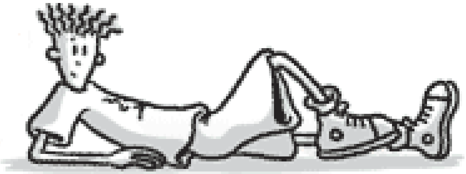
Computer Organization & Design

Hardware/Software interface

楼学庆

Lou Xueqing



玉泉校区曹光彪东楼507室

Website:10.214.47.99/index.php
Email: xqlou@zju.edu.cn
hzlou@163.com

先生印

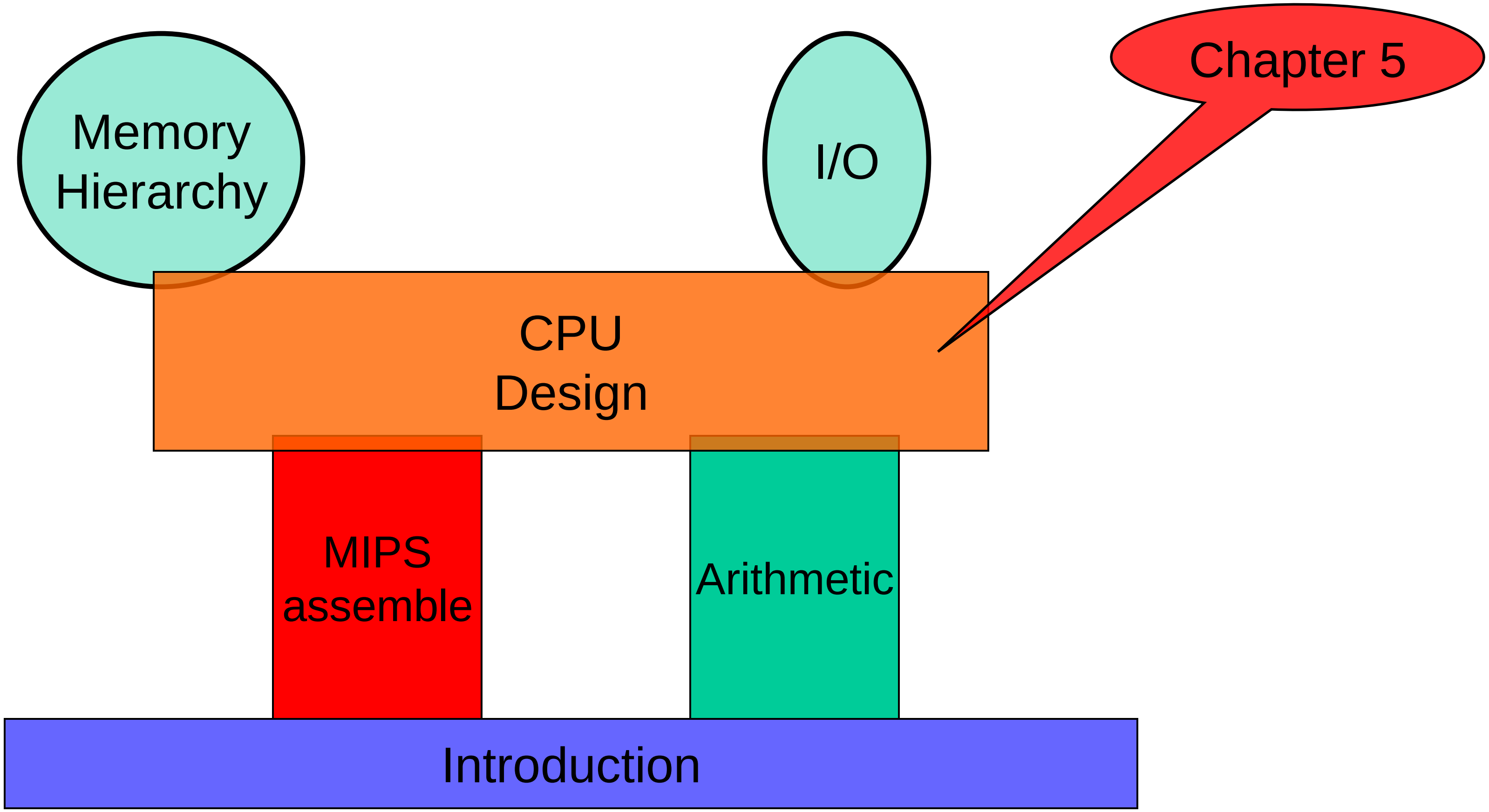
Unit 5:

Datapath & Control

Chapter 5

Chapter 2

- Topics: Arithmetic for Computer



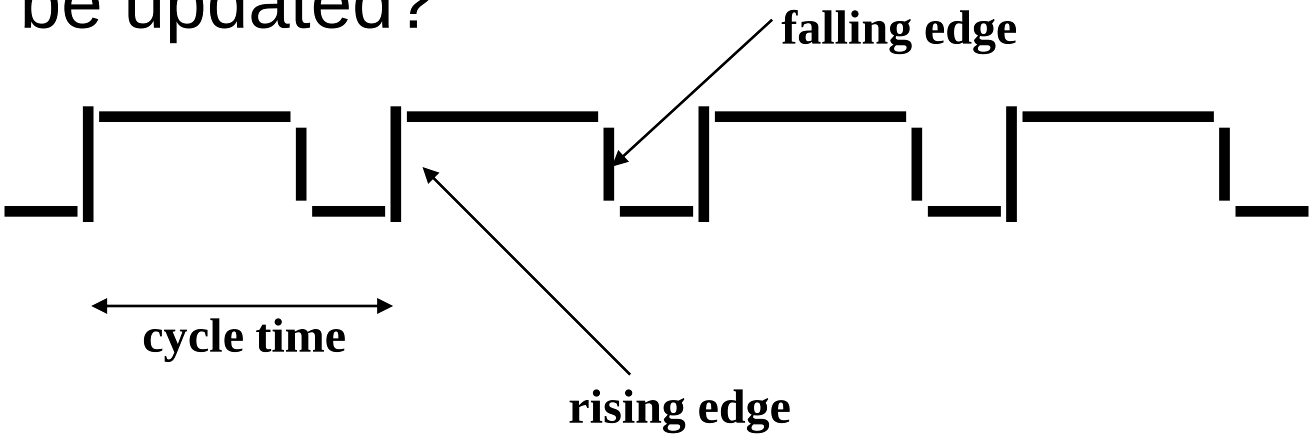
CPU ------设计

The Processor: Datapath & Control

- We're ready to 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`
- Generic Implementation:
  - ☞ use the program counter (PC) to supply instruction address
  - ☞ get the instruction from memory
  - ☞ read registers
  - ☞ use the instruction to decide exactly what to do

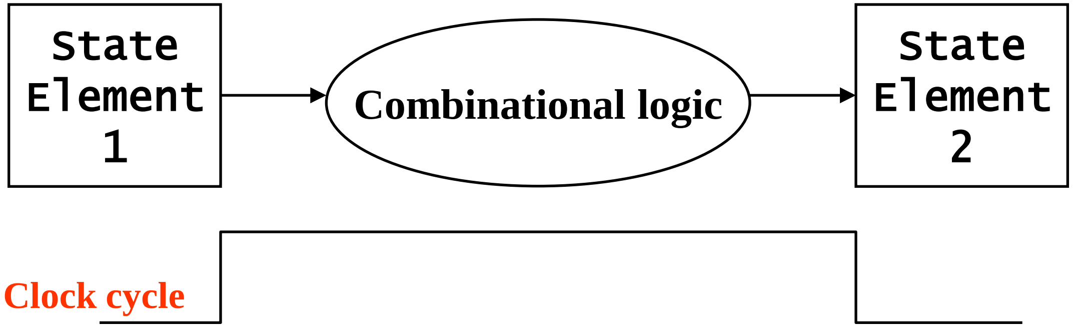
State Elements

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



Our Implementation

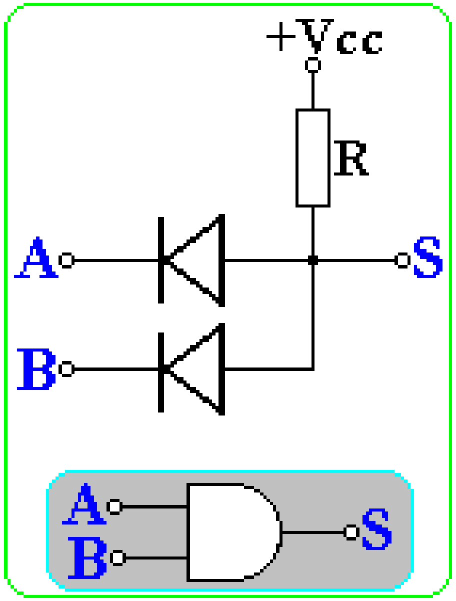
- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements

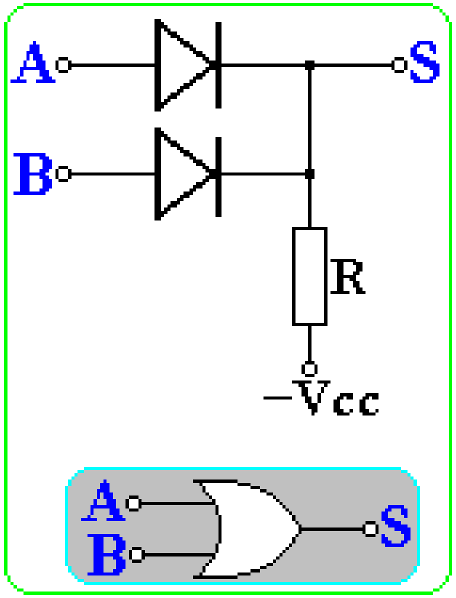


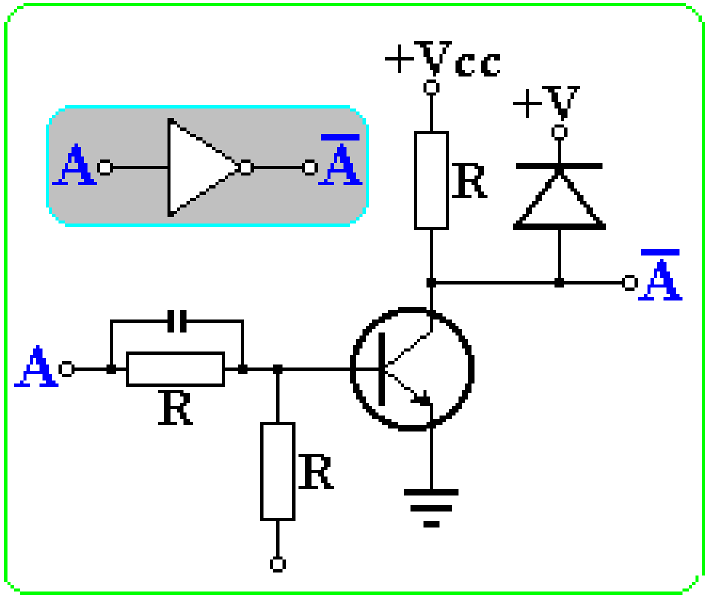
Basic elements

- Basic logic gate。
  - AND： S=A•B
  - OR ： S=A+B
  - NOT： S=~A

A	B	A•B	A+B	~A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0



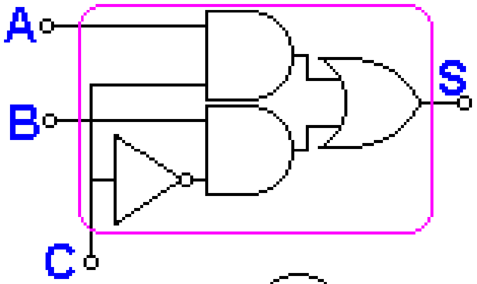


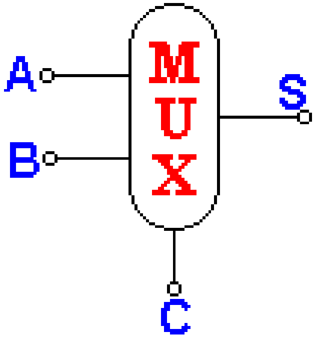


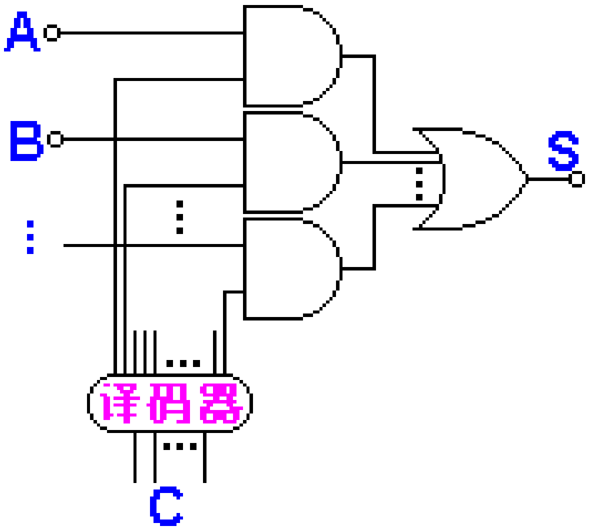
MUX

- 多路开关：有若干个输入，由控制端决定那一路输入将输出。
  - ☞ 经常是多路并行。

C	S
0	A
1	B







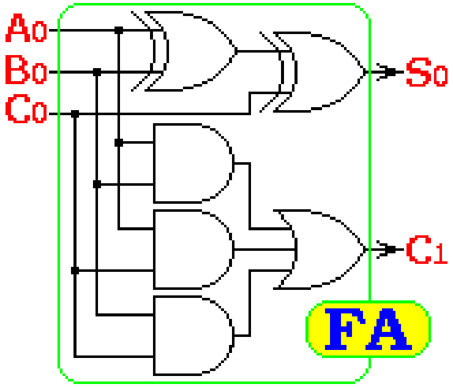
Adder

- Half adder：

A	0	0	1	1
B	0	1	0	1
S	0	1	1	0

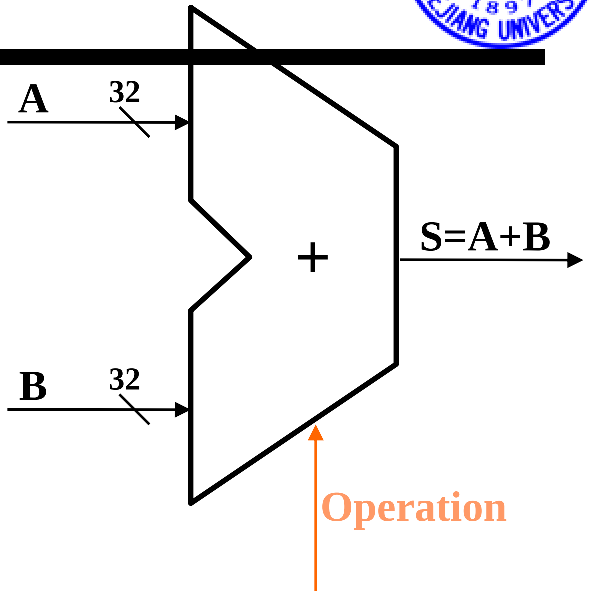
- Full adder： S=A+B
  - ☞ Input: A, B, C0. Output: Sum, CarryOut
  - ☞ S = A ⊕ B ⊕ C
  - ☞ C_{+1}= A•B + B•C + C•A

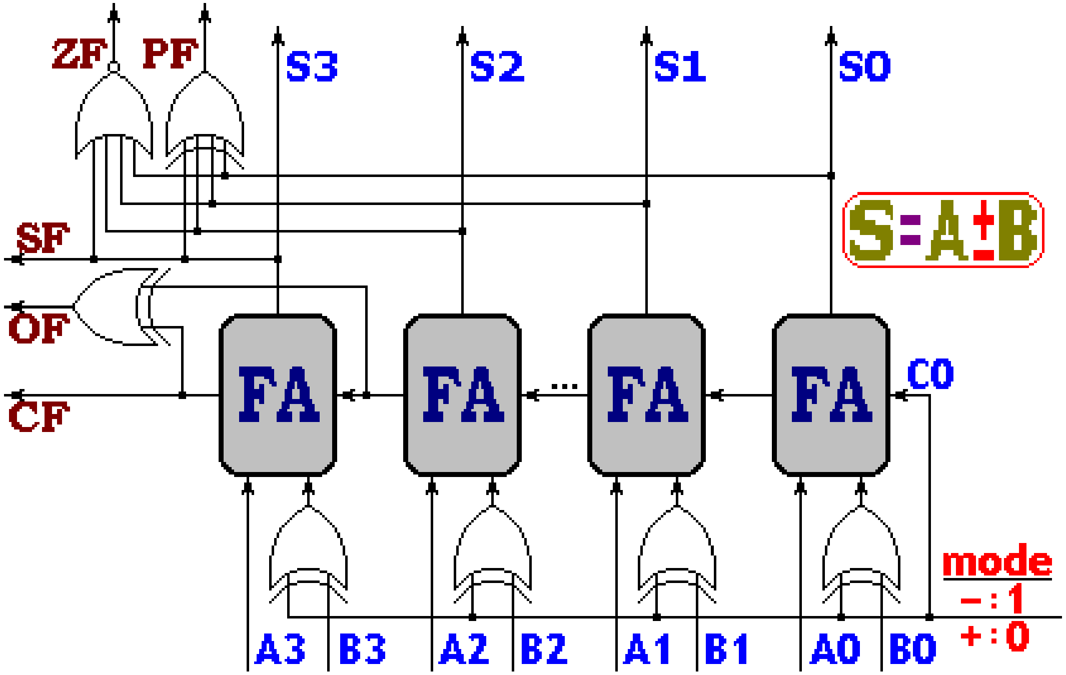
A	0	0	1	1	0	0	1	1
B	0	1	0	1	0	1	0	1
C	0	0	0	0	1	1	1	1
S	0	1	1	0	1	0	0	1
C_{+1}	0	0	0	1	0	1	1	1



串行进位加法器

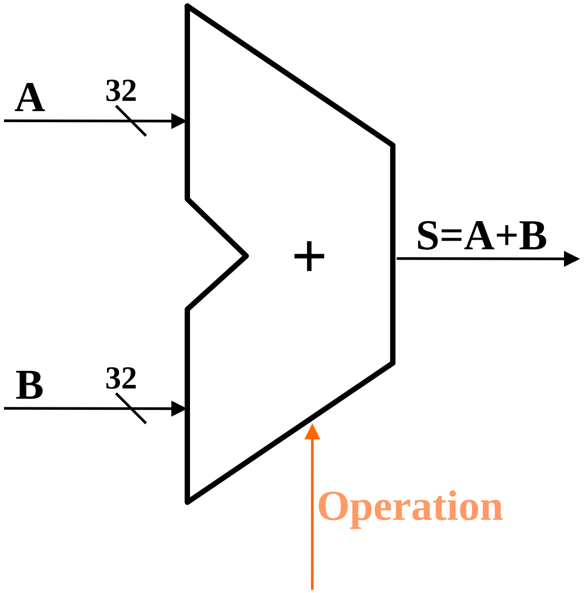
- 4位串行进位加法器：S=A+B
  - ☞ 各位串行进行。
  - ☞ 同时根据结果产生各种标记状态。





Adder

- 32-bit Adder



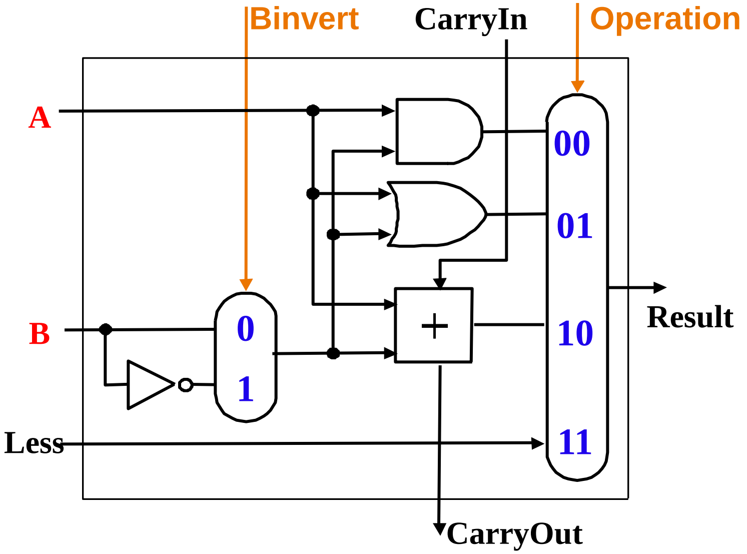
ALU

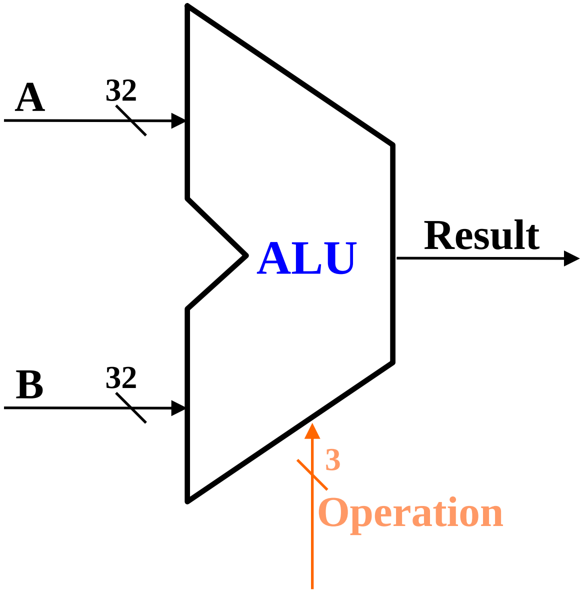
- 算术逻辑运算器ALU：即运算器
  - ☞ 5 Operations
  - ☞ “Set on less than”：

    if A<B then Result=1；

    else Result=0。

Operation	Function
000	And
001	Or
010	Add
110	Sub
111	Slt

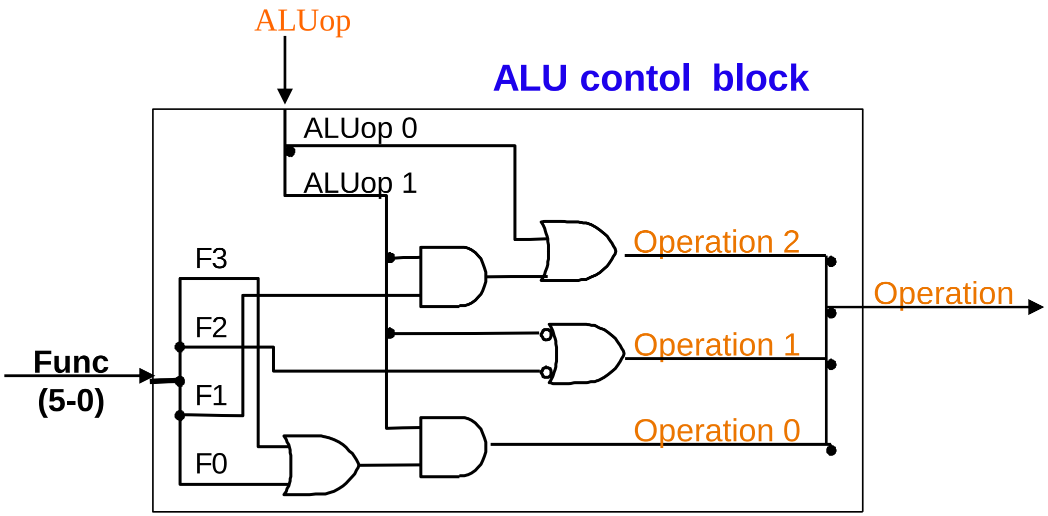


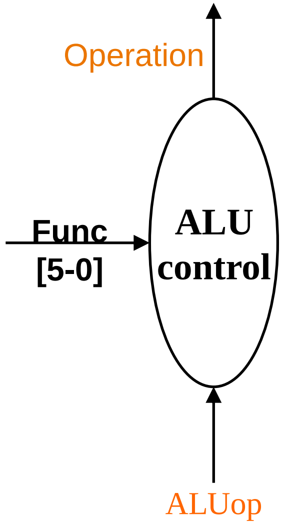


ALU Control

- ALU Control：
  - ALU由ALUcontrol控制：ALUcontrol由ALUop和指令的低6位(5-0)联合产生ALU控制码。
  - 这样的好处在于分级控制，对于用的最多的加、减操作，系统只需给出两位的ALUop即可。

