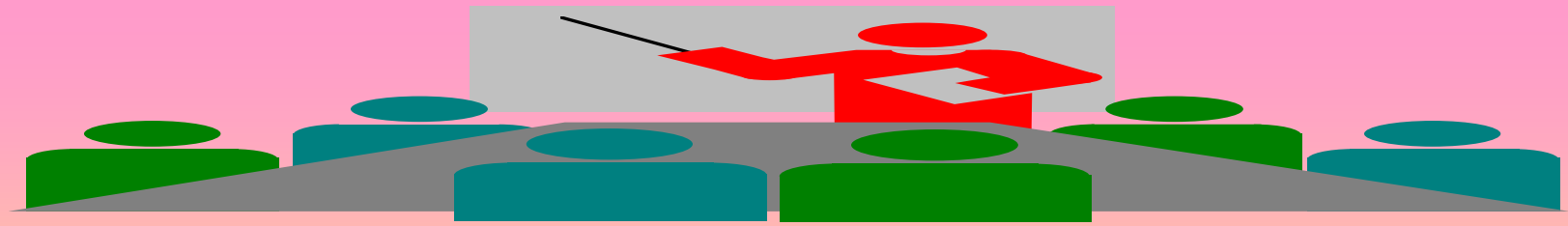


Computer Organization & Design

The Hardware/Software Interface

Chapter 5 The processor : Datapath and control

Qing-song Shi

http://10.214.26.103 **Email:** zjsqs@zju.edu.cn

Chapter 5

The processor : Datapath and control

Chapter Five

The processor : Datapath and control

5.1 Introduction

5.2 Logic Design Conventions (skip)

5.3 Building a datapath

5.4 A Simple Implementation Scheme

5.5 A Multicycle Implementation

5.5 Microprogramming

5.6 Exception

What is the MIPS?

Microprocessor without Interlocked Pipeline Stages

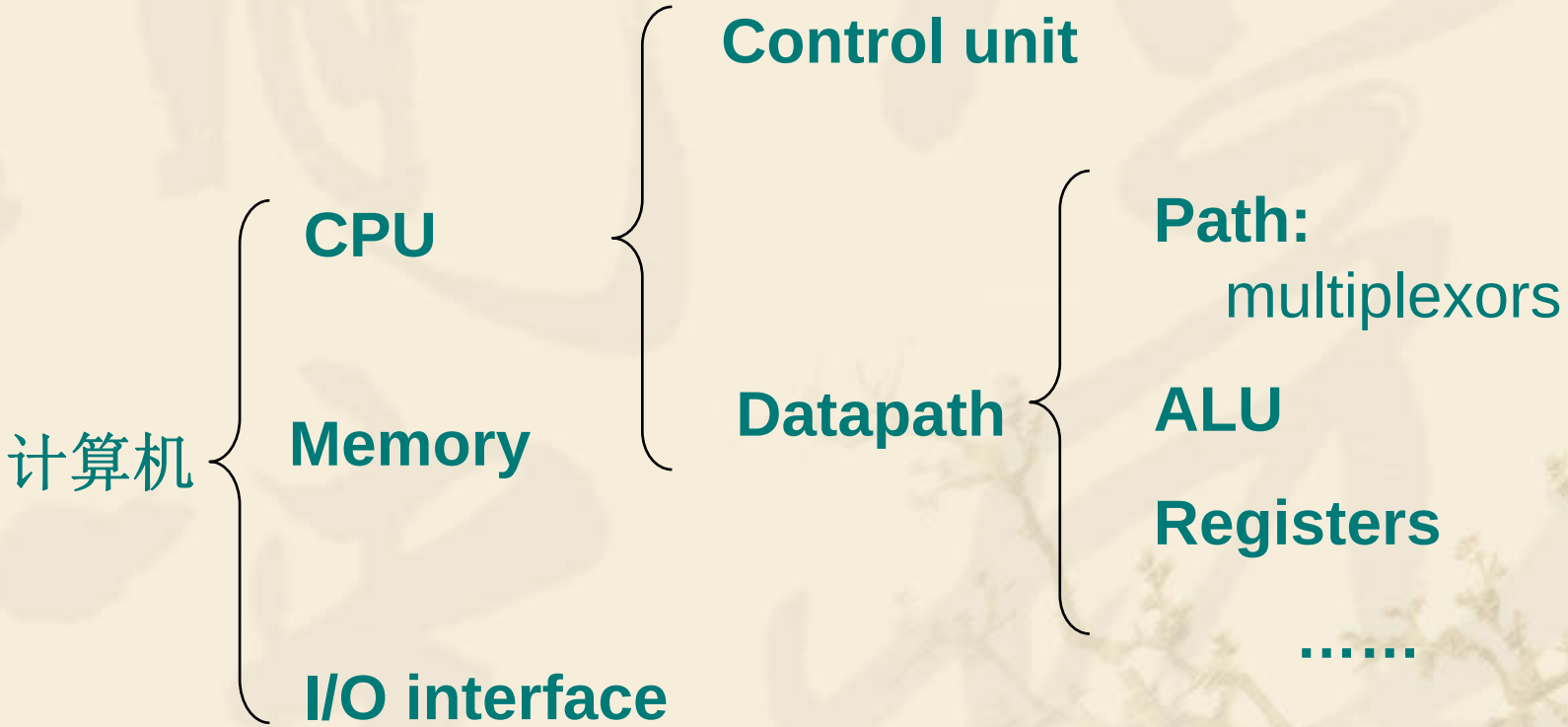
5.1 Introduction

- We'll look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- An Overview of the implementation
  - For every insruction, the first two step are identical
    1. **Fetch the instruction from the memory**
    2. **Decode and read the registers**
  - Next steps depend on the instruction class
    - **Memory-refrernce** **Arithmetic-logical** **branches**

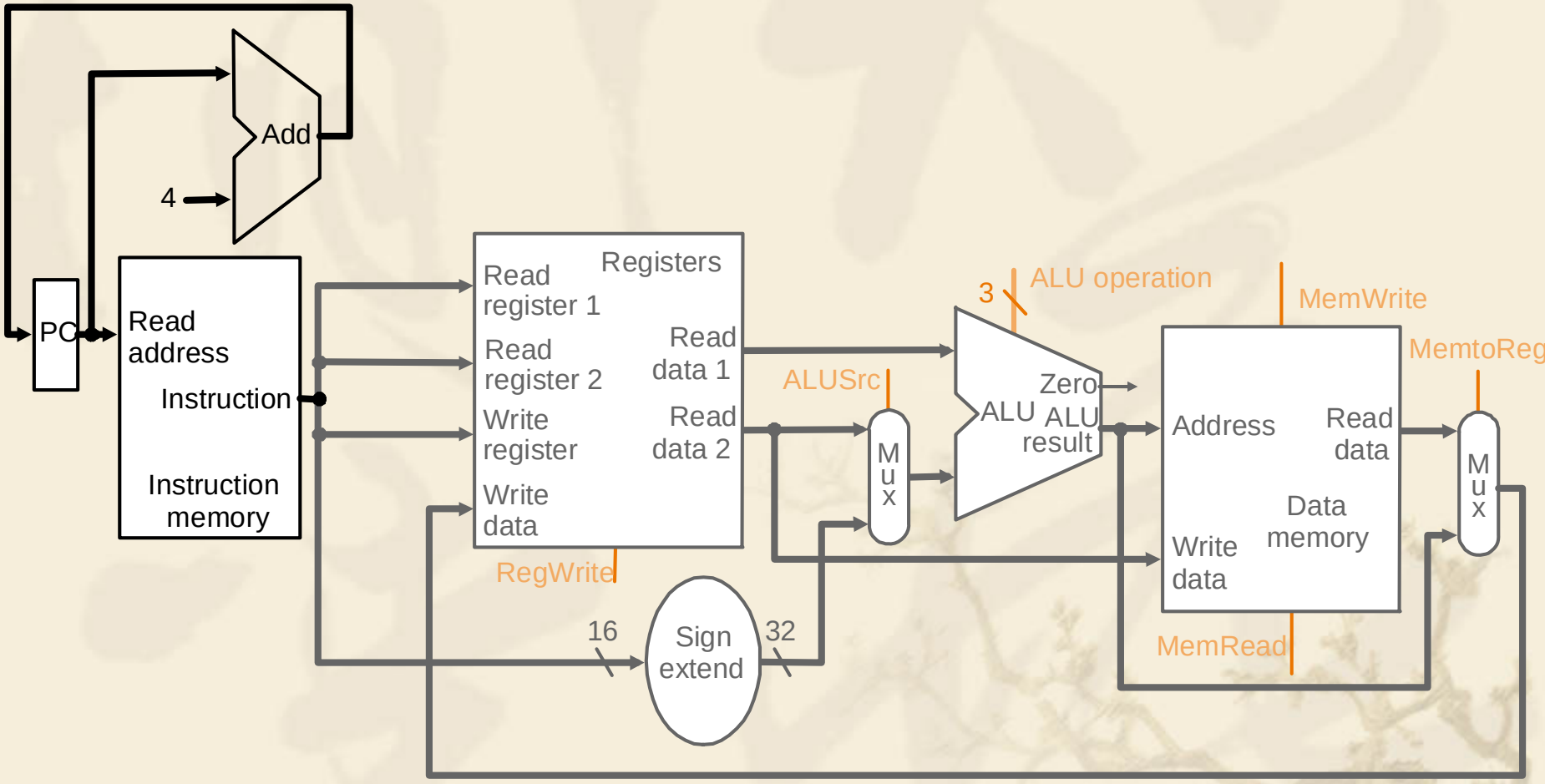
What are steps?

How many FUN.?

Computer Organization



An abstract view of the implementation of MIPS



Chapter Five

The processor : Datapath and control

5.1 Introduction

5.2 Logic Design Conventions (skip)

5.3 Building a datapath

5.4 A Simple Implementation Scheme

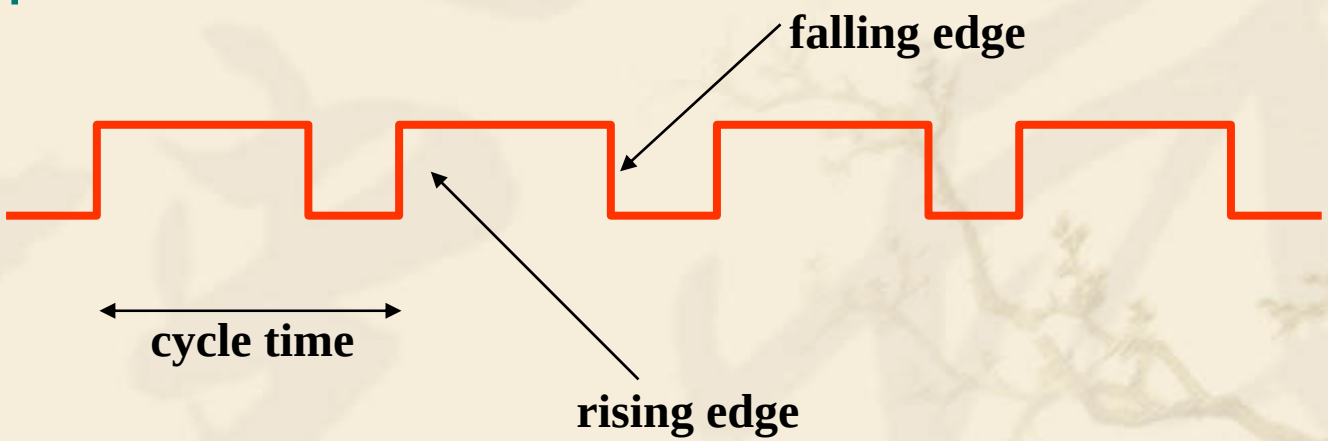
5.5 A Multicycle Implementation

5.5 Microprogramming

5.6 Exception

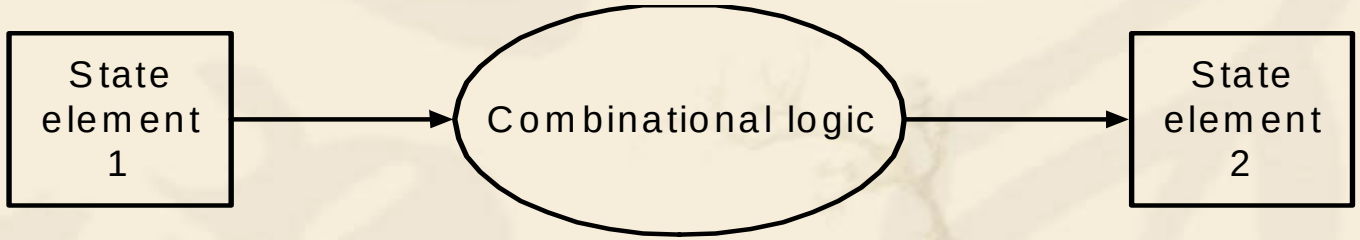
State Elements

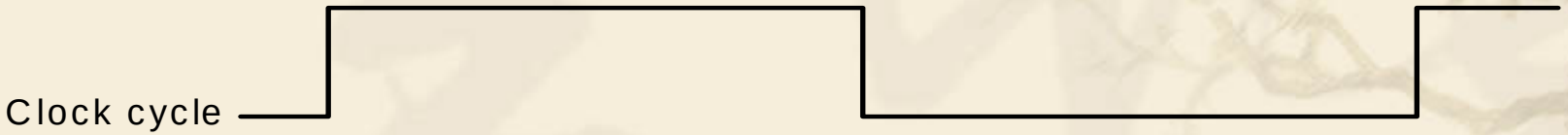
- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?



Our Implementation

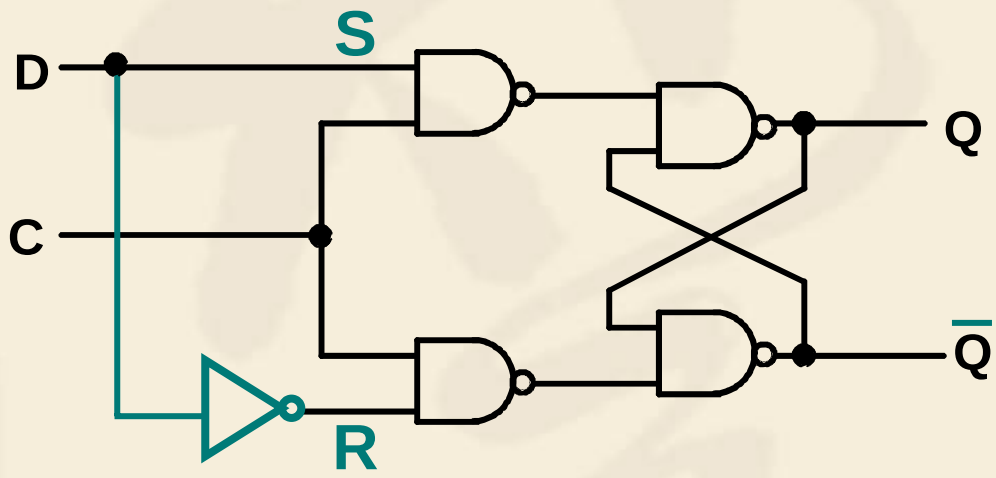
- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements





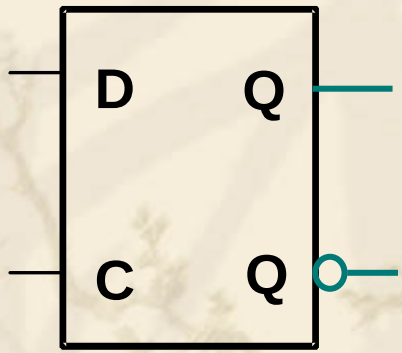
D-Latch for state

C	D	S	R	Q(t + 1)
0	X	X	X	hold
1	0	0	1	Q=0：reset
1	1	1	0	Q=1：reset



C	D	Q(t + 1)
0	X	hold
1	0	Q=0：reset
1	1	Q=1：set

D-Latch FUN. table



D-Latch symbol

Chapter Five

The processor : Datapath and control

5.1 Introduction

5.2 Logic Design Conventions (skip)

5.3 Building a datapath

5.4 A Simple Implementation Scheme

5.5 A Multicycle Implementation

5.5 Microprogramming

5.6 Exception

The datapath there are ……

Name	Example	Comments
32 register	$s0-$s7,$t0-$t9, $zero,$a0-$a3, $v0-$v1	Fast locations for data. In MIPS, data must be in registers to perform arithmetic. MIPS register $zero always equals 0. $gp(28) is the global pointer, $sp(29) is the stack pointer, $fp(30) is the frame pointer, and $ra(31) is the return address.
30 Word Addresses signals line	Memory[0], Memory[4] , …… , Menory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers, such as those saved on procedure calls.

Name	Register no.	Usage	Preserved on call
$zero	0	The constant value 0	n,.a.
$v0-$v1	2-3	Values for results and expression evaluation	no
$a0-$a3	4-7	Arguments	no
$t0-$t7	8-15	Temporaries	no
$s0-$s7	16-23	Saved	yes
$t8-$t9	24-25	More temporaries	no
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Framer pointer	yes
$ra	31	Return address	yes

ALU OP for MIPS machine language

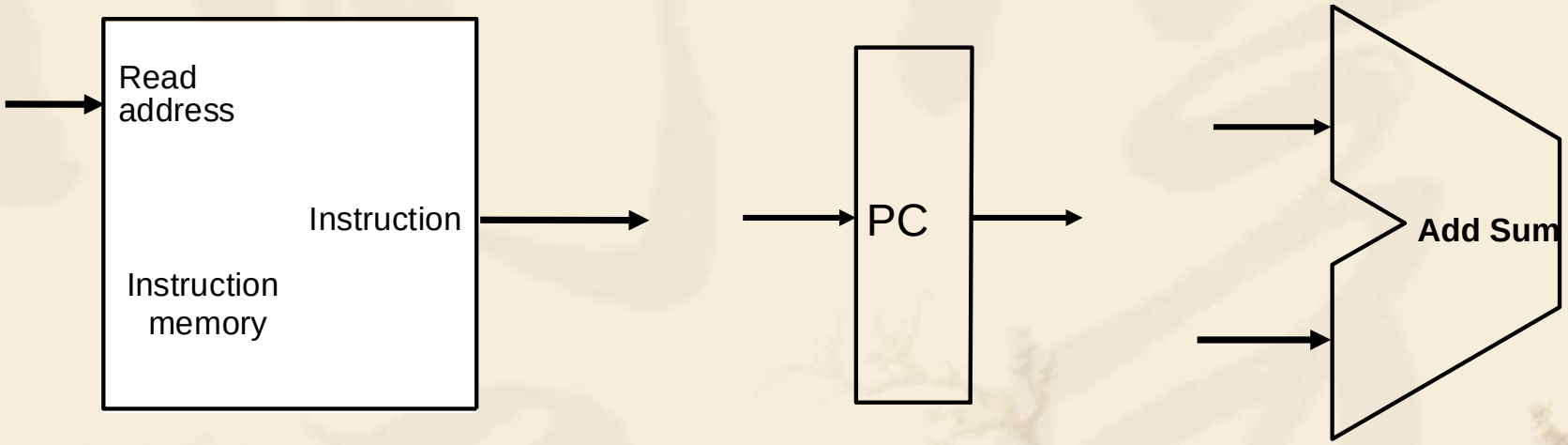
Name	Format	Example						Comment
add	R	0	18	19	17	0	32	add $s1, $s2, $s3
sub	R	0	18	19	17	0	34	sub $s1, $s2, $s3
lw	I	35	18	17	100			lw $s1, 100($s2)
sw	I	43	18	17	100			sw $s1, 100($s2)
and	R	0	18	19	17	0	36	and $s1, $s2, $s3
or	R	0	18	19	17	0	37	or $s1, $s2, $s3
nor	R	0	18	19	17	0	39	nor $s1, $s2, $s3
addi	I	12	18	17	100			addi $s1, $s2,100
ori	I	13	18	17	100			ori $s1, $s2,100
beq	I	4	17	18	25			beq $s1, $s2,100
bne	I	5	17	18	25			bne $s1, $s2,100
slt	R	0	18	19	17	0	42	slt $s1, $s2,$s3
j	J	2	2500					j 10000(see section 2.9)
jr	R	0	31	0	0	0	8	j Sra
jal	J	3	2500					jar 10000(see section 2.9)
Field size		6bits	5bits	5bits	5bits	5bits	6bits	All MIPS instruction 32 bits
R-format	R	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
i-format	I	op	rs	rt	address			Data transfer ,branch format

Instruction execution in MIPS

- Fetch :
  - Take instructions from the instruction memory
  - Modify PC to point the next instruction
- Instruction decoding & Read Operand:
  - Will be translated into machine control command
  - Reading Register Operands, whether or not to use
- Executive Control:
  - Control the implementation of the corresponding ALU operation
- Memory access:
  - Write or Read data from memory
  - Only LW/SW
- Write results to register:
  - If it is R-type instructions, ALU results are written to Rd
  - If it is I-type instructions, Results are written to Rt

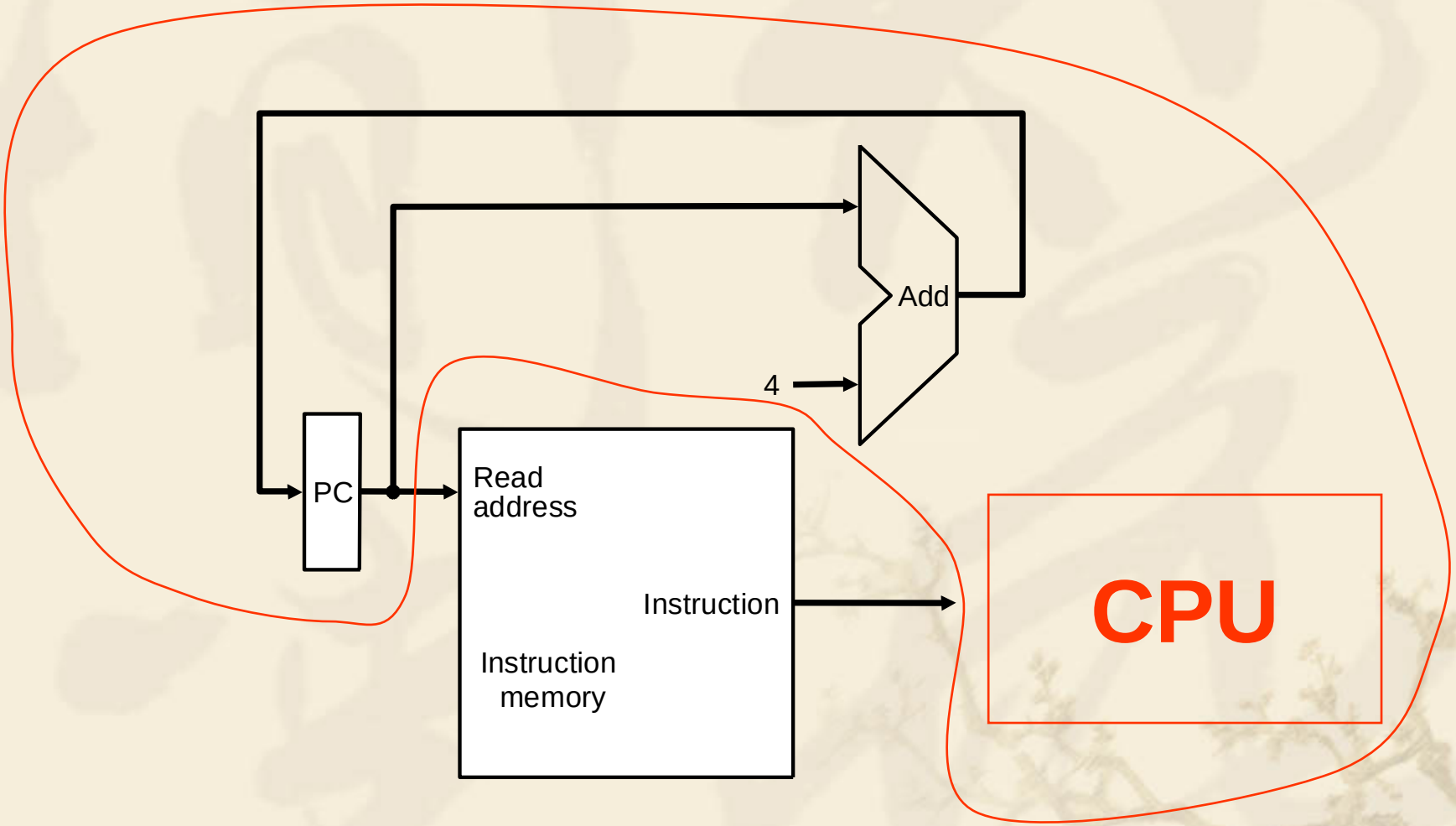
Instruction fetching three elements

How to connect? Who?



