Single Cycle Processor Design

Agenda

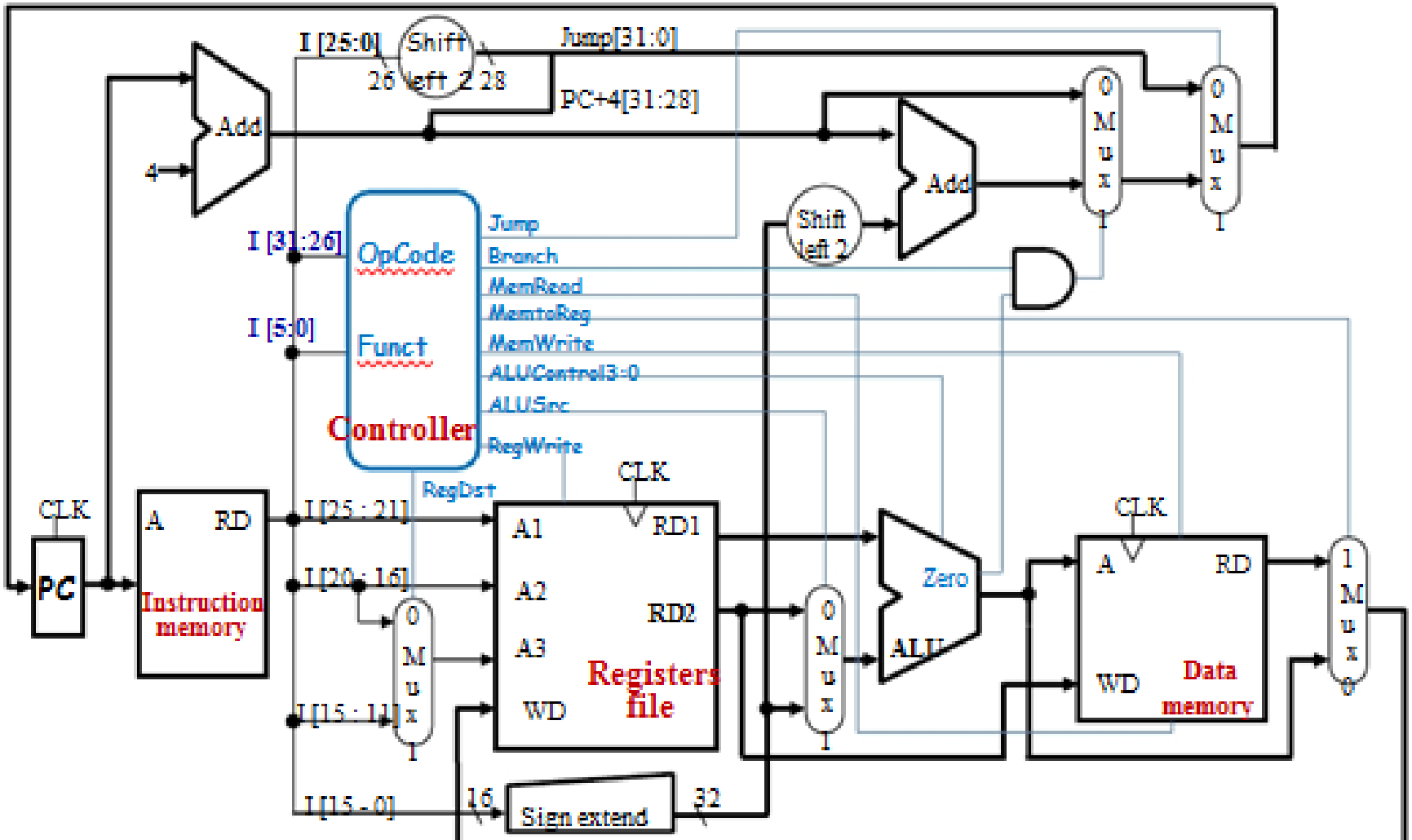
- Part-I:单周期处理器
  - Designing a Processor: Step-by-Step
  - Datapath Components and Clocking
  - 数据通路（Data Path）
  - 关键路径（Critical Path）
- Part-II ：实验
  - Lab2 MIPS处理器实现 (实验教程，第6章)

实验最初涉及的9条指令

指令	OpCode	Operation	Function	Type
LW	100011	Load word	--	I-Type
SW	101011	Store word	--	I-Type
BEQ	000100	Branch equal	--	I-Type
ADD	000000	Add	100000	R-Type
SUB	000000	Subtract	100010	R-Type
AND	000000	AND	100100	R-Type
OR	000000	OR	100101	R-Type
SLT	000000	Set on less than	101010	R-Type
J	000010	Jump to address	--	J-Type

注意： “--”表示无此项

Big Picture: Build a Processor



Single Cycle Processor

Agenda

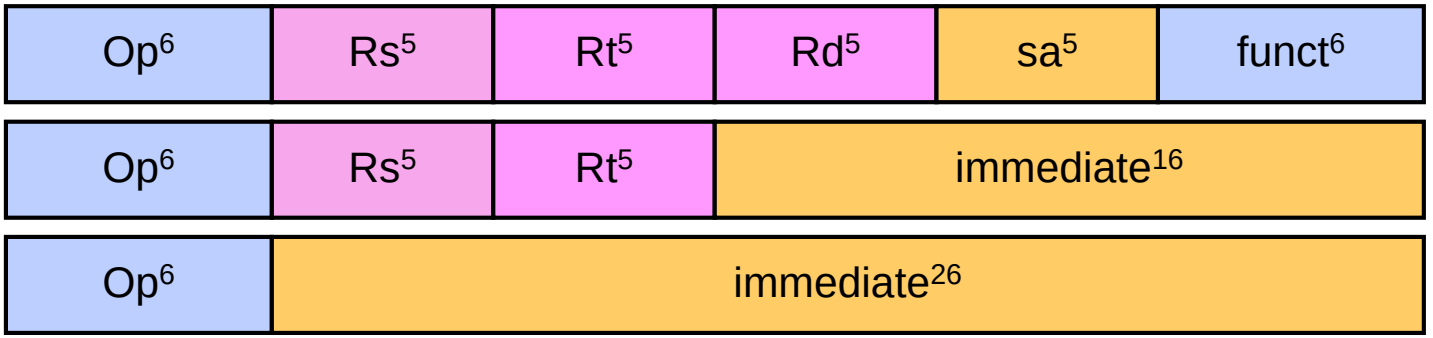
- Part-I:单周期处理器
  - Designing a Processor: Step-by-Step
  - Datapath Components and Clocking
  - 数据通路（Data Path）
  - 关键路径（Critical Path）
- Part-II :实验
  - Lab2 MIPS处理器实现 (实验教程，第6章)

Designing a Processor: Step-by-Step

- Analyze instruction set => datapath requirements
  - The meaning of each instruction is given by the register transfers
  - Datapath must include storage elements for ISA registers
  - Datapath must support each register transfer
- Select datapath components and clocking methodology
- Assemble datapath meeting the requirements
- Analyze implementation of each instruction
  - Determine the setting of control signals for register transfer
- Assemble the control logic **(汇聚控制逻辑)**

Review of MIPS Instruction Formats

- All instructions are 32-bit wide
- Three instruction formats: R-type, I-type, and J-type



  - Op^6: 6-bit opcode of the instruction
  - Rs^5, Rt^5, Rd^5: 5-bit source and destination register numbers
  - sa^5: 5-bit shift amount used by shift instructions
  - $funct^6$: 6-bit function field for R-type instructions
  - $immediate^{16}$: 16-bit immediate value or address offset
  - $immediate^{26}$: 26-bit target address of the jump instruction

MIPS Subset of Instructions

- Only a subset of the MIPS instructions are considered
  - ALU instructions (R-type): **add, sub, and, or, xor, slt**
  - Immediate instructions (I-type): **addi, slti, andi, ori, xori**
  - Load and Store (I-type): **lw, sw**
  - Branch (I-type): **beq, bne**
  - Jump (J-type): **j**
- This subset does not include all the integer instructions
- But sufficient to illustrate design of datapath and control
- Concepts used to implement the MIPS subset are used to construct a broad spectrum of computers

Details of the MIPS Subset

Instruction	Meaning	Format					
add rd, rs, rt	addition	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x20
sub rd, rs, rt	subtraction	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x22
and rd, rs, rt	bitwise and	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x24
or rd, rs, rt	bitwise or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x25
xor rd, rs, rt	exclusive or	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x26
slt rd, rs, rt	set on less than	$op^6 = 0$	rs^5	rt^5	rd^5	0	0x2a
addi rt, rs, im^{16}	add immediate	0x08	rs^5	rt^5	im^{16}		
slti rt, rs, im^{16}	slt immediate	0x0a	rs^5	rt^5	im^{16}		
andi rt, rs, im^{16}	and immediate	0x0c	rs^5	rt^5	im^{16}		
ori rt, rs, im^{16}	or immediate	0x0d	rs^5	rt^5	im^{16}		
xori rt, im^{16}	xor immediate	0x0e	rs^5	rt^5	im^{16}		
lw rt, im^{16}(rs)	load word	0x23	rs^5	rt^5	im^{16}		
sw rt, im^{16}(rs)	store word	0x2b	rs^5	rt^5	im^{16}		
beq rs, rt, im^{16}	branch if equal	0x04	rs^5	rt^5	im^{16}		
bne rs, rt, im^{16}	branch not equal	0x05	rs^5	rt^5	im^{16}		
j im^{26}	jump	0x02	im^{26}				

Register Transfer Level (RTL)

- RTL is a description of data flow between registers
- RTL gives a meaning to the instructions
- All instructions are fetched from memory at address PC