ALUop	Operation	Function
10	000	And
10	001	Or
00	010	Add
01	110	Sub
10	111	Slt

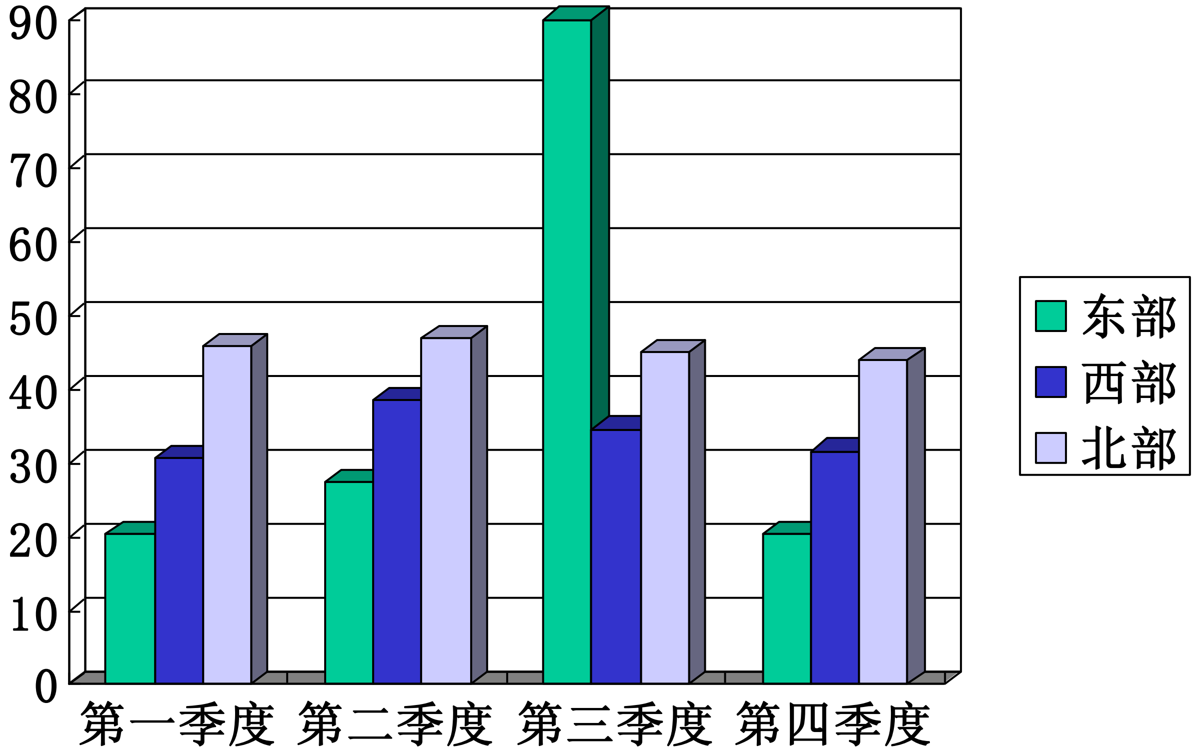




ALU Control

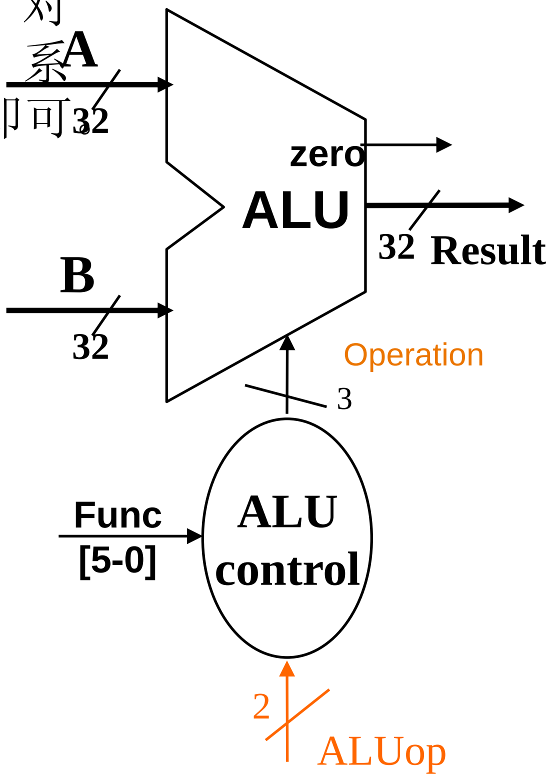
- ALUop与Func联合对ALU的控制

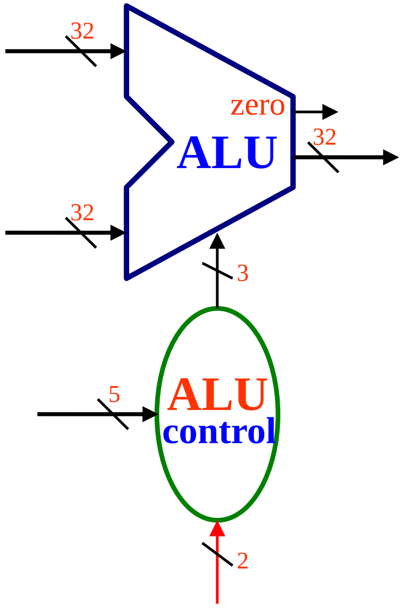
ALUOp		Funct field						Operation	ALU
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		Function
0	0	X	X	X	X	X	X	010	Add
1	1	X	X	X	X	X	X	110	Sub
1	X	X	X	0	0	0	0	010	Add
1	X	X	X	0	0	1	0	110	Sub
1	X	X	X	0	1	0	0	000	And
1	X	X	X	0	1	0	1	001	Or
1	X	X	X	1	0	1	0	111	SetonLT


90
80
70
60
50
40
30
20
10
0
第一季度
第二季度
第三季度
第四季度
🟩 东部
🟦 西部
⬜ 北部

ALU Control

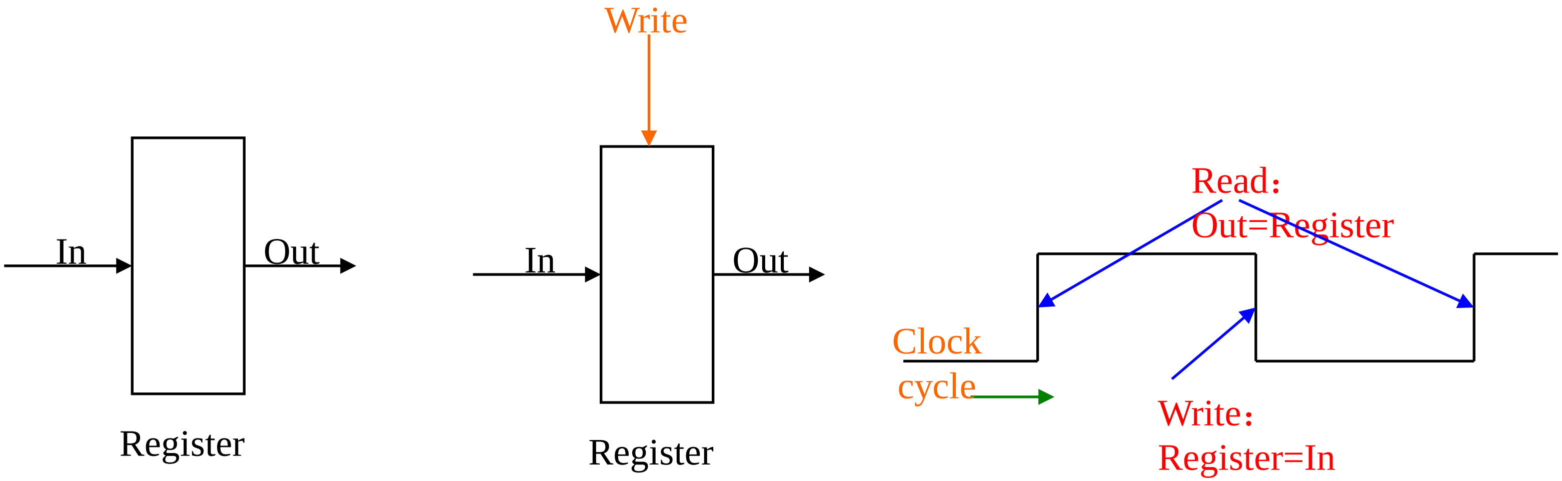
- ALU控制：
  - ALU由ALUcontrol控制：ALUcontrol由ALUop和指令的低6位(5-0)联合产生ALU控制码。
  - 这样的好处在于分级控制，对于用的最多的加、减操作，系统只需给出两位的ALUop即可





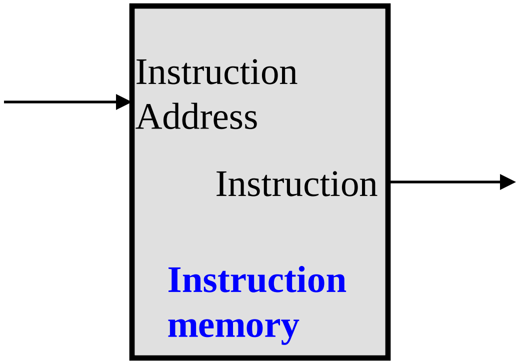
REGISTER

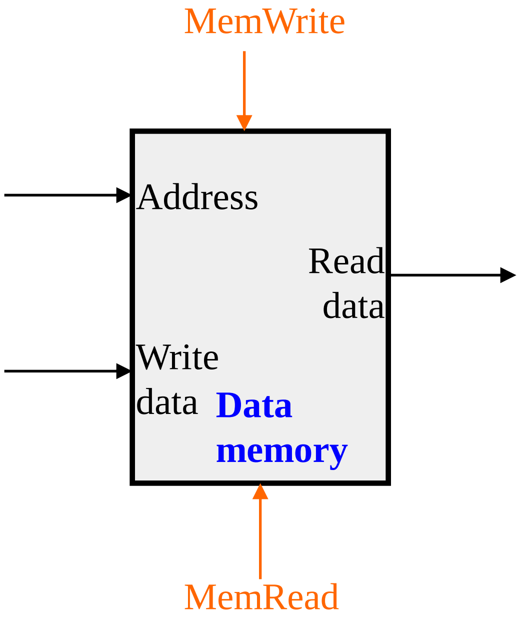
- Register
  - State element。
  - Can be controled by Write signal.



Memory

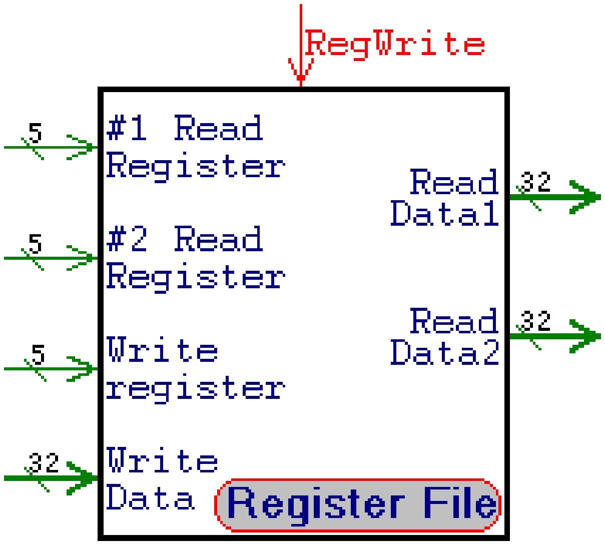
- 存储器：
  - ☞ 可分为指令存储器与数据存储器；
  - ☞ 指令存储器设为只读；输入指令地址，输出指令。
  - ☞ 数据存储器可以读写，由MemRead和MemWrite控制。按地址读出数据输入，或将写数据写入地址所指存储器单元。

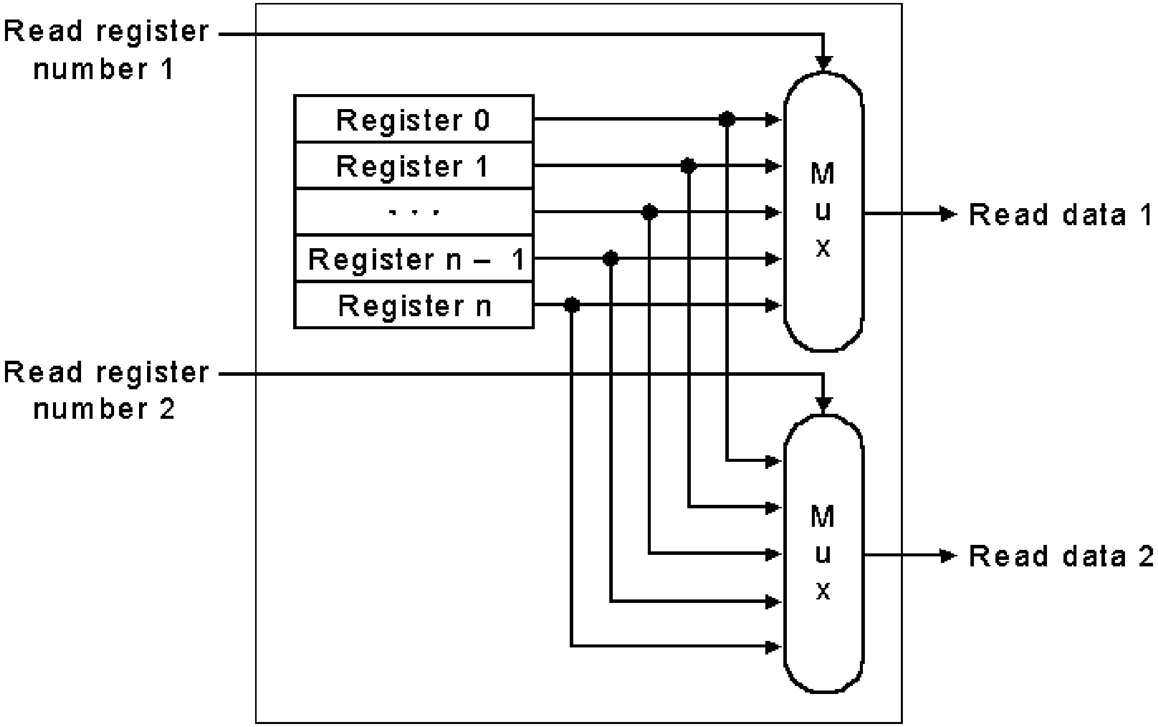




Register File

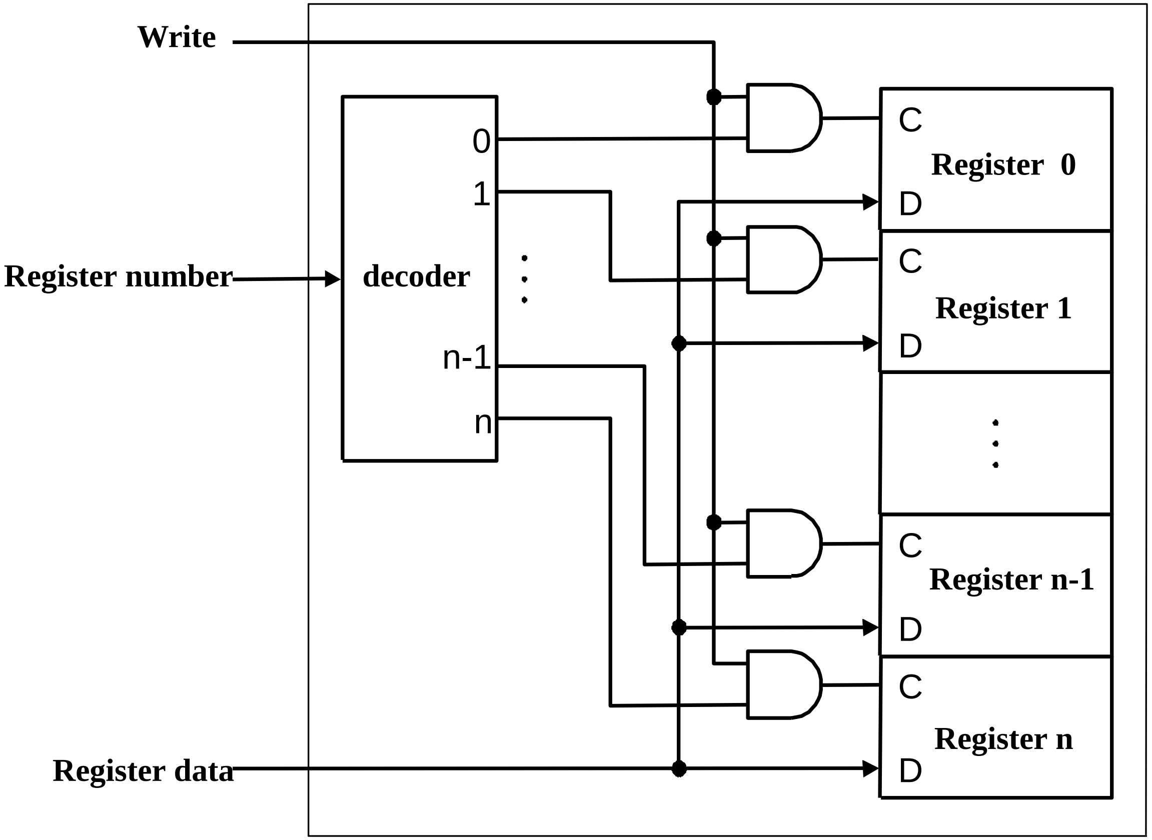
- Register File：
  - ☞ 32 32-bit Registers；
  - ☞ Input: 2 32-bit;
  - ☞ Output: 32-bit data, 32-bit register number;
  - ☞ Register write control。





Register File

- 写寄存器: we still use the real clock to determine when to write



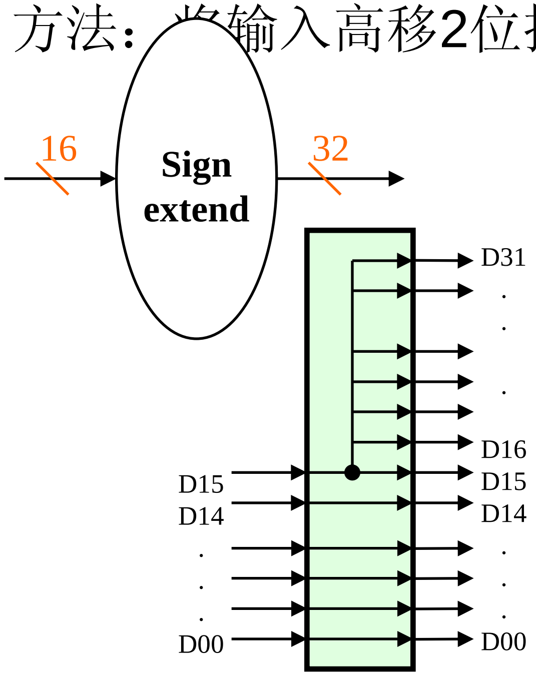
32个Register的功用

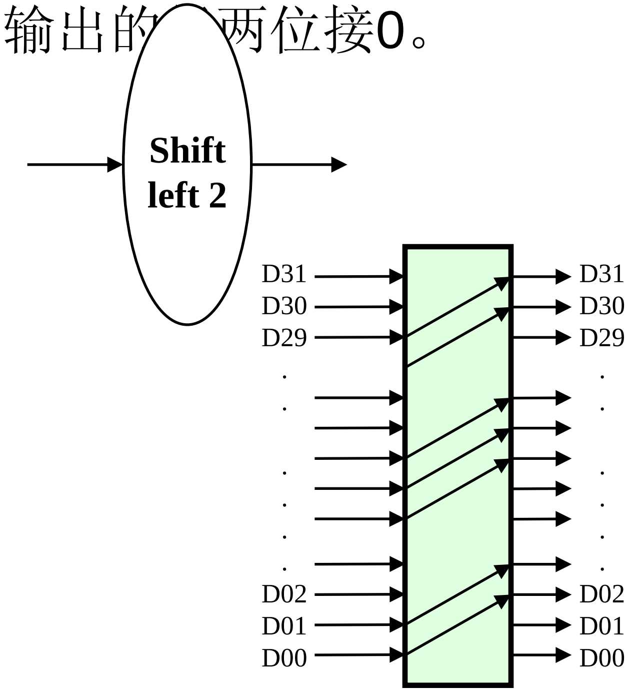
- 寄存器组有32个32位寄存器，其功用分配如下表：

Name	Register number	Usage
`$zero`	0	the constant value 0
`$at`	1	reserved for the assembler
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	reserved for the operating system
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

The other elements

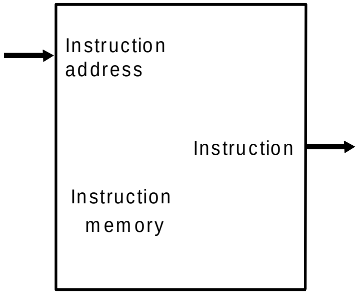
- Sign Extend：
  - ☞ 将16位的补码表示的有符号立即数，扩展为32位。
  - ☞ 方法：只需重复符号位即可。
- Shift：
  - ☞ Shift 2 bits left。
  - ☞ 方法：将输入高移2位接输出，输出的低两位接0。



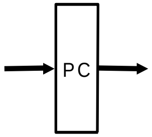


Simple Implementation

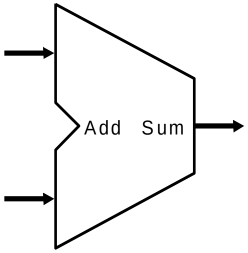
- Include the functional units we need for each instruction