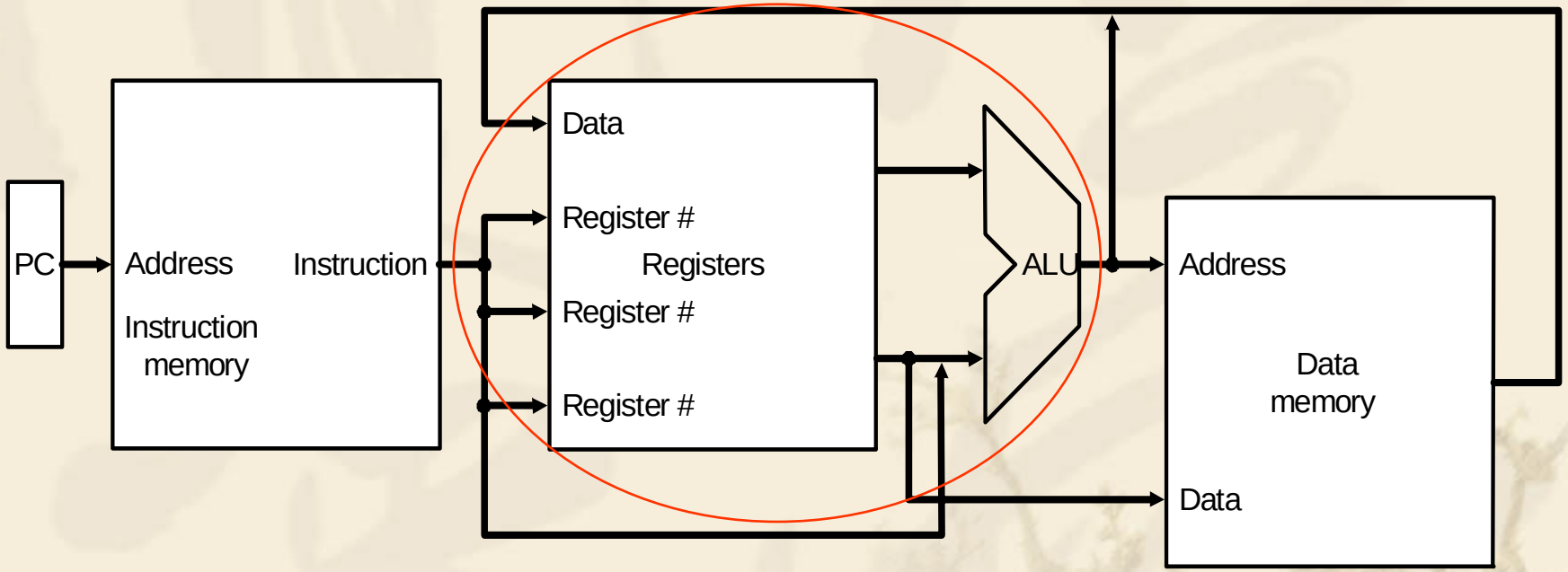
Instruction memory **Program counter** **Adder**

Instruction fetching unit



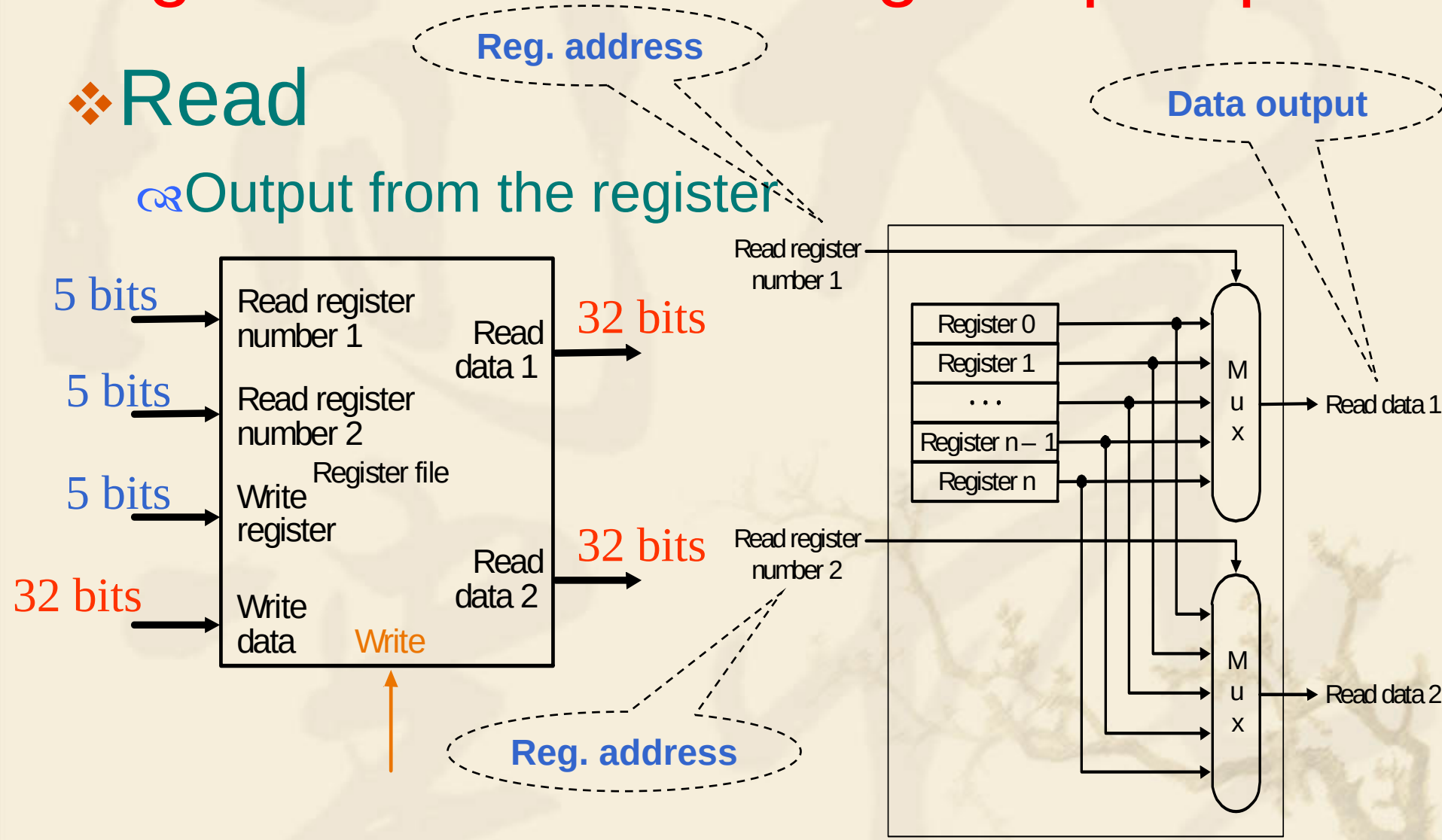
More Implementation Details

- Abstract / Simplified View:



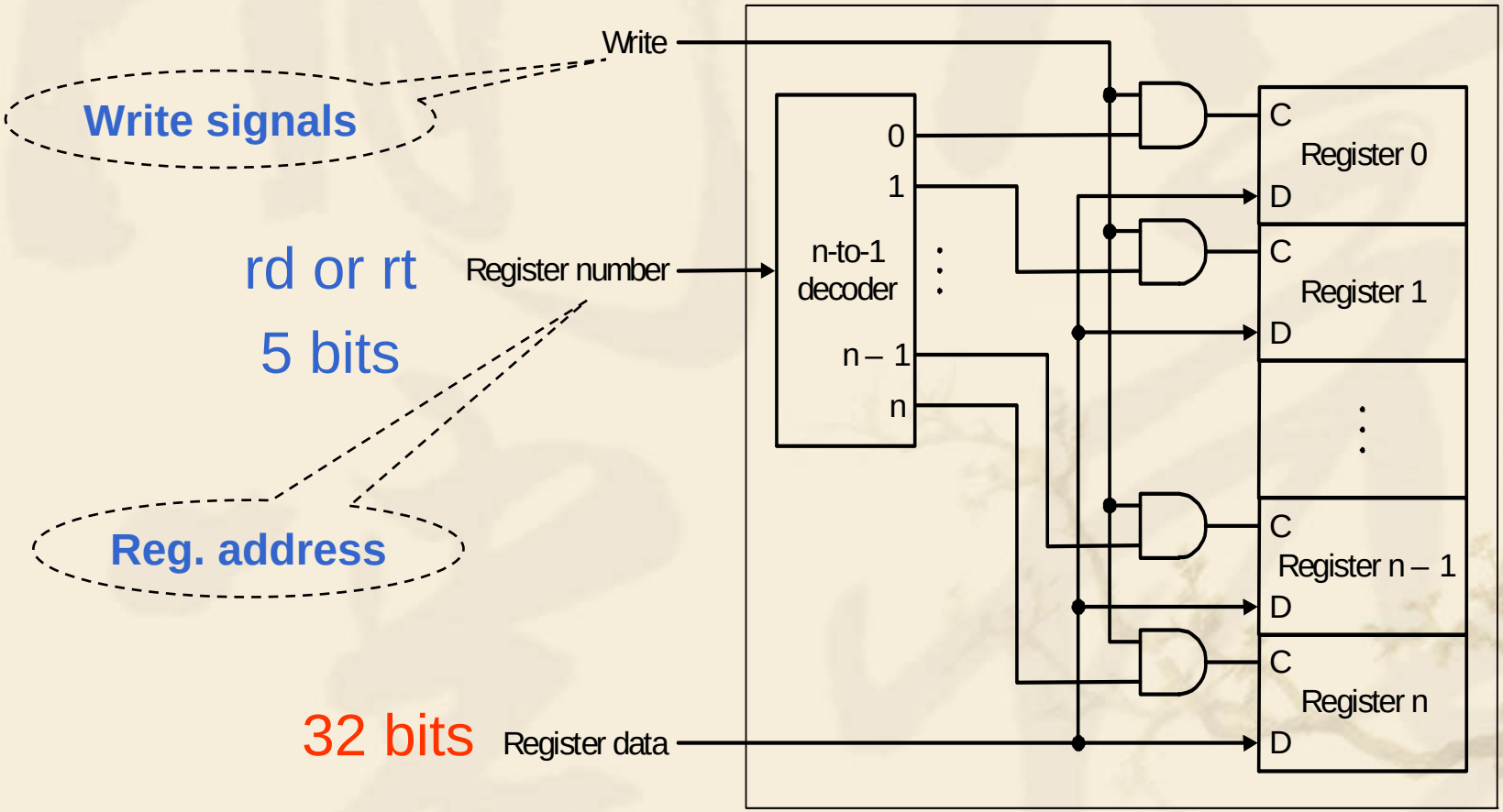
Register File--Built using D flip-flops

- Read
  - Output from the register



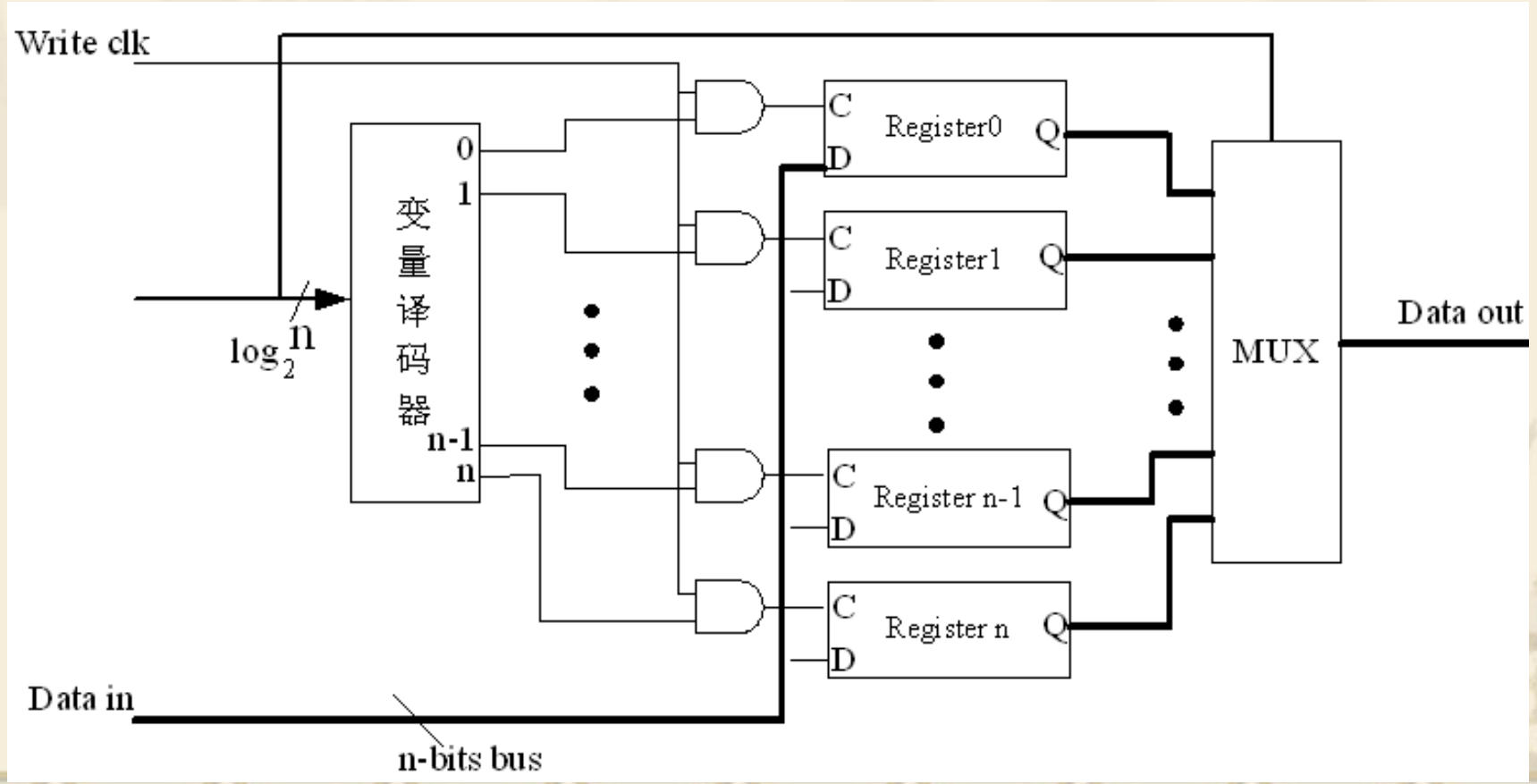
Register File

- Write
  - Written to the register



Register files

- Foundation element of Computer（Part of Datapath ）
- Aggregation of many Registers
- Register address、Control signals: Read/Write



Description: 32×32bits Register files

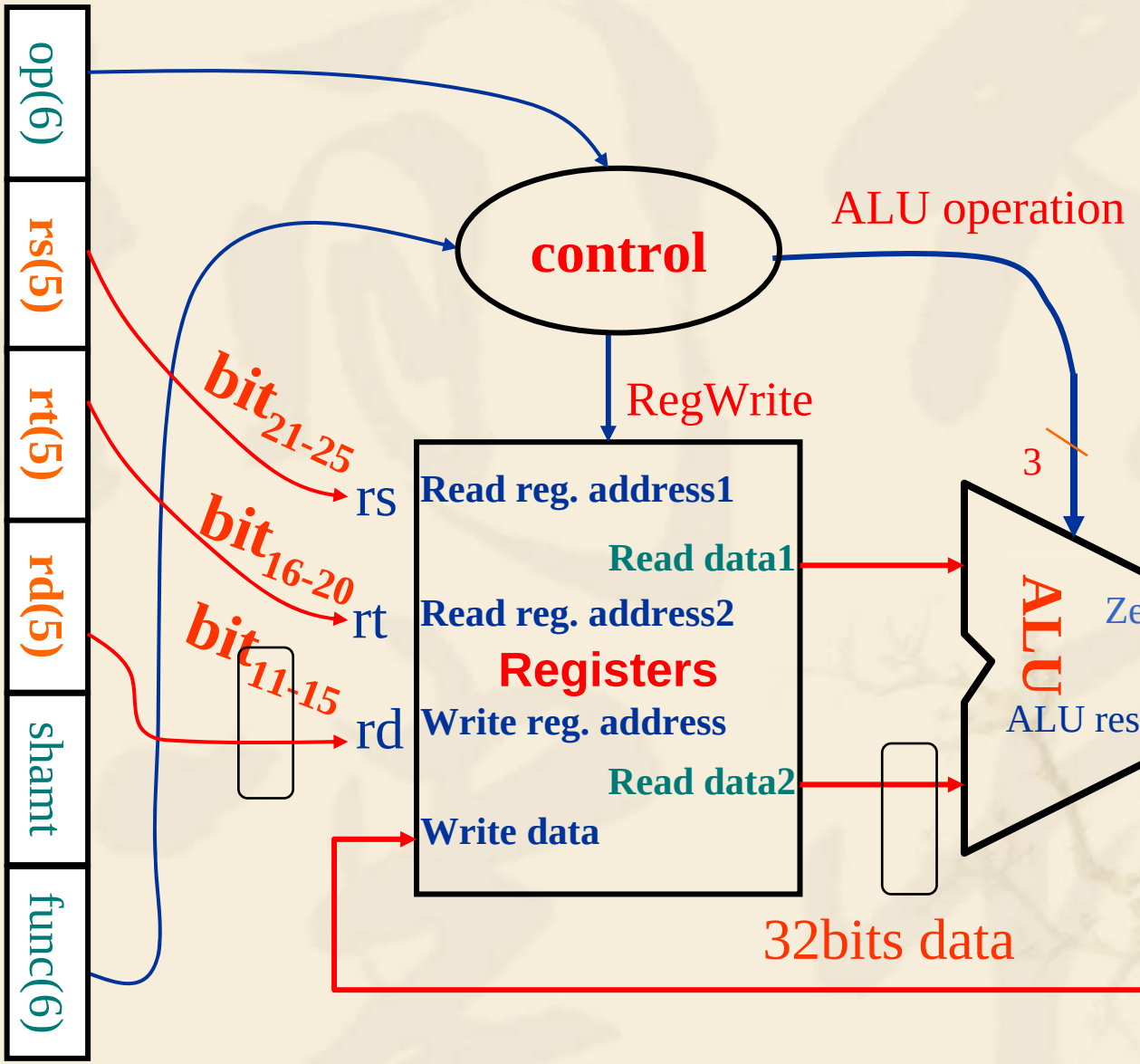
```
Module regs(clk, rst,reg_Rd_addr_A, reg_Rt_addr_B, reg_Wt_addr, wdata, we, rdata_A,
                    rdata_B);

   input clk, rst, we;
   input [4:0] reg_Rd_addr_A, reg_Rt_addr_B, reg_Wt_addr;
   input [31:0] wdata;
   output [31:0] rdata_A, rdata_B;
   reg [31:0] register [1:31];                                  // r1 - r31
    integer i;

    assign rdata_A = (reg_Rd_addr_A == 0)? 0 : register[reg_Rd_addr_A]; // read
    assign rdata_B = (reg_Rt_addr_B == 0)? 0 : register[reg_Rt_addr_B];   // read
    always @(posedge clk or posedge rst)
    begin
            if (rst==1)             begin                        // reset
               for (i=1; i<32; i=i+1)
               register[i] <= 0;
            end
            else begin
                if ((reg_Wt_addr != 0) && (we == 1))             // write
                register[reg_Wt_addr] <= wdata;
            end
    end
endmodule
```

Path Built using Multiplexer

- R-type instruction Datapath
- I-type instruction Datapath
  - For ALU
  - For memory
  - For branch
- J-type instruction Datapath
  - For Jump

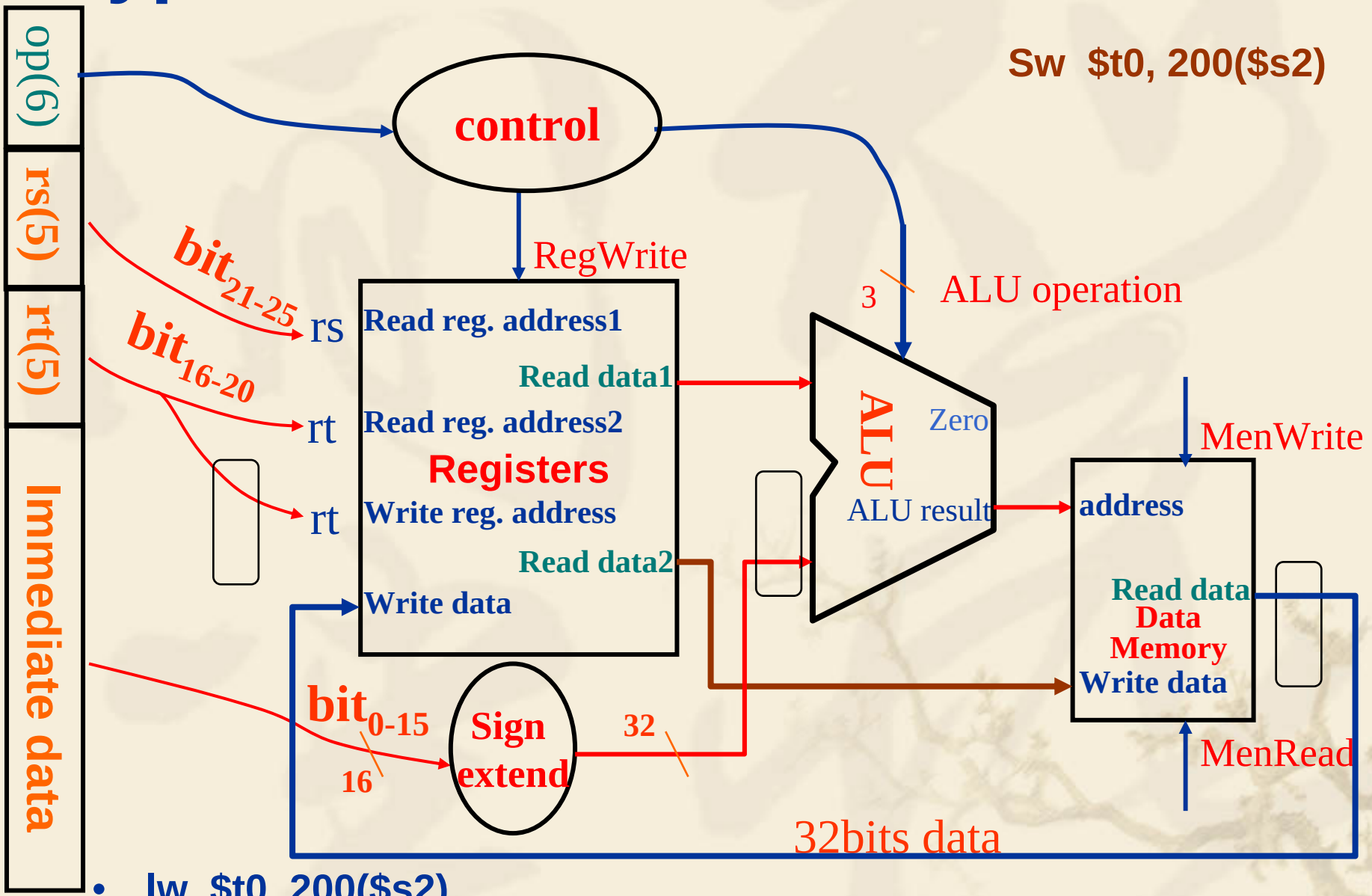
- First, Look at the data flow within instruction execution

R type Instruction & Data stream



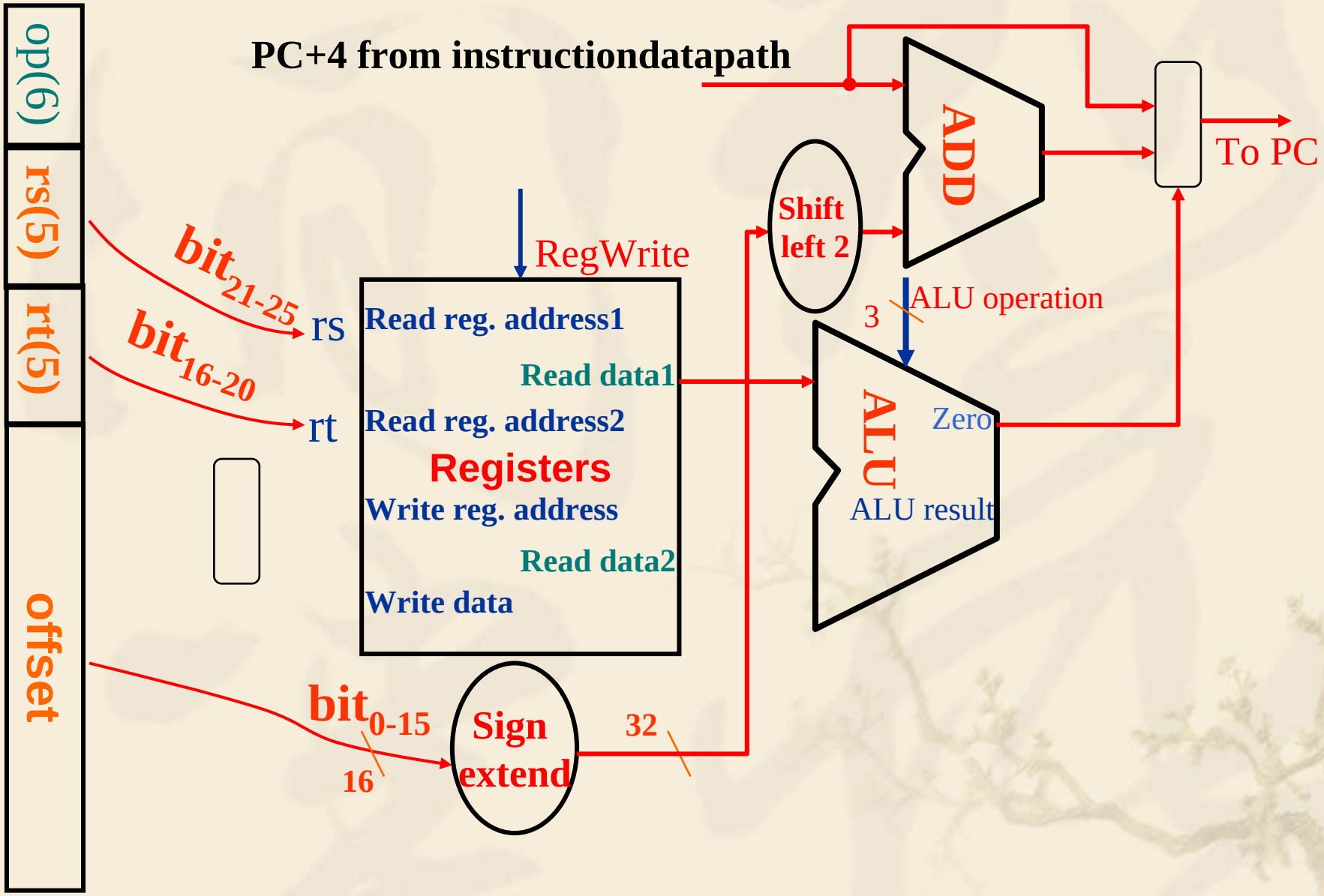
Bnegate	op	function
0	00	and
0	01	Or
0	10	Add
1	10	Sub
1	11	Slt

I type Instruction & Data stream

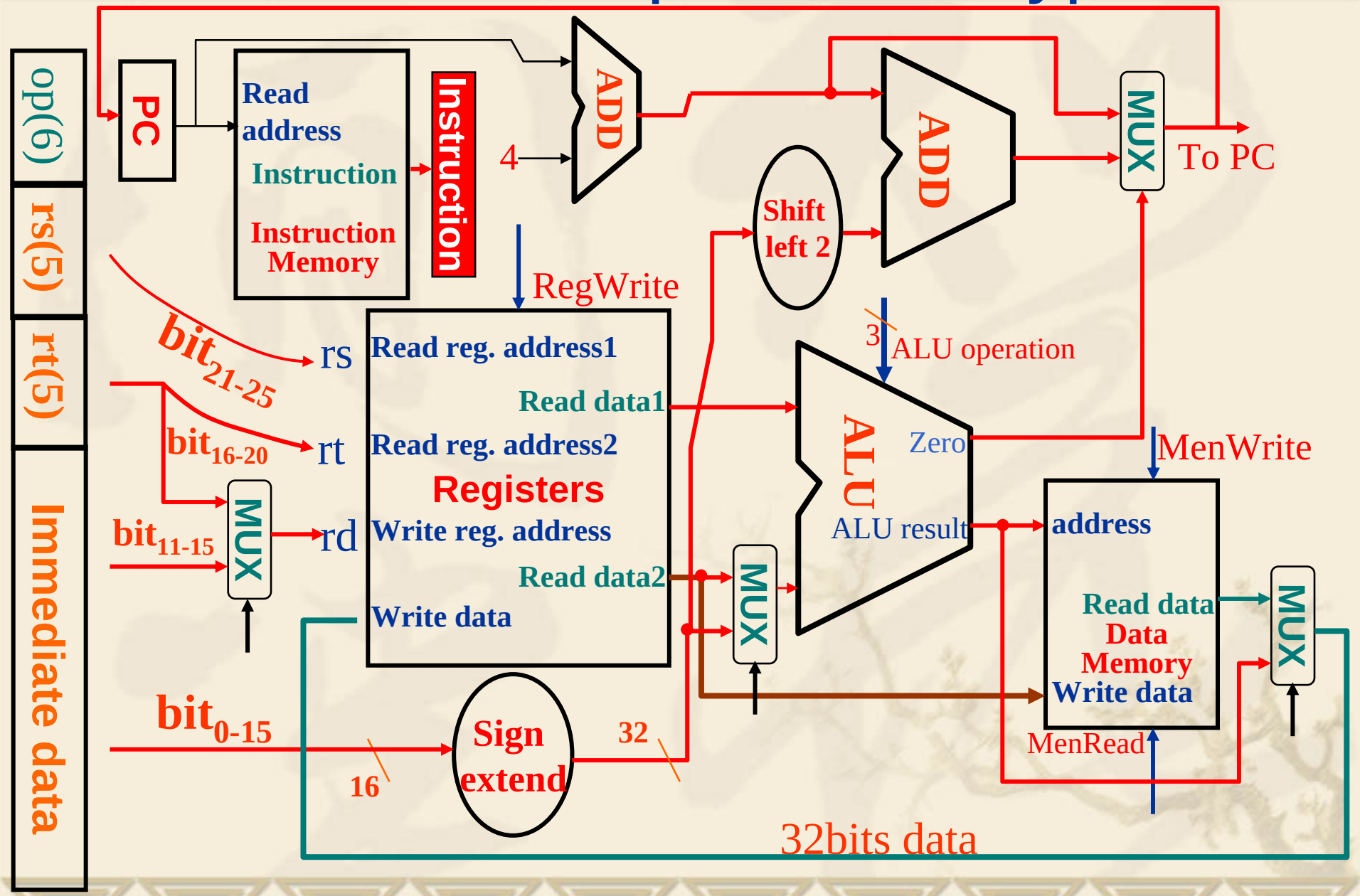


- lw $t0, 200($s2)
- if $s2=1000，it will load word in element number 1200 to $t0

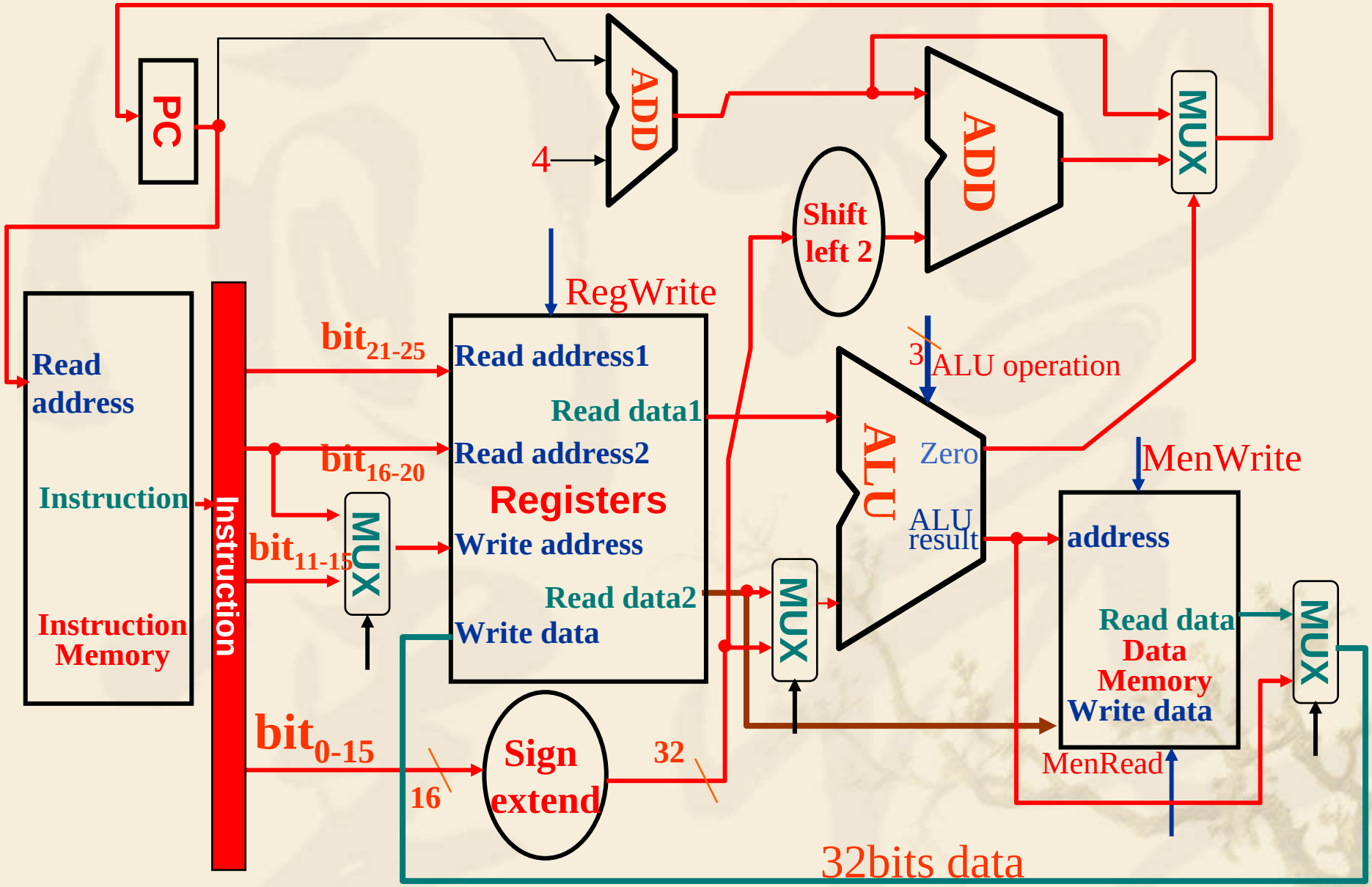
I type Instruction & Data stream of *beq*



