Principles of Computer Systems

Hardware/Software interface

楼学庆

浙江大学计算机学院

http://10.214.47.99/

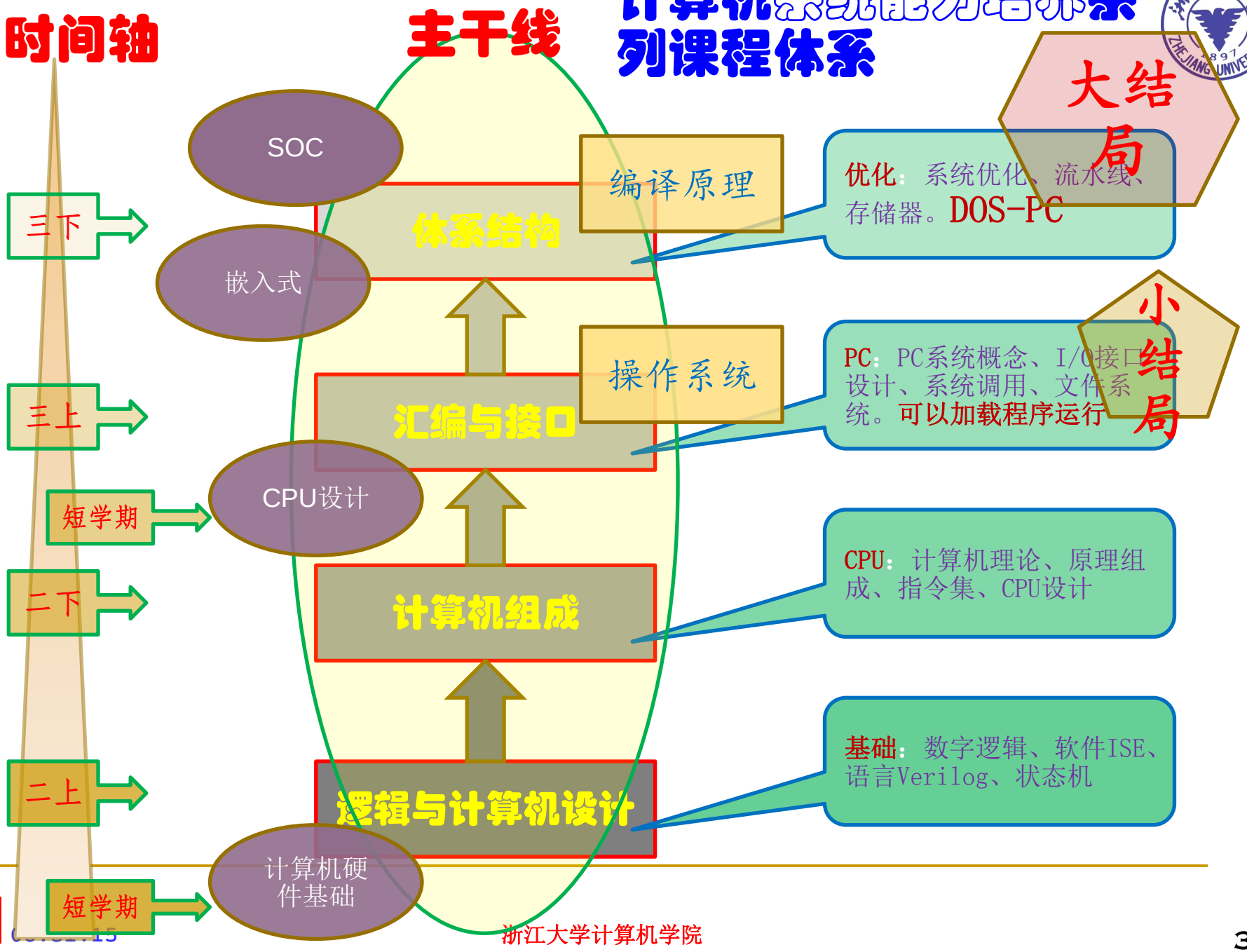
Email:hzlou@163.com

玉泉校区曹光彪东楼507室

联系方式

- 网站：
  - http://10.214.47.99
- 邮箱：
  - hzlou@163.com(不收作业)

计算机系统能力培养系列课程体系



Course outline

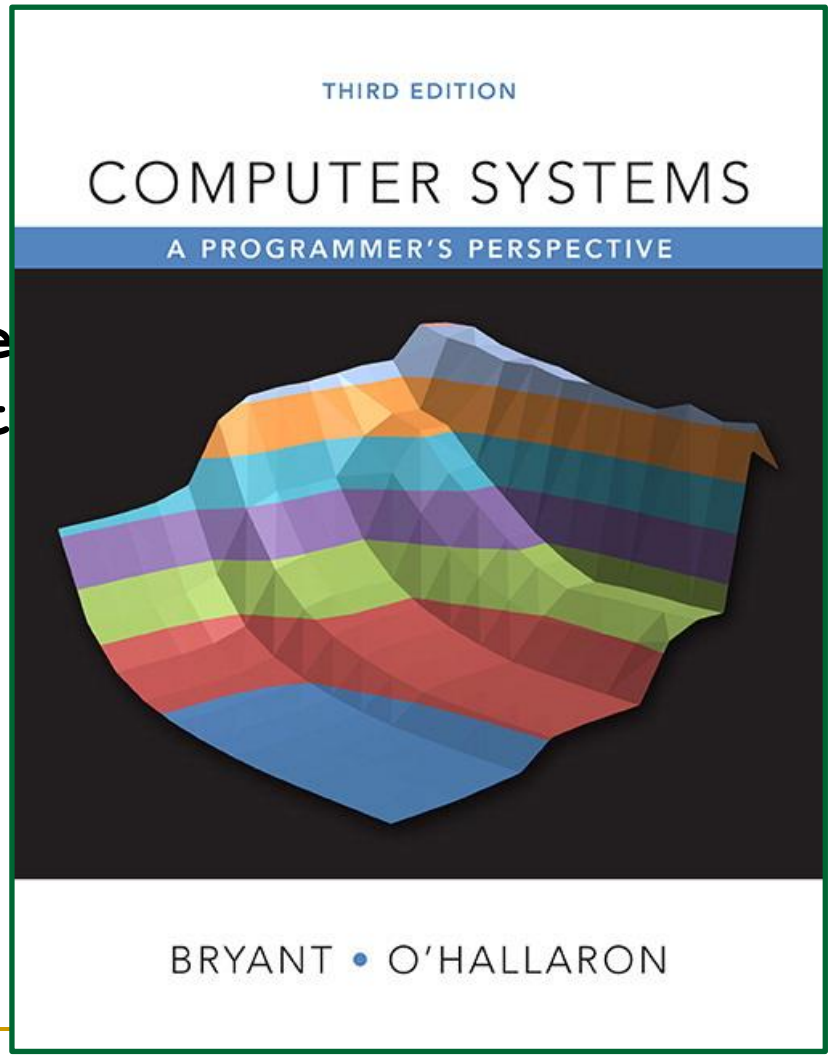
- **Name:**

  Computer Systems

- **Students:**

  Undergraduate stude
  some others department

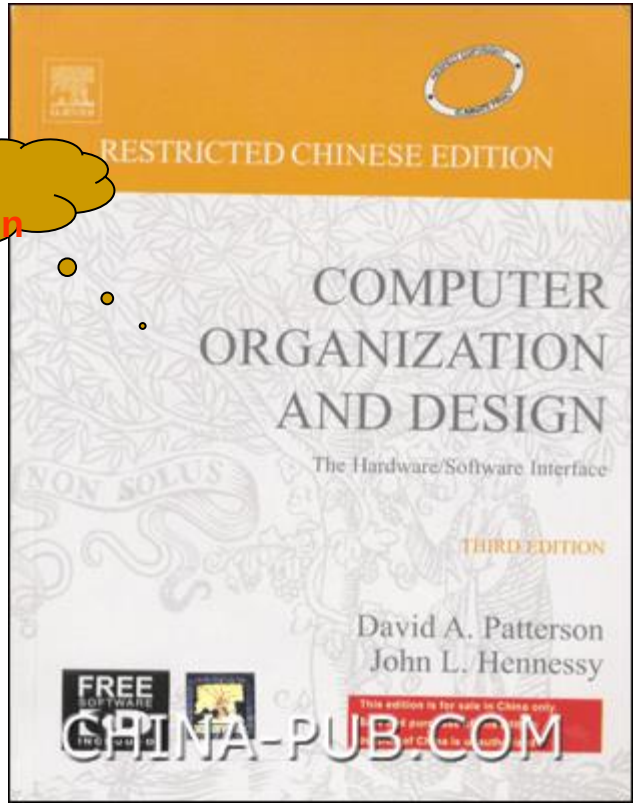
  - Score : 4.5
  - Hours/week: 3.5-2
  - Total: 88 hours



Textbook

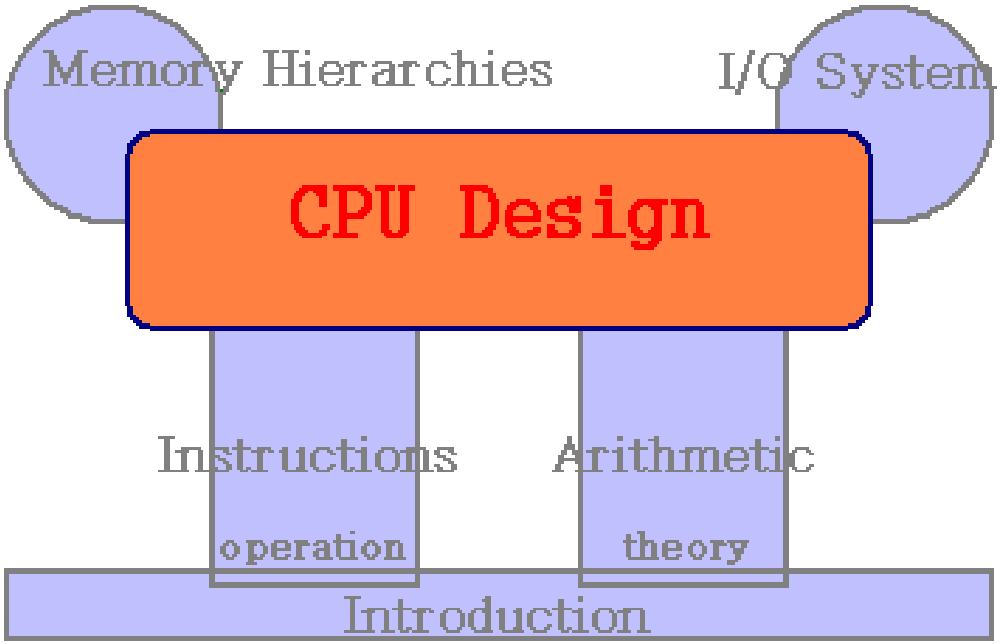
2nd Edition

3rd Edition



CPU design

- Topics: Datapath & Control



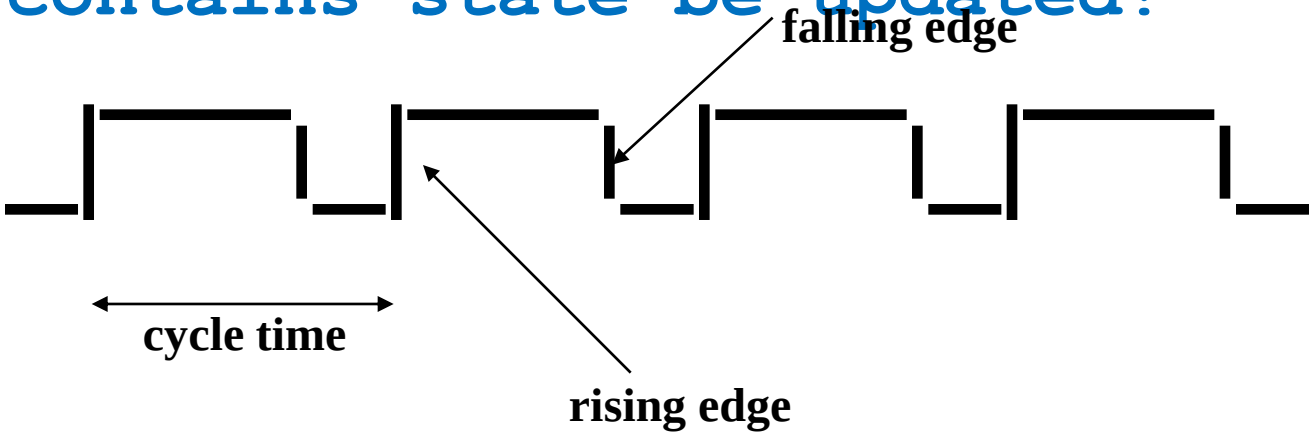

CPU
设计

Processor

- 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

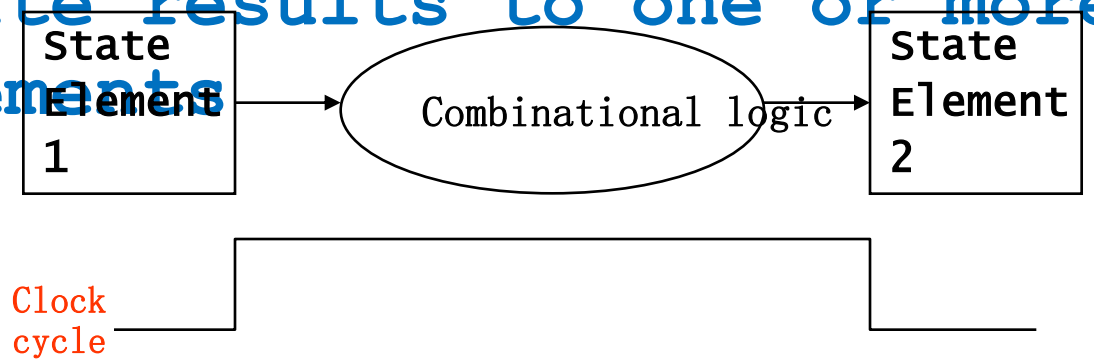
State Elements

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



Our Implementation

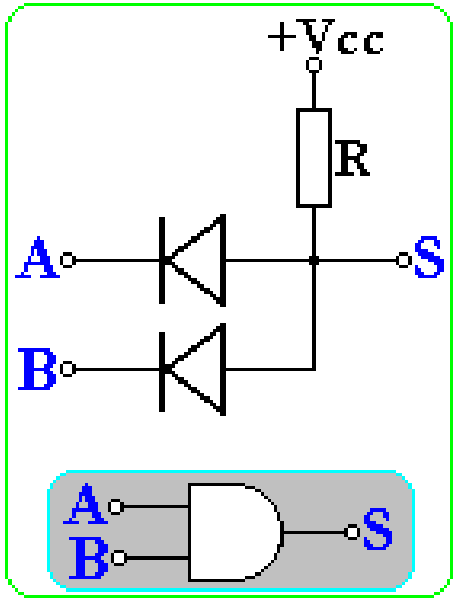
- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements

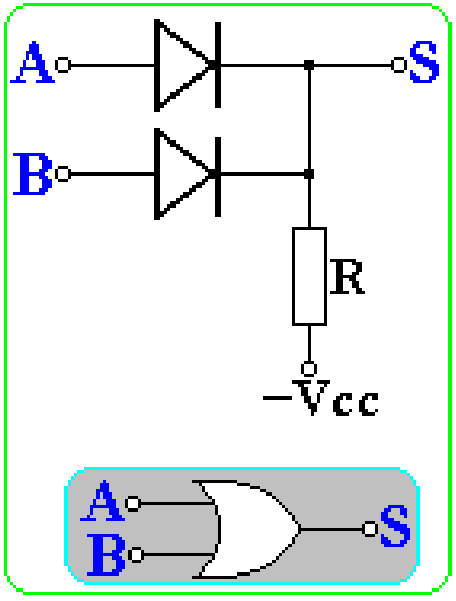


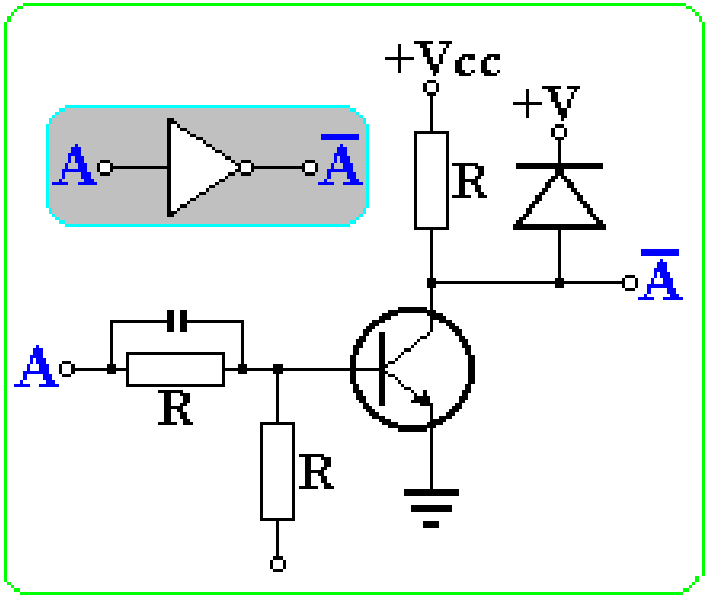
Basic elements

- 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



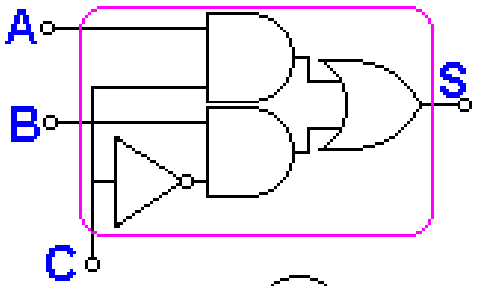


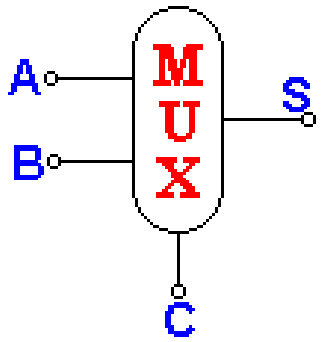


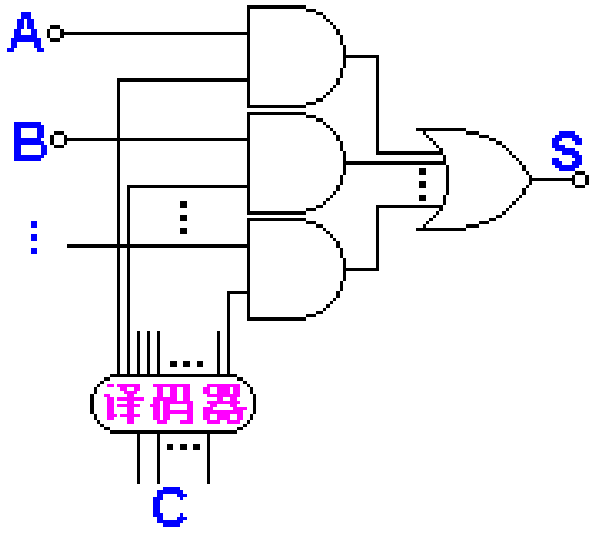
MUX

- 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







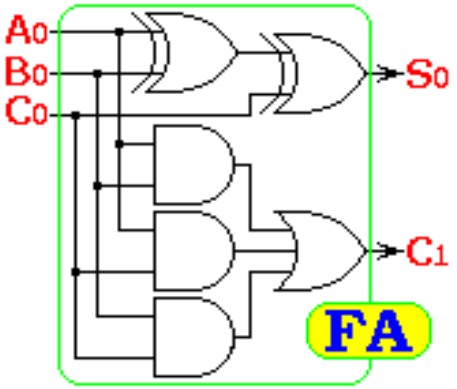
Adder

- **Half adder:**

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

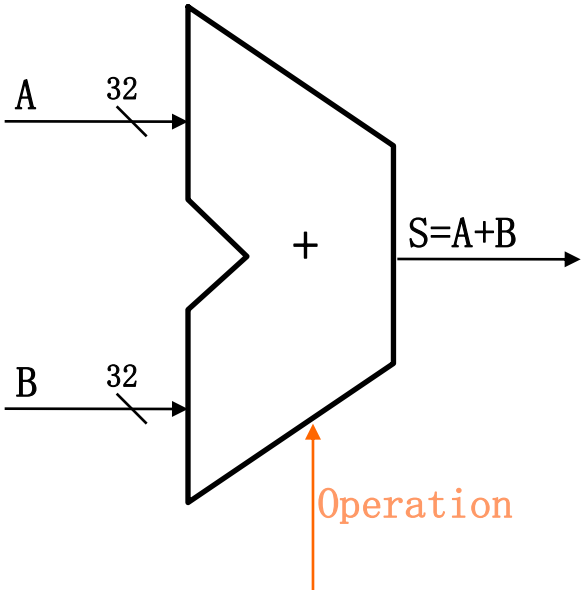
- **Full adder: S=A+B**
  - **Input: A, B, C0. Output: Sum,**
  - **$S = A \oplus B \oplus C$**
  - **$C_{+1} = A \cdot B + B \cdot C + C \cdot A$**

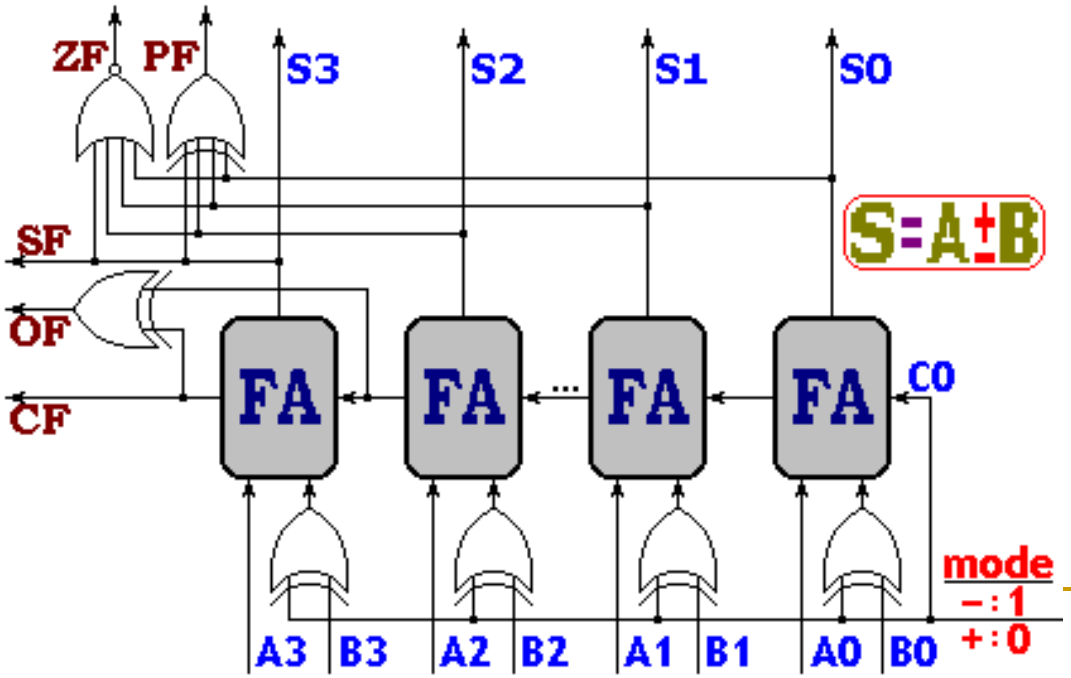
A	0	0	1	1	0	0	1	1
B	0	1	0	1	0	1	0	1
C	0	0	0	0	1	1	1	1
S	0	1	1	0	1	0	0	1
C_{+1}	0	0	0	1	0	1	1	1



串行进位加法器

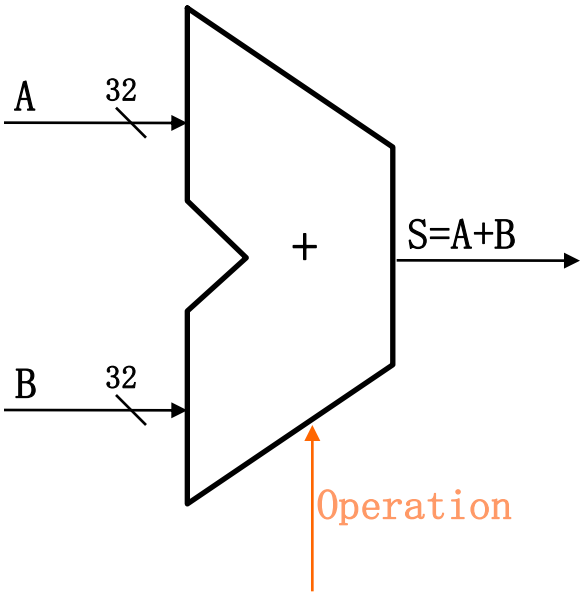
- 4位串行进位加法器：S=A+B
  - 各位串行进行。
  - 同时根据结果产生各种标记状态。





Adder

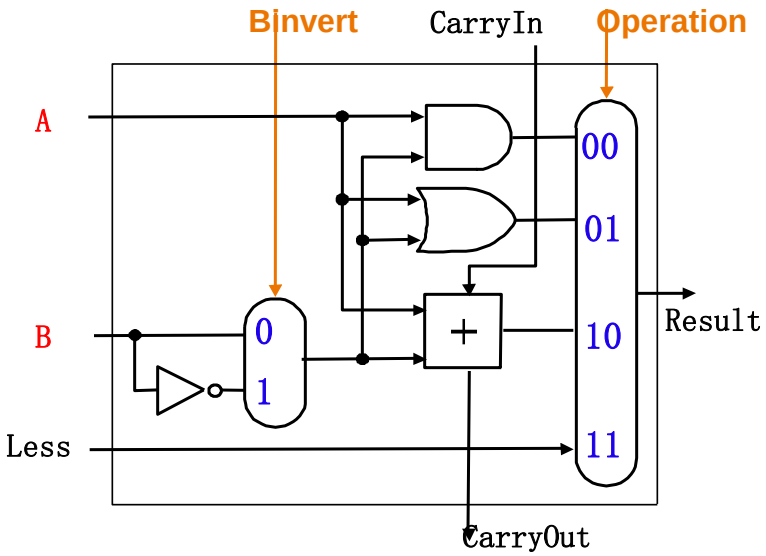
- 32-bit Adder

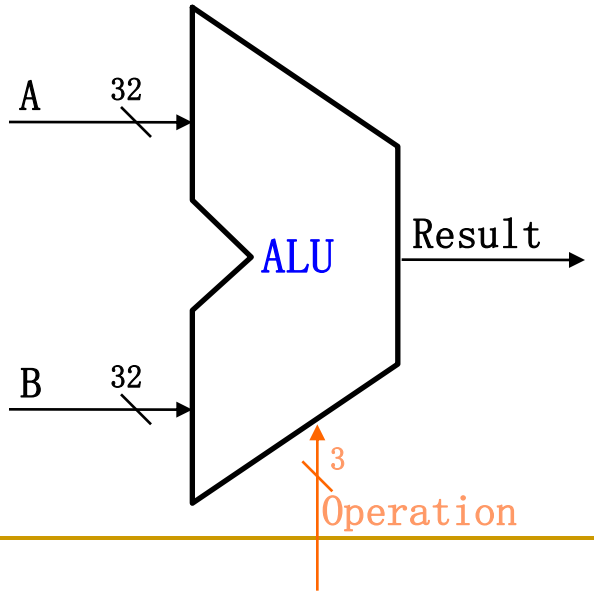


ALU

- 算术逻辑运算器ALU：即运算
  - 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

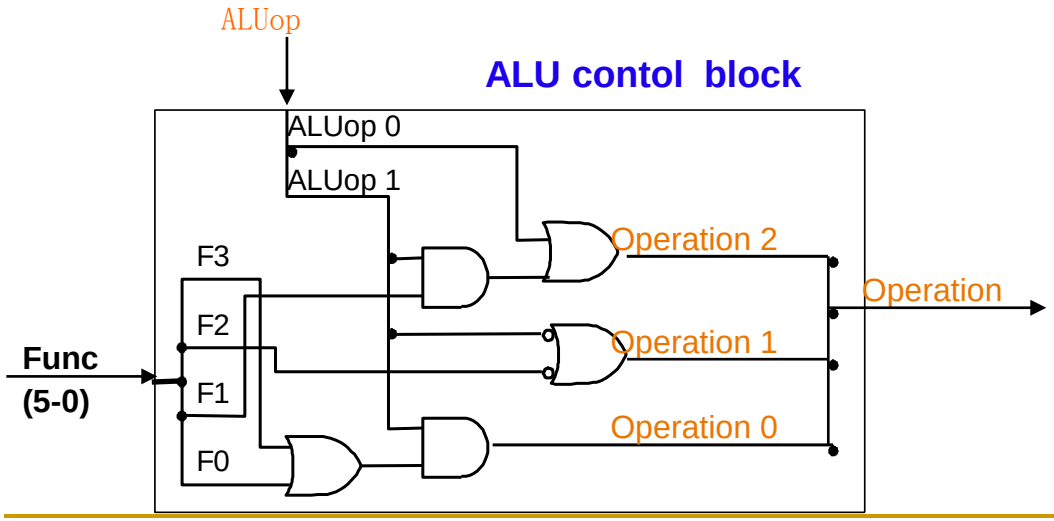


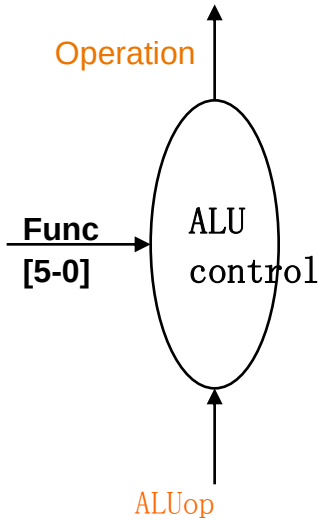


ALU Control

- **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





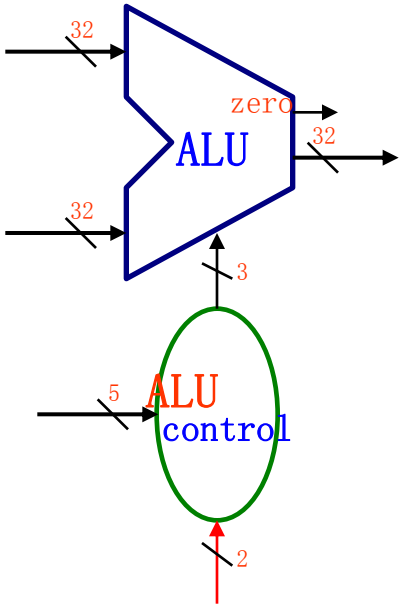
ALU Control

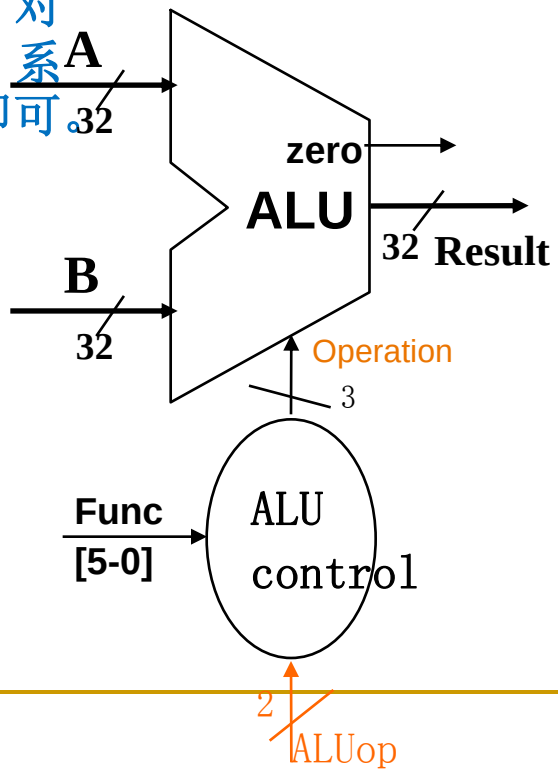
- ALUop与Func联合对ALU的控制

ALUOp		Funct field						Operation	ALU Function
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
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

ALU Control

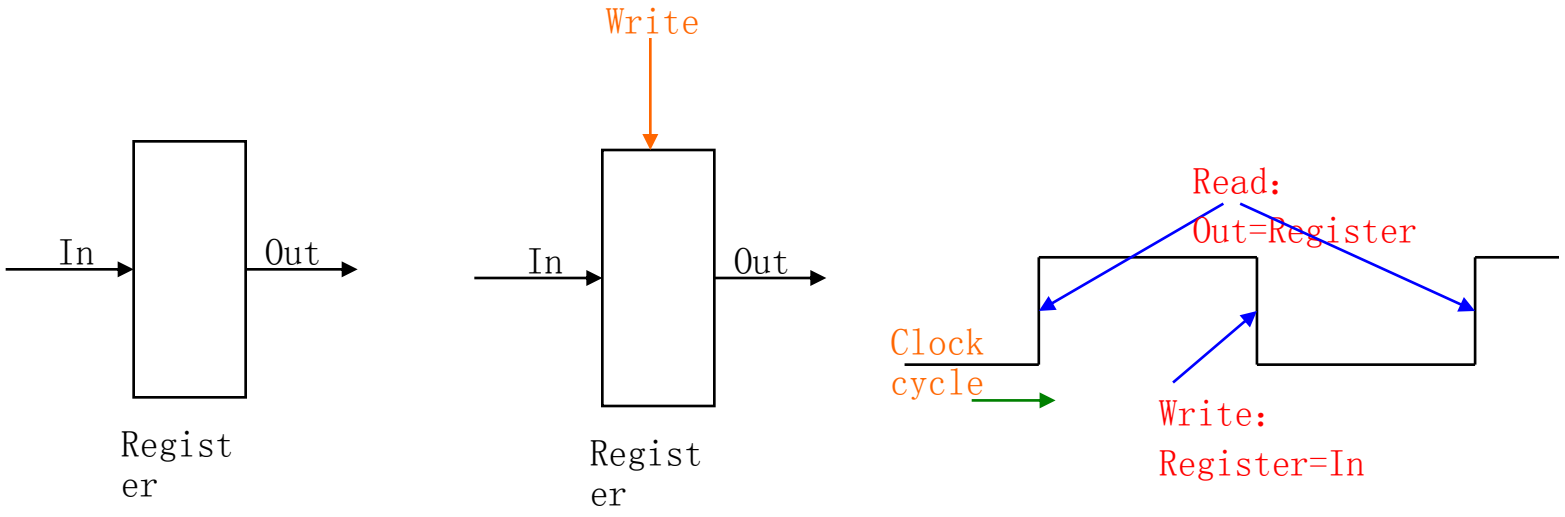
- **ALU控制：**
  - ALU由ALUcontrol控制：ALUcontrol由ALUop和指令的低6位(5-0)联合产生ALU控制码。
  - 这样的好处在于分级控制，对于用的最多的加、减操作，系统只需给出两位的ALUop即可。





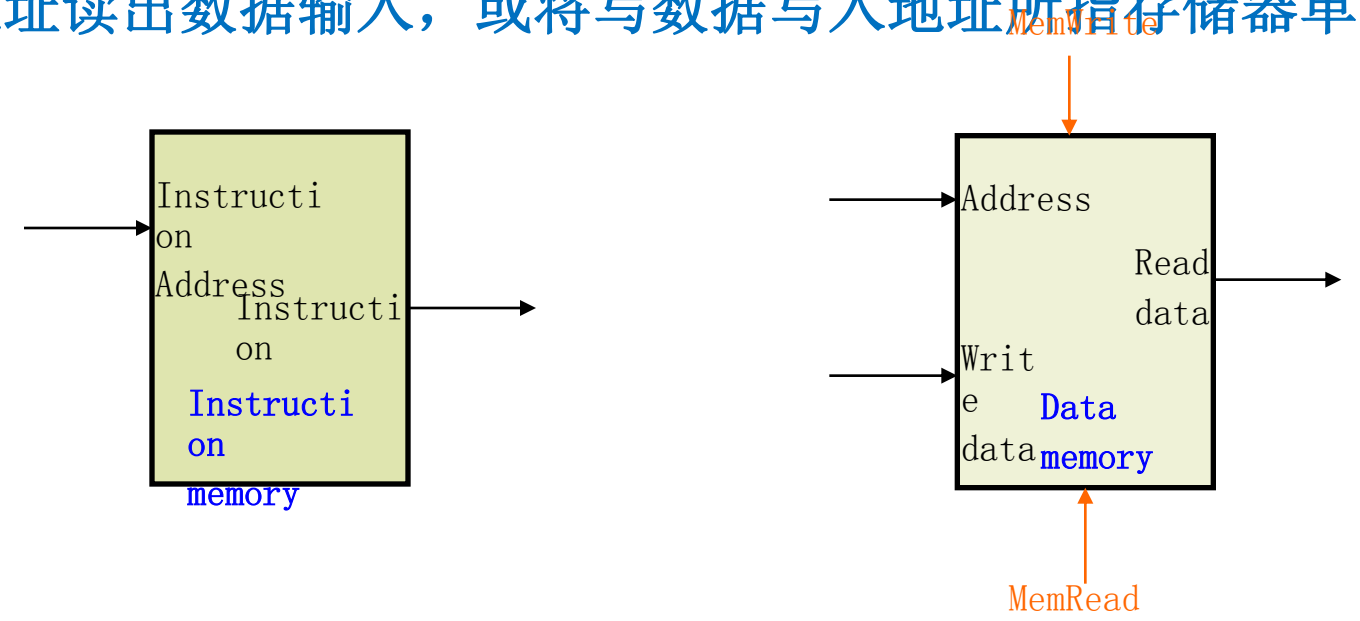
REGISTER

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



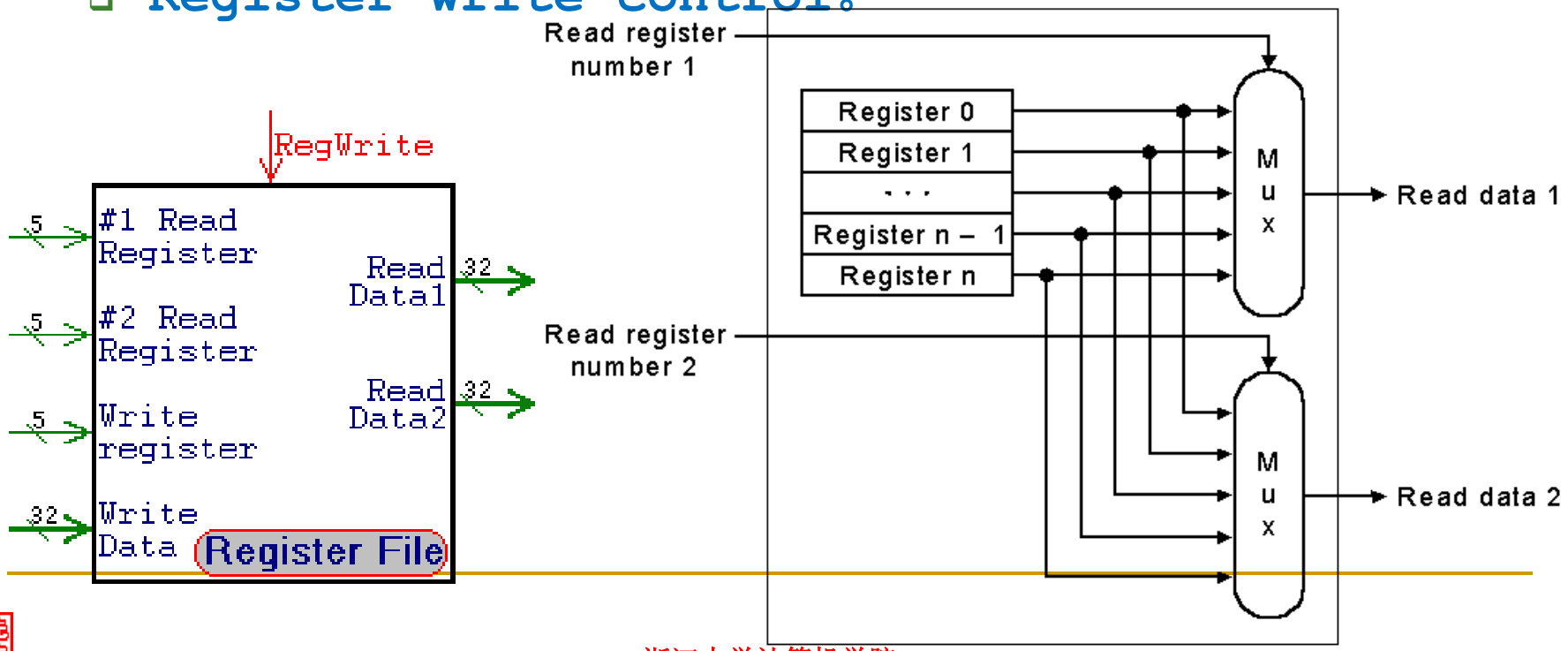
Memory

- 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。



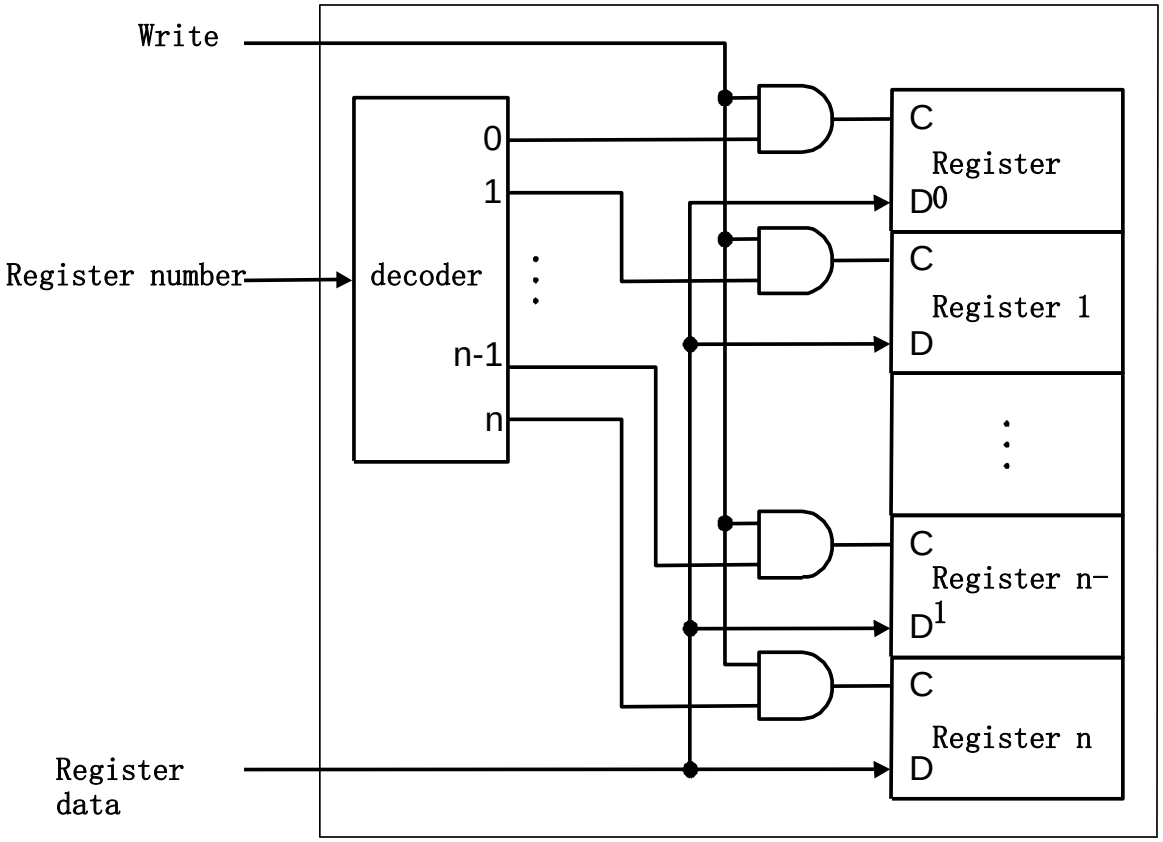
Register File

- **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



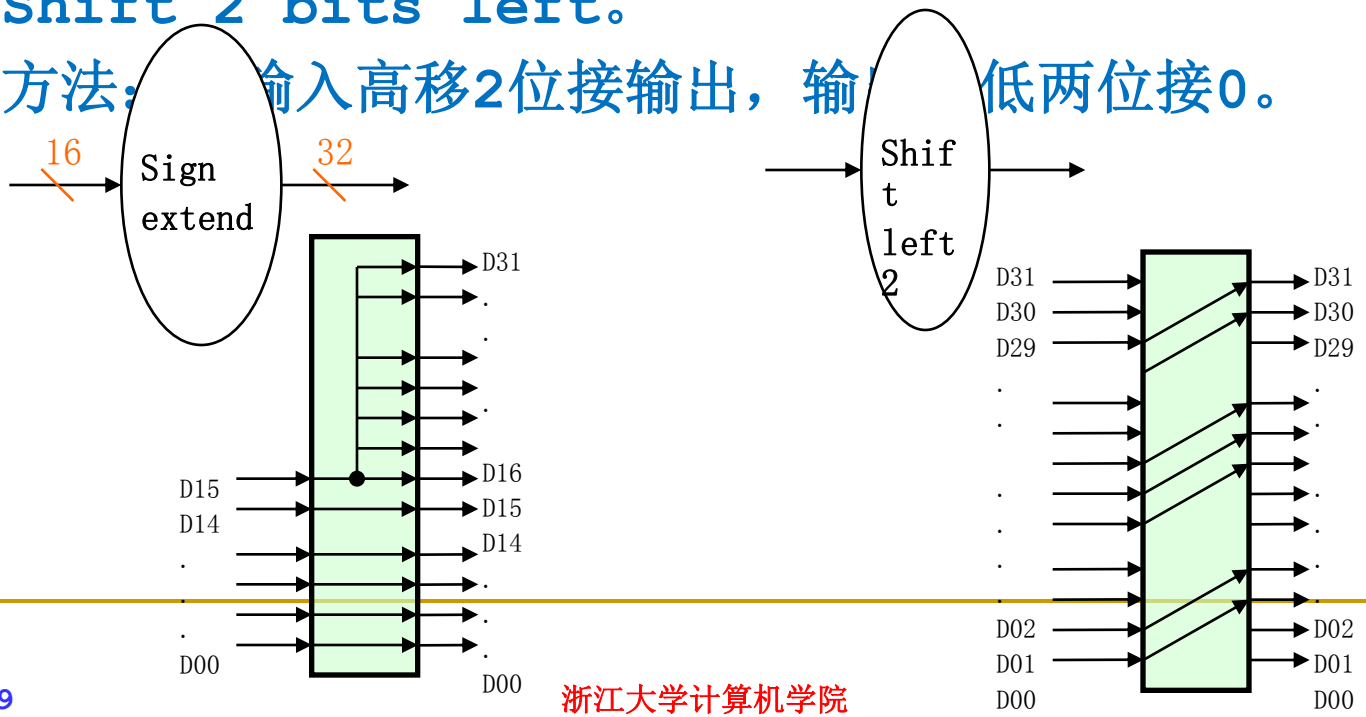
32个Register的功用

- 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

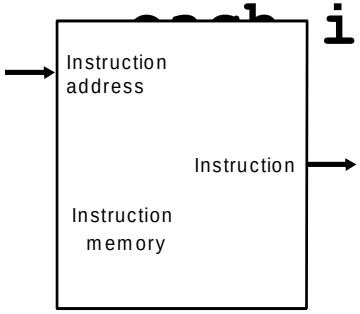
The other elements

- **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：输入高移2位接输出，输[illegible]低两位接0。

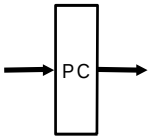


Simple Implementation

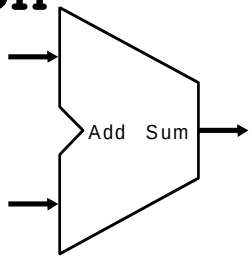
- Include the functional units we need for each instruction