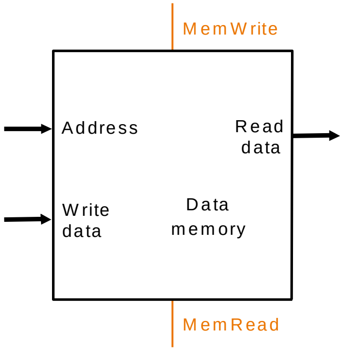
a. Instruction memory



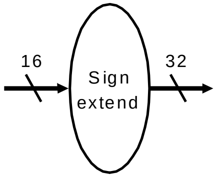
b. Program counter



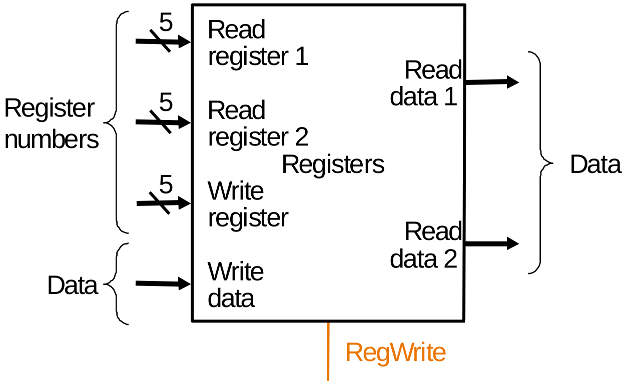
c. Adder



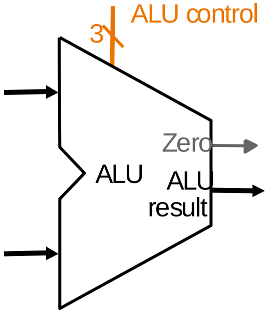
a. Data memory unit



b. Sign-extension unit



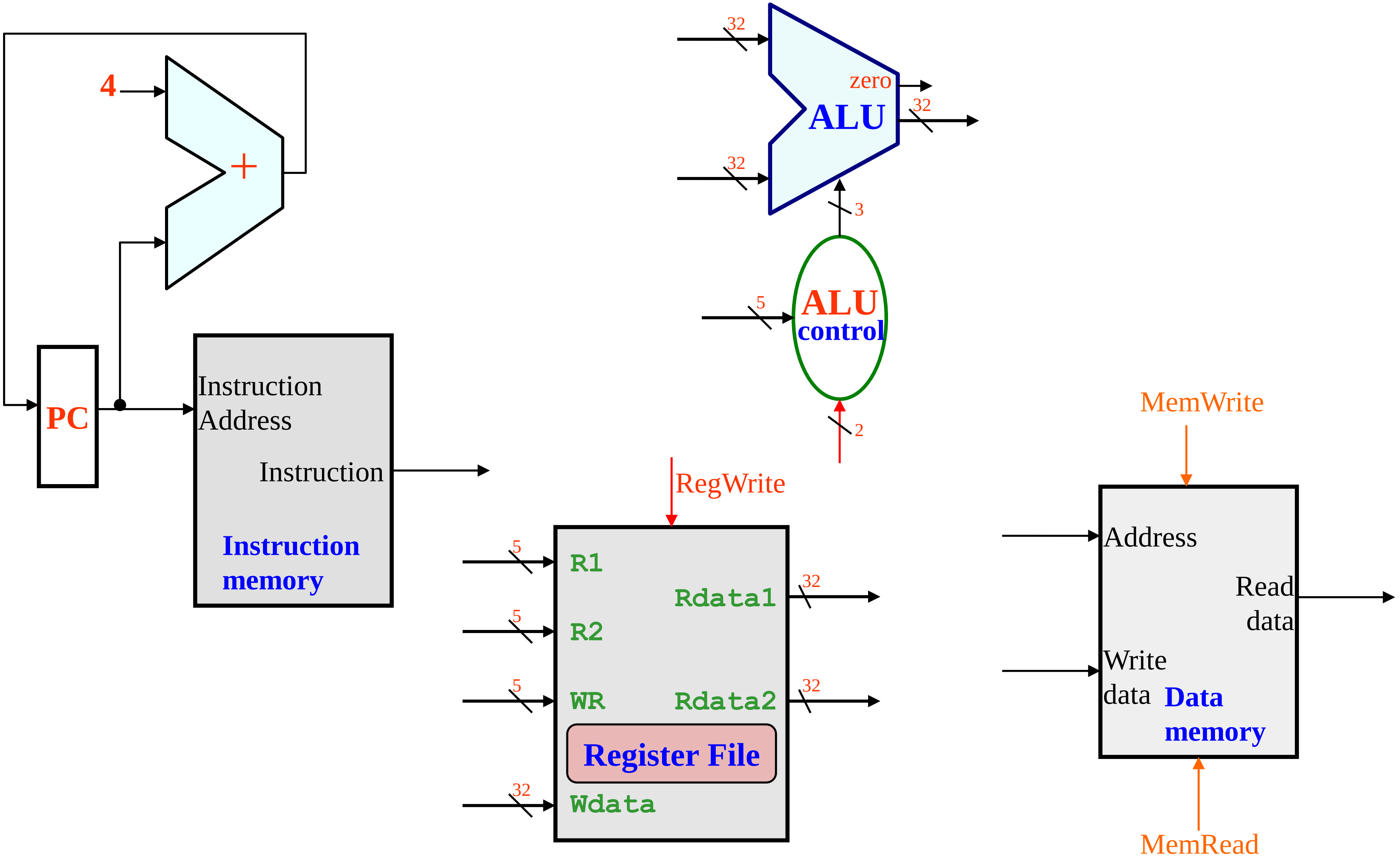
a. Registers



b. ALU

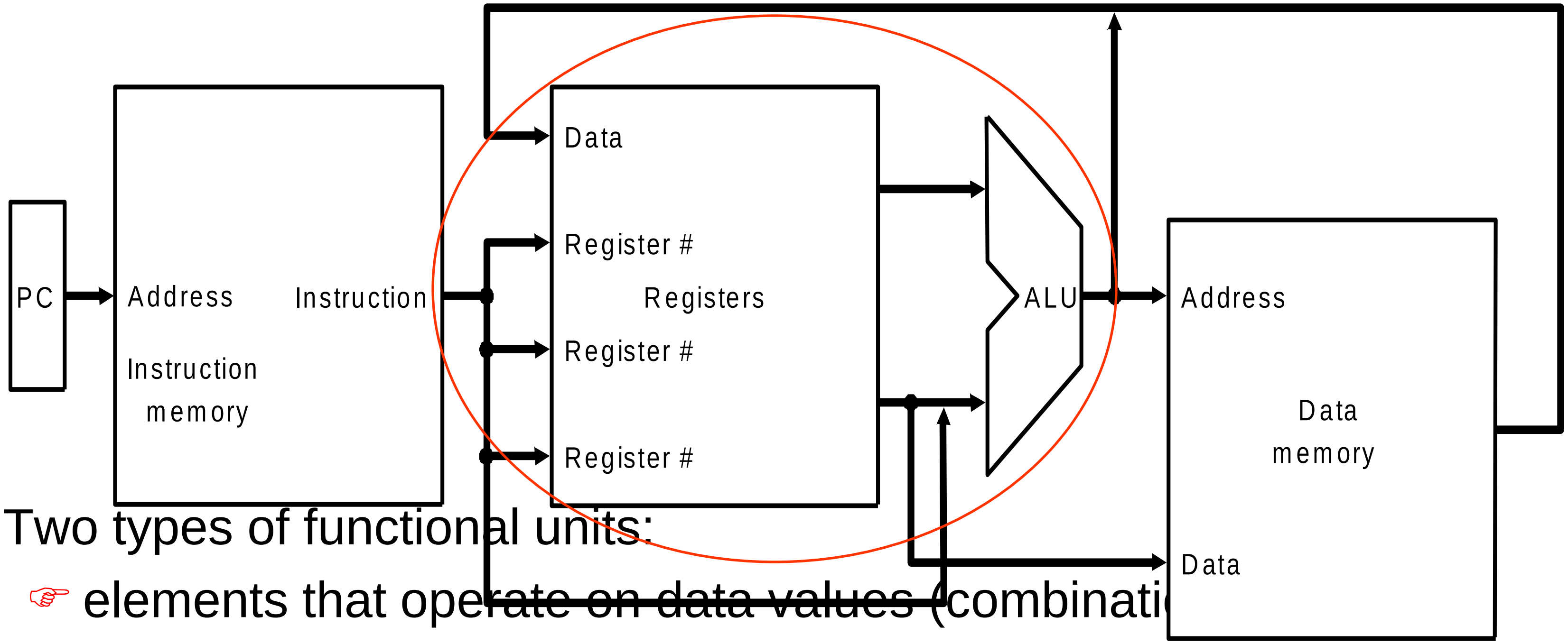
Why do we need this stuff?

Section 2   Building a datapath



More Implementation Details

- Abstract / Simplified View:

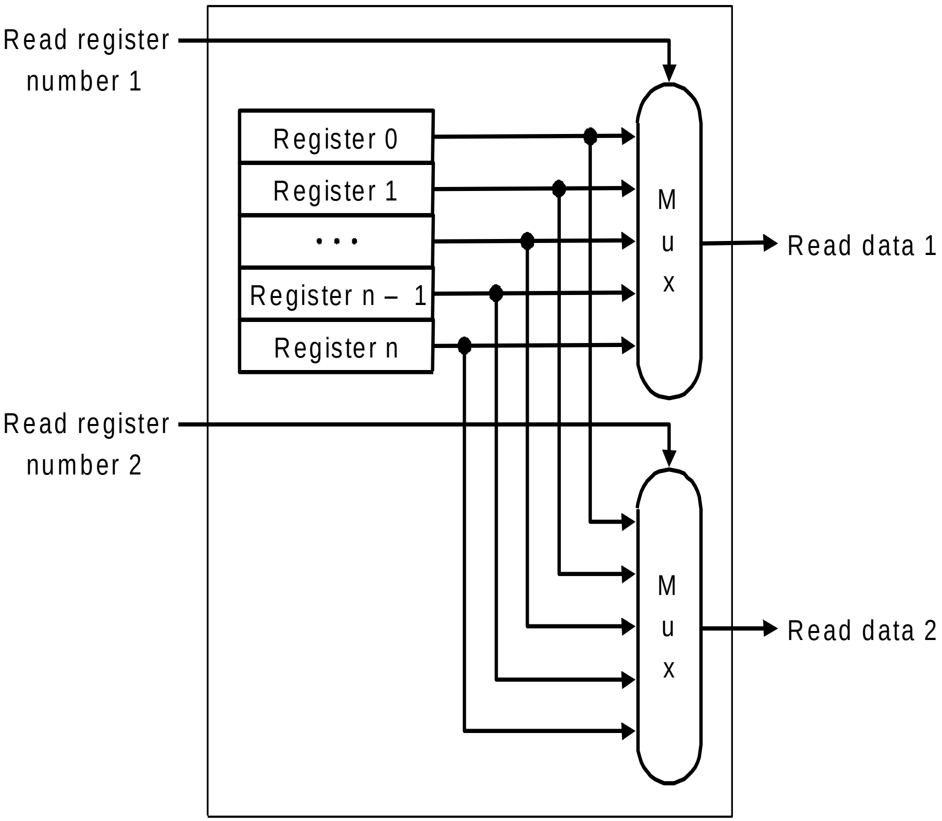


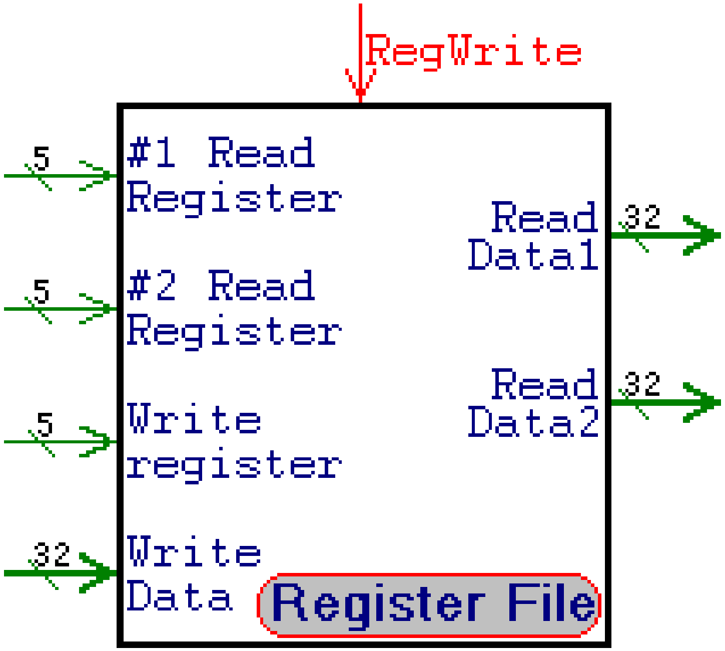
Two types of functional units:

- elements that operate on data values (combinati
- elements that contain state (sequential)

Register File

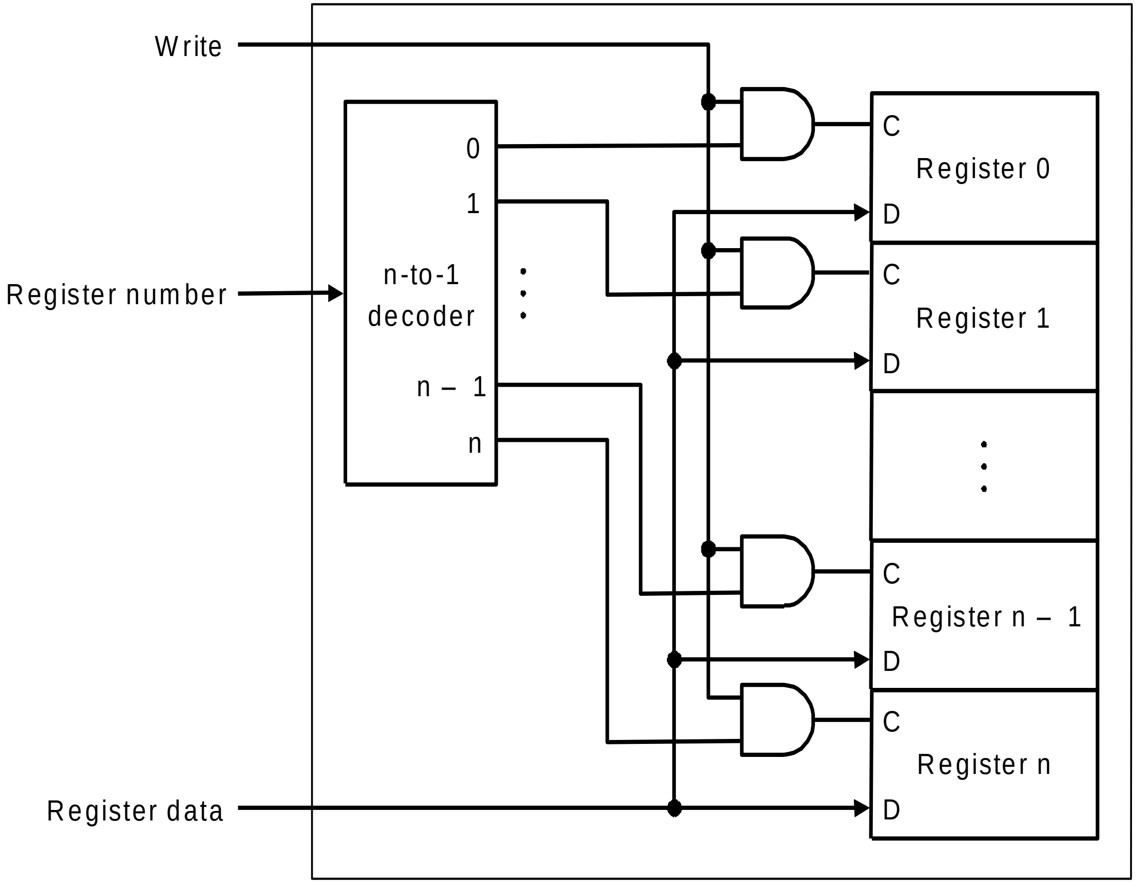
- Built using D flip-flops





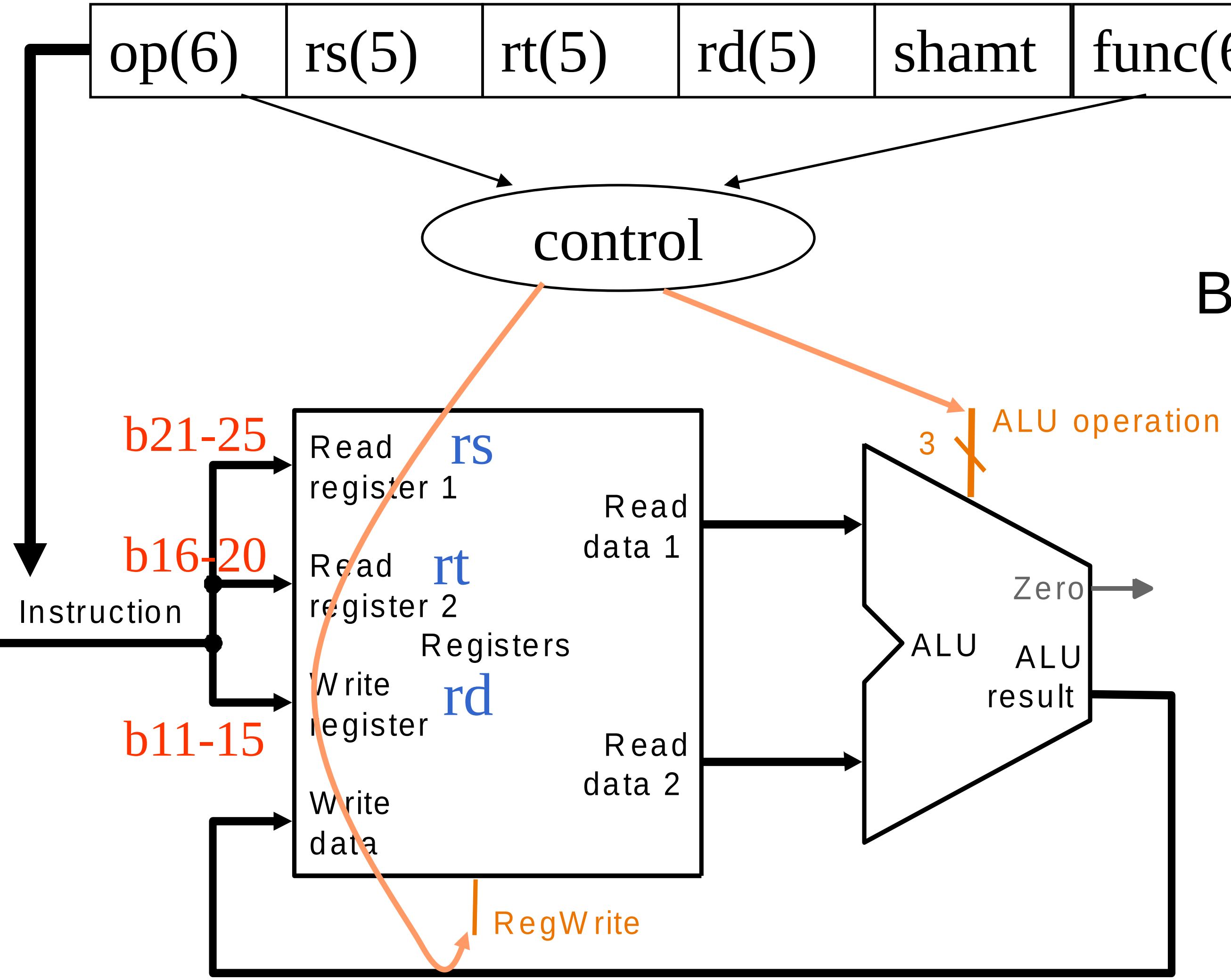
Register File

- Note: we still use the real clock to determine when to write



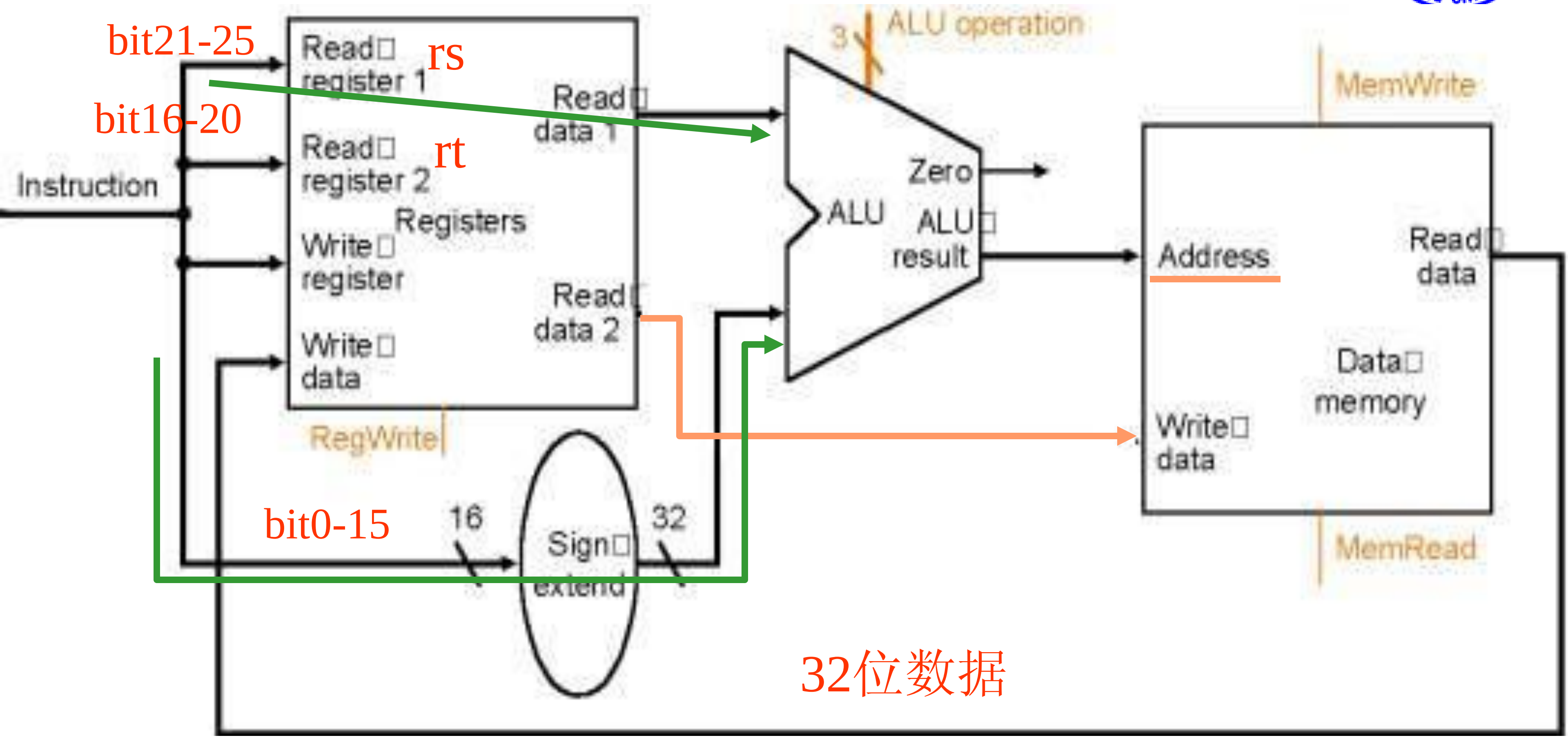
Implemente the R type instruction

instruction format：



Bnegate	op	uasge
0	00	and
0	01	or
0	10	+
1	10	-
1	11	slt

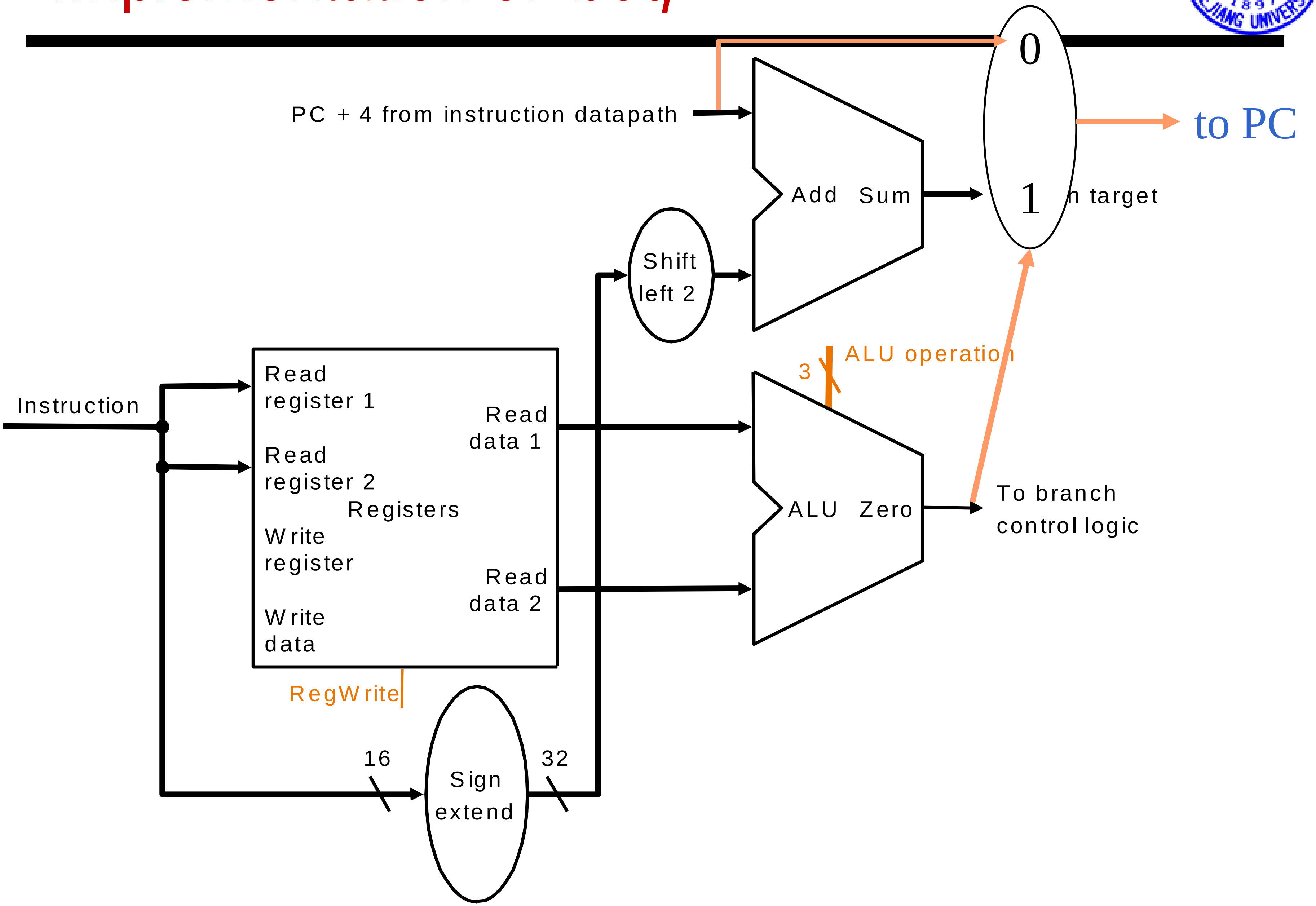
Implemente the Ｉ type instruction



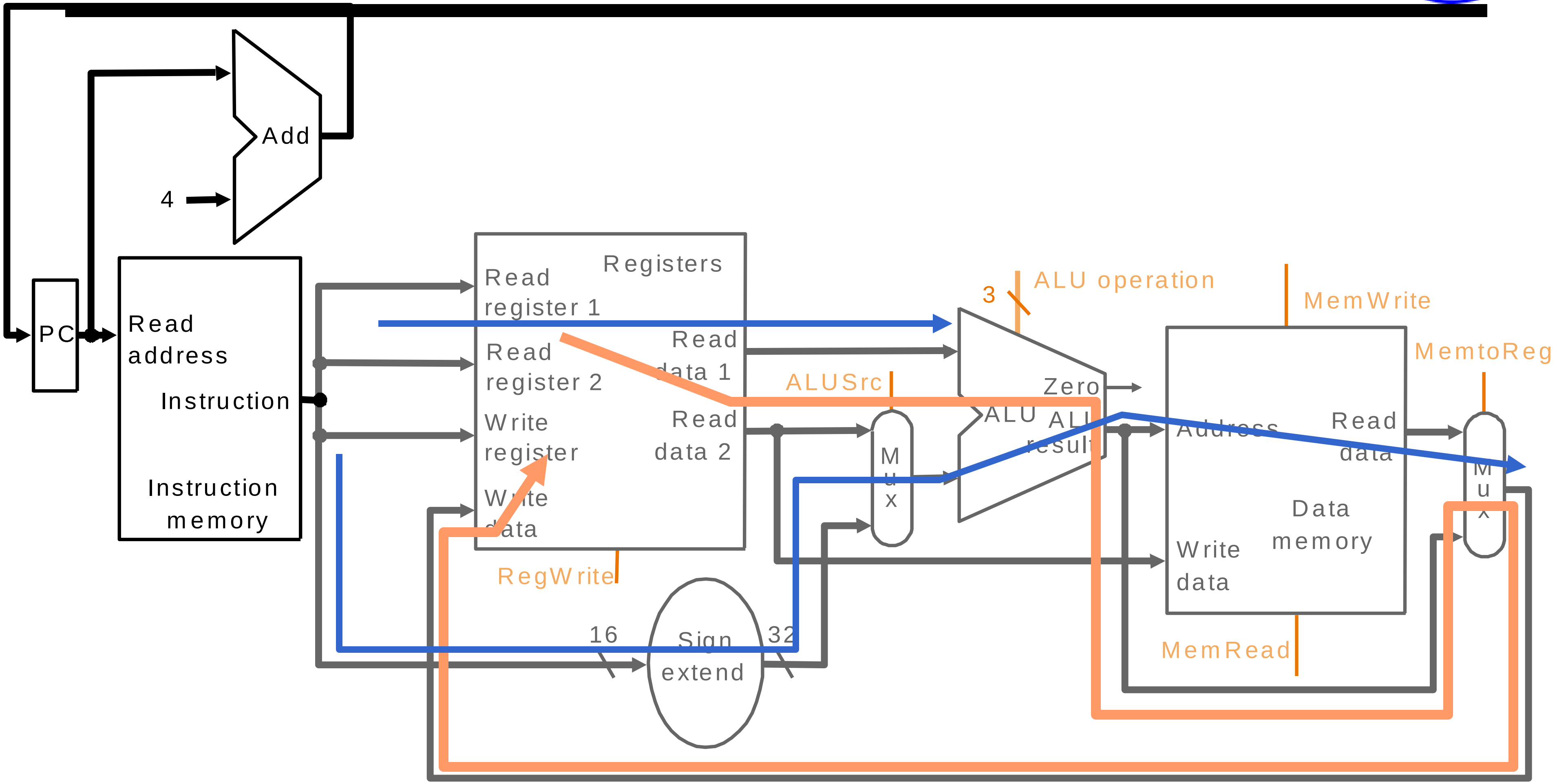
sw $t0, 200($s2)