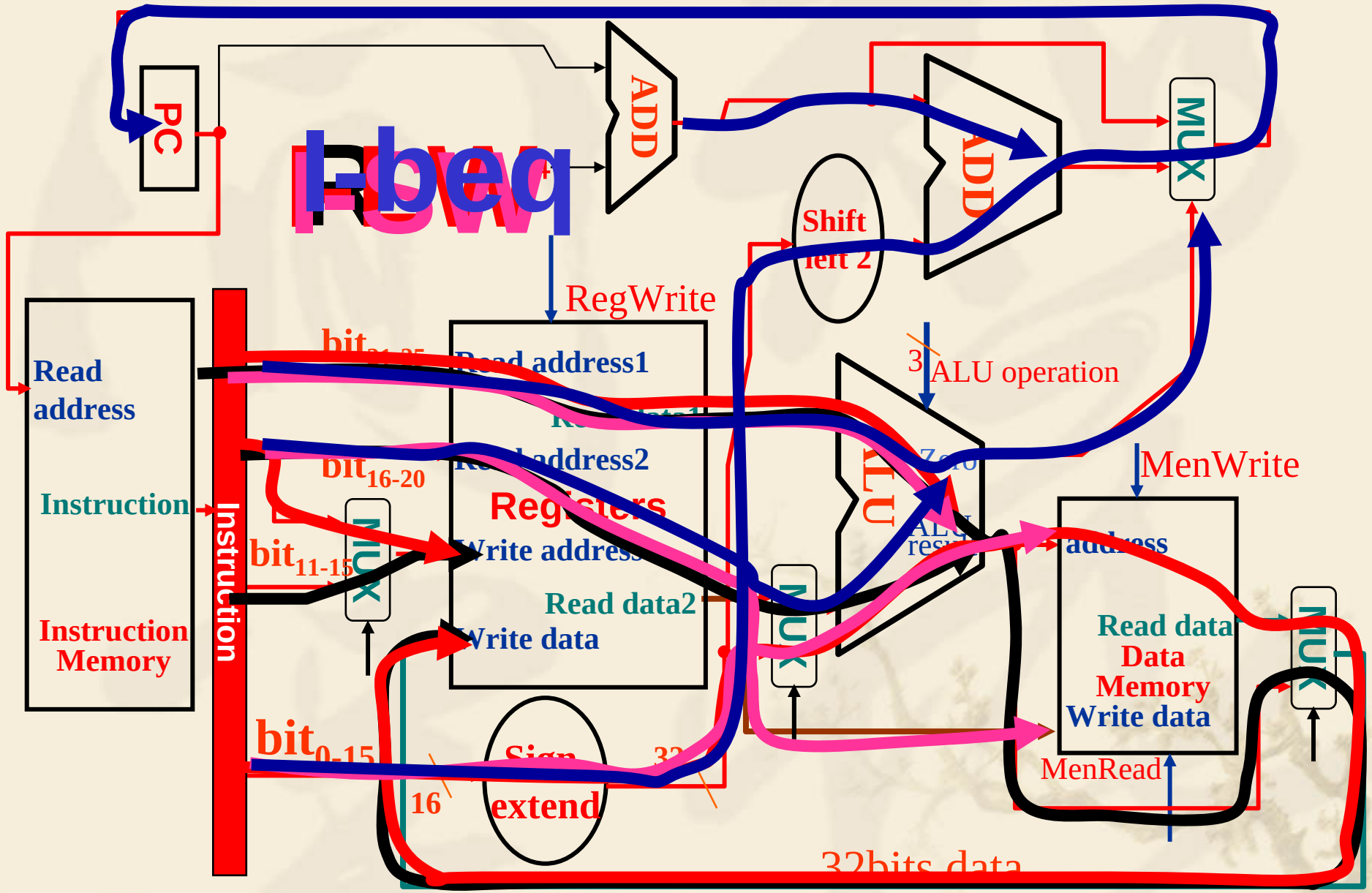
Combine the datapath R & I type



Combine the datapath R & I type



Combine the datapath R & I type



Chapter Five

The processor : Datapath and control

5.1 Introduction

5.2 Logic Design Conventions (skip)

5.3 Building a datapath

5.4 A Simple Implementation Scheme

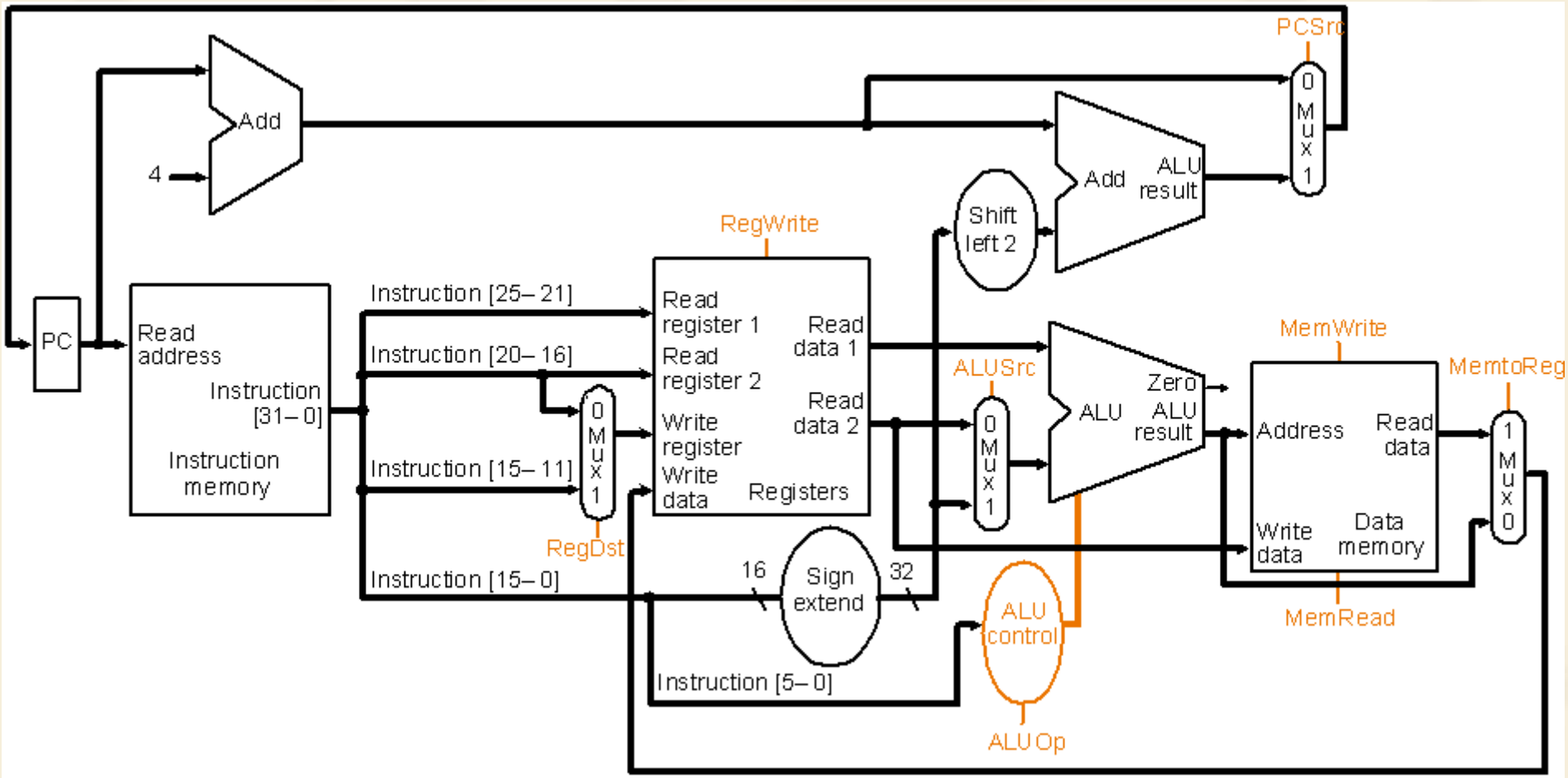
5.5 A Multicycle Implementation

5.5 Microprogramming

5.6 Exception

Building the Datapath

- Use multiplexors to stitch them together



Note : control signals Page 306 F5.16

Building Control

Analyse for cause and effect

- Information comes from the 32 bits of the instruction
- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- ALU's operation based on instruction type and function code

R-format instruction (add, sub, and, or, slt)

31–26	25–21	20–16	15–11	10–6	5–0
Op	Rs	Rt	Rd	Shamt	Funct
6 bits	5bits	5bits	5bits	5bits	6bits

I-format instruction (lw, sw, beq)

31–26	25–21	20–16	15–0
Op	Rs	Rt	Immediate
6 bits	5bits	5bits	16bits

J-format instruction (add, sub, and, or, slt)

31–26	25–0
Op	address
6 bits	26bits

Instruction Code P103

Instruction Code								
		31 21	25 21	25 16	15 11	10 6	5 0	
add	R	000000	rs	rt	rd	00000	100000	
sub	R	000000	rs	rt	rd	00000	100010	
and	R	000000	rs	rt	rd	00000	100100	
or	R	000000	rs	rt	rd	00000	100101	
slt	R	000000	rs	rt	rd	00000	101010	
lw	I	100011	rs	rt	Immediate(displacement)			
sw	I	101011	rs	rt	Immediate(displacement)			
beq	I	000100	rs	rt	Immediate(offset)			
j	J	000010	address					

What should ALU do ?

- e.g. what should the ALU do with these instructions
- Example: lw $1, 100($2)

	OP	rs	rt	16 bit displacement			ALU op
lw	(100011)35	2	1	100			00
sw	(101011)43	2	1	100			00
R	000000(00)	rs(5)	rt(5)	rd(5)	shamt	func(6)	10

- ALU control input

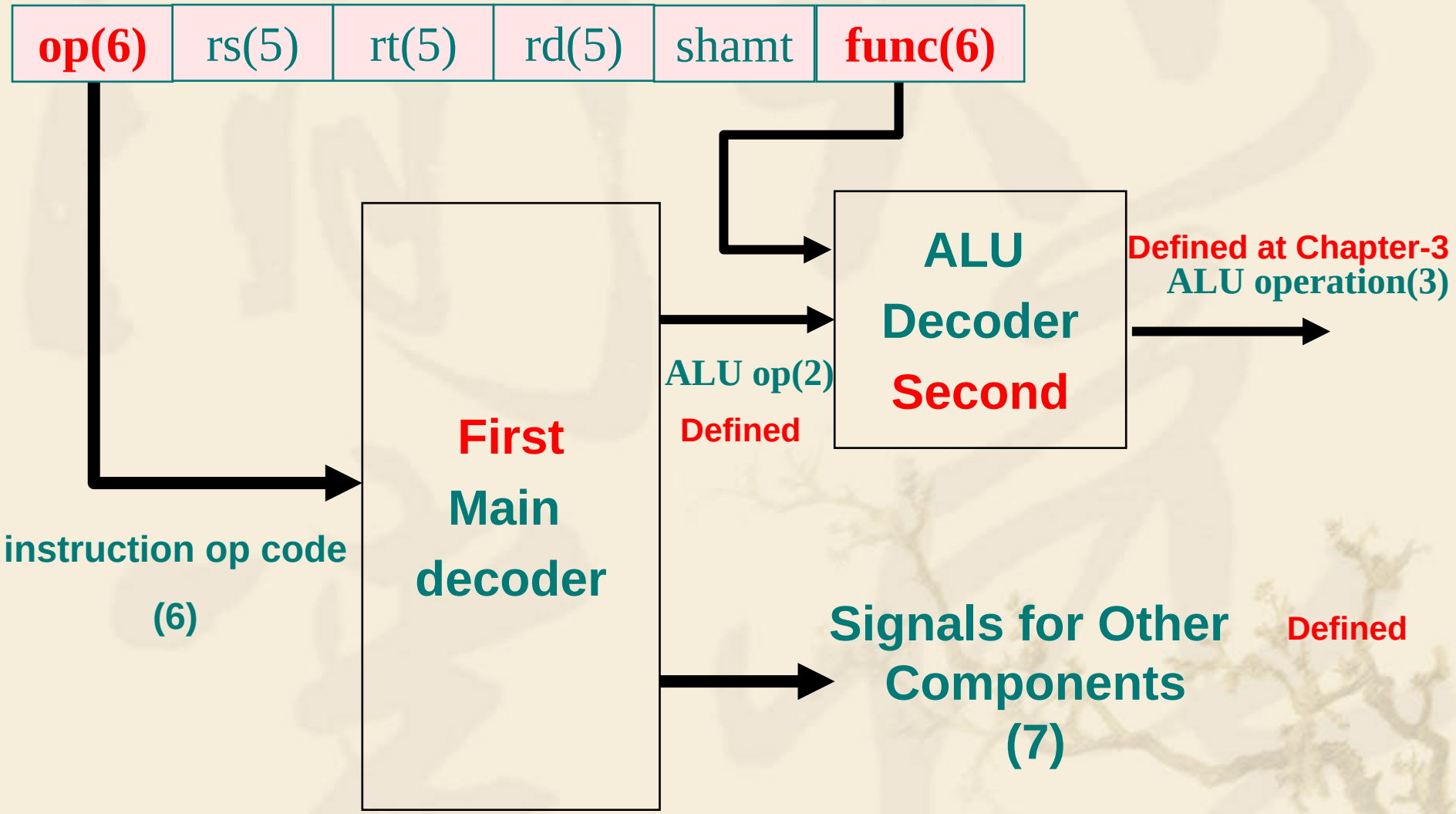
  3 -types

B negate	op	function
0	00	and
0	01	Or
0	10	Add
1	10	Sub
1	11	Slt

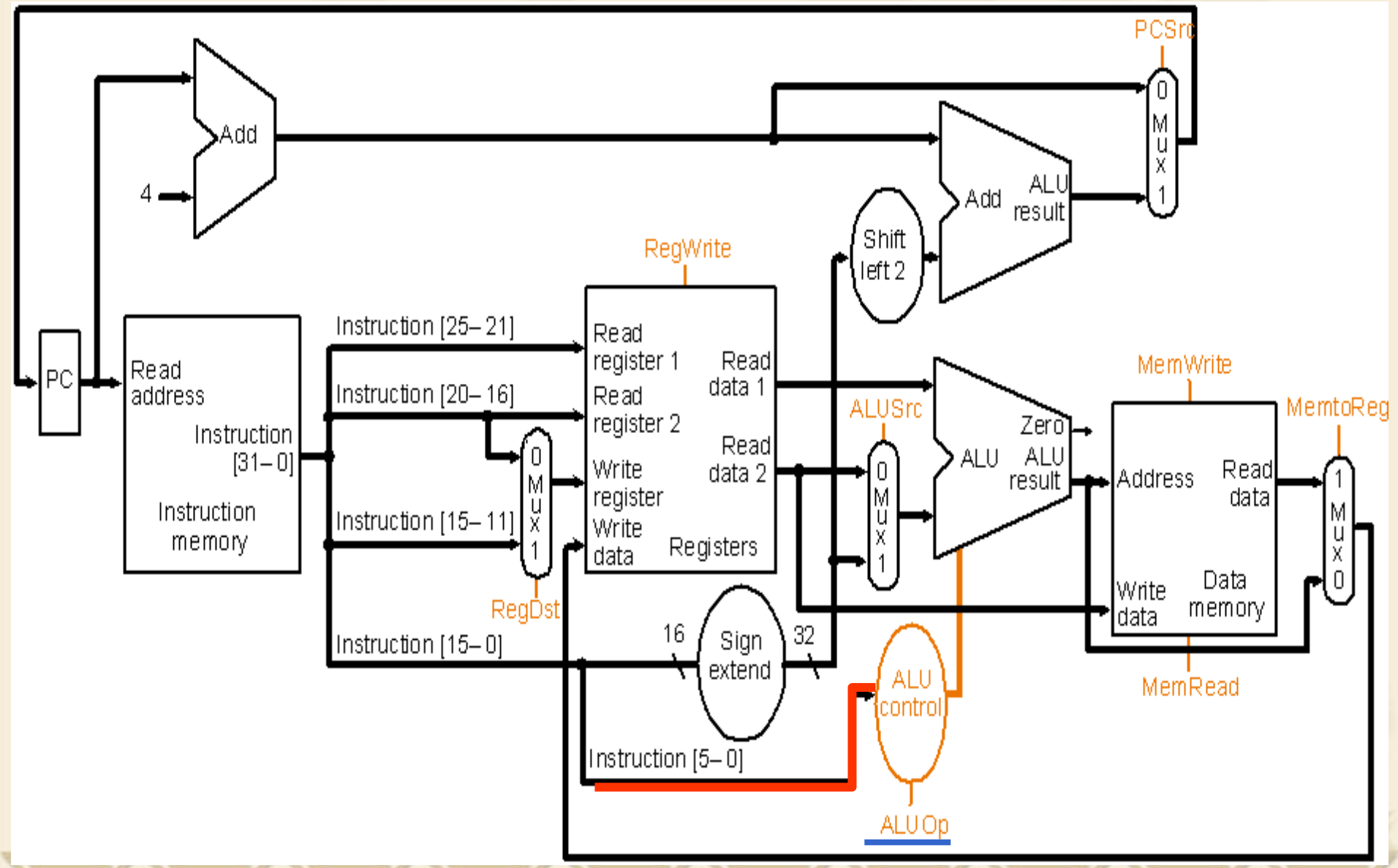
- Why is the code for subtract 110 and not 011?

Scheme of Controller

- 2-level decoder



The ALU control is where and other signals(7)



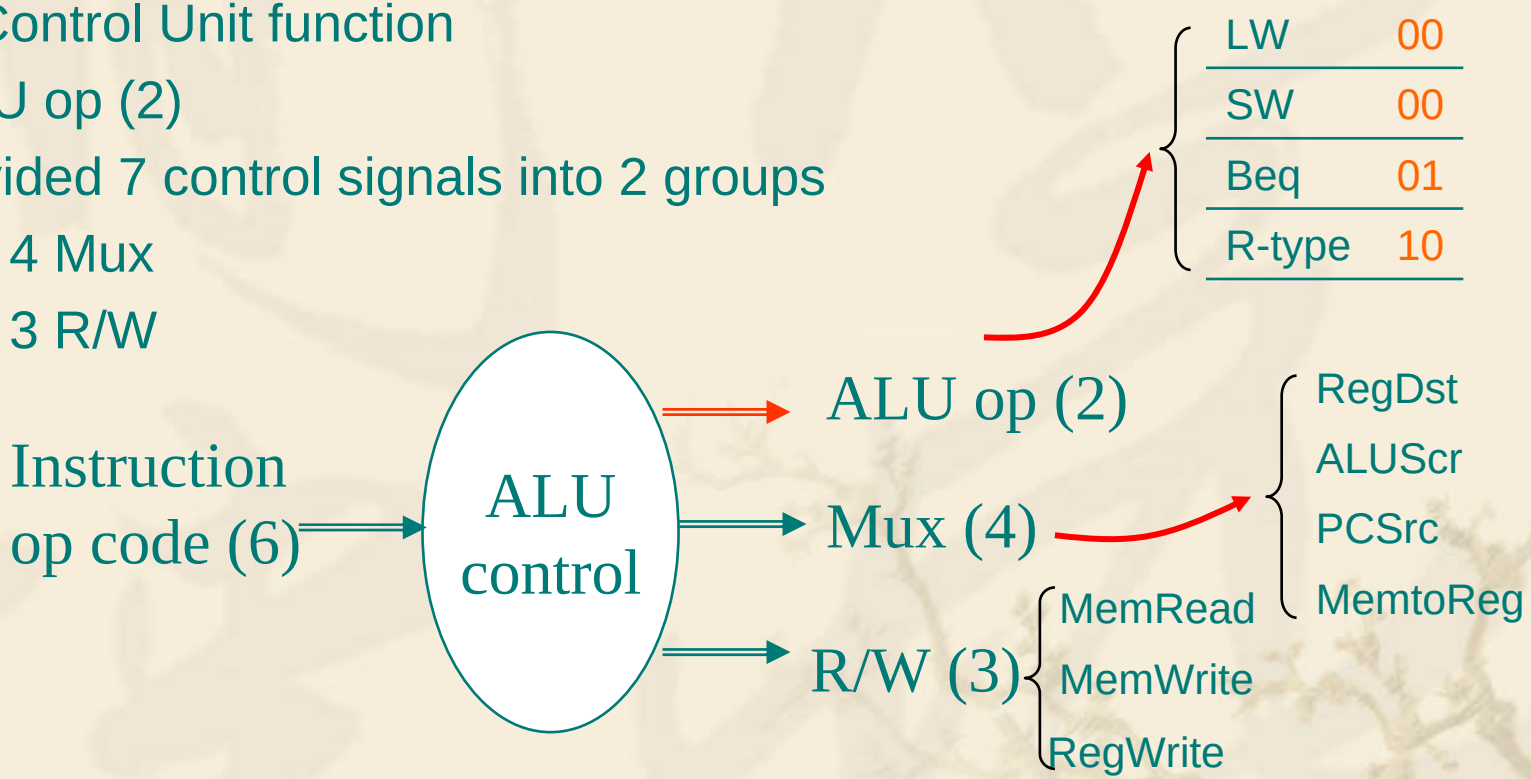
signals for datapath

Defined 7+2 control (p. 305)

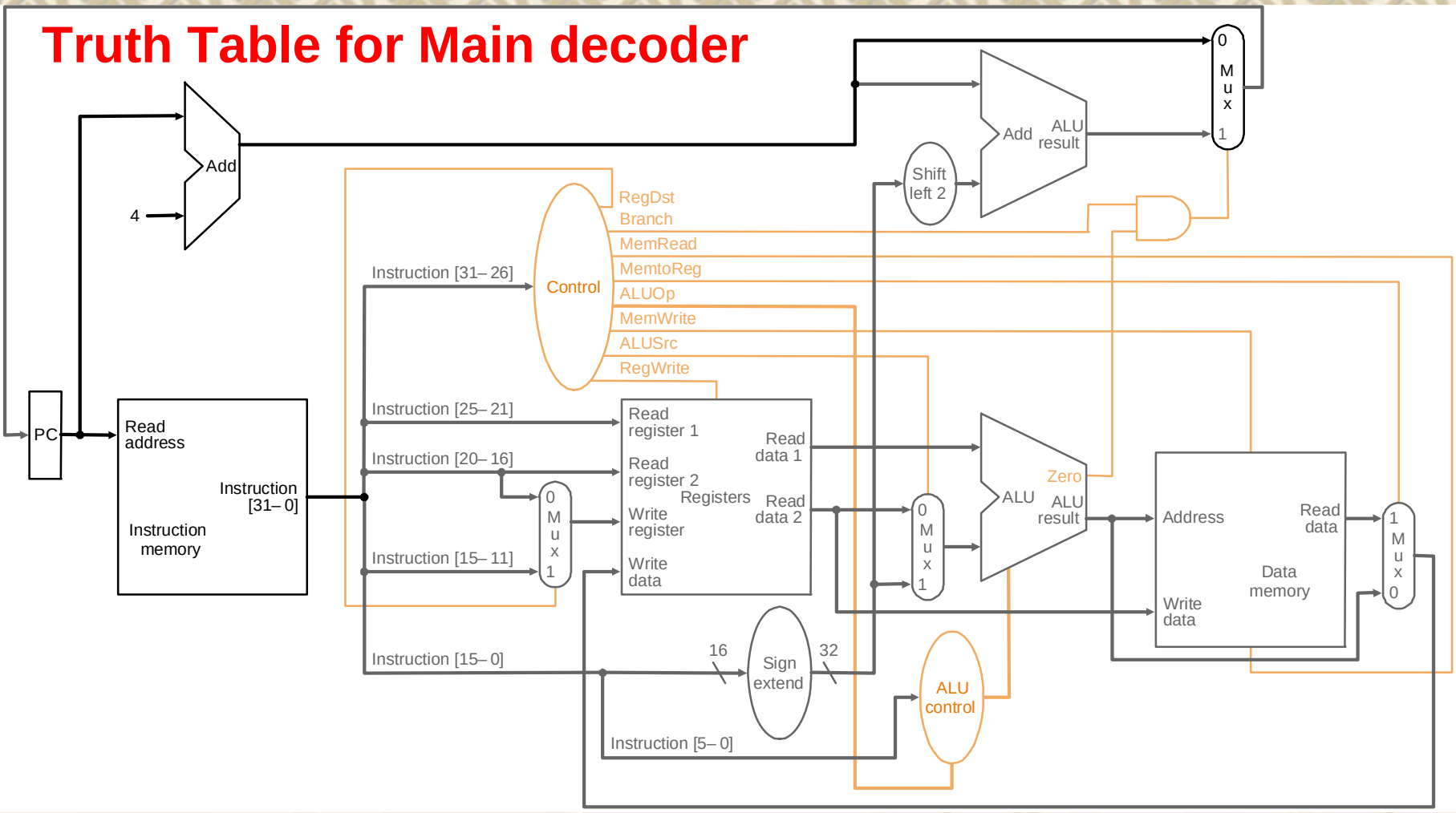
Signal name	Effect when deasserted(=0)	Effect when asserted(=1)
RegDst	Select register destination number from the rt(20:16) when WR	Select register destination number from the rd(15:11) when WB
RegWrite	None	Register destination input is written with the value on the Write data input
ALUScr	The second ALU operand come from the second register file output (Read data 2)	The second ALU operand is the sign-extended lower 16 bits of the instruction..
PCSrc	The PC is replaced by the output of the adder that computers the value PC+4	The PC is replaced by the output of the adder that computers the branch target.
MemRead	None	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None	Data memory contents designated by the address input are replaced by value on the Write data input.
MemtoReg	The value fed to register Write data input comes from the Alu	The value fed to the register Write data input comes from the data memory.

