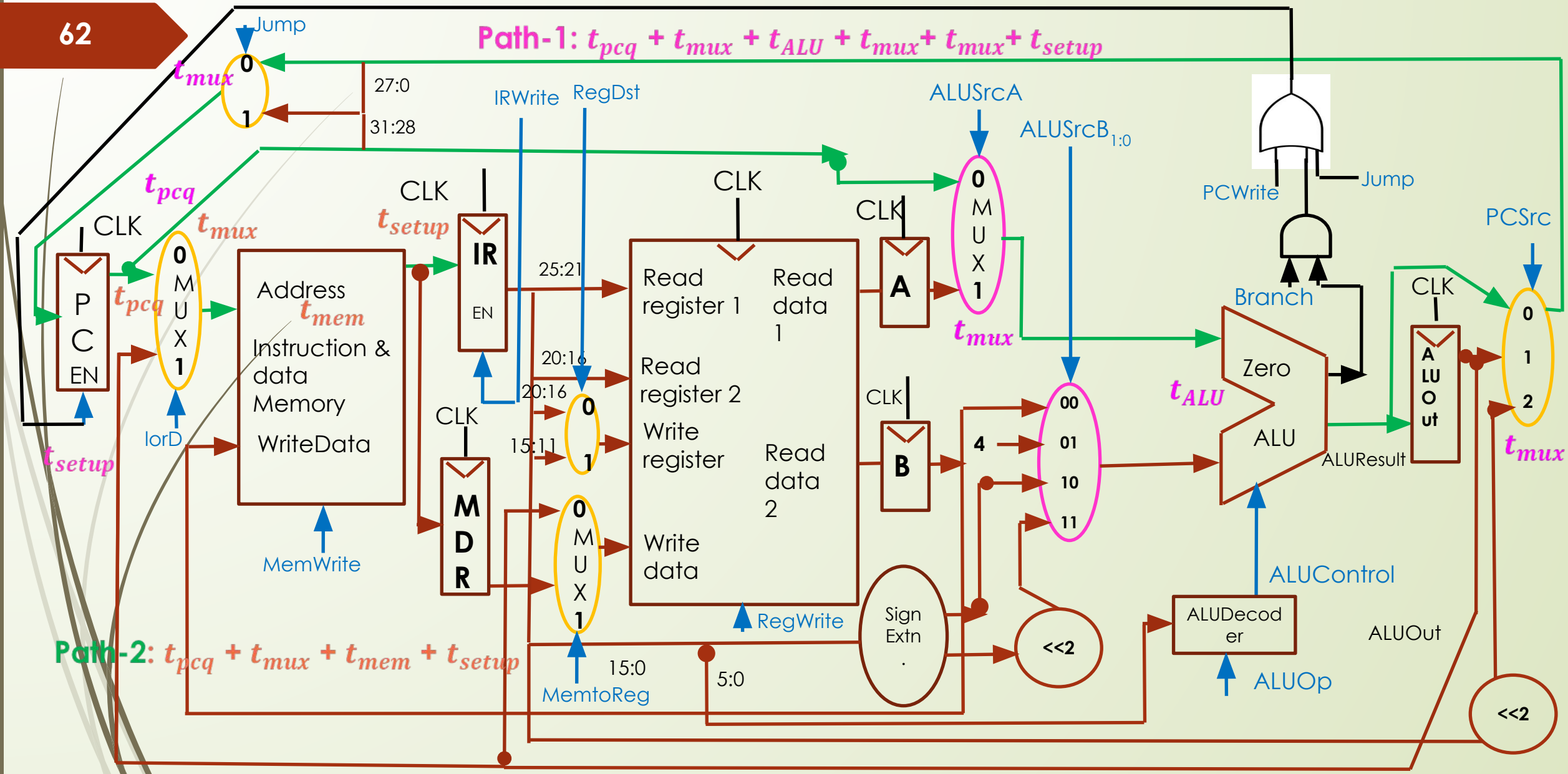
- Each cycle involved **one** ALU operation, memory access, or register file access
- Each instruction is using only one stage at any time
- Assumptions:
  - the register file is faster than the memory and
  - writing memory is faster than reading memory
- Datapath has two possible critical paths:
- $T_c = t_{pcq} + t_{mux} + \max\{t_{ALU} + t_{mux} + t_{mux}, t_{mem}\} + t_{setup}$