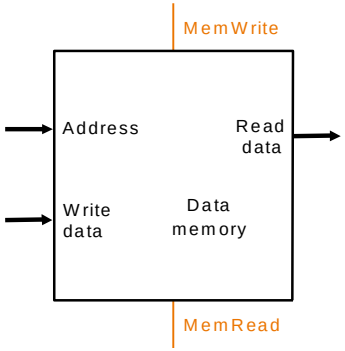
a. Instruction memory



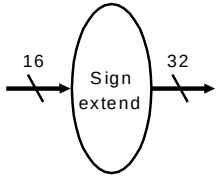
b. Program counter



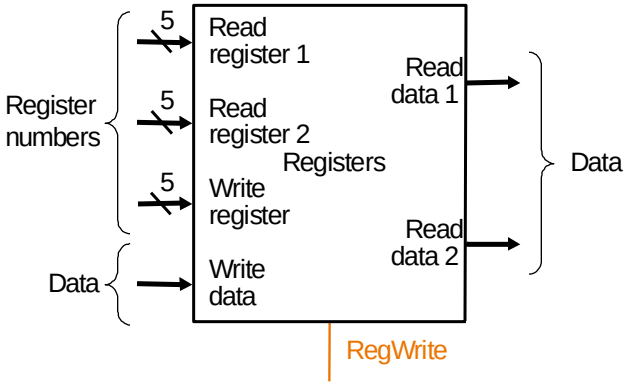
c. Adder



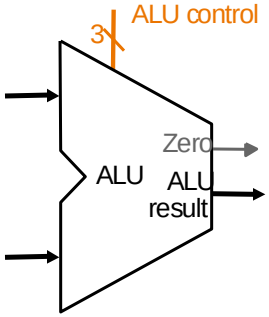
a. Data memory unit



b. Sign-extension unit



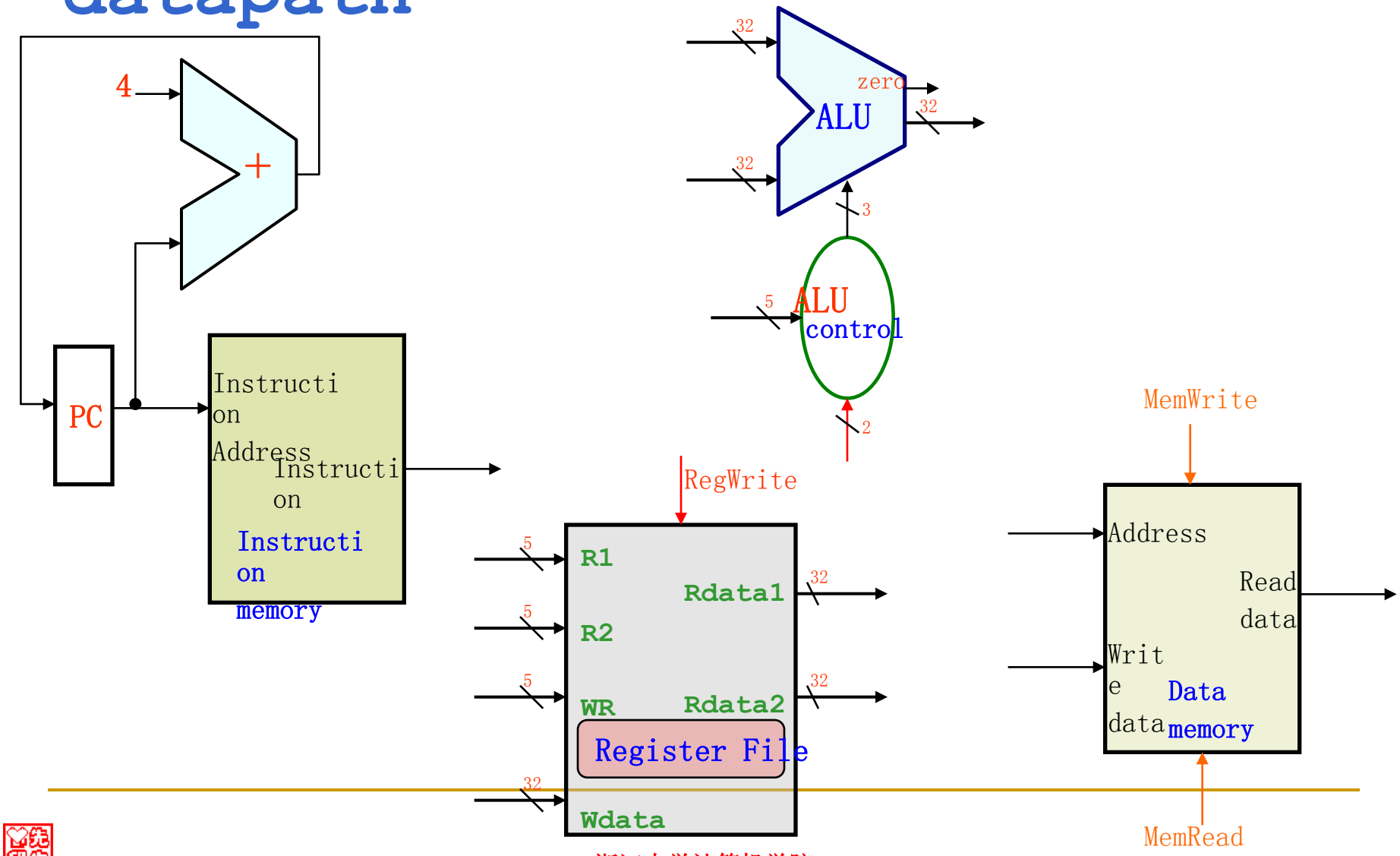
a. Registers



b. ALU

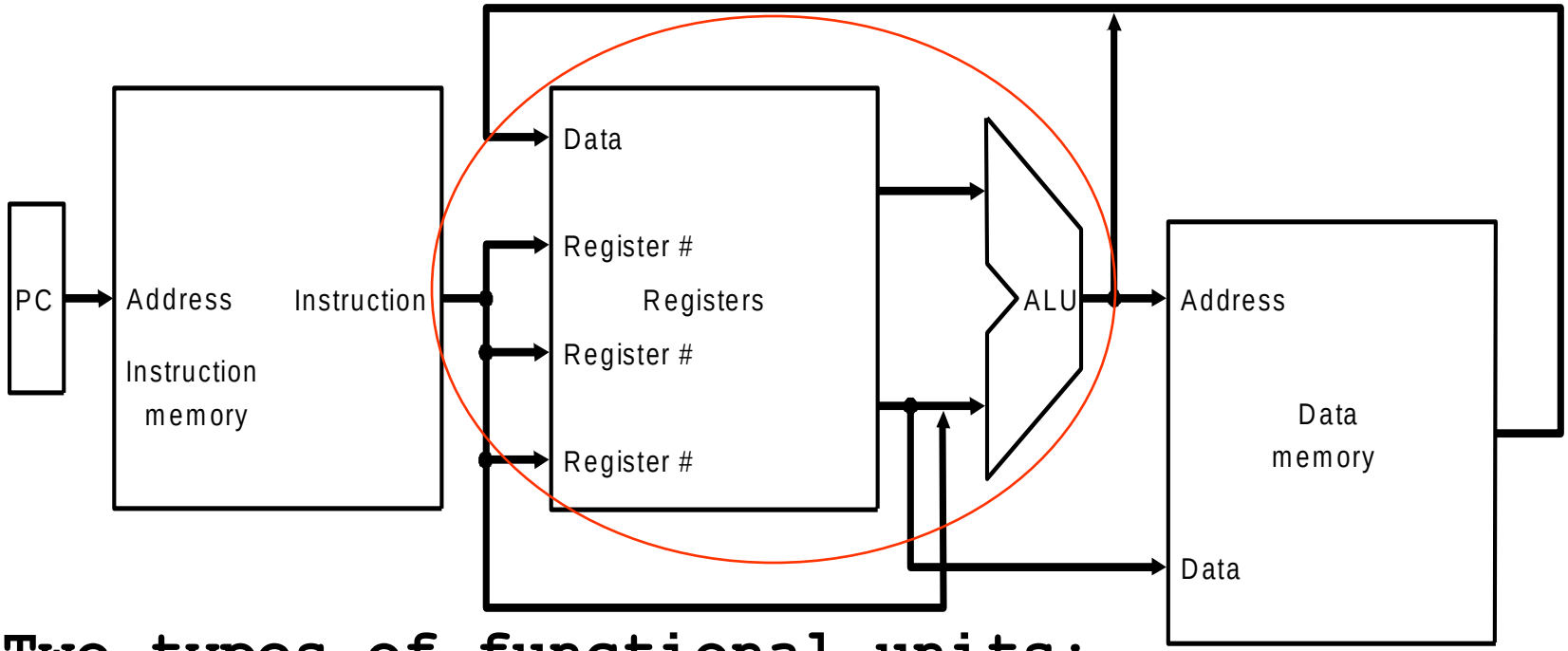
Why do we need this stuff?

Section 2 Building a datapath



More Details

- Abstract / Simplified View:

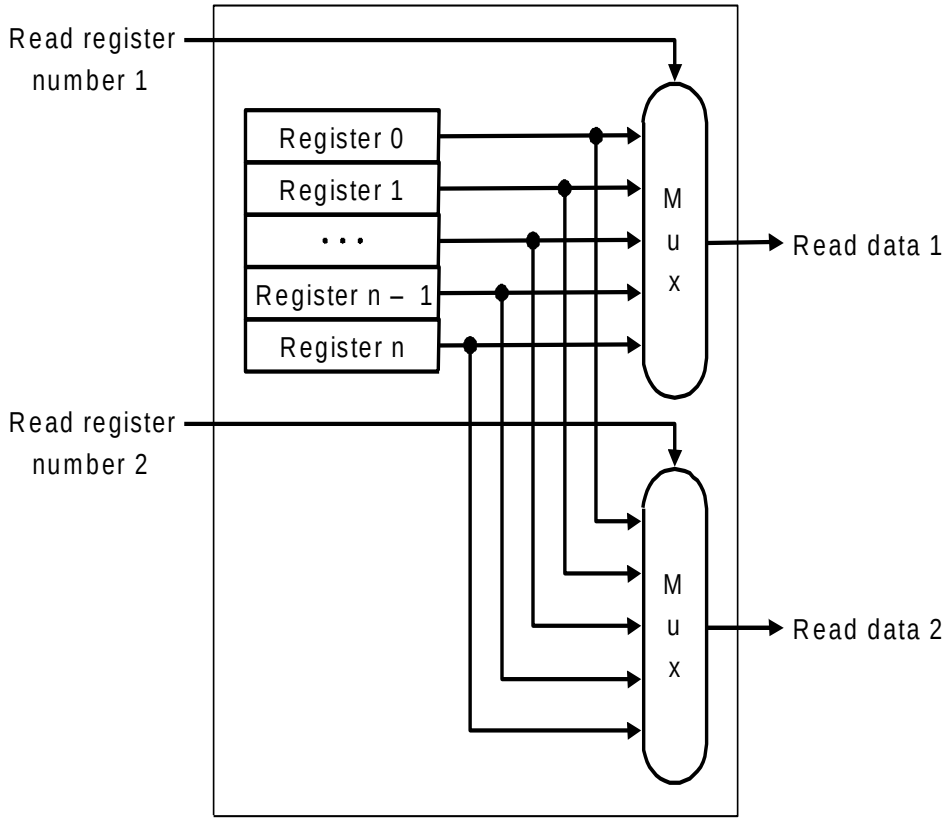


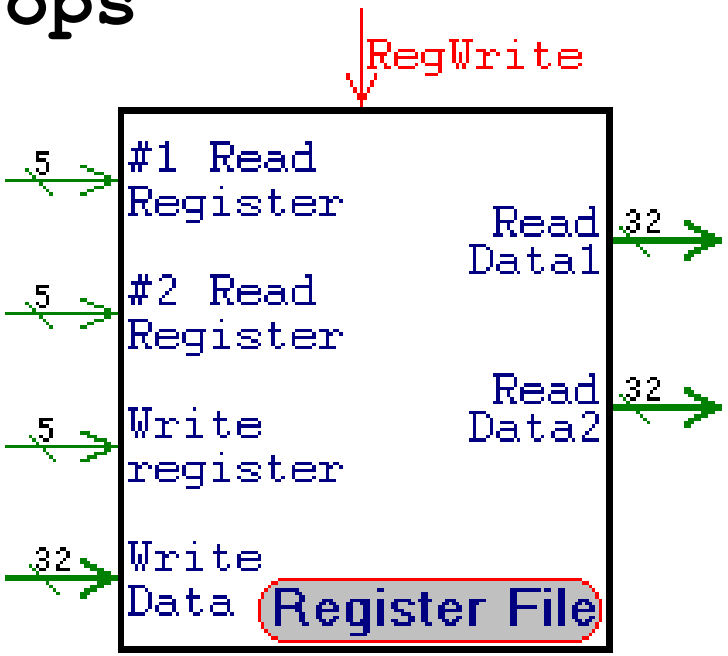
Two types of functional units:

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

Register File

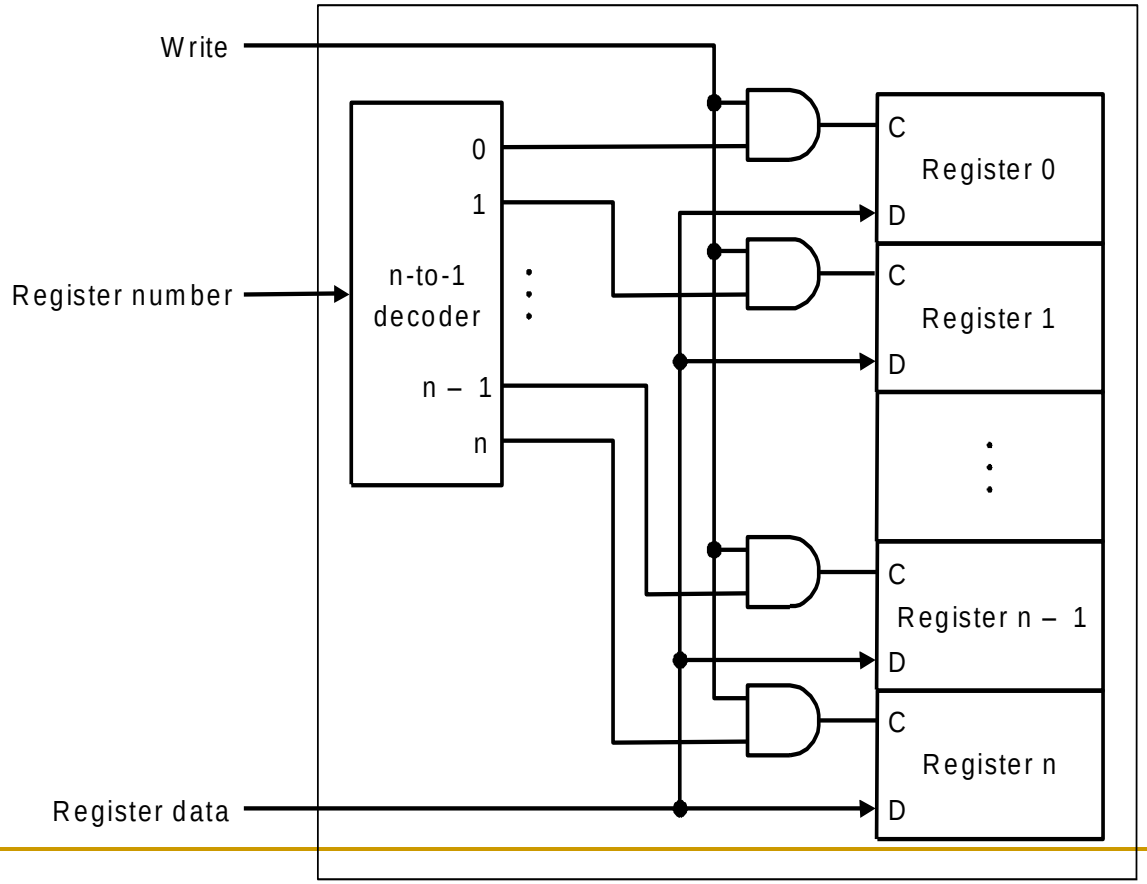
- Built using D flip-flops





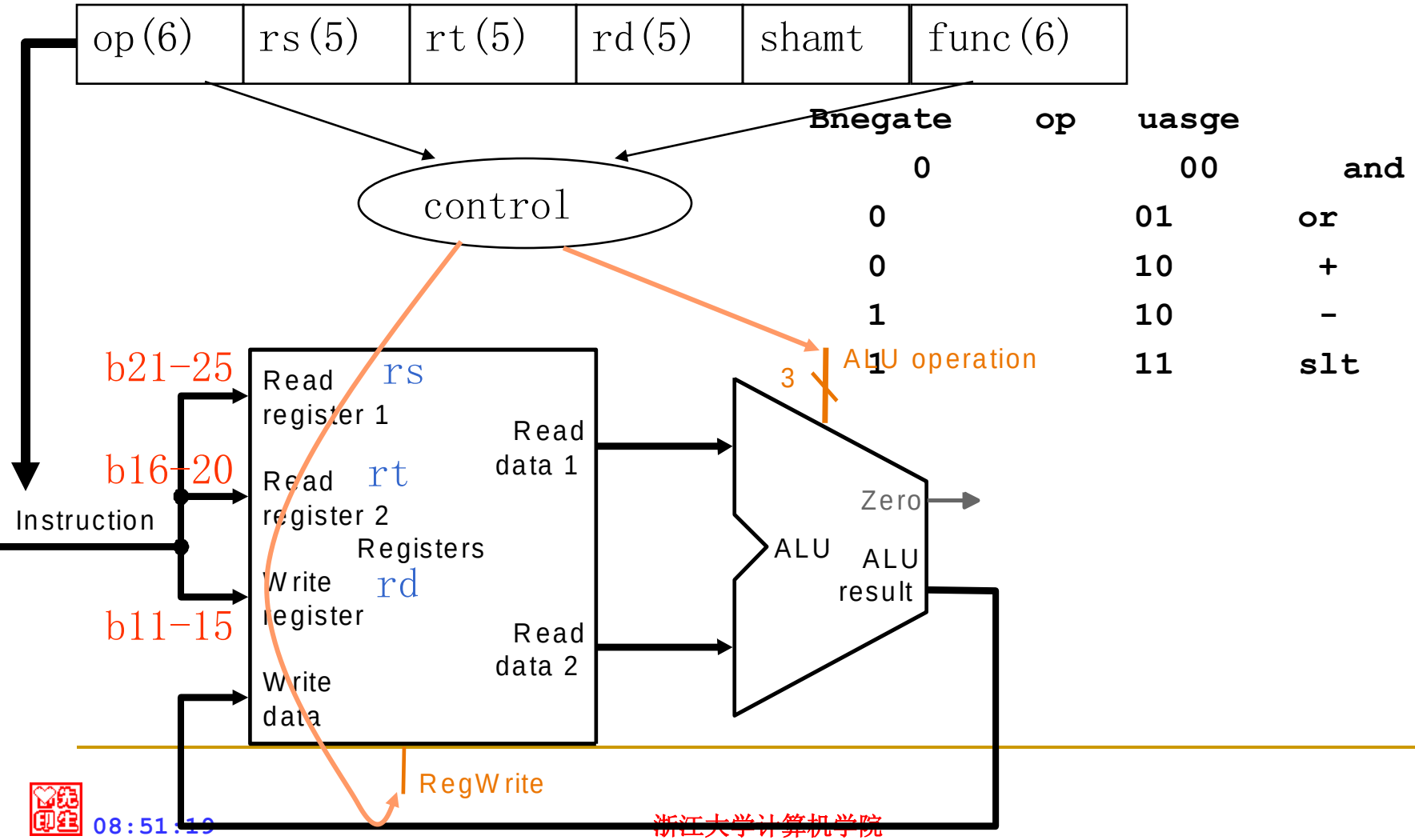
Register File

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

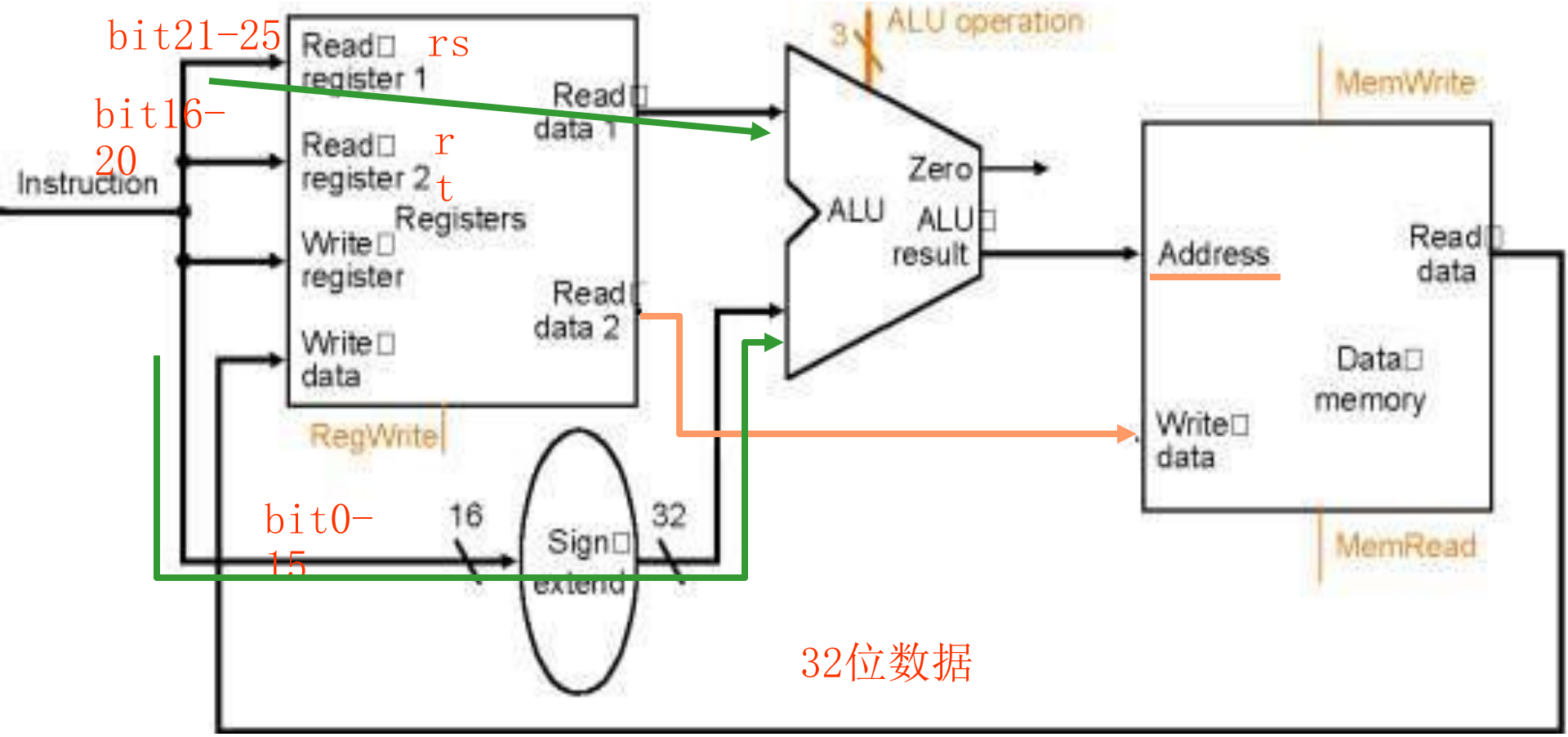


Implemente the R type instruction

instruction format:



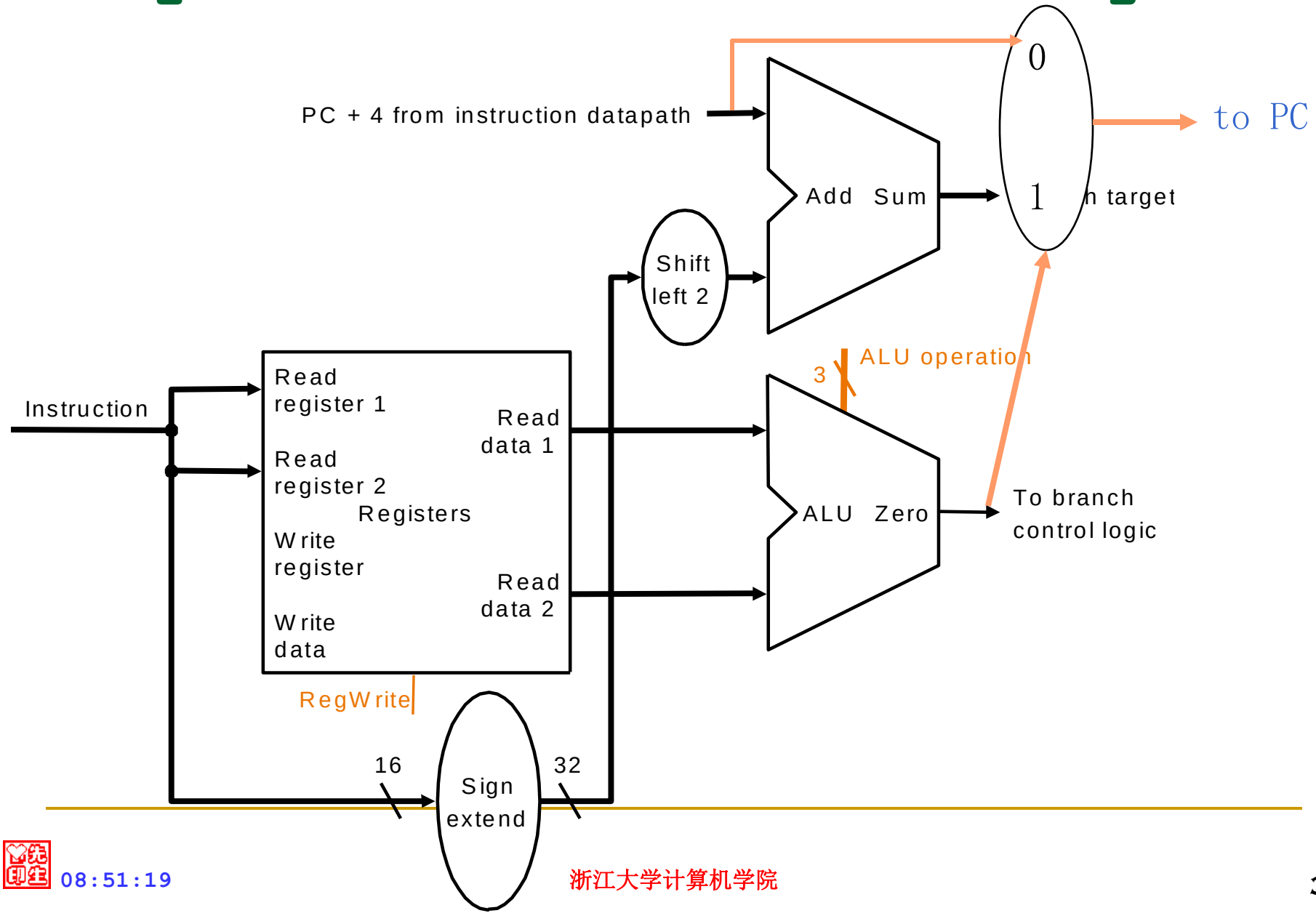
Implemente the I type



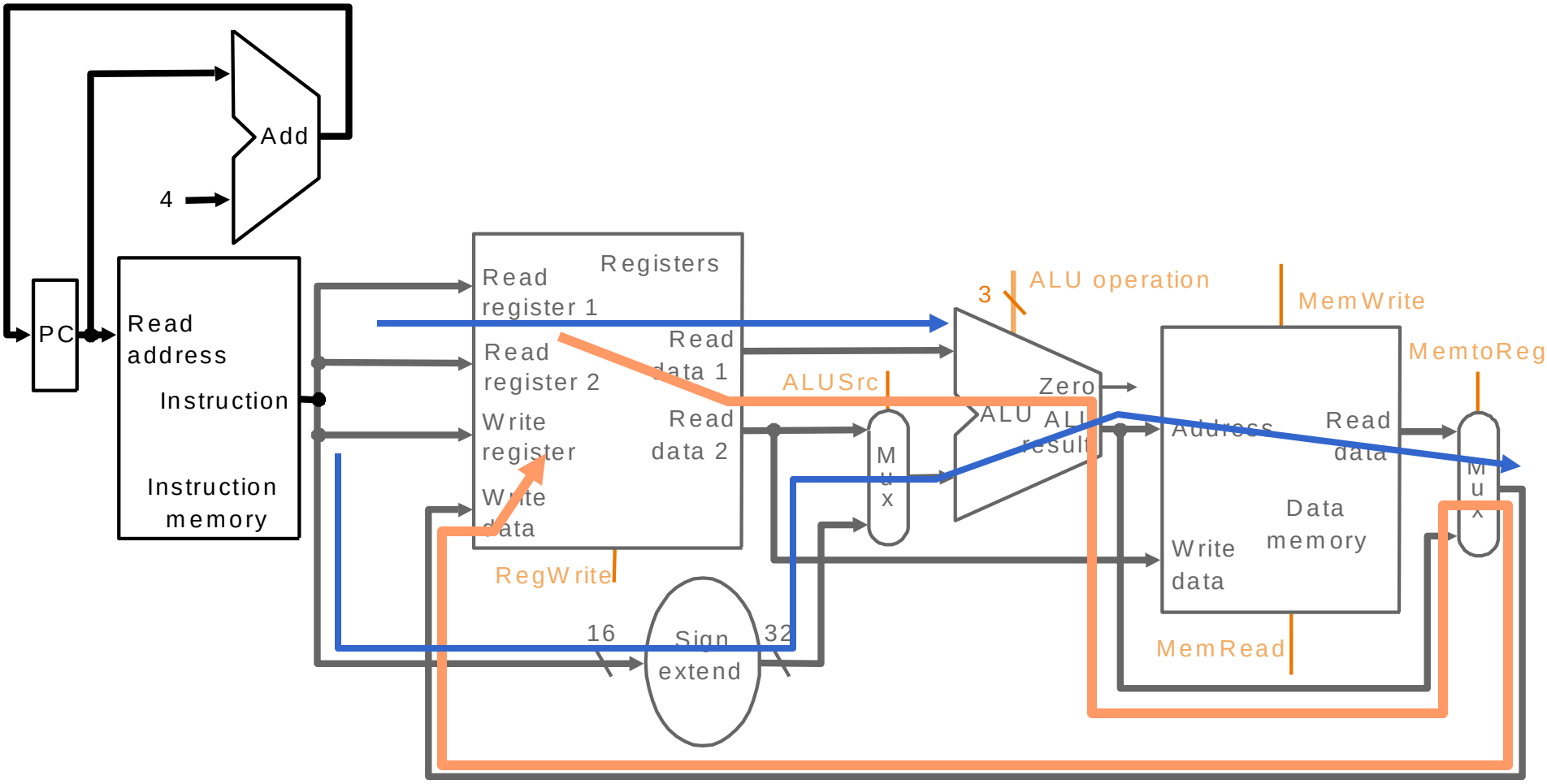
sw $t0, 200($s2)

若$s2=1000，则将$t0存入1200为地址的内存单元的一个字

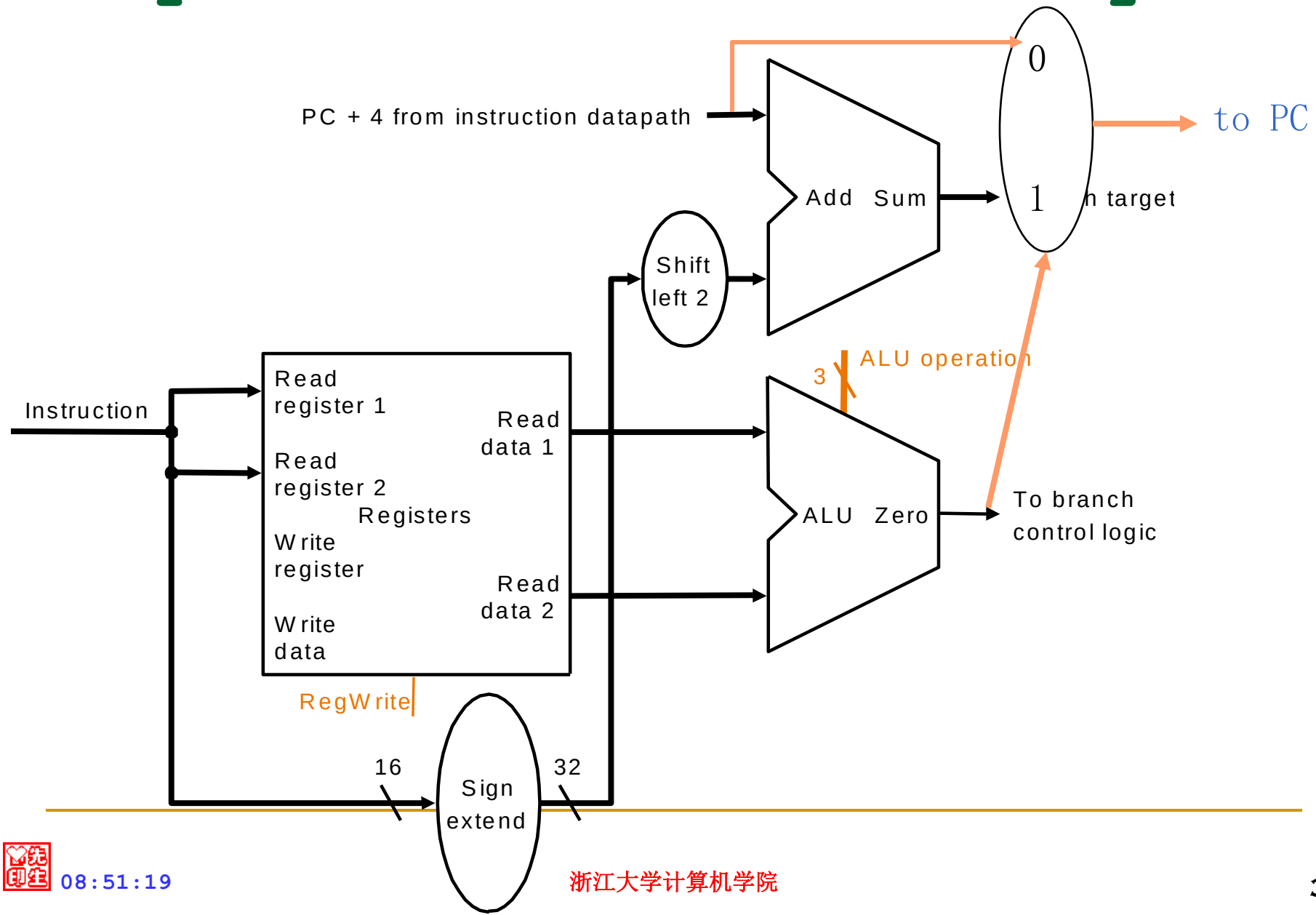
Implementation of *beq*



Combine R-type and I

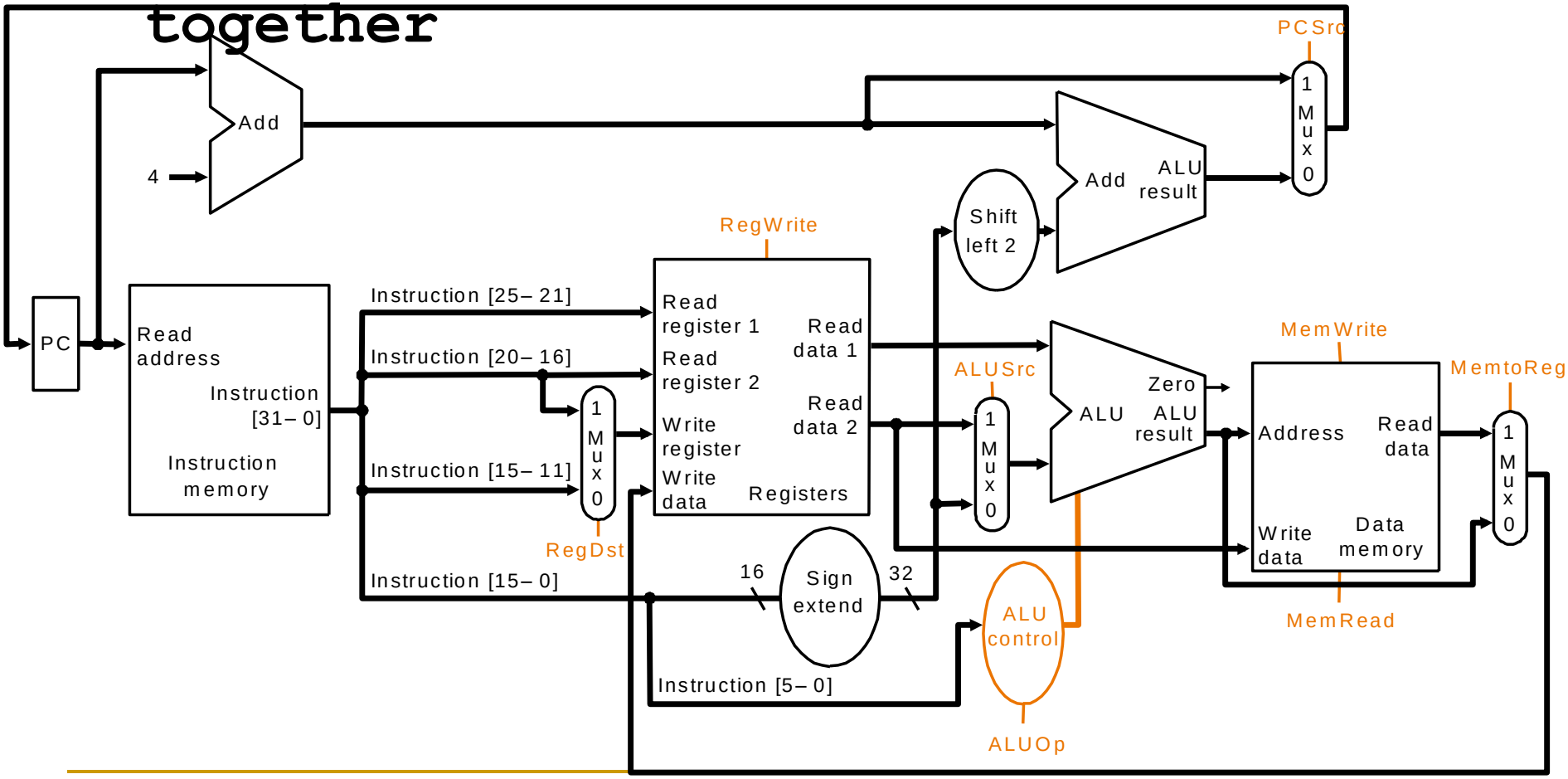


Implementation of *beq*



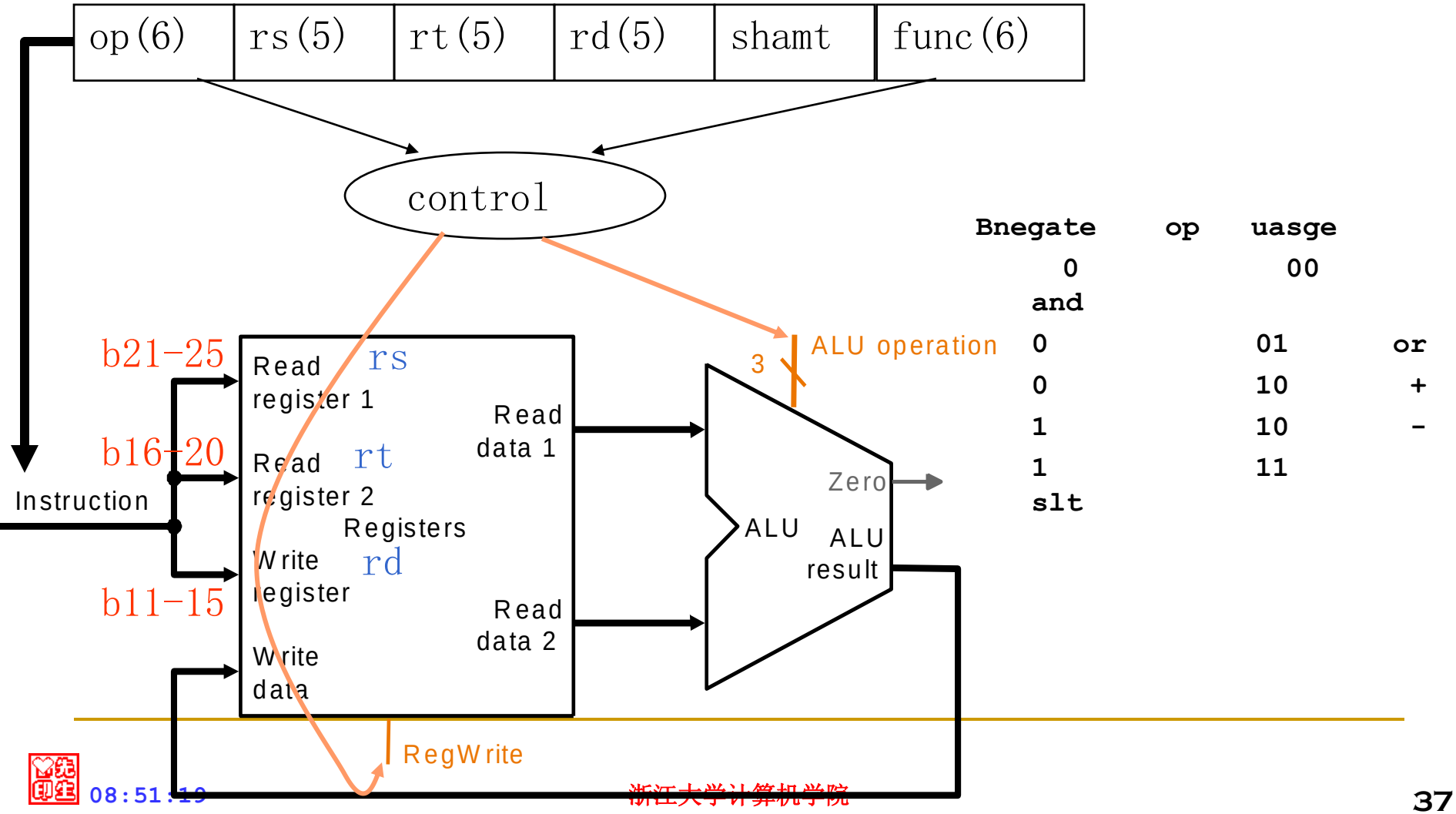
Building the Datapath

- Use multiplexors to stitch them together



The ALU control

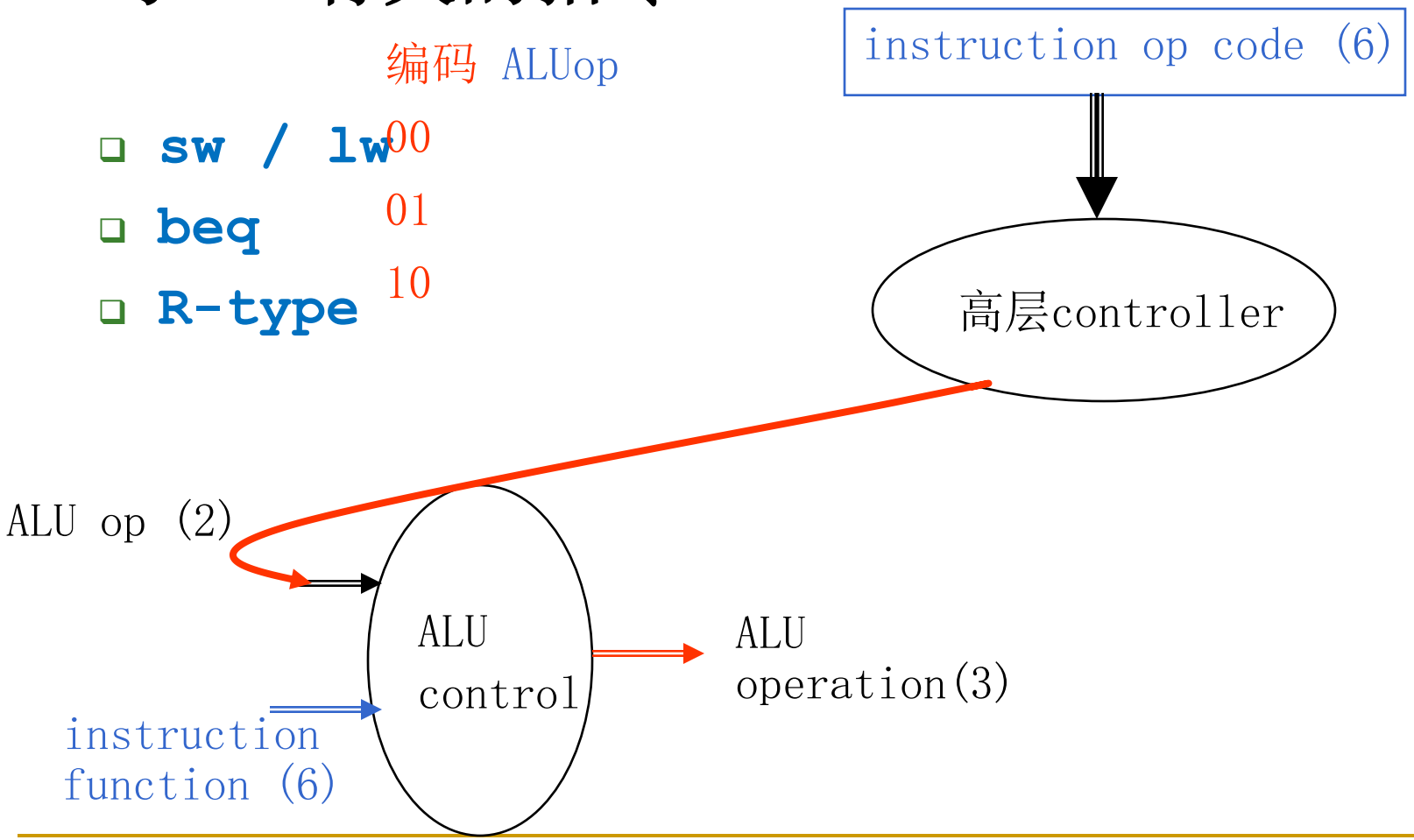
instruction format:



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

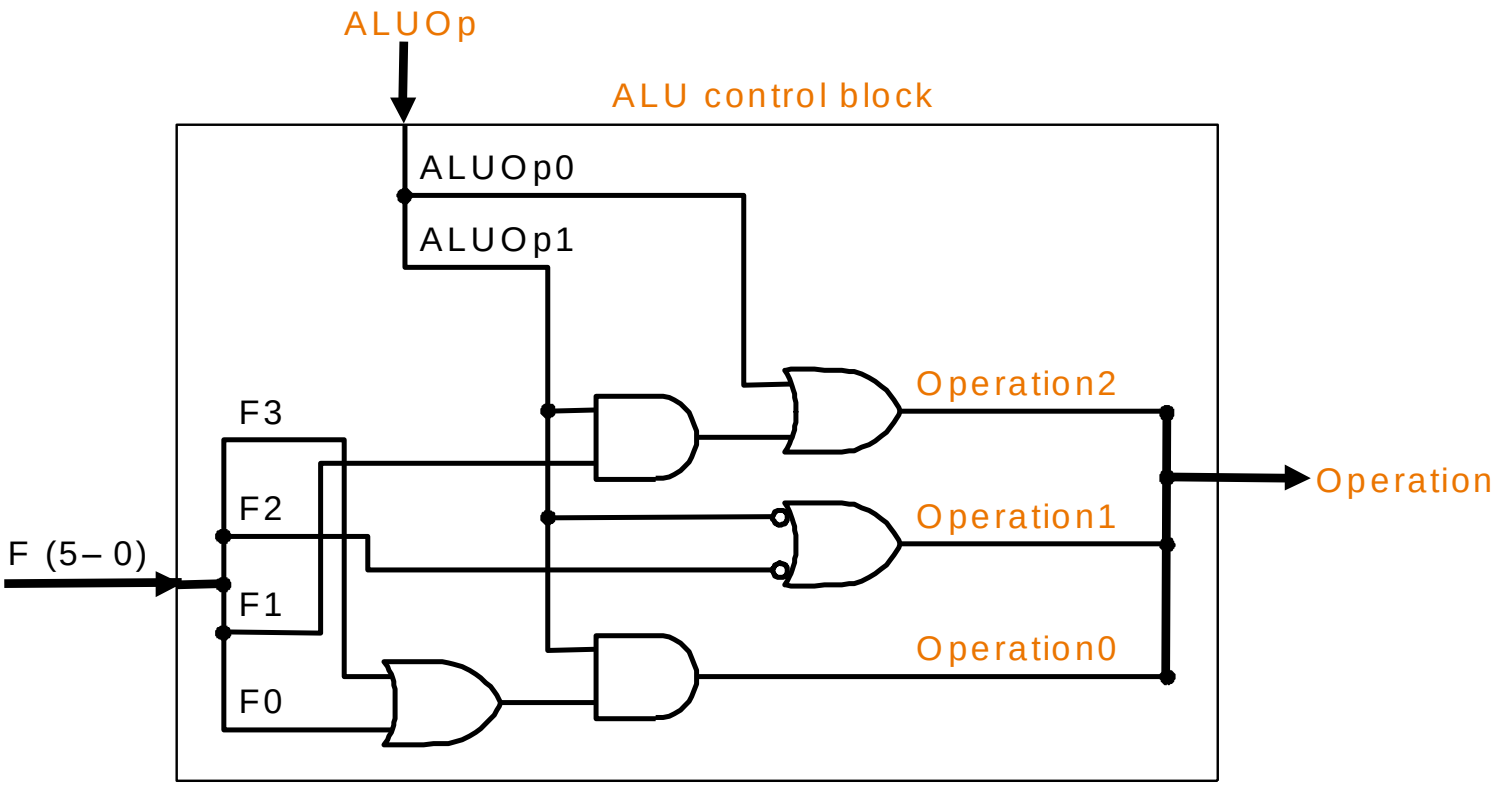
ALUOp computed from instruction type

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

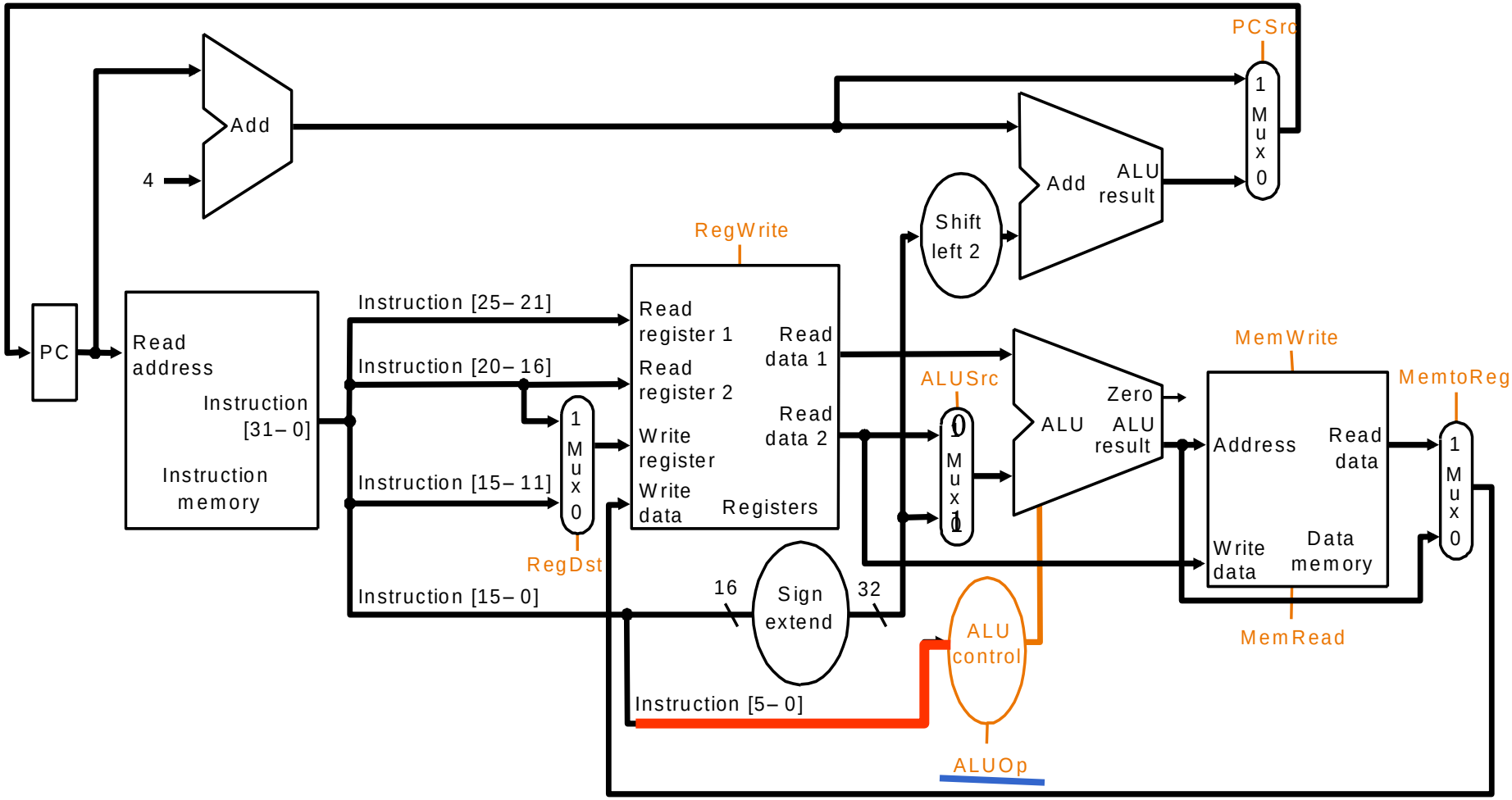
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

don’t care

The ALU control

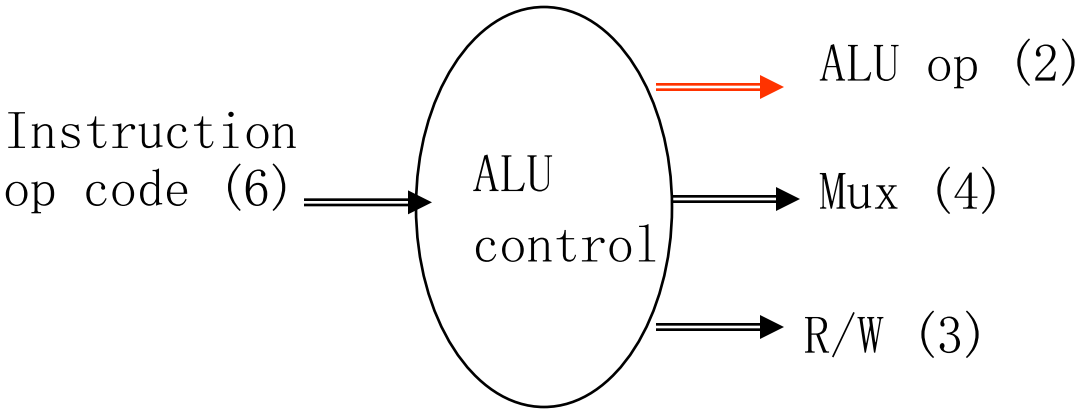


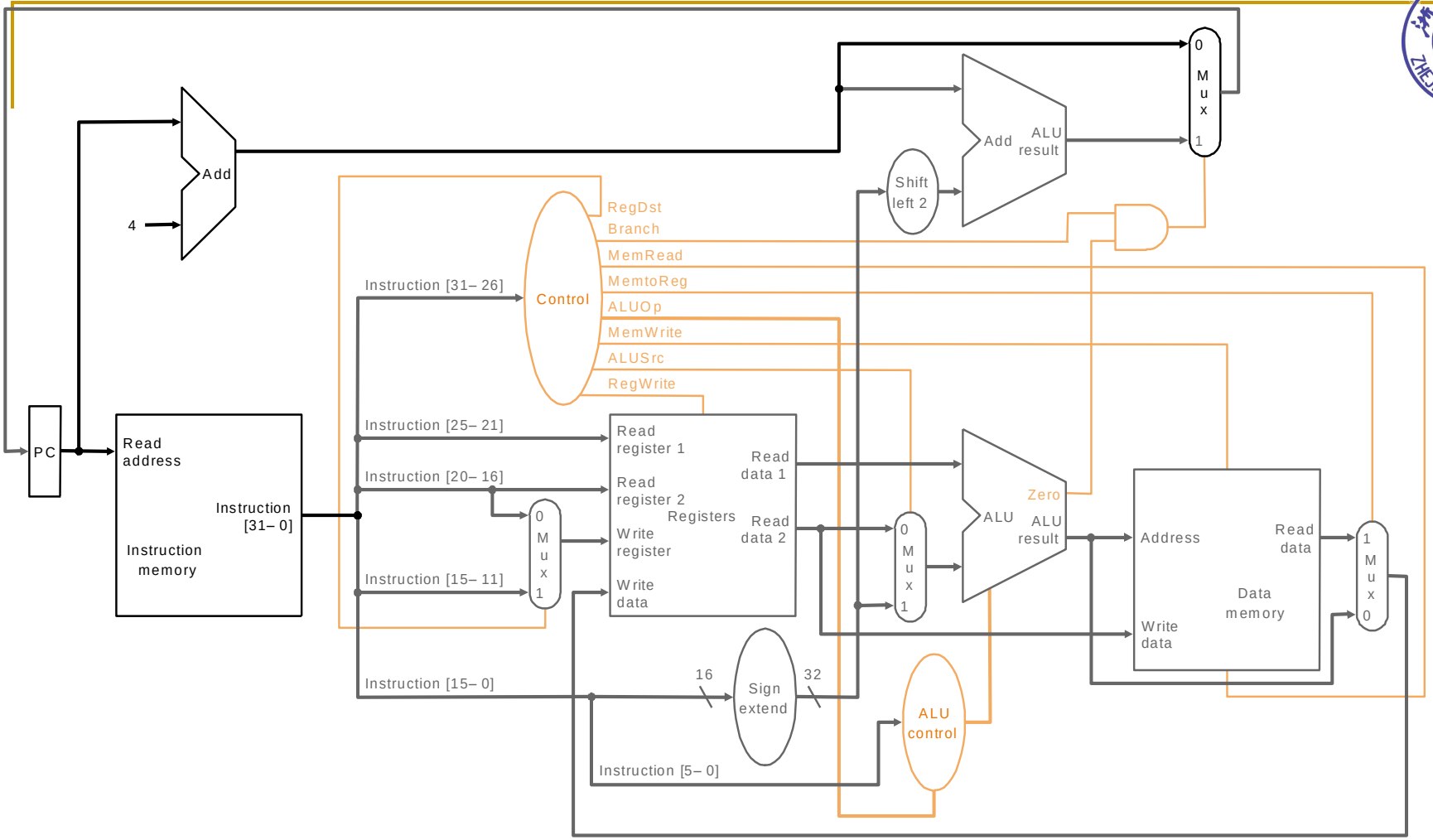
The ALU control



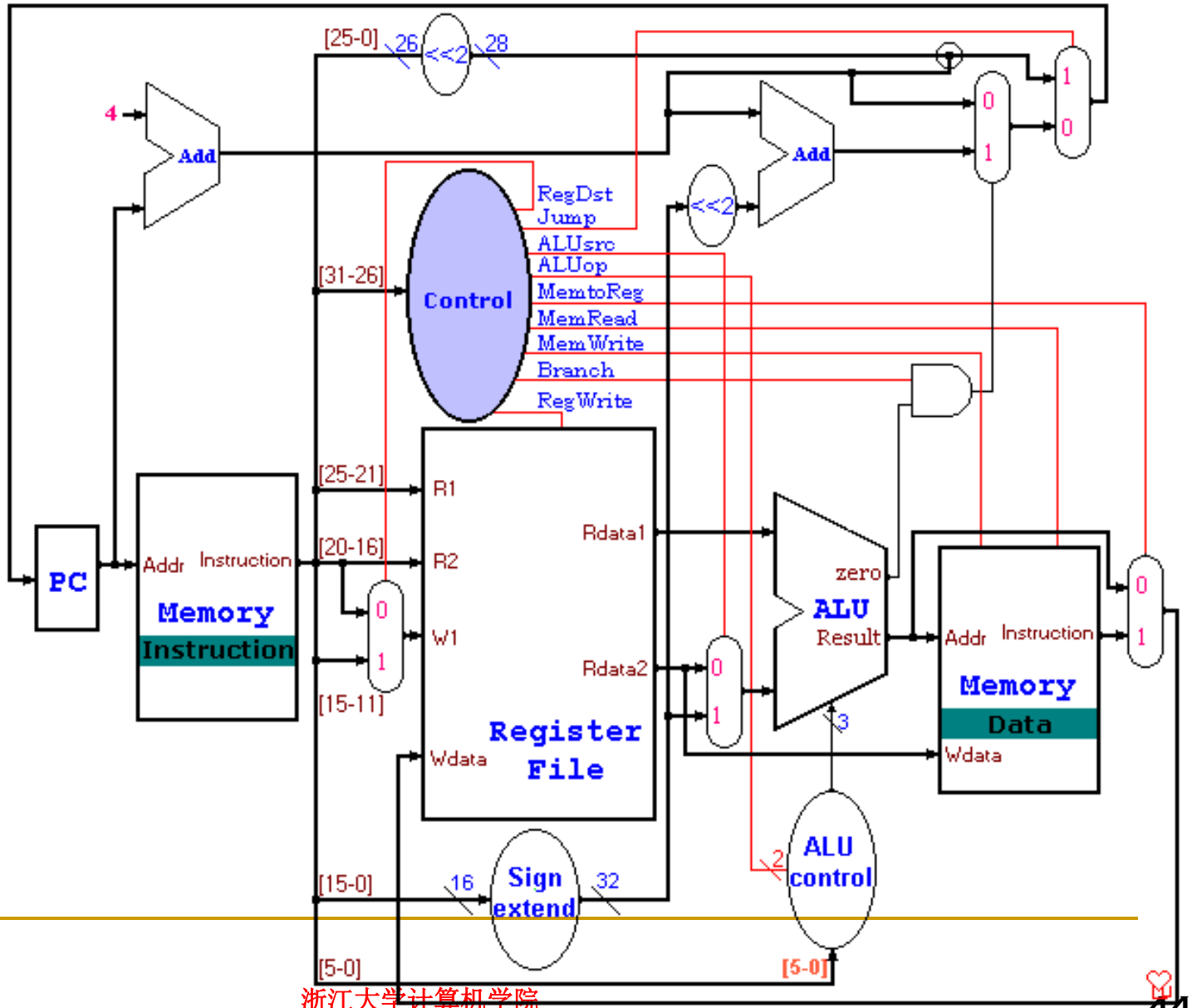
Designing the Main Control Unit

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





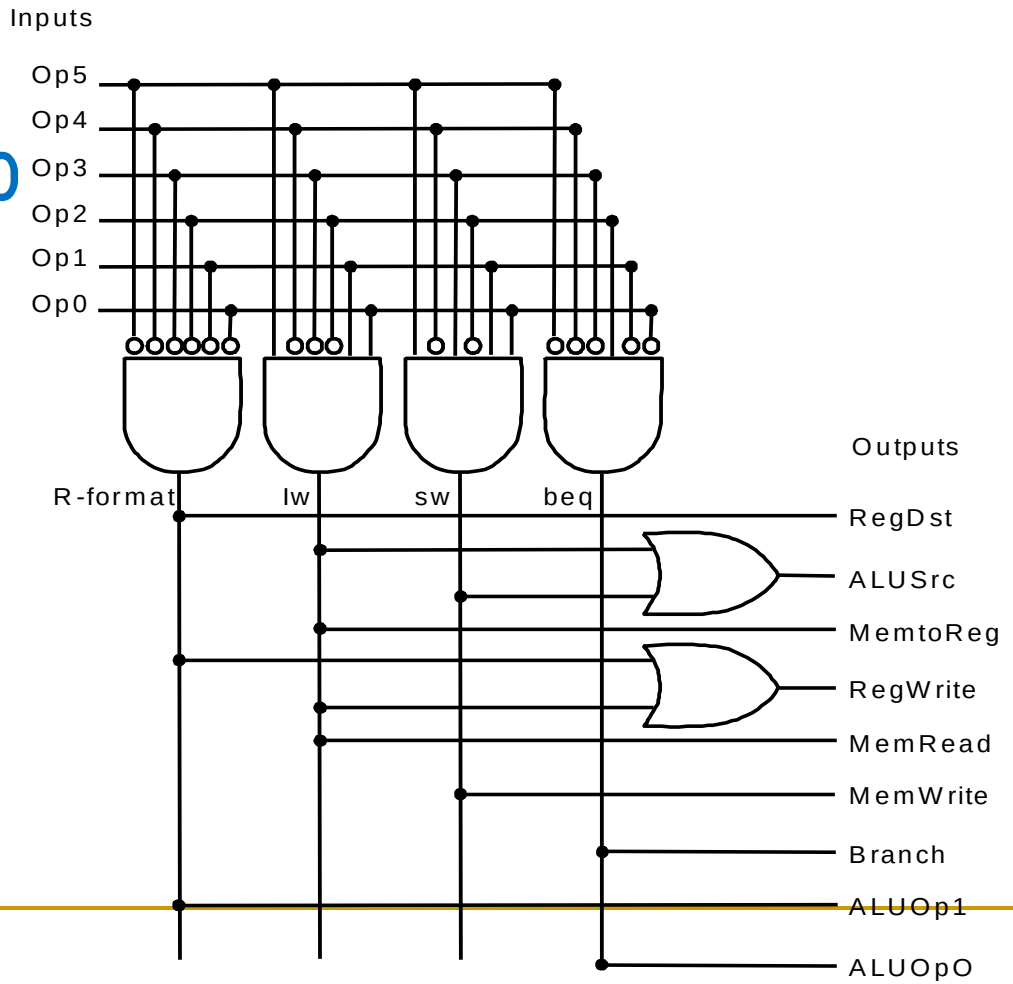
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1


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

Control

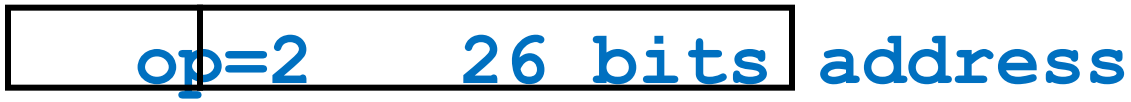
- Simple combinational logic (truth tables)

R-type	0
lw	35
sw	43
beq	4

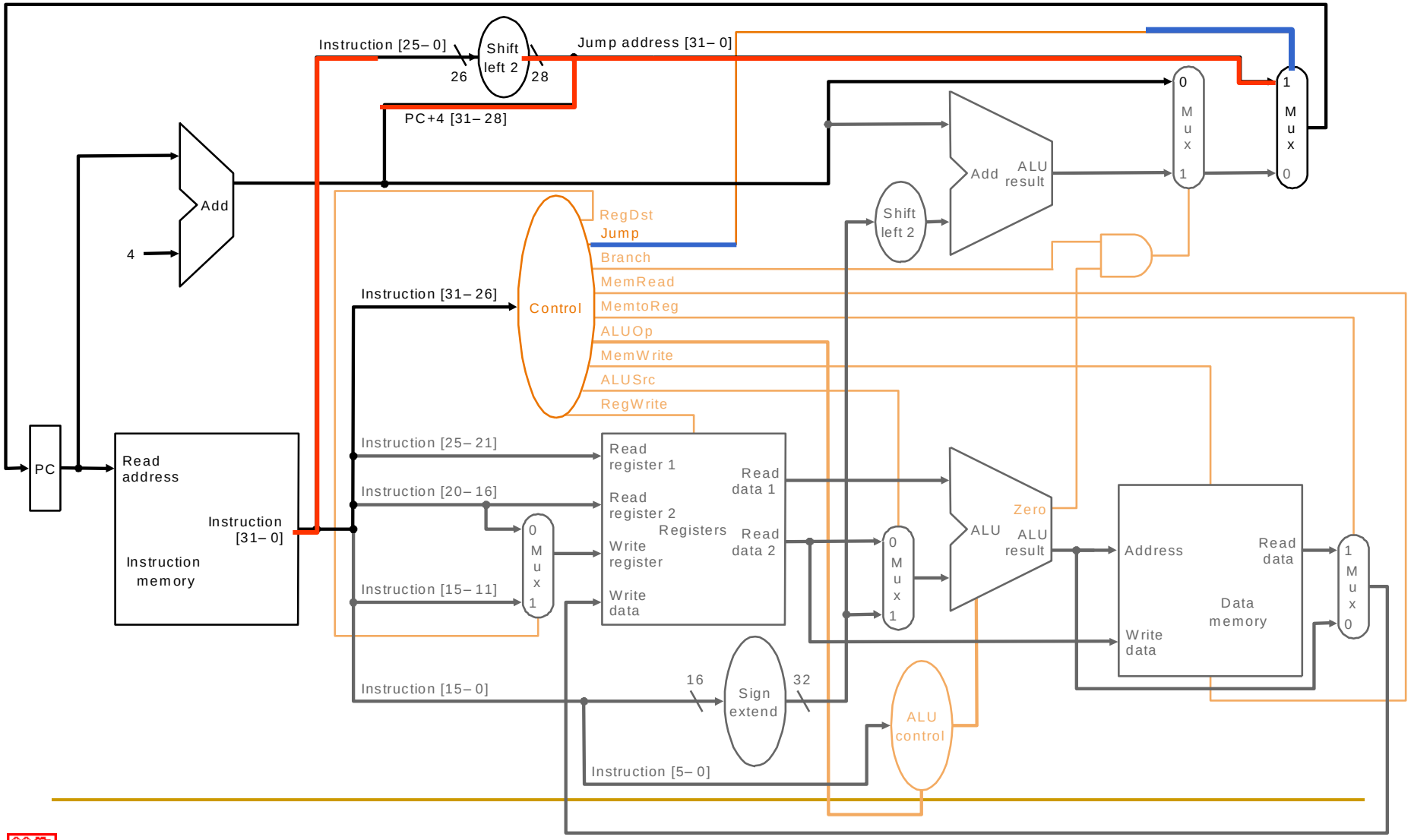


j instruction

- instruction format
  - j Label

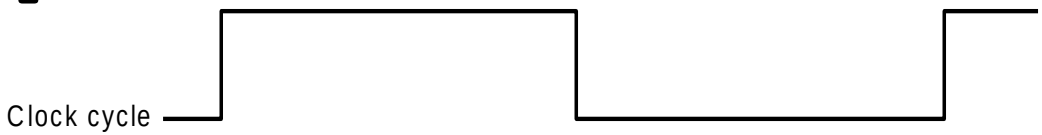


- Implementation
  - $pc = pc_{28\sim31}$ + 26bits-address * 4


Instruction [25– 0]
Shift left 2
26
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]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
ALU
Zero
ALU result
Address
Read data
Data memory
Write data
16
Sign extend
32
ALU control
Instruction [5– 0]

Simple Control Structure

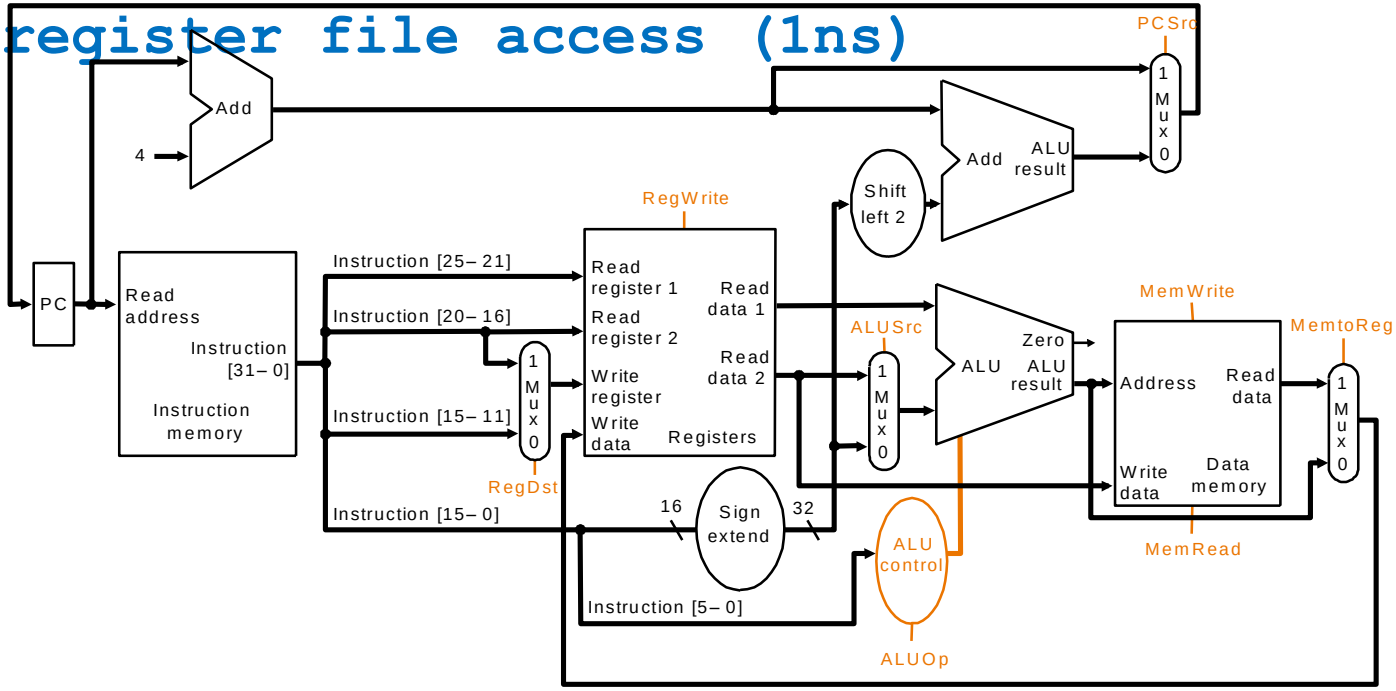
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer?right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times

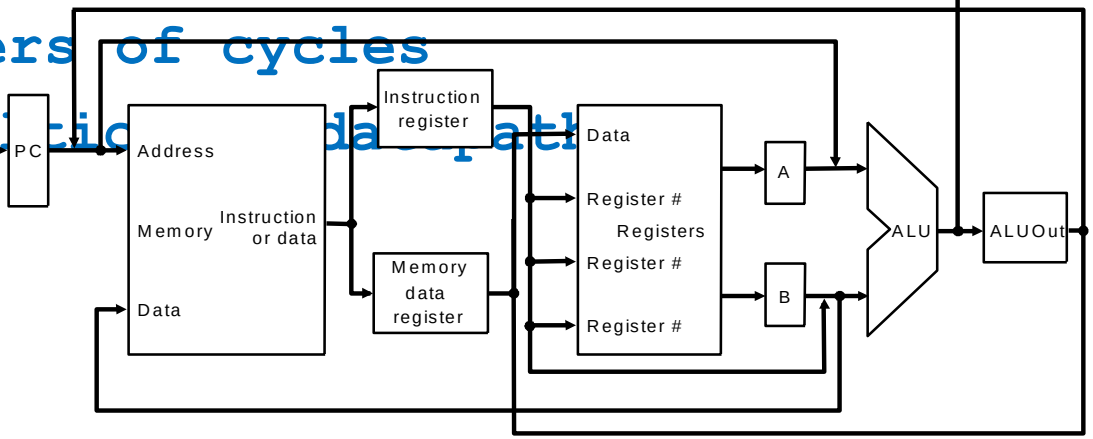
Single Cycle

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

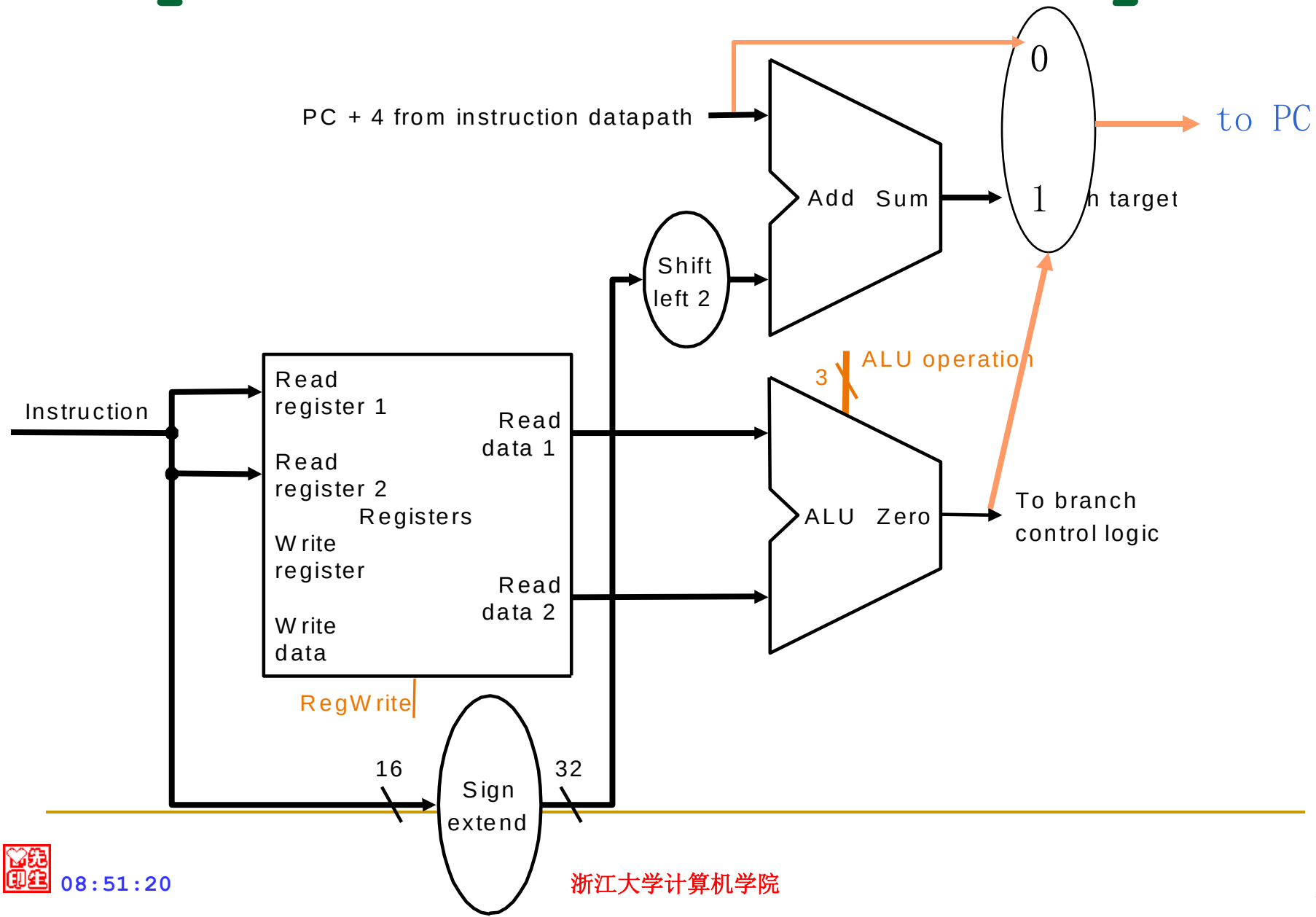


Where we are headed

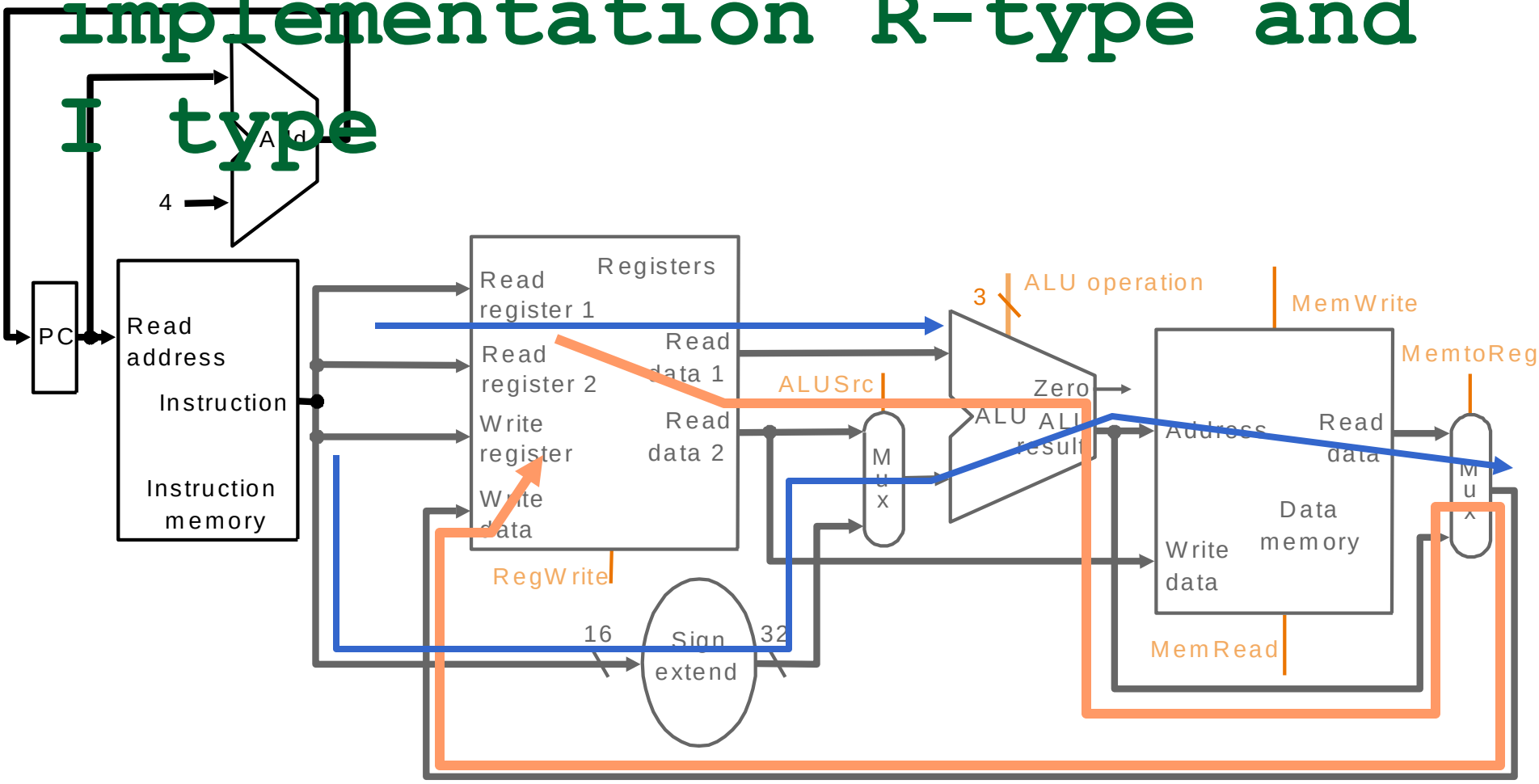
- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a 撈maller?cycle time
  - have different instructions take different numbers of cycles
  - a 撂ultic[illegible]datapath



Implementation of *beq*

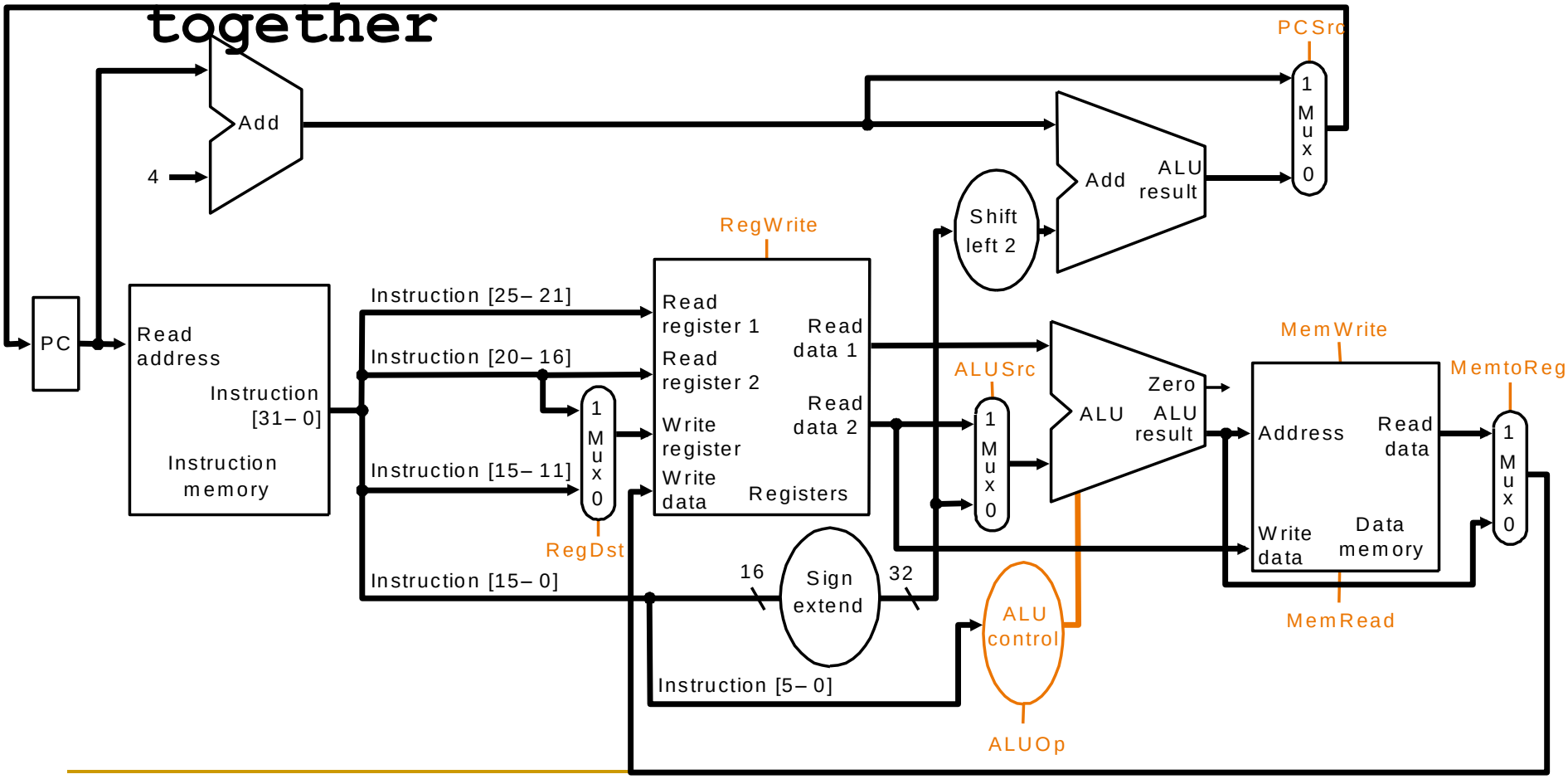


combine the implementation R-type and I type



Building the Datapath

- Use multiplexors to stitch them together



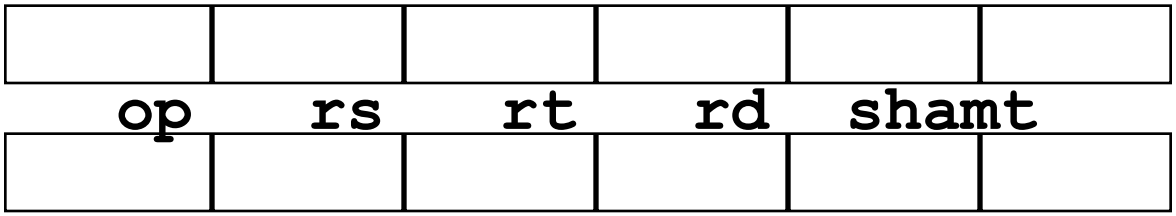
Control

- 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

Control

- **Example:**

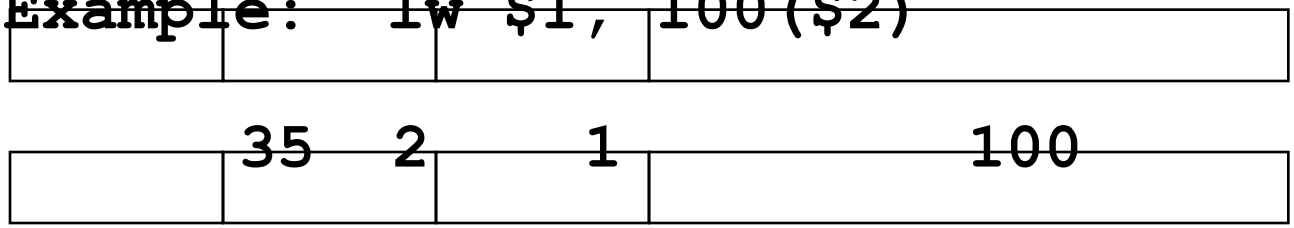
```
add $8, $17, $18    Instruction Format:

000000      10001      10010      01000
 00000     100000
```



- **ALU's operation based on instruction type and function code**

Control

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



op rs rt 16 bit offset

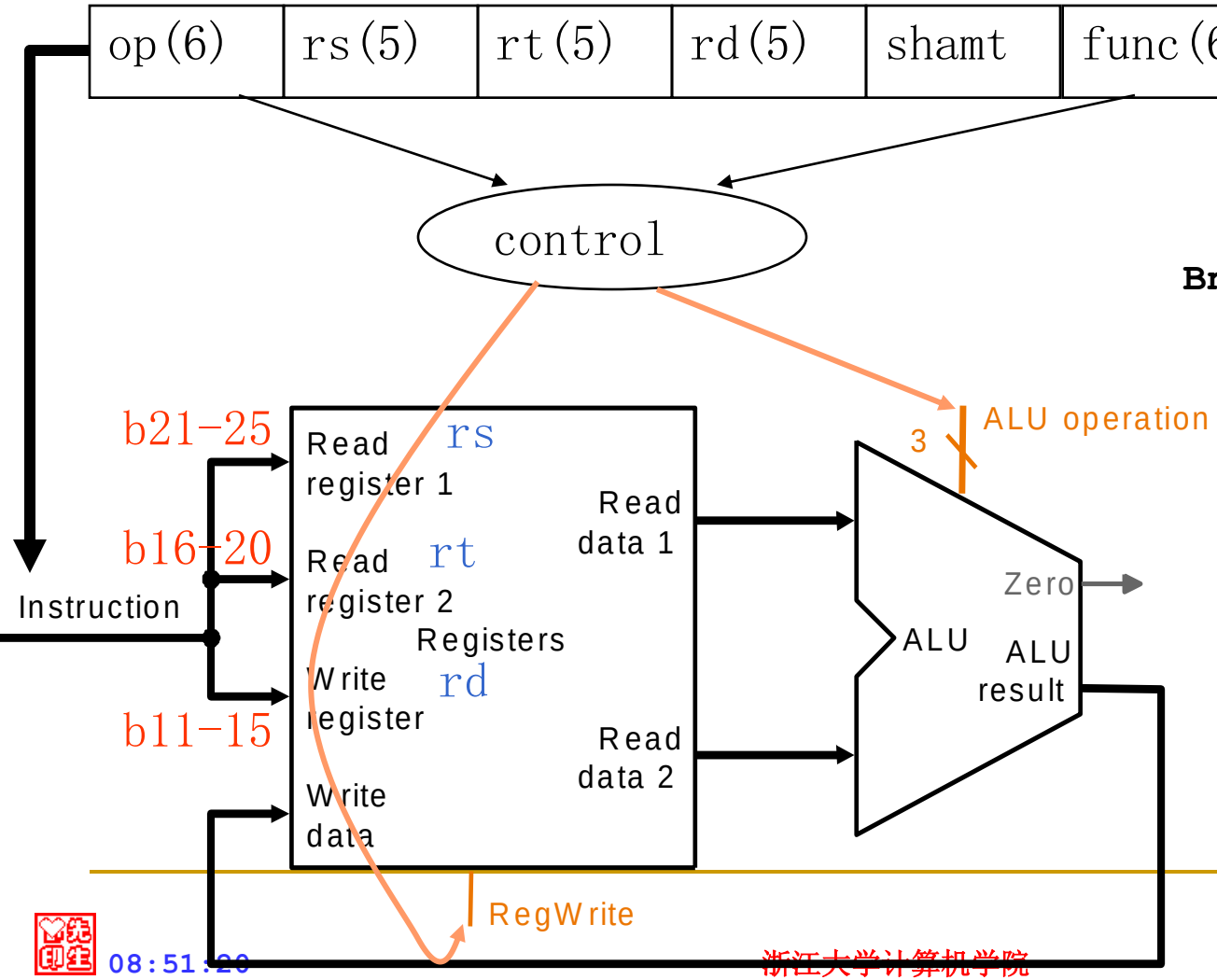
ALU control input

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:



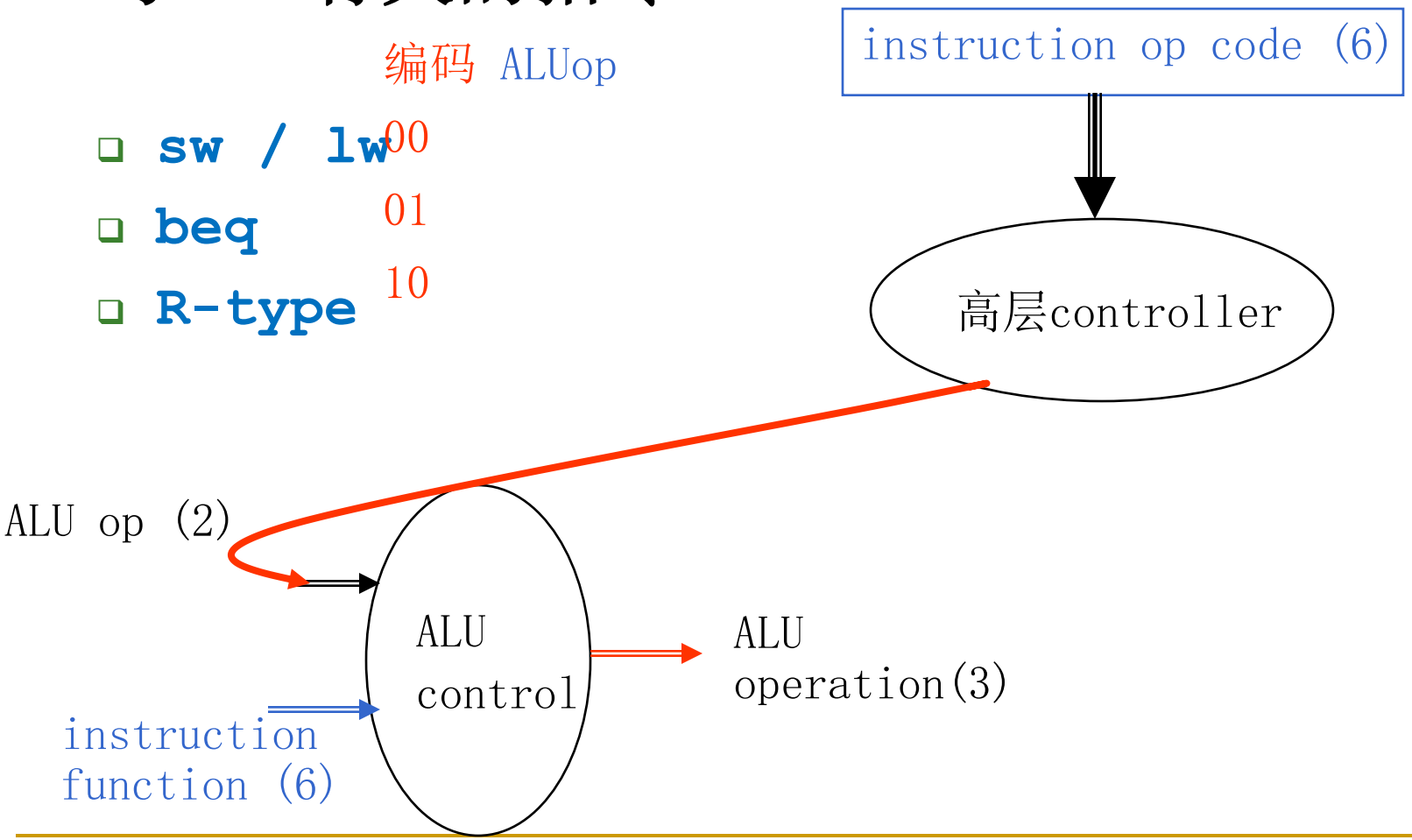
Bnegate	op	uasge
0		00
and		
0		01
or		
0		10
+		
1		10
-		
1		11
slt		