Designing the Main Control Unit (First level)

- Main Control Unit function
  - ALU op (2)
  - Divided 7 control signals into 2 groups
    - 4 Mux
    - 3 R/W



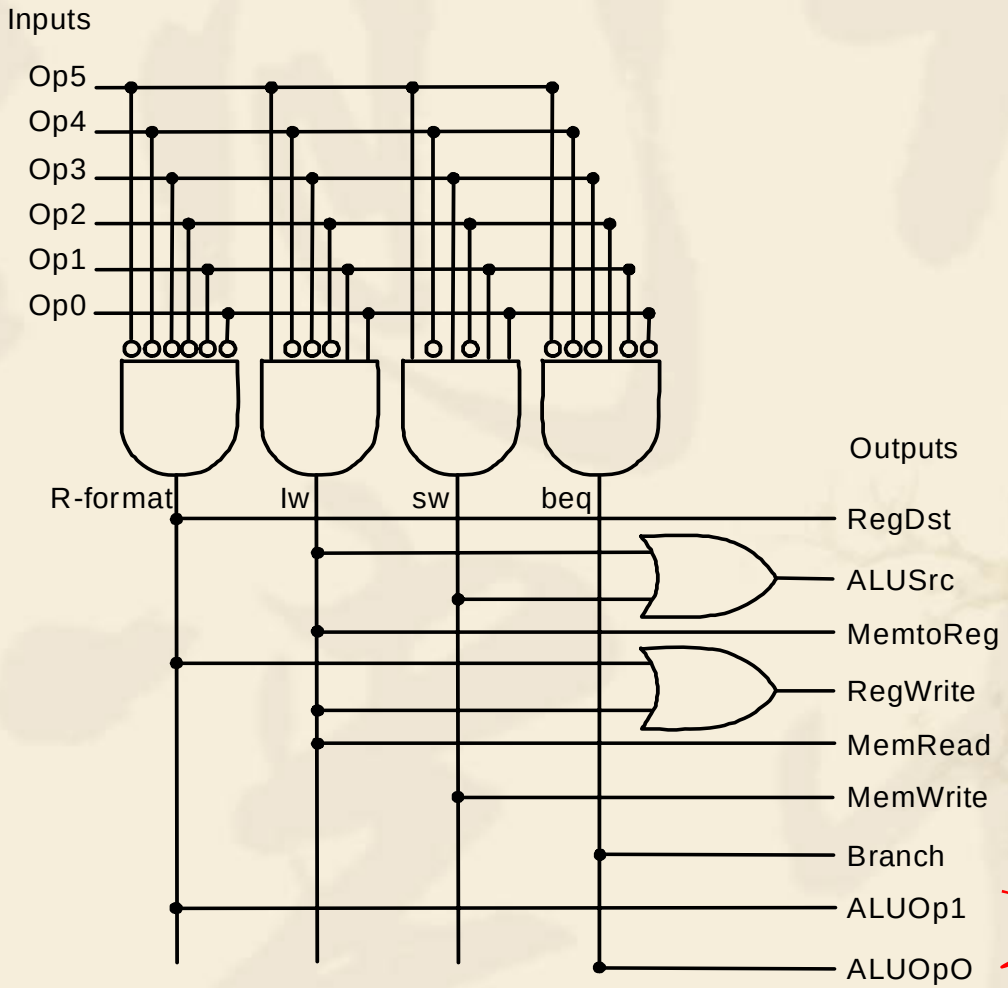
Truth Table for Main decoder



Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALU_{op1}	ALU_{op0}
beq	X	0	X	0	0	0	1	0	1
R-format	1	0	0	1	0	0	0	1	0
LW	0	1	1	1	1	0	0	0	0
SW	X	1	X	0	0	1	0	0	0

Circuitry of main Controller

- Simple combinational logic (truth tables)



opcode	output	
000000	R-format	
100011	lw	
101011	sw	
000100	beq	

L/S 00

beq 01

R-type 10

Designing the ALU decoder (Second level)

- Must describe hardware to compute 3-bit ALU conrol input

Instruction opcode	ALUop	Instruction operation	Funct field	Desired ALU action	ALU control Input
LW	00	Load word	xxxxxx	Load word	0010
SW	00	Store word	xxxxxx	Store word	0010
Beq	01	branch equal	xxxxxx	branch equal	0110
R-type	10	add	100000	add	0010
R-type	10	subtract	100010	subtract	0110
R-type	10	AND	100100	AND	0000
R-type	10	OR	100101	OR	0001
R-type	10	Set on less than	101010	Set on less than	0111

Truth Table for ALU decoder

- Describe it using a truth table (can turn into gates):

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	*210*
0	0	X	X	X	X	X	X	**010**
X	1	X	X	X	X	X	X	**110**
1	X	X	X	0	0	0	0	**010**
1	X	X	X	0	0	1	0	**110**
1	X	X	X	0	1	0	0	**000**
1	X	X	X	0	1	0	1	**001**
1	X	X	X	1	0	1	0	**111**

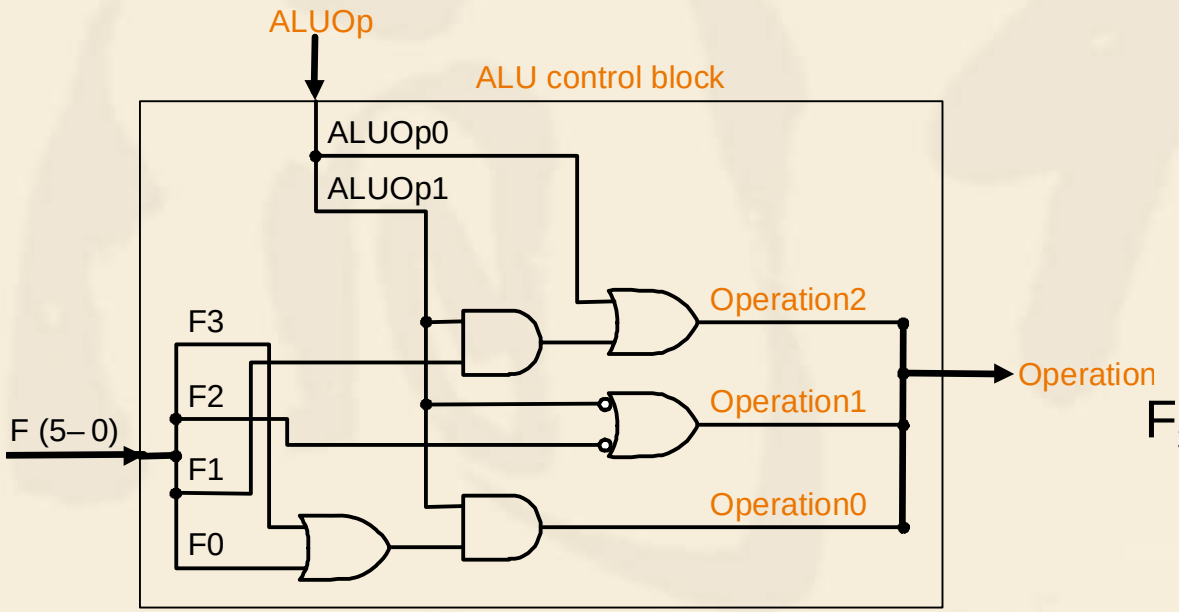
don't care

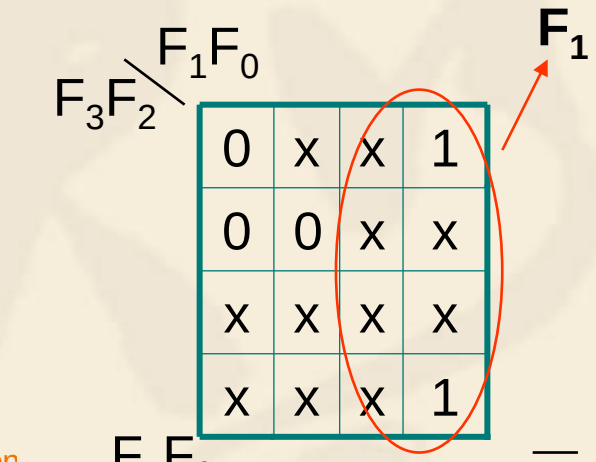
$$Operation_2 = ALU_{op0} + ALU_{op1}(\overline{F}_3 \overline{F}_2 F_1 \overline{F}_0 + F_3 \overline{F}_2 F_1 \overline{F}_0)$$

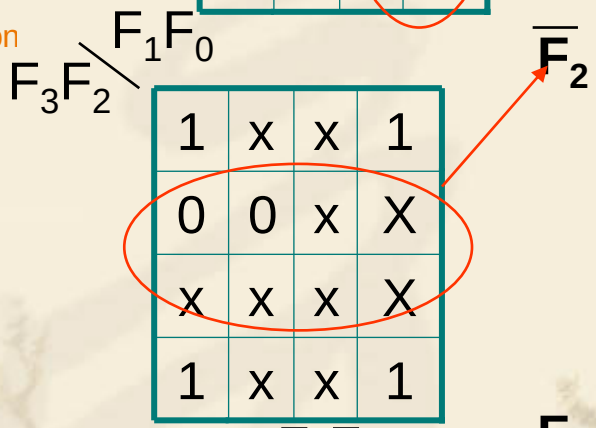
$$Operation_1 = \overline{ALU_{op1} \overline{F}_3 F_2 \overline{F}_1 \overline{F}_0 + ALU_{op1} \overline{F}_3 F_2 \overline{F}_1 F_0}$$

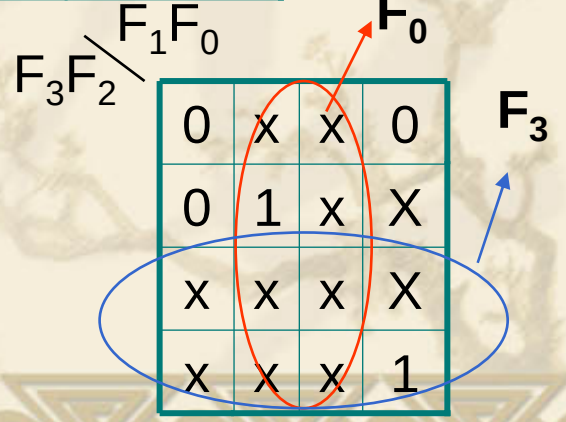
$$Operation_0 = ALU_{op1} \overline{F}_3 F_2 \overline{F}_1 F_0 + ALU_{op1} F_3 \overline{F}_2 F_1 \overline{F}_0$$

The ALU control signals----logic circuit









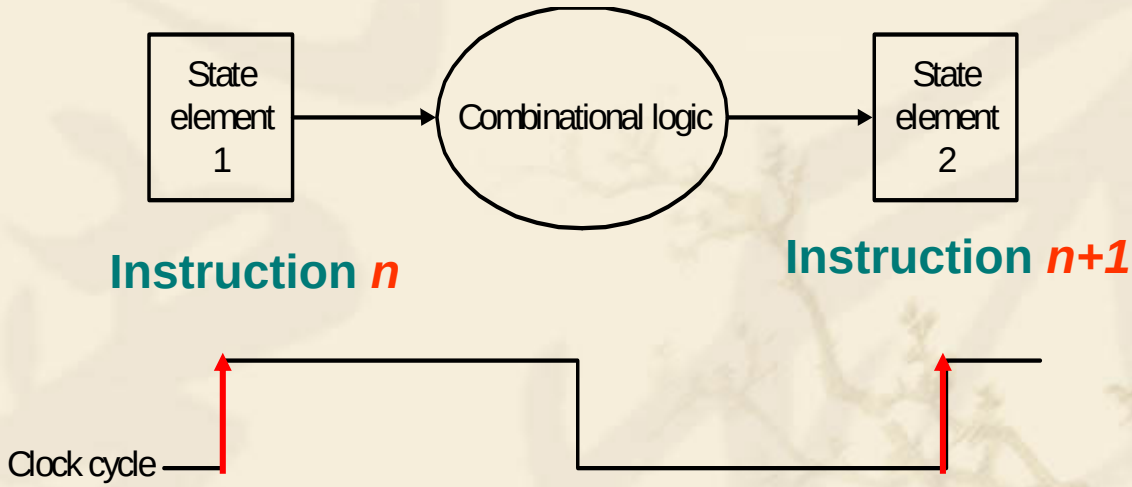
$$Operation_2 = ALU_{op0} + ALU_{op1}F_1$$

$$Operation_1 = \overline{ALU}_{op1} + \overline{F}_2$$

$$Operation_0 = ALU_{op1} + (F_0 + F_3)$$

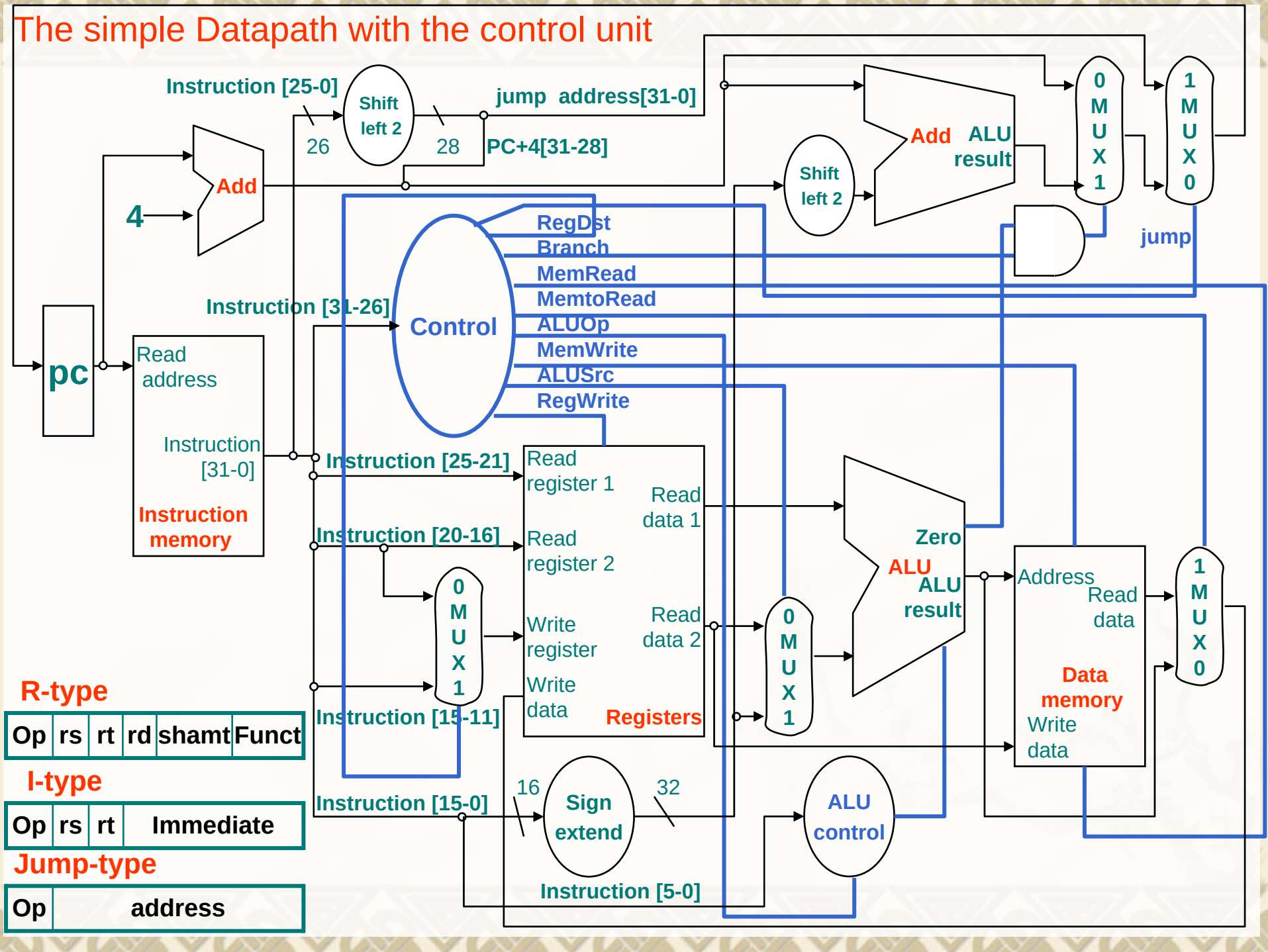
Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer? right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

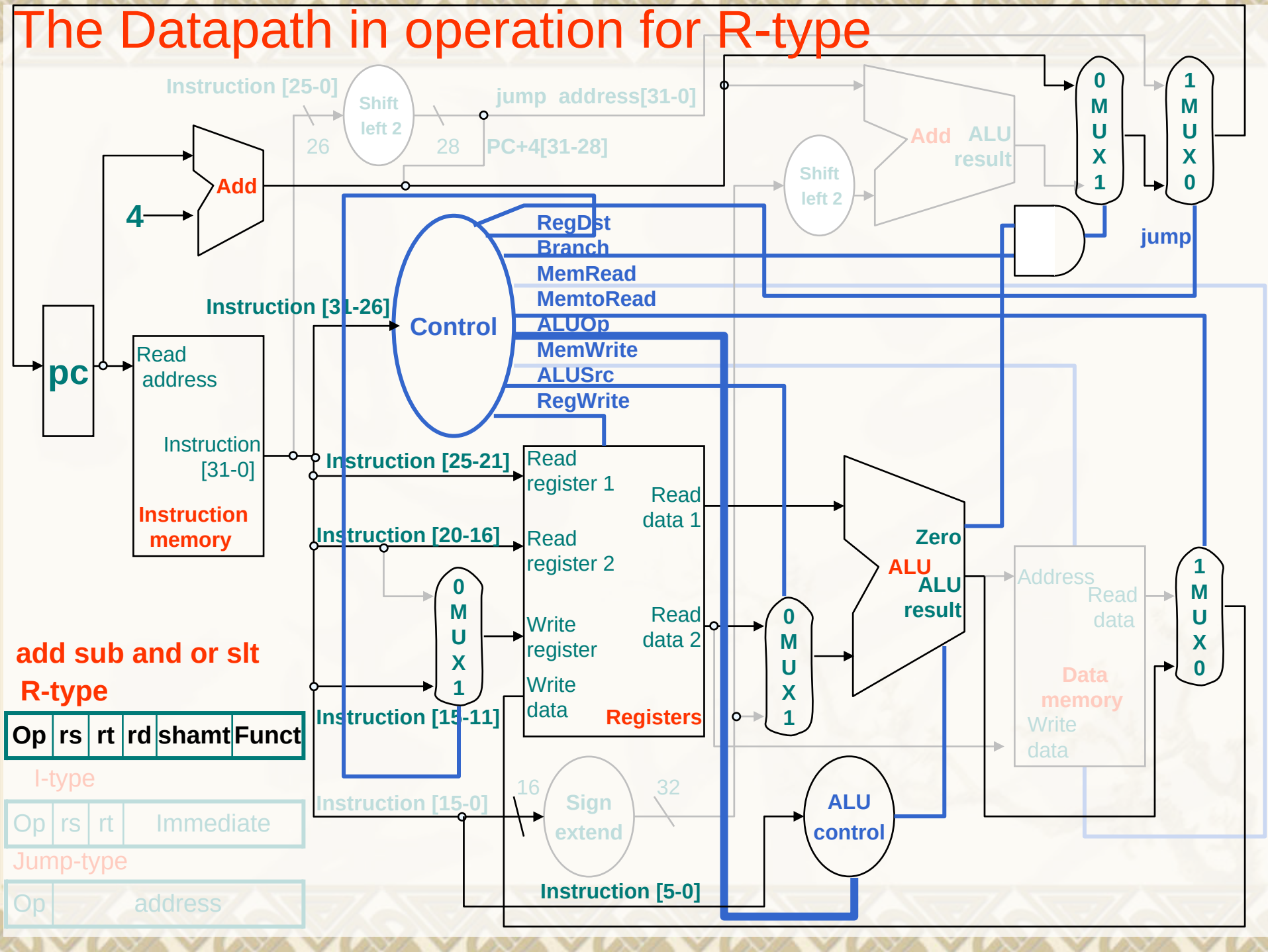


We are ignoring some details like setup and hold times

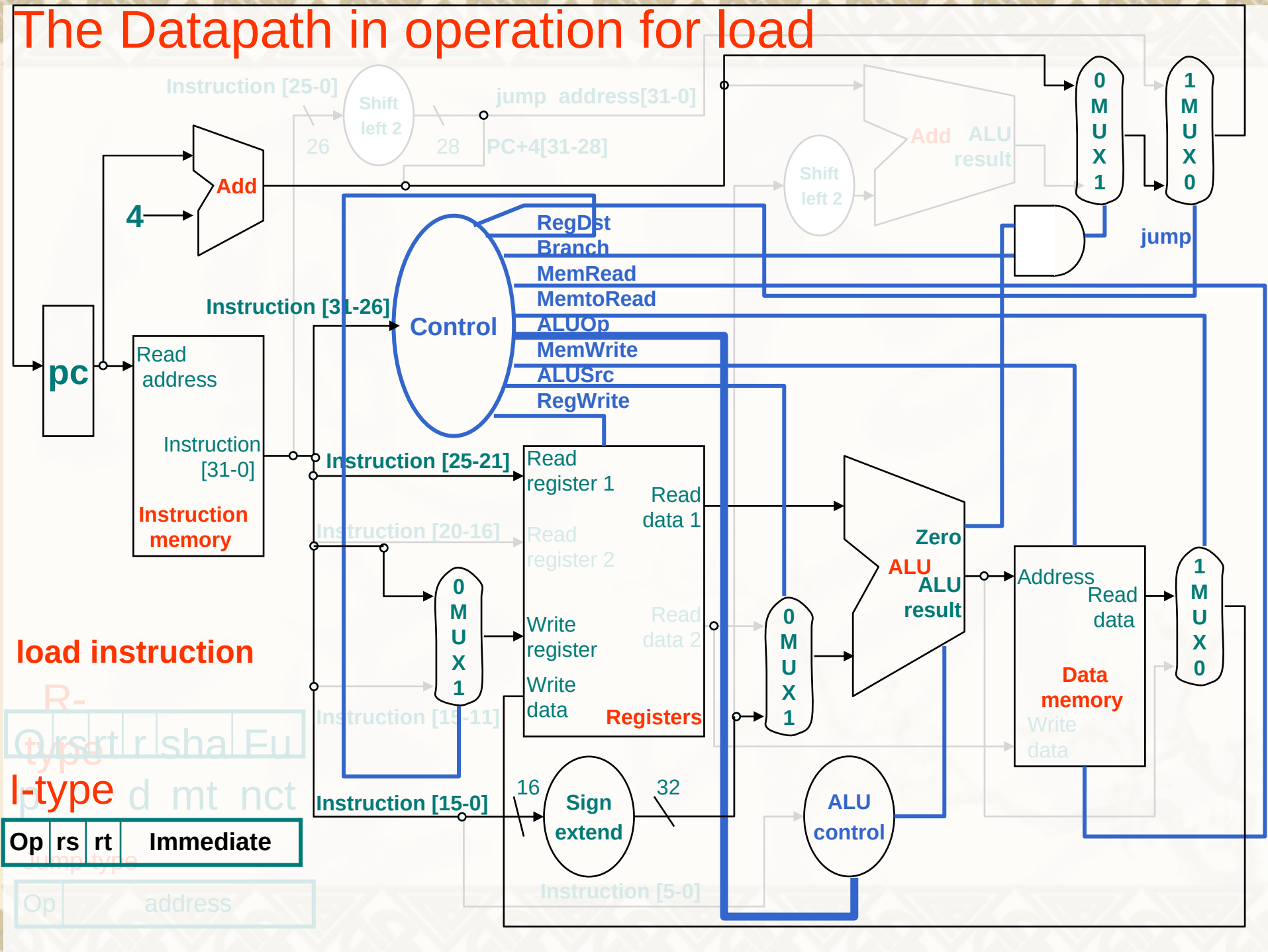
The simple Datapath with the control unit