Instruction	RTL Description	
ADD	Reg(Rd) ← Reg(Rs) + Reg(Rt);	PC ← PC + 4
SUB	Reg(Rd) ← Reg(Rs) – Reg(Rt);	PC ← PC + 4
ORI	Reg(Rt) ← Reg(Rs) | zero_ext(Im16);	PC ← PC + 4
LW	Reg(Rt) ← MEM[Reg(Rs) + sign_ext(Im16)];	PC ← PC + 4
SW	MEM[Reg(Rs) + sign_ext(Im16)] ← Reg(Rt);	PC ← PC + 4
BEQ	if (Reg(Rs) == Reg(Rt)) PC ← PC + 4 + 4 × sign_extend(Im16) else PC ← PC + 4	

Instructions are Executed in Steps

- **R-type**

Fetch instruction:	Instruction ← MEM[PC]
Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
Execute operation:	ALU_result ← func(data1, data2)
Write ALU result:	Reg(Rd) ← ALU_result
Next PC address:	PC ← PC + 4

- **I-type**

Fetch instruction:	Instruction ← MEM[PC]
Fetch operands:	data1 ← Reg(Rs), data2 ← Extend(imm16)
Execute operation:	ALU_result ← op(data1, data2)
Write ALU result:	Reg(Rt) ← ALU_result
Next PC address:	PC ← PC + 4

- **BEQ**

Fetch instruction:	Instruction ← MEM[PC]
Fetch operands:	data1 ← Reg(Rs), data2 ← Reg(Rt)
Equality:	zero ← subtract(data1, data2)
Branch:	if (zero) PC ← PC + 4 + 4×sign_ext(imm16)
	else PC ← PC + 4

Instruction Execution – cont'd

- **LW**

Fetch instruction:	Instruction ← MEM[PC]
Fetch base register:	base ← Reg(Rs)
Calculate address:	address ← base + sign_extend(imm16)
Read memory:	data ← MEM[address]
Write register Rt:	Reg(Rt) ← data
Next PC address:	PC ← PC + 4

- **SW**

Fetch instruction:	Instruction ← MEM[PC]
Fetch registers:	base ← Reg(Rs), data ← Reg(Rt)
Calculate address:	address ← base + sign_extend(imm16)
Write memory:	MEM[address] ← data
Next PC address:	PC ← PC + 4

concatenation

- **Jump**

Fetch instruction:	Instruction ← MEM[PC]
Target PC address:	target ← PC[31:28] || Imm26 || '00'
Jump:	PC ← target

Requirements of the Instruction Set

- Memory
  - Instruction memory where instructions are stored
  - Data memory where data is stored
- Registers
  - 31 × 32-bit general purpose registers, R0 is always zero
  - Read source register Rs
  - Read source register Rt
  - Write destination register Rt or Rd
- Program counter PC register and Adder to increment PC
- Sign and Zero extender for immediate constant
- ALU for executing instructions

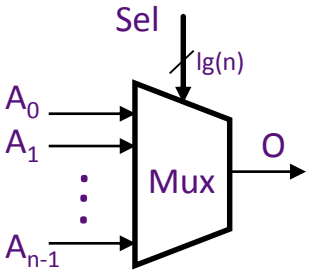
Next . . .

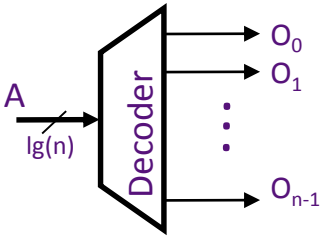
- **Part-I:单周期处理器**
  - Designing a Processor: Step-by-Step
  - **Datapath Components and Clocking**
  - 数据通路（Data Path）
  - 关键路径（Critical Path）
- **Part-II :实验**
  - Lab2 MIPS处理器实现 (实验教程，第6章)

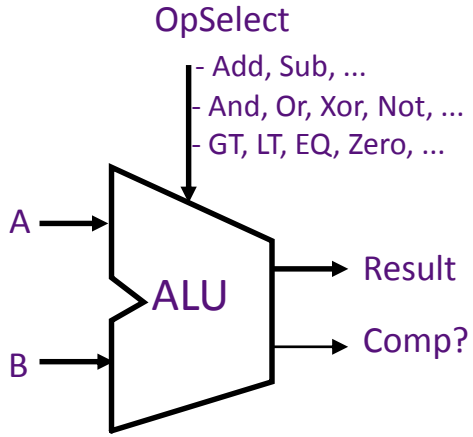
Hardware Elements

❿ Combinational circuits

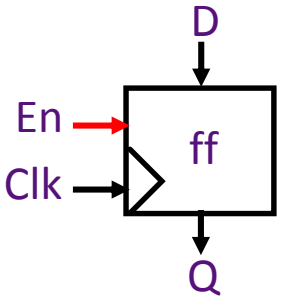
- **Mux, Decoder, ALU, ...**

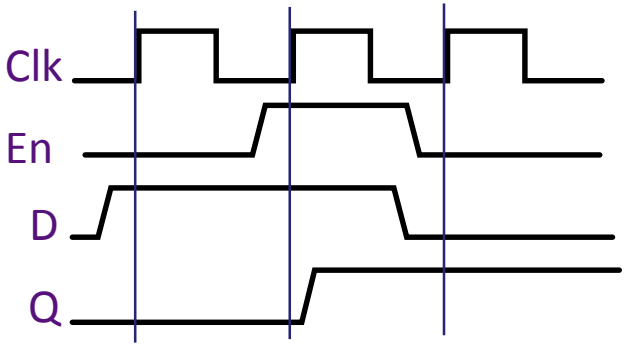






- **Synchronous state elements**
  - **Flipflop, Register, Register file, SRAM, DRAM**

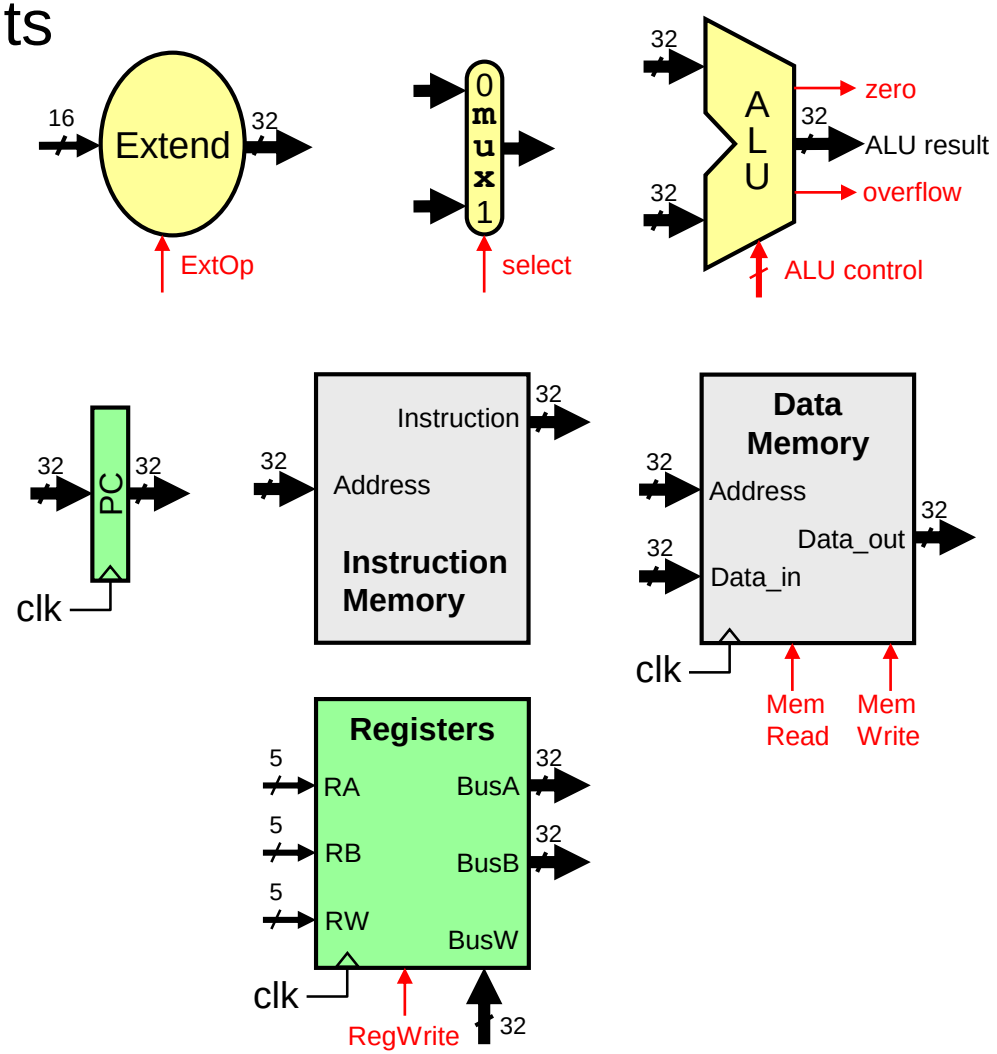




Edge-triggered: Data is sampled at the rising edge

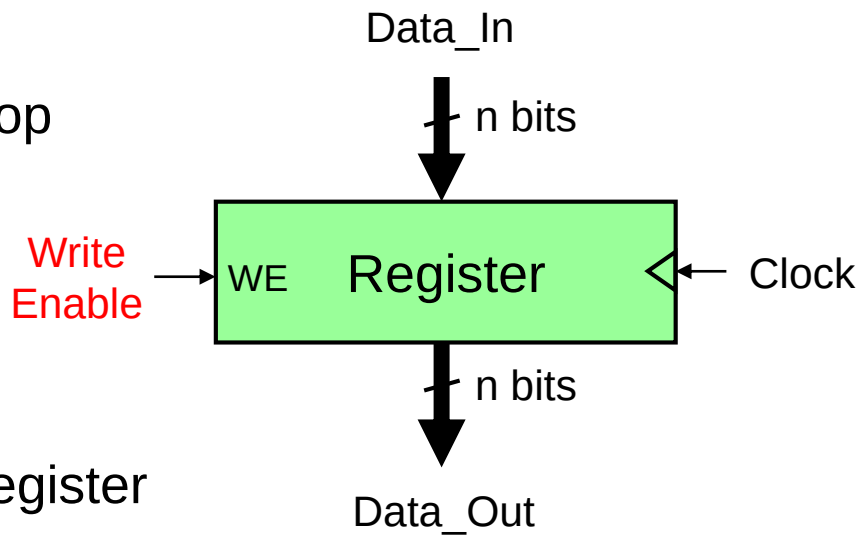
Components of the Datapath

- Combinational Elements
  - ALU, Adder
  - Immediate extender
  - Multiplexers
- Storage Elements
  - Instruction memory
  - Data memory
  - PC register
  - Register file
- Clocking methodology
  - Timing of writes