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

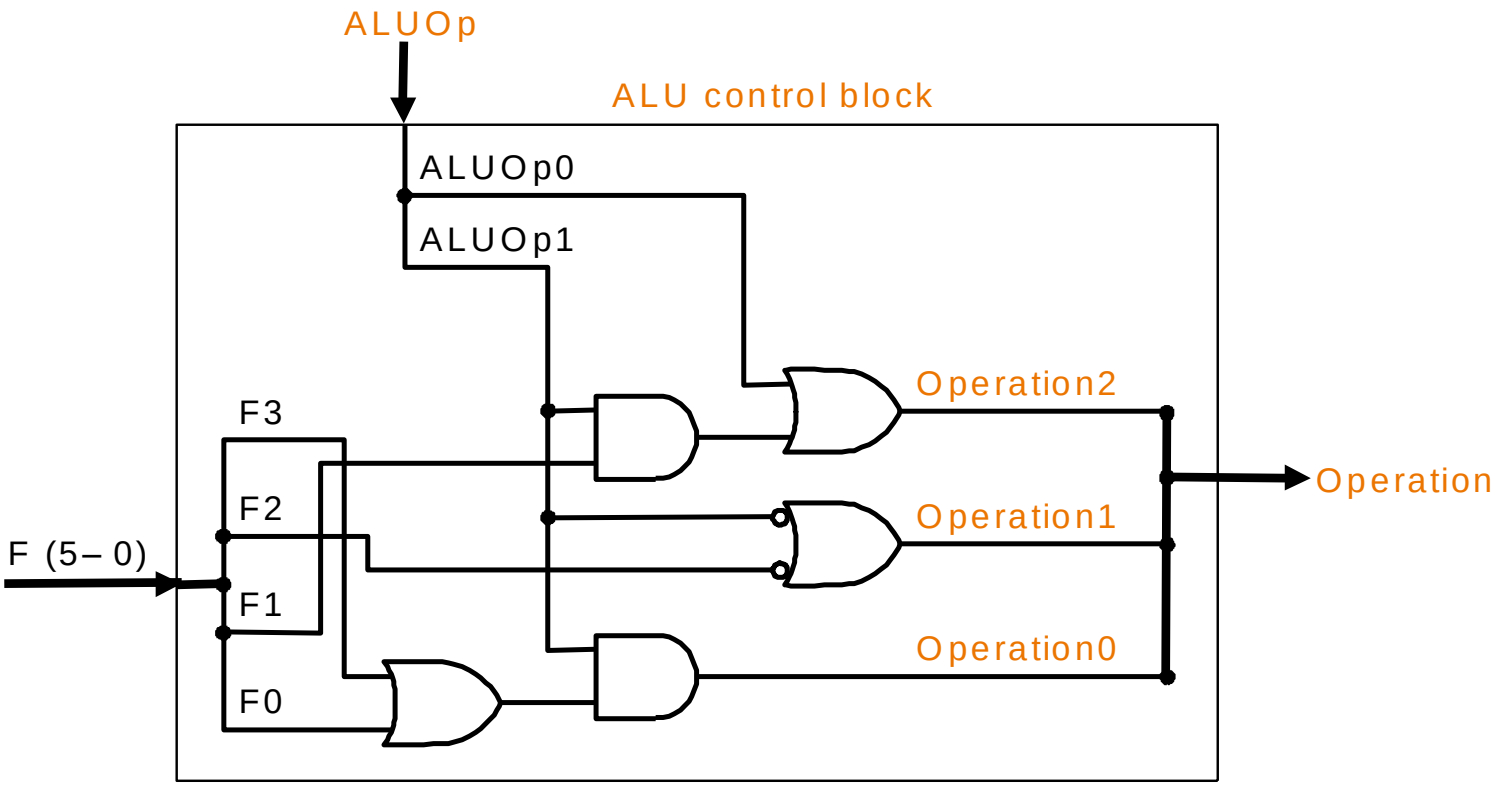
ALUOp computed from instruction type

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

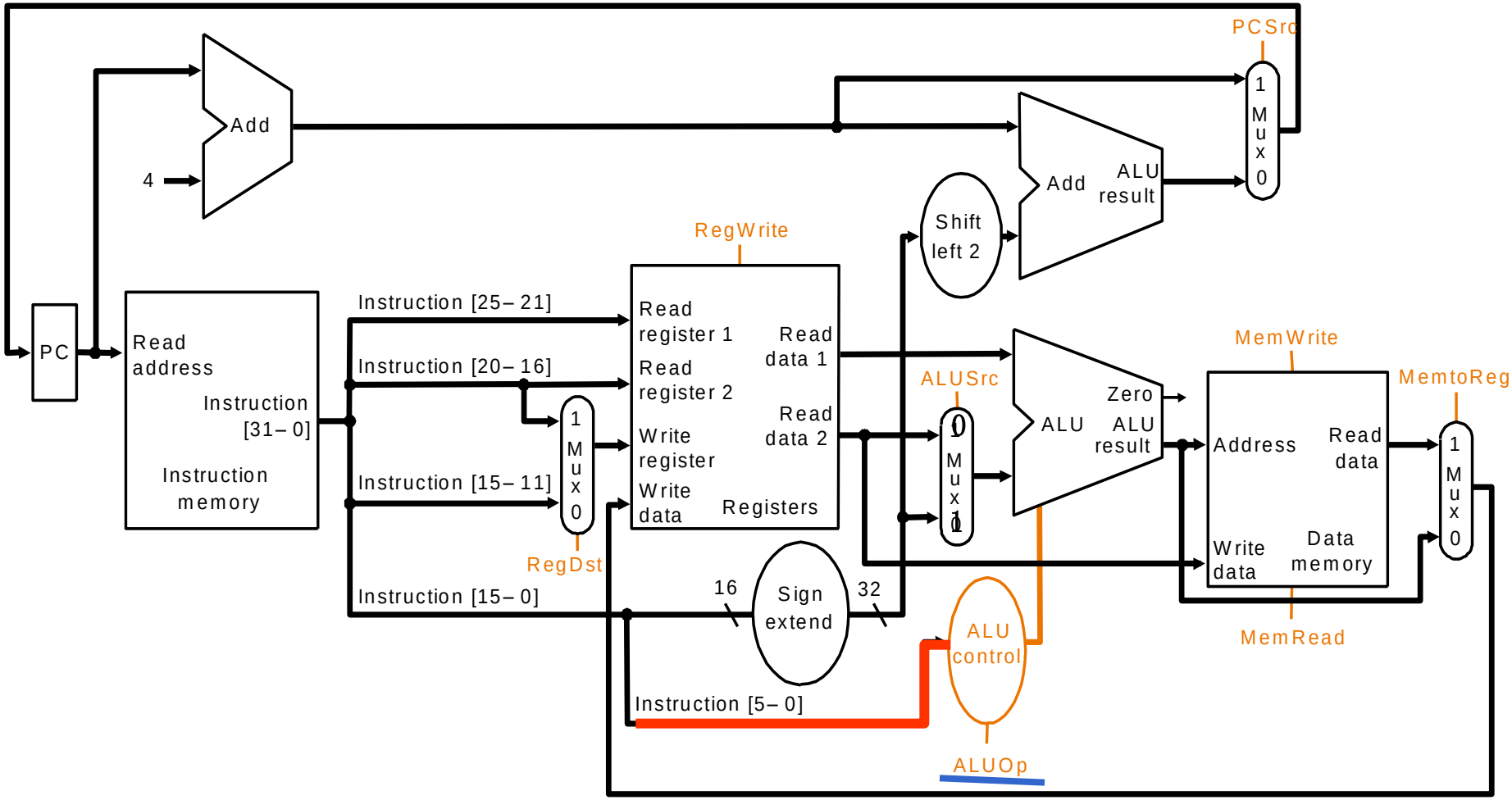
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	X	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

don’t care

The ALU control

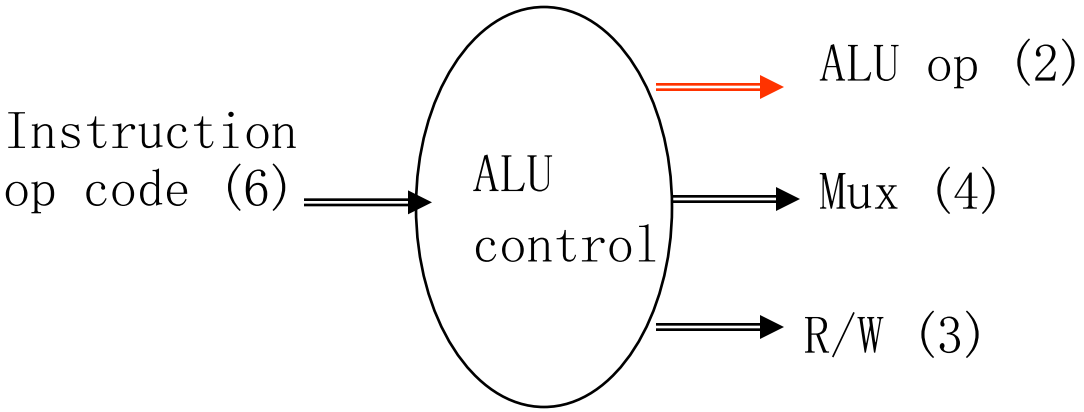


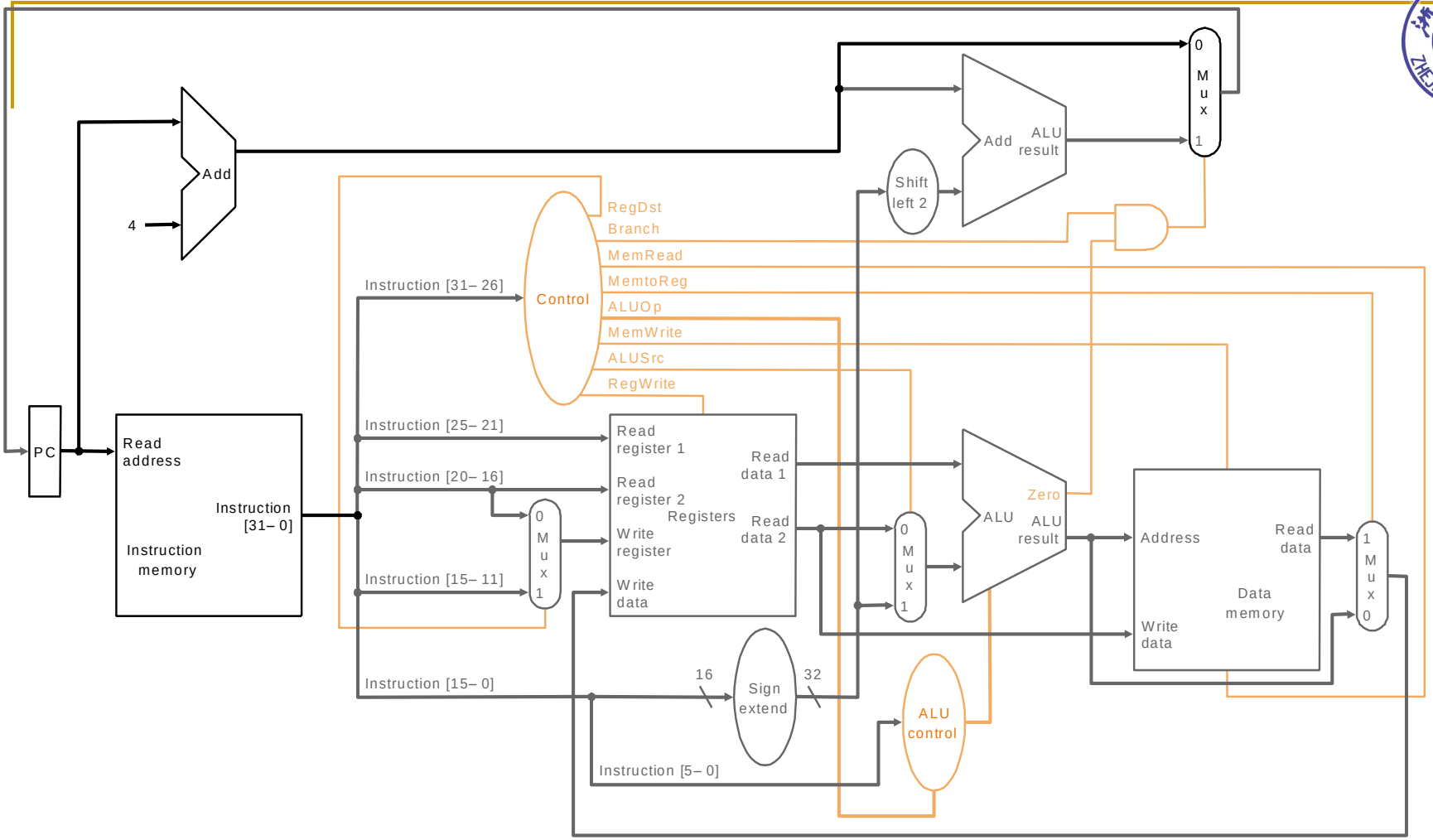
The ALU control



Designing the Main Control Unit

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



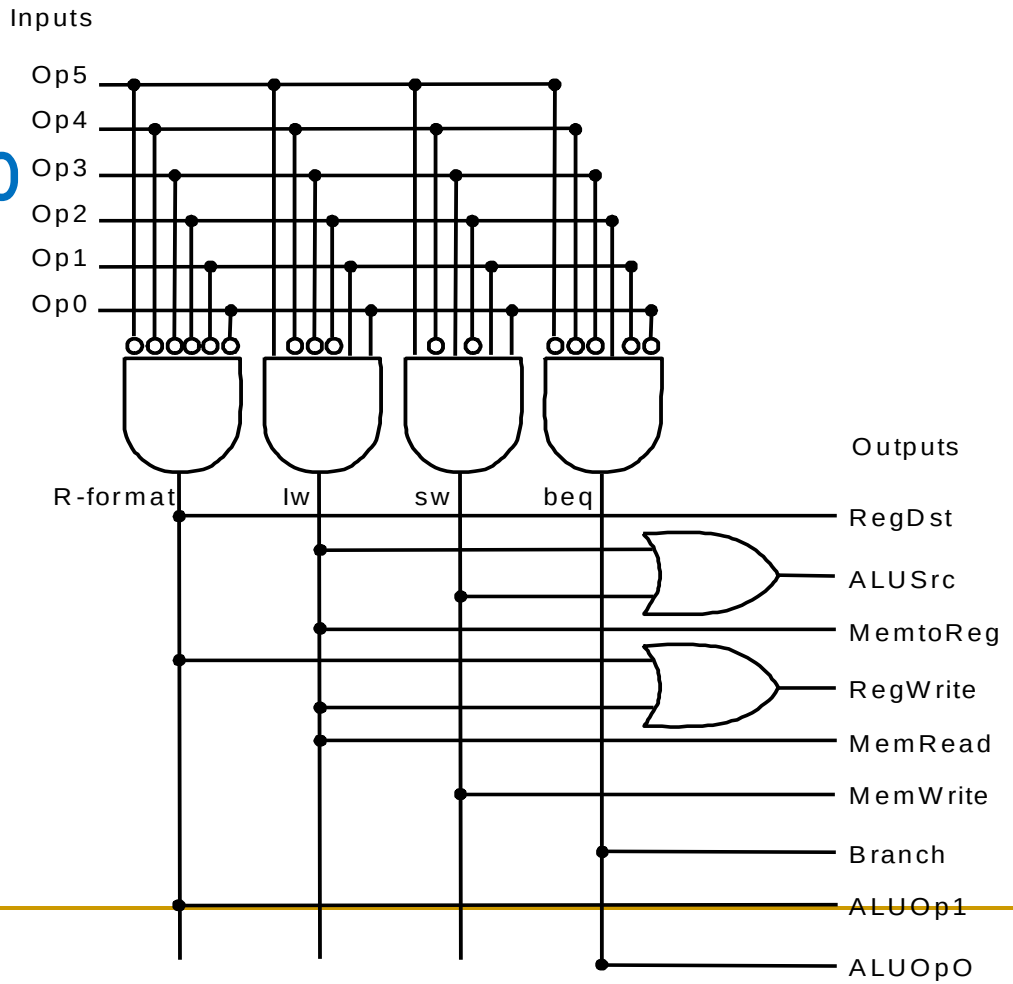


Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Control

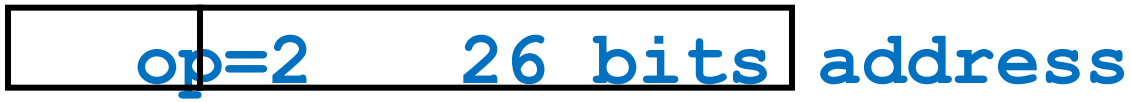
- Simple combinational logic (truth tables)

R-type	0
lw	35
sw	43
beq	4

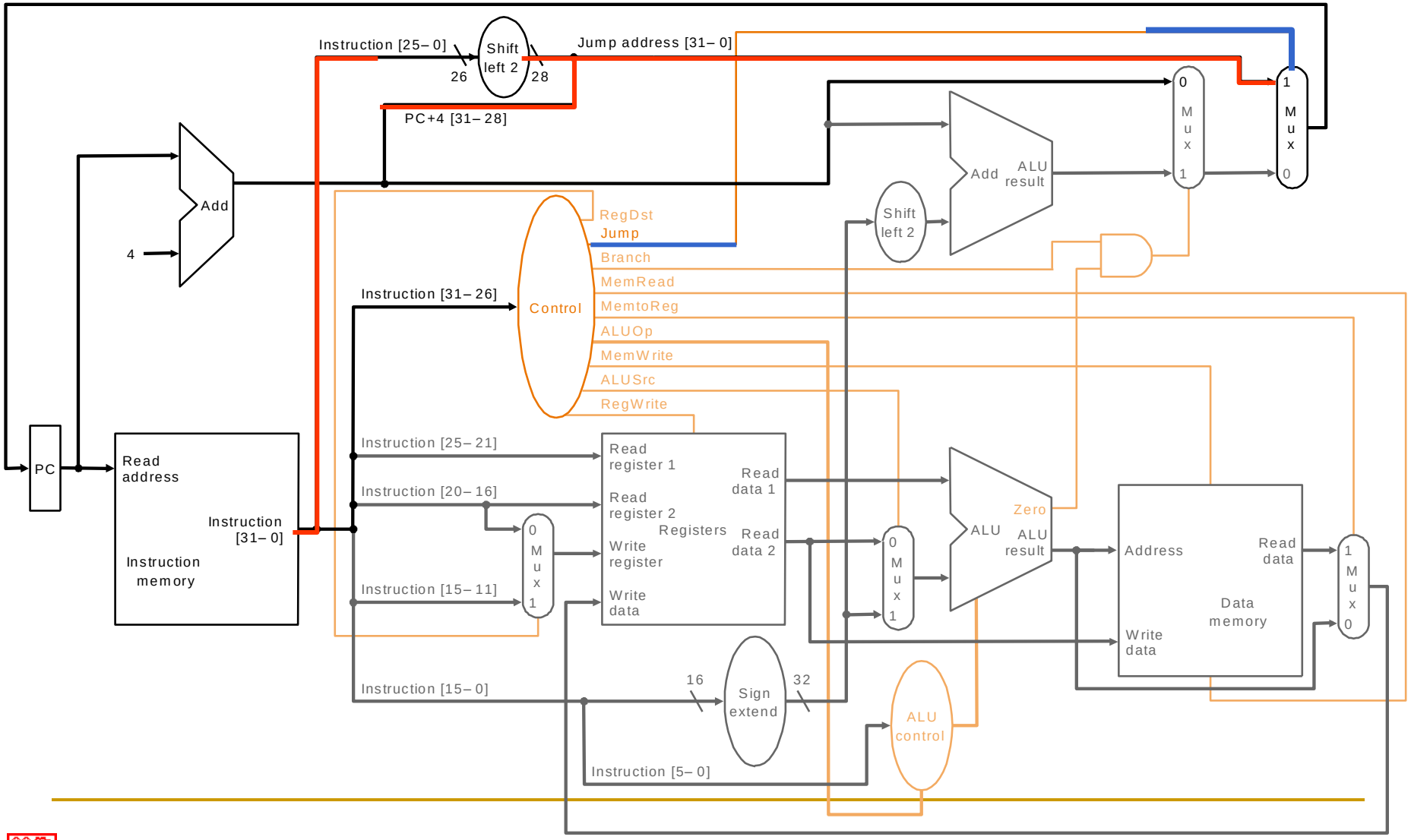


j instruction

- instruction format
  - j Label

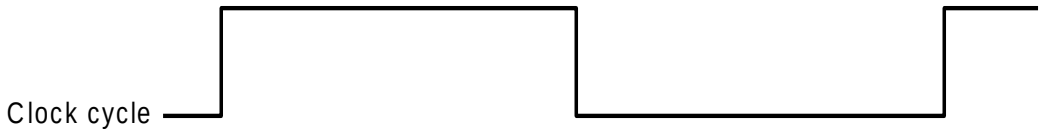


- Implementation
  - $pc = pc_{28\sim31} + 26bits\text{-}address * 4$


Instruction [25– 0]
Shift left 2
26
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]
Instruction [5– 0]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
16
Sign extend
32
ALU
Zero
ALU result
ALU control
Address
Read data
Data memory
Write data

Our Simple Control Structure

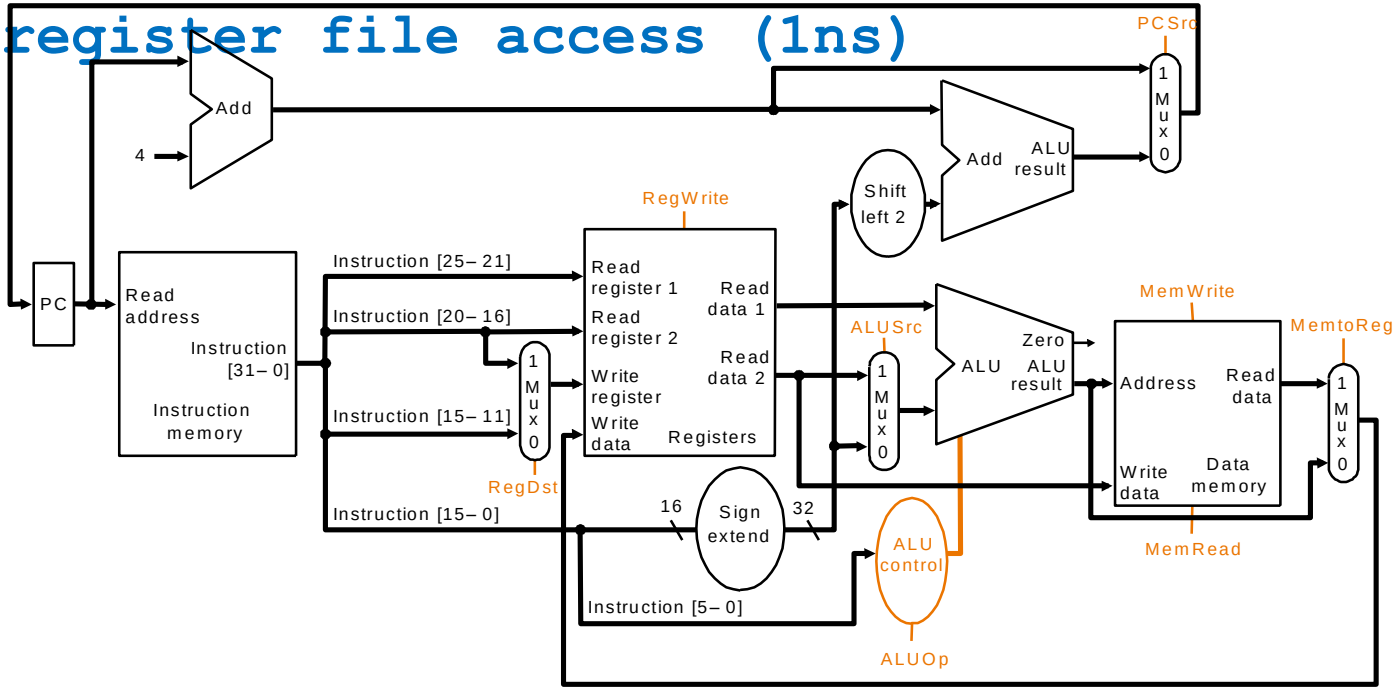
- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce right answer?right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



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)

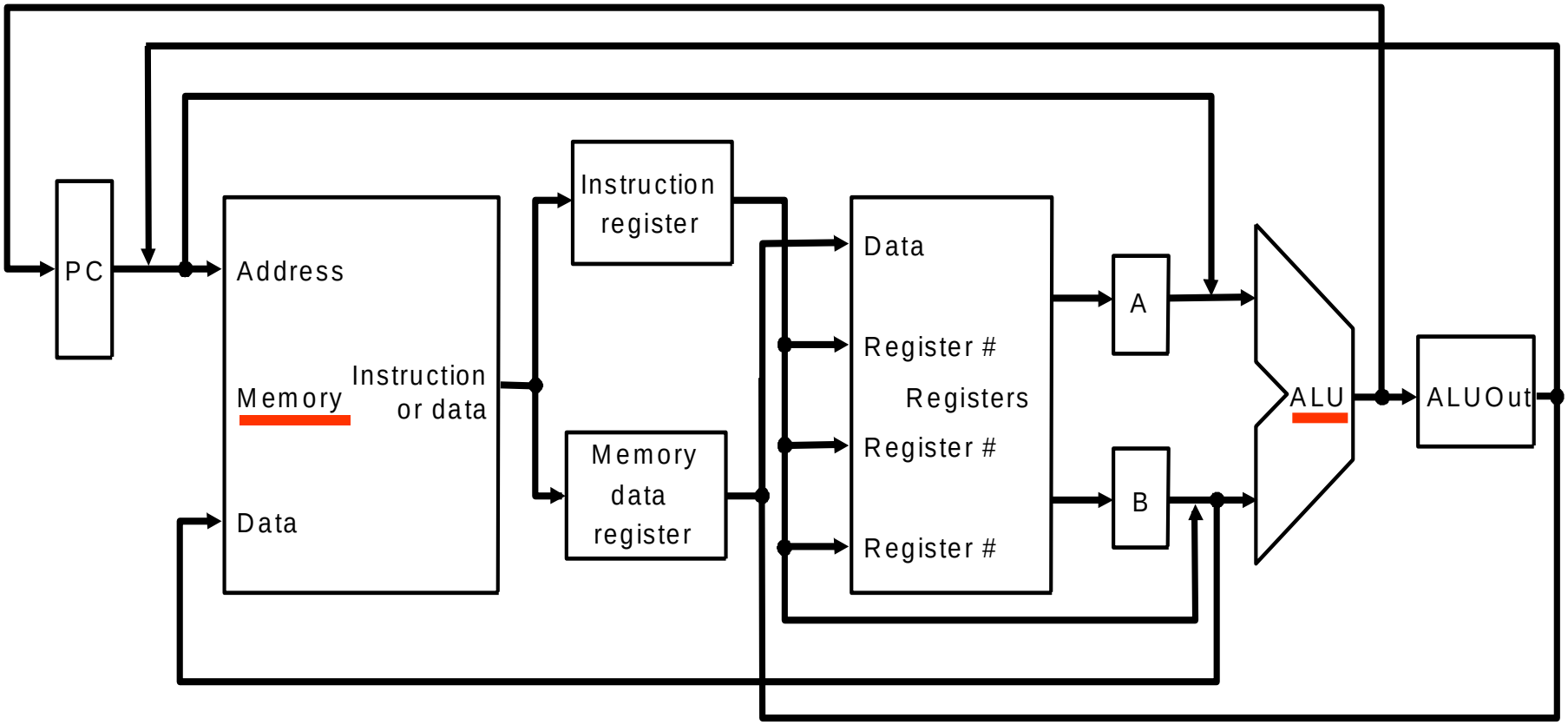


Where we are headed

- 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

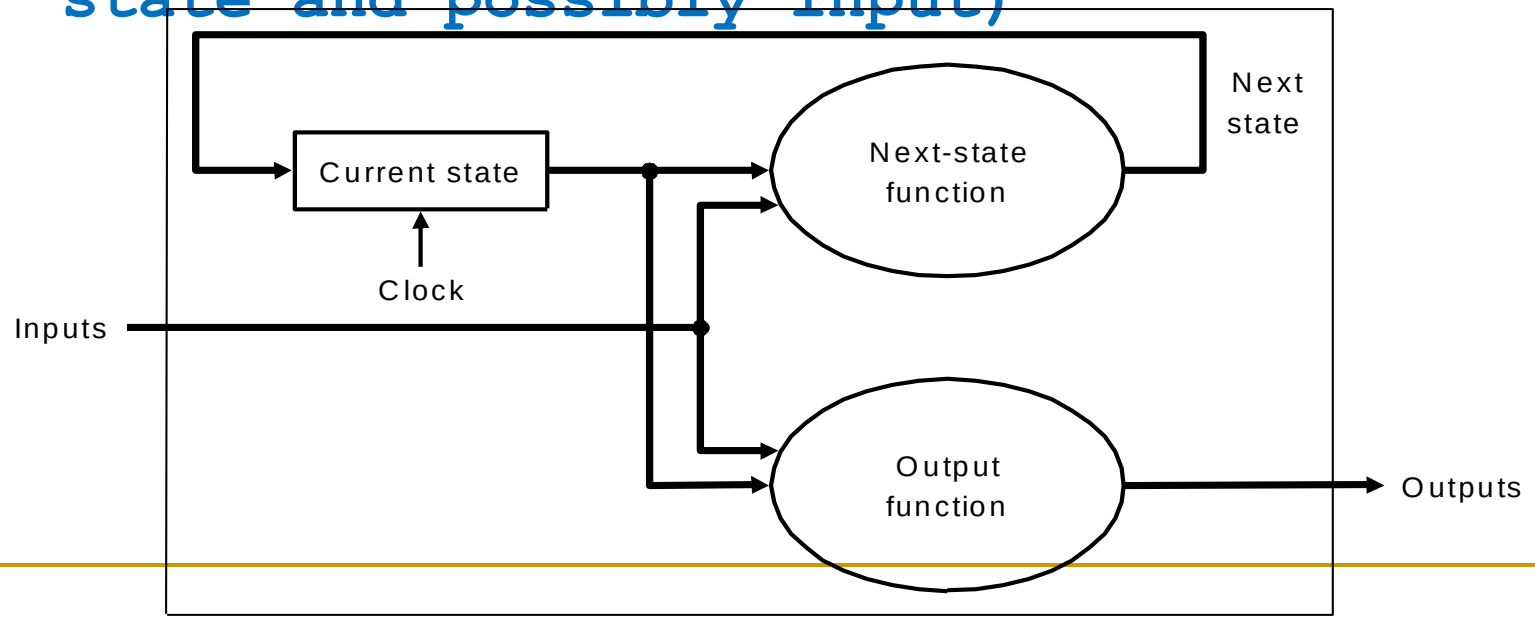
Multicycle Approach

- 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)

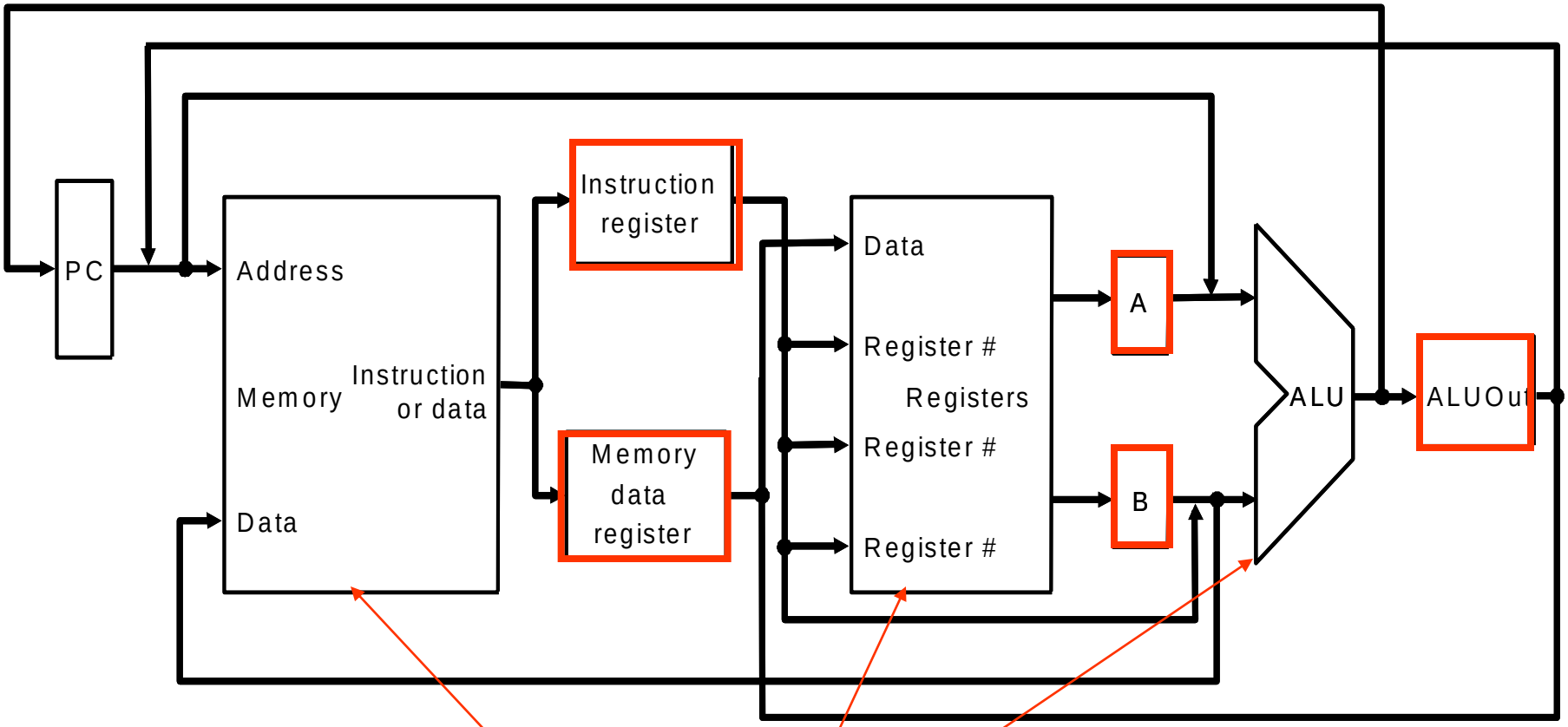


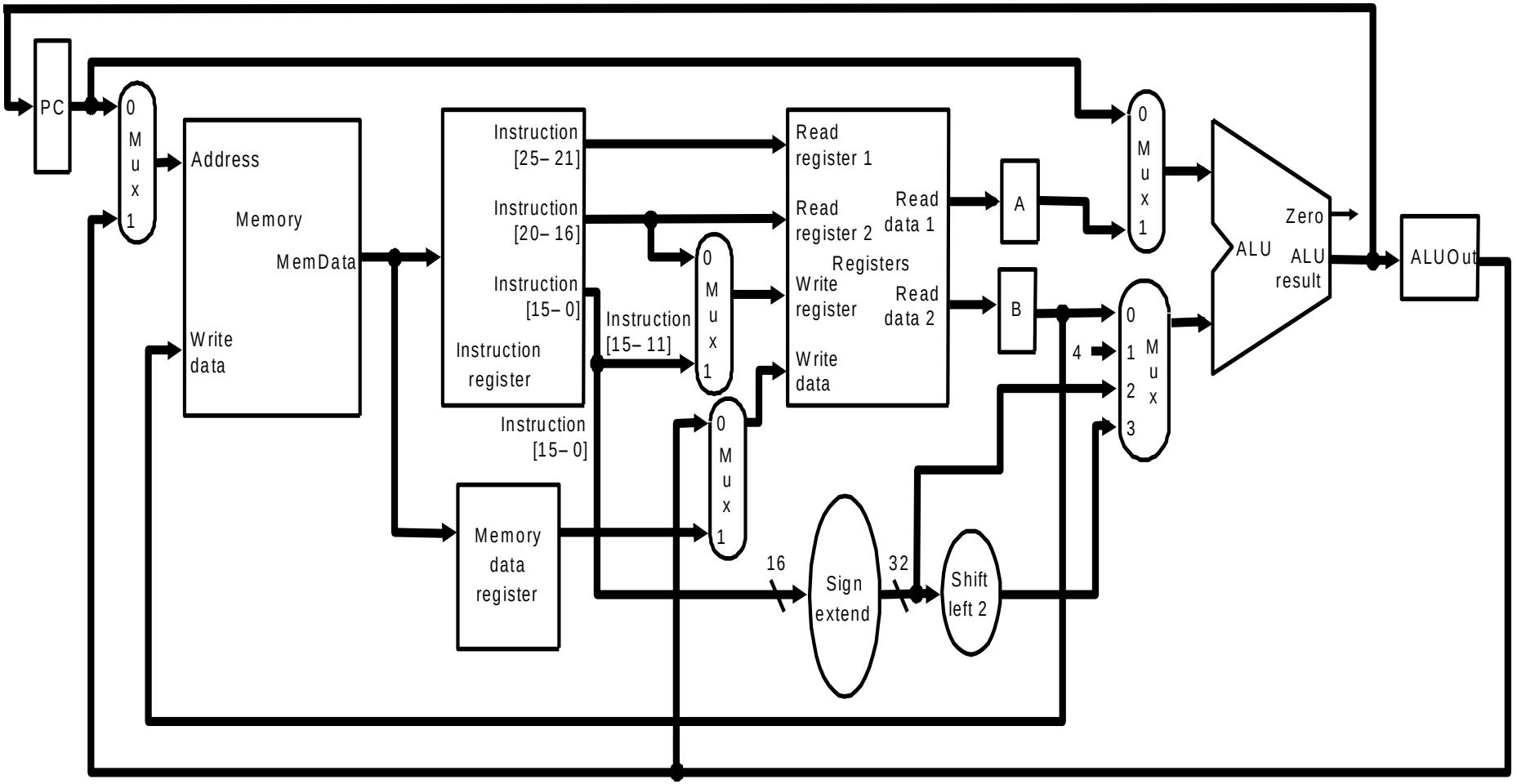
Multicycle Approach

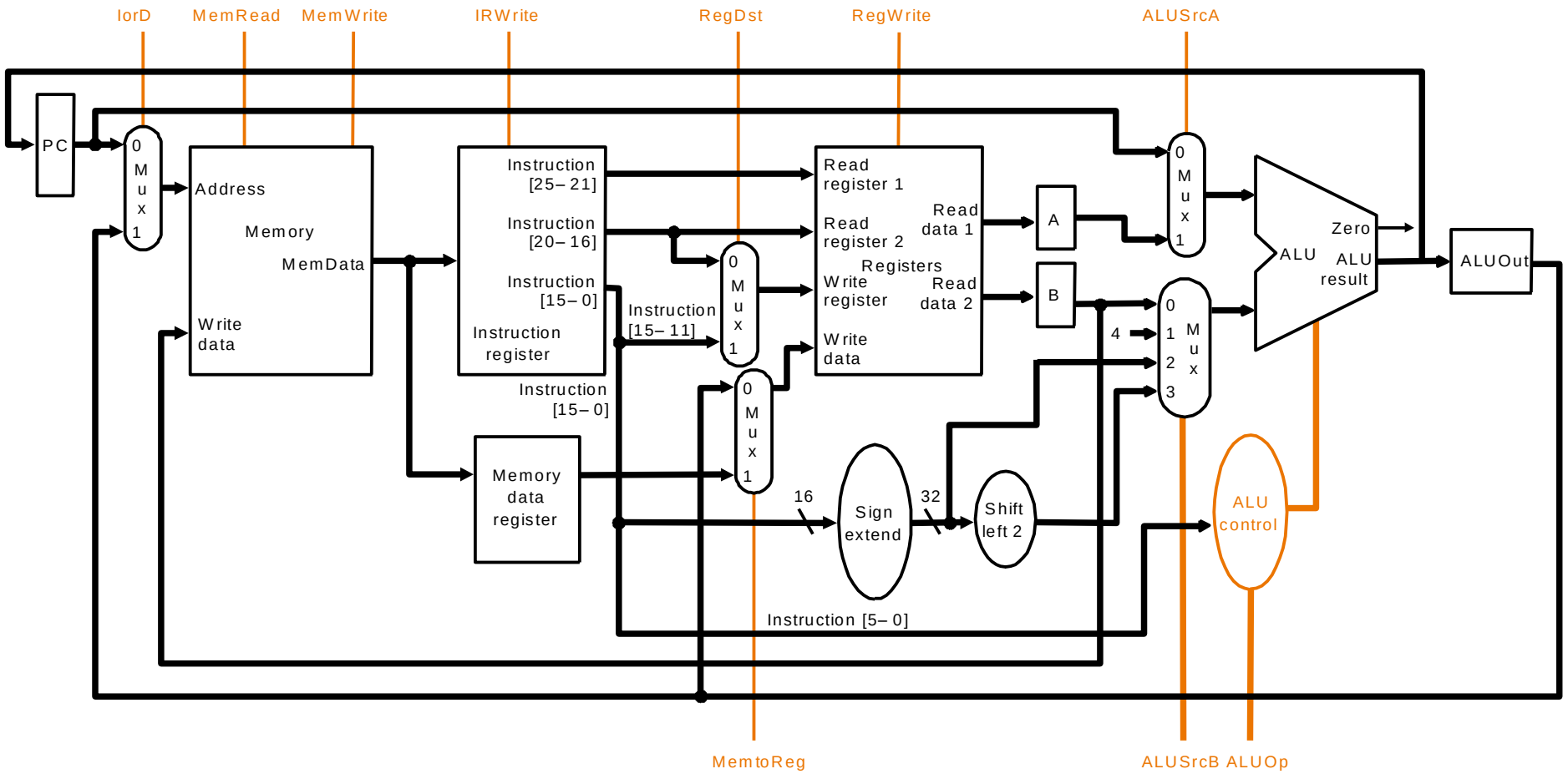
- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional internal registers

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– 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
ALU
Zero
ALU
result
ALUOut

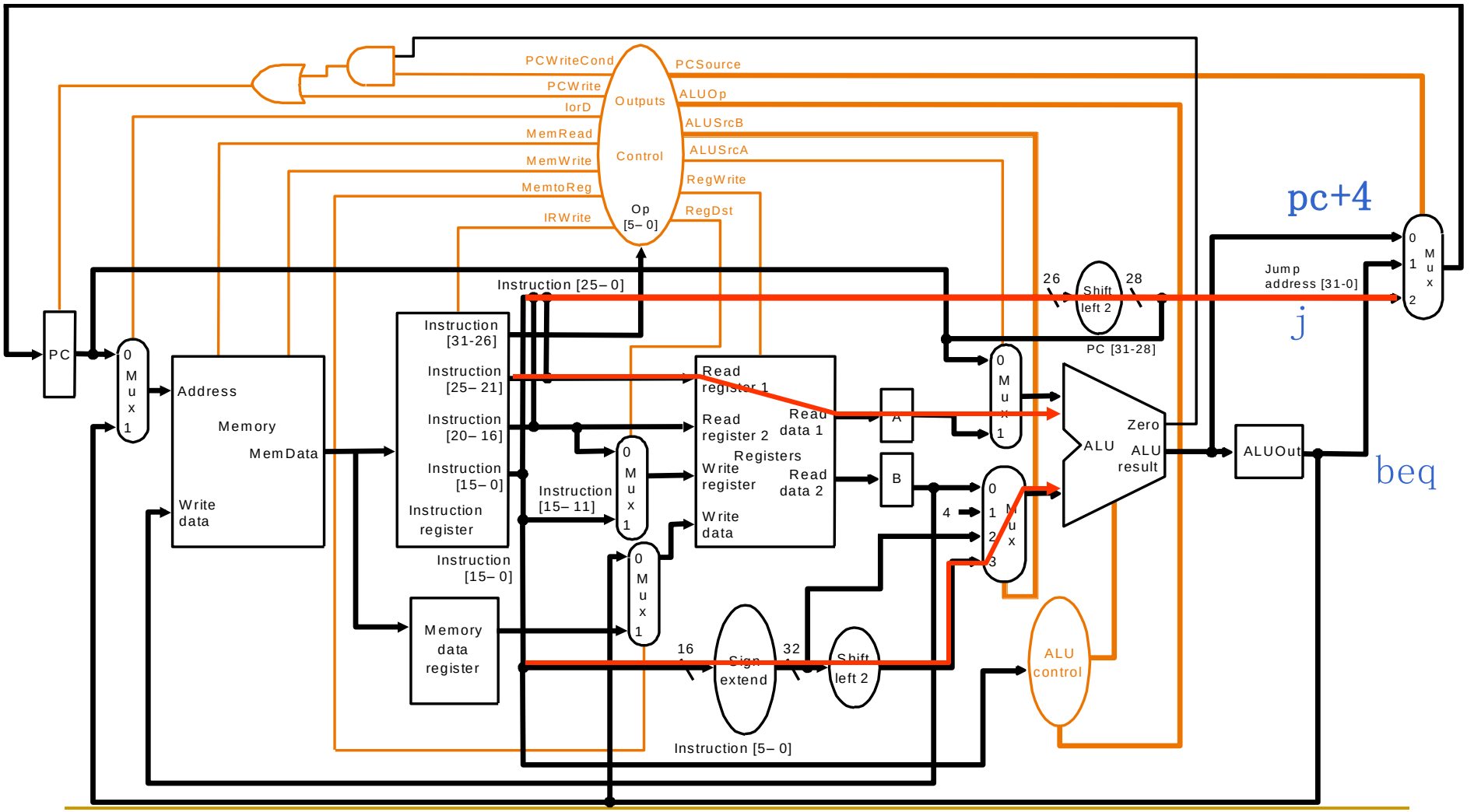

IorD
MemRead
MemWrite
IRWrite
RegDst
RegWrite
ALUSrcA
PC
0
M
u
x
1
Address
Memory
MemData
Write
data
Instruction
[25– 21]
Instruction
[20– 16]
Instruction
[15– 0]
Instruction
register
Instruction
[15– 0]
Instruction
[15– 11]
Memory
data
register
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
ALU
Zero
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

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

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

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

Computer Organization & Design

第07章：Pipeline

楼学庆

http://10.214.47.99/

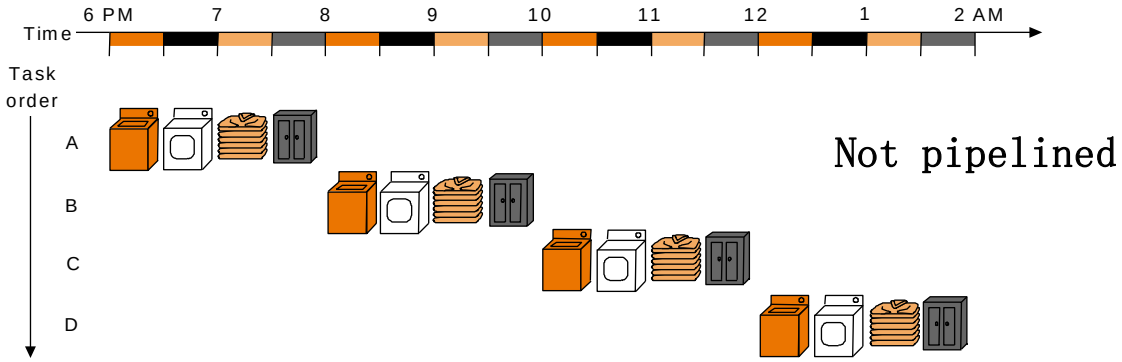
Email:hzlou@163.com

玉泉校区曹光彪东楼507室

Enhancing Performance with Pipelining

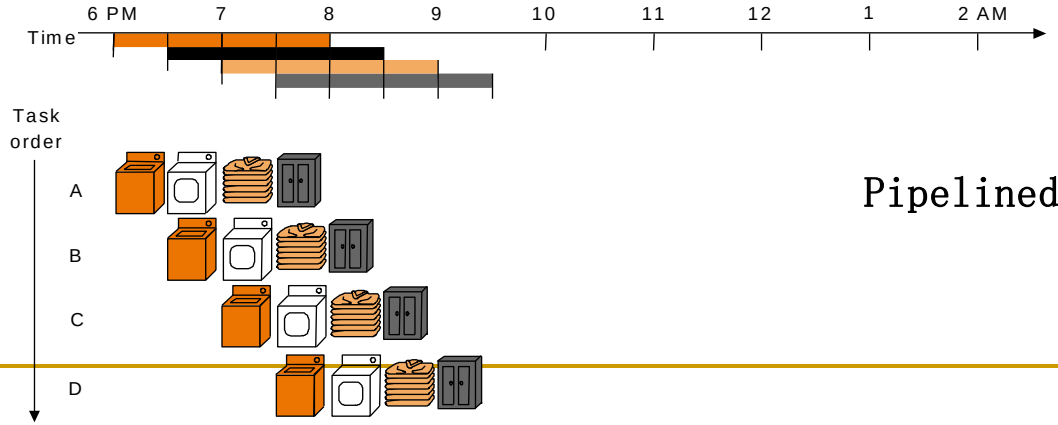
Pipelining

- **Start work ASAP!! Do not waste time!**



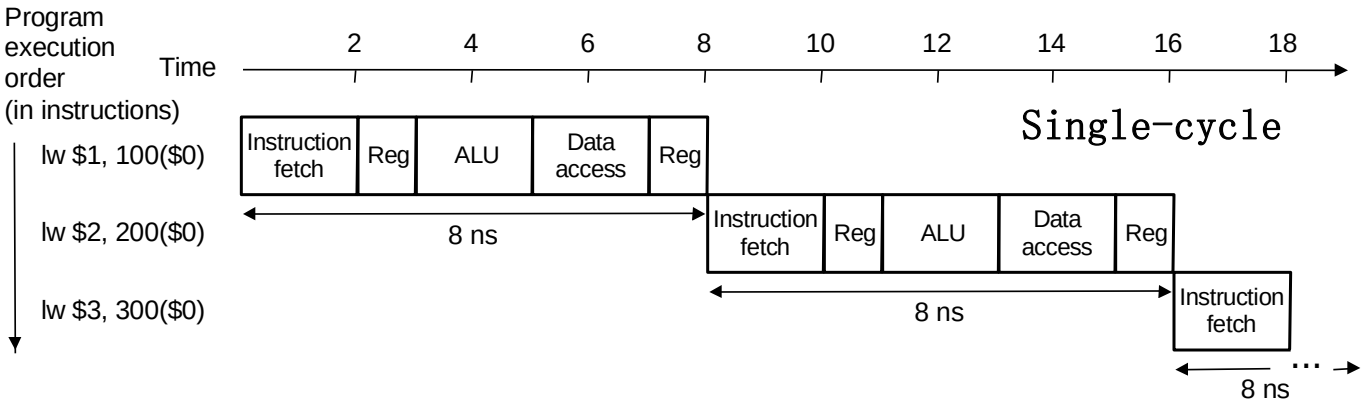
Not pipelined

Assume 30 min. each task - wash, dry, fold, store - and that separate tasks use separate hardware and so can be overlapped

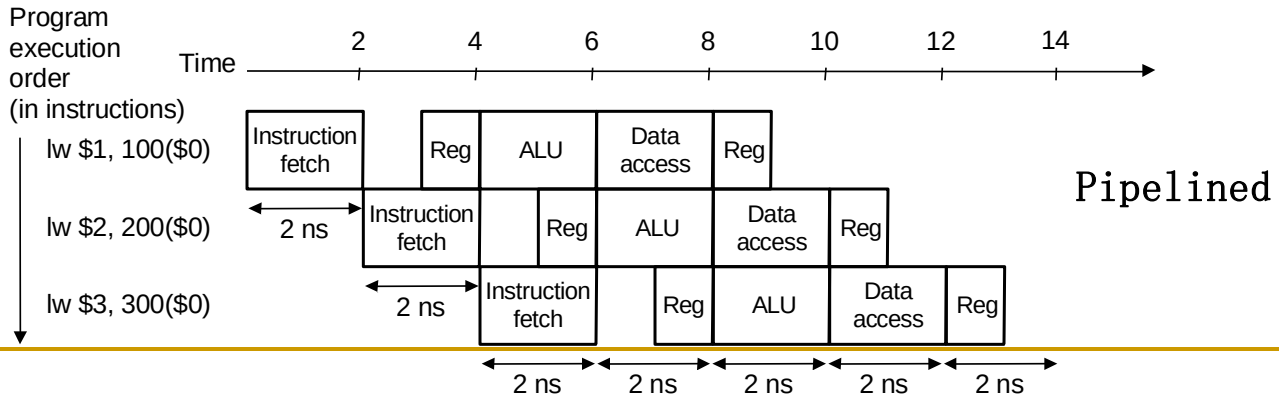


Pipelined

Pipelined vs. Single-Cycle Instruction



Assume 2 ns for memory access, ALU operation; 1 ns for register access: therefore, single cycle clock 8 ns; pipelined clock cycle 2 ns.