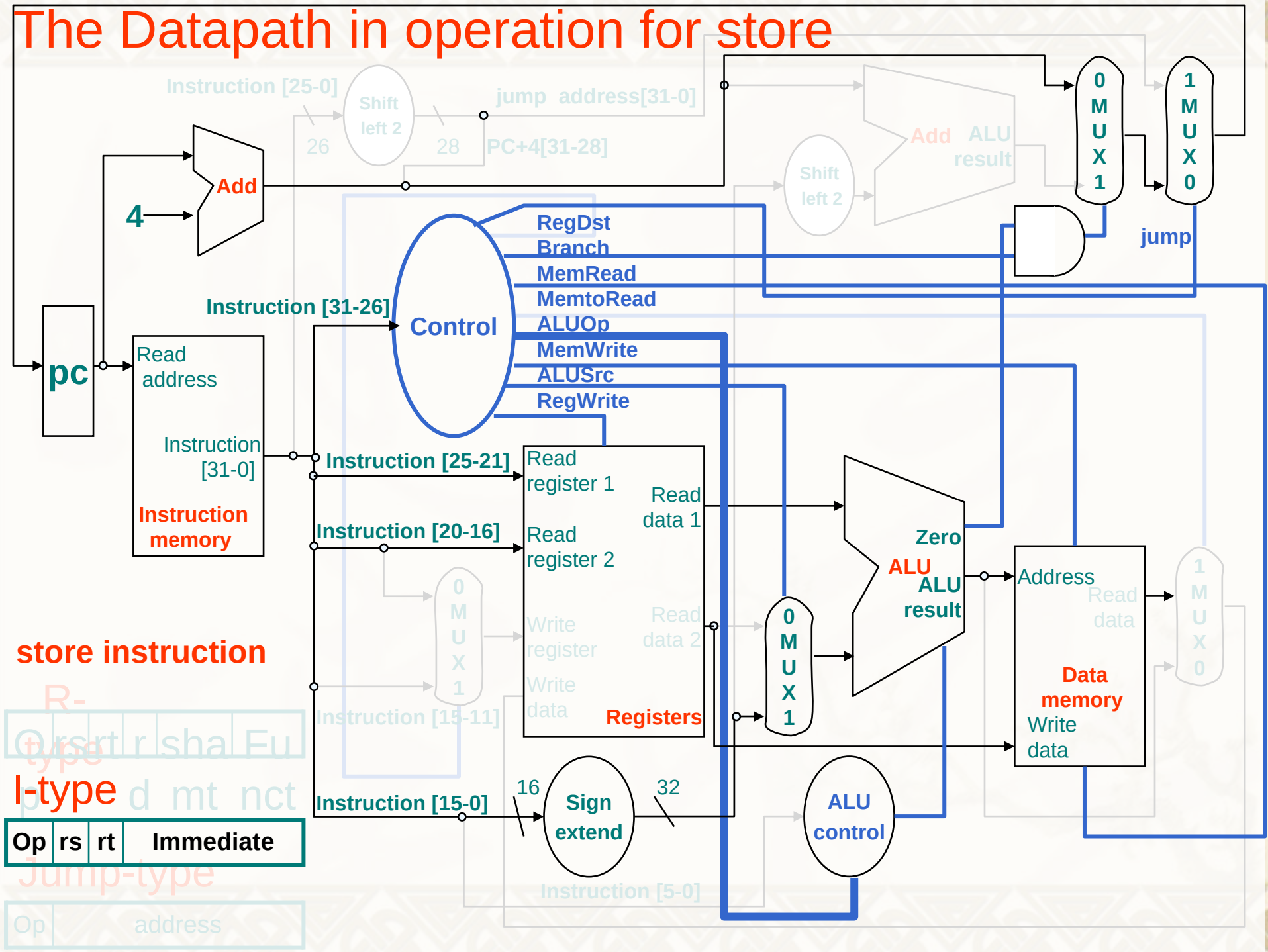
The Datapath in operation for R-type



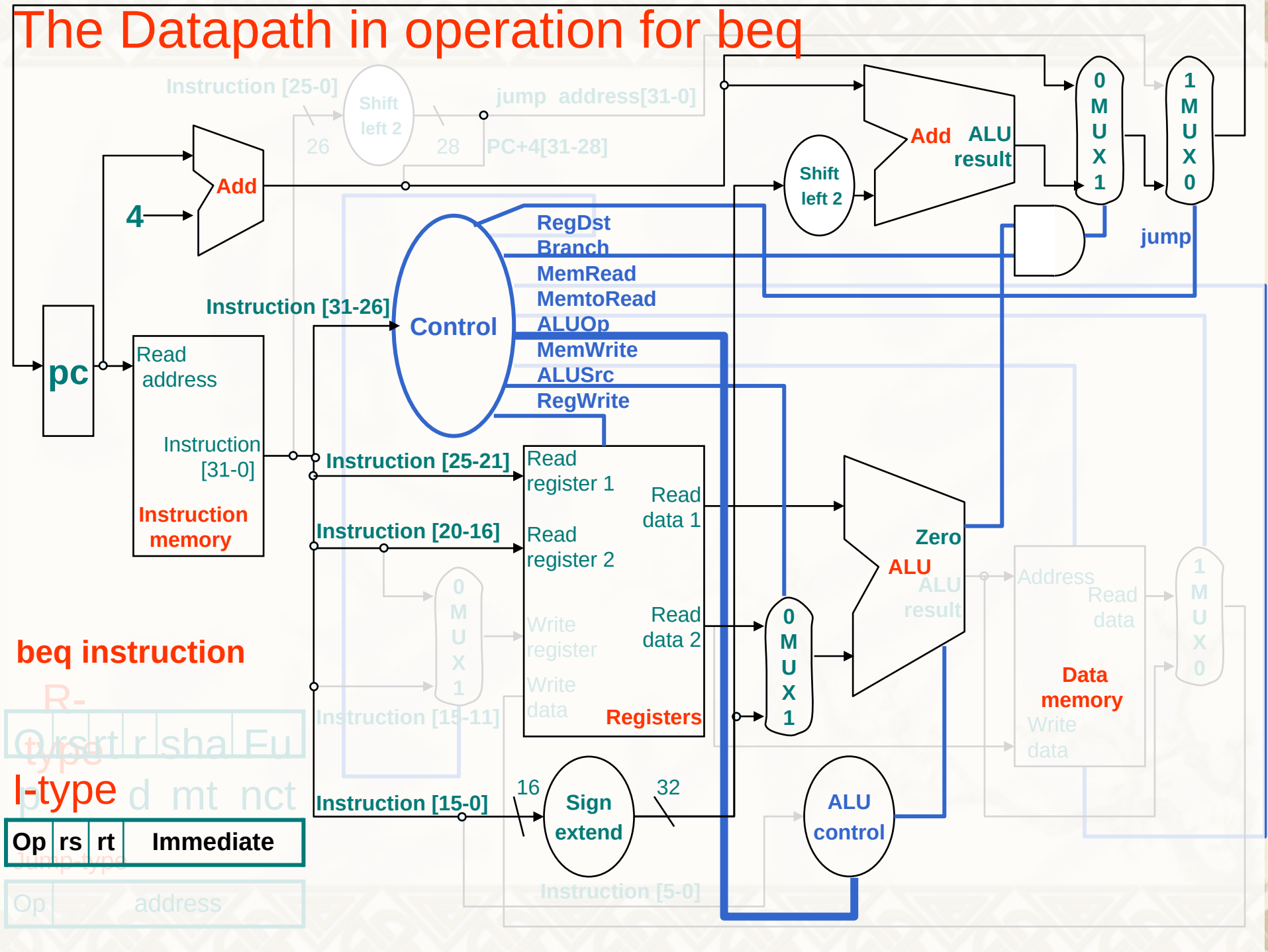
The Datapath in operation for load



The Datapath in operation for store



The Datapath in operation for beq



beq instruction

I-type

Op	rs	rt	Immediate

j instruction

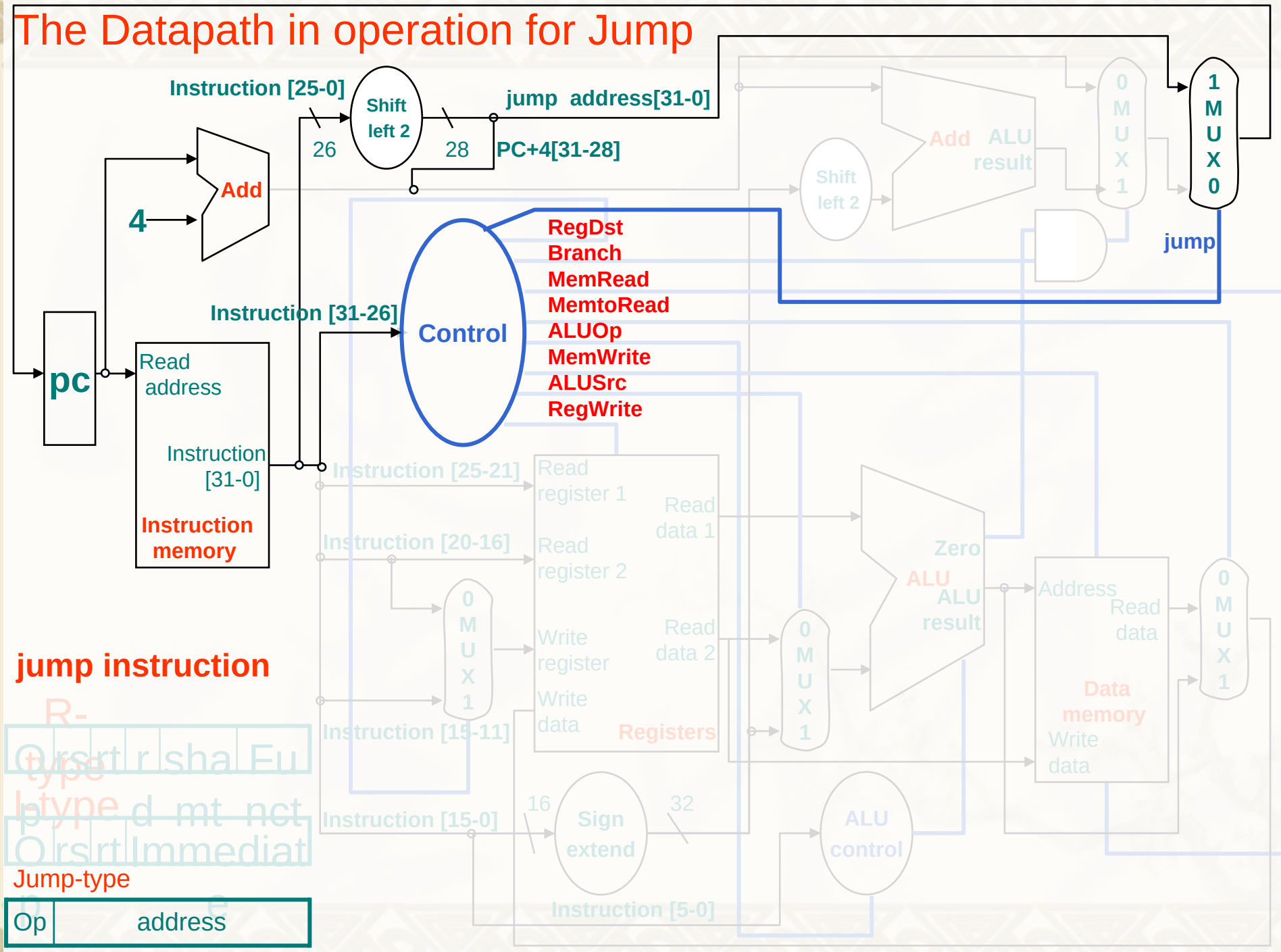
- instruction format
  - j Label

(000010)2	26 bits address

- Implementation

  $pc = pc_{28\sim31}$ ## 26bits-address $\times$ 4

The Datapath in operation for Jump



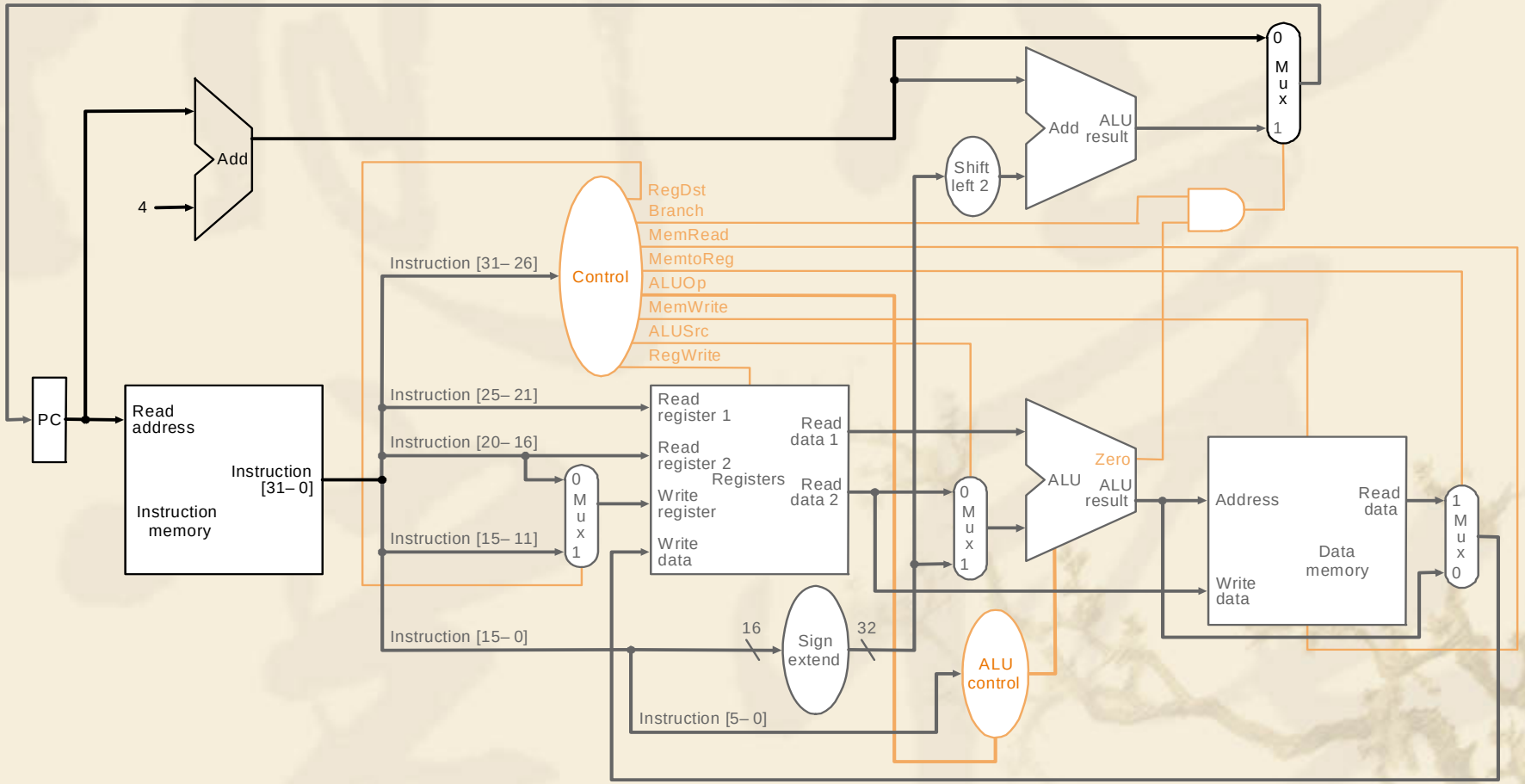
jump instruction

Jump-type

Op	address

Single Cycle Implementation performance for lw

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)



2ns 1+1=ns 2ns 2ns

Performance in Single Cycle Implementation

- Let's see the following table:

Instruction class	Instruction memory	Register read	ALU	Data memory	Register write	Total
R-format	2	1	2		1	6 ns
Load word	2	1	2	2	1	8 ns
Store word	2	1	2	2		7 ns
Branch	2	1	2			5 ns
Jump	2					2 ns

- **The conclusion:**

  Different instructions needs different time.

  The clock cycle must meet the need of the slowest instruction. So,some time will be wasted.

The CPU Performance Equation

$$\text{CPU time} = \text{CPU clock cycles for a program} \times \text{Clock cycle time}$$

$$\text{CPU time} = \frac{\text{CPU clock cycles for a program}}{\text{Clock rate}}$$

$$\text{CPU}_{\text{time}} = I \; \text{CPI} \; \tau$$

$$\text{CPI} = \frac{\text{CPU clock cycles for a program}}{\text{Instruction count}}$$

$$\text{CPU time} = \text{Instruction count} \times \text{Clock cycle time} \times \text{Cycles per instruction}$$

$$\text{CPU time} = \frac{\text{Instruction count} \times \text{Clock cycle time}}{\text{Clock rate}}$$

$$\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} = \frac{\text{Seconds}}{\text{Program}} = \text{CPU time}$$

- CPU performance is dependent upon three characteristics:
  - clock cycle (or rate)
  - clock cycles per instruction
  - and instruction count.
- It is difficult to change one parameter in complete isolation from others because the basic technologies involved in changing each characteristic are interdependent:
  - *Clock cycle time*—Hardware technology and organization
  - *Clock cycle time*—Hardware technology and organization
  - *Instruction count*—Instruction set architecture and compiler technology

MIPS *(million instruction per second)*

$$\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Clock rate}}{\text{CPI} \times 10^6}$$

$$\text{Execution time} = \frac{\text{Instruction count}}{\text{MIPS} \times 10^6}$$

The bigger the MIPS, the faster the machine.

- **Three problems with MIPS:**
  - MIPS is dependent on the instruction set, making it difficult to compare MIPS of computers with different instruction sets.
  - MIPS varies between programs on the same computer.
  - Most importantly, MIPS can vary inversely to performance!

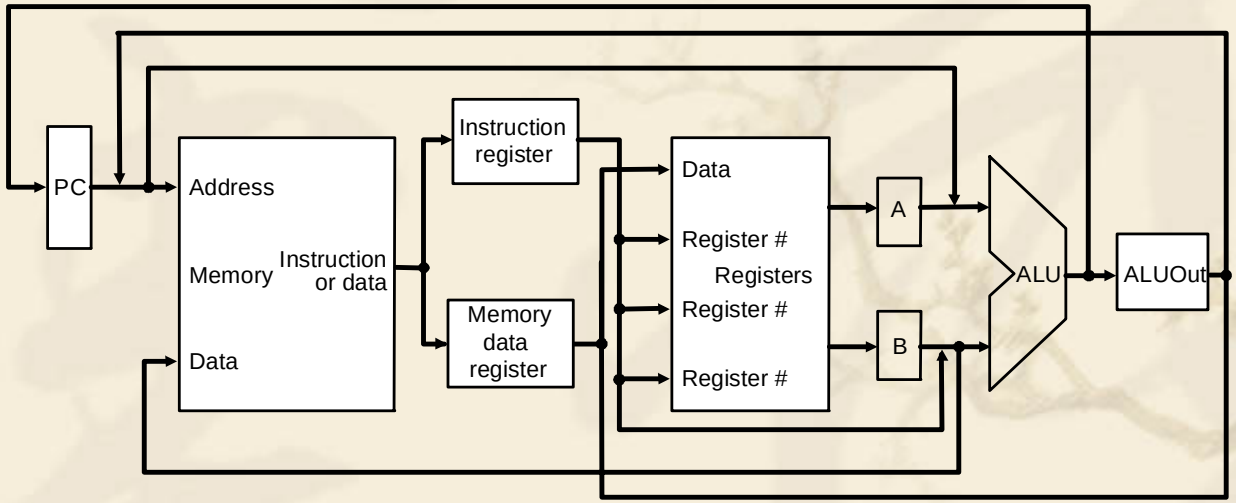
Single Cycle Problems

- :
  - what if we had a more complicated instruction like floating point?

    If so,the waste of time will be more serious.
  - wasteful of area. The reason is the following:
    - **Let's see the instruction 'mult'.This instruction needs to use the ALU repeatedly.**
    - **But,in the single cycle implementation,one ALU can be used only once in one clock cycle.**
    - **So,the instruction 'mult' will need many ALUs.The CPU will be very large.**

One Solution for Single Cycle Problems

- One Solution:
  - Use a smaller cycle time
  - Let different instructions take different numbers of cycles
- a Multicycle datapath:



Chapter Five

The processor : Datapath and control

5.1 Introduction

5.2 Logic Design Conventions (skip)

5.3 Building a datapath

5.4 A Simple Implementation Scheme

5.5 A Multicycle Implementation

5.5 Microprogramming

5.6 Exception

Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles
  - introduce additional internal registers

Analyse events: Five Execution Steps

- **IF：Instruction Fetch**
- **ID：Instruction Decode and Register Fetch**
- **EX（BC）：Execution, Memory Address Computation, or Branch Completion**
- **MEM（WB）：Memory Access or R-type instruction completion**
- **WB：Write-back step**

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- **Use PC to get instruction and put it in the Instruction Register.**
  - **`IR = Memory[PC];`**
- **Increment the PC by 4 and put the result back in the PC.**
  - **`PC = PC + 4;`**
- **Can be described simply using RTL "Register-Transfer Language"**

```
IR = Memory[PC];
PC = PC + 4;
```

- **Can we figure out the values of the control signals?**
- **What is the advantage of updating the PC now?**

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- **ALU is performing one of three functions, based on instruction type**

- **Memory Reference (lw / sw):**

```
ALUOut = A + sign-extend(IR[15-0]);
```

- **R-type:**

```
ALUOut = A op B;
```

- **Branch:**

```
if (A==B) PC = ALUOut;
```

- **jump:**

```
pc = pc31-28 + IR25-0 << 2
```

Step 4 (R-type or memory-access)

- **Loads and stores access memory**

```
MDR = Memory[ALUOut]; # for lw
```

or

```
Memory[ALUOut] = B;   # for sw
```

- **R-type instructions finish**

```
Reg[rd]=Reg[ IR[15-11] ] = ALUOut;
```

The write actually takes place at the end of the cycle on the edge

Write-back step (step 5)

- **`lw`**
  - **`Reg[rt]=Reg[IR[20-16]]= MDR;`**

What about all the other instructions?

Summary:

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) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
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		

Simple Questions

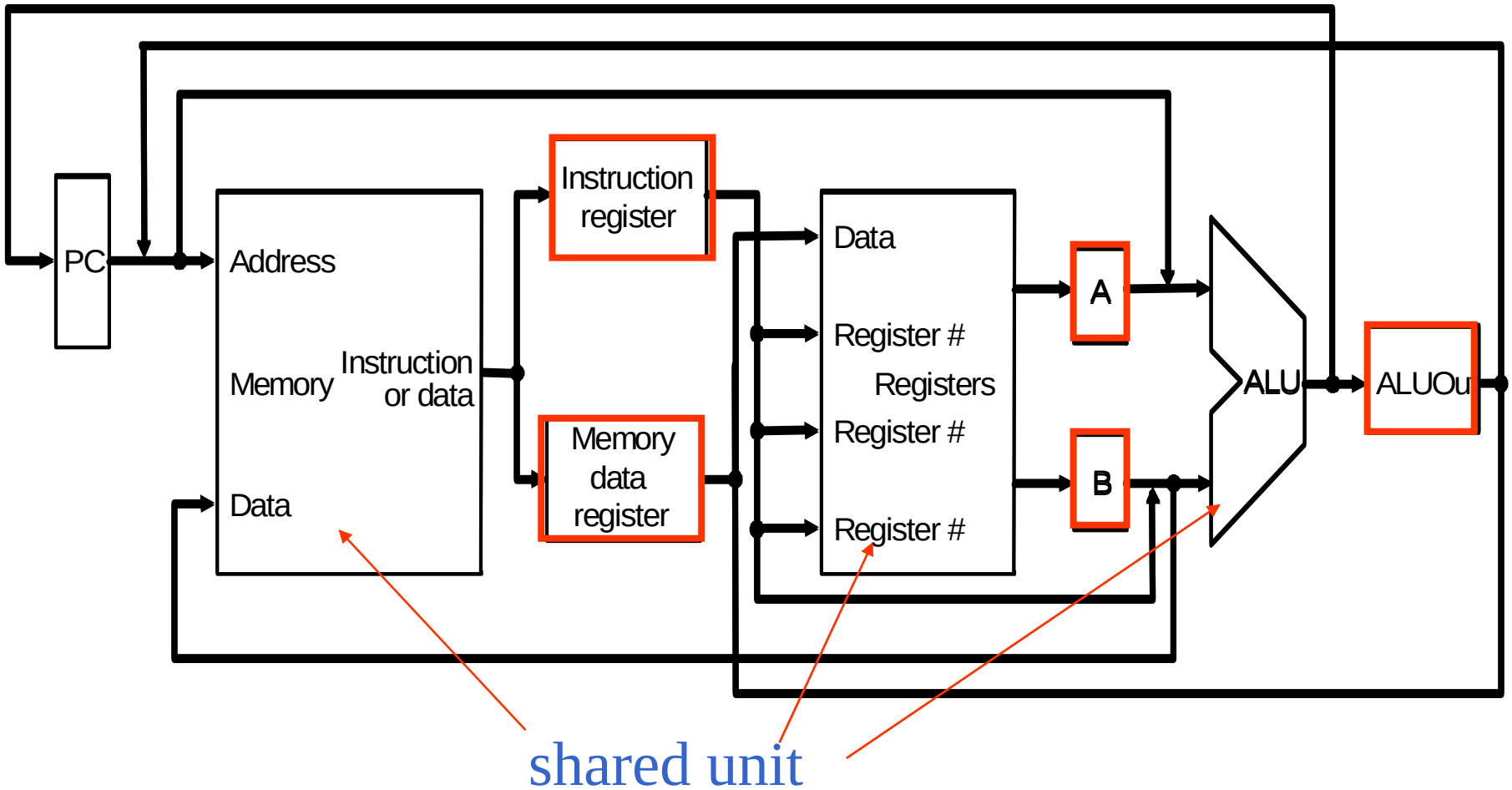
- How many cycles will it take to execute this code? (12)

```
            lw $t2, 0($t3)
            lw $t3, 4($t3)
            beq $t2, $t3, Label     #assume not
            add $t5, $t2, $t3
            sw $t5, 8($t3)
Label:      ...
```

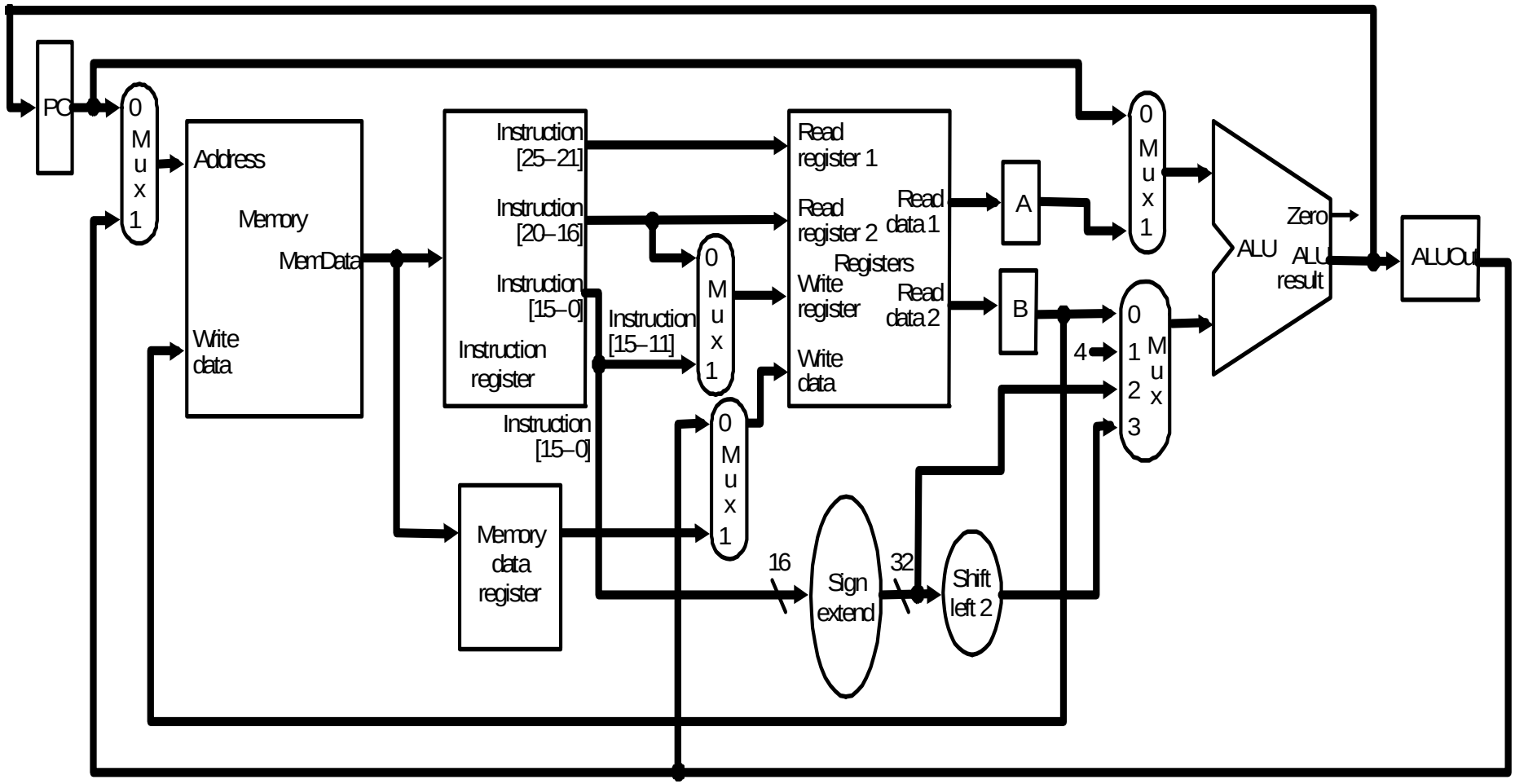
- What is going on during the 8th cycle of execution?
  - Answer: Calculating memory address
- In what cycle does the actual addition of `$t2` and `$t3` takes place?
  - No. 9

Reusing Resource

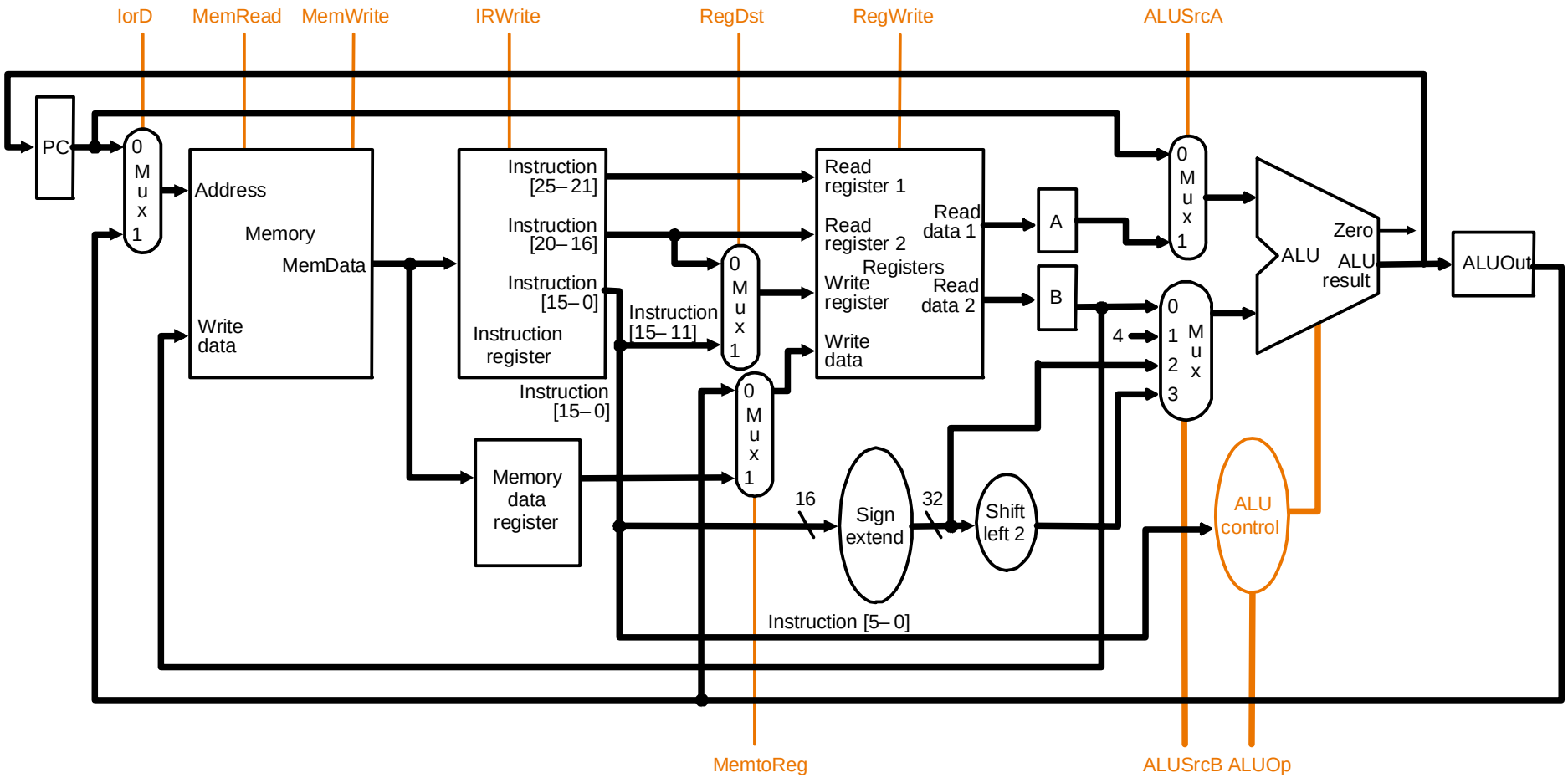
- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- We will use a finite state machine for control



Scheme of Controller of Multicycle



How does it control in Multicycle Approach



signals for datapath of Multicycle Defined 10+6 control (p. 324)

Signal name	Effect when deasserted(=0)	Effect when asserted(=1)
RegDst	Select register destination number from the rt(20:16) when WR.	Select register destination number from the rd(15:11) when WB.
RegWrite	None	Register destination input is written with the value on the Write data input
ALUScrA	The first ALU operand is the PC	The first ALU operand come from the A register.
MemRead	None	Memory contents at the location specified by the address input is put on the Memory data out.
MemWrite	None	Memory contents at the location specified by the address input are replaced by value on the Write data input.
MemtoReg	The value fed to register Write data input comes from the ALUOut	The value fed to the register Write data input comes from the MDA.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None	The output of memory is written into the IR.
PCWrite	None	The is written;the source is controlled by PCSource.
PCWriteCond	None	The PC is written if the zero output from the ALU is also active.