若$s2=1000，则将$t0存入1200为地址的内存单元的一个字

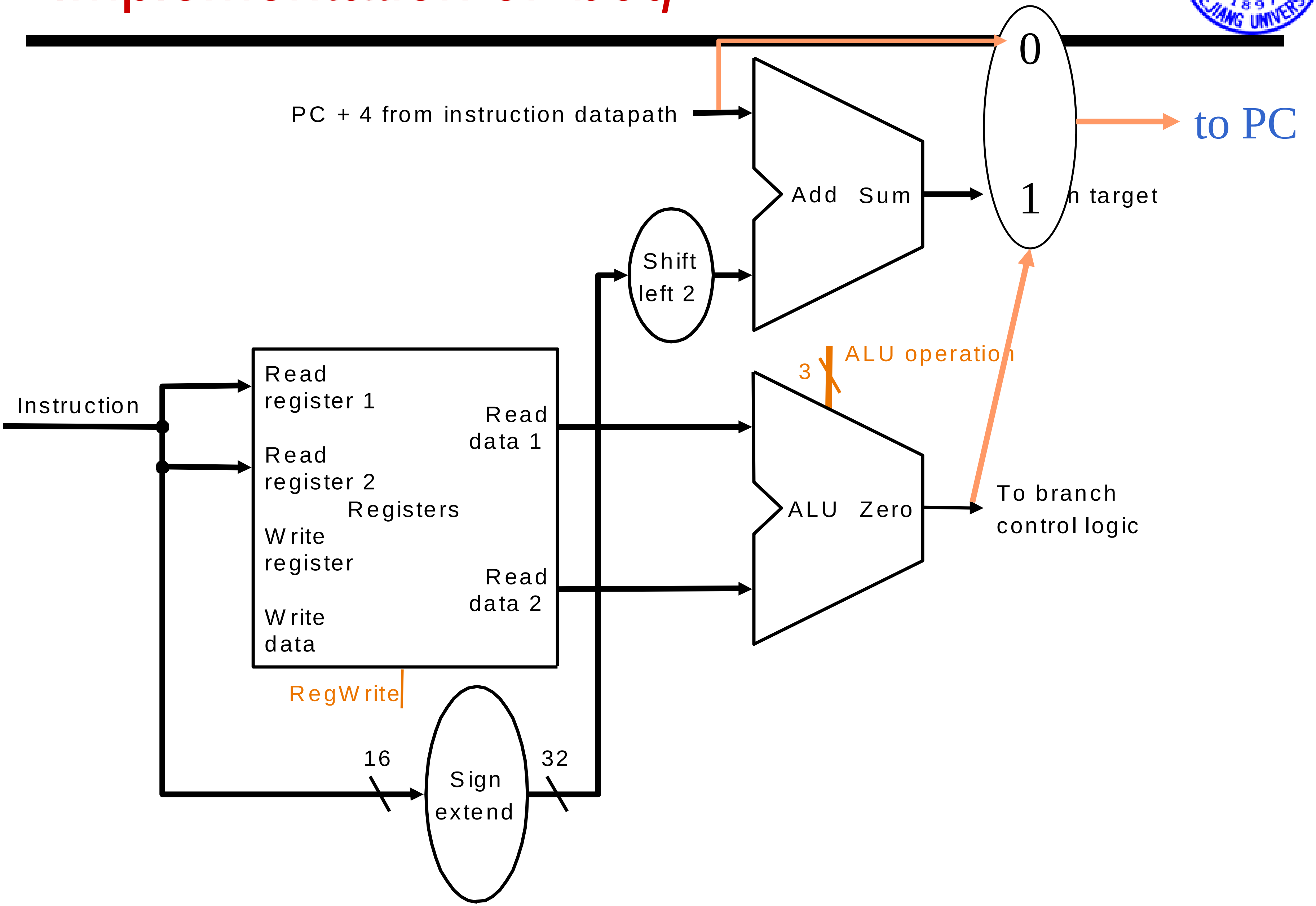
Implementation of *beq*



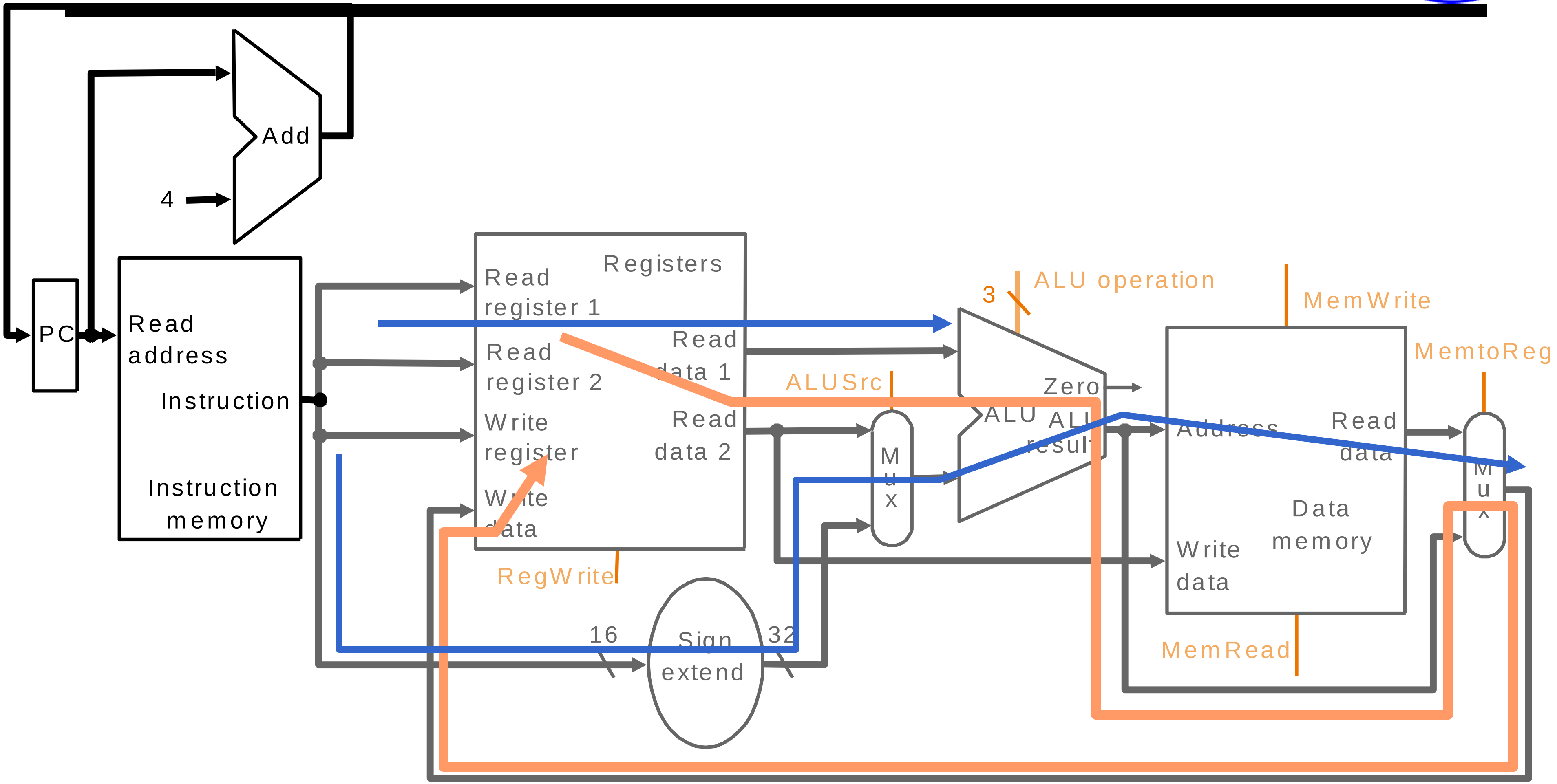
combine the implementation R-type and I type



Implementation of *beq*

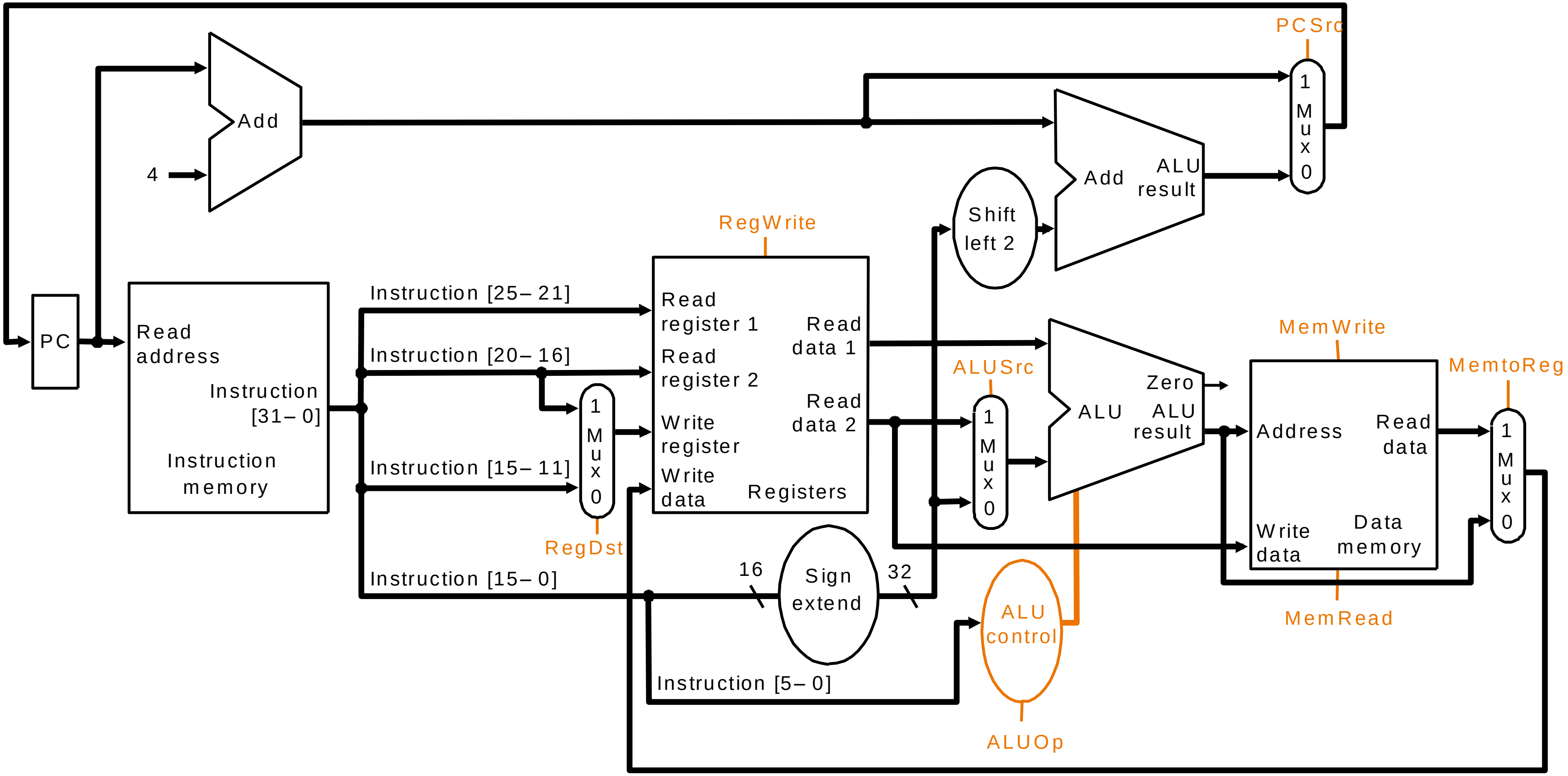


combine the implementation R-type and I type



Building the Datapath

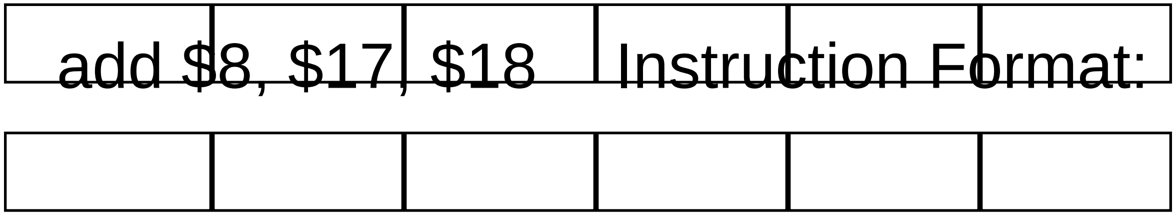
- Use multiplexors to stitch them together



Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:

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



000000   10001   10010   01000   00000   100000

op   rs   rt   rd   shamt   funct

Control

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

op	rs	rt	16 bit offset
35	2	1	100

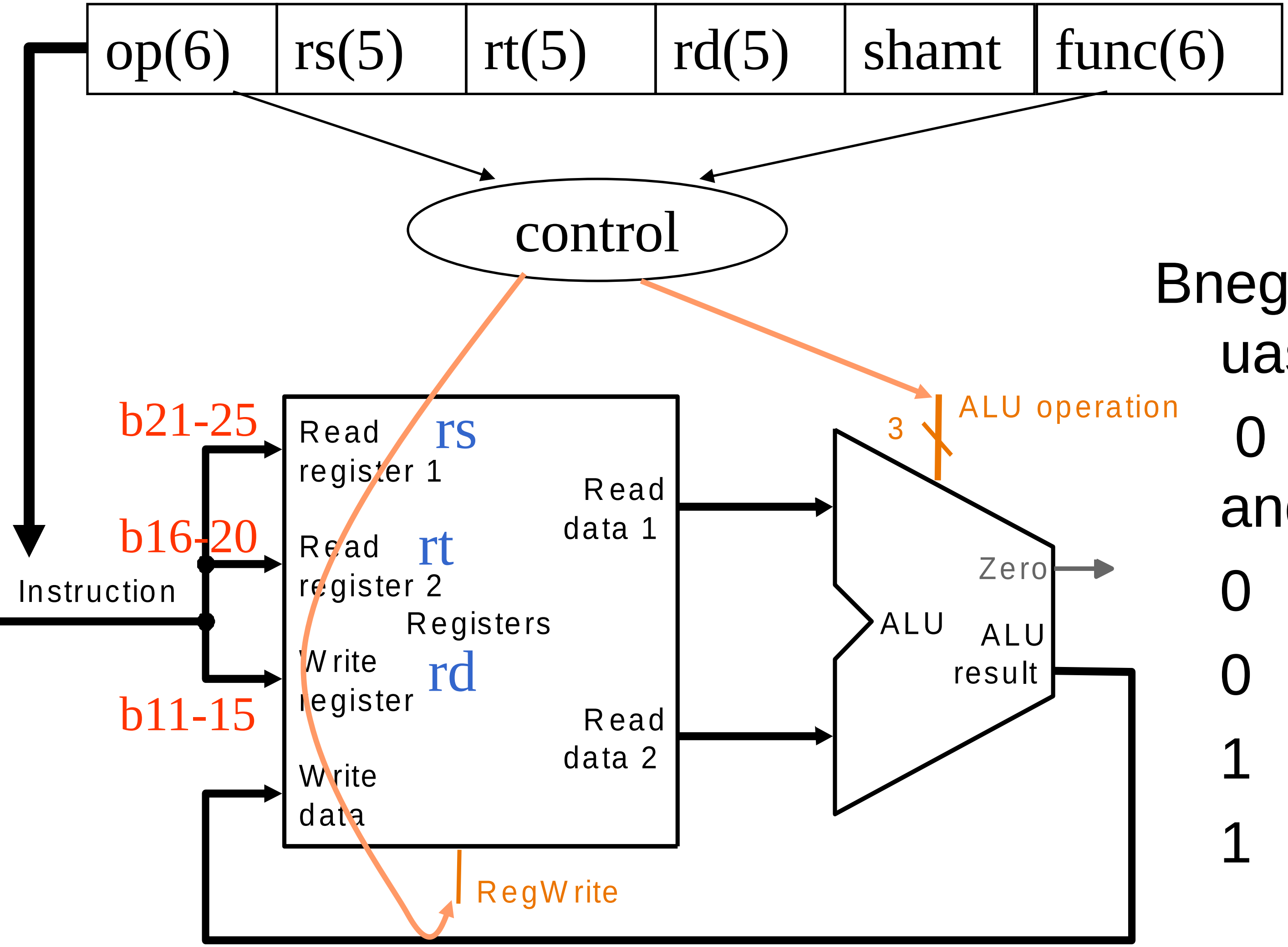
- ALU control input

000	AND
001	OR
010	add
110	subtract
111	set-on-less-than

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

The ALU control

instruction format：

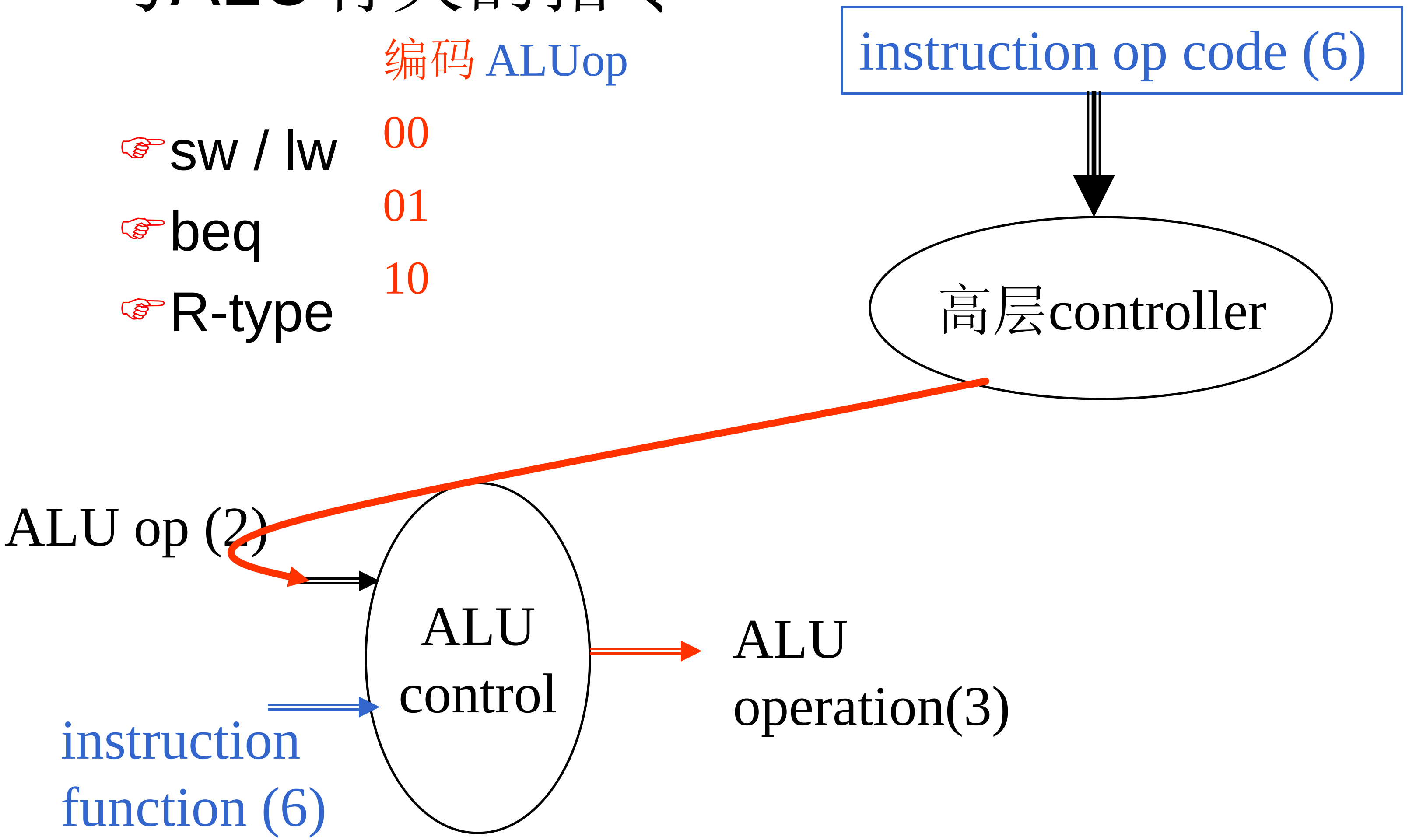


Bnegate	op	uasge
0	00	and
0	01	or
0	10	+
1	10	-
1	11	slt

The ALU control

- 与ALU有关的指令

	编码 ALUop
☞sw / lw	00
☞beq	01
☞R-type	10



The ALU control

- Must describe hardware to compute 3-bit ALU conrol input
  - given instruction type
    00 = lw, sw
    01 = beq,
    10 = R-type
  - function code for arithmetic