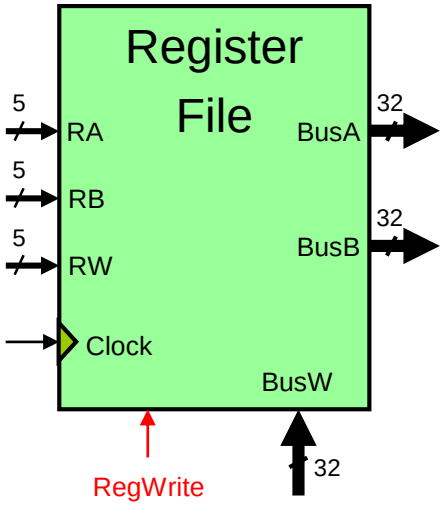
Register Element

- Register
  - Similar to the D-type Flip-Flop
- n-bit input and output
- Write Enable (WE):
  - Enable / disable writing of register
  - Negated (0): Data_Out will not change
  - Asserted (1): Data_Out will become Data_In after clock edge
- Edge triggered Clocking
  - Register output is modified at clock edge



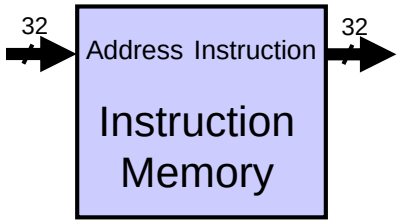
MIPS Register File

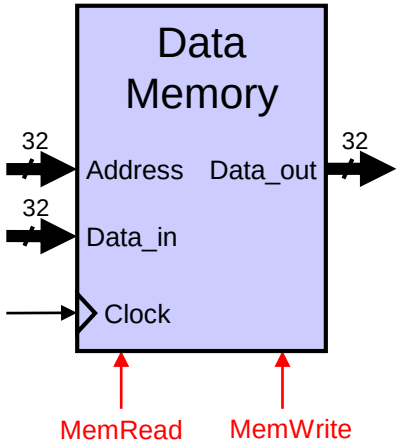
- Register File consists of 32 × 32-bit registers
  - BusA and BusB: 32-bit output busses for reading 2 registers
  - BusW: 32-bit input bus for writing a register when RegWrite is 1
  - Two registers read and one written in a cycle
- Registers are selected by:
  - RA selects register to be read on BusA
  - RB selects register to be read on BusB
  - RW selects the register to be written
- Clock input
  - The clock input is used ONLY during write operation
  - During read, register file behaves as a combinational logic block
    - RA or RB valid => BusA or BusB valid after access time



Instruction and Data Memories

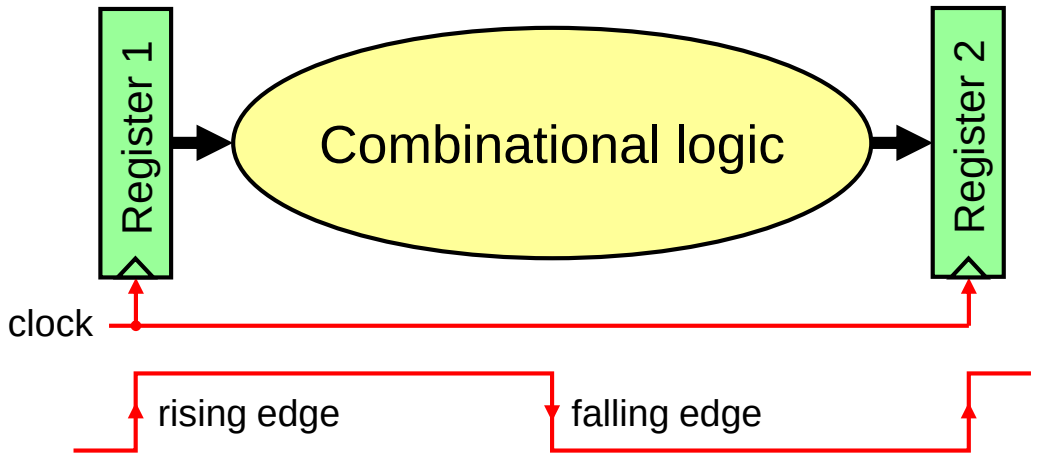
- Instruction memory needs only provide read access
  - Because datapath does not write instructions
  - Behaves as combinational logic for read
  - Address selects Instruction after access time
- Data Memory is used for load and store
  - MemRead: enables output on Data_out
    - Address selects the word to put on Data_out
  - MemWrite: enables writing of Data_in
    - Address selects the memory word to be written
    - The Clock synchronizes the write operation
- Separate instruction and data memories
  - Later, we will replace them with caches





Clocking Methodology (时钟控制方法)

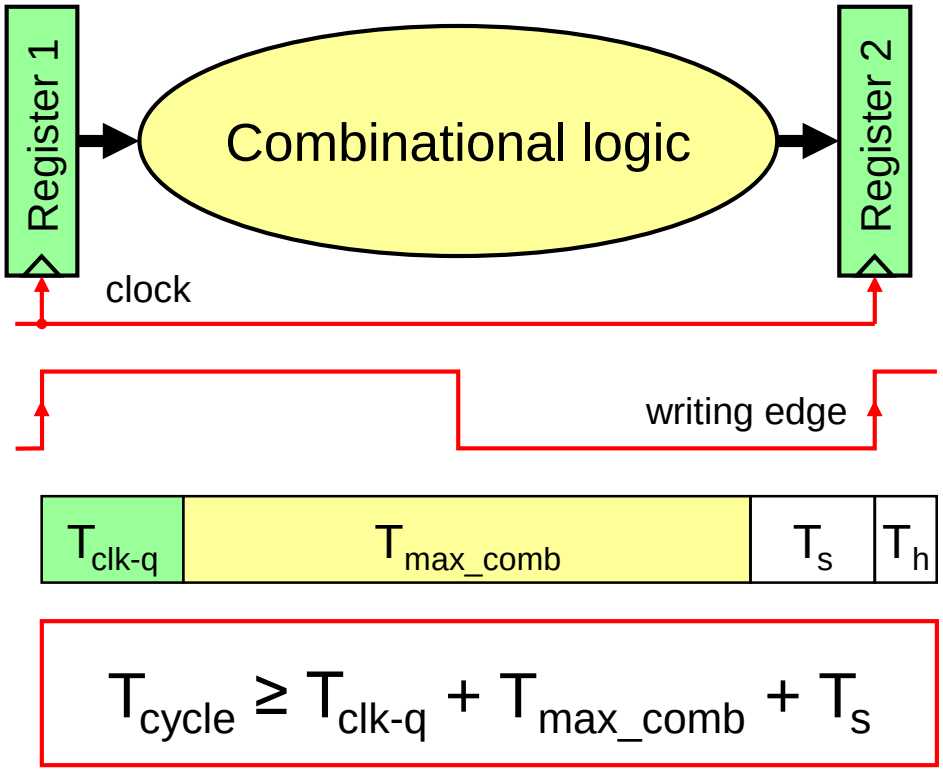
- Clocks are needed in a sequential logic to decide when a state element (register) should be updated
- To ensure correctness, a clocking methodology defines when data can be written and read



- We assume edge-triggered clocking
- All state changes occur on the same clock edge
- Data must be valid and stable before arrival of clock edge
- Edge-triggered clocking allows a register to be read and written during same clock cycle

Determining the Clock Cycle

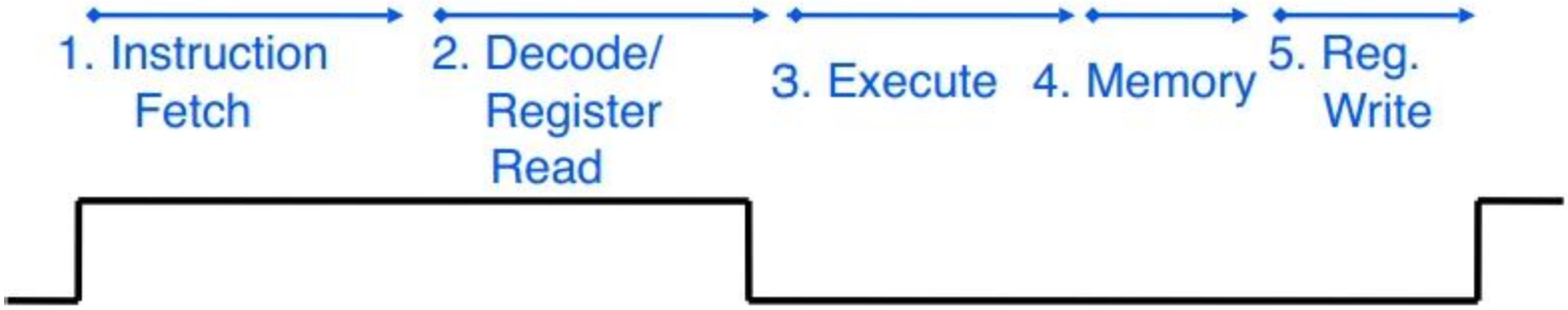
- With edge-triggered clocking, the clock cycle must be long enough to accommodate the path from one register through the combinational logic to another register