Path-1: $t_{pcq} + t_{mux} + t_{ALU} + t_{mux} + t_{mux} + t_{setup}$

Path-2: $t_{pcq} + t_{mux} + t_{mem} + t_{setup}$

Combined Datapath of all types of instructions

Green line decides the Critical paths



Performance Analysis

- XYZ-organization is contemplating building the multi-cycle MIPS processor instead of the single-cycle processor. For both designs, the organization plans on using a 65-nm CMOS manufacturing process. The organization has determined that the logic elements have the delays given in Table. Help the organization compare each processor's execution time for a program with 100 billion instructions

Parameter	Delay (ps)
	30
	250
	20
	200
	25
	20

Performance Analysis of Multi-Cycle Implementation

- XYZ-organization is contemplating building the multi-cycle MIPS processor instead of the single-cycle processor. For both designs, the organization plans on using a 65-nm CMOS manufacturing process. The organization has determined that the logic elements have the delays given in Table. Help the organization compare each processor's execution time for a program with 100 billion instructions

Parameter	Delay (ps)
	30
	250
	20
	200
	25
	20

- $T_c = t_{pcq\_PC} + t_{mux} + \max\{t_{ALU} + t_{mux} + t_{mux}, t_{mem}\} + t_{setup}$
- T_c = 30 + 25 + 250 + 20 = **350 ps**
- Execution time =

  (100 * 10^9 instrs.) * (**4.12** cycle/instrs.) * (350 * 10^{-12} s/cycle)

  = 133.9 seconds

Performance Analysis: A Comparison

- For multi-cycle, T_c = 350 ps and CPI = 4.12
- For single-cycle, T_c = 925 ps and CPI = 1
- For multi-cycle, execution time = 133.9 seconds
- For single-cycle, execution time = 92.5 seconds
- This example shows multi-cycle processor is slow than the single-cycle processor; why is it so?
  - Sequencing overhead: 30 (clk-Q) + 20 (t_{setup})
- Multi-cycle processor is less expensive
- It has 5-nonarchitectural elements

Para meter	Delay (ps)
	30
	250
	20
	200
	25
	20

Homeworks

- How many cycles are required to run the following program on the multicycle MIPS-processor? What is the CPI of this program?

```
Addi $s1, $s2, 5
Sub $t0, $t1, $t2
Lw $t3, 15($s1)
Sw $t5, 72($t0)
Or  $t2, $s4, $s5
```