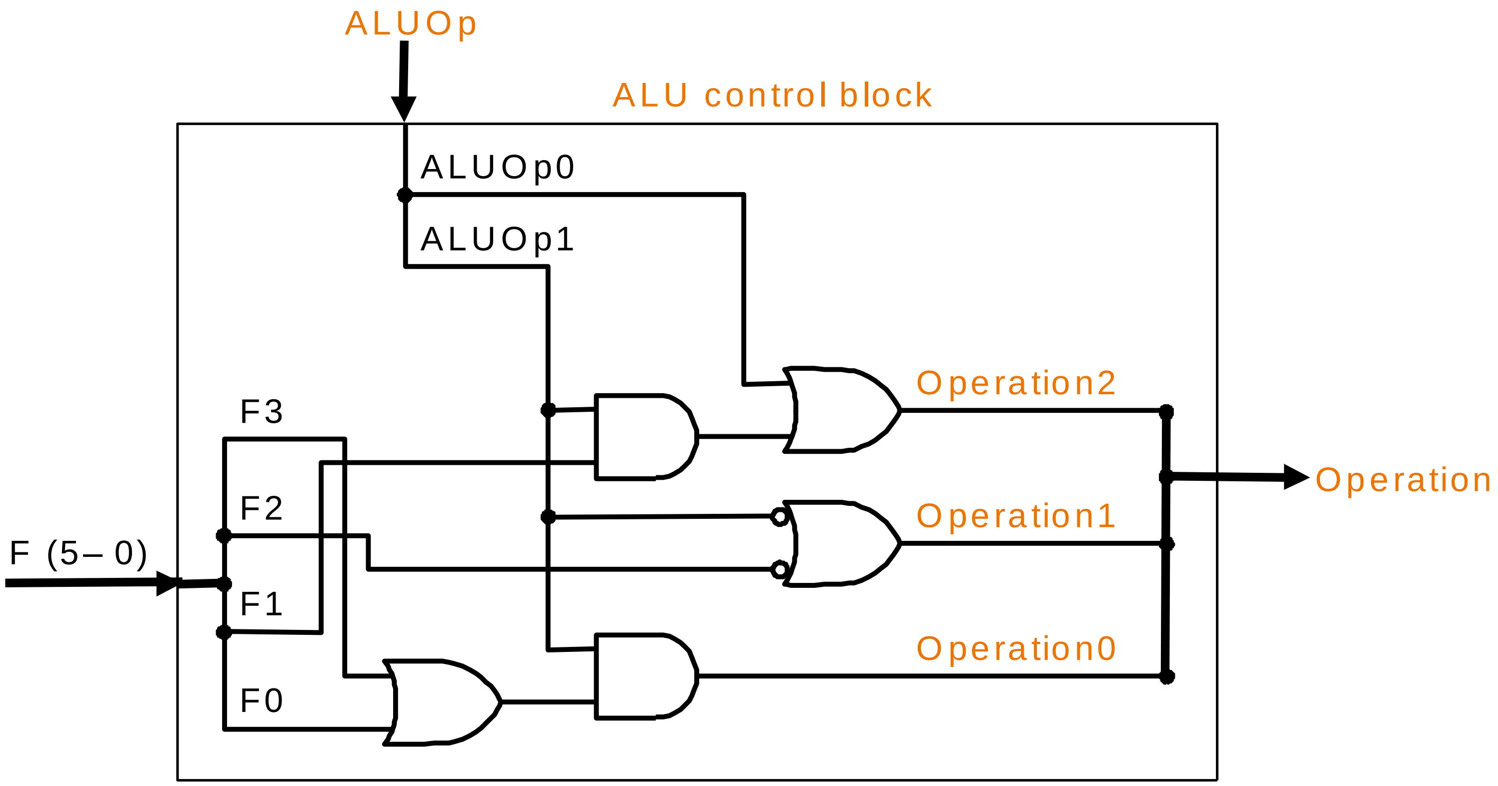
ALUOp computed from instruction type

- Describe it es):

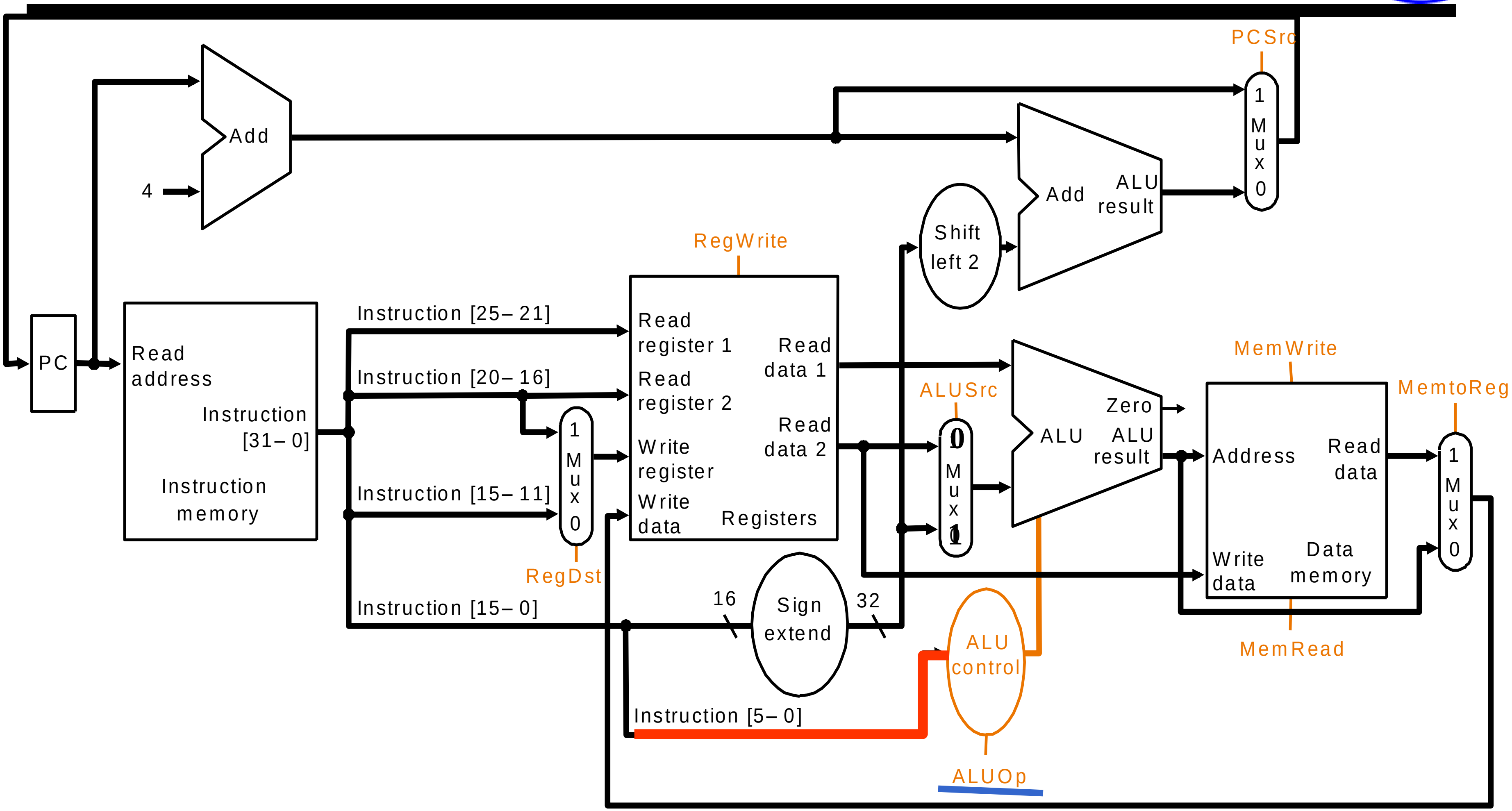
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

don’t care

The ALU control

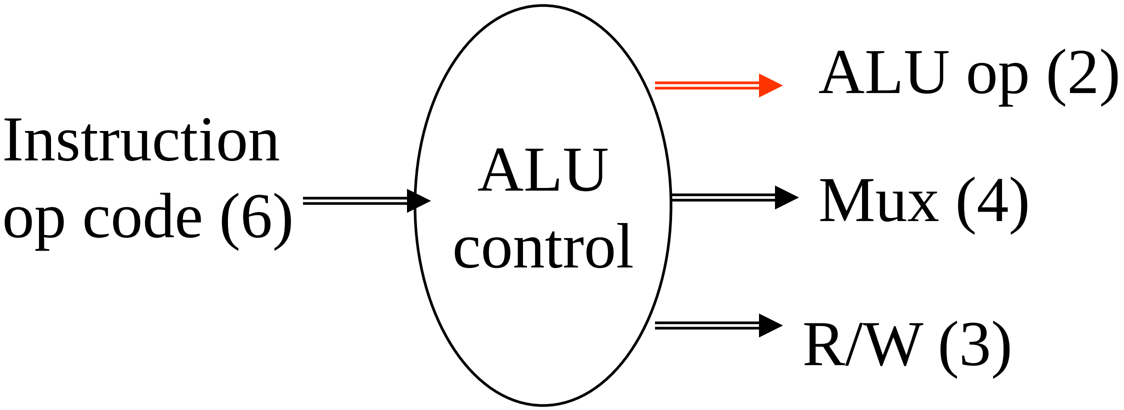


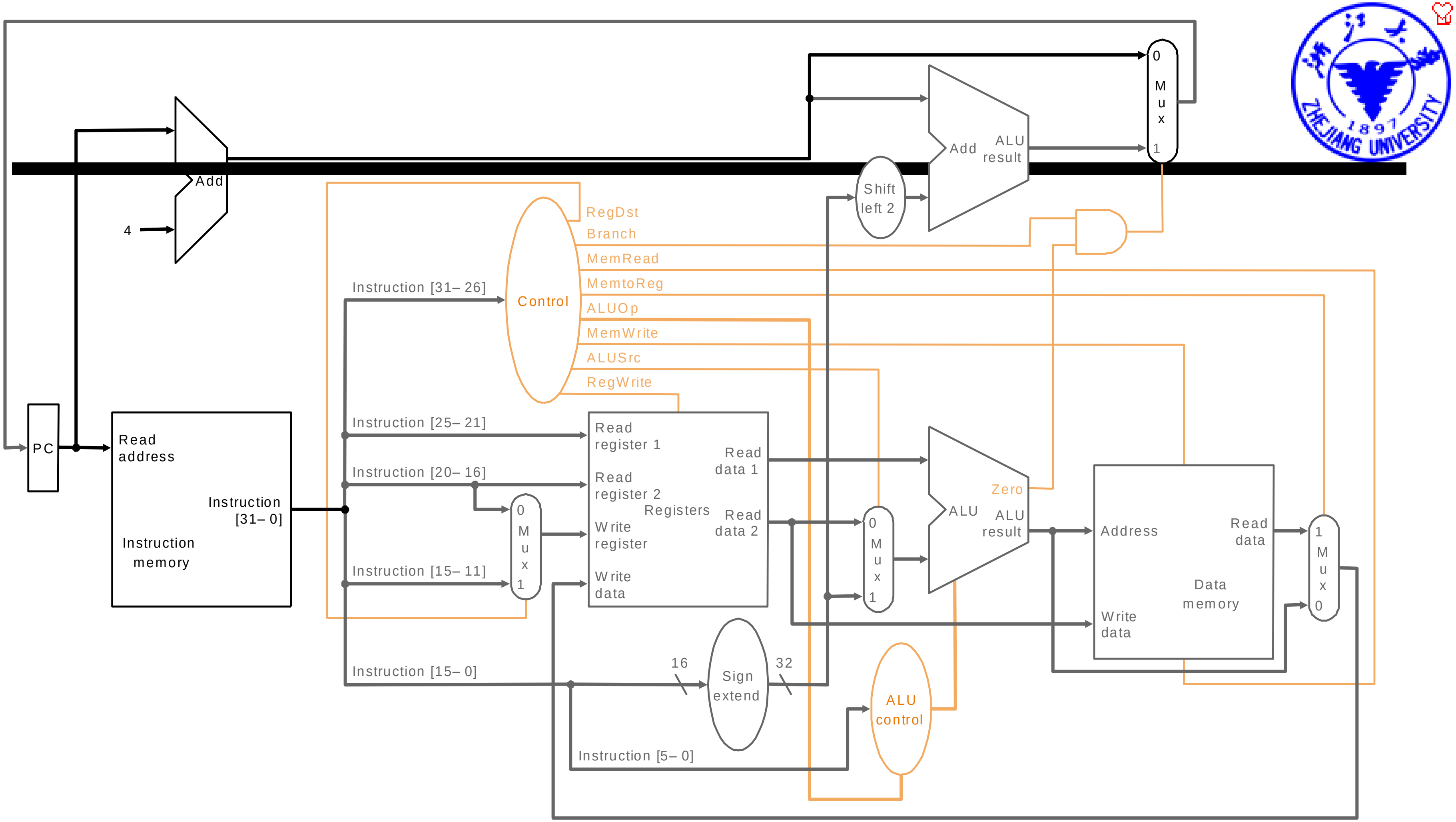
The ALU control



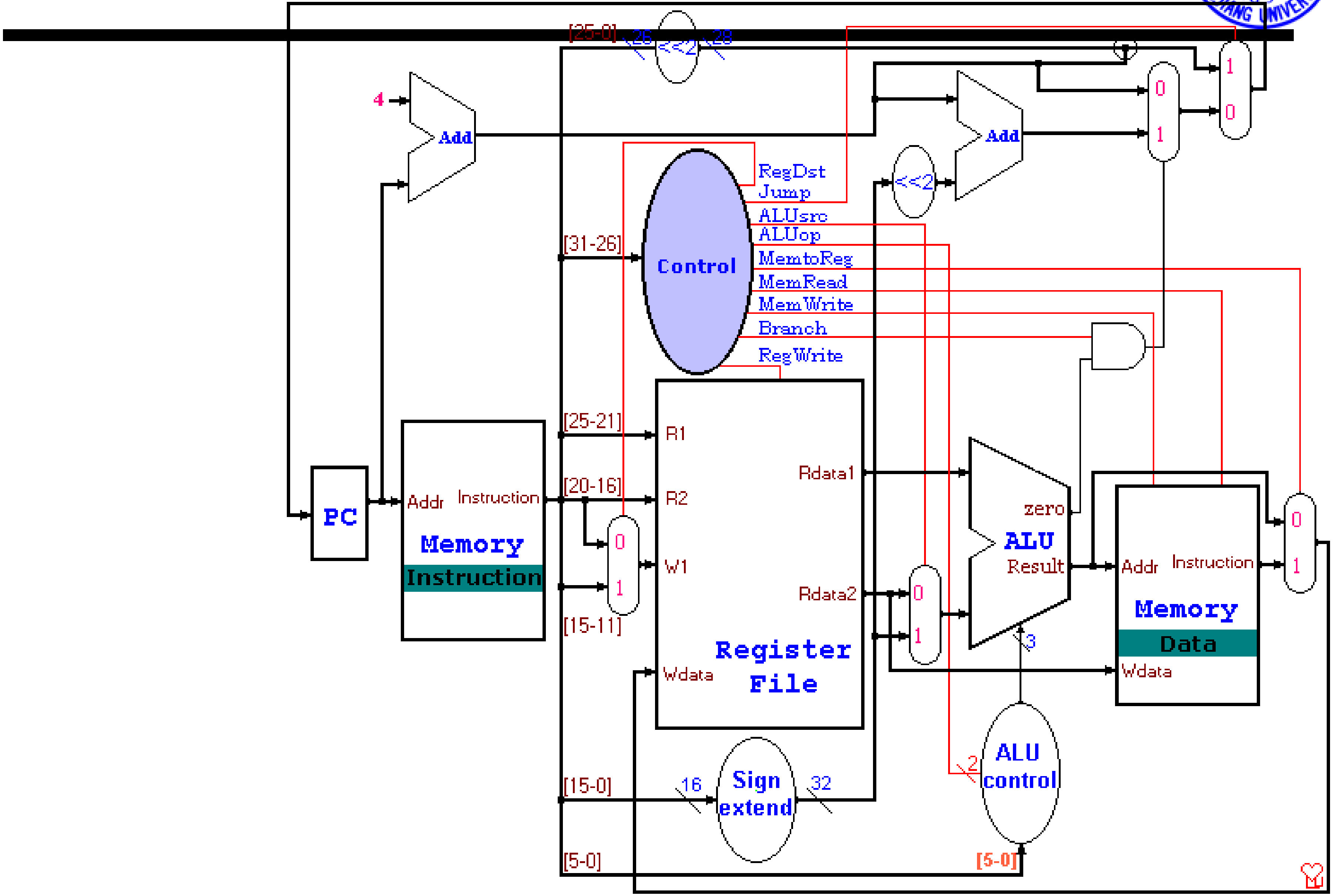
Designing the Main Control Unit

- Main Control Unit function
  - ☞ALU op (2)
  - ☞other control signals (p. 359)
    - ✧4 Mux
    - ✧3 R/W





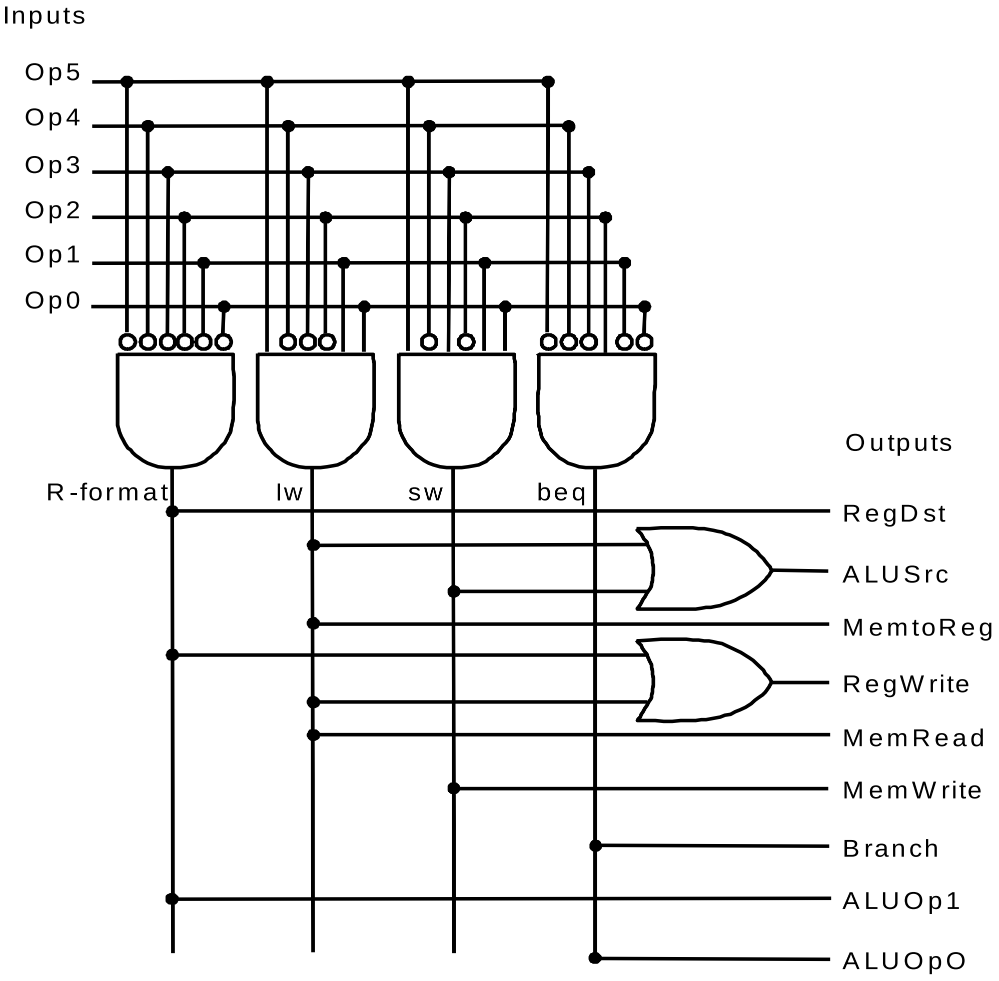
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1


[25-0]
26
<<2
28
4
Add
Add
<<2
RegDst
Jump
ALUsrc
ALUop
MemtoReg
MemRead
MemWrite
Branch
RegWrite
[31-26]
Control
0
1
0
1
[25-21]
[20-16]
[15-11]
[15-0]
[5-0]
PC
Addr
Instruction
Memory
Instruction
R1
R2
W1
Wdata
Rdata1
Rdata2
Register
File
0
1
0
1
zero
ALU
Result
3
Addr
Instruction
Memory
Data
Wdata
0
1
16
Sign
extend
32
2
ALU
control
[5-0]

Control

- Simple combinational logic (truth tables)

R-type	0
lw	35
sw	43
beq	4

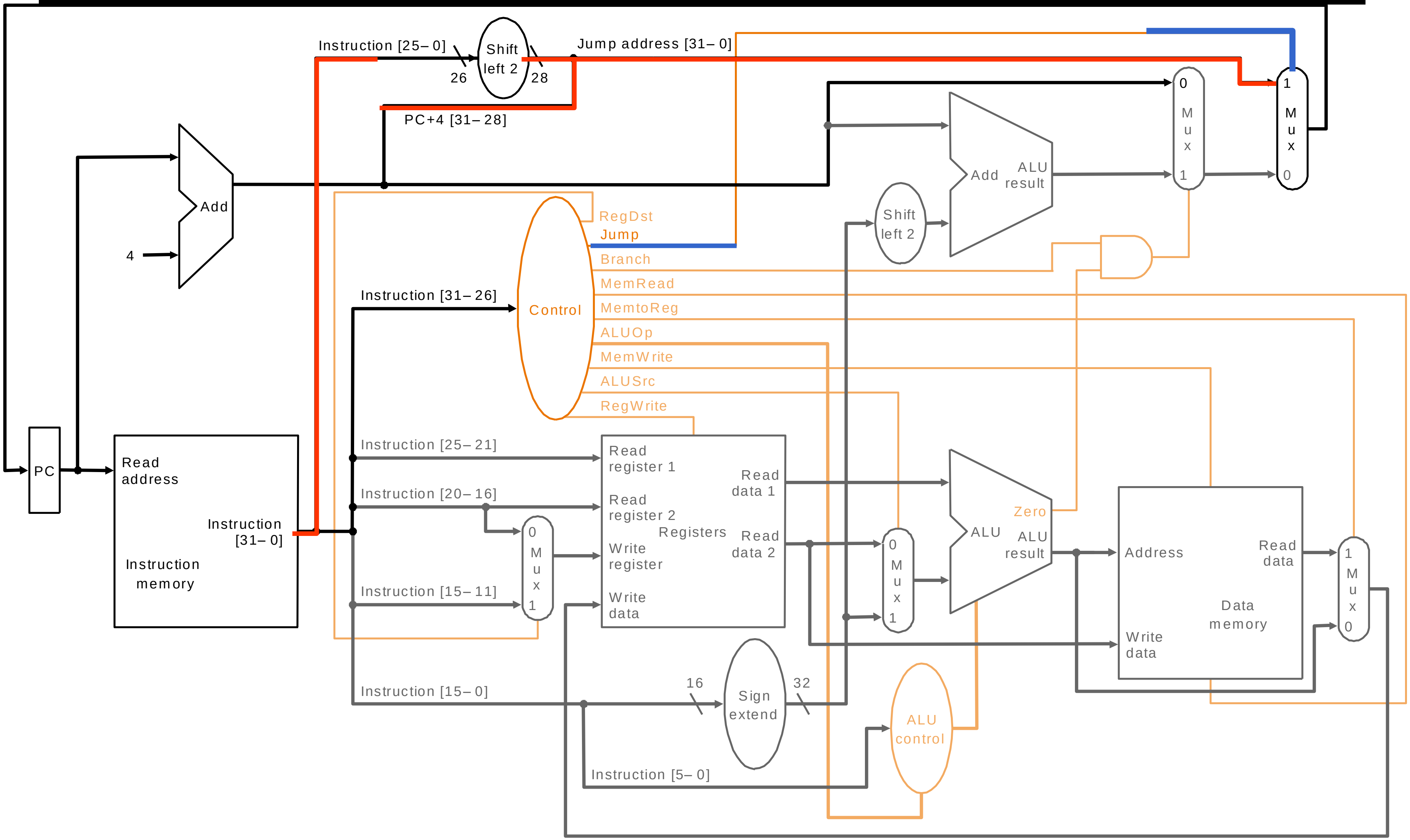


j instruction

- instruction format
  - ☞j Label

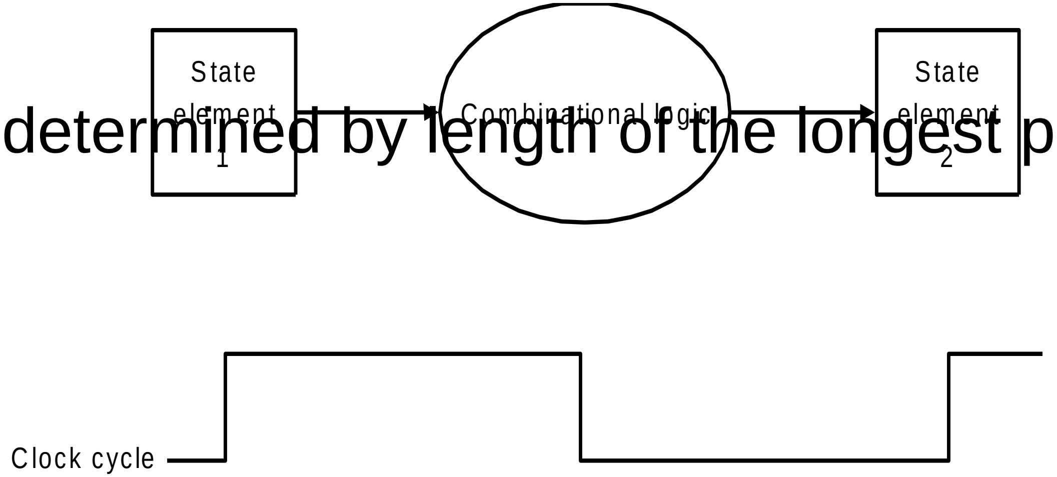
op=2	26 bits address

- Implementation
  - ☞$pc = pc_{28\sim31} + 26bits\text{-}address * 4$


Instruction [25– 0]
26
Shift
left 2
28
Jump address [31– 0]
PC+4 [31– 28]
Add
4
0
M
u
x
1
1
M
u
x
0
Add
ALU
result
Shift
left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31– 26]
Control
PC
Read
address
Instruction
[31– 0]
Instruction
memory
Instruction [25– 21]
Instruction [20– 16]
Instruction [15– 11]
Instruction [15– 0]
0
M
u
x
1
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
0
M
u
x
1
ALU
Zero
ALU
result
Address
Read
data
Write
data
Data
memory
1
M
u
x
0
16
Sign
extend
32
ALU
control
Instruction [5– 0]