Pipelining: Keep in Mind

- Pipelining *does not reduce latency* of a single task, it *increases throughput* of entire workload
- Pipeline rate *limited by longest stage*
  - *potential* speedup = number pipe stages
  - *unbalanced lengths* of pipe stages reduces speedup
- Time to *fill* pipeline and time to *drain* it – when there is *slack* in the pipeline – reduces speedup

Example Problem

- ***Problem: for the laundry fill in the following table when***
  1. ***the stage lengths are 30, 30, 30 30 min., resp.***
  2. ***the stage lengths are 20, 20, 60, 20 min., resp.***

Person	Unpipelined finish time	Pipeline 1 finish time	Ratio unpipelined to pipeline 1
1			
2			
3			
4			
n			

- ***Come up with a formula for pipeline speed-up!***

Pipelining MIPS

- What makes it easy with MIPS?
  - all *instructions are same length*
    - so fetch and decode stages are similar for all instructions
  - just a *few instruction formats*
    - simplifies instruction decode and makes it possible in one stage
  - *memory operands appear only in load/stores*
    - so memory access can be deferred to exactly one later stage
  - *operands are aligned in memory*
    - one data transfer instruction requires one memory access stage

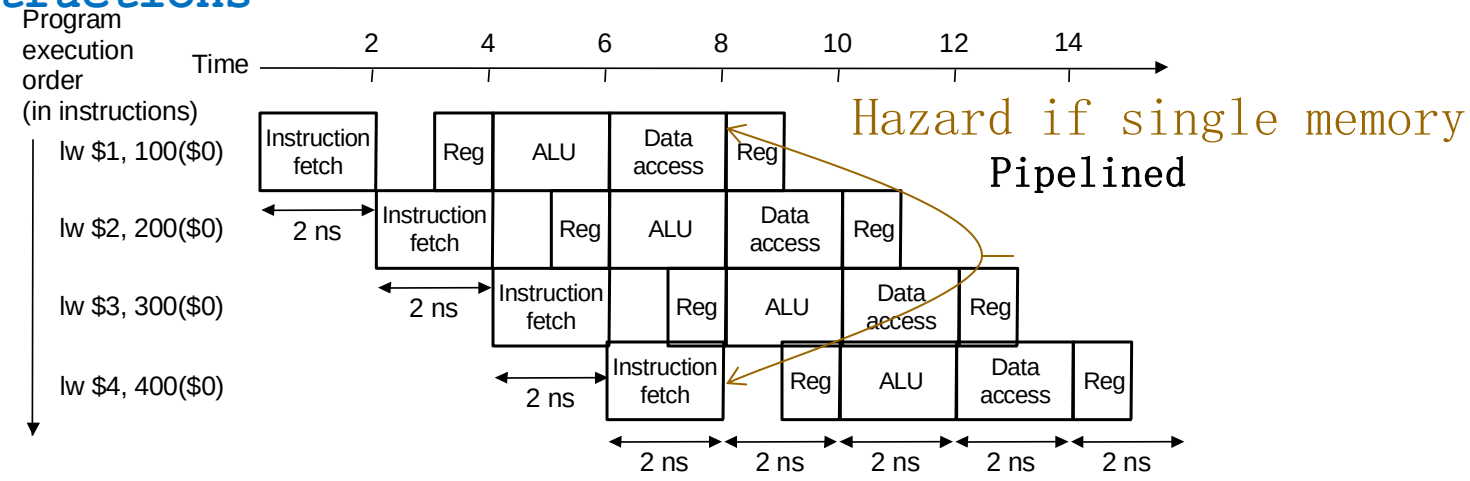
Pipelining MIPS

- What makes it hard?
  - *structural hazards*: different instructions, at different stages, in the pipeline want to use the same hardware resource
  - *control hazards*: succeeding instruction, to put into pipeline, depends on the outcome of a previous branch instruction, already in pipeline
  - *data hazards*: an instruction in the pipeline requires data to be computed by a previous instruction still in the pipeline

- Before actually building the pipelined datapath and control we first briefly examine these potential hazards individually…

Structural Hazards

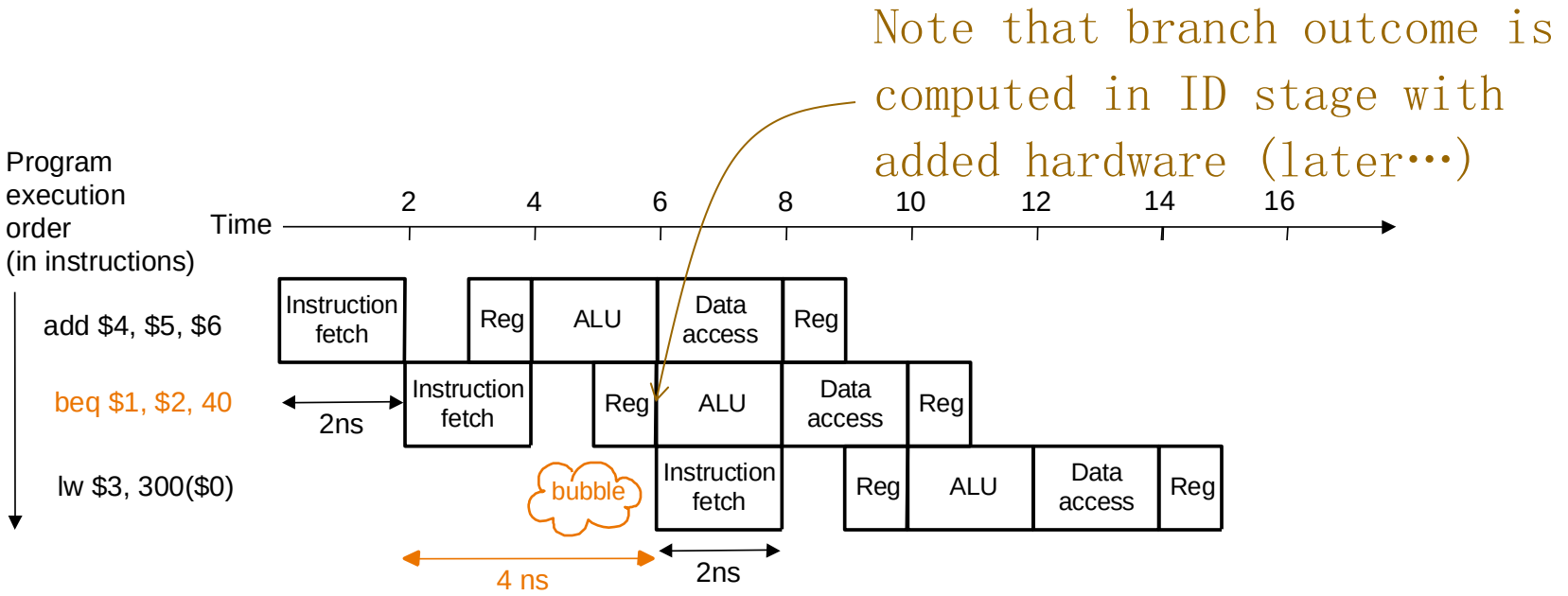
- *Structural hazard*: inadequate hardware to simultaneously support all instructions in the pipeline in the same clock cycle
- E.g., suppose *single – not separate* – instruction and data memory in pipeline below with *one read port*
  - then a structural hazard between first and fourth lw instructions



- *MIPS was designed to be pipelined*: structural hazards are easy to avoid!

Control Hazards

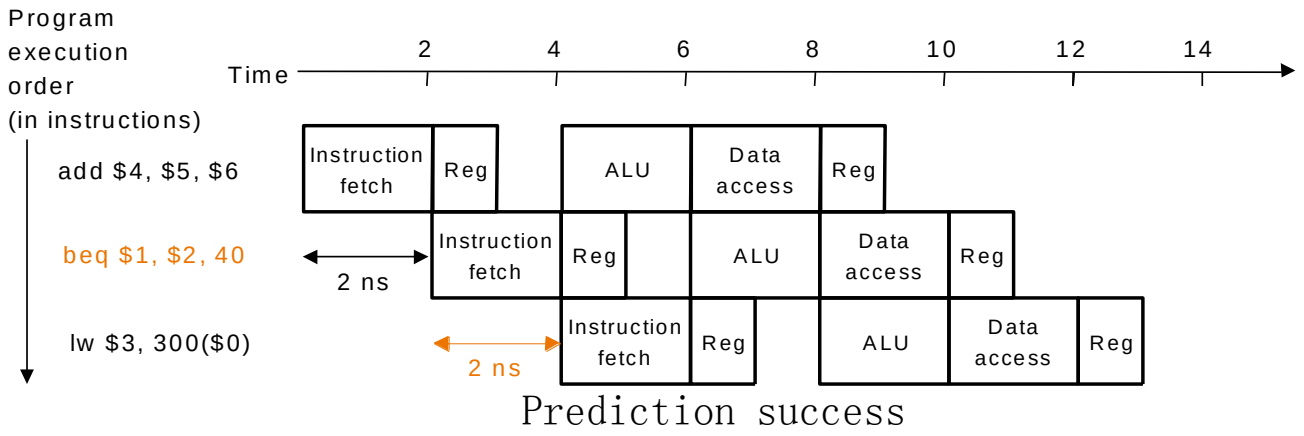
- ***Control hazard***: **need to make a decision based on the result of a previous instruction still executing in pipeline**
- **<u>Solution 1</u> *Stall* the pipeline**



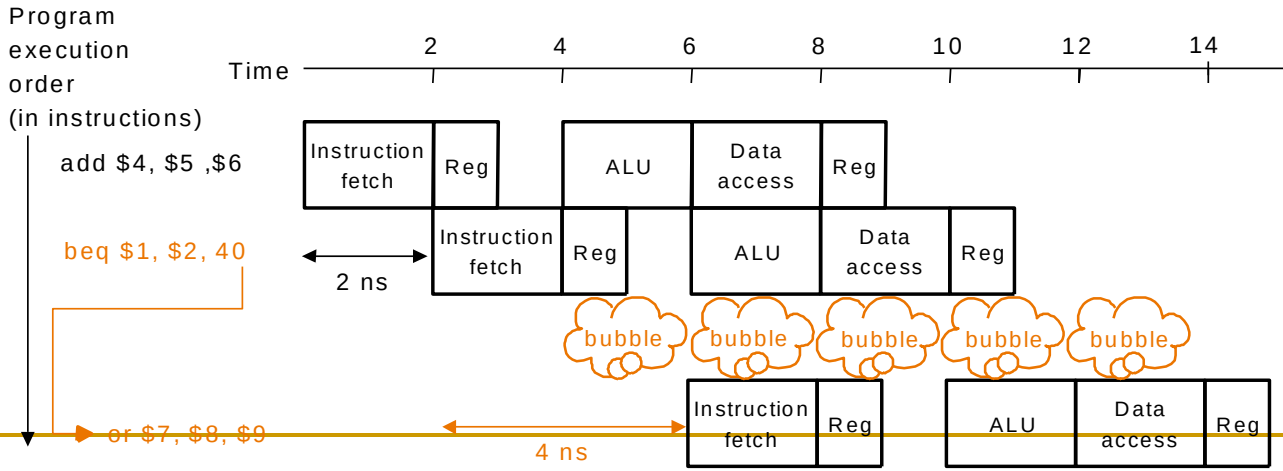
Pipeline stall

Control Hazards

- **Solution 2 *Predict* branch outcome**
  - **e.g., predict *branch-not-taken* :**



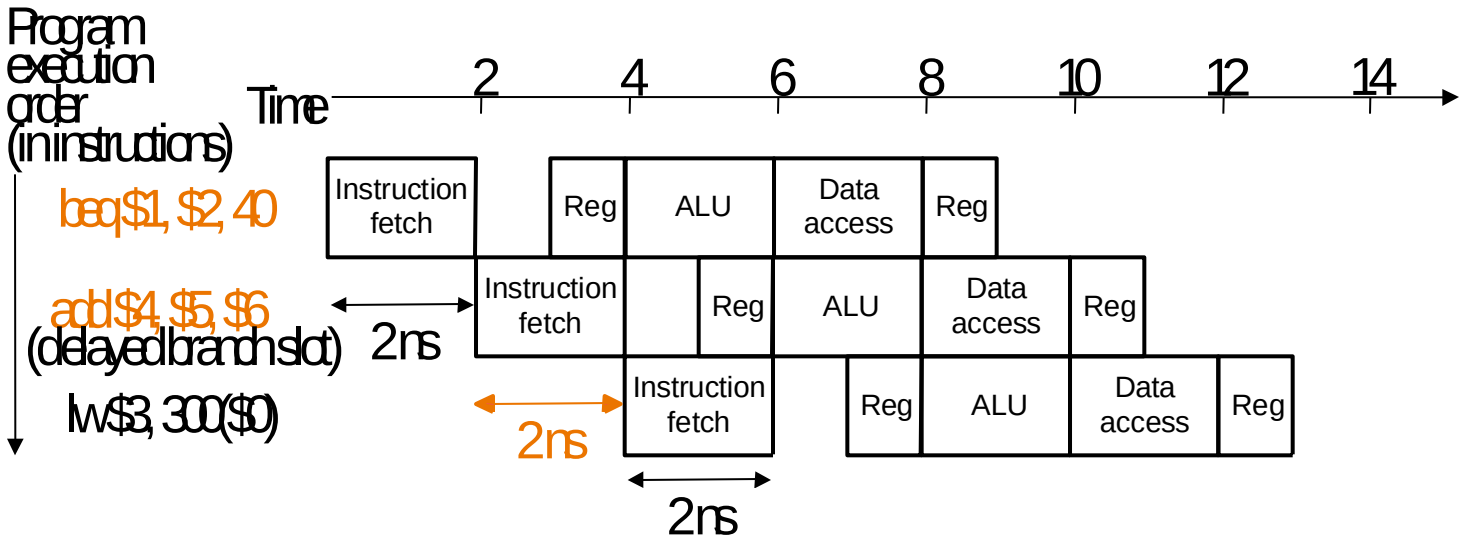
Prediction success



Prediction failure: undo (=flush) lw

Control Hazards

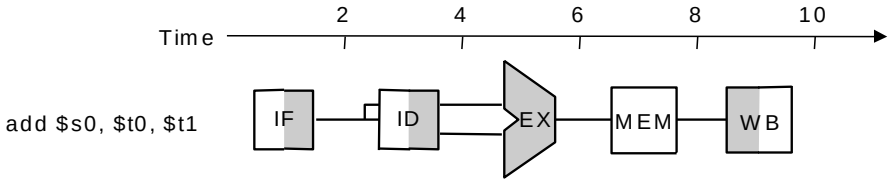
- **Solution 3** ***Delayed branch:*** **always execute the sequentially next statement with the branch executing after one instruction delay – compiler's job to find a statement that can be put in the slot that is independent of branch outcome**
  - **MIPS does this – but it is an option in SPIM (Simulator -> Settings)**



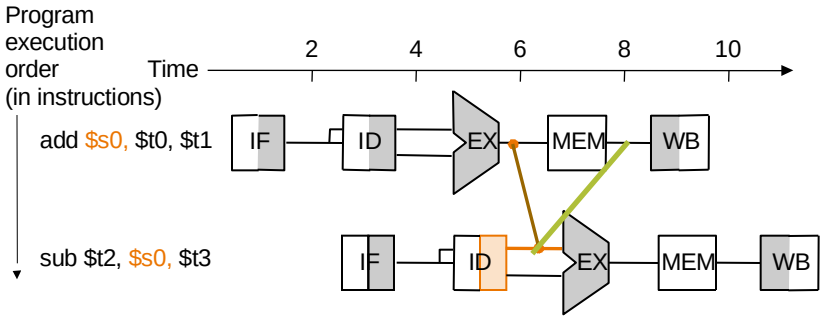
Delayed branch **beq** is followed by **add** that is independent of branch outcome

Data Hazards

- ***Data hazard*: instruction needs data from the result of a previous instruction still executing in pipeline**
- **<u>Solution</u> *Forward* data if possible…**



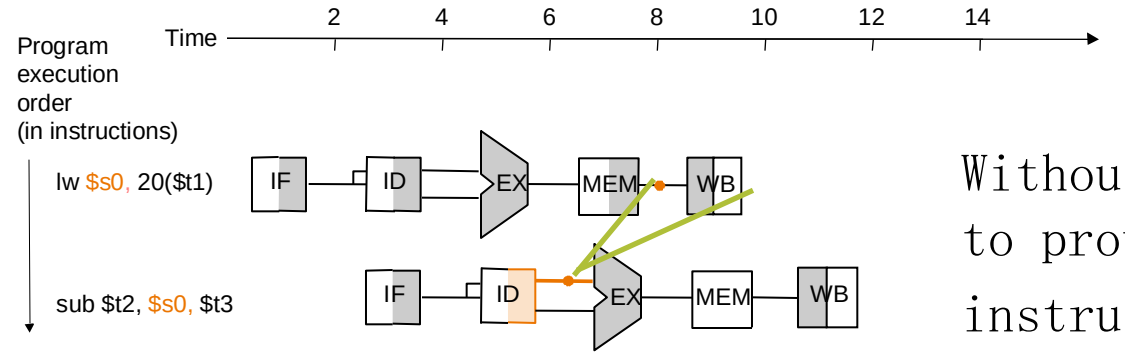
Instruction pipeline diagram: shade indicates use - left=write, right=read



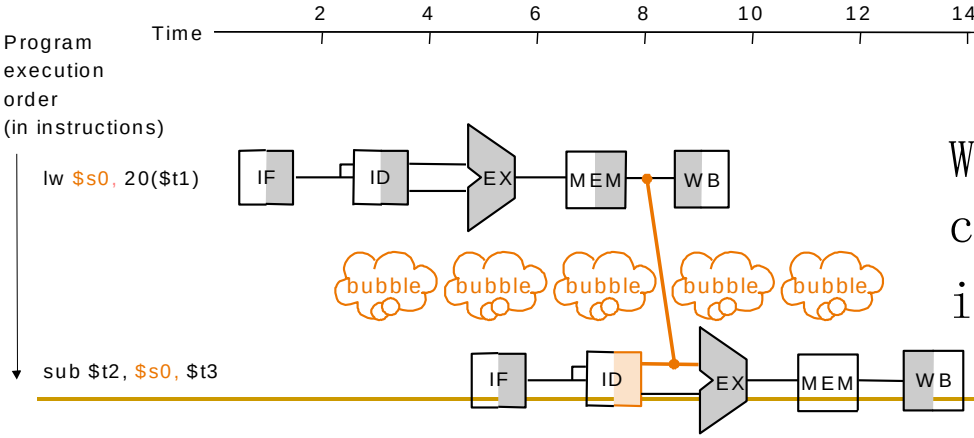
Without forwarding - blue line - data has to go back in time; with forwarding - red line - data is available in time

Data Hazards

- **Forwarding may not be enough**
  - **e.g., if an R-type instruction following a load uses the result of the load - called *load-use data hazard***



Without a stall it is impossible to provide input to the sub instruction in time



With a one-stage stall, forwarding can get the data to the sub instruction in time

Reordering Code to Avoid Pipeline Stall (Software Solution)

- **Example:**

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t2, 0($t1)
sw $t0, 4($t1)
```

Data hazard

- **Reordered code:**

```
lw $t0, 0($t1)
lw $t2, 4($t1)
sw $t0, 4($t1)
sw $t2, 0($t1)
```

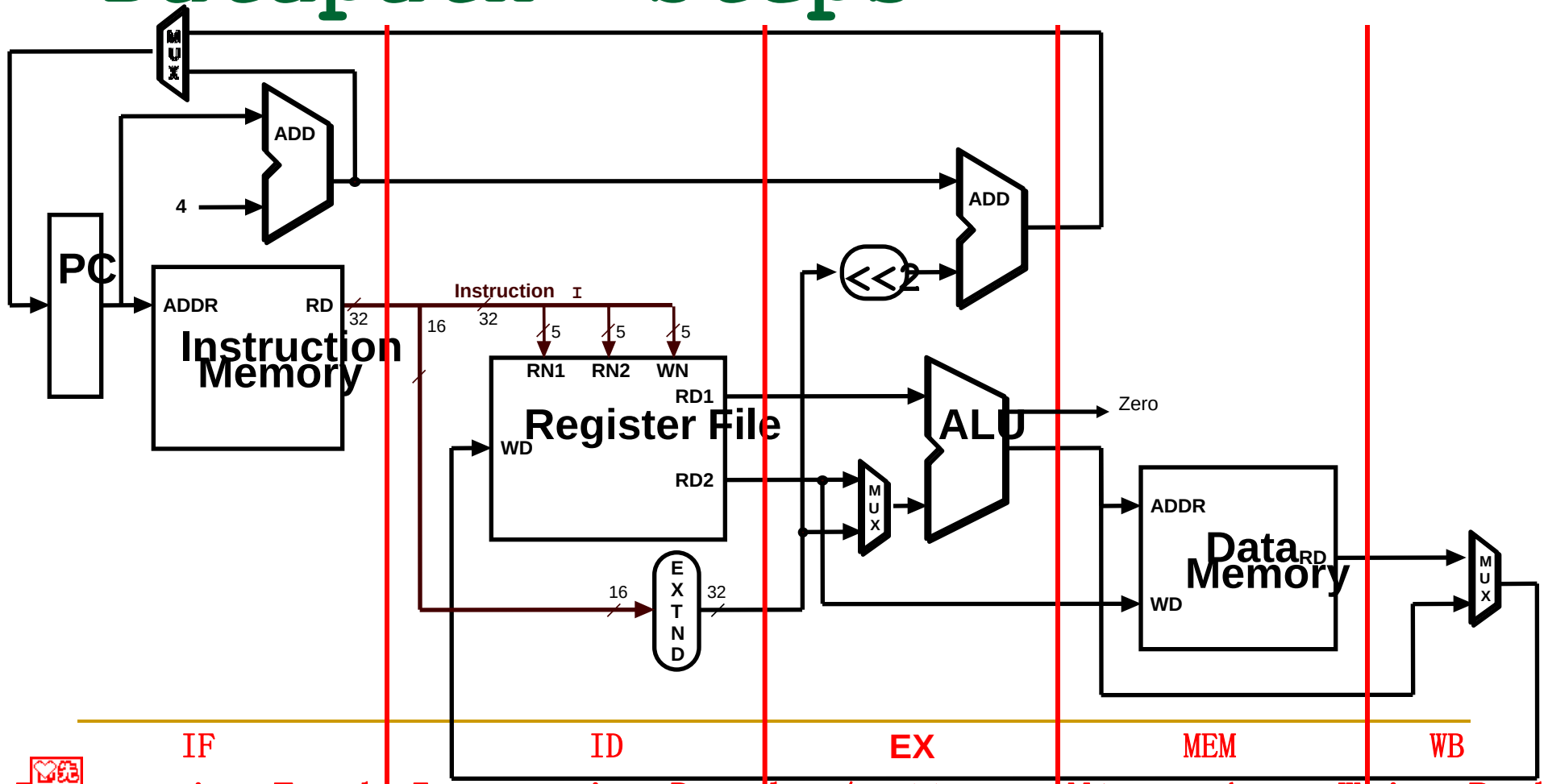
Interchanged

Pipelined Datapath

- We now move to actually building a pipelined datapath
- First recall the 5 steps in instruction execution
  1. Instruction Fetch & PC Increment (IF)
  2. Instruction Decode and Register Read (ID)
  3. Execution or calculate address (EX)
  4. Memory access (MEM)
  5. Write result into register (WB)
- Review: single-cycle processor
  - all 5 steps done in a single clock cycle
  - dedicated hardware required for each step

- *What happens if we break the execution into multiple cycles, but keep the extra hardware?*

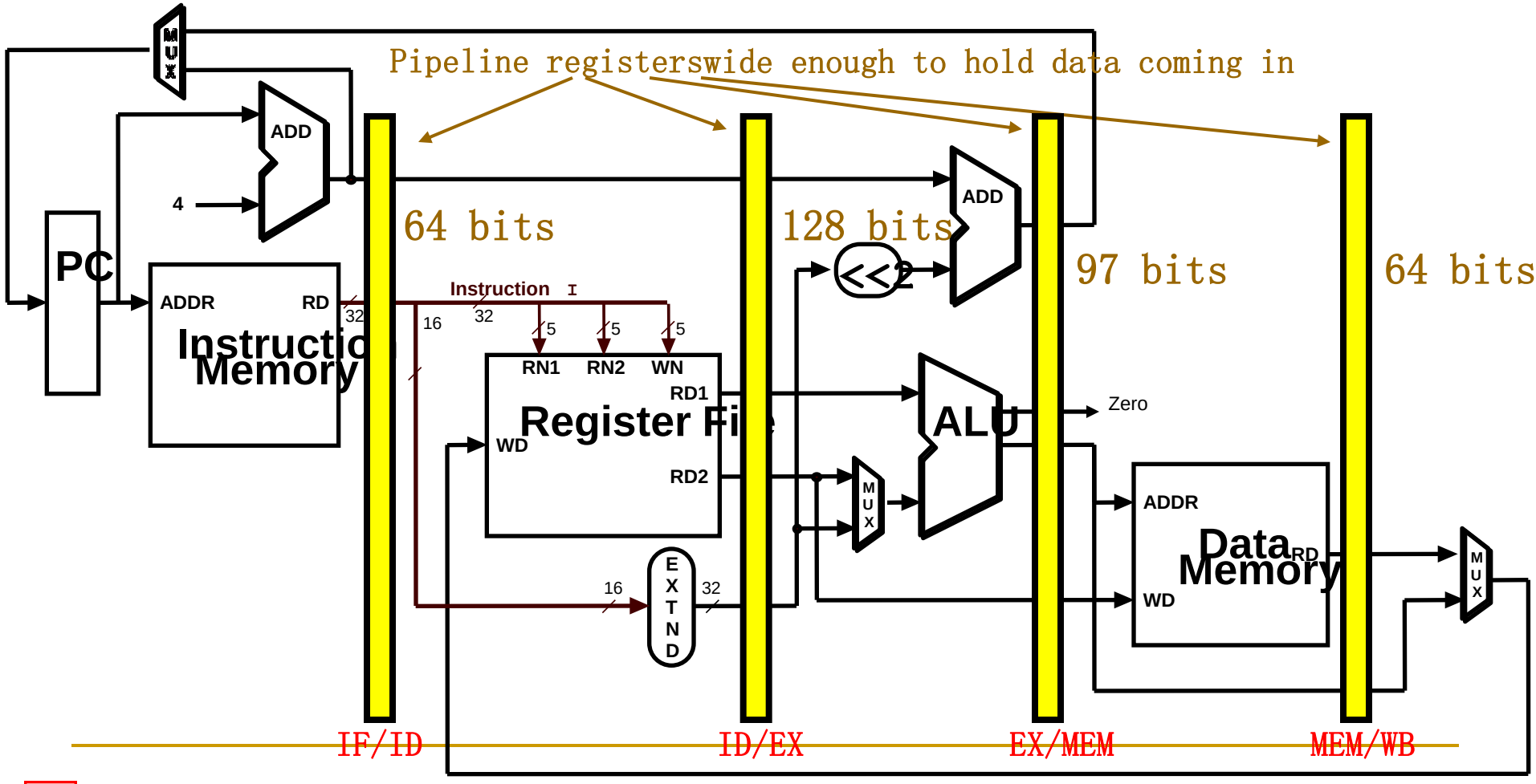
Review - Single-Cycle Datapath "Steps"



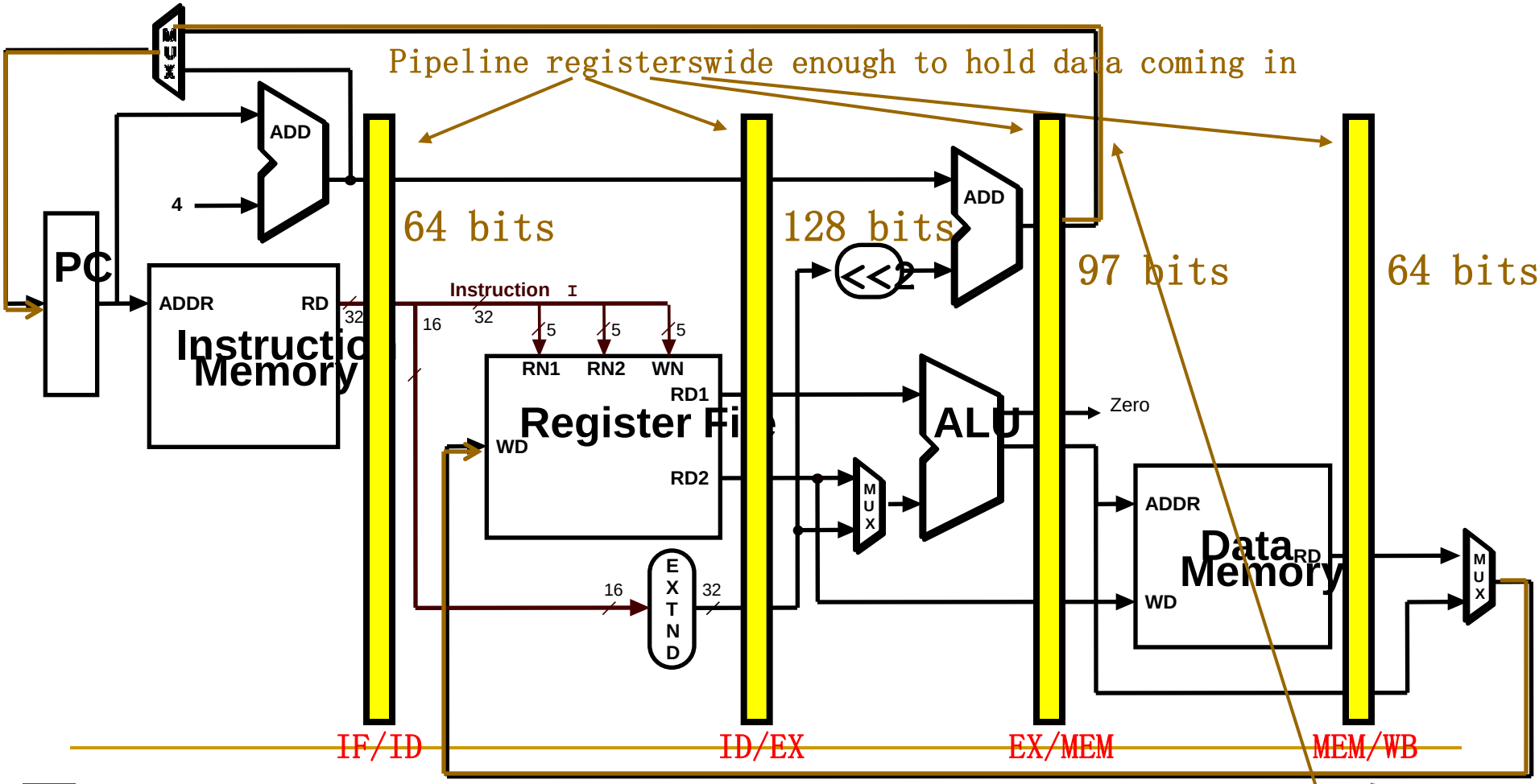
Pipelined Datapath – Key Idea

- ***What happens if we break the execution into multiple cycles, but keep the extra hardware?***
  - **Answer:** *We may be able to start executing a new instruction at each clock cycle* **- pipelining**
- **…but we shall need *extra* registers to hold data between cycles – *pipeline registers***

Pipelined Datapath

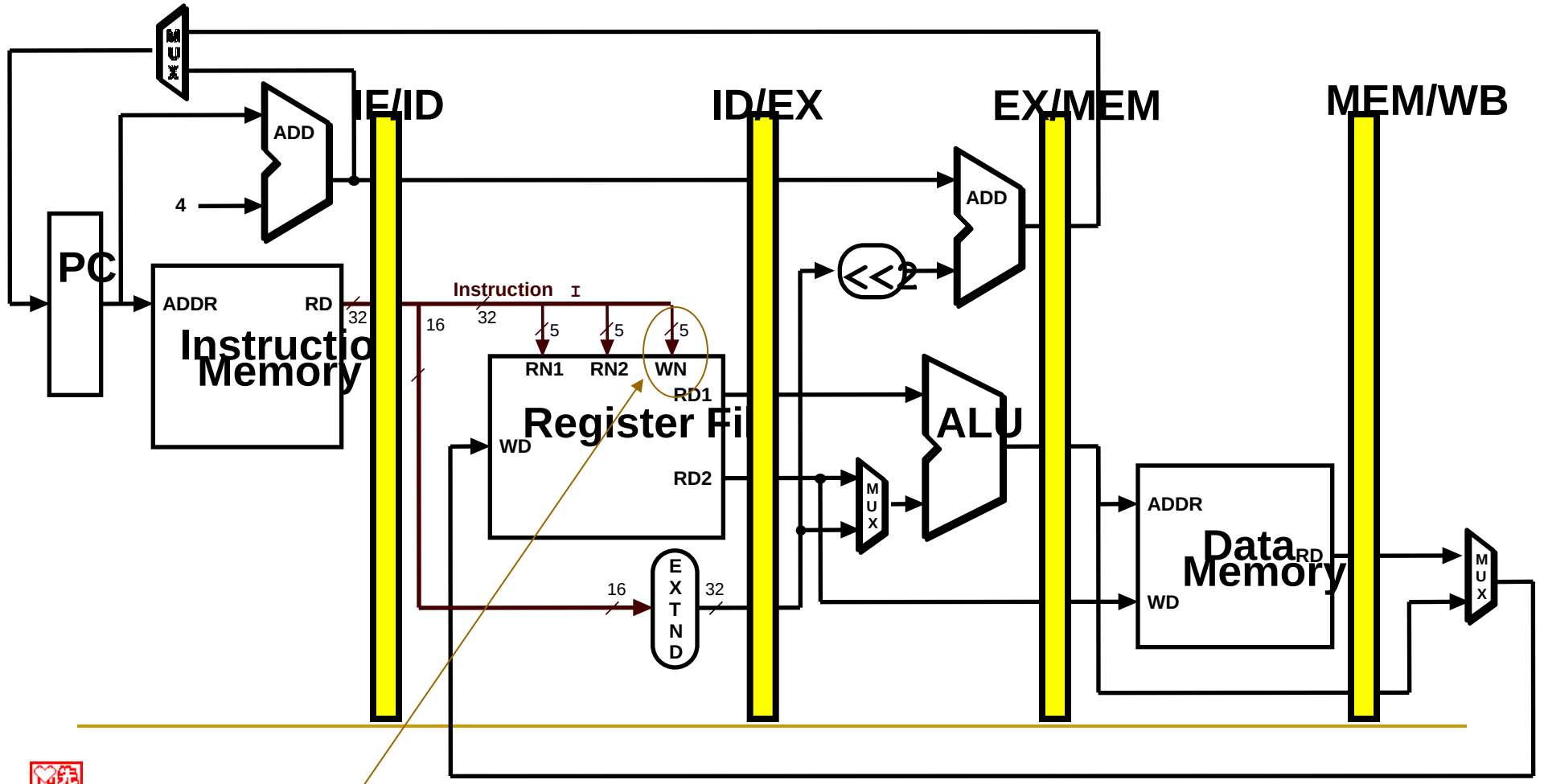


Pipelined Datapath



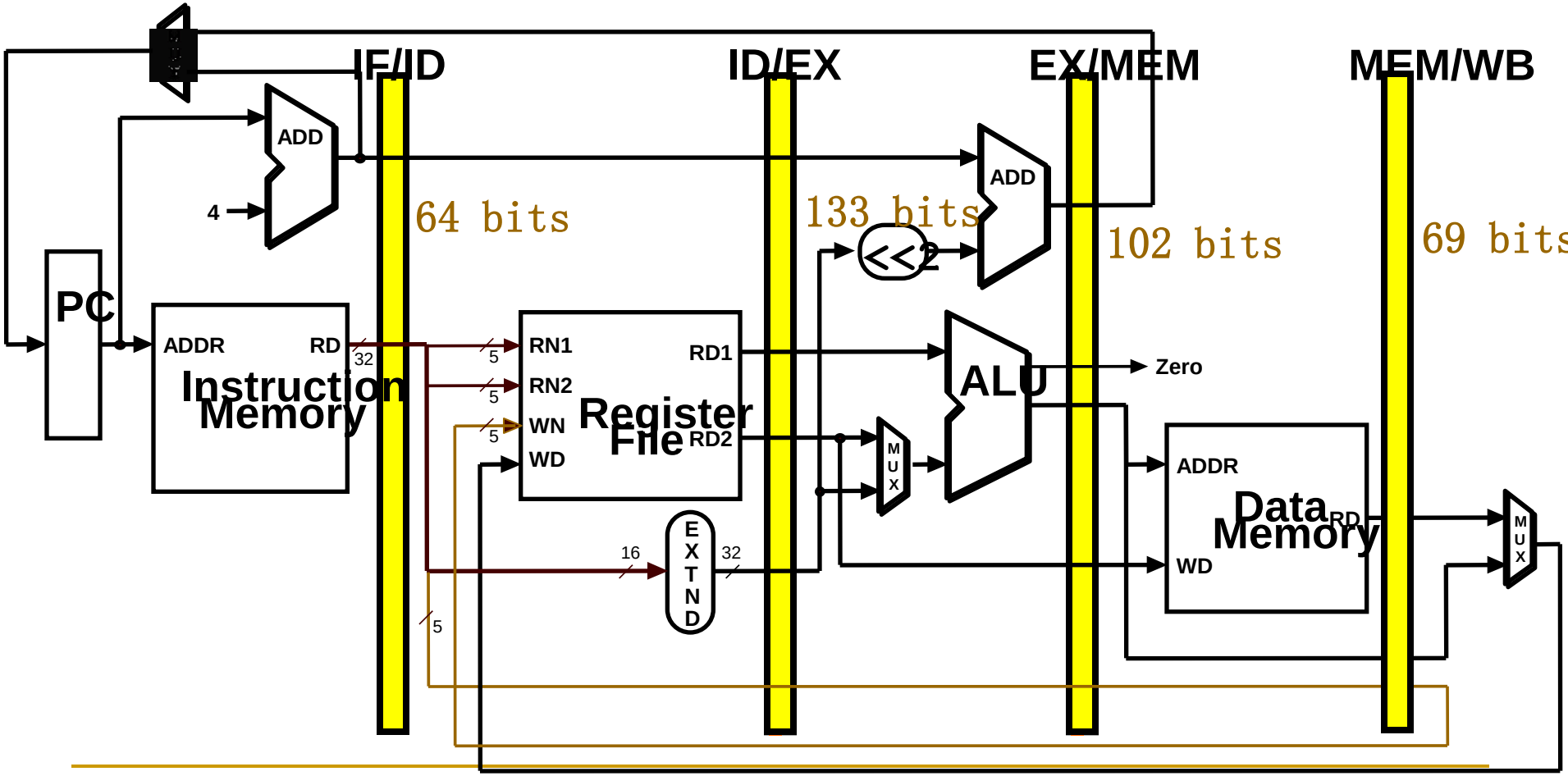
Only data flowing right to left may cause hazard…, why?

Bug in the Datapath



Write register number comes from another *later* instruction!

Corrected Datapath



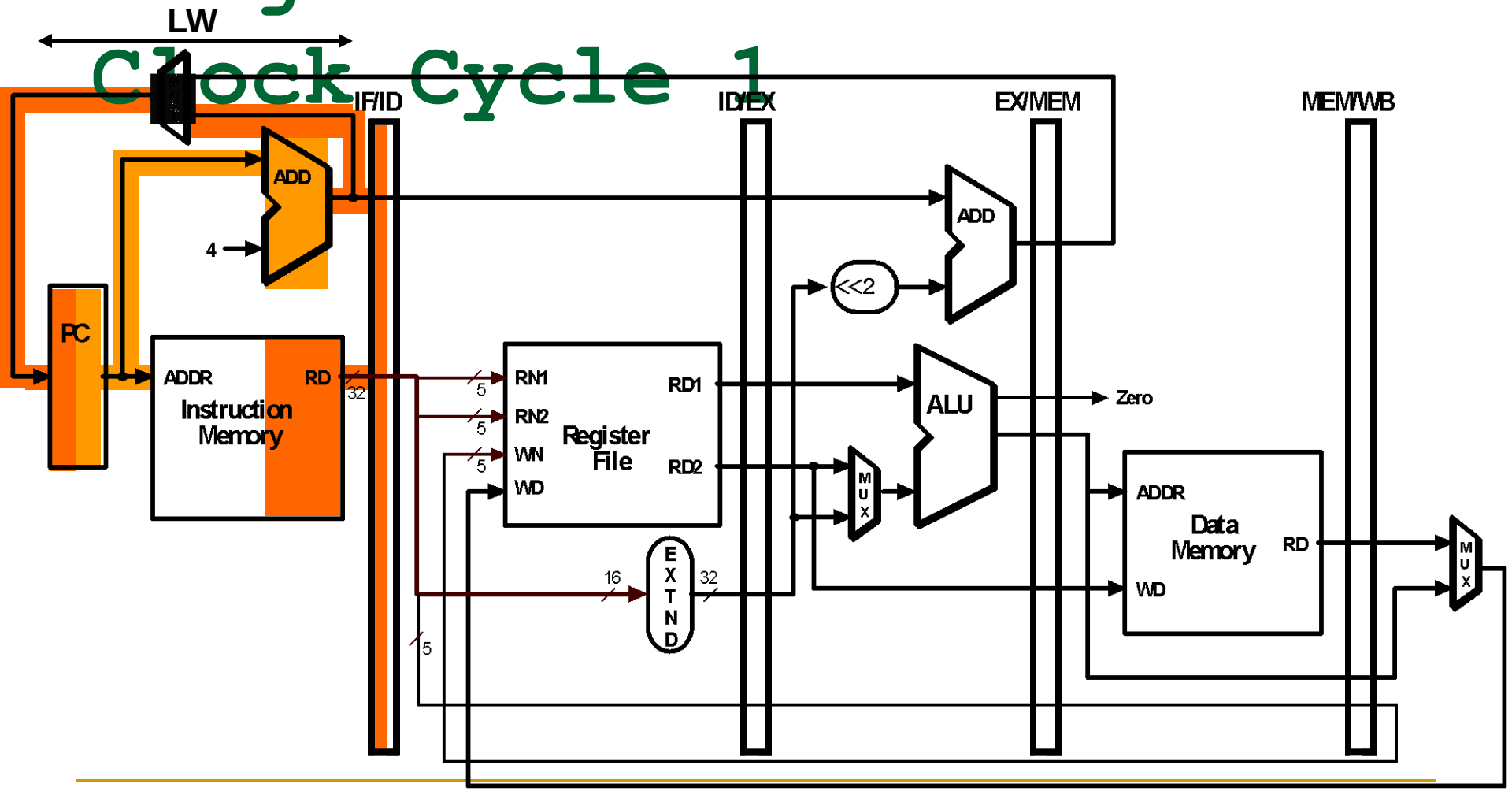
Destination register number is also passed through ID/EX, EX/MEM and MEM/WB registers, which are now wider by 5 bits

Pipelined Example

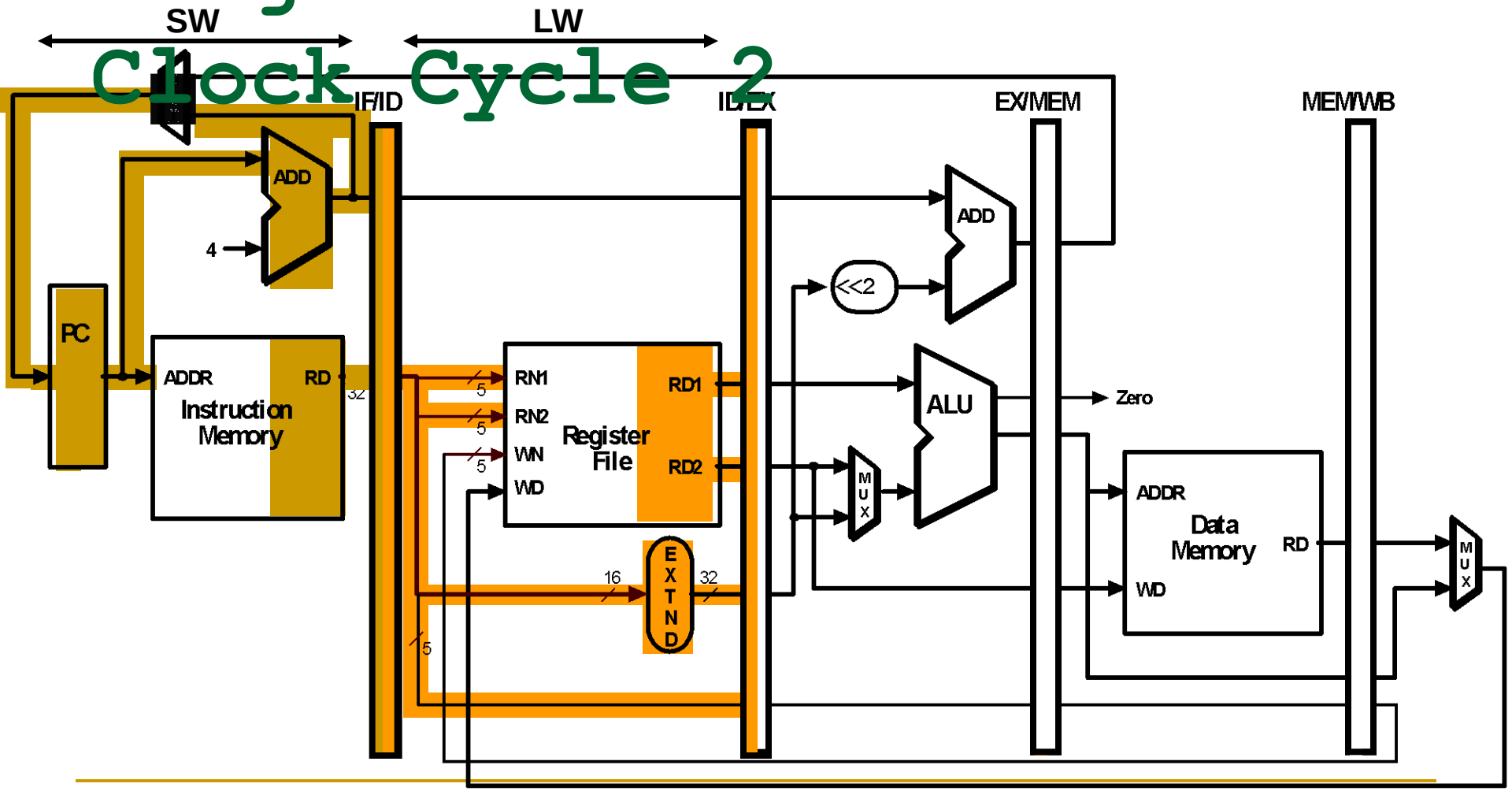
- Consider the following instruction sequence:

```
lw  $t0,  10($t1)
sw  $t3, 20($t4)
add $t5,  $t6,  $t7
sub $t8,  $t9,  $t10
```

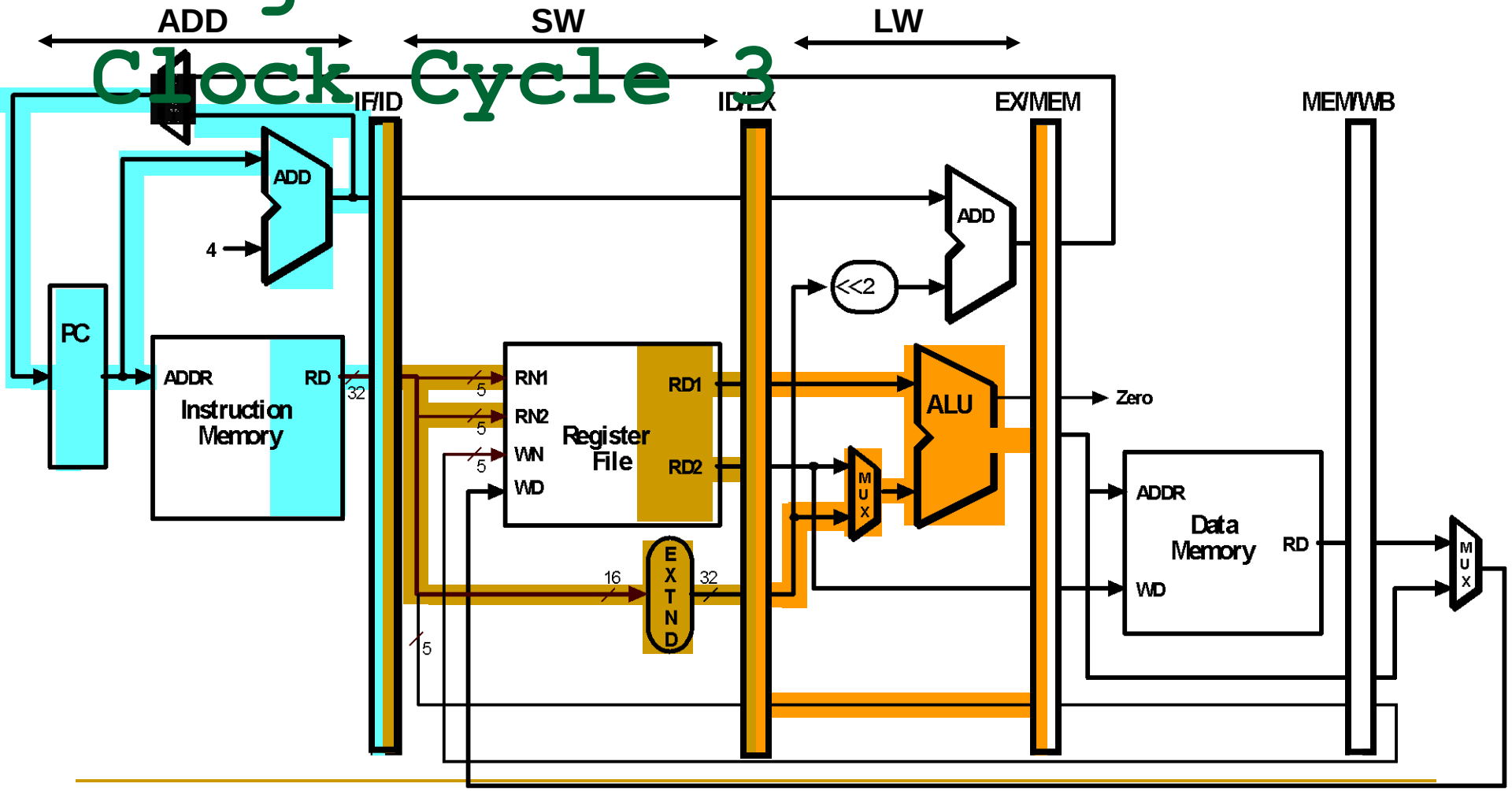
Single-Clock-Cycle Diagram:



Single-Clock-Cycle Diagram: Clock Cycle 2

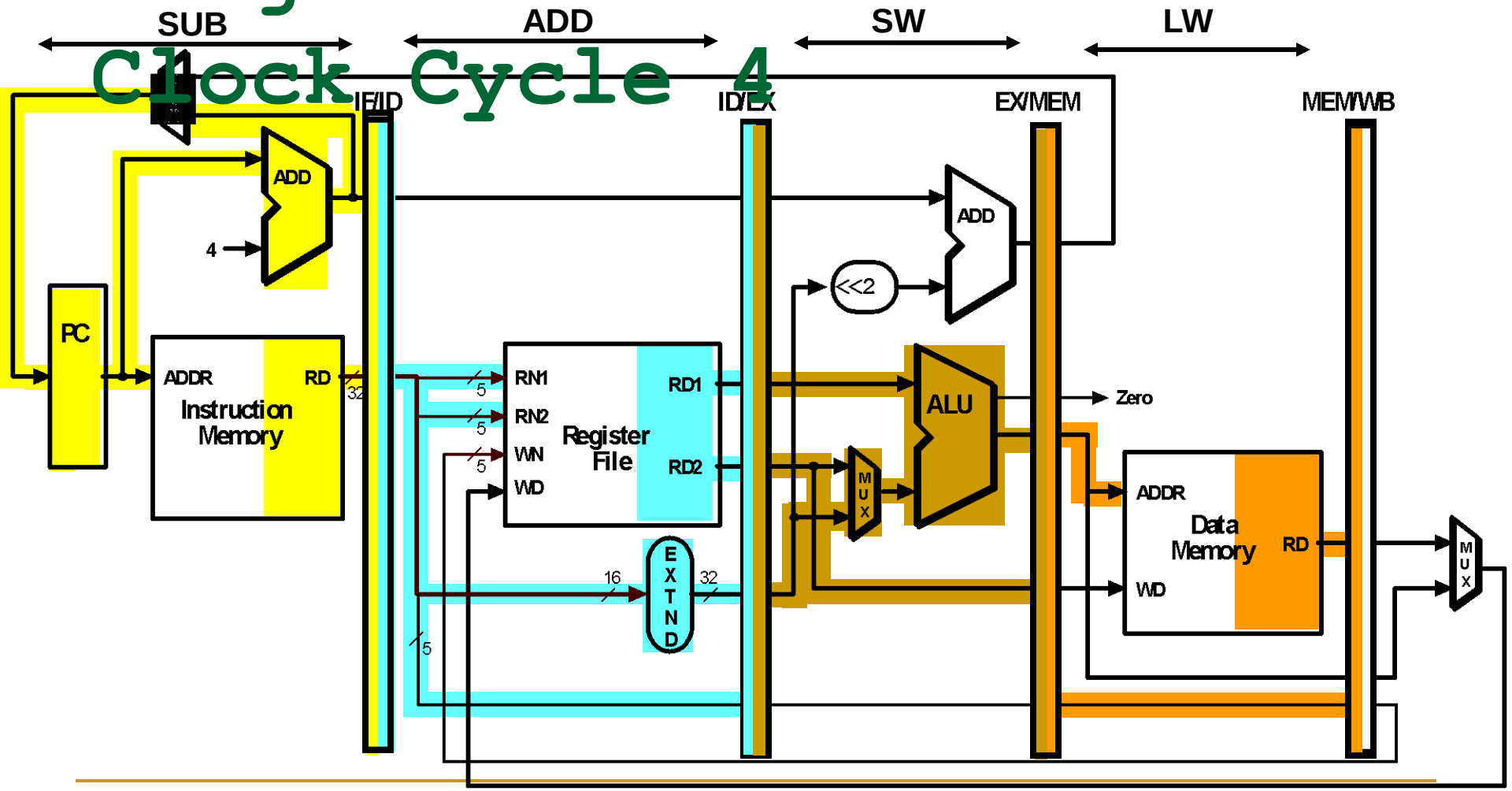


Single-Clock-Cycle Diagram: Clock Cycle 3

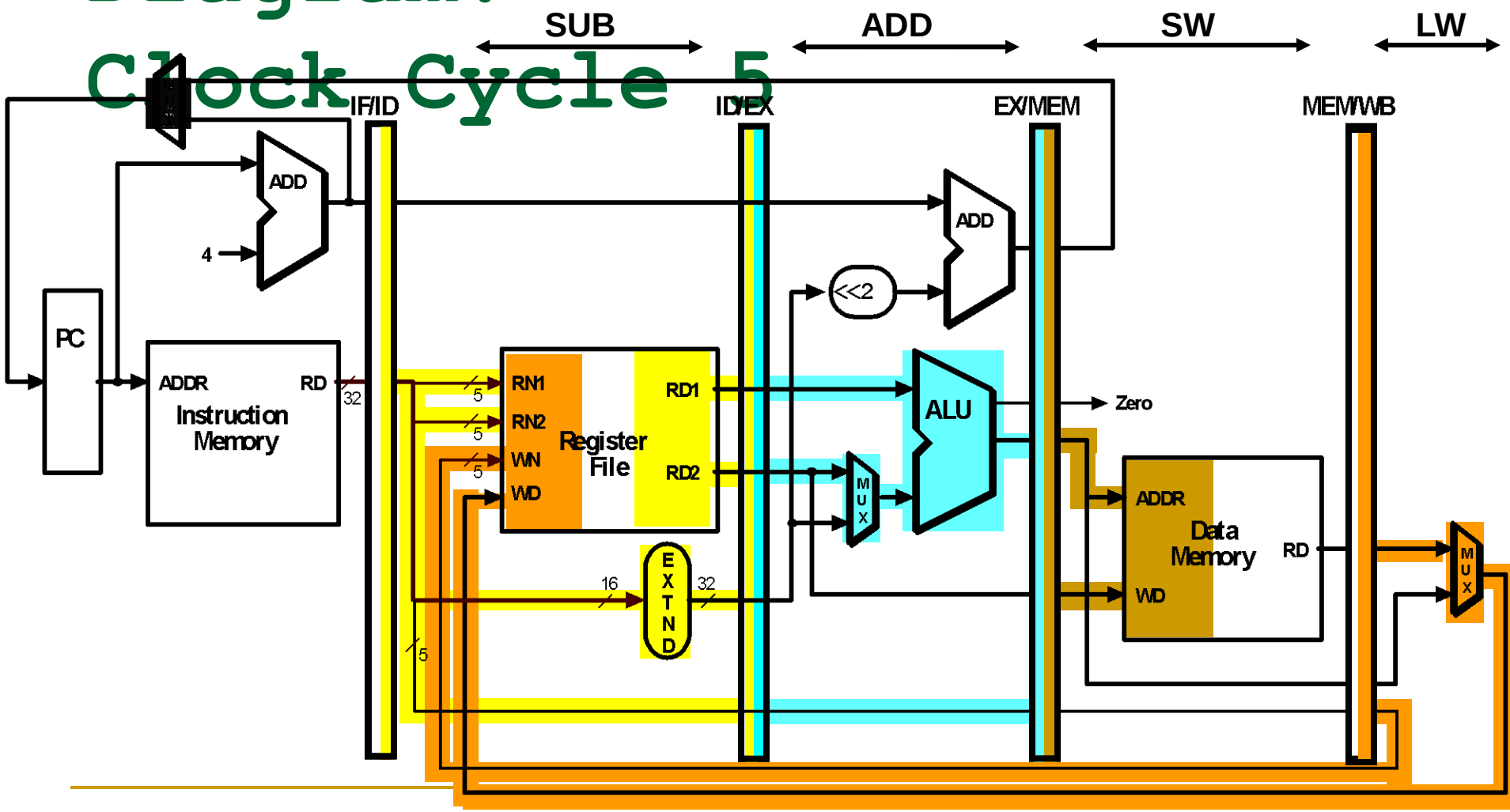


Single-Clock-Cycle Diagram:

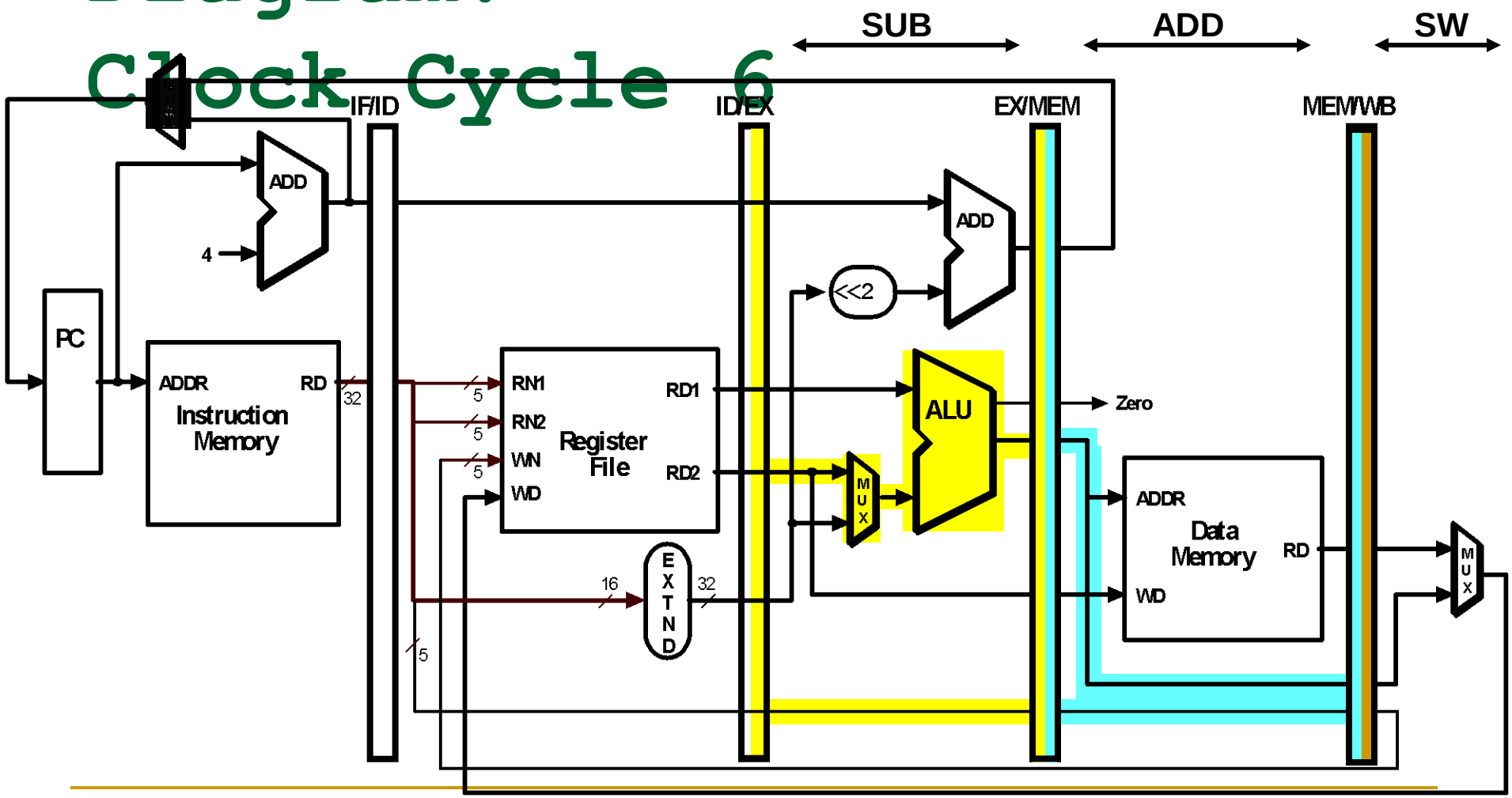
Clock Cycle 4



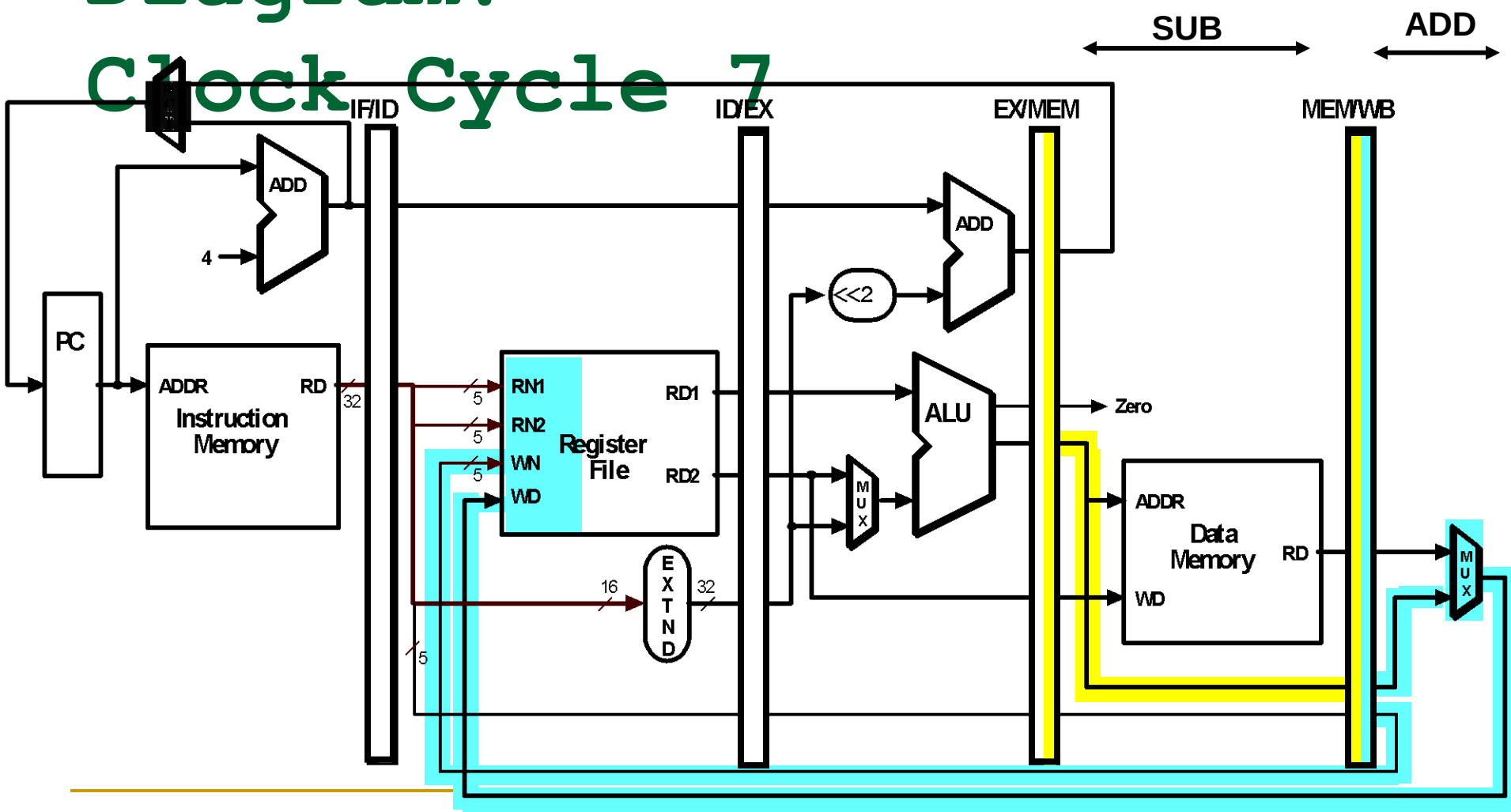
Single-Clock-Cycle Diagram: Clock Cycle 5



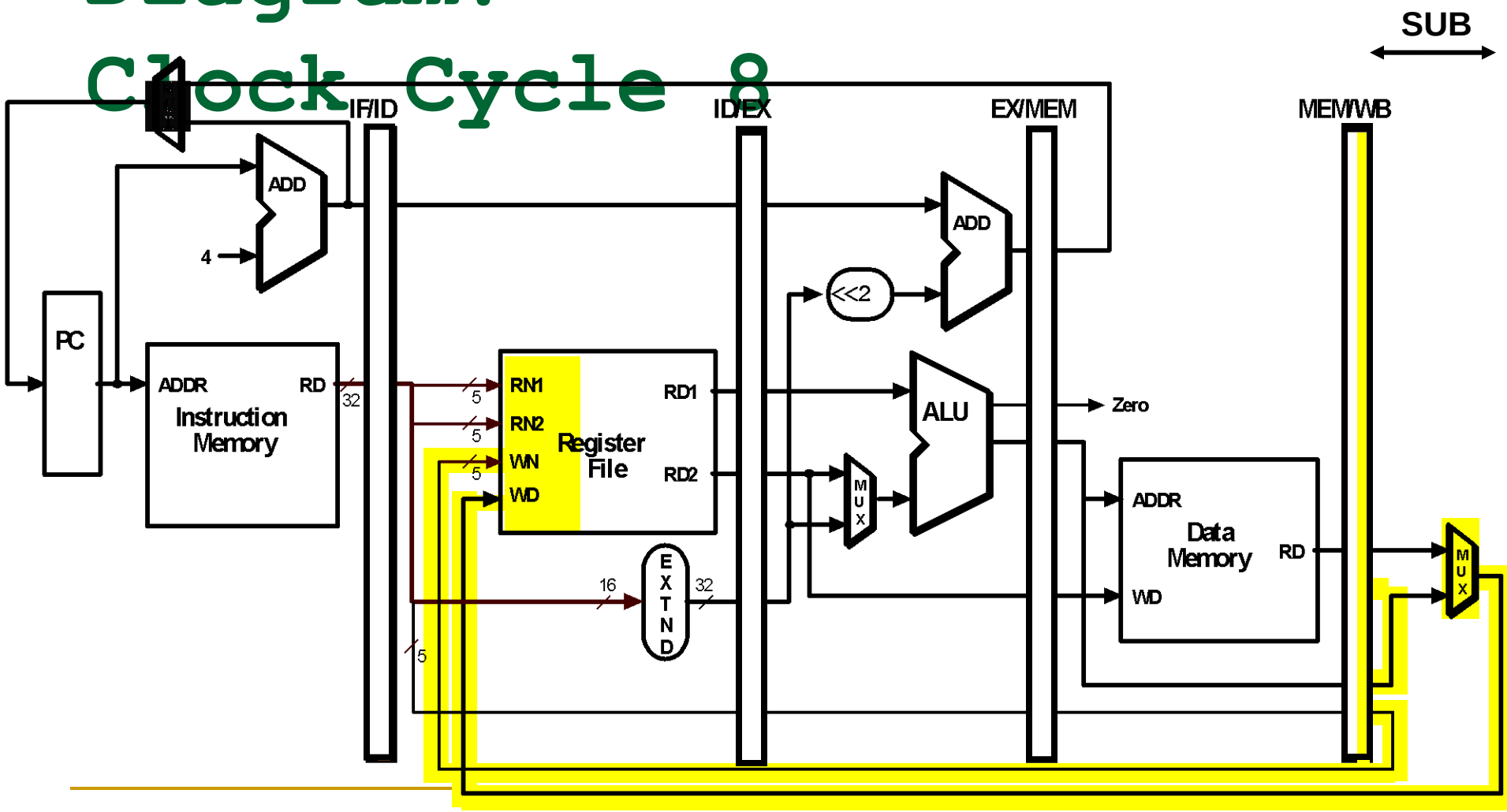
Single-Clock-Cycle Diagram: Clock Cycle 6



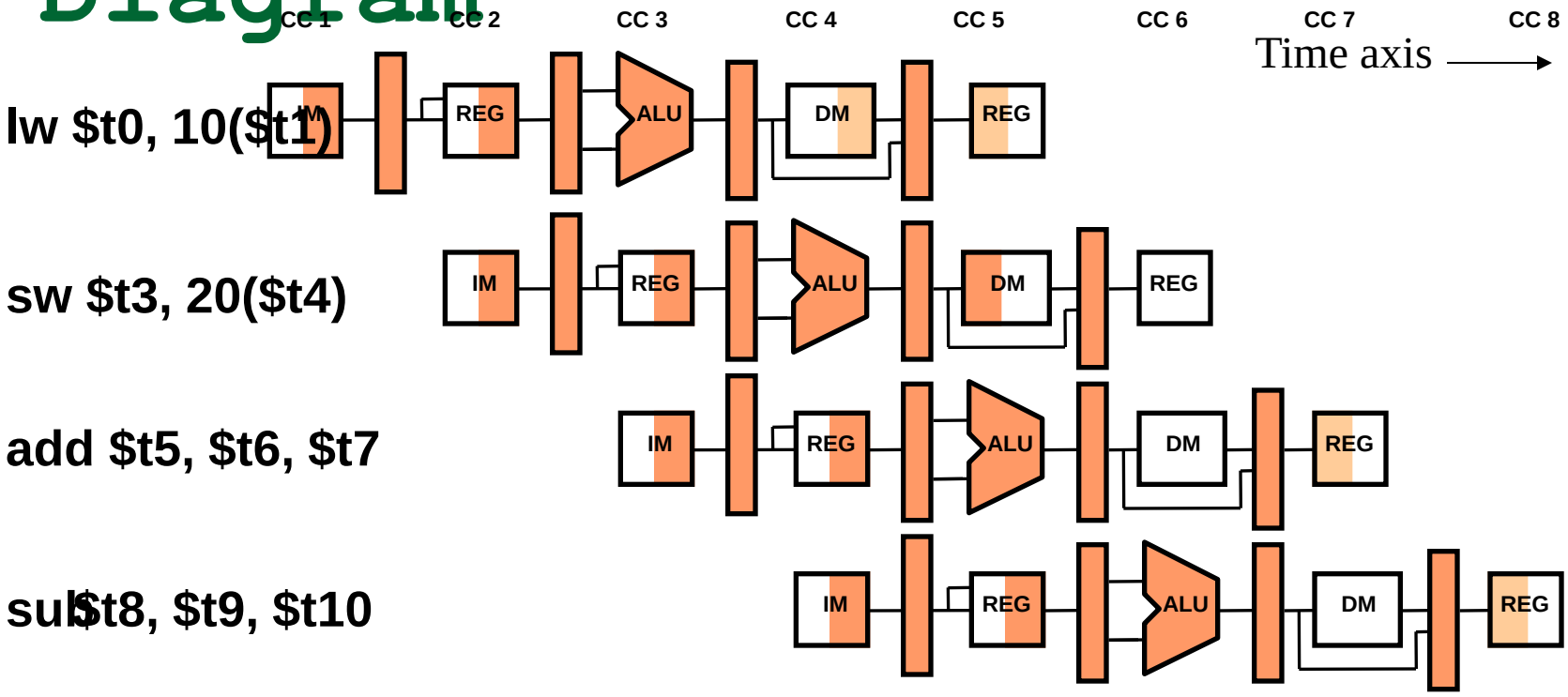
Single-Clock-Cycle Diagram: Clock Cycle 7



Single-Clock-Cycle Diagram: Clock Cycle 8



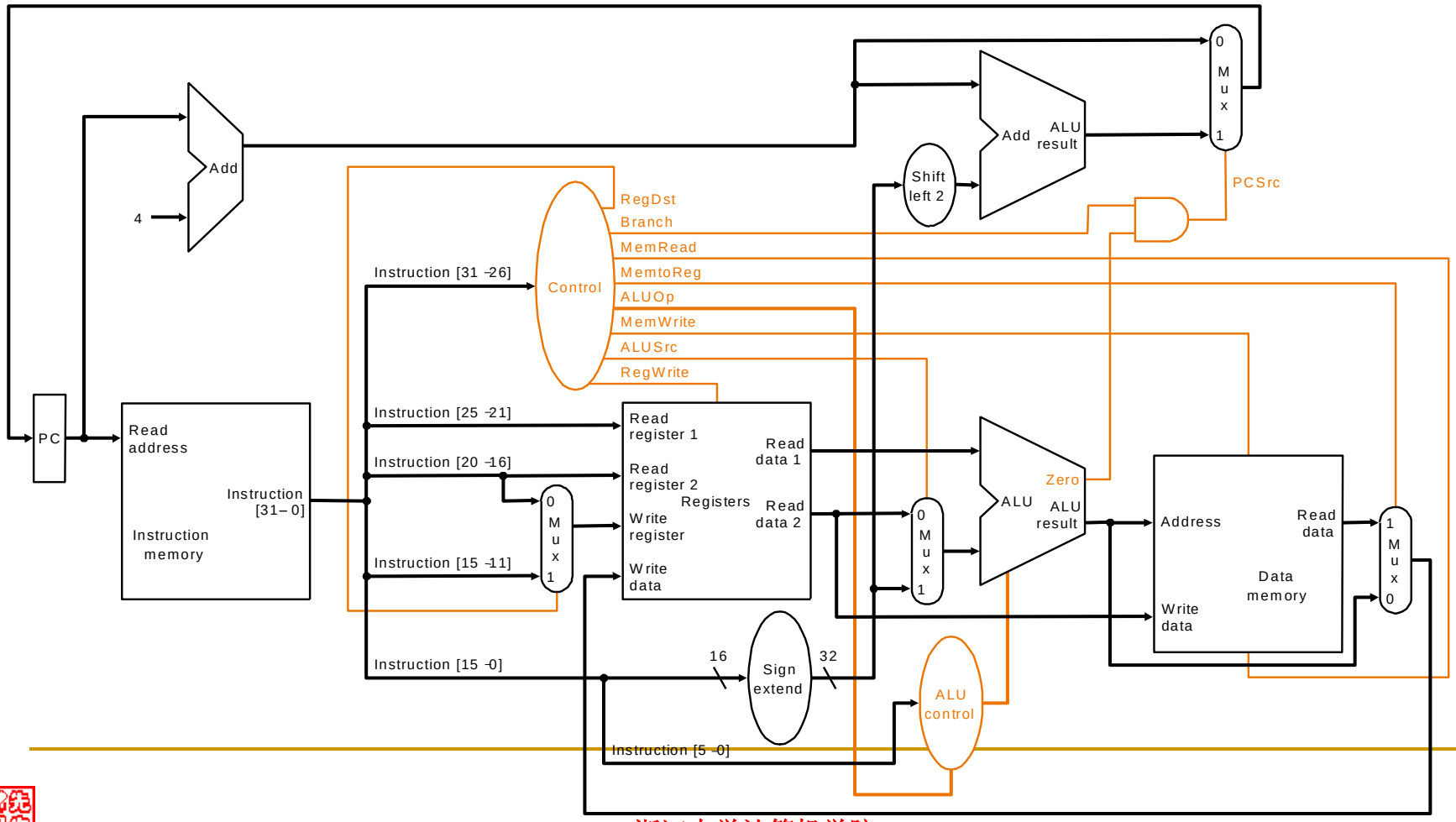
Alternative View – Multiple-Clock-Cycle Diagram



Notes

- One significant difference in the execution of an R-type instruction between multicycle and pipelined implementations:
  - register write-back for the R-type instruction is the 5th (the last write-back) pipeline stage vs. the 4th stage for the multicycle implementation. *Why?*
  - think of *structural hazards* when writing to the register file…
- Worth repeating: the *essential difference* between the pipeline and multicycle implementations is the insertion of pipeline registers to *decouple the 5 stages*
- The CPI of an *ideal pipeline* (no stalls) is 1. *Why?*
- The RaVi Architecture Visualization Project of Dortmund U. has pipeline simulations – see link in our Additional Resources page
- As we develop control for the pipeline keep in mind that the text *does not consider* jump – should not be too hard to implement!

Recall Single-Cycle Control – the Datapath



Recall Single-Cycle – ALU Control

Instruction opcode	AluOp Desired ALU action	Instruction input	Funct Field operation		
LW	00	load word	xxxxxx	add	010
SW	00	store word	xxxxxx	add	010
Branch eq	01	branch eq	xxxxxx	subtract	110
R-type	10	add	100000	add	010
R-type	10	subtract	100010	subtract	110
R-type	10	AND	100100	and	000
R-type					001
R-type					ess 111

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	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

Truth table for ALU control bits

Control Signals

Effect of control bits

Signal Name	Effect when deasserted	Effect when as
RegDst	The register destination number for the	The register
	destination number for the	
	Write register comes from the rt field (bits 20-16)	Write regist
	from the rd field (bits 15-11)	
RegWrite	None	T
	register on the Write register input is written	
	with the value on the Write data input	
AlUSrc	The second ALU operand comes from the	The second ALU ope
	the sign-extended,	
	second register file output (Read data 2)	lower 16
	the instruction	
PCSrc	The PC is replaced by the output of the adder	The PC is repla
	the output of the adder	
	that computes the value of PC + 4	that computes the
	target	
MemRead	None	Data memory contents designat
	addr	
MemWrite	None	Data memory contents designat
	address	
	input	input are replaced by the value of the
MemtoReg	The value fed to the register Write data input	The value fed to th
	register Write data input	

Deter-mining control bits

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

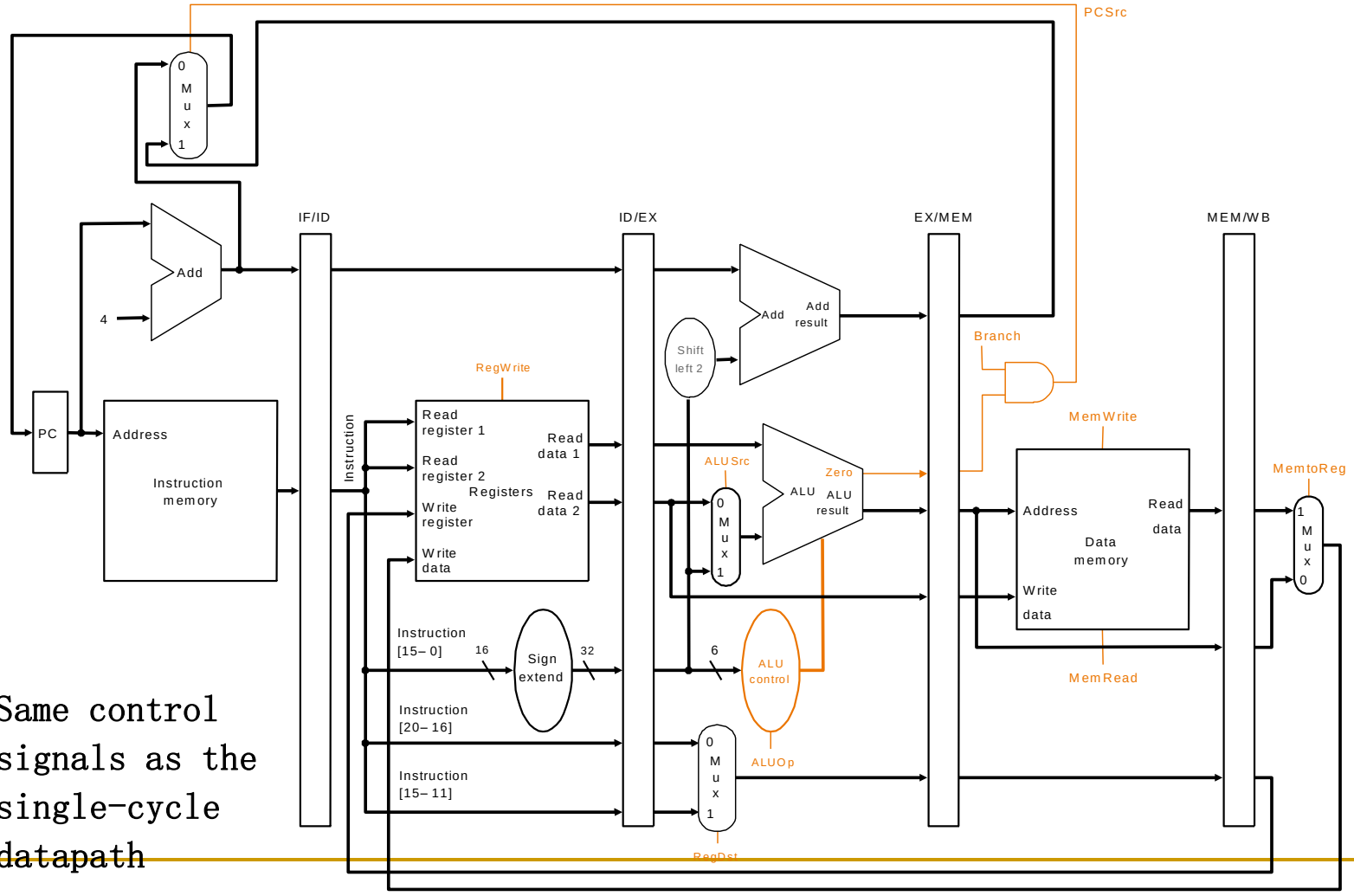
Pipeline Control

- Initial design – *motivated by single-cycle datapath control* – use the *same* control signals
- Observe:
  - No separate write signal for the PC as it is written every cycle
  - No separate write signals for the pipeline registers as they are written every cycle
  - *No separate read signal for instruction memory* as it is read every clock cycle
  - *No separate read signal for register file* as it is read every clock cycle

Will be modified by hazard detection unit!!

- Need to *set control signals during each pipeline stage*
- Since control signals are associated with components active during a single pipeline stage, can *group control lines into five groups according to pipeline stage*

Pipelined Datapath with Control I



Same control signals as the single-cycle datapath

Pipeline Control Signals

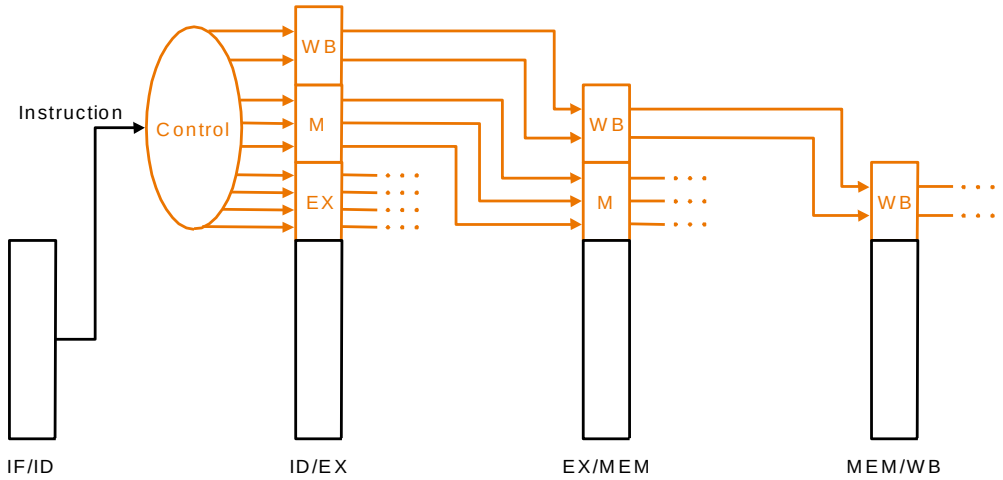
- There are five stages in the pipeline
  - *instruction fetch / PC increment*
  - *instruction decode / register fetch*
  - *execution / address calculation*
  - *memory access*
  - *write back*

Nothing to control as ins
read and PC write are alw

Instruction	Execution/Address Calculation stage control lines				Memory access stage control lines			stage control lines	
	Reg Dst	ALU Op1	ALU Op0	ALU Src	Branch	Mem Read	Mem Write	Reg write	Mem to Reg
R-format	1	1	0	0	0	0	0	1	0
lw	0	0	0	1	0	1	0	1	1
sw	X	0	0	1	0	0	1	0	X
beq	X	0	1	0	1	0	0	0	X

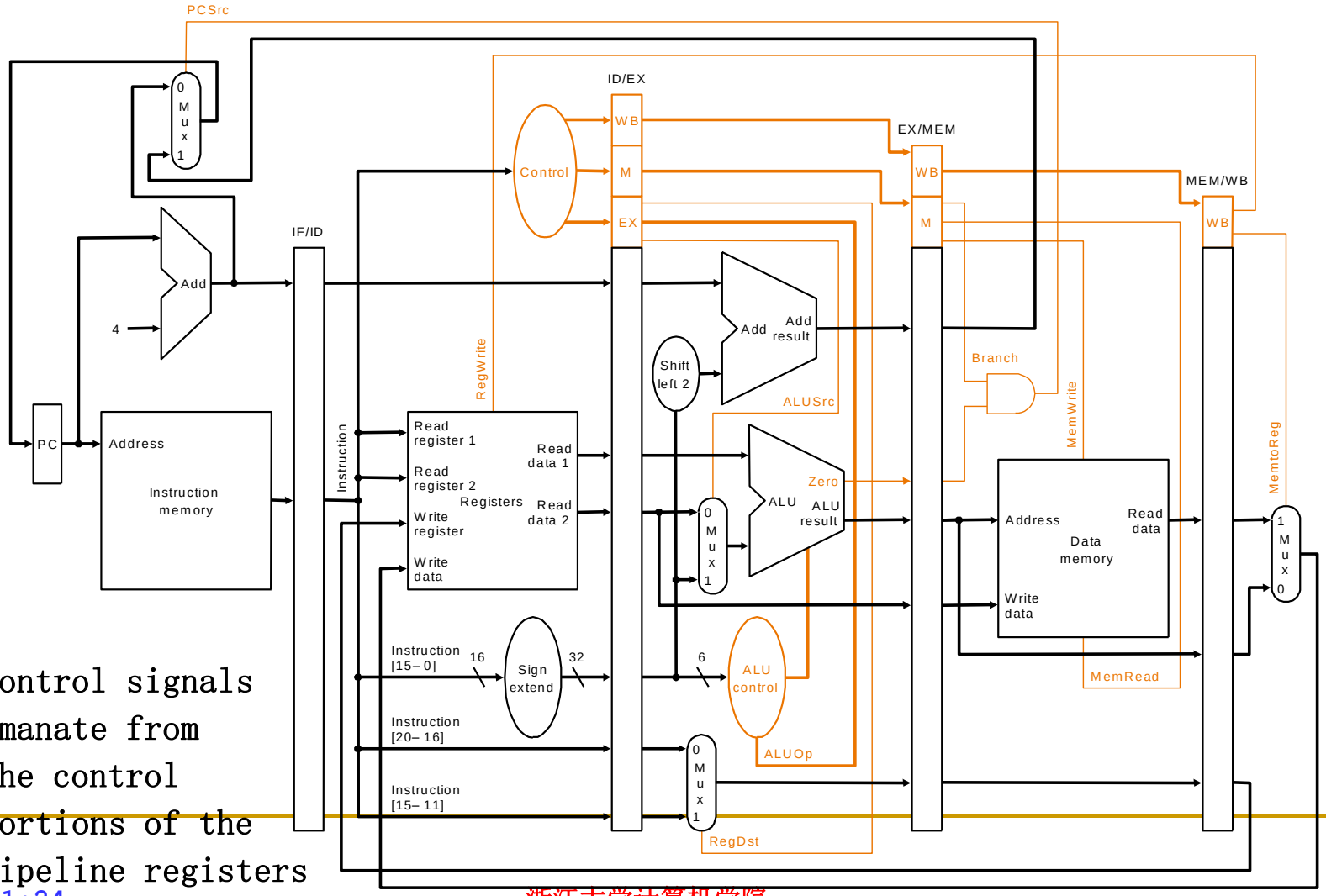
Pipeline Control Implementation

- ***Pass control signals along just like the data*** – extend each pipeline register to hold needed control bits for succeeding stages



- ***Note***: The 6-bit ***funct field*** of the instruction required in the EX stage to generate ALU control can be retrieved as the 6 least significant bits of the immediate field which is sign-extended and passed from the IF/ID register to the ID/EX register

Pipelined Datapath with Control II



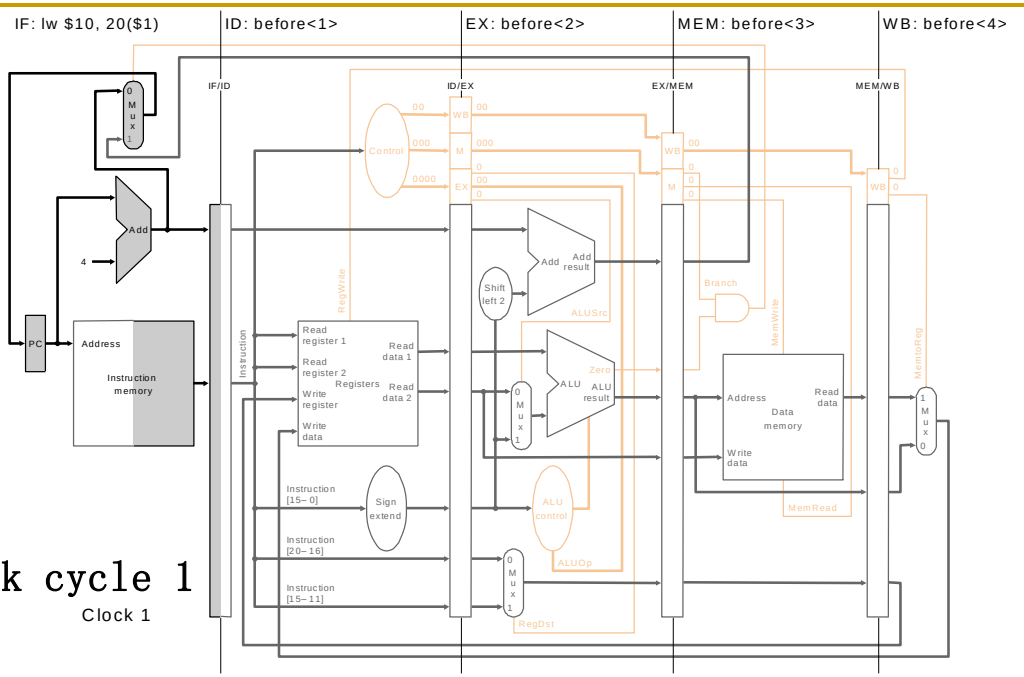
Control signals emanate from the control portions of the pipeline registers

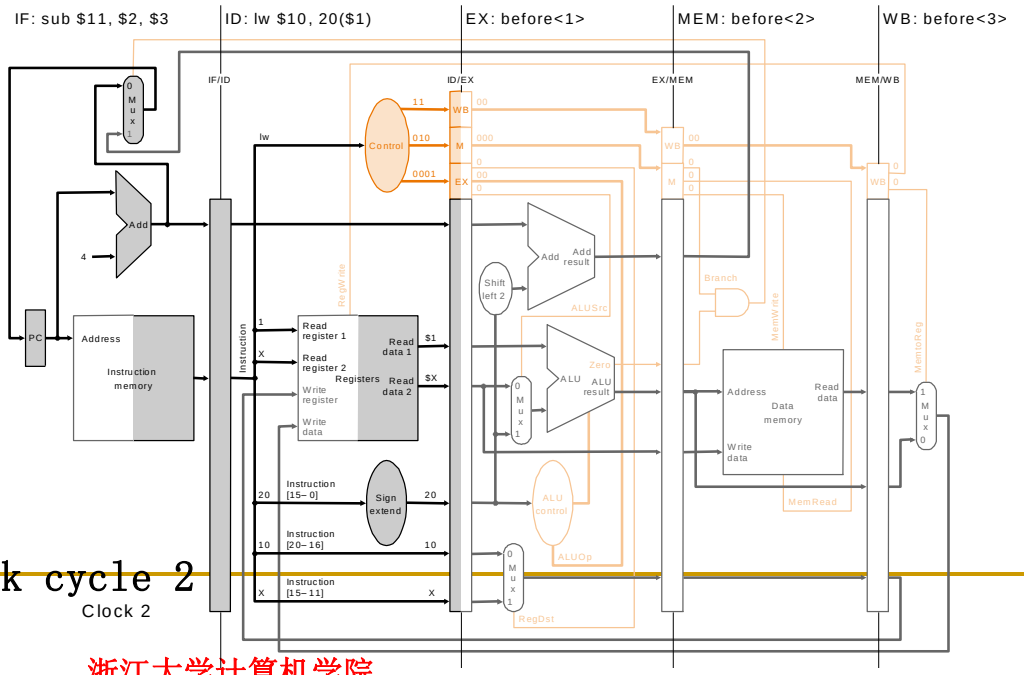
Pipelined Execution and Control

- **Instruction sequence:**

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

Label "before<i>" means i th instruction before **lw**

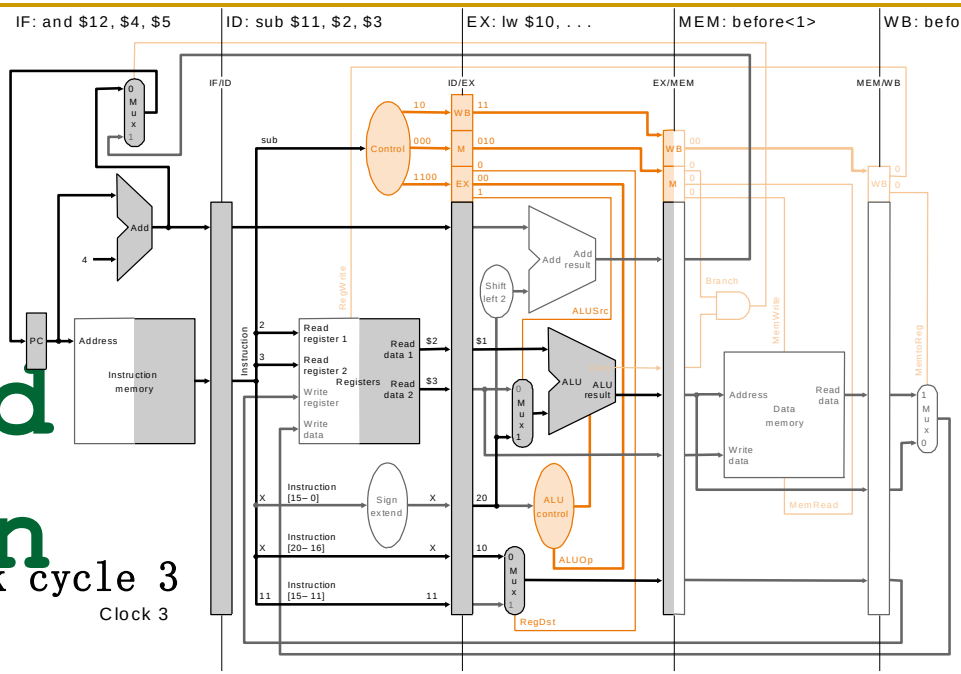


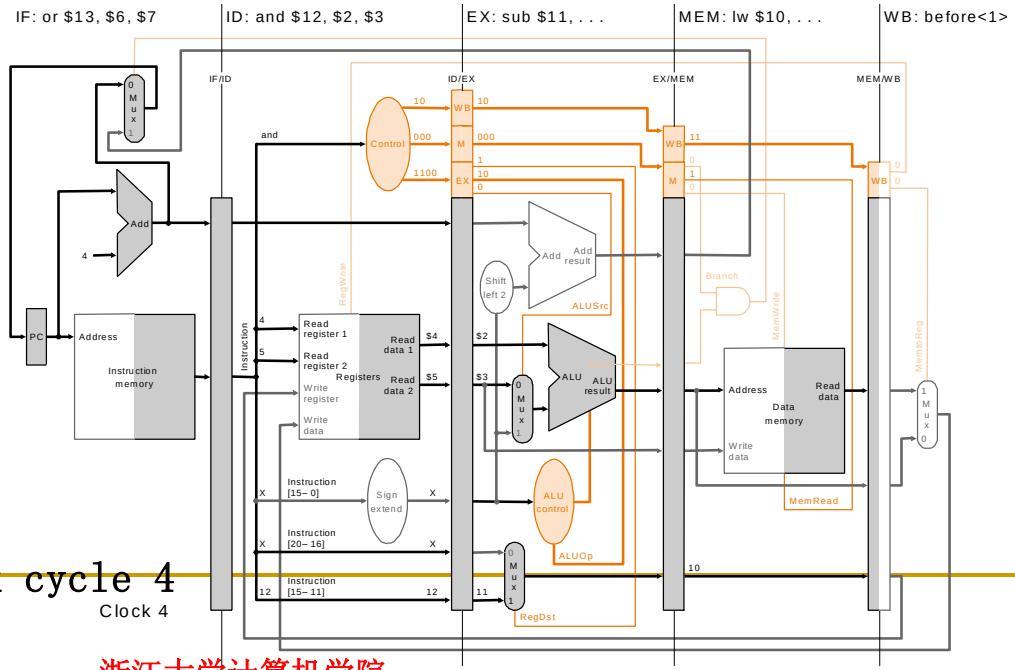


Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```



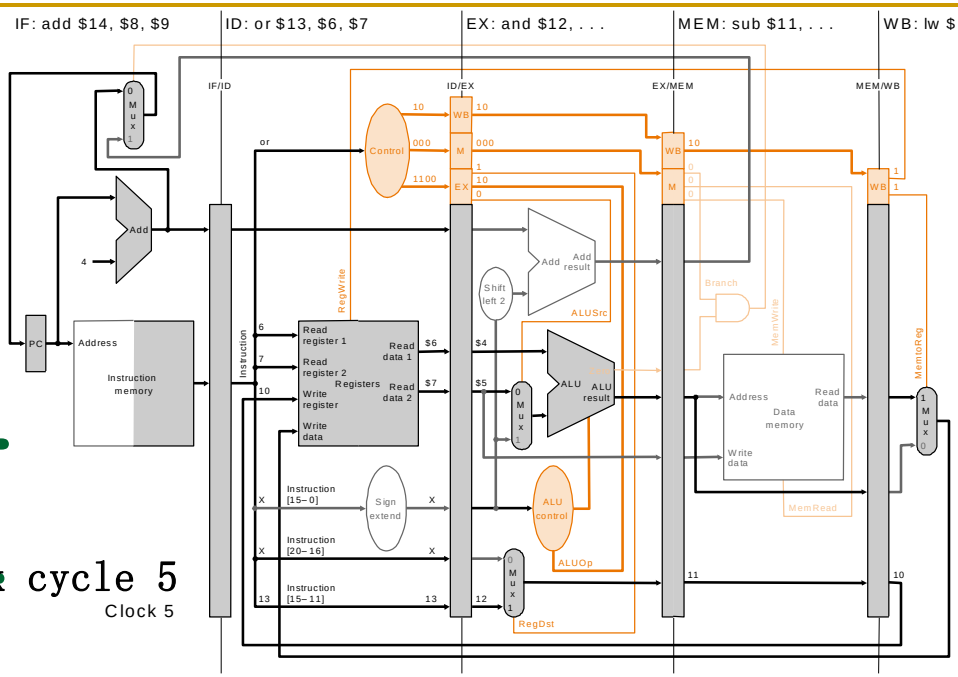


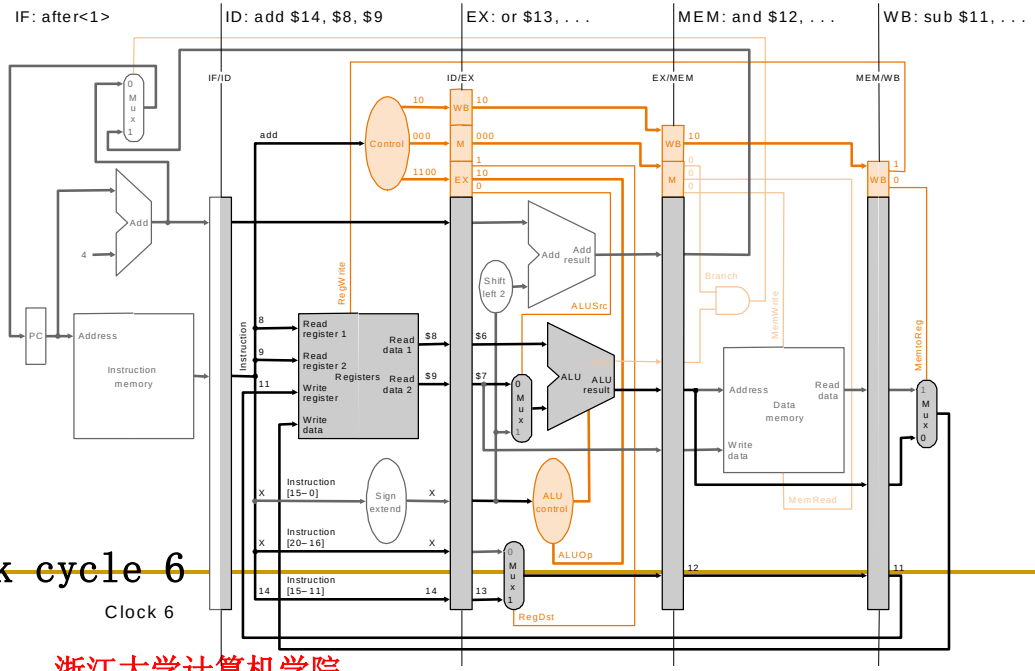
Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

Label “after<i>” means i th instruction after **add**

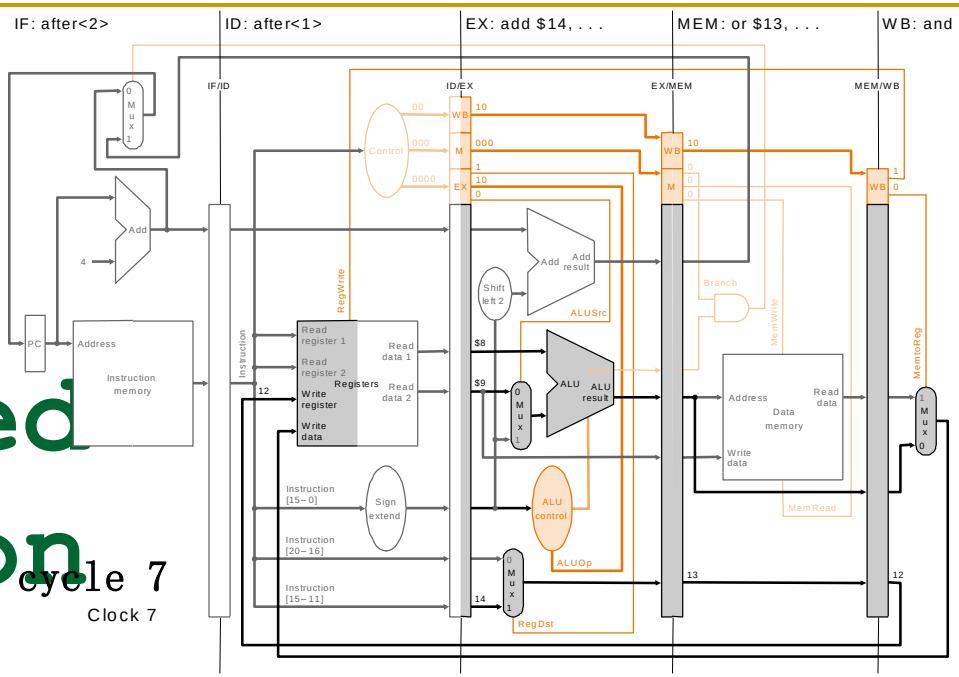


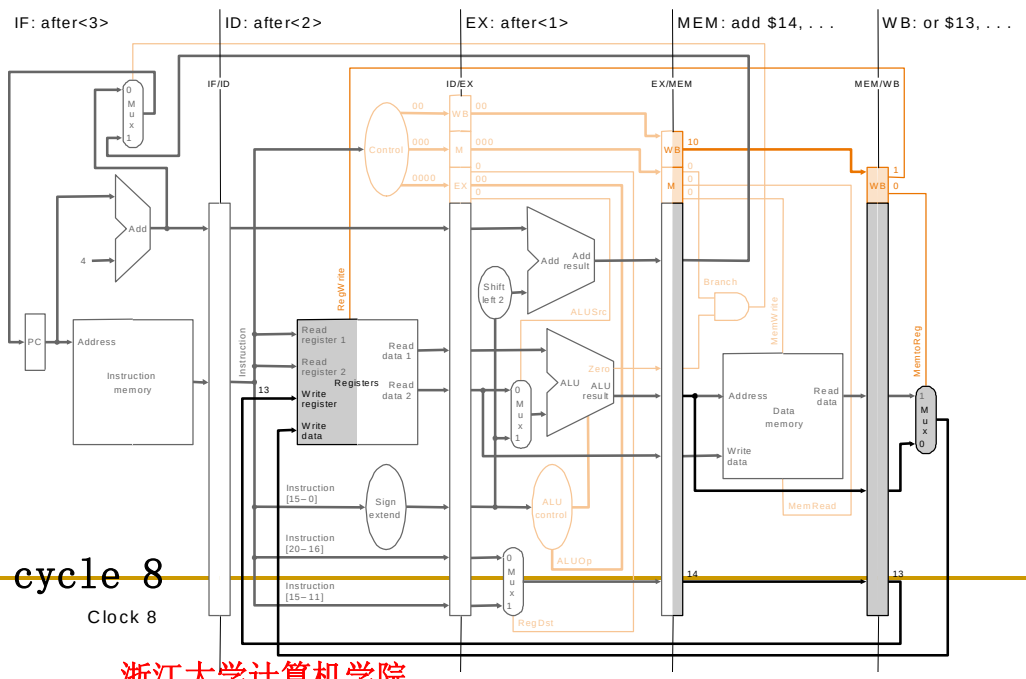


Pipelined Execution and Control

- Instruction sequence:

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```

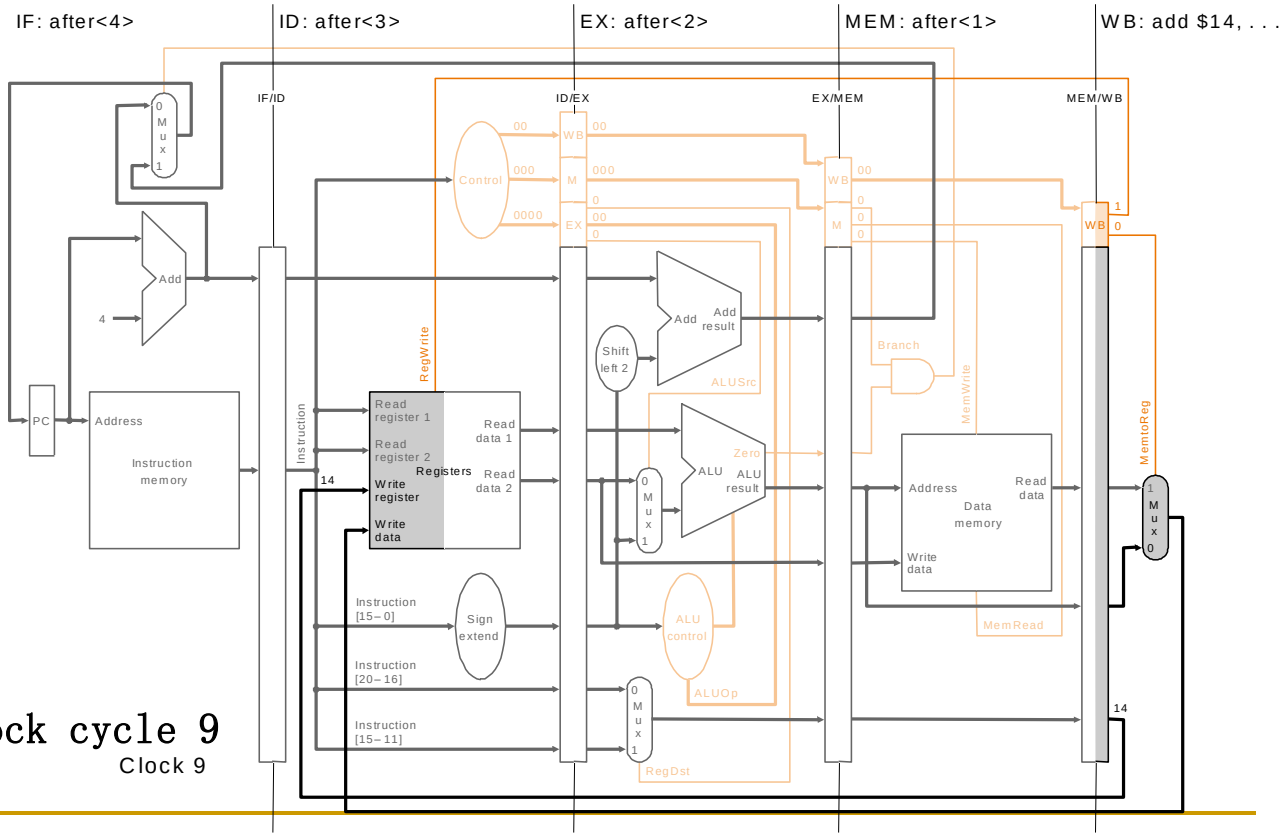




Pipelined Execution and Control

- **Instruction sequence:**

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $7
or  $13, $6, $7
add $14, $8, $9
```