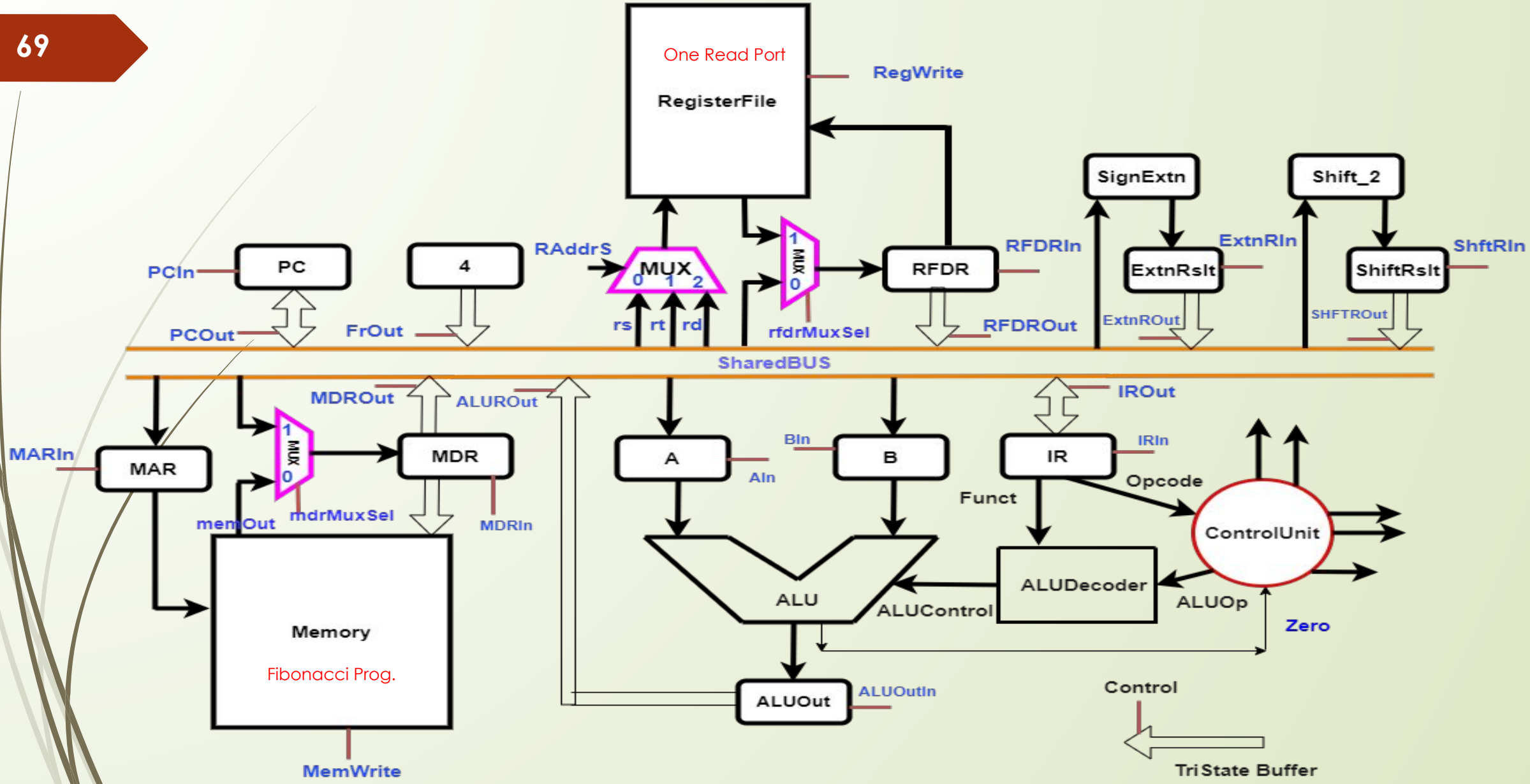
Homework

- Write CPP programs which can simulate the behaviors of the following instructions at **functional level** for MIPS-processor with **single cycle** and **multi-cycle** datapath:
  - R-type: ADD
  - M-type: LW & SW
  - B-type: BEQ
  - I-type: ADDI
  - J-type: J
- Write a CPP program for microprogramme-based CU for MIPS ISA

Processor Design With Constraints on Interconnection

Will it be a Single-Cycle or Multi-Cycle Processor?

Design this 32 bits MIPS Processor: LW, SW, ADDI, ADD, BNE



Design this 32 bits MIPS Processor: LW, SW, ADDI, ADD, BNE

Fetch Stage

Machine state	Operation	Control signals	Next state
T0	MAR □ PC A □ PC	PCOut=1, MARIn=1 Ain=1	T1
T1	MDR □ Mem[MAR] B□4	mdrMuxSel=0, MDRIn=1, FrOut=1, Bin=1, ALUOp=00	T2
T2	ALUOut □A+B	ALUOutIn=1	T3
T3	PC □ ALUOut	PCIn=1, ALUROut=1	T4
T4	IR □ MDR	MDROut=1, IRIn=1	xxx