- T_{clk-q} : clock to output delay through register
- T_{max_comb} : longest delay through combinational logic
- T_s : setup time that input to a register must be stable before arrival of clock edge
- T_h: hold time that input to a register must hold after arrival of clock edge
- Hold time (T_h) is normally satisfied since $T_{clk-q} > T_h$

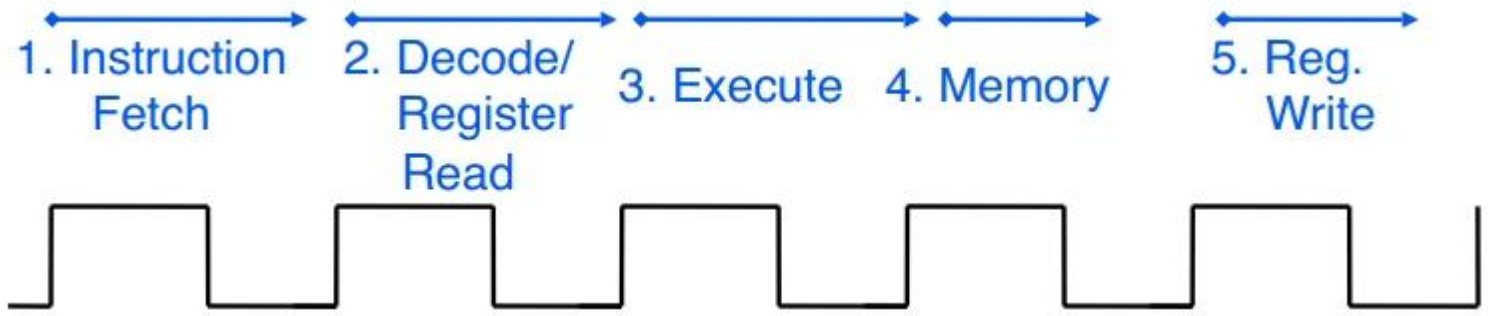
CPU Clocking

- **Single Cycle CPU** : All stages of an instruction are completed within one long clock cycle.
  - The clock cycle is made sufficient long to allow each instruction to complete all stages without interruption and within one cycle.



CPU Clocking

- **Multiple-cycle CPU: Only one stage of instruction per** clock **cycle.**
  - **The clock is made as long as the slowest stage.**



Several significant advantages over single cycle execution: Unused stages in a particular instruction can be skipped OR instructions can be pipelined (overlapped, 重疊).

Next . . .

- Part-I:单周期处理器
  - Designing a Processor: Step-by-Step
  - Datapath Components and Clocking
  - 数据通路（Data Path）
  - 关键路径（Critical Path）
- Part-II :实验
  - Lab2 MIPS处理器实现 (实验教程，第6章)

1. Load and Store Word

- Load Word Instruction (Word = 4 bytes in MIPS)

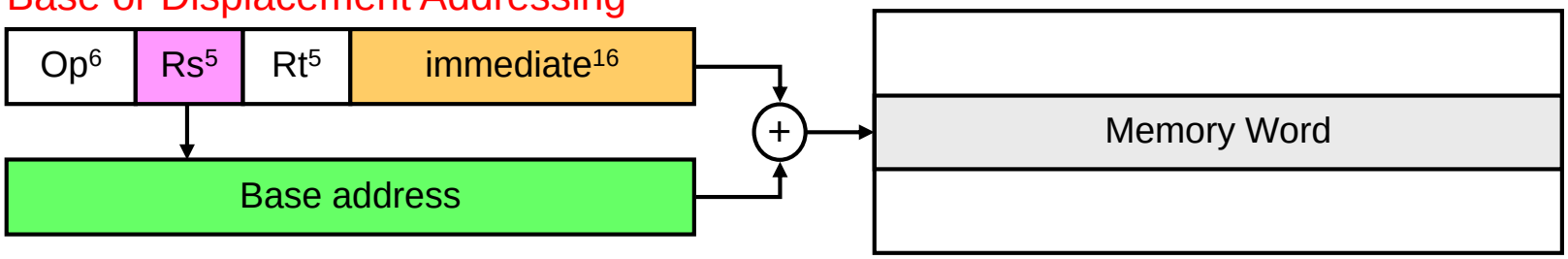
  `lw Rt, imm16(Rs)  # Rt ← MEMORY[Rs+imm16]`

- Store Word Instruction

  `sw Rt, imm16(Rs)  # Rt → MEMORY[Rs+imm16]`

- Base or Displacement （偏移量）addressing is used
  - Memory Address = Rs (base) + Immediate^{16} (displacement)
  - Immediate^{16} is sign-extended to have a signed displacement

Base or Displacement Addressing

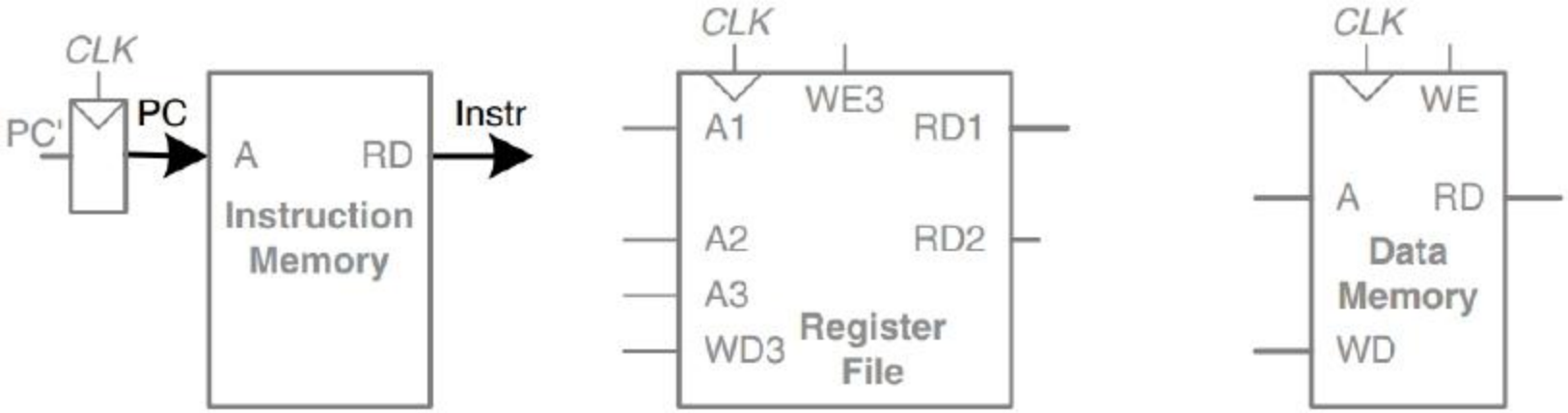


Single-Cycle Datapath: lw fetch

- First consider executing lw

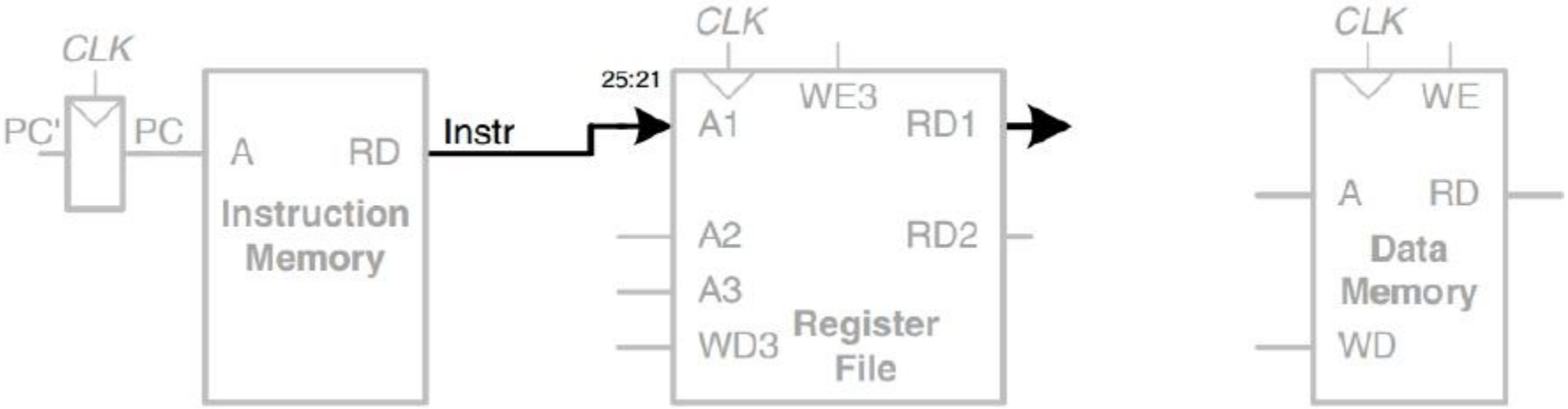
  R[rt] ← DMEM[R[rs] + sign_ext(Imm16)]