Our Simple Control Structure

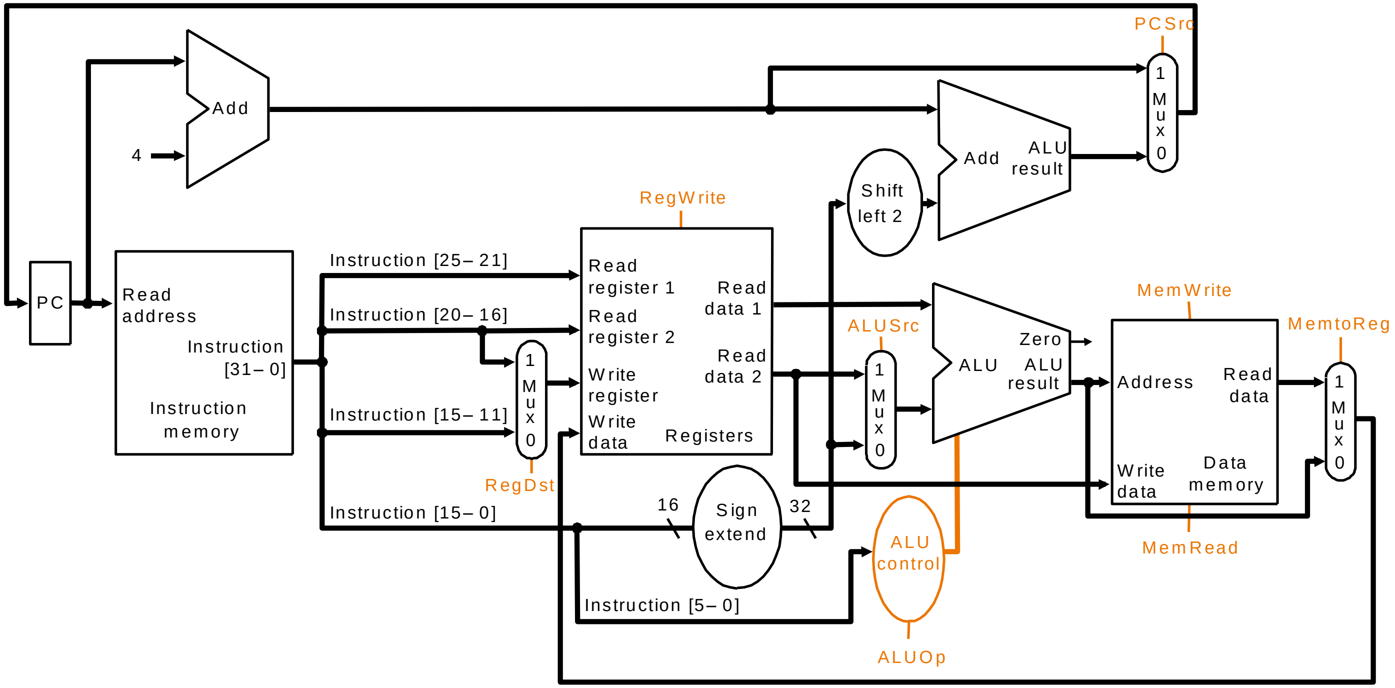
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer?right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

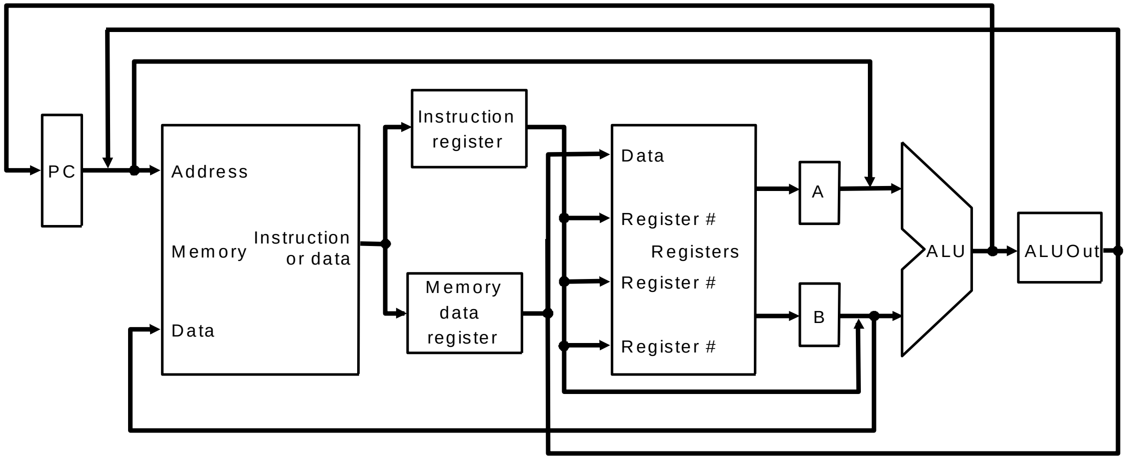
Single Cycle Implementation

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

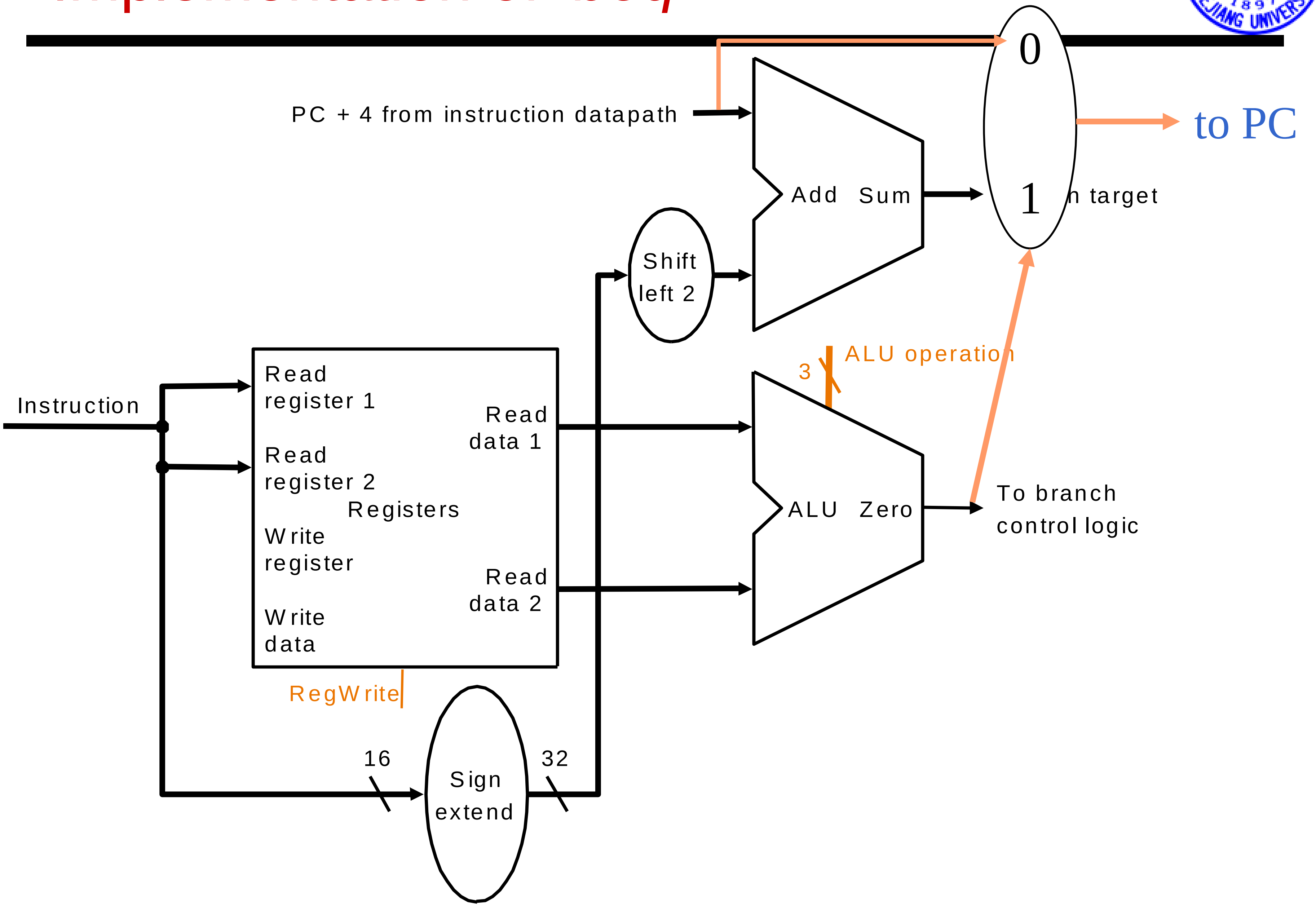


Where we are headed

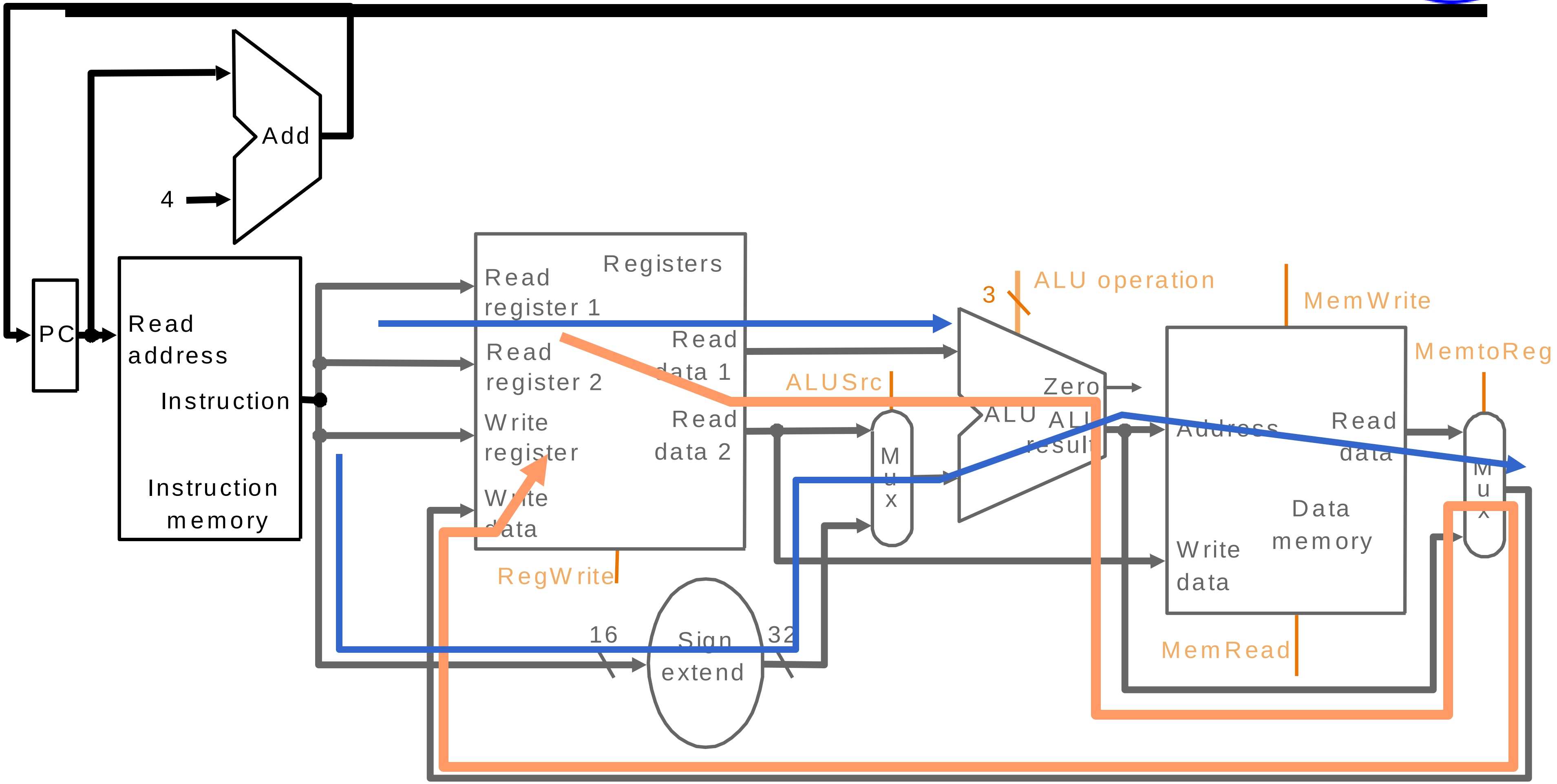
- Single Cycle Problems:
  - ☞ what if we had a more complicated instruction like floating point?
  - ☞ wasteful of area
- One Solution:
  - ☞ use a 搒maller?cycle time
  - ☞ have different instructions take different numbers of cycles
  - ☞ a 搈ulticycle?datapath:



Implementation of *beq*

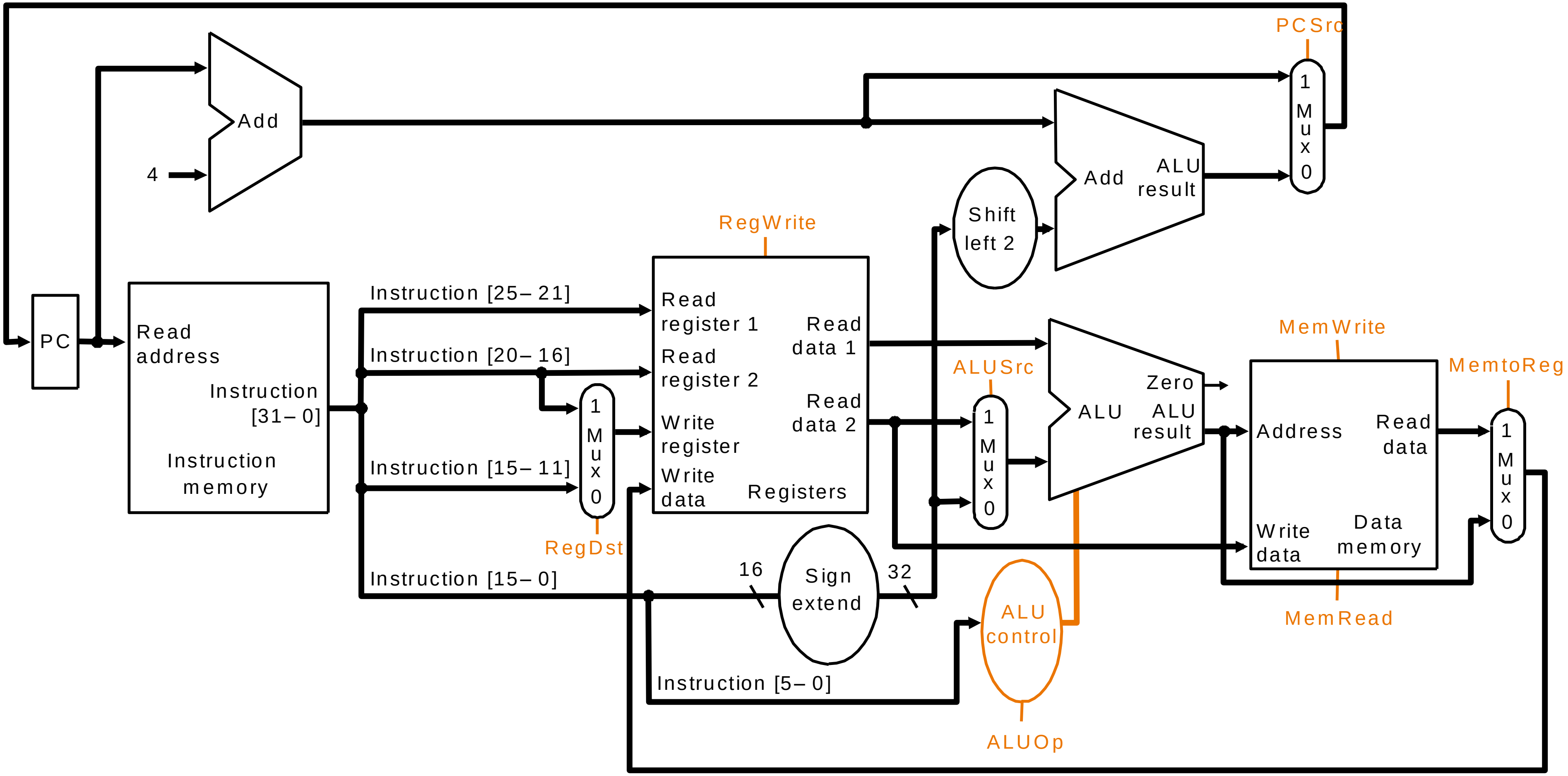


combine the implementation R-type and I type



Building the Datapath

- Use multiplexors to stitch them together



Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:

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

000000   10001   10010   01000
00000   100000

op   rs   rt   rd   shamt   funct

Control

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

op	rs	rt	16 bit offset
35	2	1	100

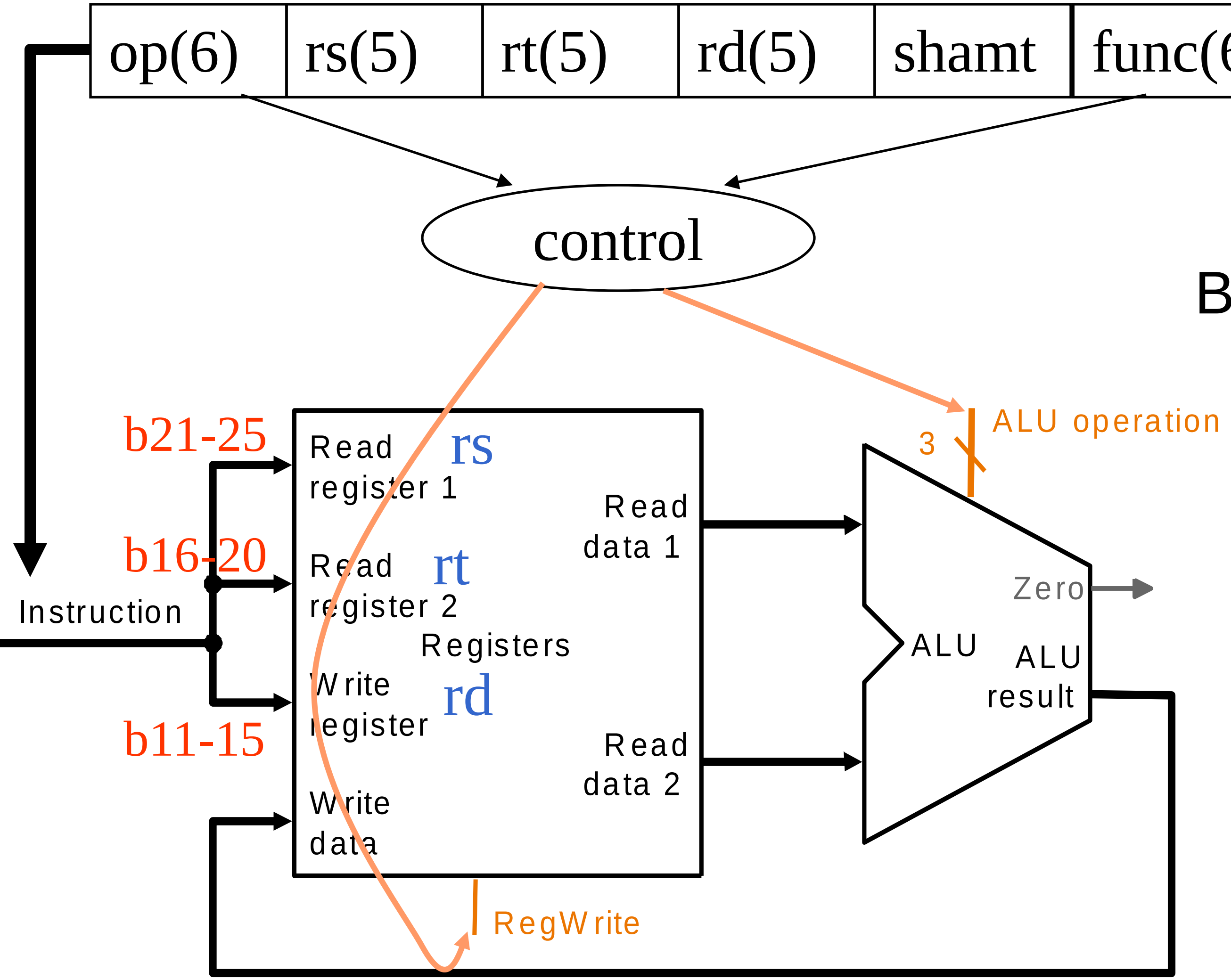
- ALU control input

000	AND
001	OR
010	add
110	subtract
111	set-on-less-than

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

The ALU control

instruction format：

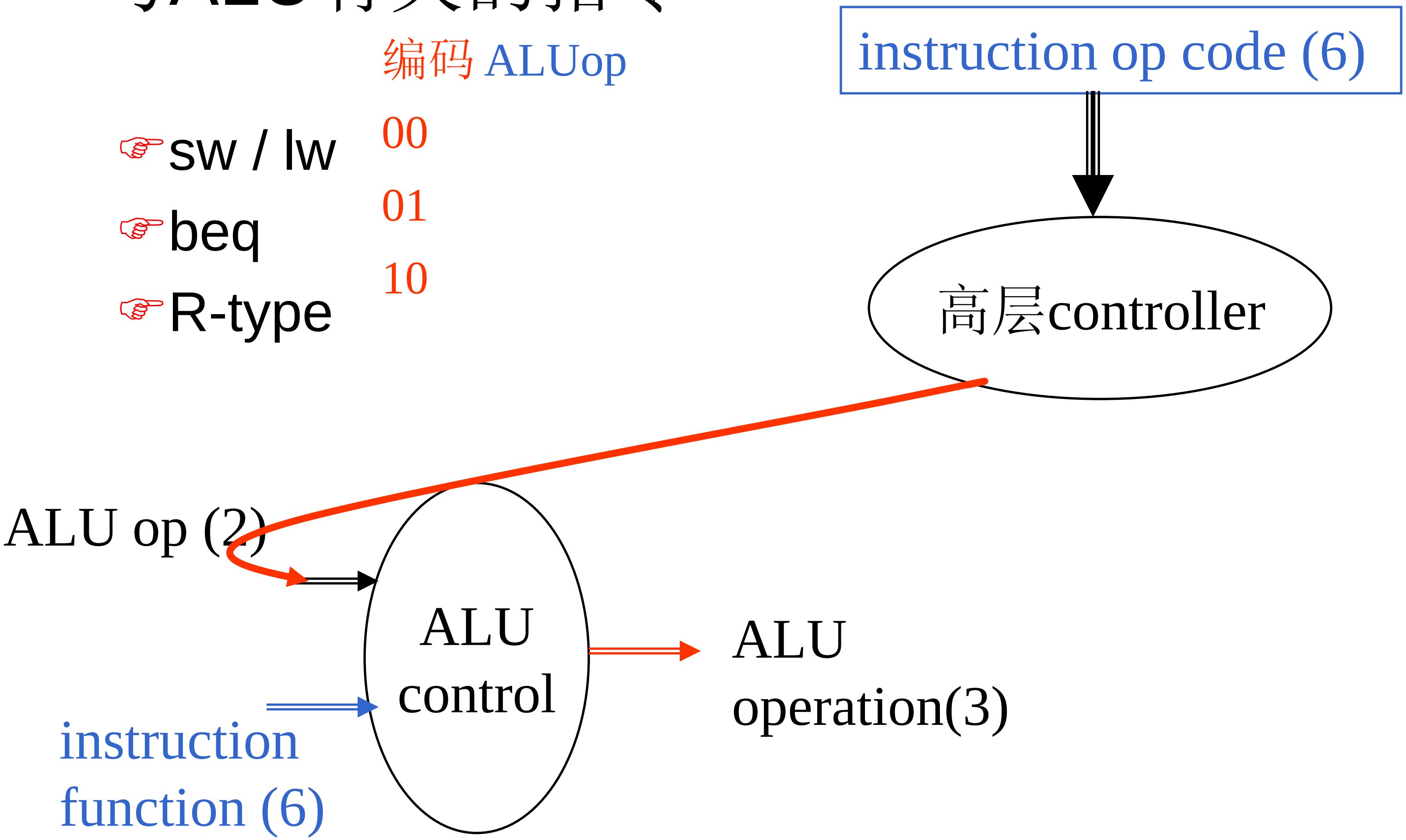


Bnegate	op	uasge
0	00	and
0	01	or
0	10	+
1	10	-
1	11	slt

The ALU control

- 与ALU有关的指令

	编码 ALUop
☞sw / lw	00
☞beq	01
☞R-type	10



The ALU control

- Must describe hardware to compute 3-bit ALU conrol input
  - ☞given instruction type
    - 00 = lw, sw
    - 01 = beq,
    - 10 = R-type
  - ☞function code for arithmetic