Design this 32 bits MIPS Processor: LW, SW, ADDI, ADD, BNE

ADD

Machine state	Operation	Control signals	Next state

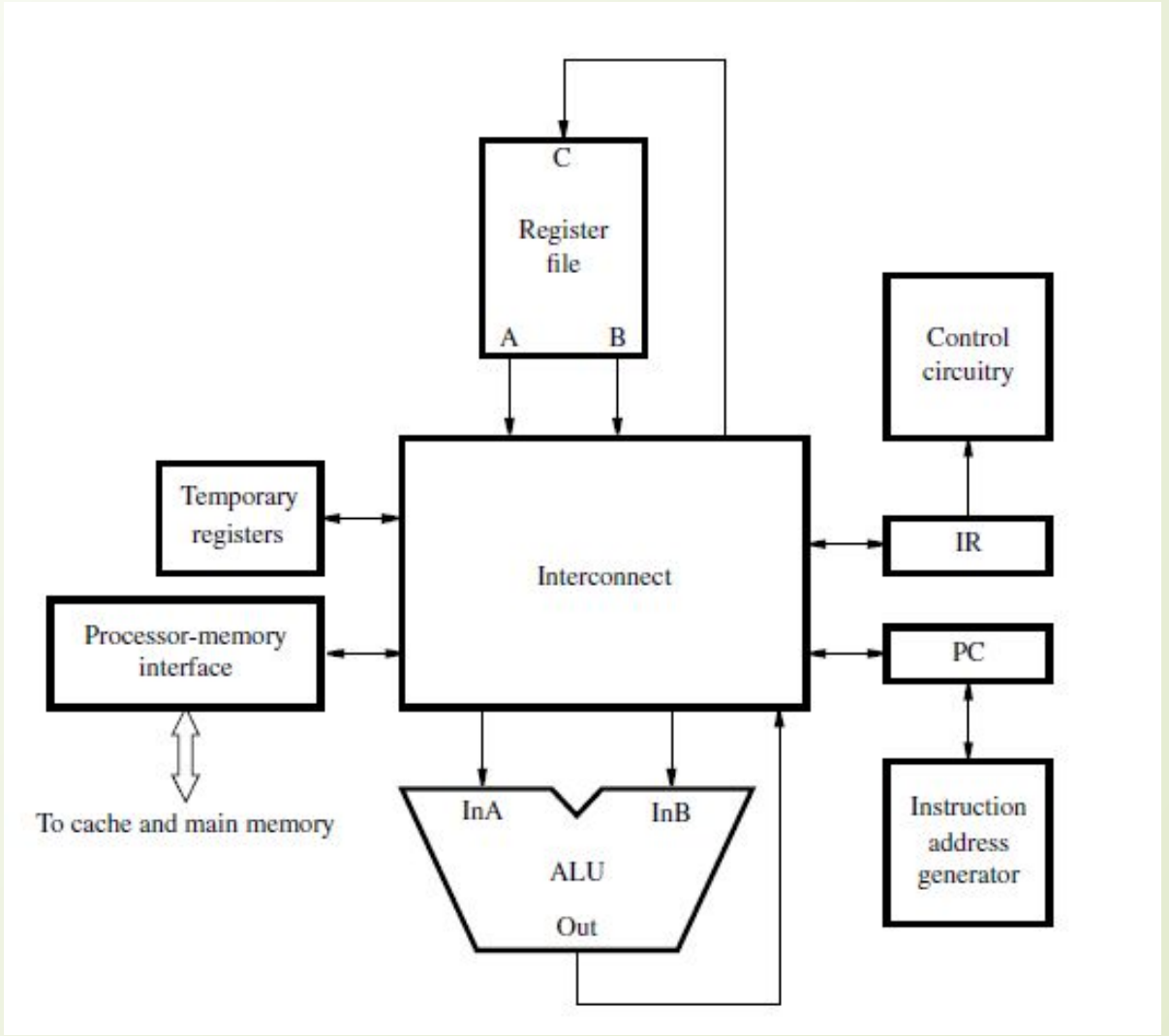
Homework: Design this 32 bits MIPS Processor: LW, SW, ADDI, ADD, BNE