- STEP1: Fetch instruction



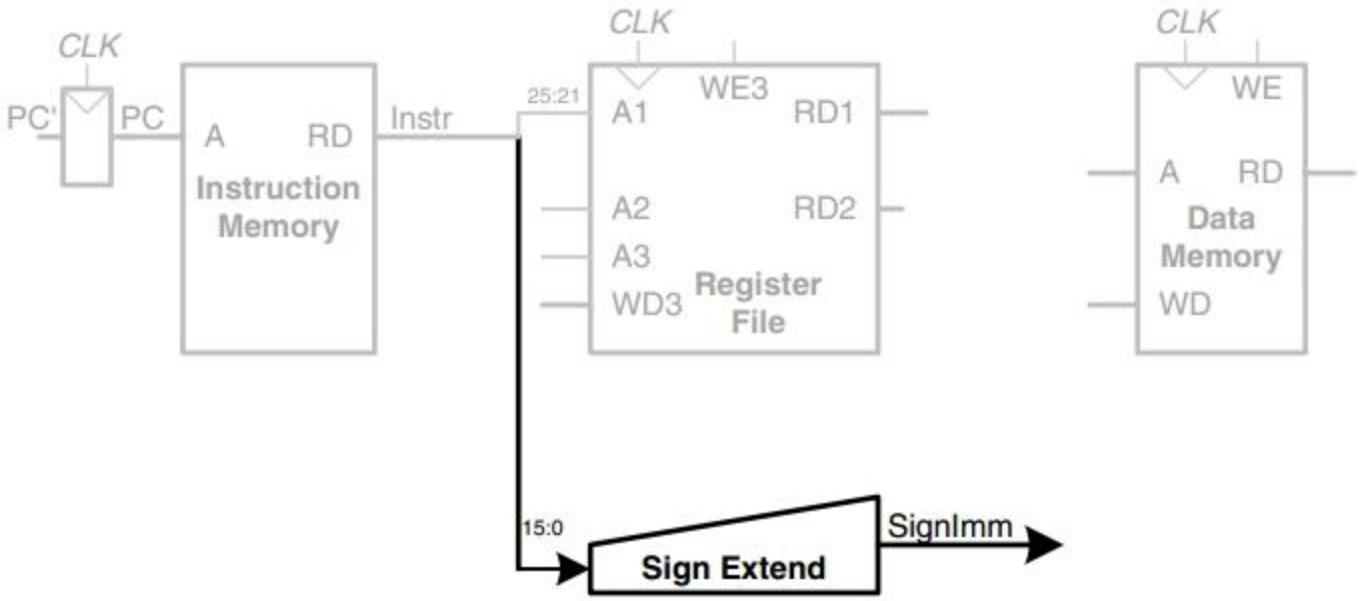
Single-Cycle Datapath: lw register read

- STEP 2: Read source operands from register file



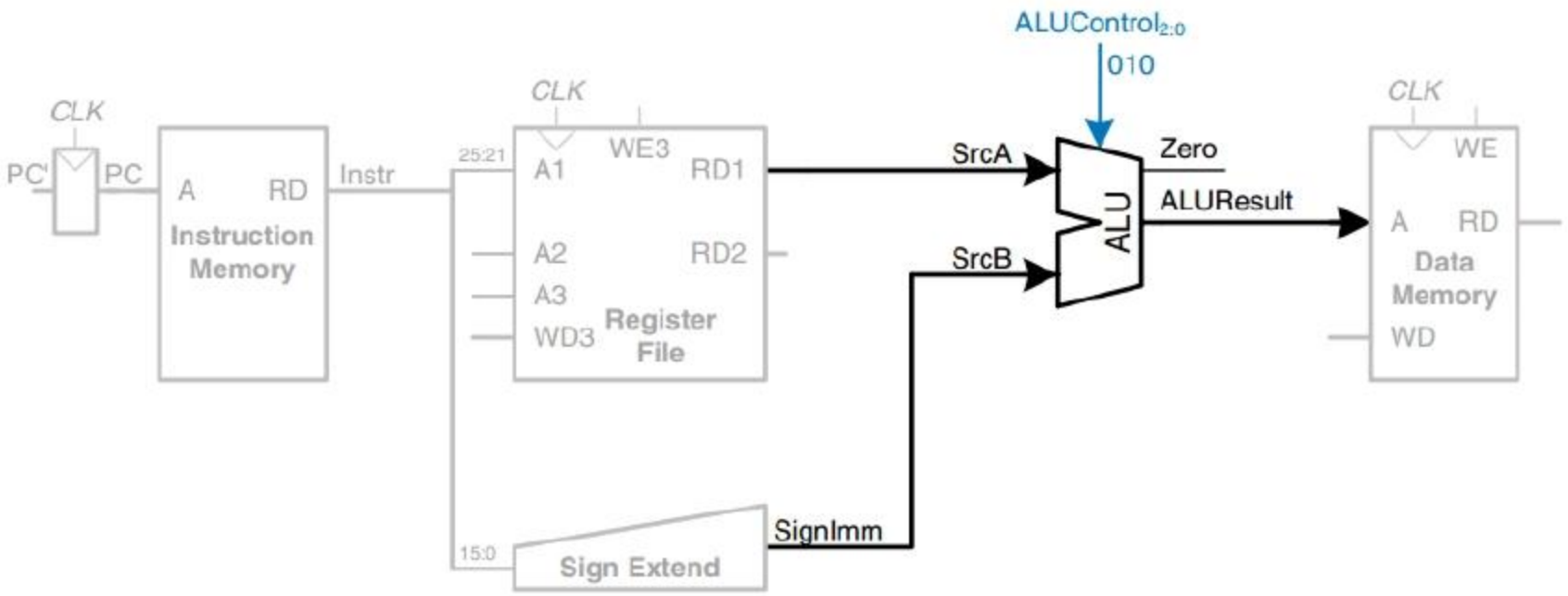
Single-Cycle Datapath: lw immediate

- STEP 3: Sign-extend the immediate



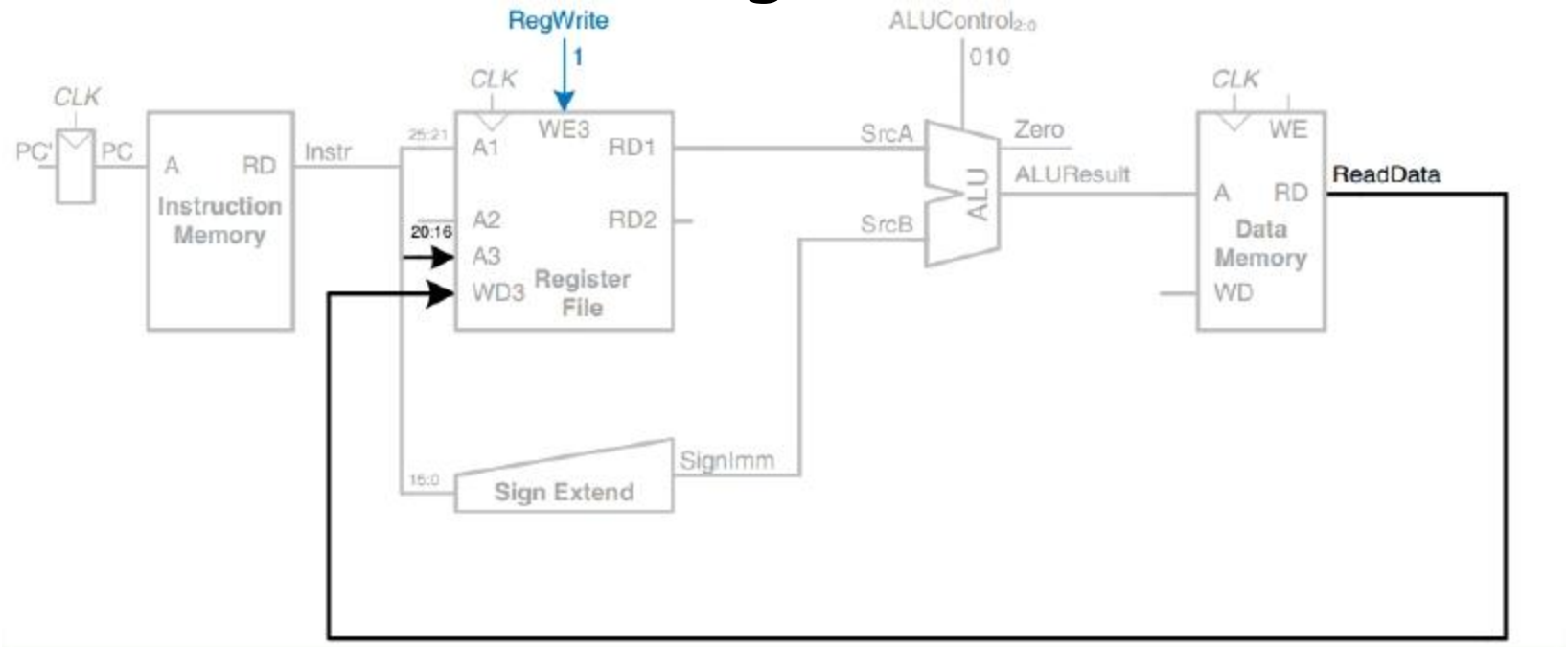
Single-Cycle Datapath: lw address

- STEP 4: Compute the memory address



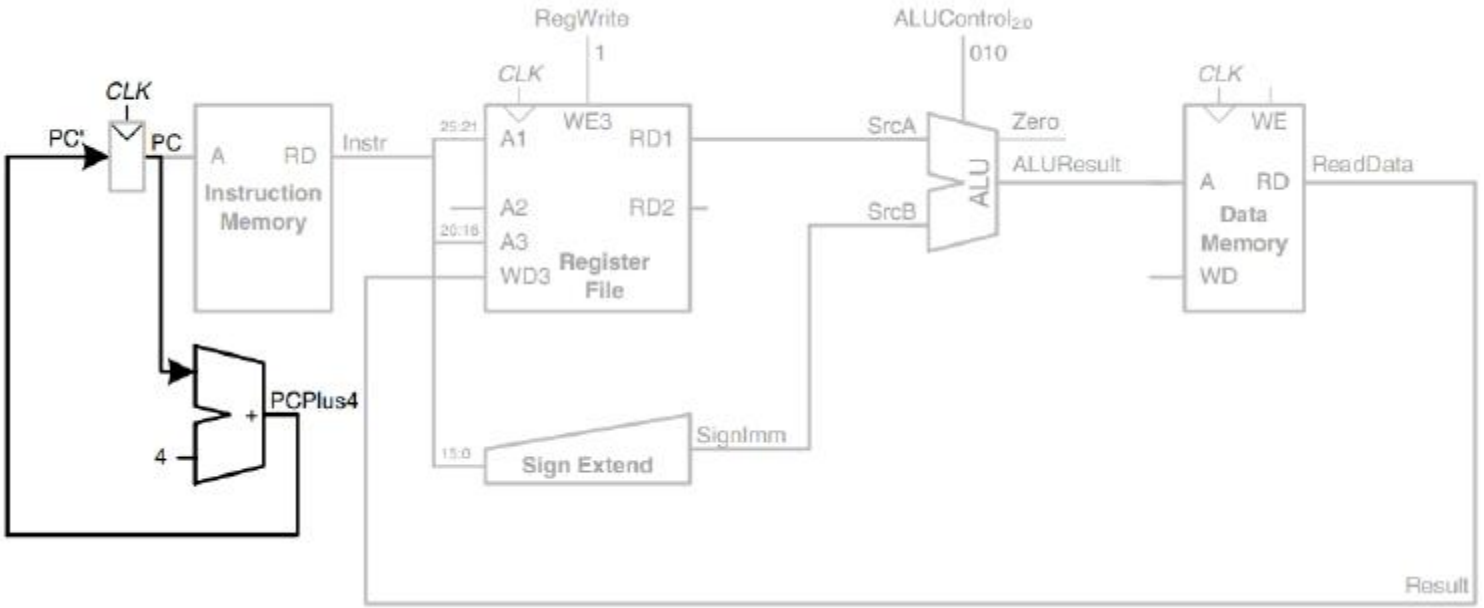
Single-Cycle Datapath: lw memory read