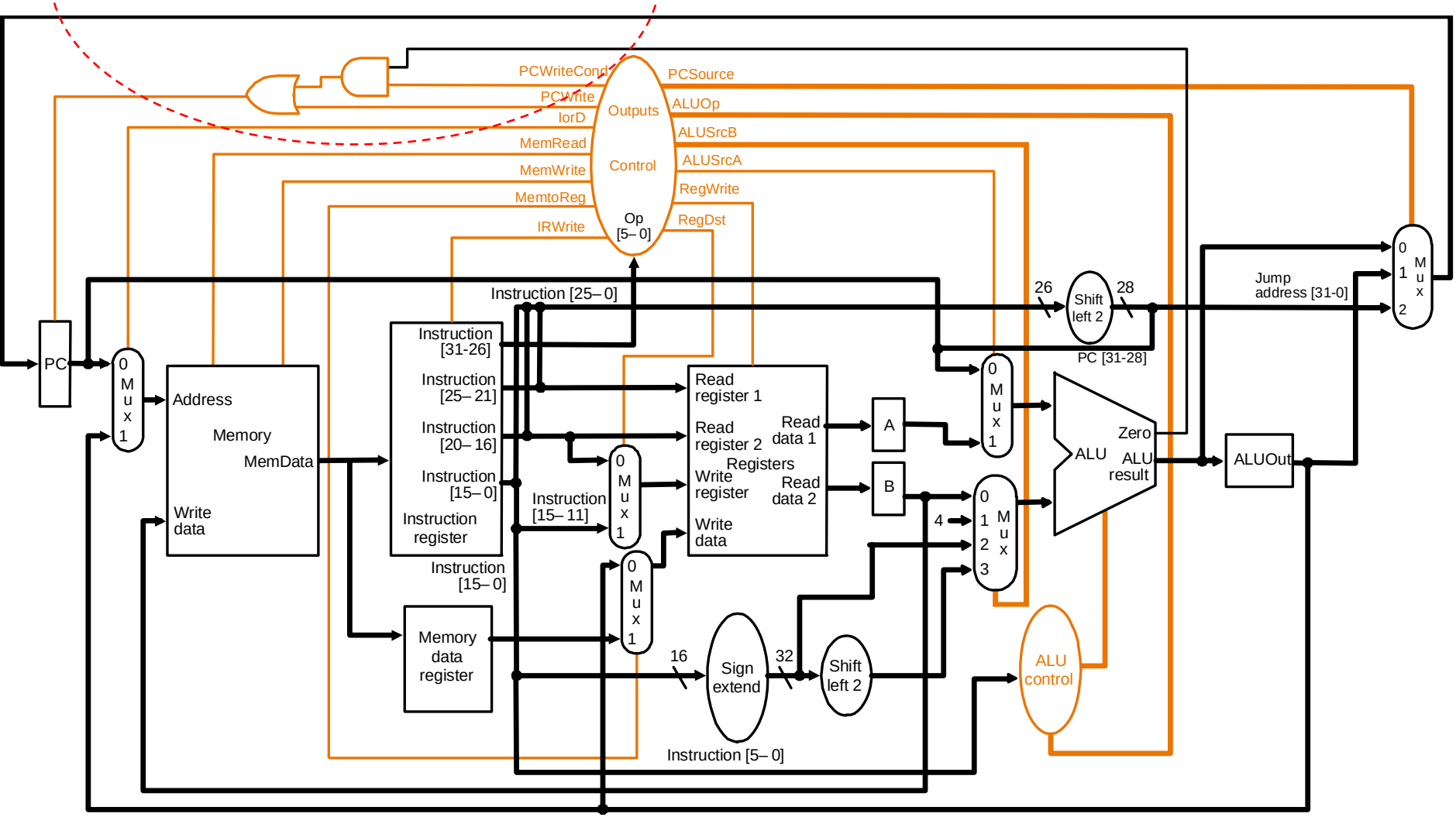
Control signals

Signal name	Value	Effect
ALUOp	00	The ALU performs an add operation.
	01	The ALU performs an subtract operation.
	10	The funct field of the instruction determines the ALUoperation
ALUScrB	00	The second input to the ALU comes from the B register.
	01	The second input to the ALU is the constant 4.
	10	The second input to the ALU is the sign-extended, lower 16 bits of the IR.
	11	The second input to the ALU is the sign-extended, lower 16 bits of the IR shift 2 bits.
PCSource	00	Output of the ALU(PC+4) is sent to the PC for writing.
	01	The contents of ALUOut (the branch target address) are sent to the PC writing.
	10	The jump target address (IR[25:0]shifted left 2 bits and concatenated with PC+4[31:28]) is sent to the PC for writing.

seq: pc = pc + 4

beq: pc = pc + offset * 4

j : $pc = pc_{31-28} + IR_{25-0} << 2$



Implementing the Control

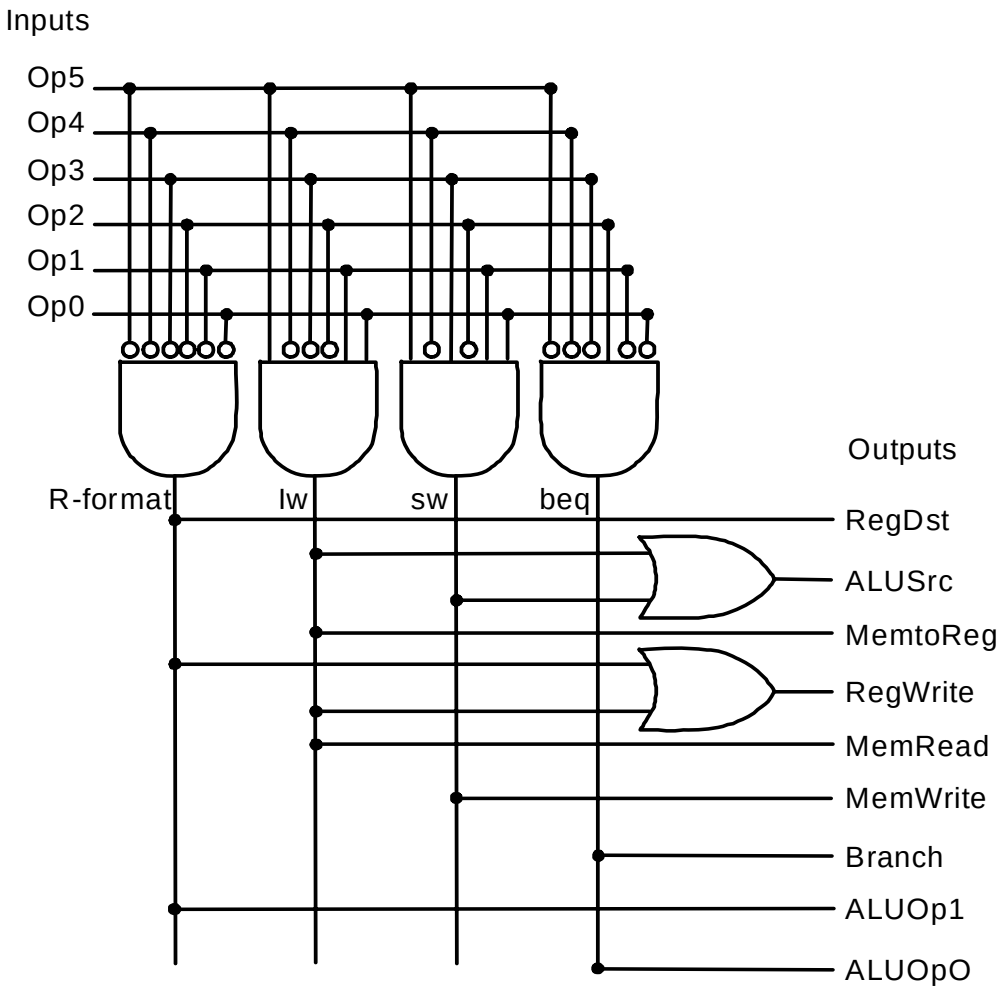
- **Value of control signals is dependent upon:**
  - **what instruction is being executed**
  - **which step is being performed**

- **Let's go over the main control unit in the single-cycle implementation.**

- **Review for singlecycle**

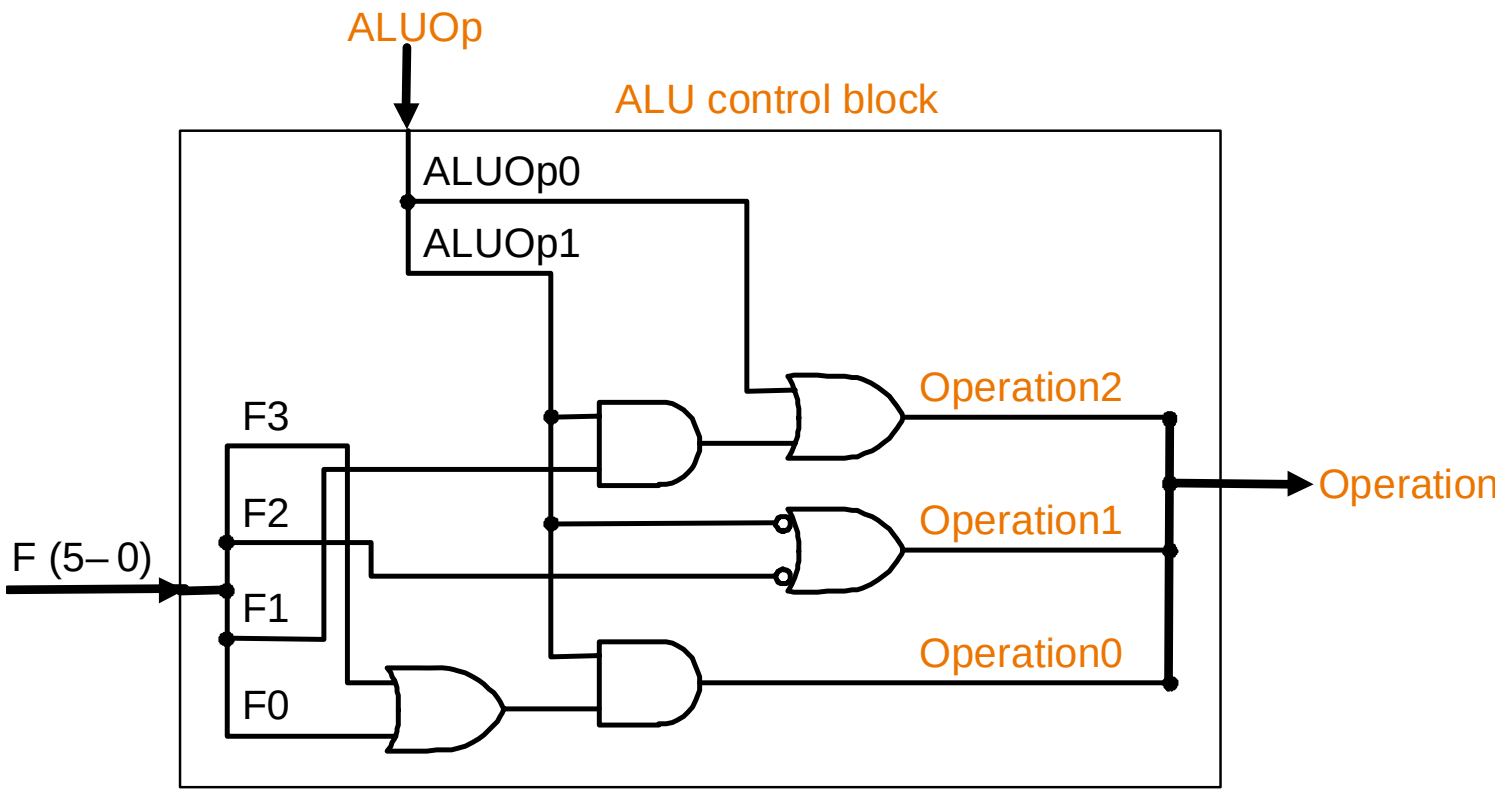
- **Main control unit in the single-cycle implementation**

R-type	**0**
lw	**35**
sw	**43**
beq	**4**



• Review

- **Now,let's look at the AlU control unit.It does not need to changed.**



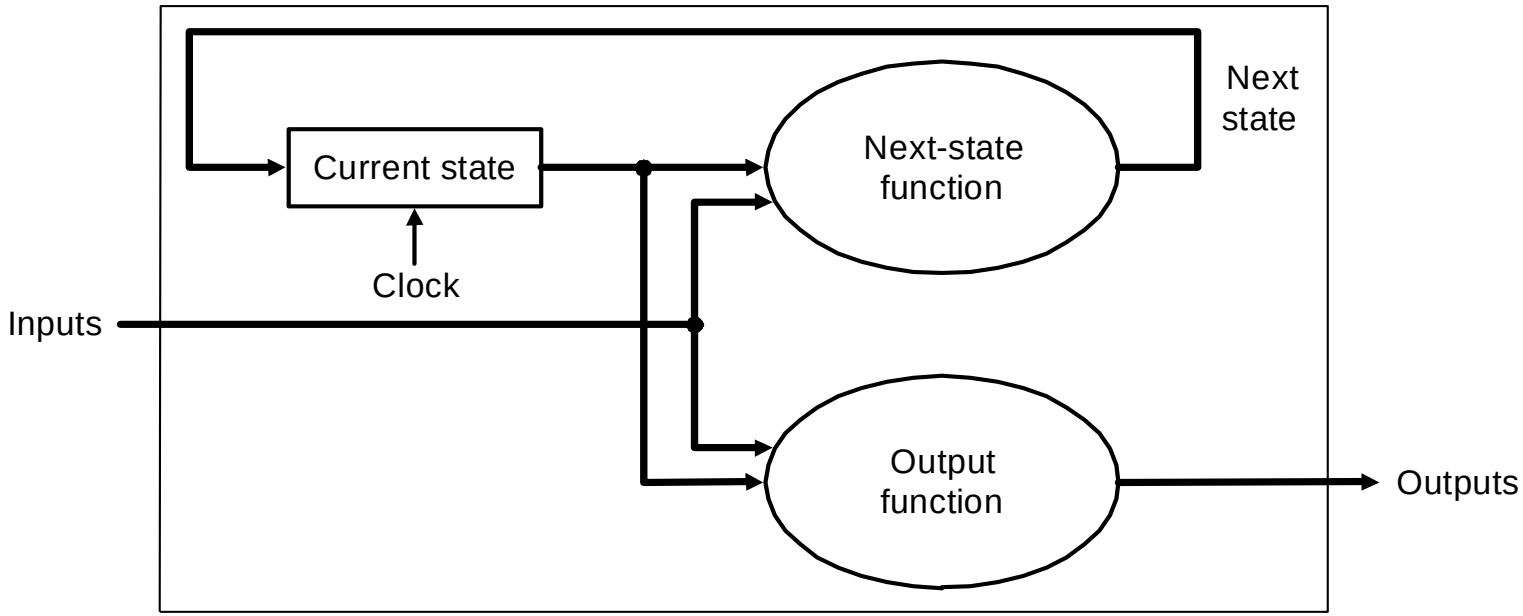
- **As same as the single-cycle**

- **So,the following truth table remains the same.**

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

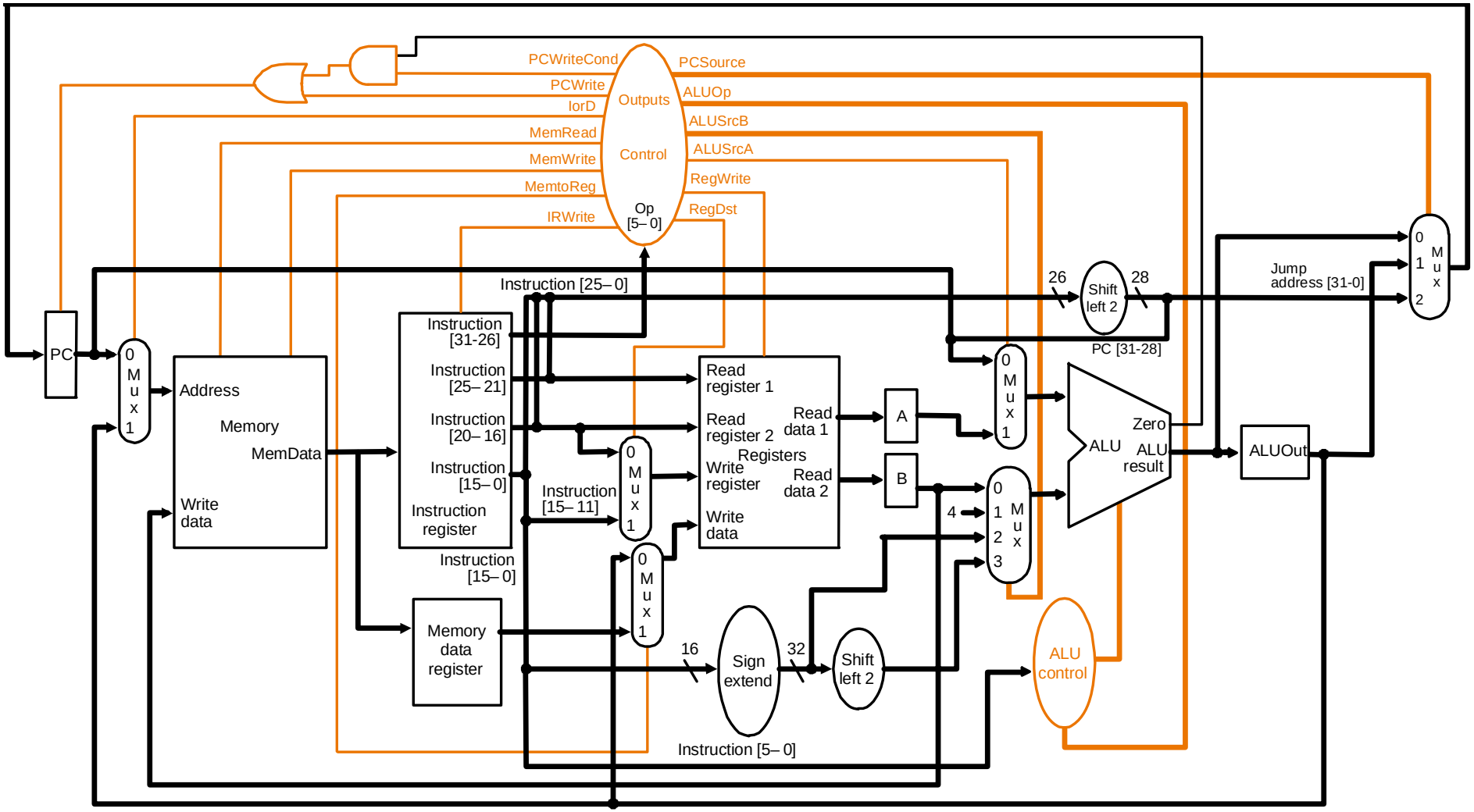
Review: Finite state machines

- **Finite state machines:**
  - **a set of states**
  - **next state function (determined by current state and the input)**
  - **output function (determined by current state and possibly input)**



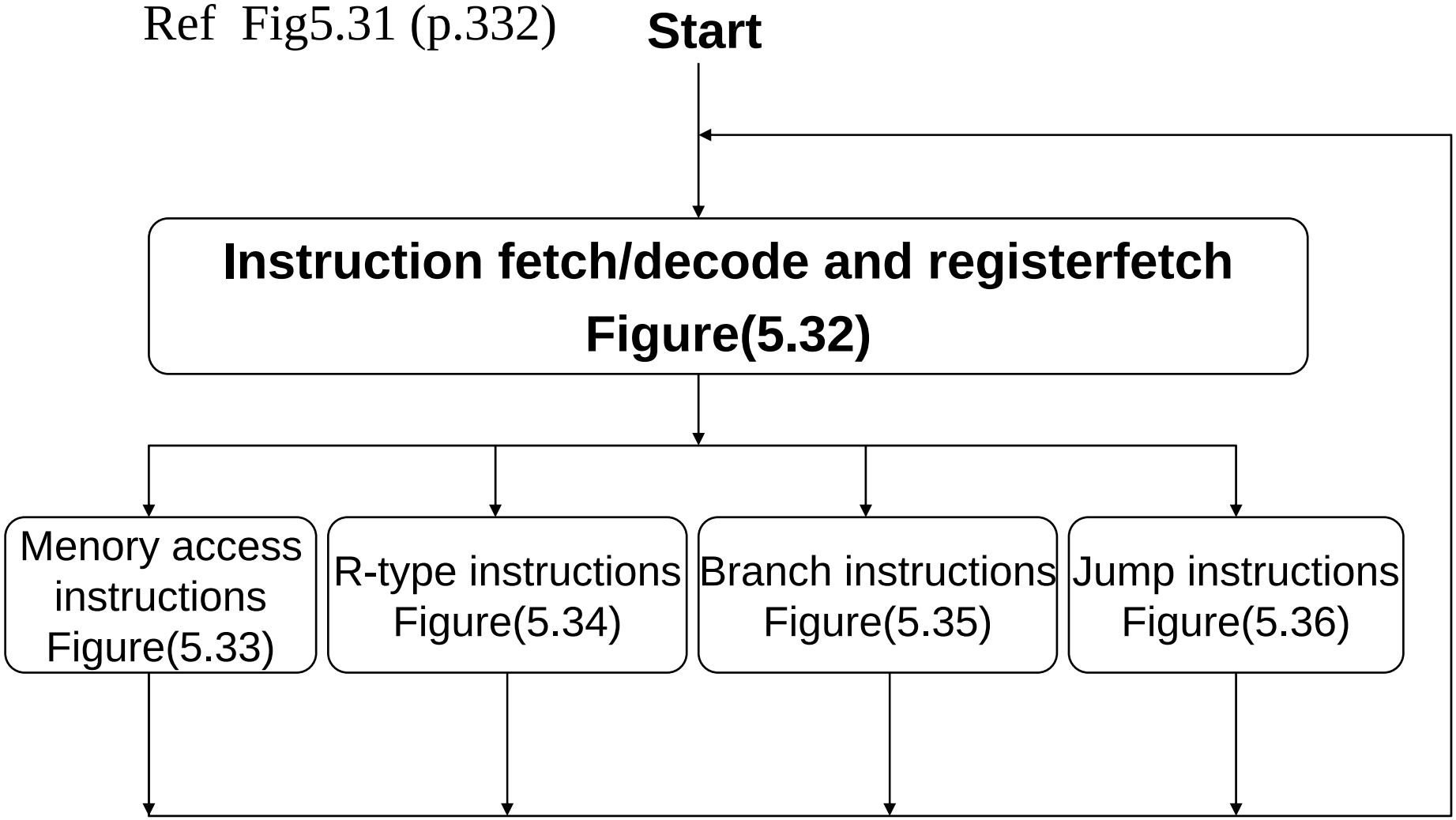
• Difference in controller

- **Do something different in each cycle**
- **Tow main ways to implement the control**
  - **1. Finite state machine**
  - **2. use microprogramming**
- **Next,we will discuss the first way.**

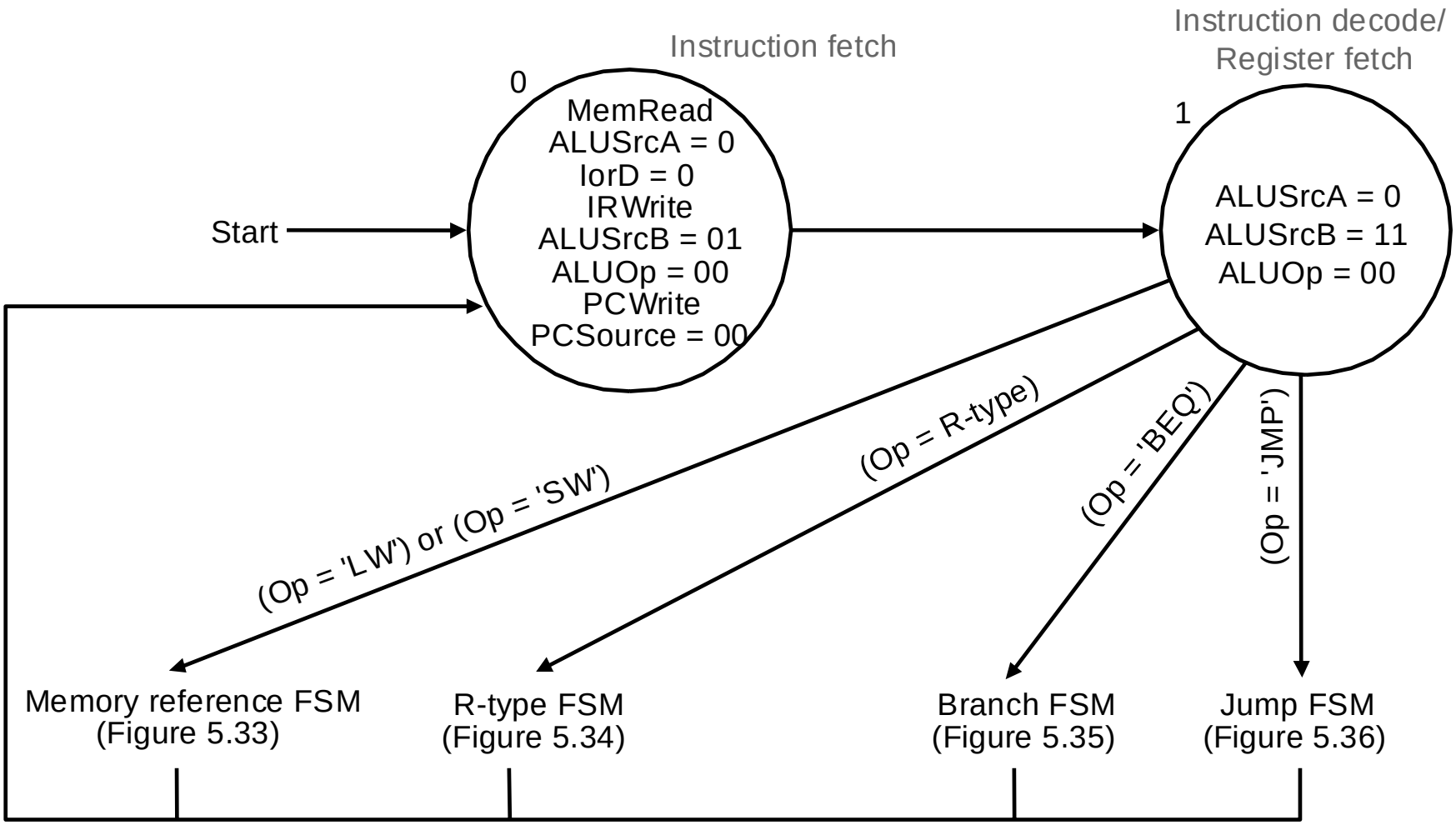

PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5– 0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[31-26]
Instruction
[25– 21]
Instruction
[20– 16]
Instruction
[15– 0]
Instruction
register
Instruction [25– 0]
Instruction
[15– 11]
Instruction
[15– 0]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
16
Sign
extend
32
Shift
left 2
26
Shift
left 2
28
Jump
address [31-0]
PC [31-28]
ALU
Zero
ALU
result
ALUOut
ALU
control
Instruction [5– 0]