All other Instructions

No of states: T0 to T29

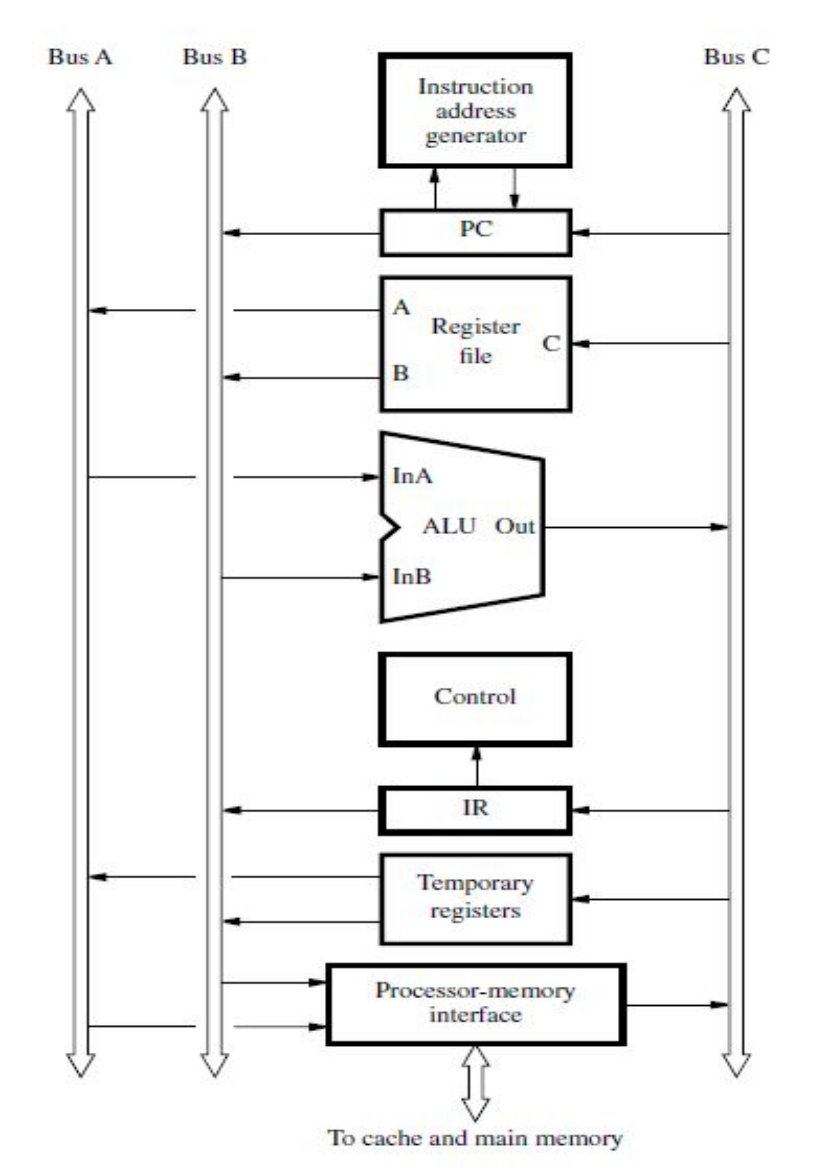
Machine state	Operation	Control signals	Next state

Organization of a CISC-style processor.



Book-Hamacher-ch5

Three-bus CISC-style processor organization.



Book-Hamacher-ch5

Summary

- Disadvantages of Single-cycle processor
- Necessity of balanced design approach and multi-cycle processor
- Datapath design of multi-cycle processor
- Design of Controls for multi-cycle processor
- Performance analysis of multi-cycle processor
- Comparison between single-cycle and multi-cycle processor
- Disadvantages of multi-cycle processor