- STEP 5: Read data from memory and write it back to register file



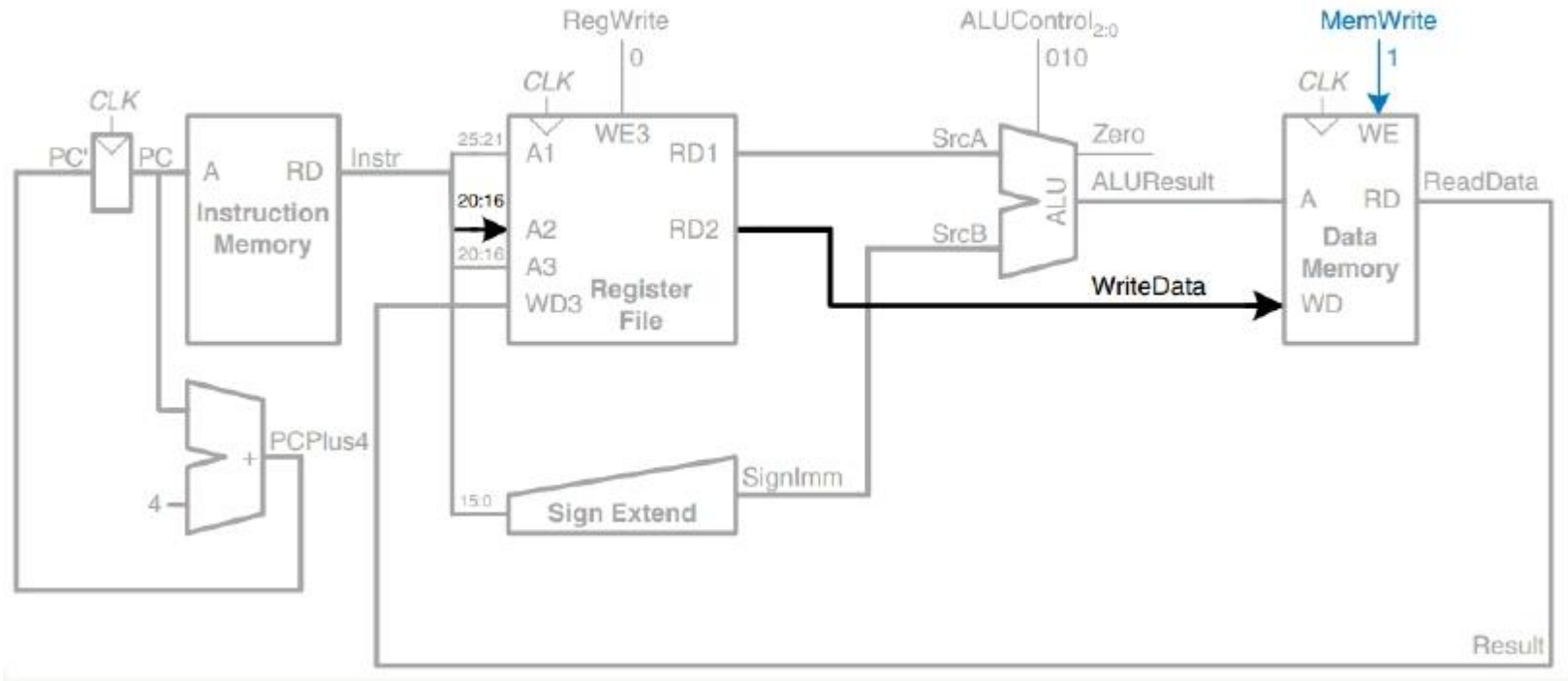
Single-Cycle Datapath: lw PC increment

- STEP 6: Determine the address of the next instruction



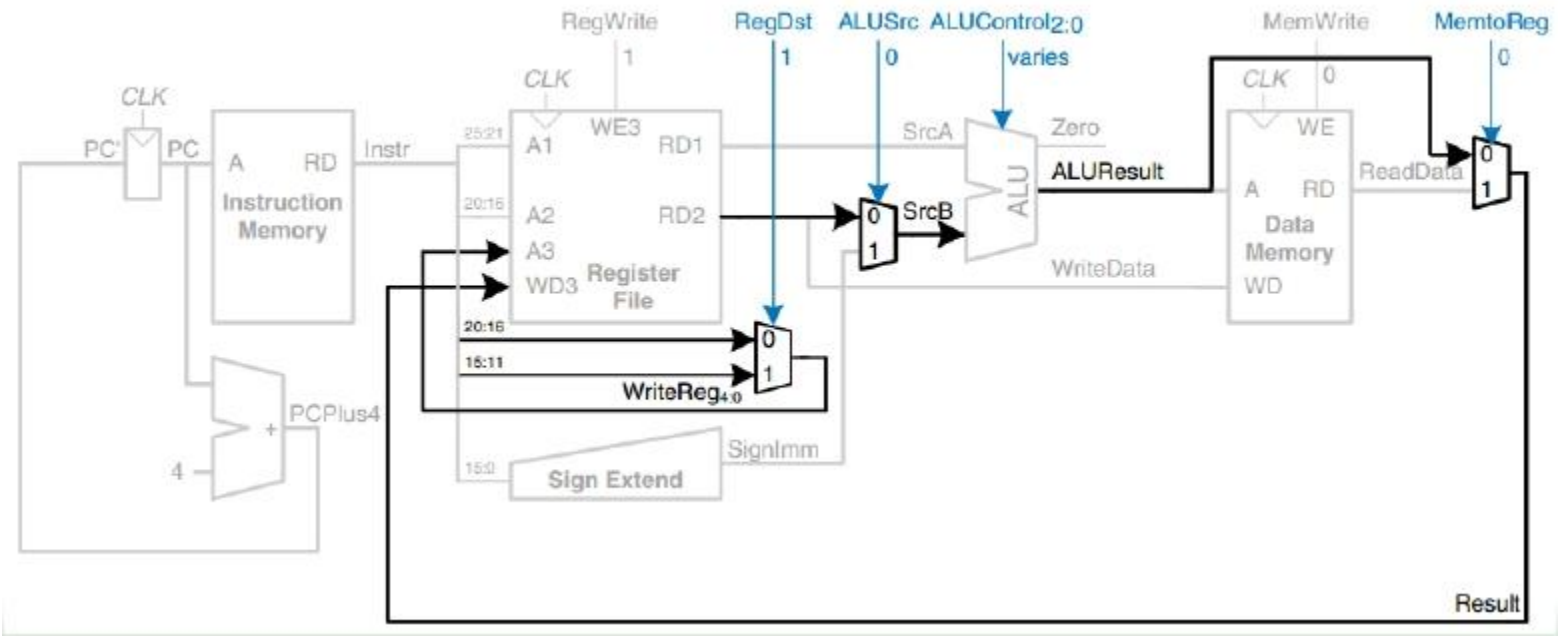
Single-Cycle Datapath: sw

- Write data in rt to memory



2. Single-Cycle Datapath:
R-type instructions

- Read from rs and rt
- Write ALUResult to register file
- Write to rd (instead of rt)