State diagram for Instruction execute flow

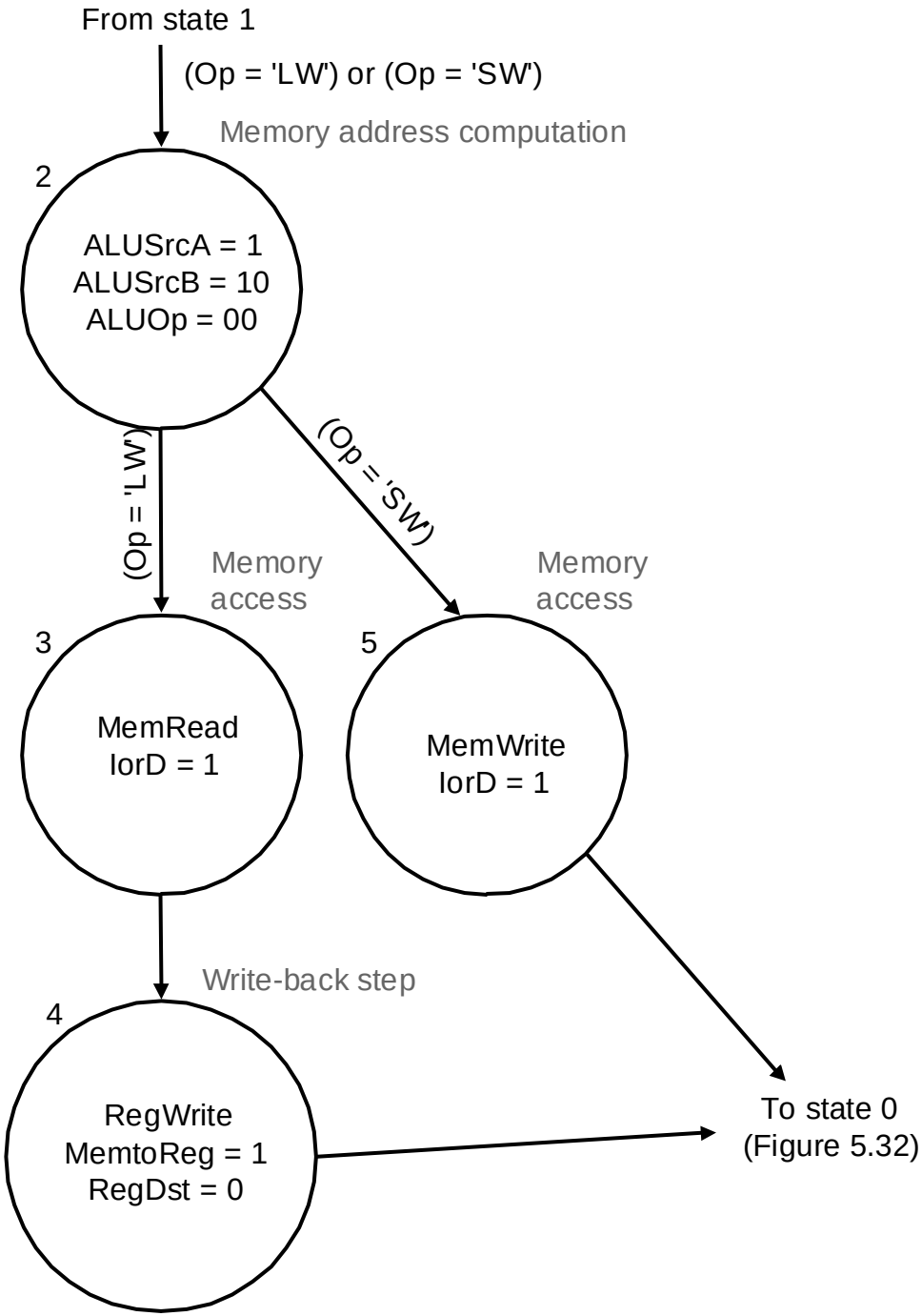
Ref Fig5.31 (p.332)



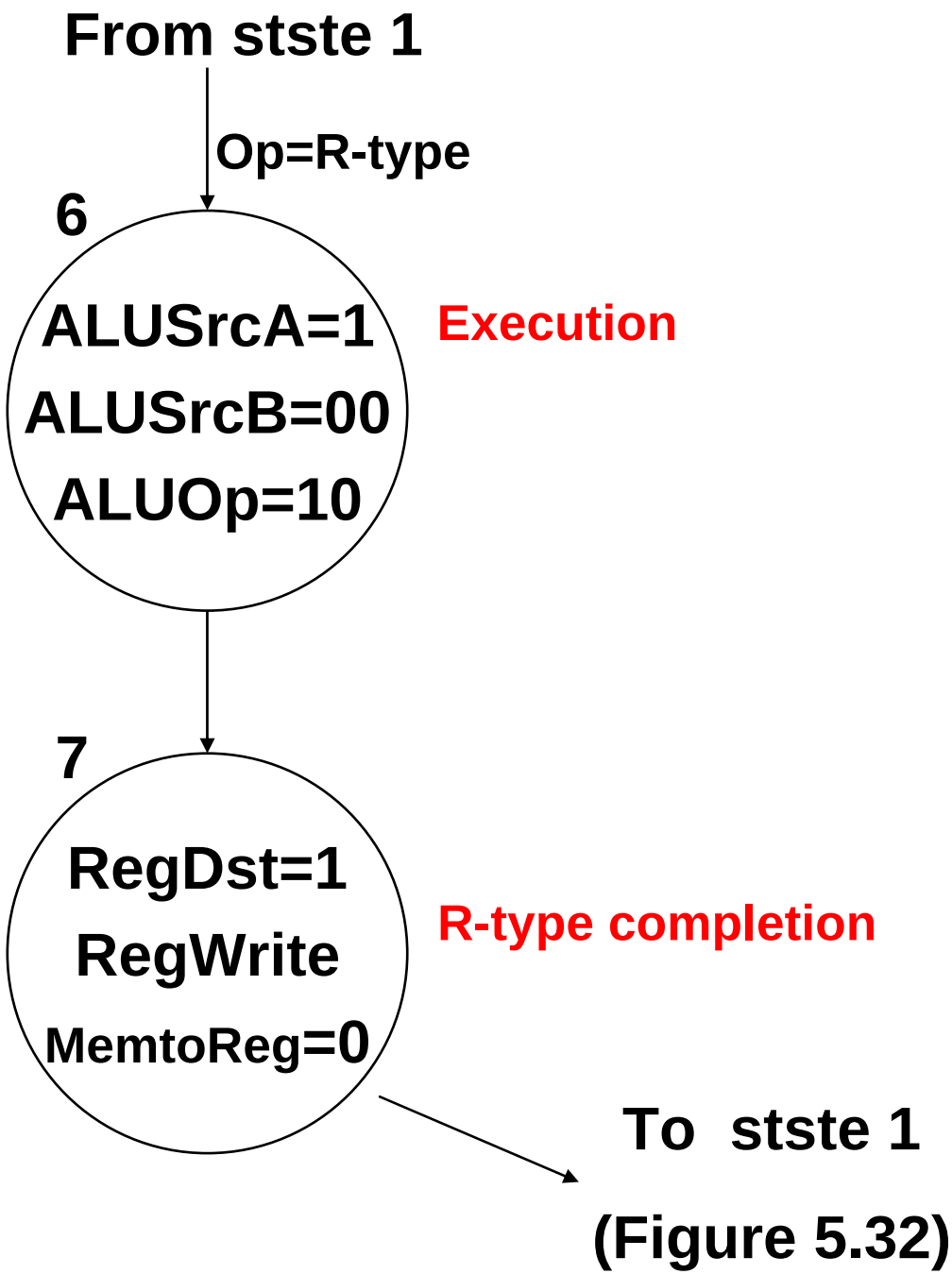
Instruction fetch / decode and Reg fetch



lw and *sw*

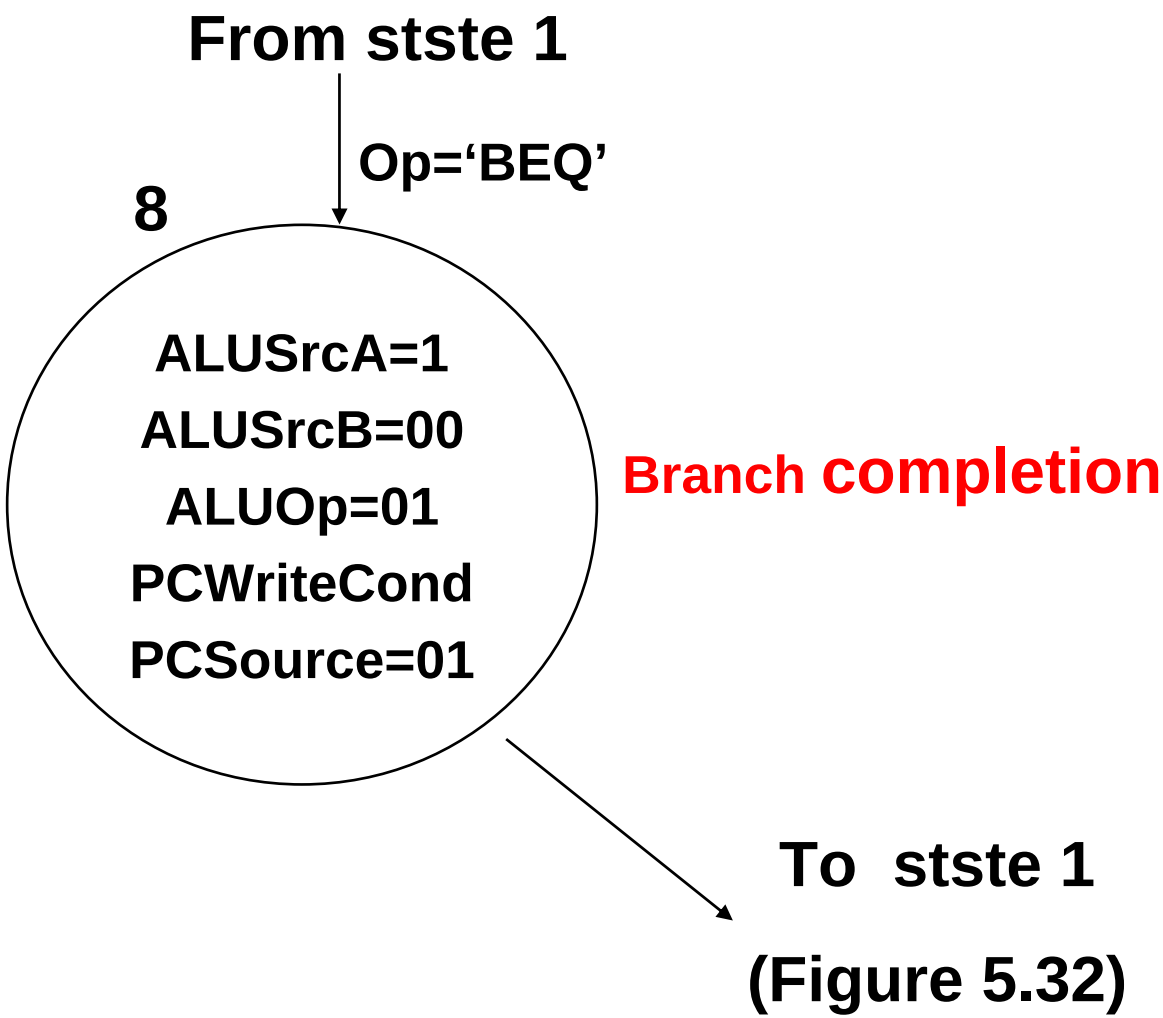

From state 1
(Op = 'LW') or (Op = 'SW')
Memory address computation
2
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00
(Op = 'LW')
(Op = 'SW')
Memory
access
Memory
access
3
MemRead
IorD = 1
5
MemWrite
IorD = 1
Write-back step
4
RegWrite
MemtoReg = 1
RegDst = 0
To state 0
(Figure 5.32)

R-type

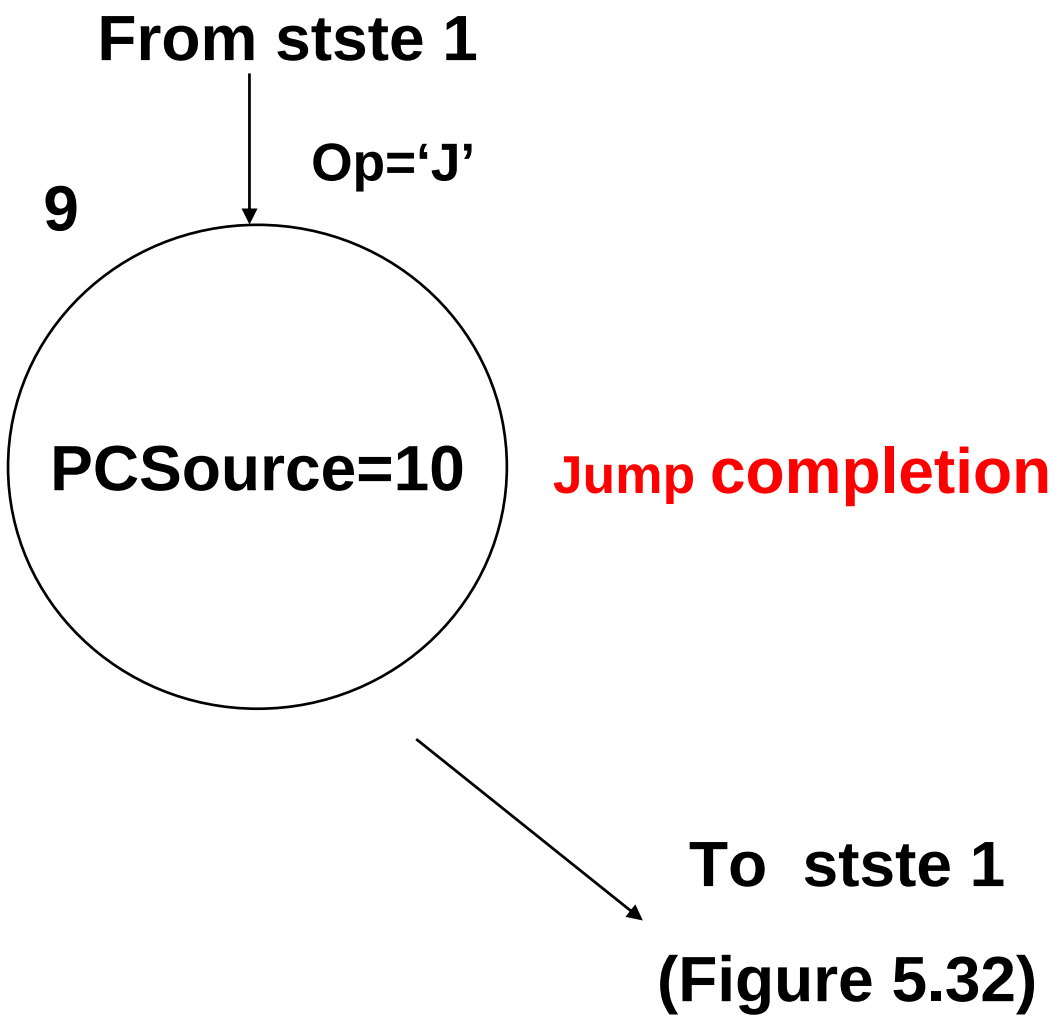


(Figure 5.32)

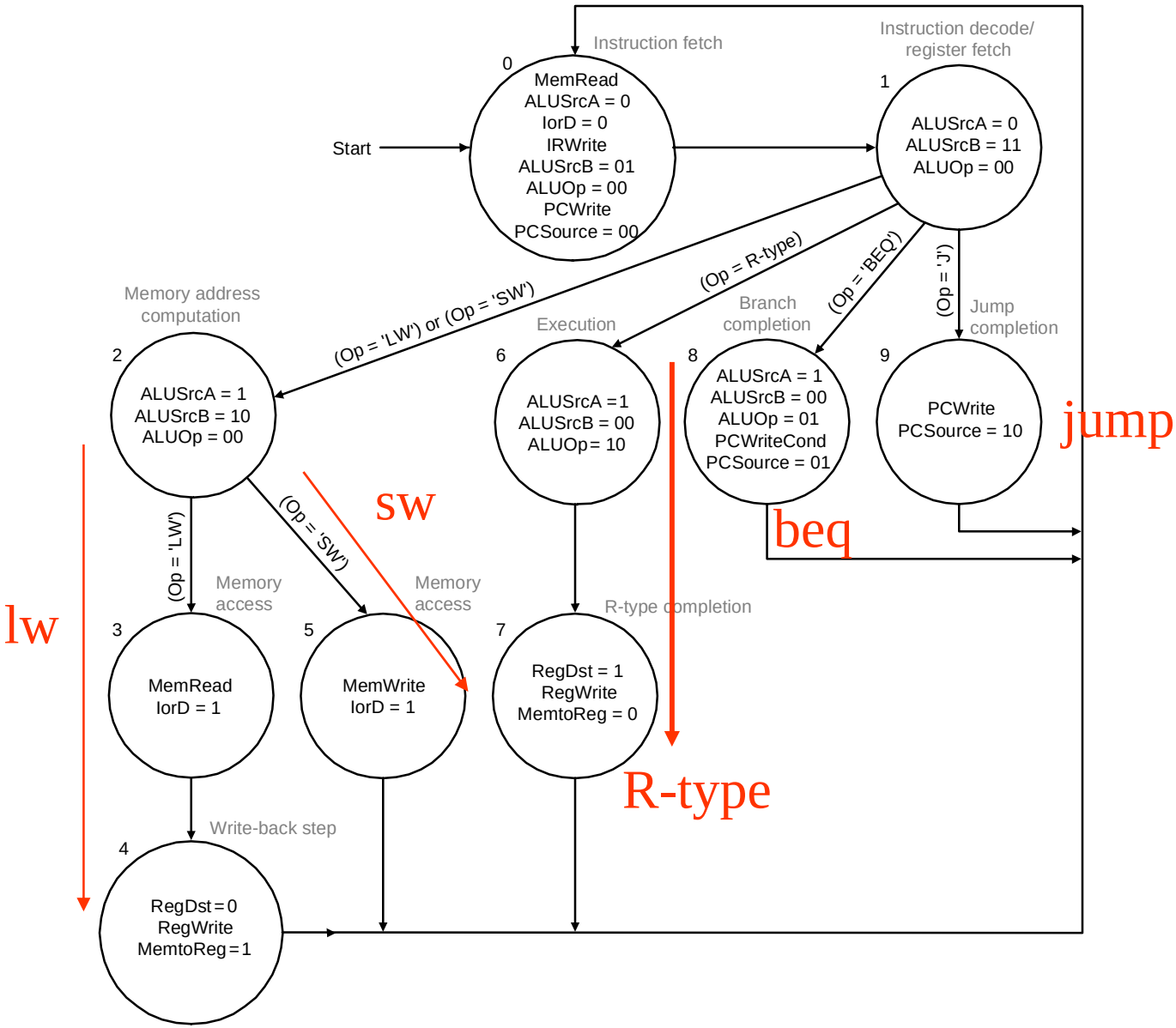
Branch


From stste 1
Op='BEQ'
8
ALUSrcA=1
ALUSrcB=00
ALUOp=01
PCWriteCond
PCSource=01
Branch completion
To stste 1
(Figure 5.32)

J-type


From stste 1
Op='J'
9
PCSource=10
Jump completion
To stste 1
(Figure 5.32)

Graphical Specification of FSM



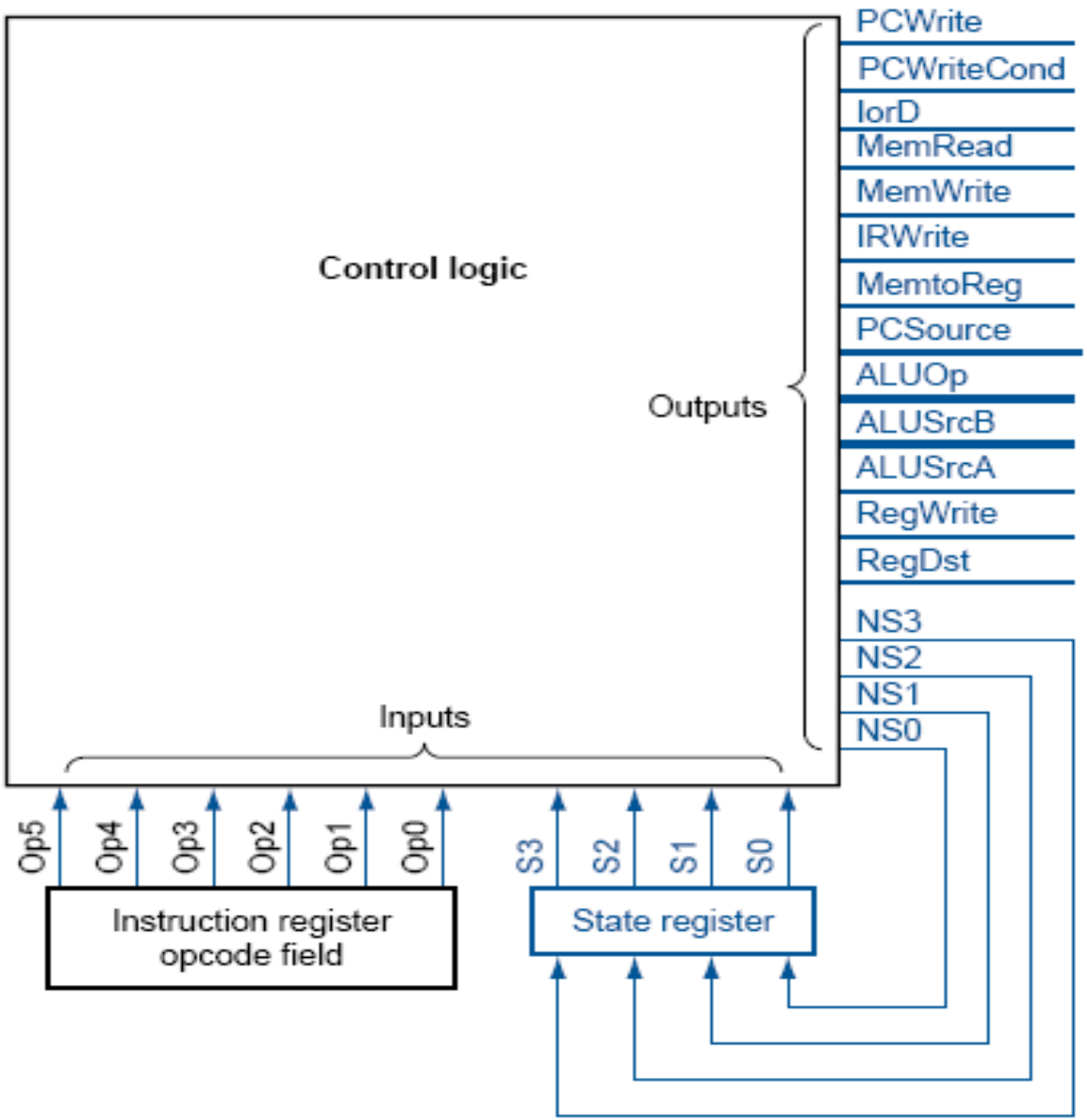
The truth table for the 16 datapath control outputs, which depend only on the state inputs.

Outputs	Input values (S[3–0])									
	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001
PCWrite	1	0	0	0	0	0	0	0	0	1
PCWriteCond	0	0	0	0	0	0	0	0	1	0
IorD	0	0	0	1	0	1	0	0	0	0
MemRead	1	0	0	1	0	0	0	0	0	0
MemWrite	0	0	0	0	0	1	0	0	0	0
IRWrite	1	0	0	0	0	0	0	0	0	0
MemtoReg	0	0	0	0	1	0	0	0	0	0
PCSource1	0	0	0	0	0	0	0	0	0	1
PCSource0	0	0	0	0	0	0	0	0	1	0
ALUOp1	0	0	0	0	0	0	1	0	0	0
ALUOp0	0	0	0	0	0	0	0	0	1	0
ALUSrcB1	0	1	1	0	0	0	0	0	0	0
ALUSrcB0	1	1	0	0	0	0	0	0	0	0
ALUSrcA	0	0	1	0	0	0	1	0	1	0
RegWrite	0	0	0	0	1	0	0	1	0	0
RegDst	0	0	0	0	0	0	0	1	0	0

The logic equations for the control unit shown in a shorthand form

Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

Implemented using a block of combinational logic and a register to hold the the current state



Datapath control output

Next state

Chapter Five

The processor : Datapath and control

5.1 Introduction

5.2 Logic Design Conventions (skip)

5.3 Building a datapath

5.4 A Simple Implementation Scheme

5.5 A Multicycle Implementation

5.5 Microprogramming

5.6 Exception

Microinstruction format

- **Microinstruction**
  - **control signal**
  - **position of next Microinstruction**
- **Representation**
  - **split into some fields**

  - **next position**
    - sequential
    - dispatch table (jump)

Microinstruction format

Field name	Value	Signals active
ALU control	Add	ALUOp = 00
	Subt	ALUOp = 01
	Func code	ALUOp = 10
SRC1	PC	ALUSrcA = 0
	A	ALUSrcA = 1
SRC2	B	ALUSrcB = 00
	4	ALUSrcB = 01
	Extend	ALUSrcB = 10
	Extshft	ALUSrcB = 11
Register control	Read (Reg fetch)	A=Reg[IR_{25-21}], B=Reg[IR_{20-16}]
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1

Microinstruction format

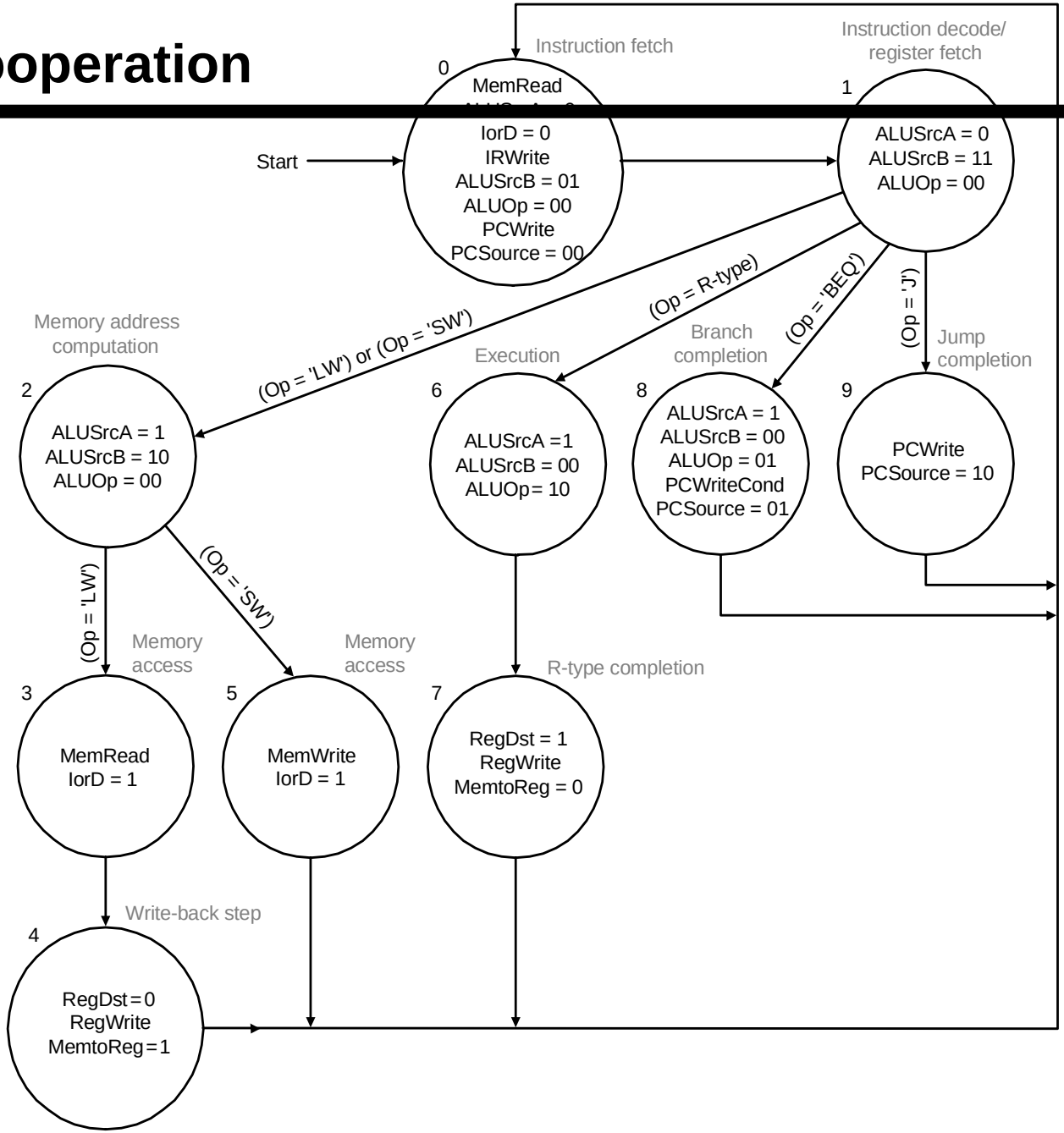
Mem address is from

Memory	Read PC	MemRead, IorD = 0
	Read ALU	MemRead, IorD = 1
	Write ALU	MemWrite, IorD = 1
PC write control	ALU	PCSource = 00 PCWrite
	ALUOut-cond	PCSource = 01, PCWriteCond
	jump address	PCSource = 10, PCWrite
Sequencing	Seq	AddrCtl = 11
	Fetch	AddrCtl = 00
	Dispatch 1	AddrCtl = 01
	Dispatch 2	AddrCtl = 10