Clock cycle 9

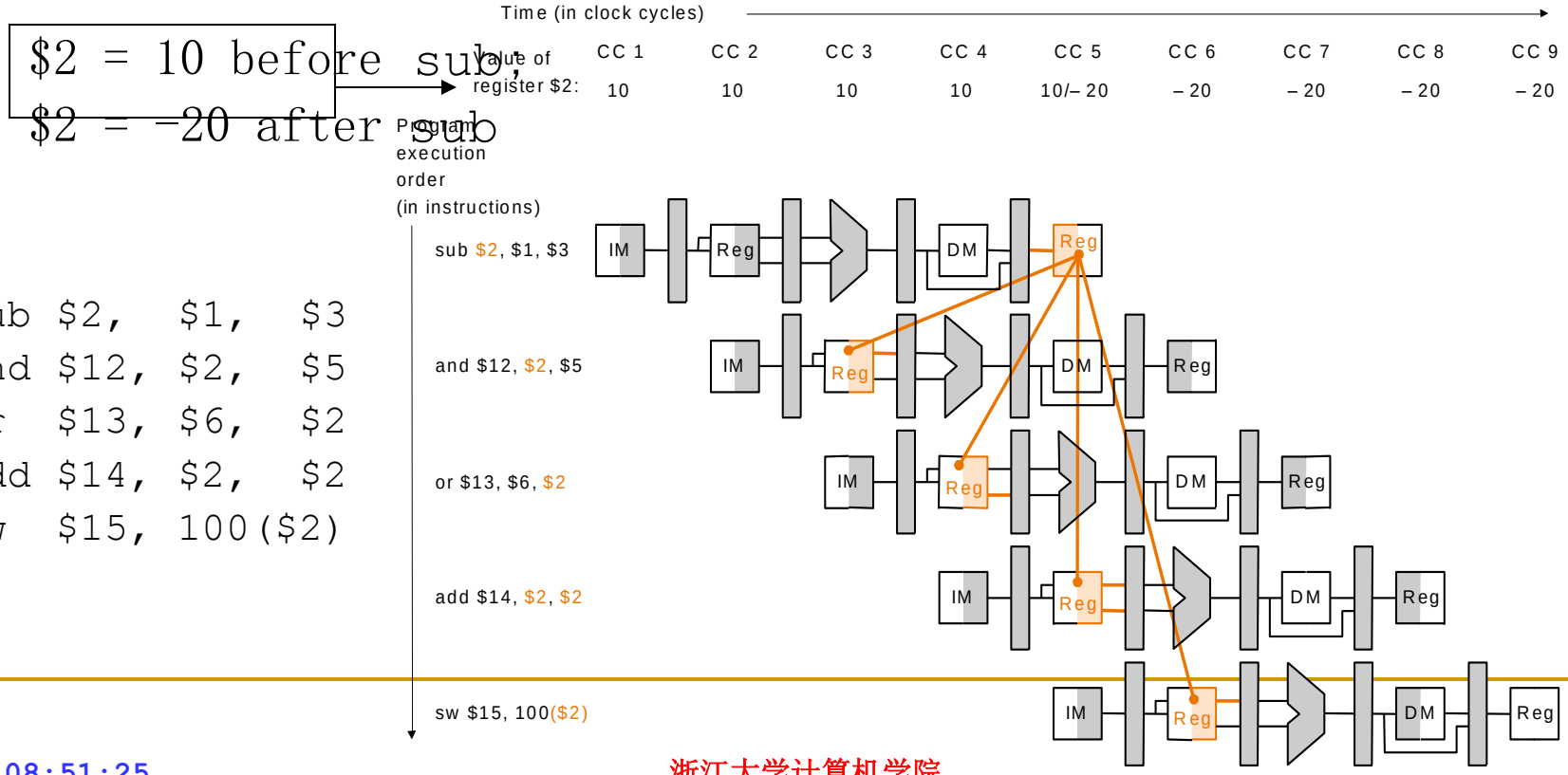
Clock 9

Revisiting Hazards

- So far our datapath and control have ignored hazards
- We shall revisit *data hazards* and *control hazards* and enhance our datapath and control to handle them in *hardware*…

Data Hazards and Forwarding

- **Problem with starting an instruction before previous are finished:**
  - **data dependencies that go backward in time – called *data hazards***




```
sub $2,   $1,   $3
and $12,  $2,   $5
or  $13,  $6,   $2
add $14,  $2,   $2
sw  $15,  100($2)
```

Software Solution

- **Have compiler guarantee *never* any data hazards!**
  - **by *rearranging instructions to insert independent instructions between instructions* that would otherwise have a data hazard between them,**
  - **or, if such rearrangement is not possible, *insert* nops**

```
sub   $2,  $1, $3          sub   $2,  $1, $3
lw    $10, 40($3)          nop
slt   $5, $6, $7           nop
and   $12, $2, $5    or    and   $12, $2, $5
or    $13, $6, $2          or    $13, $6, $2
add   $14, $2, $2          add   $14, $2, $2
sw    $15, 100($2)         sw    $15, 100($2)
```

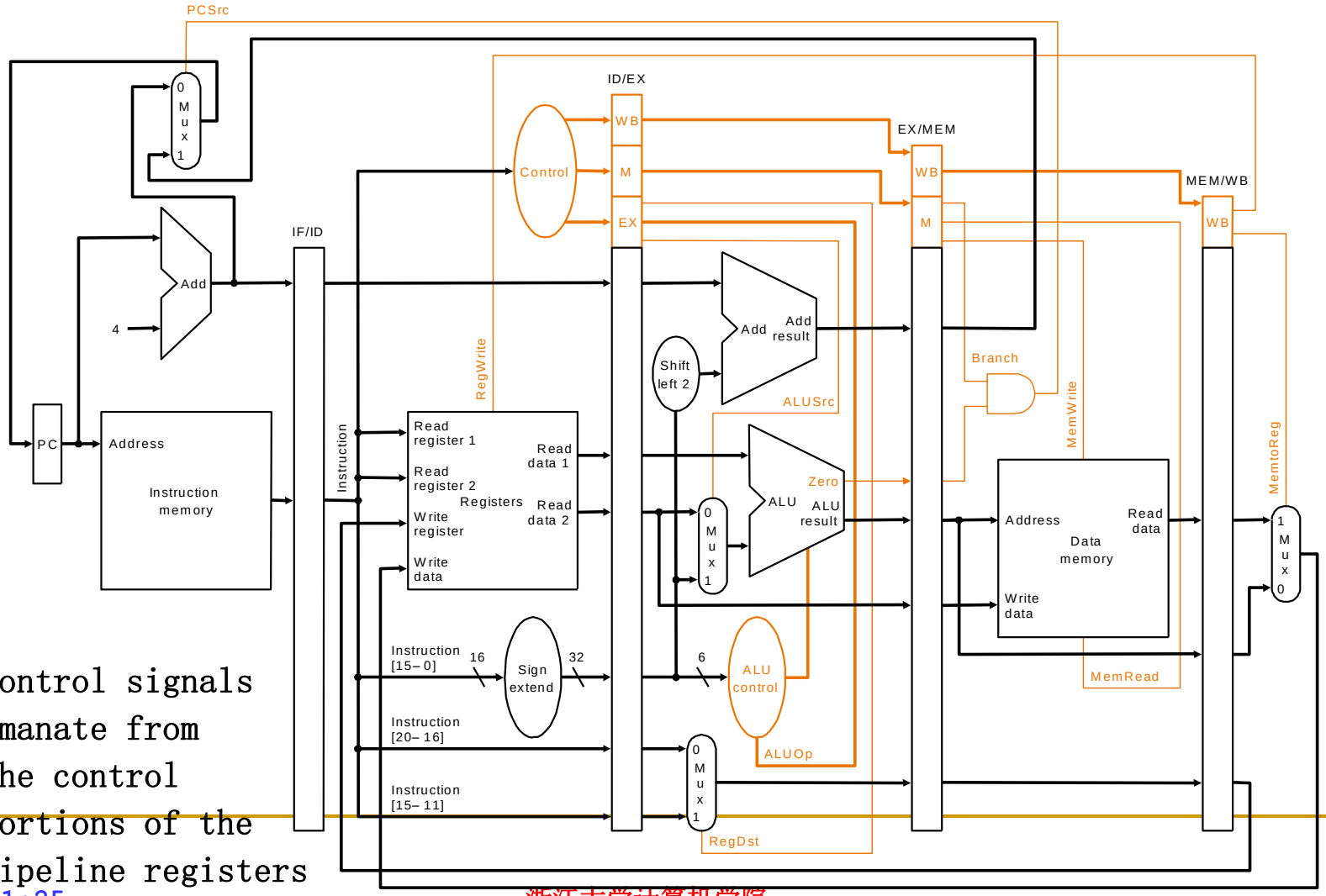
- **Such compiler solutions may not always be possible, and nops slow the machine down**

MIPS: nop = "no operation" = 00…0 (32b

Hardware Solution: Forwarding

- Idea: *use intermediate data*, do not wait for result to be finally written to the destination register. Two steps:
  1. *Detect* data hazard
  2. *Forward* intermediate data to resolve hazard

Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

Hazard Detection

- **Hazard conditions:**

1a. EX/MEM.RegisterRd = ID/EX.RegisterRs

1b. EX/MEM.RegisterRd = ID/EX.RegisterRt

2a. MEM/WB.RegisterRd = ID/EX.RegisterRs

2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

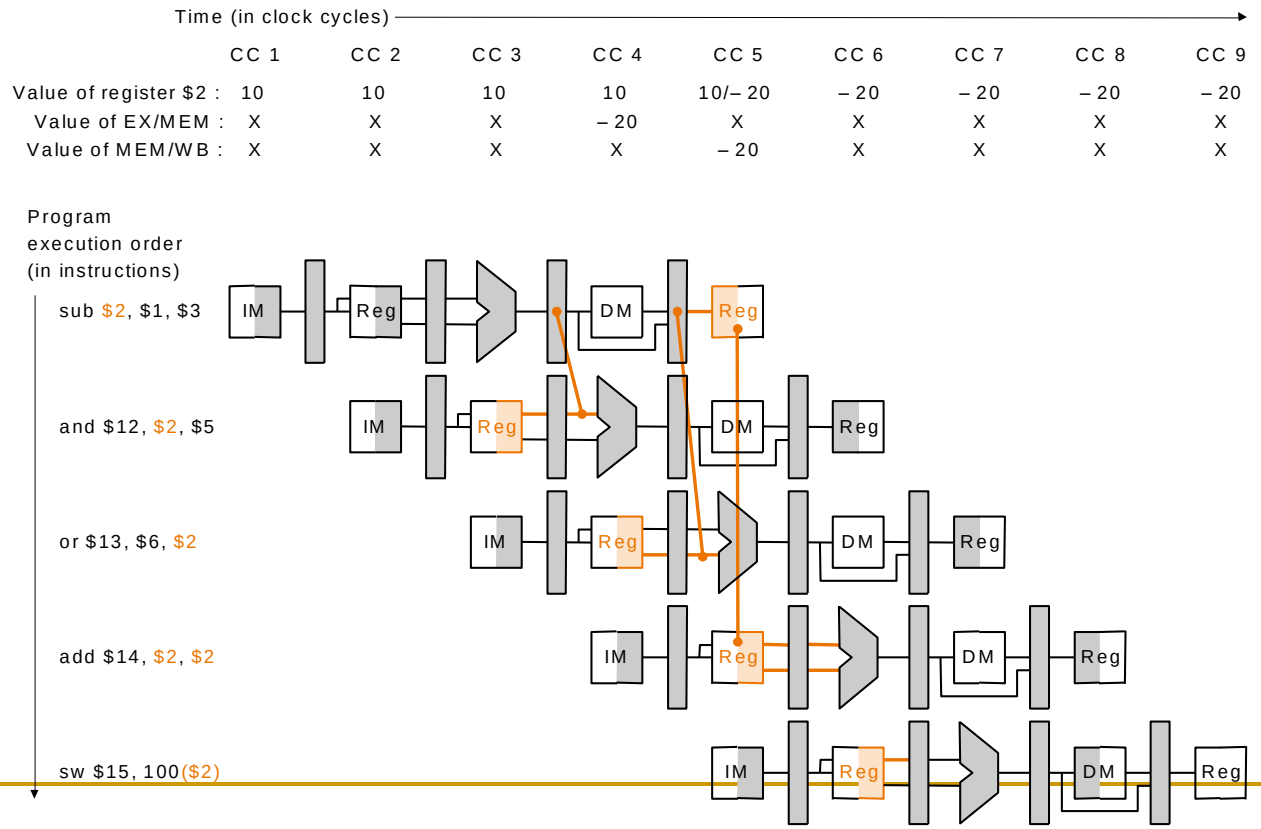
  - Eg., in the earlier example, first hazard between `sub $2, $1, $3` and `and $12, $2, $5` is detected when the `and` is in EX stage and the `sub` is in MEM stage because
    - EX/MEM.RegisterRd = ID/EX.RegisterRs = $2 (1a)

- **Whether to forward also depends on:**
  - *if the later instruction is going to write a register* – if *not*, no need to forward, even if there is register number match as in conditions above
  - *if the destination register of the later instruction is $0* – in which case there is no need to forward value ($0 is always 0 and never overwritten)

Data Forwarding

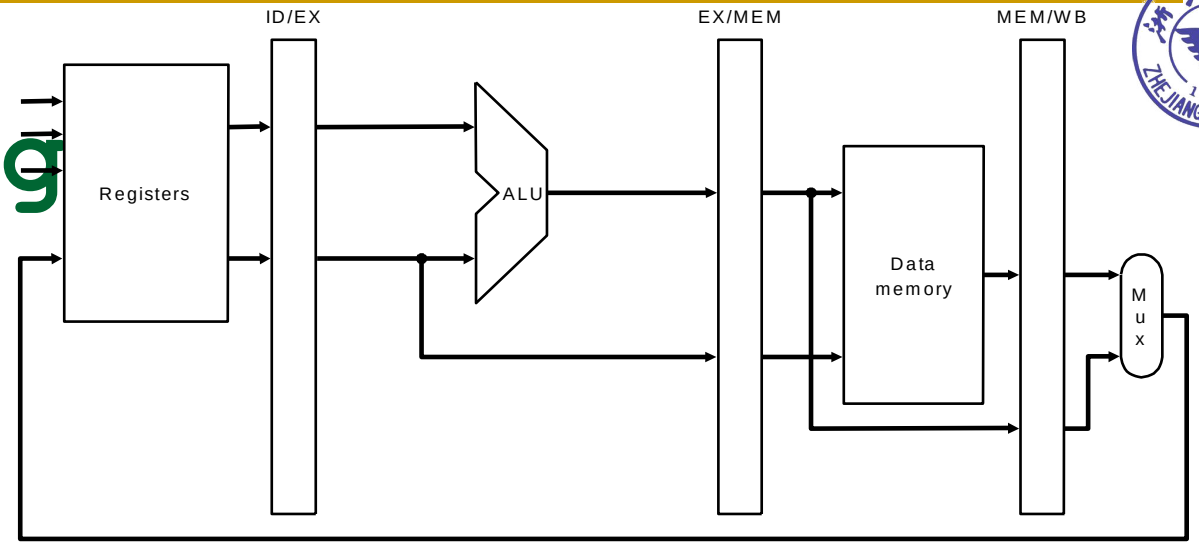
- **Plan:**
  - *allow inputs to the ALU not just from ID/EX, but also later pipeline registers, and*
  - *use multiplexors and control signals to choose appropriate inputs to ALU*

```
sub $2,   $1,   $3
and $12,  $2,   $5
or  $13,  $6,   $2
add $14,  $2,   $2
sw  $15,  100($2)
```



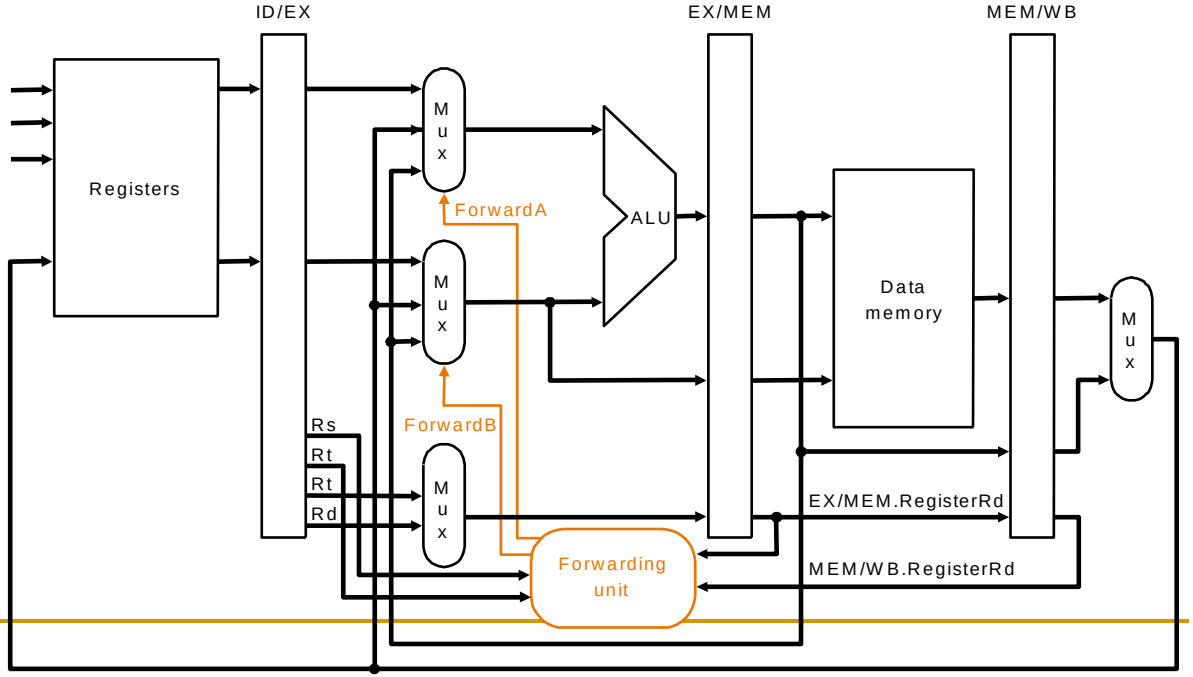
Dependencies between pipelines move forward in time

Forwarding Hardware



a. No forwarding

Datapath before adding forwarding hardware

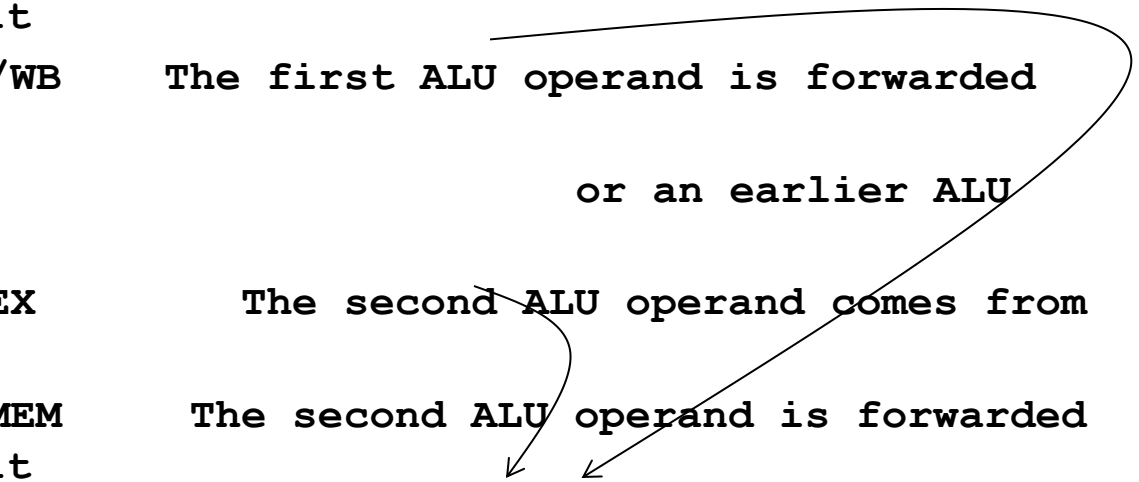


b. With forwarding

Datapath after adding forwarding hardware

Forwarding Hardware: Multiplexor Control

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from prior ALU result
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result
ForwardB = 00	ID/EX	The second ALU operand comes from the register file
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from prior ALU result
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result



Data Hazard: Detection and Forwarding

- Forwarding unit determines multiplexor control according to the following rules:

1. EX hazard

```
     if (          EX/MEM.RegWrite
// if there is a write…
           and ( EX/MEM.RegisterRd ≠ 0 )
// to a non-$0 register…
           and ( EX/MEM.RegisterRd = ID/EX.RegisterRs ) )
// which matches, then…     ForwardA = 10

     if (          EX/MEM.RegWrite
// if there is a write…
           and ( EX/MEM.RegisterRd ≠ 0 )
// to a non-$0 register…
           and ( EX/MEM.RegisterRd = ID/EX.RegisterRt ) )
// which matches, then… ForwardB = 10
```

Data Hazard: Detection and Forwarding

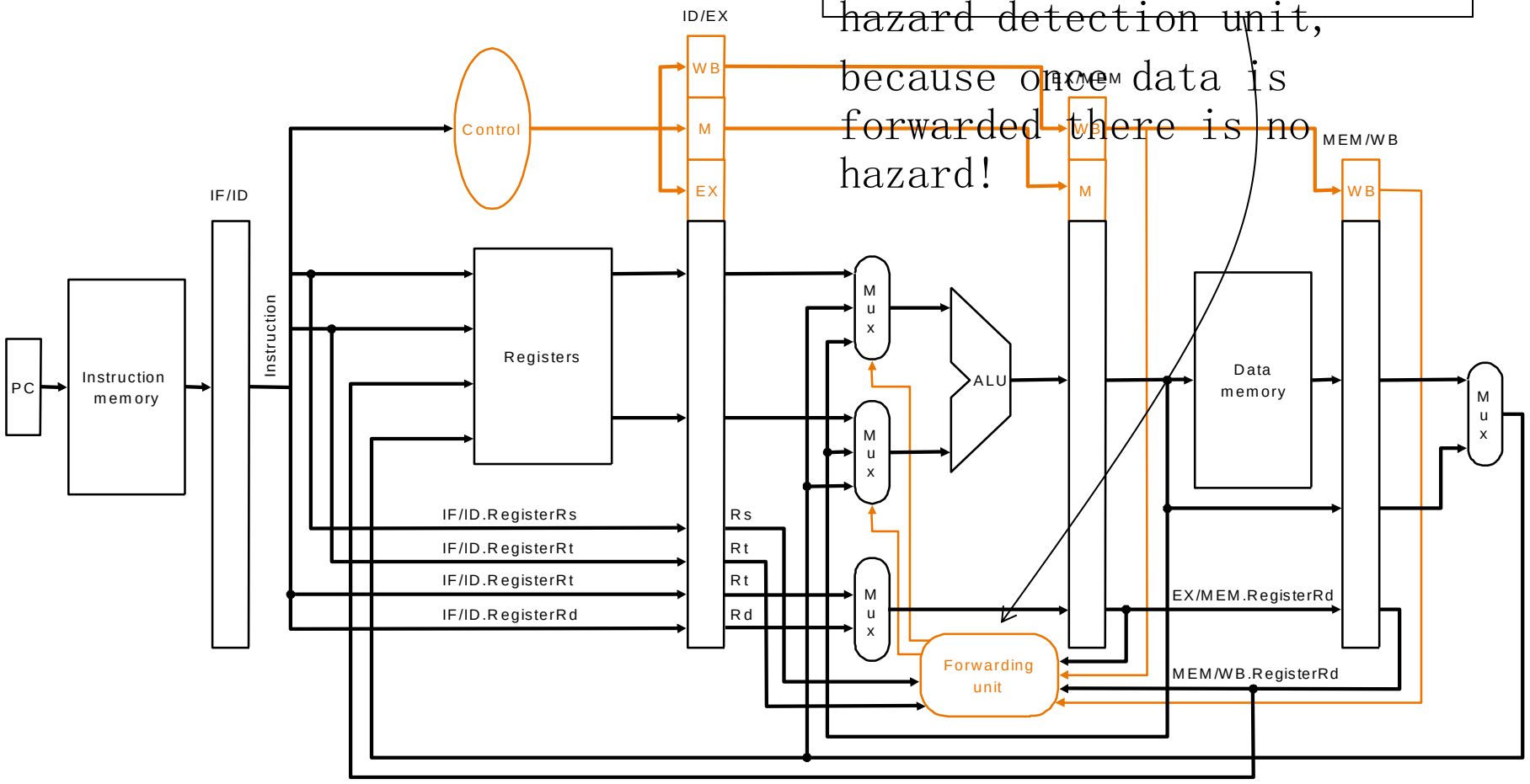
2. MEM hazard

```
      if (          MEM/WB.RegWrite
// if there is a write…
          and ( MEM/WB.RegisterRd ≠ 0 )
// to a non-$0 register…
          and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRs )      // and
not already a register match

// with earlier pipeline register…
          and ( MEM/WB.RegisterRd = ID/EX.RegisterRs ) )  // but
match with later pipeline

register, then…
      ForwardA = 01


      if (          MEM/WB.RegWrite
// if there is a write…
          and ( MEM/WB.RegisterRd ≠ 0 )
// to a non-$0 register…
          and ( EX/MEM.RegisterRd ≠ ID/EX.RegisterRt )      // and
not already a register match
```

This check is necessary, e.g., for sequences such as add $1, $1, $2; (array summing); where each earlier pipeline (EX/MEM) register has

Forwarding Hardware with Control

Called forwarding unit, not hazard detection unit, because once data is forwarded there is no hazard!



Datapath with forwarding hardware and control wires - certain details, e.g., branching hardware, are omitted to simplify the drawing

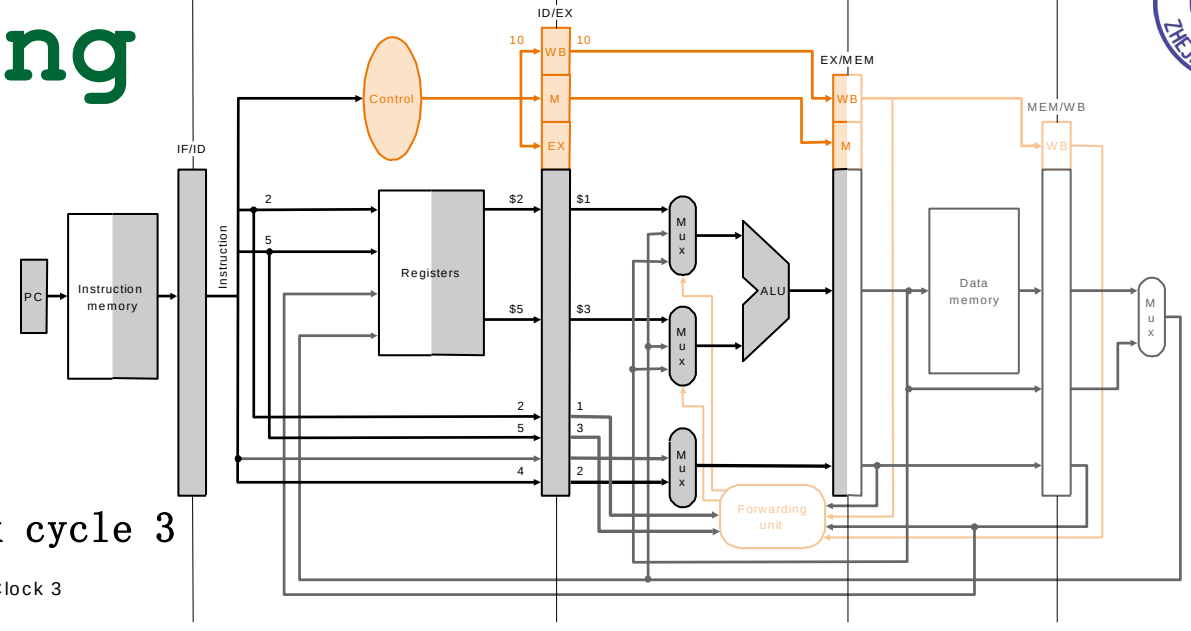
Note: so far we have only handled forwarding to R-type instructions…!