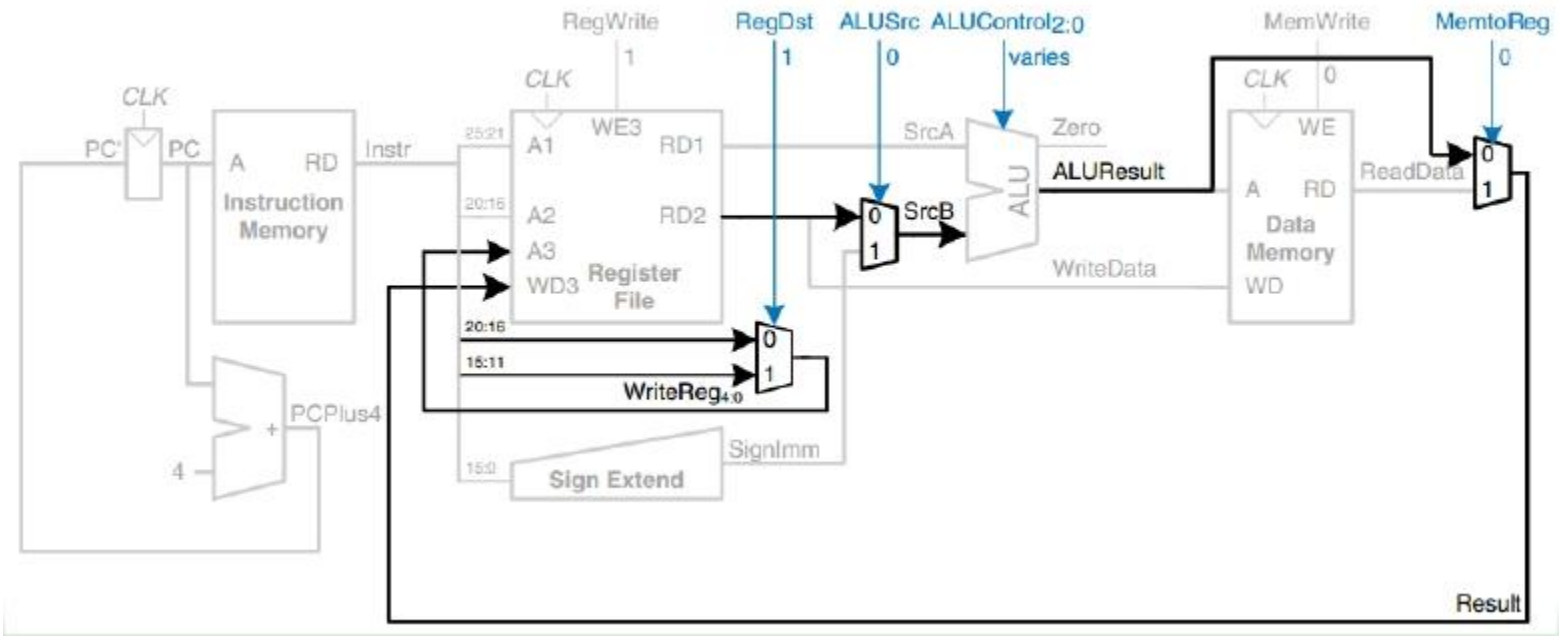
Integer Add /Subtract Instructions

Instruction	Meaning	R-Type Format					
add $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x20
addu $s1, $s2, $s3	$s1 = $s2 + $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x21
sub $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x22
subu $s1, $s2, $s3	$s1 = $s2 – $s3	op = 0	rs = $s2	rt = $s3	rd = $s1	sa = 0	f = 0x23

- add & sub: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register
- addu & subu: same operation as add & sub
  - However, no arithmetic exception can occur
  - **Overflow is ignored**
- Many programming languages ignore overflow
  - The + operator is translated into **addu**
  - The – operator is translated into **subu**

Single-Cycle Datapath:
R-type instructions

- Read from rs and rt
- Write ALUResult to register file
- Write to rd (instead of rt)



3. Conditional Branch Instructions

- MIPS compare and branch instructions:

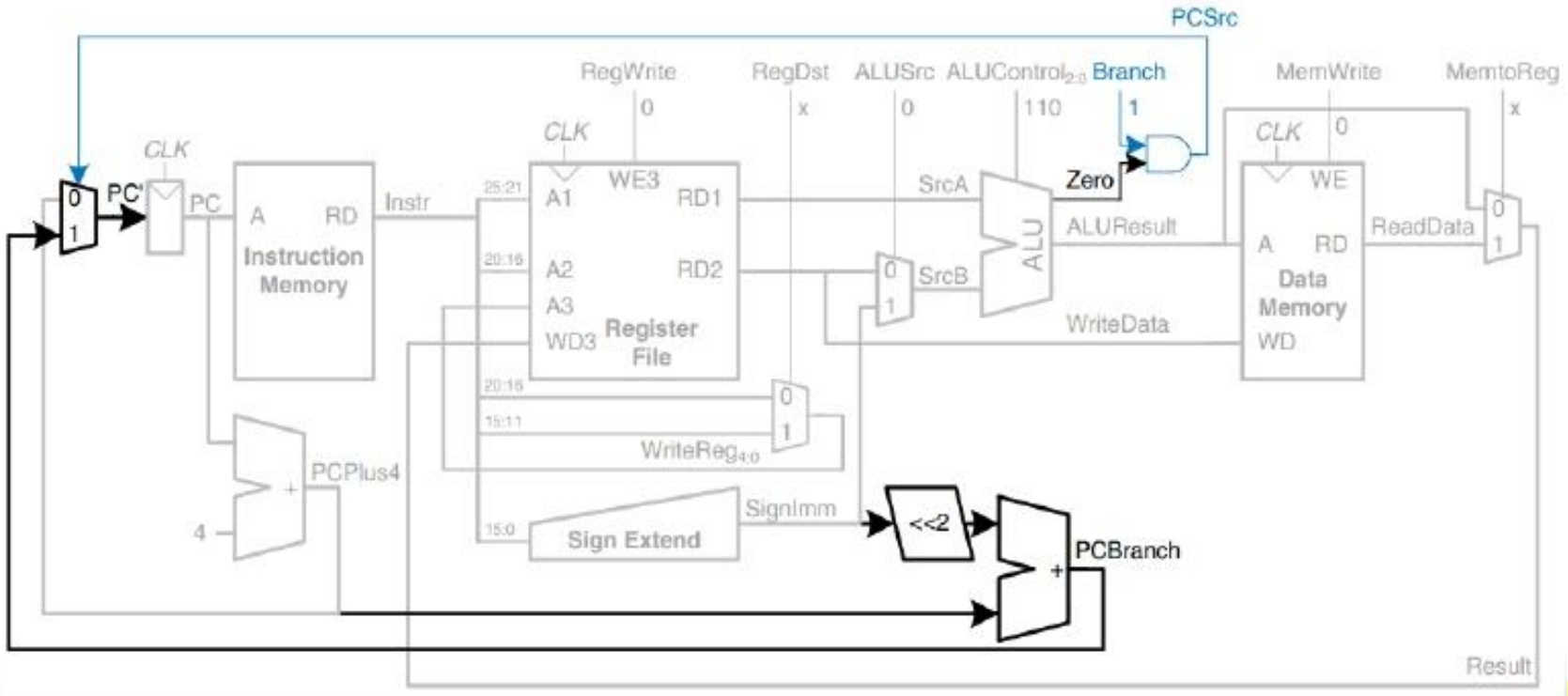
  `beq Rs,Rt,label` branch to **`label`** if (**`Rs == Rt`**)

  `bne Rs,Rt,label` branch to **`label`** if (**`Rs != Rt`**)