Steps of Instructions

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) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
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		

Microoperation



Microprogramming

- **A specification methodology**
  - **appropriate if hundreds of opcodes, modes, cycles, etc.**
  - **signals specified symbolically using microinstructions**

address	Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
0000	Fetch	Add	PC	4		Read PC	ALU	Seq
0001		Add	PC	Extshft	Read			Dispatch 1
0010	Mem1	Add	A	Extend				Dispatch 2
0011	LW2					Read ALU		Seq
0100					Write MDR			Fetch
0101	SW2					Write ALU		Fetch
0110	Rformat1	Func code	A	B				Seq
0111					Write ALU			Fetch
1000	BEQ1	Subt	A	B			ALUOut-cond	Fetch
1001	JUMP1						Jump address	Fetch

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

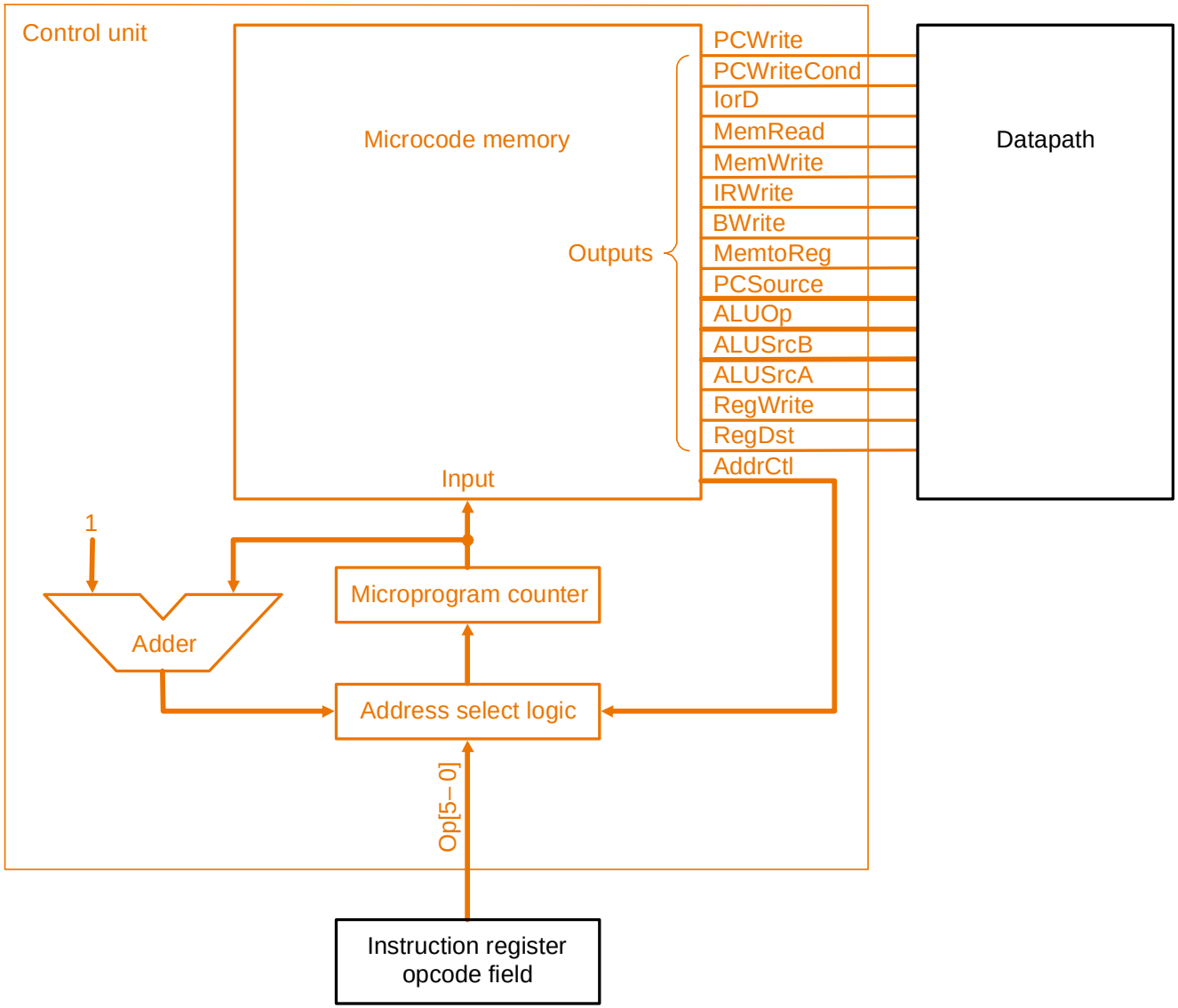
微指令编码

ALU		SRC1	SRC2		Register control				Memory control			PCWrite				Seq	
0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1

ALUOp1
ALUOp0
ALUSrcA
ALUSrcB1
ALUSrcB0
IRWrite
RegWrite
RegDst
MemtoReg
MemRead
MemRead
IorD
PCSource1
PCSource0
PCWrite
PCWriteCond
Addrctl1
Addrctl0

Microprogramming

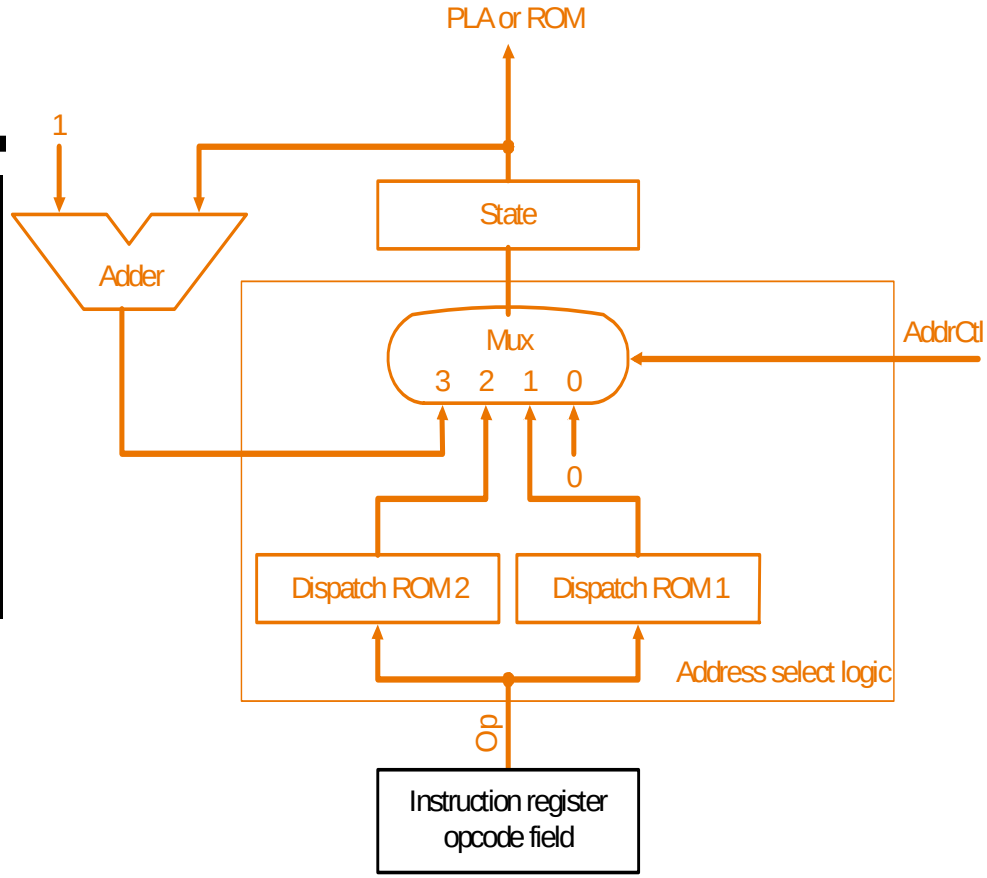
- **What are the Microinstructions?**



Details

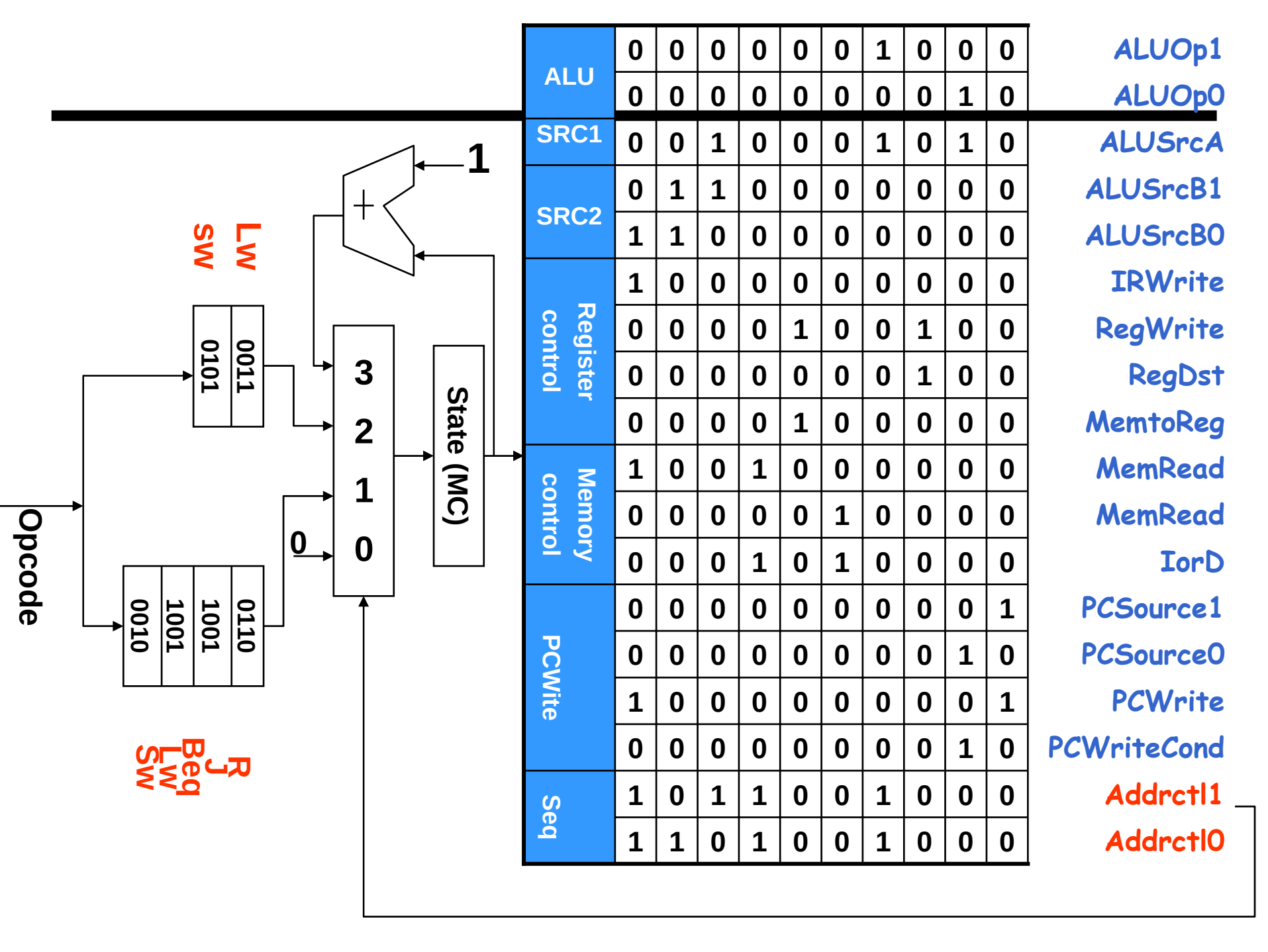
Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	0011
101011	sw	0101



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

Seeing to CD


Opcode
0010
1001
1001
0110
0101
0011
Lw
Sw
R
J
Beq
Lw
Sw
0
1
3
2
1
0
State (MC)
ALU
SRC1
SRC2
Register control
Memory control
PCWite
Seq
ALUOp1
ALUOp0
ALUSrcA
ALUSrcB1
ALUSrcB0
IRWrite
RegWrite
RegDst
MemtoReg
MemRead
MemRead
IorD
PCSource1
PCSource0
PCWrite
PCWriteCond
Addrctl1
Addrctl0

Chapter Five

The processor : Datapath and control

5.1 Introduction

5.2 Logic Design Conventions (skip)

5.3 Building a datapath

5.4 A Simple Implementation Scheme

5.5 A Multicycle Implementation

5.5 Microprogramming

5.6 Exception

5.6 Exception

- **The cause of changing CPU's work flow :**
  - **Control instructions in program (bne/beq, j, jal , etc)**

    It is foreseeable in programming flow
  - **Something happen suddenly (Exception and Interruption)**

    It is unpredictable
- **Unexpected events**
  - **Exception: from within processor (overflow, undefined instruction, etc)**
  - **Interruption : from outside processor (input /output)**

5.6 Exception

- **Exception**

  An Exception is a unexpected event from within processor.

- **We follow the MIPS convention, using the term exception to refer to any unexpected change in control flow.**
- **Here we will discuss two types exceptions :**
  - **arithmetic overflow**
  - **undefined instruction**

How Exceptions Are Handled

- When exception happens, the processor must do something.
- The predefined process routines are saved in memory when computer starts.
- Problem: how can CPU goto relative routine when an exception occurs.
- CPU should know
  - the cause of exception
  - which instruction generate the exception

How Exceptions Are Handled

- **Design**
  - **add a register: exception program counter(EPC)**
    save the address of the offending instruction
  - **add a status register: cause register(CauseReg)**
    hold a field that indicates the reason for the exception.

$$\text{bit0} = \begin{cases} 0 & \text{undefined instruction} \\ 1 & \text{overflow} \end{cases}$$

  - ***Another method is to use vector interrupts***

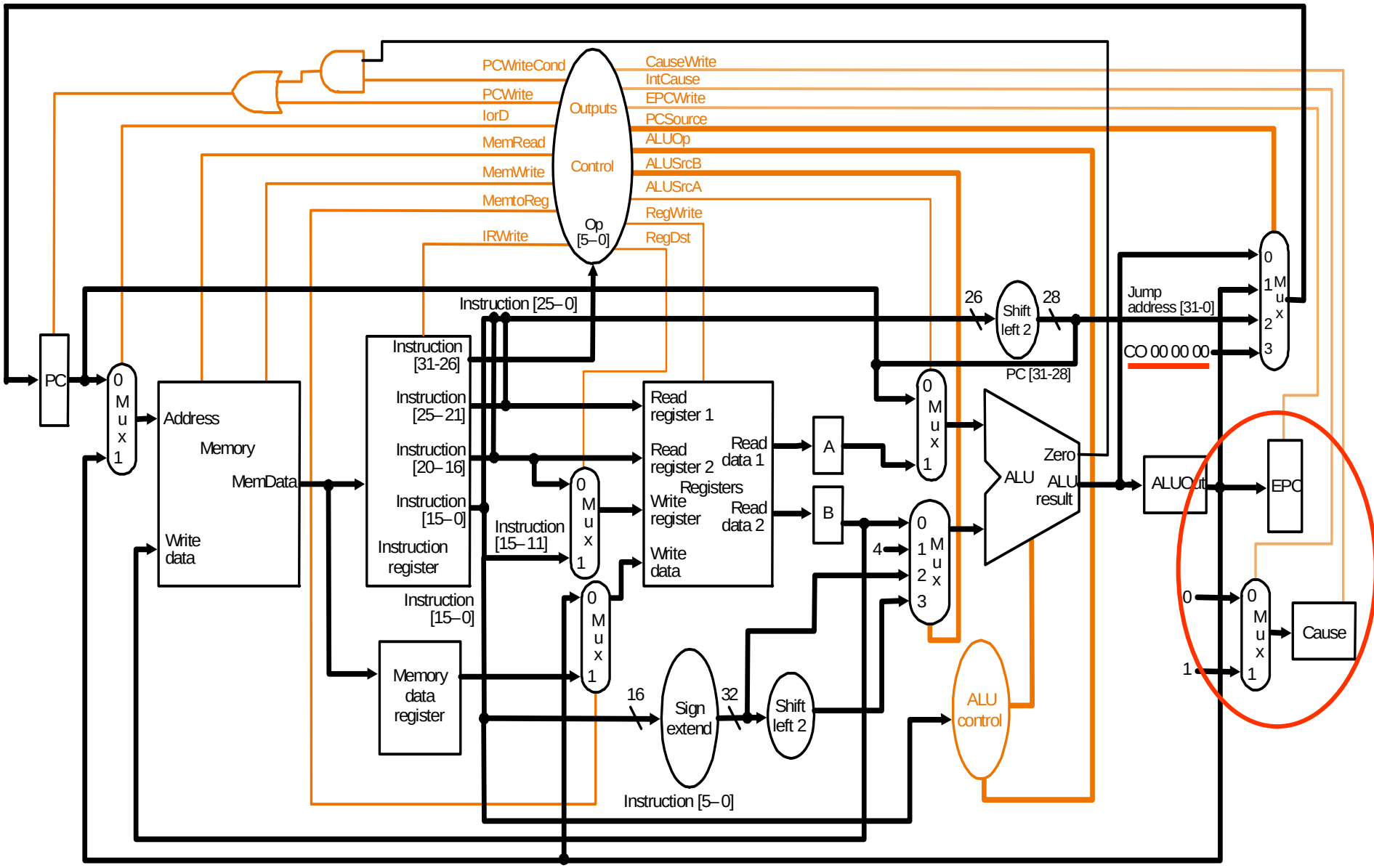
Exception type	vector address
undefine instr	$c0\ 00\ 00\ 00_H$
overflow	$c0\ 00\ 00\ 20_H$

How Control Checks for Exceptions

- add control signal
  - CauseWrite for CauseReg
  - EPCWrite for EPC

    EPC = PC - 4 (completed by ALU)
- process of control
  - CauseReg = 0 or 1
  - EPC = PC - 4
  - PC <--- address of process routine (ex. c0000000)

How Control Checks for Exceptions

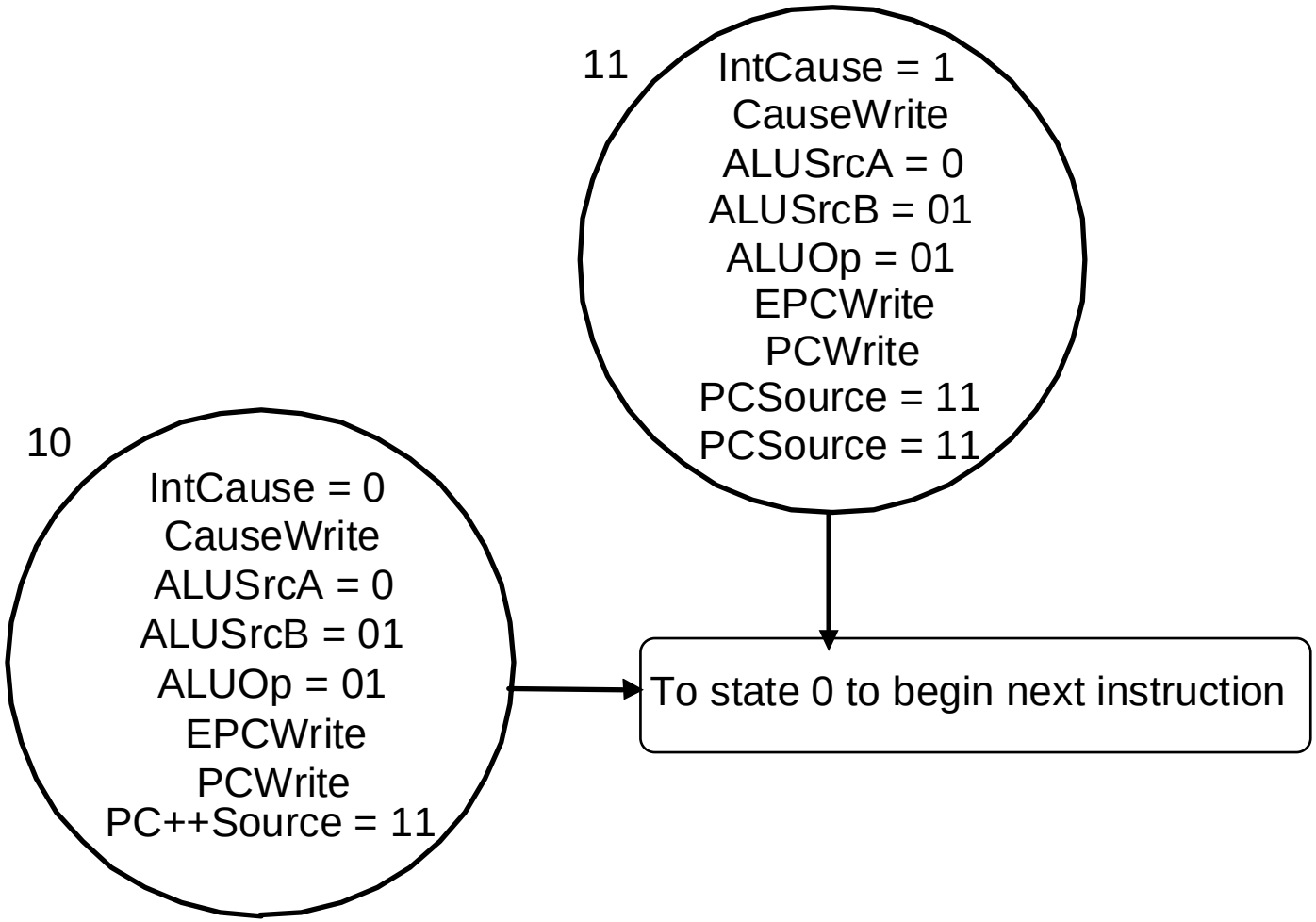


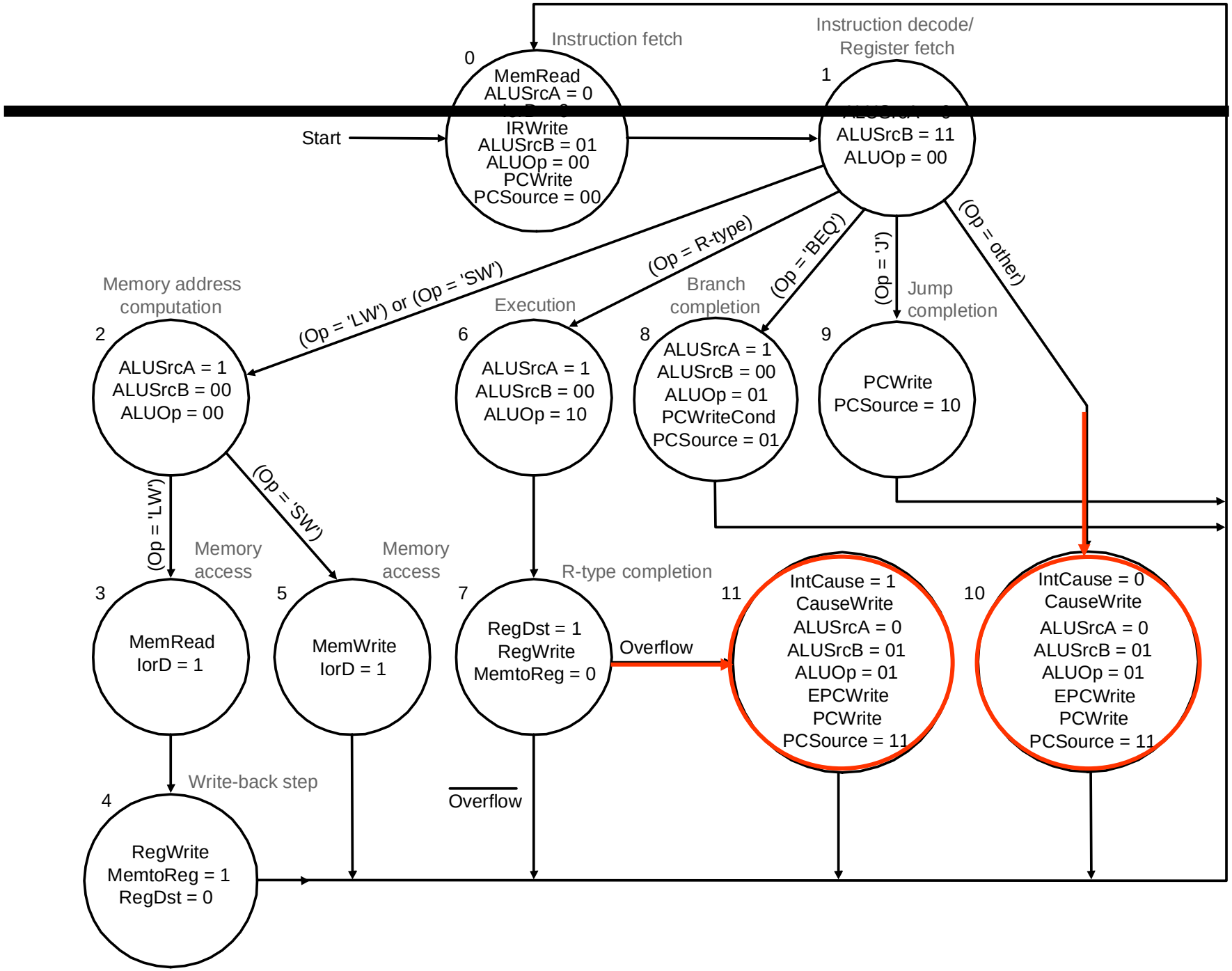
How Control Checks for Exceptions

- **detect exceptions**
  - **Undefined instruction**

    when no next state is defined from state 1 for op value. New state 10 is introduced.
  - **Overflow**

    Overflow is occured only in R-type instruction. Overflow is provided as an output from the ALU. This signal is used in the modified FSM to specify an additional state 11 for state 7 .


11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PC++Source = 11
To state 0 to begin next instruction


Instruction fetch
0
MemRead
ALUSrcA = 0
IRWrite
ALUSrcB = 01
ALUOp = 00
PCWrite
PCSource = 00
Start
Instruction decode/
Register fetch
1
ALUSrcB = 11
ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
(Op = other)
Memory address
computation
2
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 00
Execution
6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10
Branch
completion
8
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 01
PCWriteCond
PCSource = 01
Jump
completion
9
PCWrite
PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory
access
3
MemRead
IorD = 1
Memory
access
5
MemWrite
IorD = 1
R-type completion
7
RegDst = 1
RegWrite
MemtoReg = 0
Overflow
11
IntCause = 1
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
10
IntCause = 0
CauseWrite
ALUSrcA = 0
ALUSrcB = 01
ALUOp = 01
EPCWrite
PCWrite
PCSource = 11
Write-back step
4
RegWrite
MemtoReg = 1
RegDst = 0
Overflow