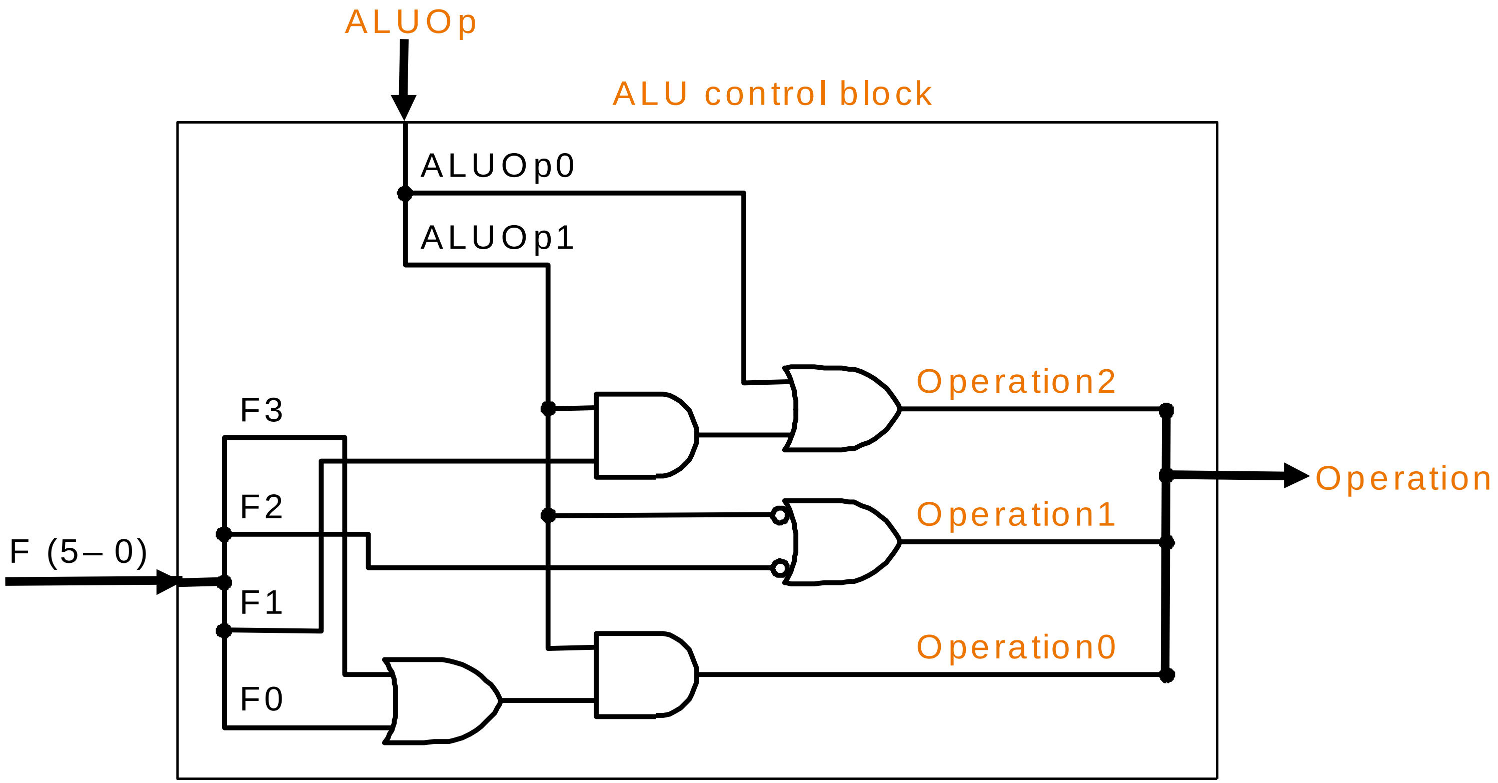
ALUOp computed from instruction type

- Describe it [illegible] es):

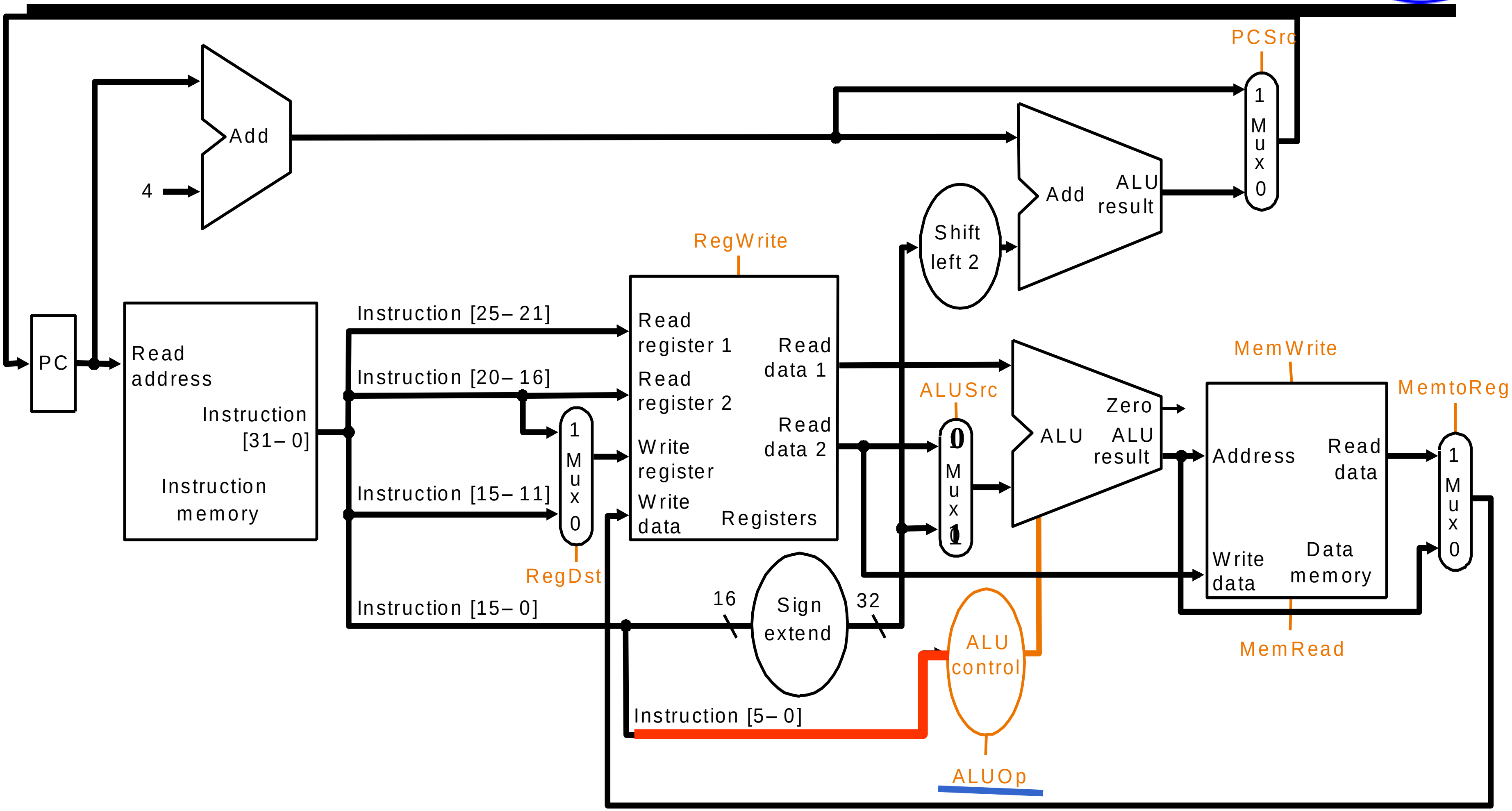
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

don’t care

The ALU control

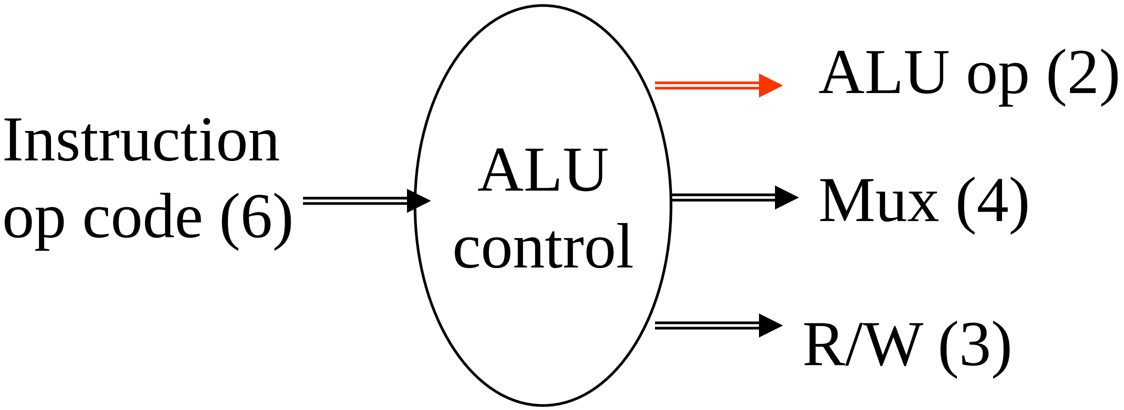


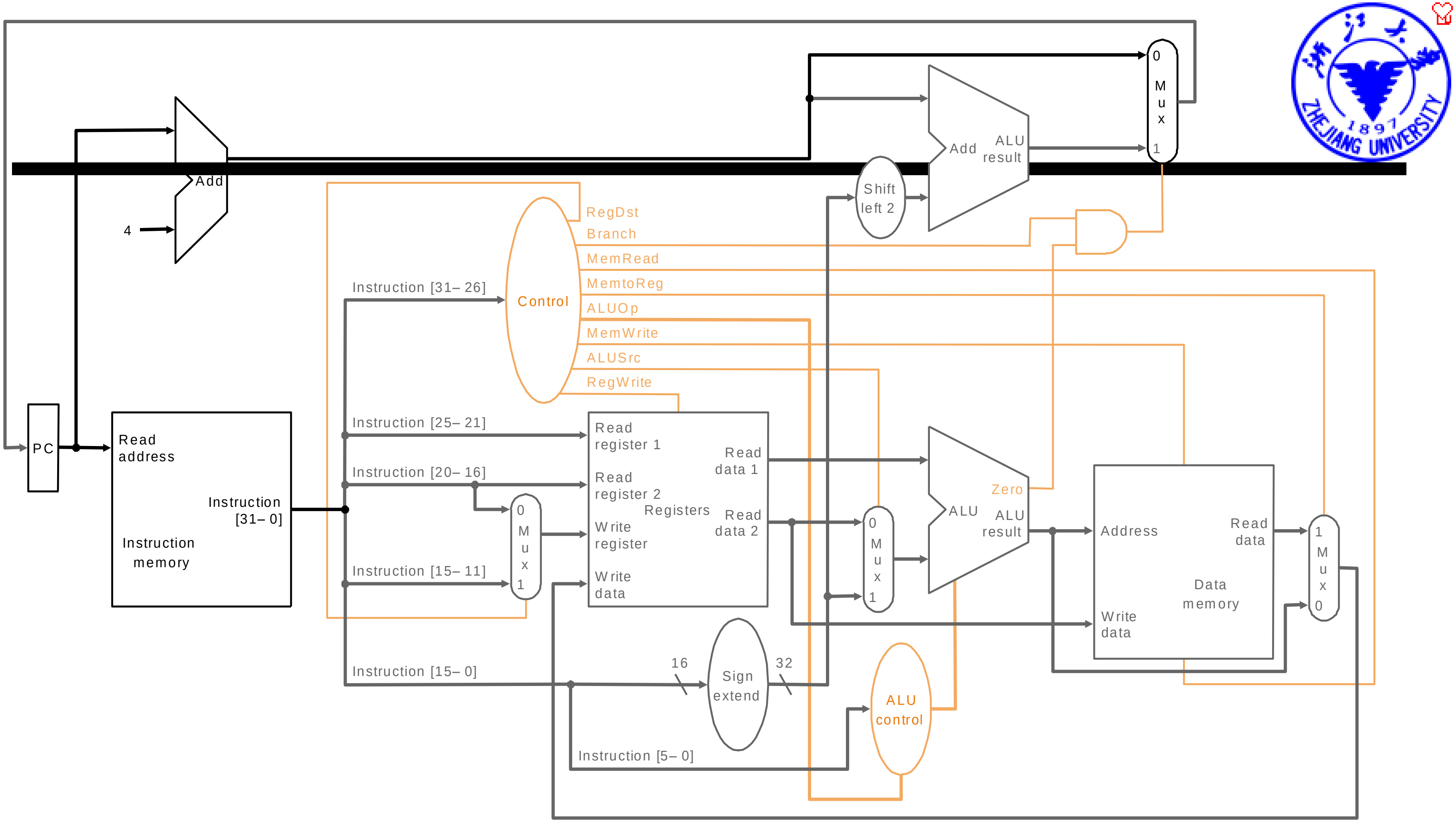
The ALU control



Designing the Main Control Unit

- Main Control Unit function
  - ☞ALU op (2)
  - ☞other control signals (p. 359)
    - ✧4 Mux
    - ✧3 R/W



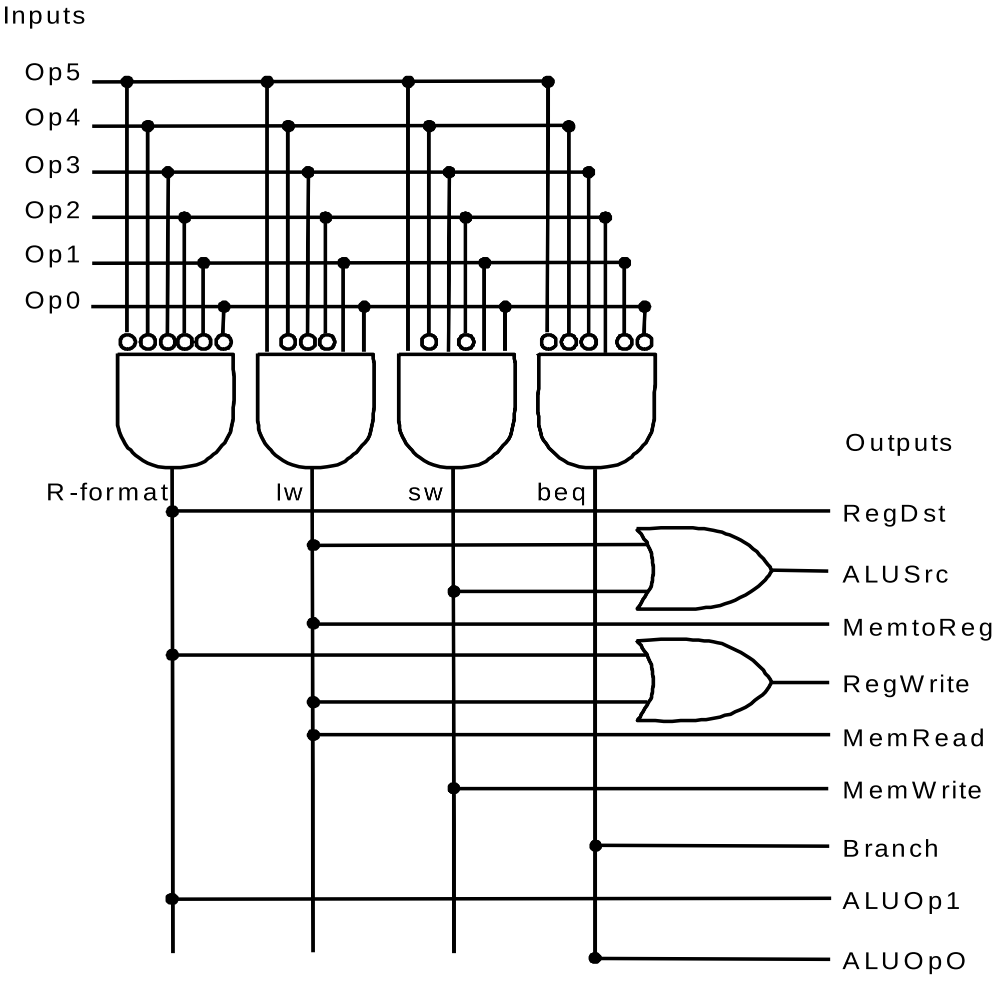


Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Control

- Simple combinational logic (truth tables)

R-type	0
lw	35
sw	43
beq	4

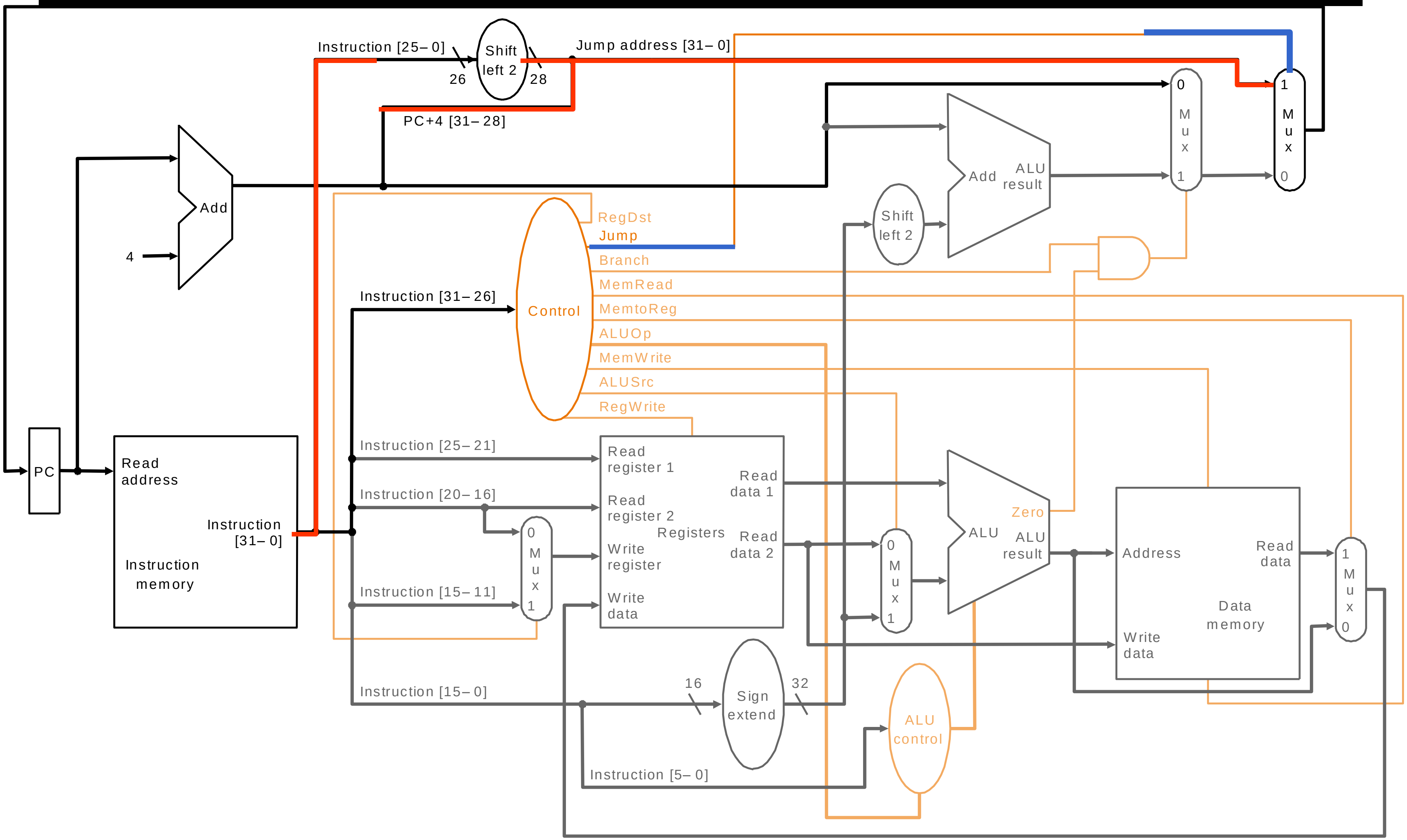


j instruction

- instruction format
  - j Label

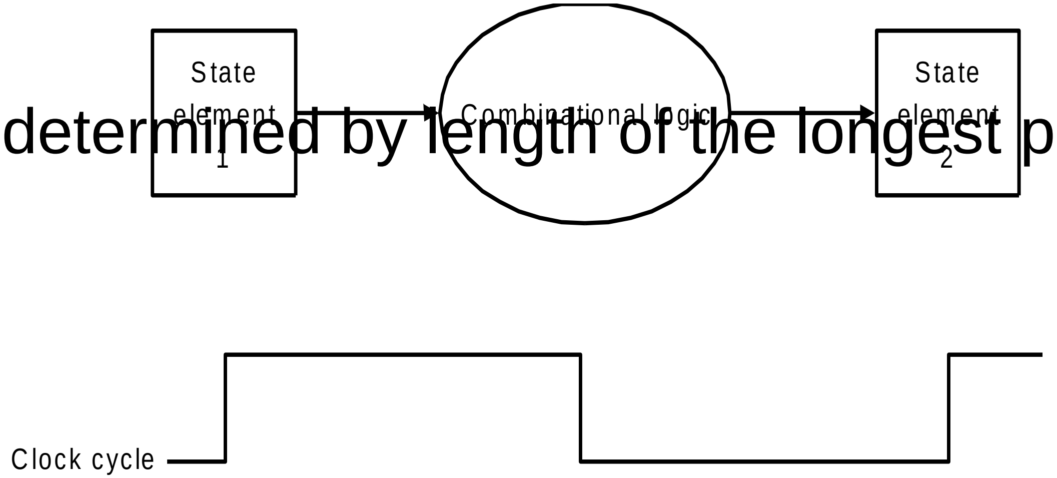
op=2	26 bits address

- Implementation
  - $pc = pc_{28\sim31} + 26\text{bits-address} * 4$


Instruction [25– 0]
26
Shift
left 2
28
Jump address [31– 0]
PC+4 [31– 28]
0
M
u
x
1
1
M
u
x
0
Add
ALU
result
Add
4
Shift
left 2
RegDst
Jump
Branch
MemRead
MemtoReg
ALUOp
MemWrite
ALUSrc
RegWrite
Instruction [31– 26]
Control
PC
Read
address
Instruction
[31– 0]
Instruction
memory
Instruction [25– 21]
Instruction [20– 16]
Instruction [15– 11]
0
M
u
x
1
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
0
M
u
x
1
Zero
ALU
ALU
result
Address
Read
data
Data
memory
Write
data
1
M
u
x
0
Instruction [15– 0]
16
Sign
extend
32
ALU
control
Instruction [5– 0]

Our Simple Control Structure

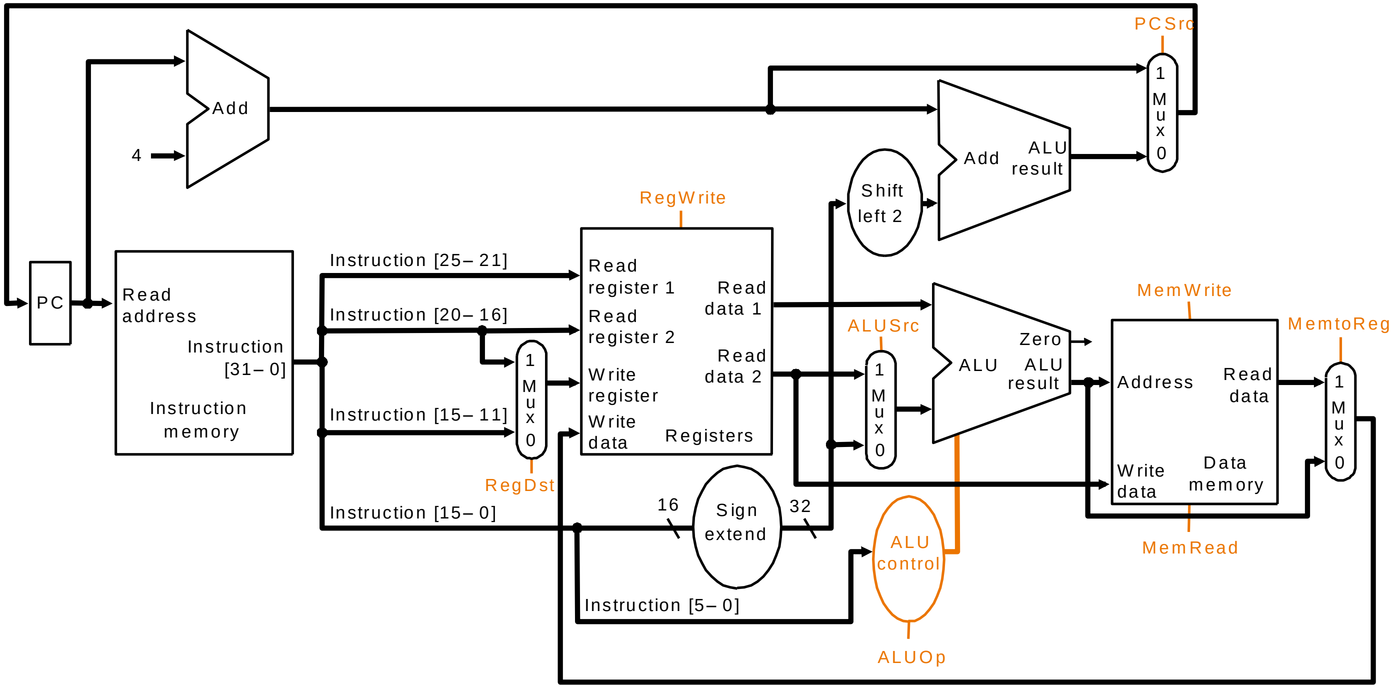
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer?right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