Forwarding

- **Execution example:**

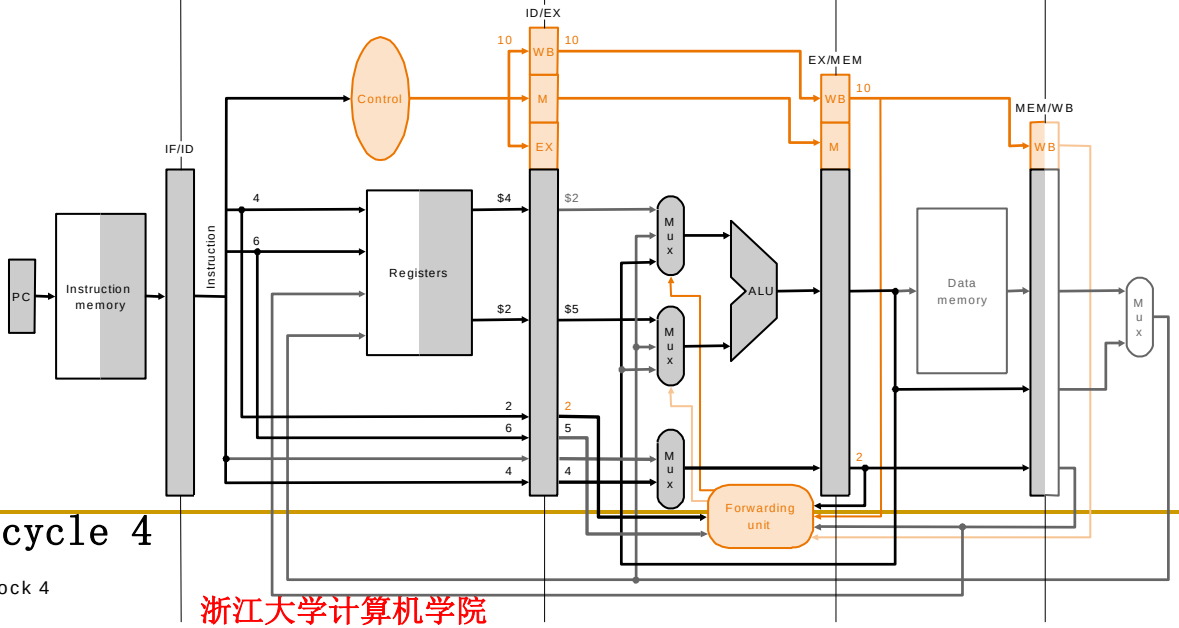
```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

Clock cycle 3



Clock 3

Clock cycle 4



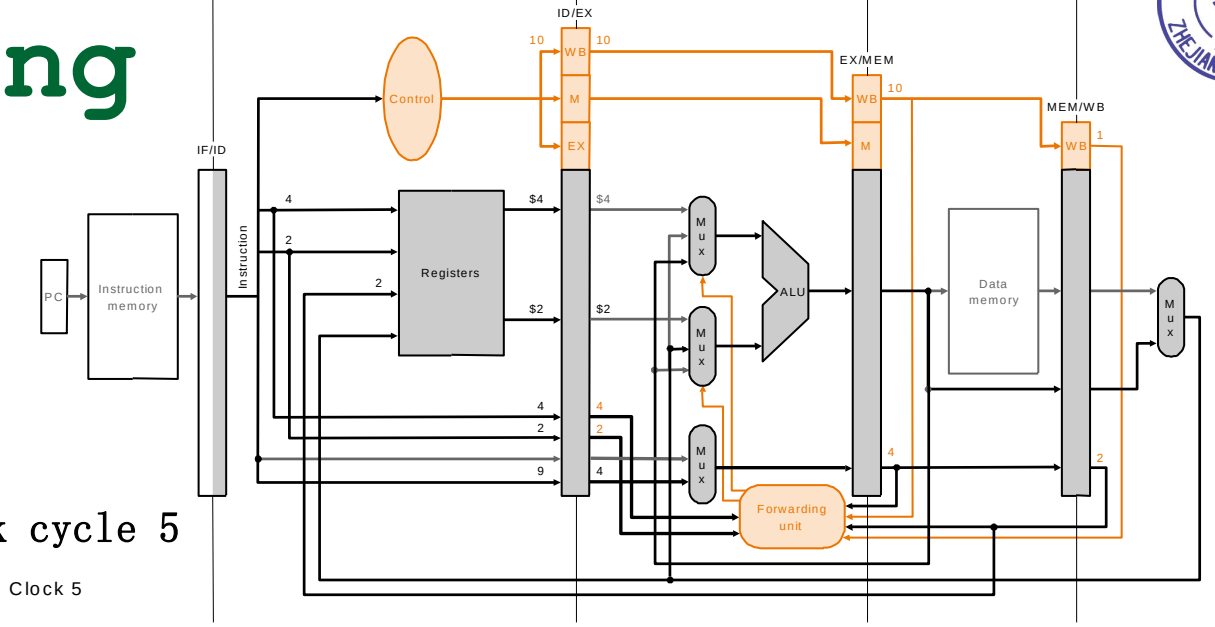
Clock 4

Forwarding

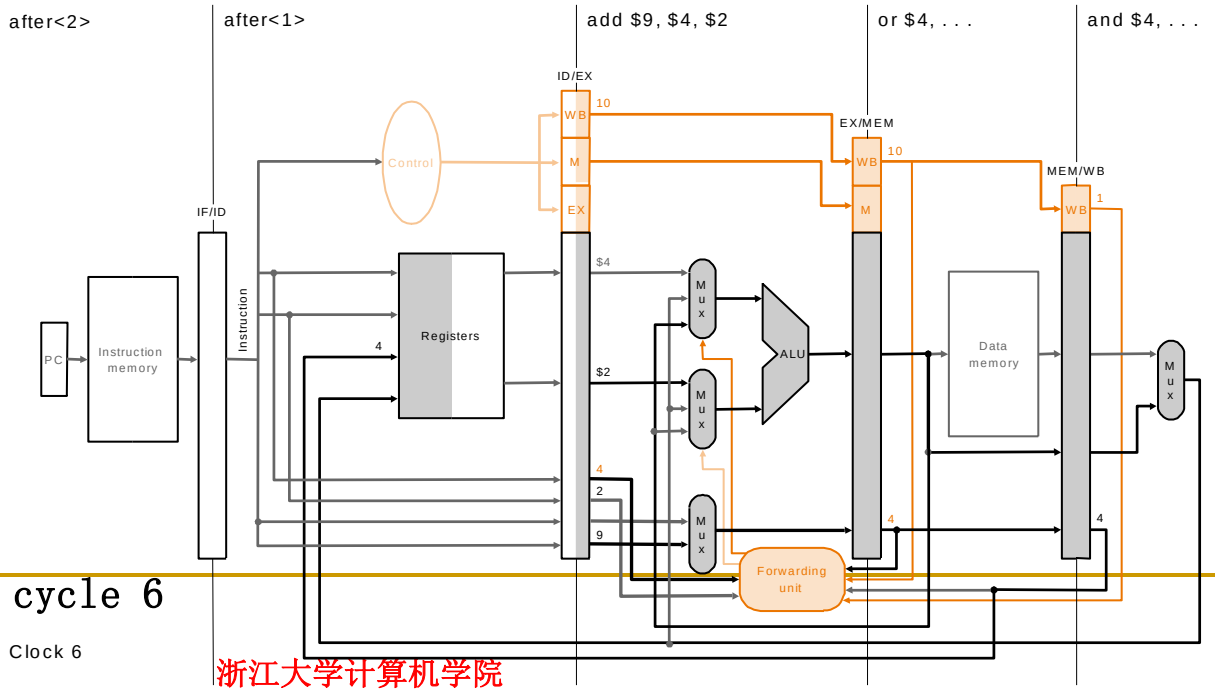
- **Execution example (cont.):**

```
sub $2, $1, $3
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

Clock cycle 5



Clock cycle 6

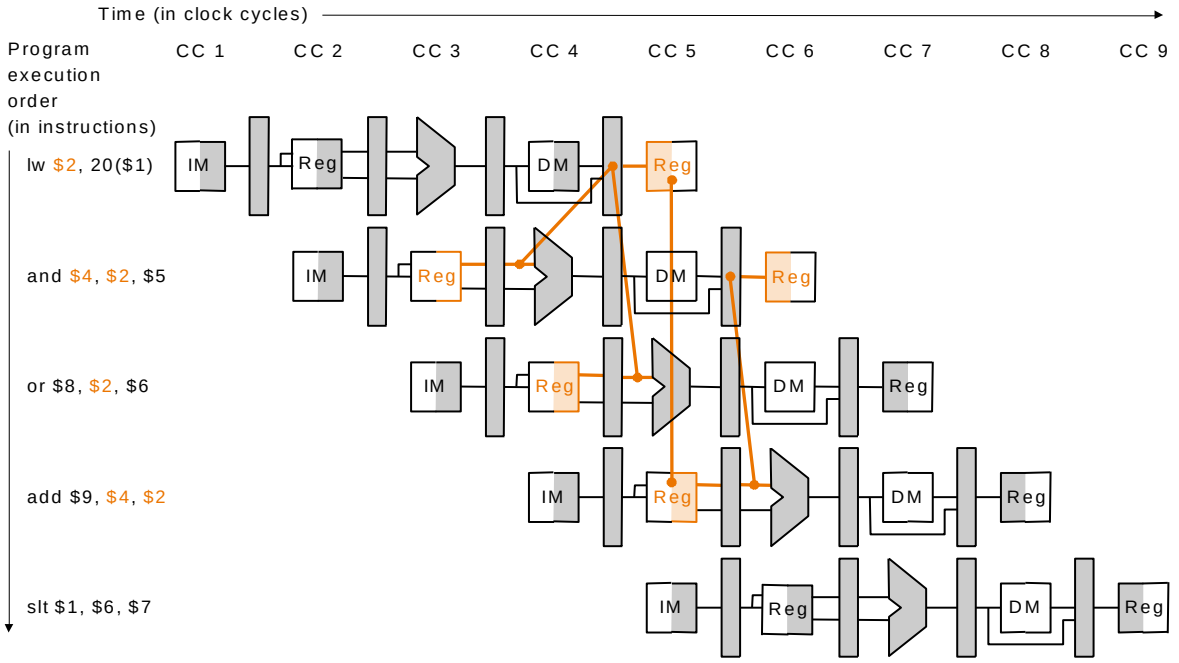


Data Hazards and Stalls

- **Load word can still cause a hazard:**
  - an instruction tries to read a register following a load instruction that writes to the same register

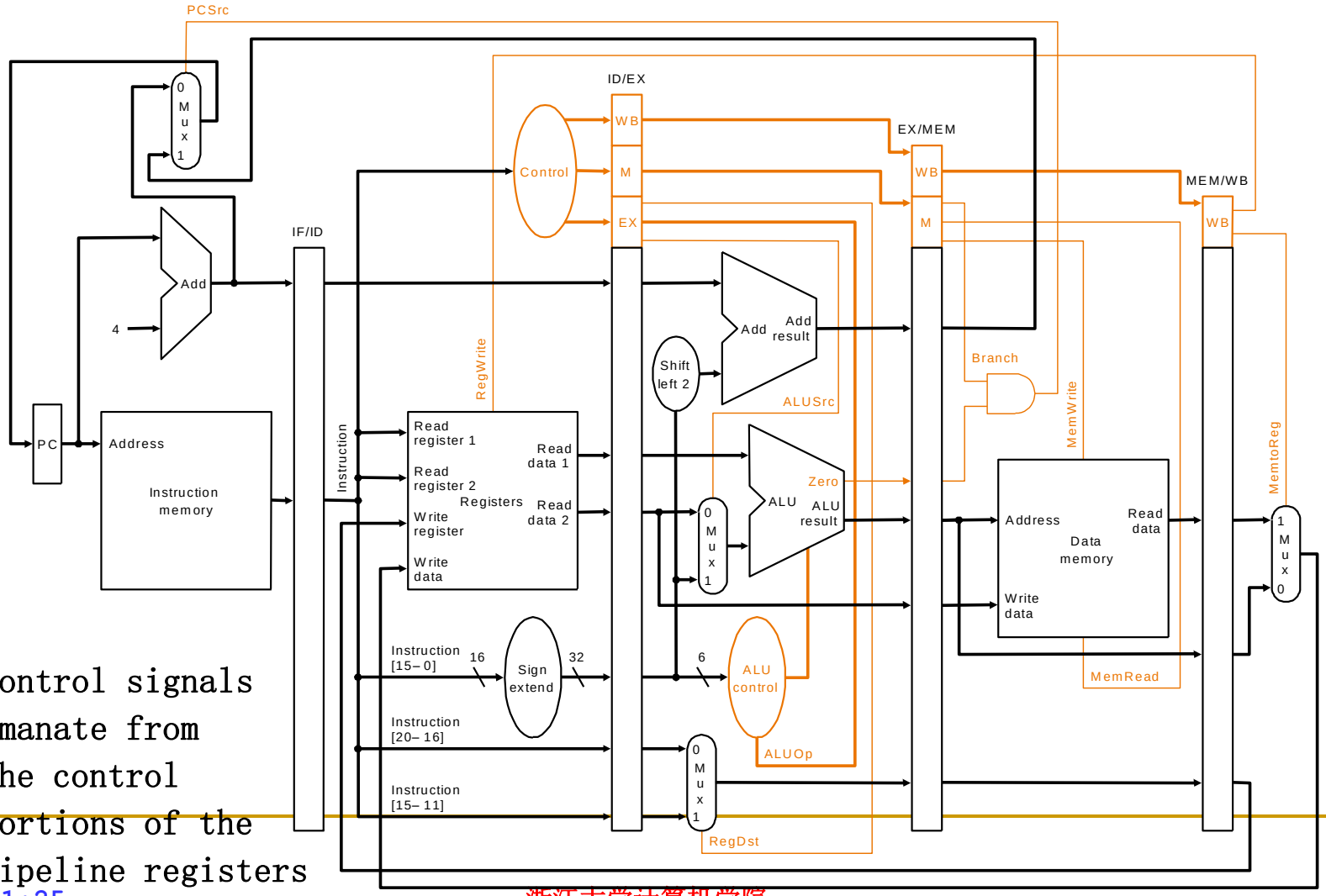
```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
Slt $1, $6, $7
```

As even a pipeline dependency goes backward in time forwarding will not solve the hazard



  - therefore, we need a *hazard detection unit* to *stall* the pipeline after the load instruction

Pipelined Datapath with Control II (as before)



Control signals emanate from the control portions of the pipeline registers

Hazard Detection Logic to Stall

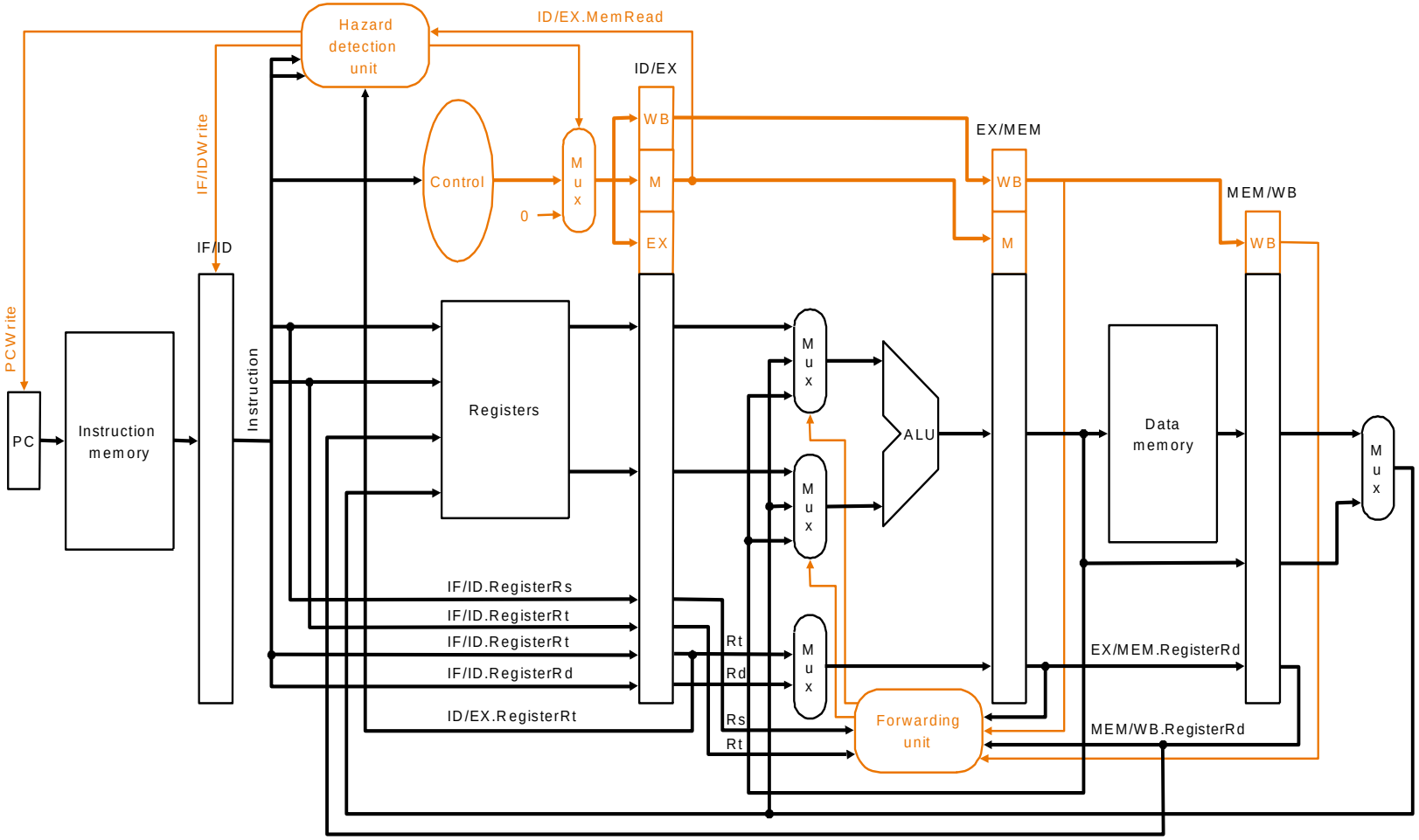
- Hazard detection unit implements the following check if to stall

```
if ( ID/EX.MemRead                                    // if the
   instruction in the EX stage is a load…
     and ( ( ID/EX.RegisterRt = IF/ID.RegisterRs )          // and
   the destination register
        or   ( ID/EX.RegisterRt = IF/ID.RegisterRt ) ) )   //
   matches either source register
                                                            //
   of the instruction in the ID stage, then…
stall the pipeline
```

Mechanics of Stalling

- If the check to stall verifies, then the *pipeline needs to stall only 1 clock cycle* after the load as after that the forwarding unit can resolve the dependency
- What the hardware does to stall the pipeline 1 cycle:
  - *does not let the IF/ID register change (disable write!)* – this will cause the instruction in the ID stage to repeat, i.e., *stall*
  - therefore, the instruction, just behind, in the IF stage must be stalled as well – so hardware *does not let the PC change (disable write!)* – this will cause the instruction in the IF stage to repeat, i.e., *stall*
  - *changes all the EX, MEM and WB control fields in the ID/EX pipeline register to 0*, so effectively the instruction just behind the load becomes a nop – a *bubble* is said to have been inserted

Hazard Detection Unit

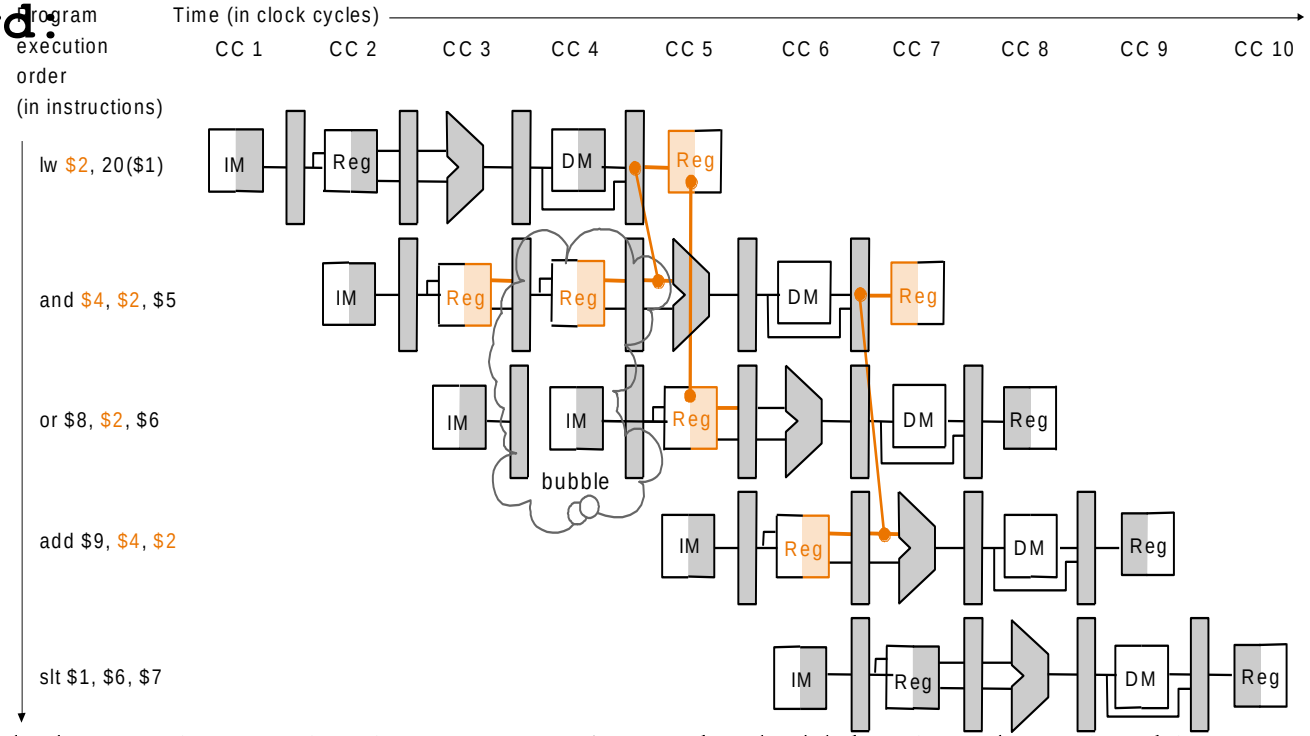


Datapath with forwarding hardware, the hazard detection unit and controls wires - certain details, e.g., branching hardware are omitted to simplify the drawing

Stalling Resolves a Hazard

- **Same instruction sequence as before for which forwarding by itself could not resolve the hazard.**

```
lw   $2, 20($1)
and  $4, $2, $5
or   $8, $2, $6
add  $9, $4, $2
Slt  $1, $6, $7
```

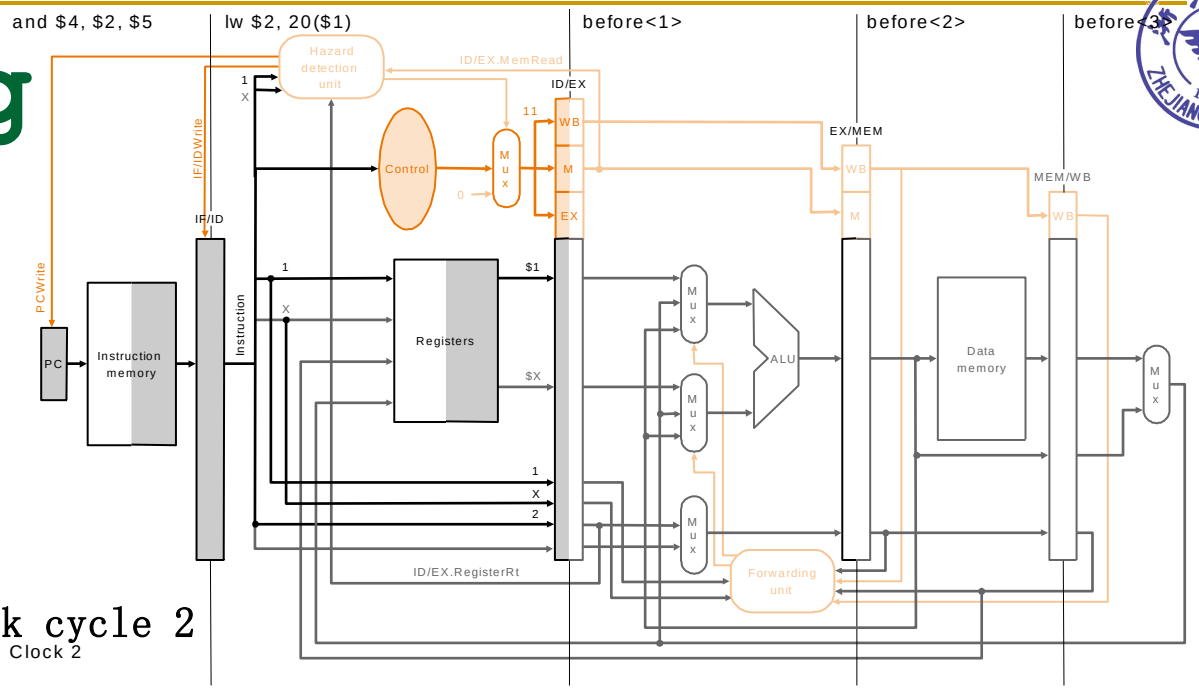


Hazard detection unit inserts a 1-cycle bubble in the pipeline, after which all pipeline register dependencies go forward so then the forwarding unit can handle them and there are no more hazards

Stalling

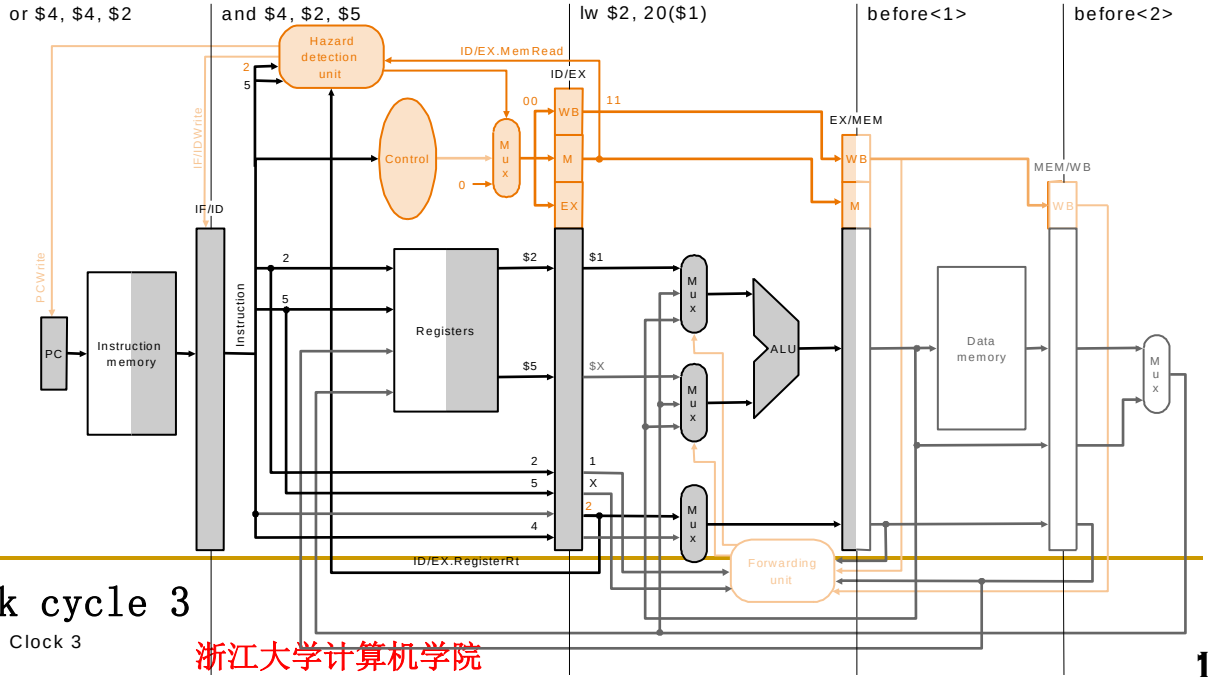
- **Execution example:**

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 2

Clock 2



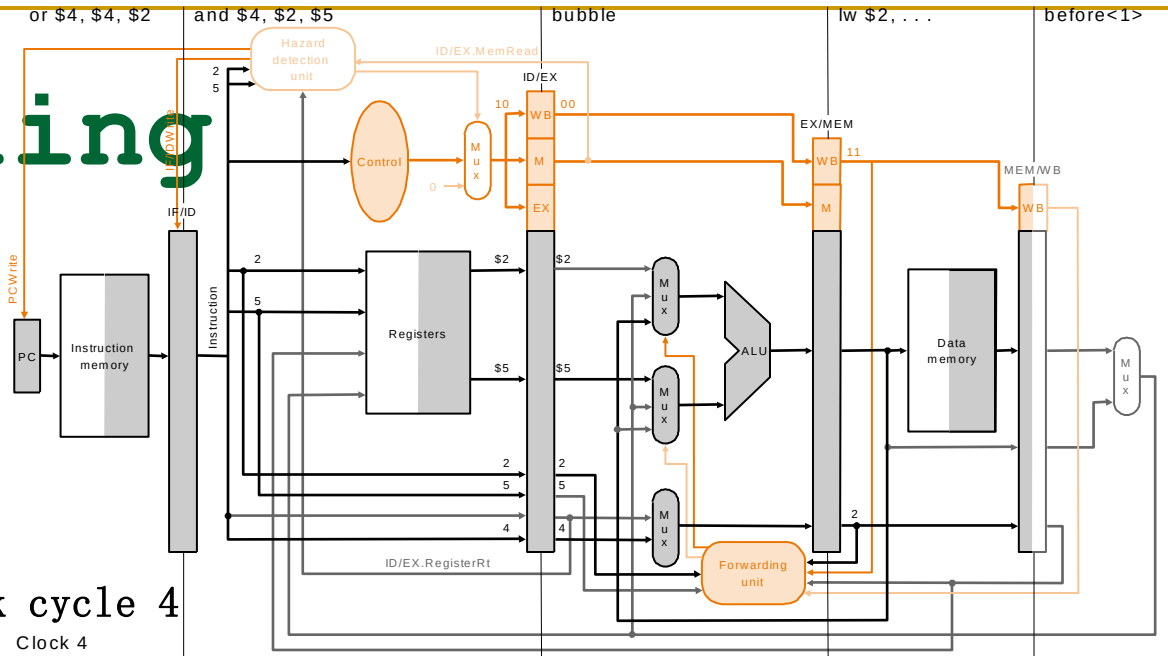
Clock cycle 3

Clock 3

Stalling

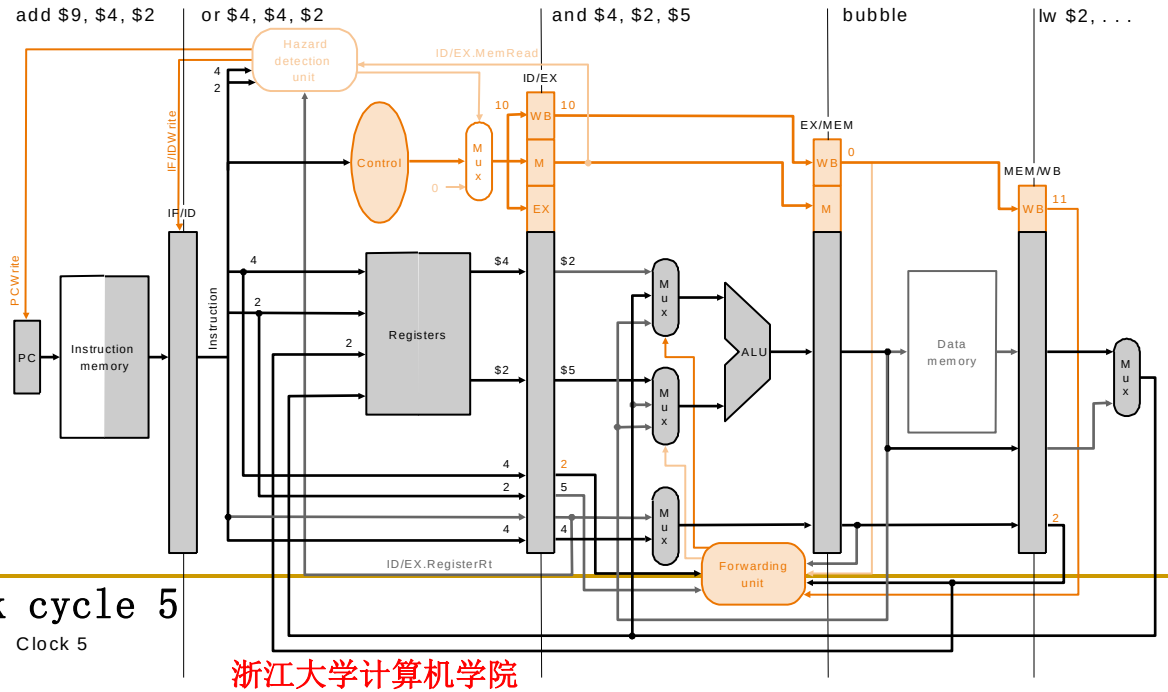
- **Execution example (cont.):**

```
lw   $2, 20($1)
and  $4, $2, $5
or   $4, $4, $2
add  $9, $4, $2
```



Clock cycle 4

Clock 4



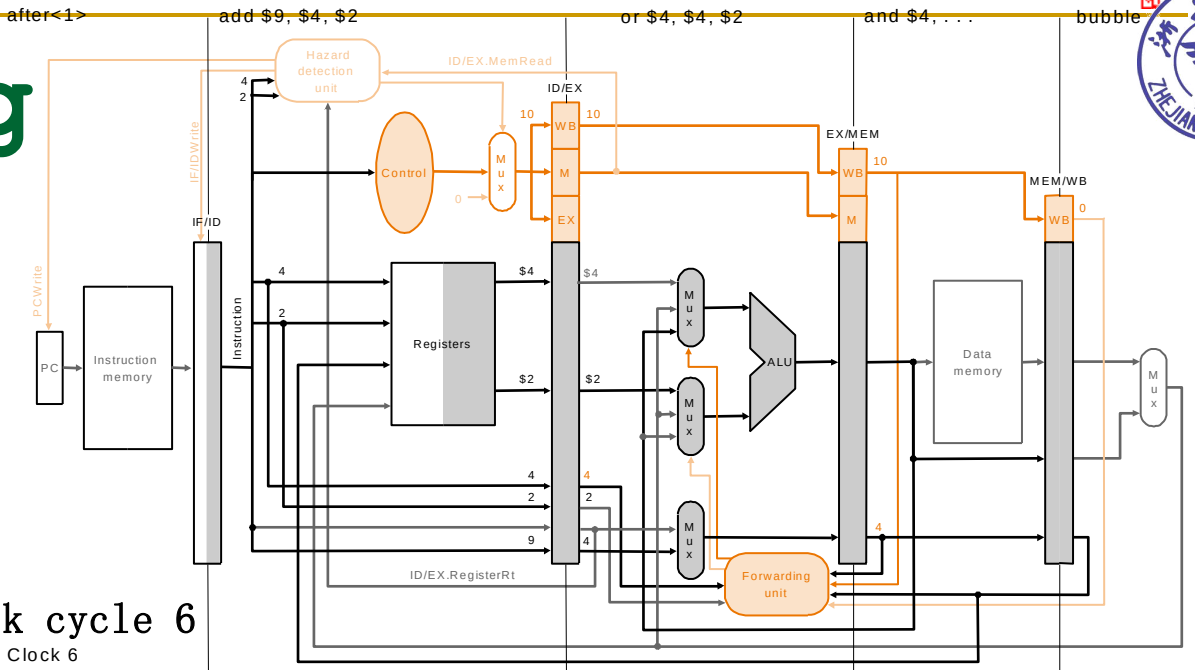
Clock cycle 5

Clock 5

Stalling

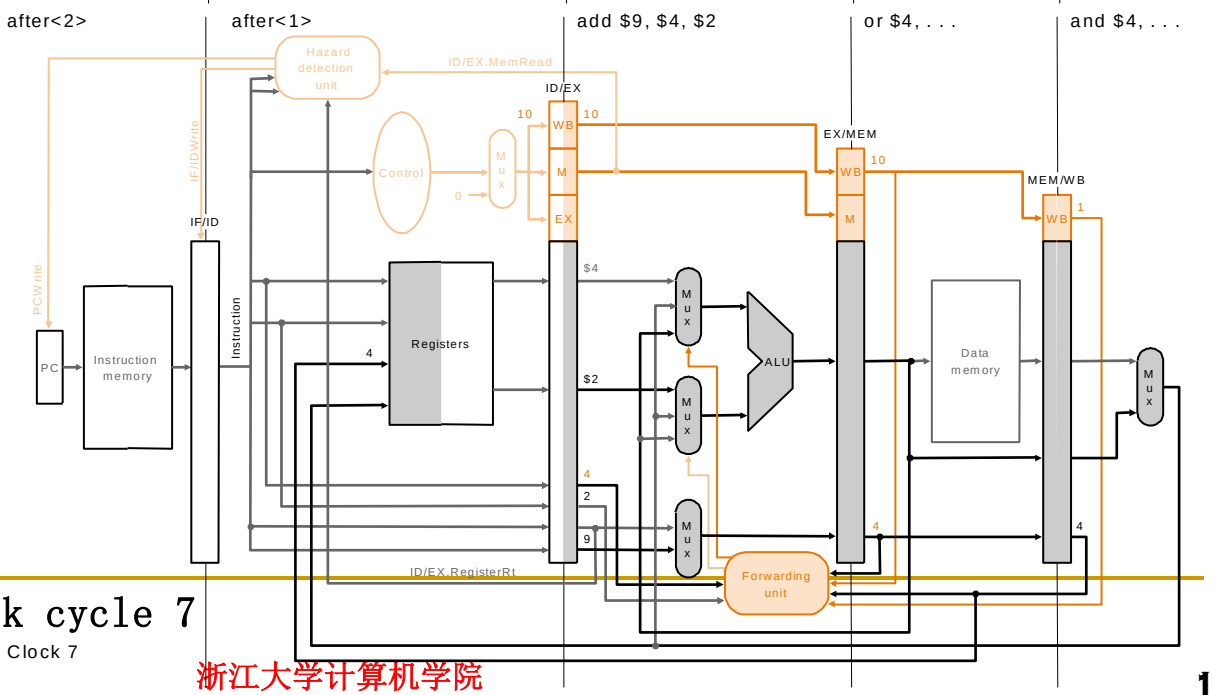
- **Execution example (cont.):**

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



Clock cycle 6

Clock 6



Clock cycle 7

Clock 7

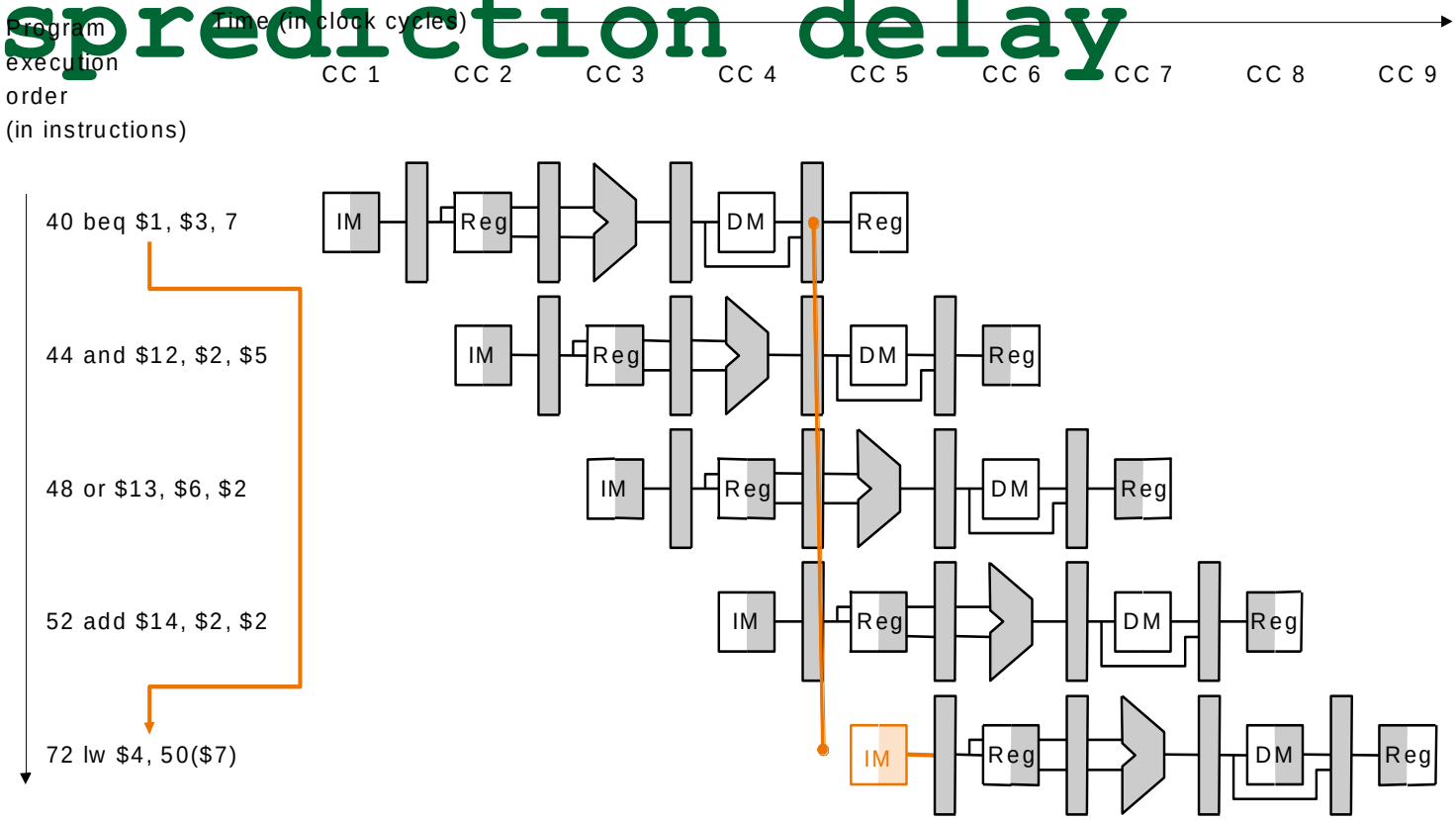
Control (or Branch) Hazards

- Problem with branches in the pipeline we have so far is that the *branch decision is not made till the MEM stage* – so *what instructions, if at all, should we insert into the pipeline following the branch instructions*?

- Possible solution: *stall* the pipeline till branch decision is known
  - not efficient, slow the pipeline significantly!

- Another solution: *predict* the branch outcome
  - e.g., always predict *branch-not-taken – continue with next sequential instructions*
  - if the prediction is wrong have to *flush* the pipeline behind the branch – discard instructions already fetched or decoded – and *continue execution at the branch target*

Predicting Branch-not-taken: Misprediction delay



The outcome of branch taken (prediction wrong) is decided only when **beq** is in the MEM stage, so the following three sequential instructions already in the pipeline have to be flushed and execution resumes at **lw**

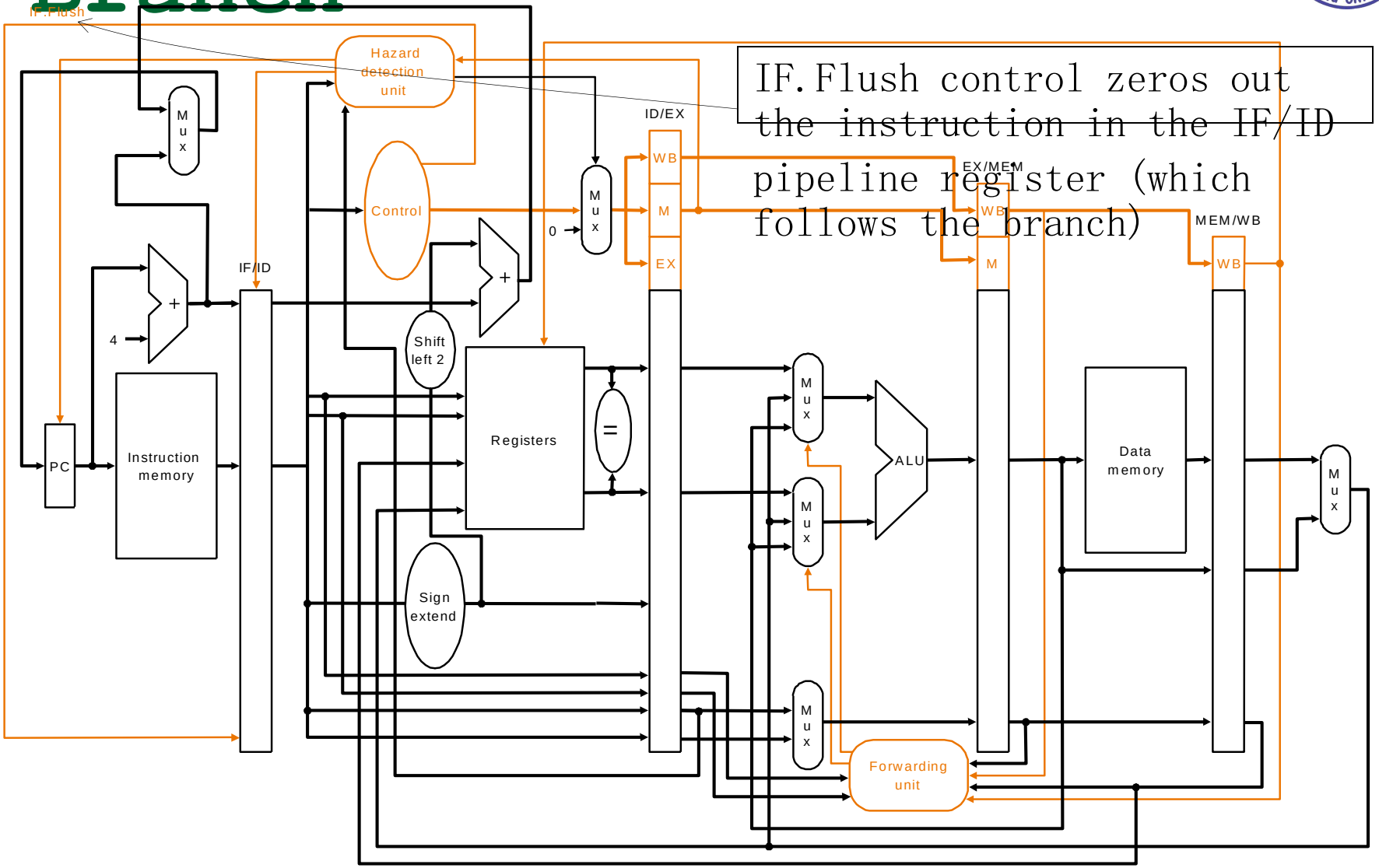
Optimizing the Pipeline to Reduce Branch Delay

- ***Move the branch decision*** **from the MEM stage (as in our current pipeline)** ***earlier to the ID stage***
  - ***calculating the branch target address*** **involves moving the branch adder from the MEM stage to the ID stage – inputs to this adder, the PC value and the immediate fields are already available in the IF/ID pipeline register**
  - ***calculating the branch decision*** **is efficiently done, e.g., for equality test, by XORing respective bits and then ORing all the results and inverting, rather than using the ALU to subtract and then test for zero (when there is a carry delay)**
    - **with the more efficient equality test we can put it in the ID stage without significantly lengthening this stage – remember the objective of pipeline**

Flushing on Misprediction

- Same strategy as for stalling on load-use data hazard…
- Zero out all the control values (or the instruction itself) in pipeline registers for the instructions following the branch that are already in the pipeline – effectively turning them into nops – so they are flushed
  - in the optimized pipeline, with branch decision made in the ID stage, we have to flush only one instruction in the IF stage – the branch delay penalty is then only one clock cycle

Optimized Datapath for Branch



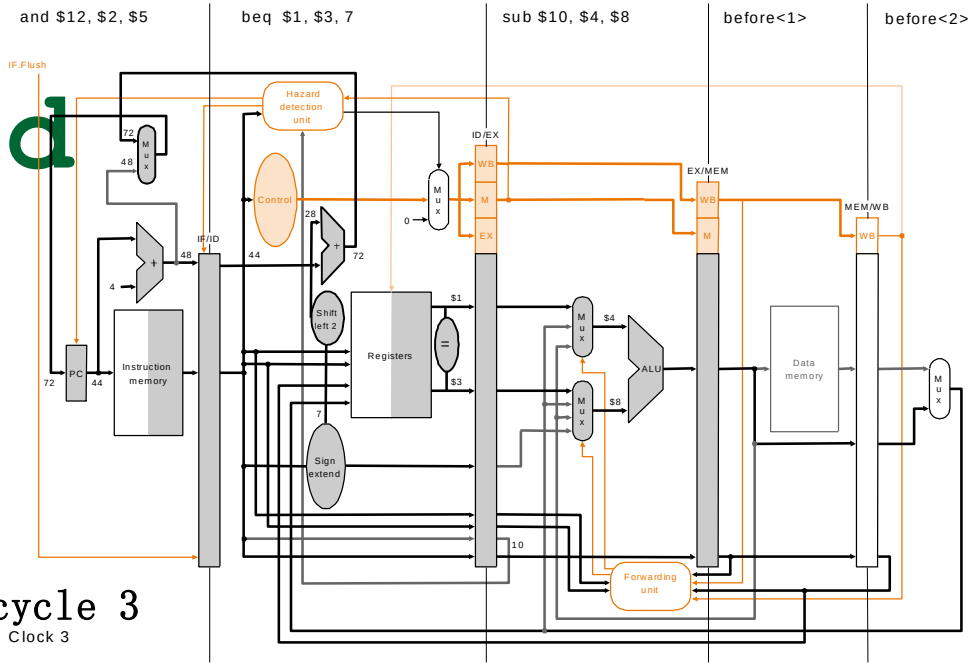
Branch decision is moved from the MEM stage to the ID stage - simplified drawing not showing enhancements to the forwarding and hazard detection units

Pipelined Branch

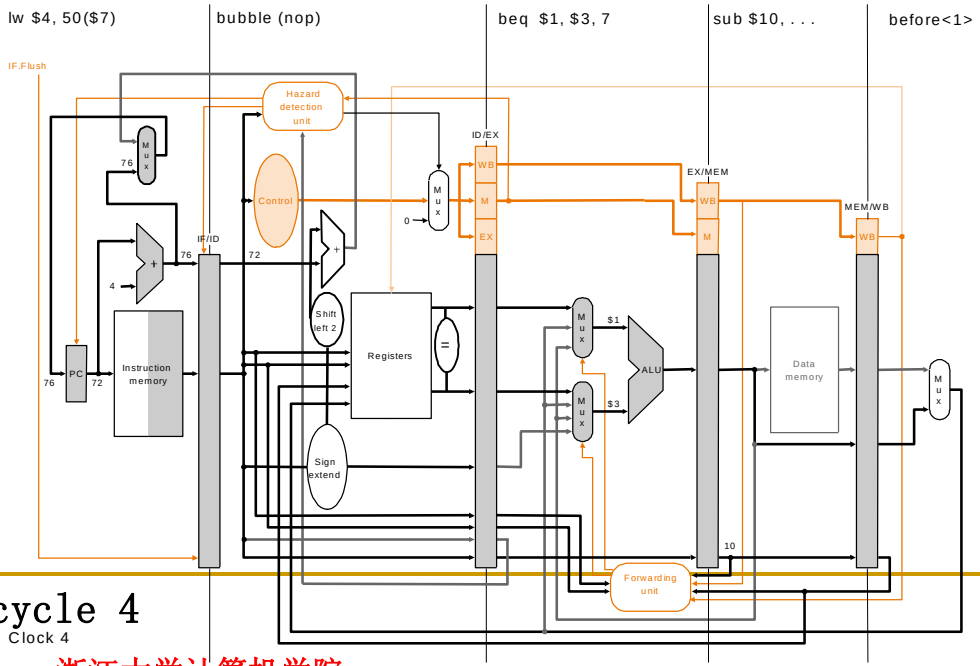
- **Execution example:**

```
36 sub $10, $4, $8
40 beq $1,  $3,  7
44 and $12  $2, $5
48 or  $13  $2, $6
52 add $14, $4, $2
56 slt $15, $6, $7
...
72 lw  $4,  50($7)
```

Optimized pipeline with only one bubble as a result of the taken branch



Clock cycle 3



Clock cycle 4

Simple Example: Comparing Performance

- *Compare performance for single-cycle, multicycle, and pipelined datapaths using the gcc instruction mix*
  - assume 2 ns for memory access, 2 ns for ALU operation, 1 ns for register read or write
  - assume gcc instruction mix 23% loads, 13% stores, 19% branches, 2% jumps, 43% ALU
  - for pipelined execution assume
    - 50% of the loads are followed immediately by an instruction that uses the result of the load
    - 25% of branches are mispredicted
    - branch delay on misprediction is 1 clock cycle
    - jumps always incur 1 clock cycle delay so their average time is 2 clock cycles

Simple Example: Comparing Performance

- *Single-cycle* (p. 373): average instruction time 8 ns
- *Multicycle* (p. 397): average instruction time 8.04 ns
- *Pipelined*:
  - loads use 1 cc (clock cycle) when no load-use dependency and 2 cc when there is dependency – given 50% of loads are followed by dependency the average cc per load is 1.5
  - stores use 1 cc each
  - branches use 1 cc when predicted correctly and 2 cc when not – given 25% misprediction average cc per branch is 1.25
  - jumps use 2 cc each
  - ALU instructions use 1 cc each
  - therefore, average CPI is

$$1.5 \times 23\% + 1 \times 13\% + 1.25 \times 19\% + 2 \times 2\% + 1 \times 43\% =$$


Thank You!
非常天空
浙江大学计算机学院