Single Cycle Implementation

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

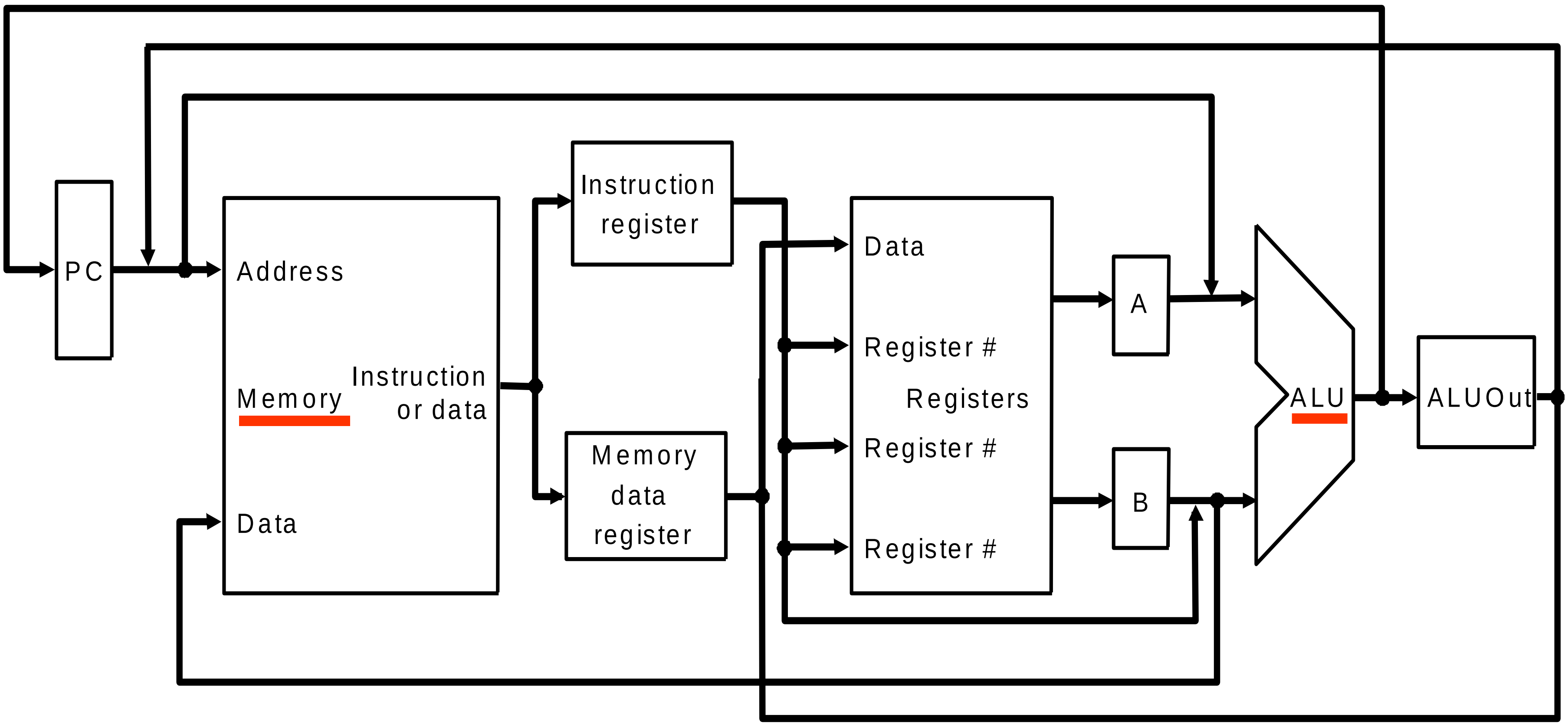


Where we are headed

- Single Cycle Problems:
  - ☞what if we had a more complicated instruction like floating point?
  - ☞wasteful of area
- One Solution:
  - ☞use a smaller cycle time
  - ☞have different instructions take different numbers of cycles

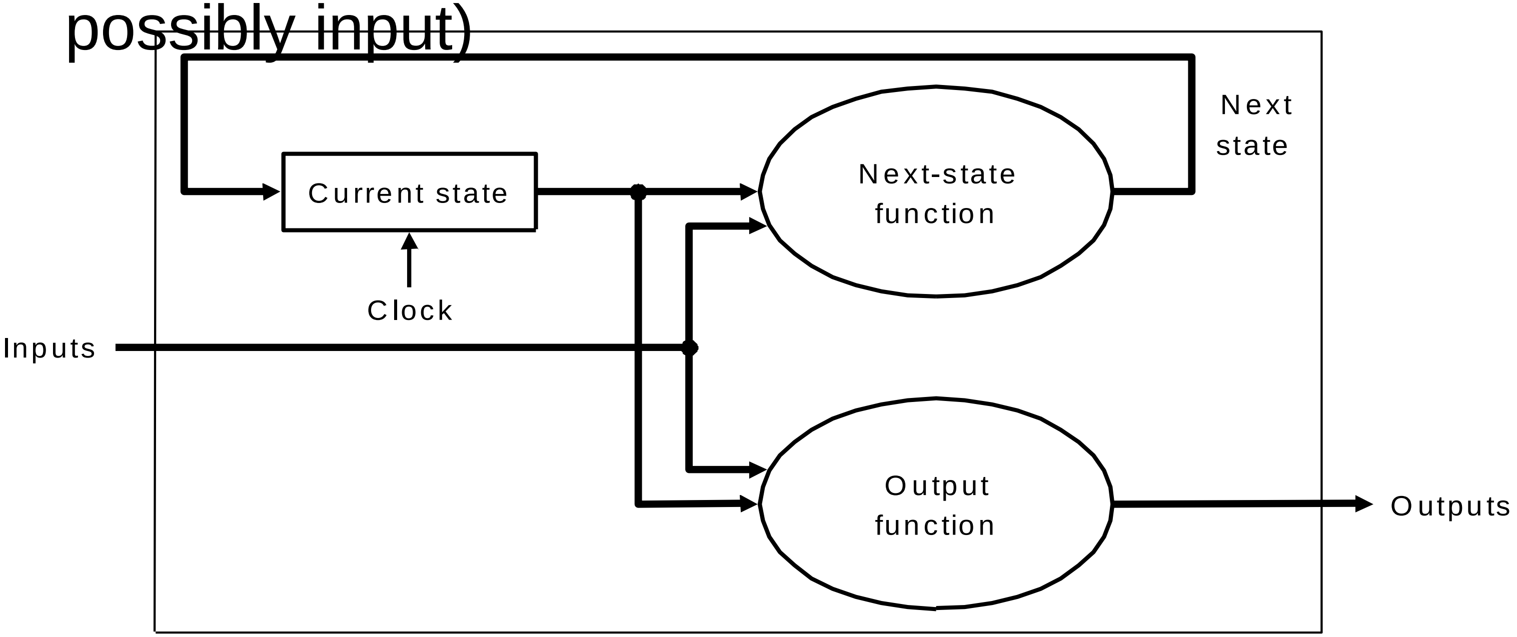
Multicycle Approach

- We will be reusing functional units
  - ☞ALU used to compute address and to increment PC
  - ☞Memory used for instruction and data
- Our control signals will not be determined soley by instruction
  - ☞e.g., what should the ALU do for a subtract instruction?
- We will use a finite state machine for control


PC
Address
Memory
Instruction
or data
Data
Instruction
register
Memory
data
register
Data
Register #
Registers
Register #
Register #
A
B
ALU
ALUOut

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)

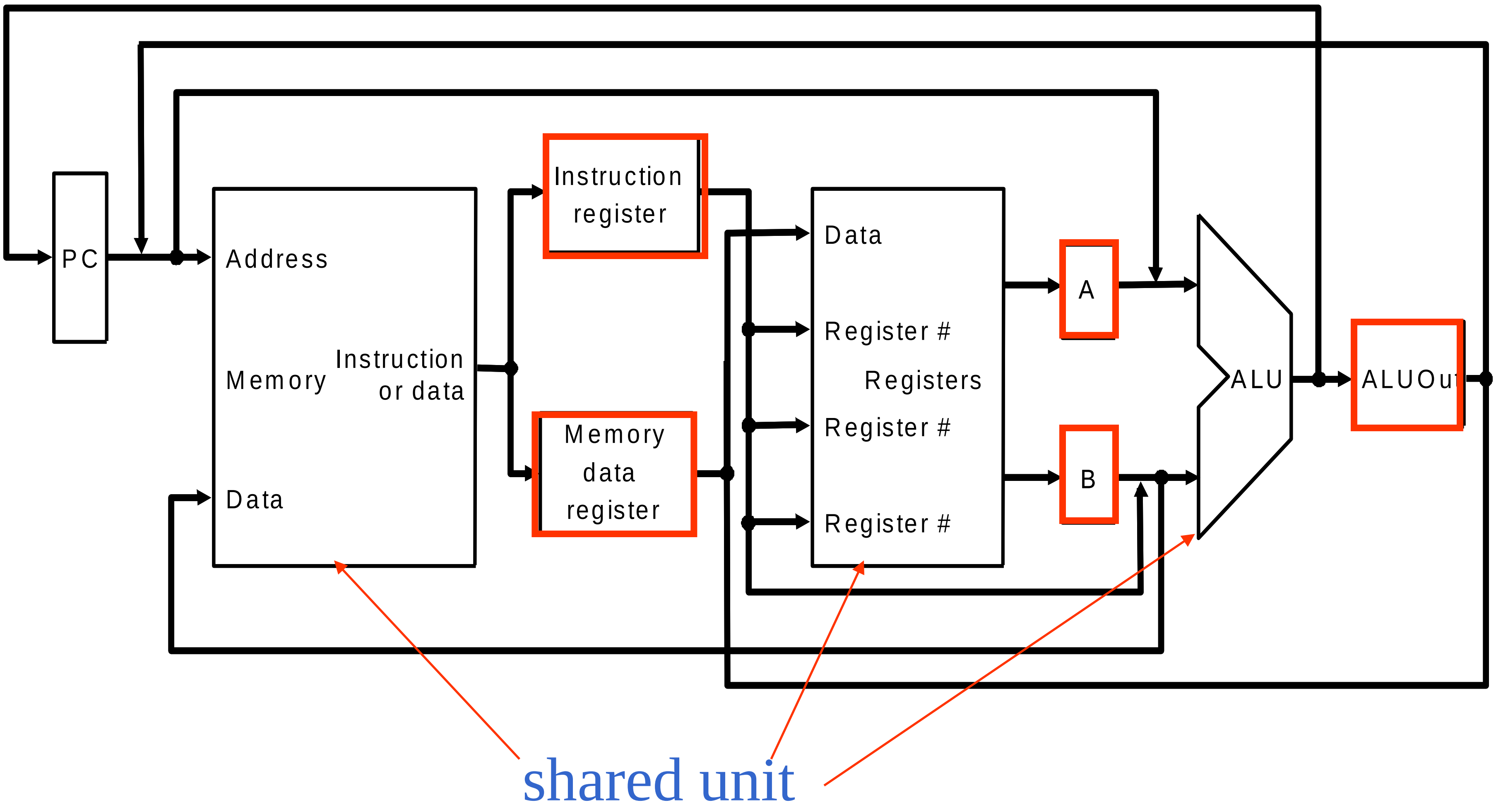


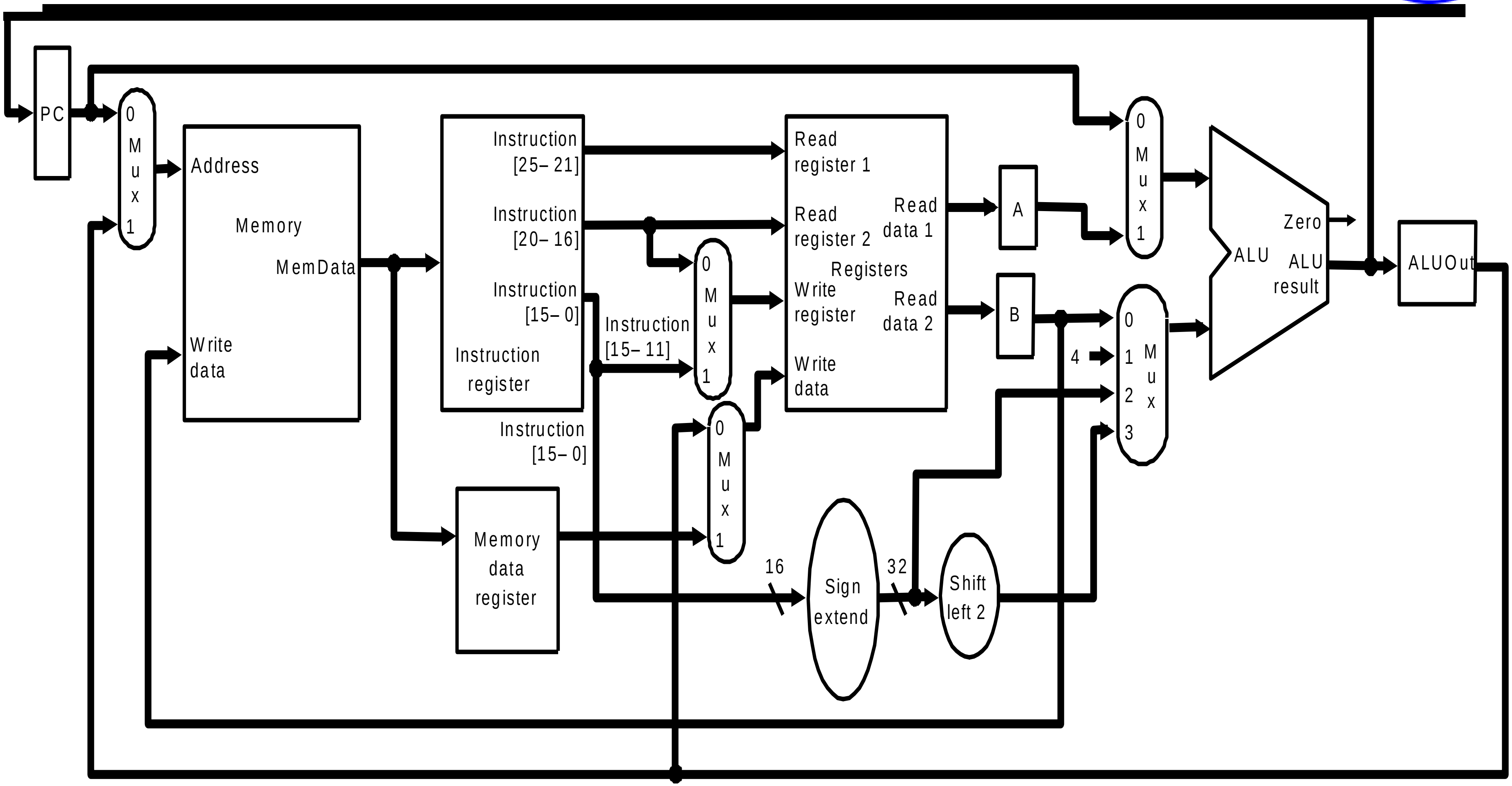
Multicycle Approach

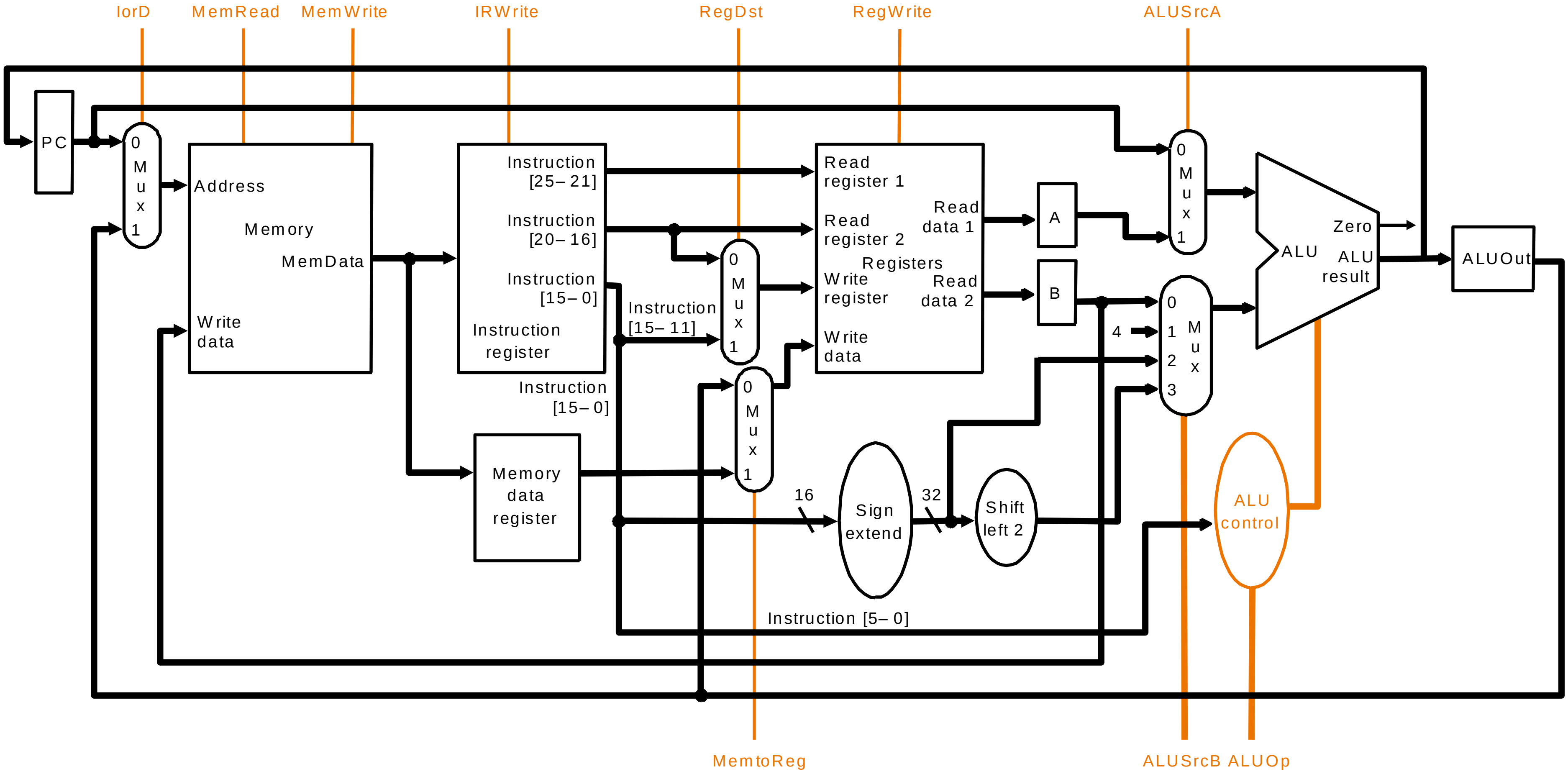
- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional internal registers

Multicycle Approach

- a Multicycle datapath




PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[25– 21]
Instruction
[20– 16]
Instruction
[15– 0]
Instruction
register
Instruction
[15– 0]
Instruction
[15– 11]
0
M
u
x
1
0
M
u
x
1
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
16
Sign
extend
32
Shift
left 2
4
0
M
u
x
1
0
1
2
3
M
u
x
ALU
Zero
ALU
result
ALUOut

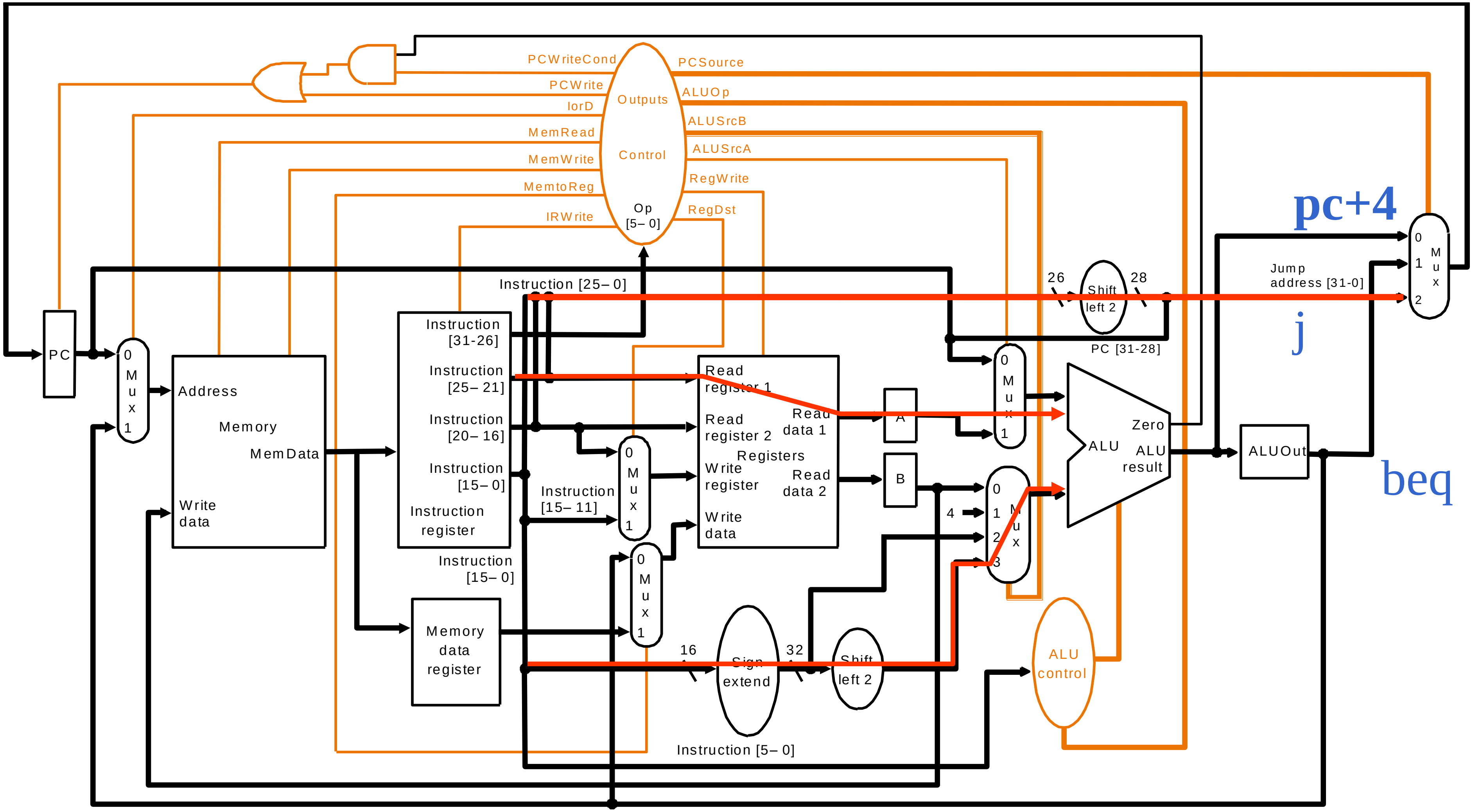

IorD
MemRead
MemWrite
IRWrite
RegDst
RegWrite
ALUSrcA
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[25– 21]
Instruction
[20– 16]
Instruction
[15– 0]
Instruction
register
Instruction
[15– 0]
Memory
data
register
Instruction
[15– 11]
0
M
u
x
1
0
M
u
x
1
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
0
M
u
x
1
4
0
1
2
3
M
u
x
Zero
ALU
ALU
result
ALUOut
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5– 0]
MemtoReg
ALUSrcB
ALUOp

- PC 的改变方式
  - ☞seq: $pc = pc + 4$
  - ☞beq： $pc = pc + offset * 4$
  - ☞j ： $pc = pc_{31-28} + IR_{25-0} << 2$

seq: pc = pc + 4

beq： pc = pc + offset * 4

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



Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly 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:

Step 4 (R-type or memory-access)

- Loads and stores access memory

```
    MDR = Memory[ALUOut];
or
    Memory[ALUOut] = B;
```

- R-type instructions finish

```
    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[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?

```
        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?

- In what cycle does the actual addition of `$t2` and `$t3` takes place?

Implementing the Control

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

- Use the information were accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming

- Implementation can be derived from specification