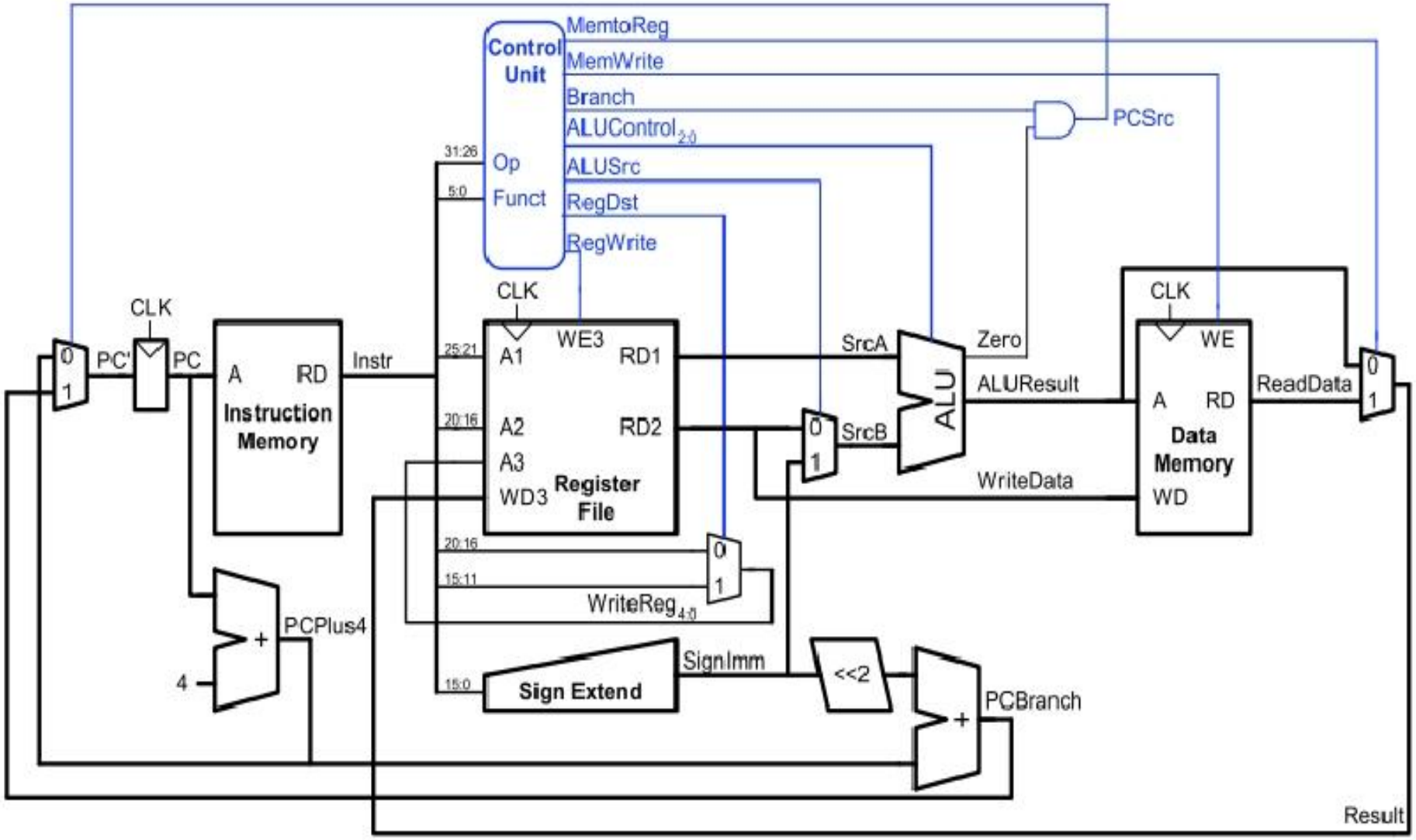
Single-Cycle Datapath: beq

if (R[rs] == R[rt]) then PC ← PC + 4 + {sign_ext(Imm16), 00}

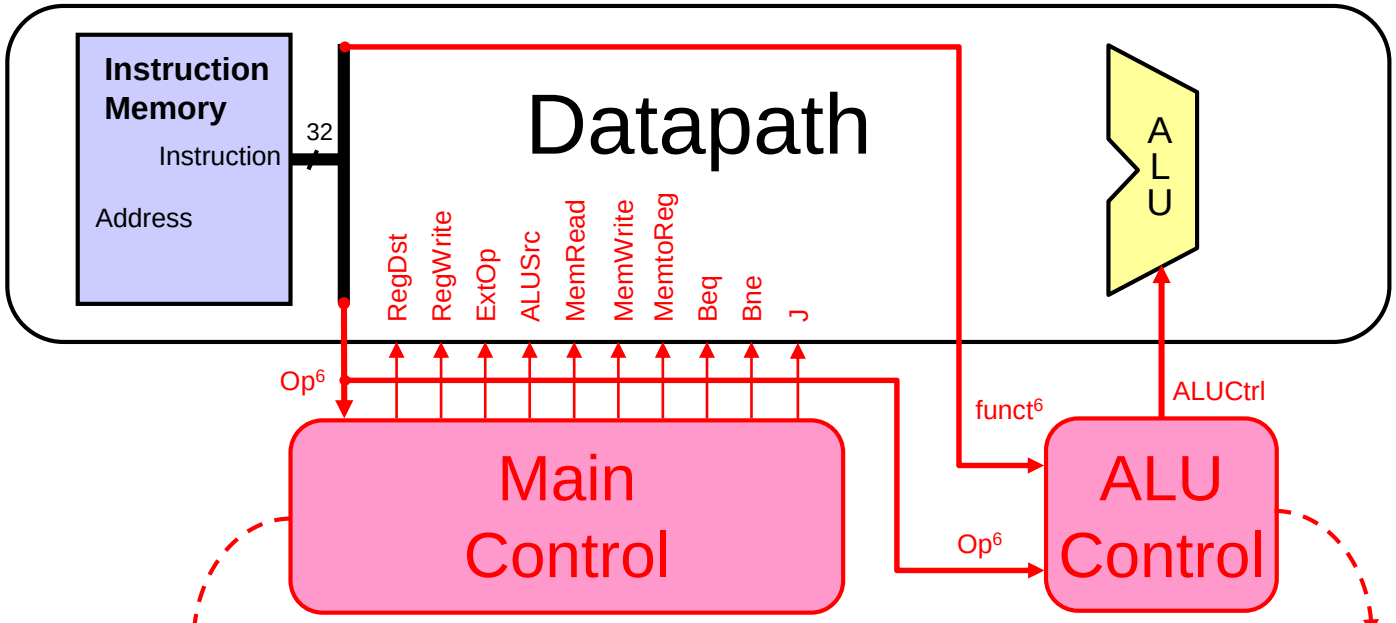
- Determine whether values in rs and rt are equal
- Calculate branch target address:
- BTA = (sign-extended immediate << 2) + (PC+4)



Complete Single-Cycle Processor



Main Control and ALU Control



Main Control Input:

- 6-bit opcode field from instruction

Main Control Output:

- 10 control signals for the Datapath

ALU Control Input:

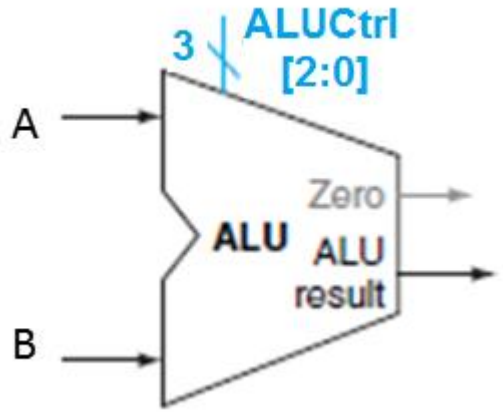
- 6-bit opcode field from instruction
- 6-bit function field from instruction

ALU Control Output:

- ALUCtrl signal for ALU

3.4.1 ALU

- ALU 输入
  - 运算操作数A、B
  - ALU控制信号
- ALU 输出
  - 运算结果ALU Result
  - Zero信号(例如分支指令)

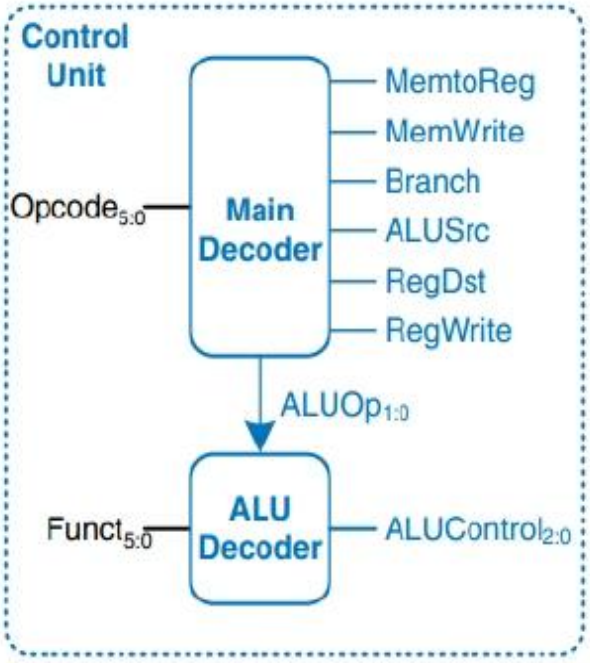


ALUCtrl [2:0]	Operation
000	A & B
001	A | B
010	A + B
110	A - B
111	SLT

- 对于lw、sw，ALU做加法运算，获取存储地址。
- 对于R型指令，根据指令最低6位的funct字段，确定ALU执行5种运算之一（与、或、加、减、小于则置1）
- 对beq指令，ALU做减法运算

Control Unit: ALU Decoder（表3-2）

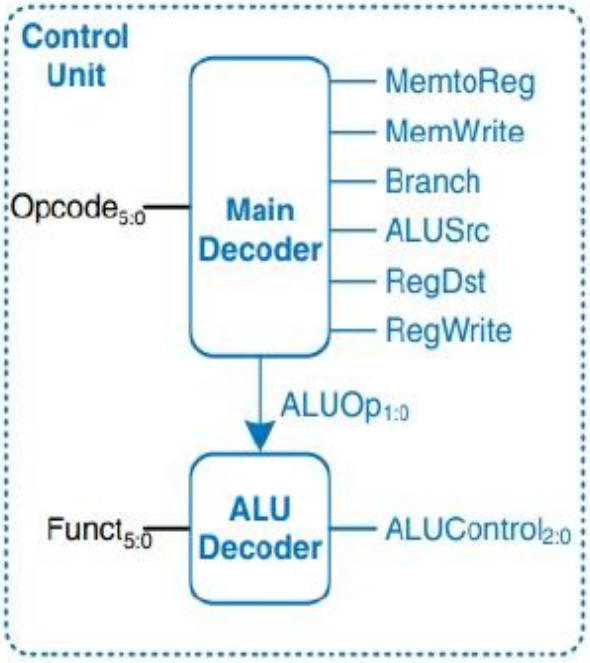
- 对于lw、sw，ALU做加法运算，获取存储地址。
- 对于R型指令，根据指令最低6位的funct字段，确定ALU执行5种运算之一（与、或、加、减、小于则置1）
- 对beq指令，ALU做减法运算
- 这3类指令，可以用ALUOp[1:0]进行区分



指令	ALUOp [1:0]	6位功能码	ALU的运算	ALU控制信号
LW	00	XXXXXX	加	0010
SW	00	XXXXXX	加	0010
BEQ	01	XXXXXX	减	0110
R型指令	10	100000	加	0010
R型指令	10	100010	减	0110
R型指令	10	100100	与	0000
R型指令	10	100101	或	0001
R型指令	10	101010	小于设置	0111

Control Unit: ALU Decoder（表3-3）

- 对于lw、sw，ALU做加法运算，获取存储地址。
- 对于R型指令，根据指令最低6位的funct字段，确定ALU执行5种运算之一（与、或、加、减、小于则置1）
- 对beq指令，ALU做减法运算
- 这3类指令，可以用ALUOp[1:0]进行区分

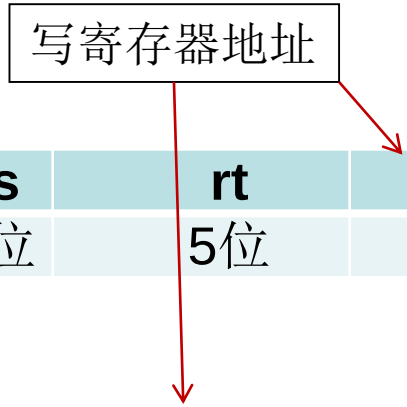


ALUOp	Funct	ALUControl
00	X	010 (add)
X1	X	110 (subtract)
1X	100000 (add)	010 (add)
1X	100010 (sub)	110 (subtract)
1X	100100 (and)	000 (and)
1X	100101 (or)	001 (or)
1X	101010 (slt)	111 (set less than)

表3-3 ALU 控制信号真值表

操作码		功能码						ALU控制信号
位1	位0	位5	位4	位3	位2	位1	位0	
0	0	X	X	X	X	X	X	0010
0	1	X	X	X	X	X	X	0110
1	0	X	X	0	0	0	0	0010
1	0	X	X	0	0	1	0	0110
1	0	X	X	0	1	0	0	0000
1	0	X	X	0	1	0	1	0001
1	0	X	X	1	0	1	0	0111

主控制器



R 型指令

op	rs	rt	rd	sa	funct
6位	5位	5位	5位	5位	6位

I 型指令

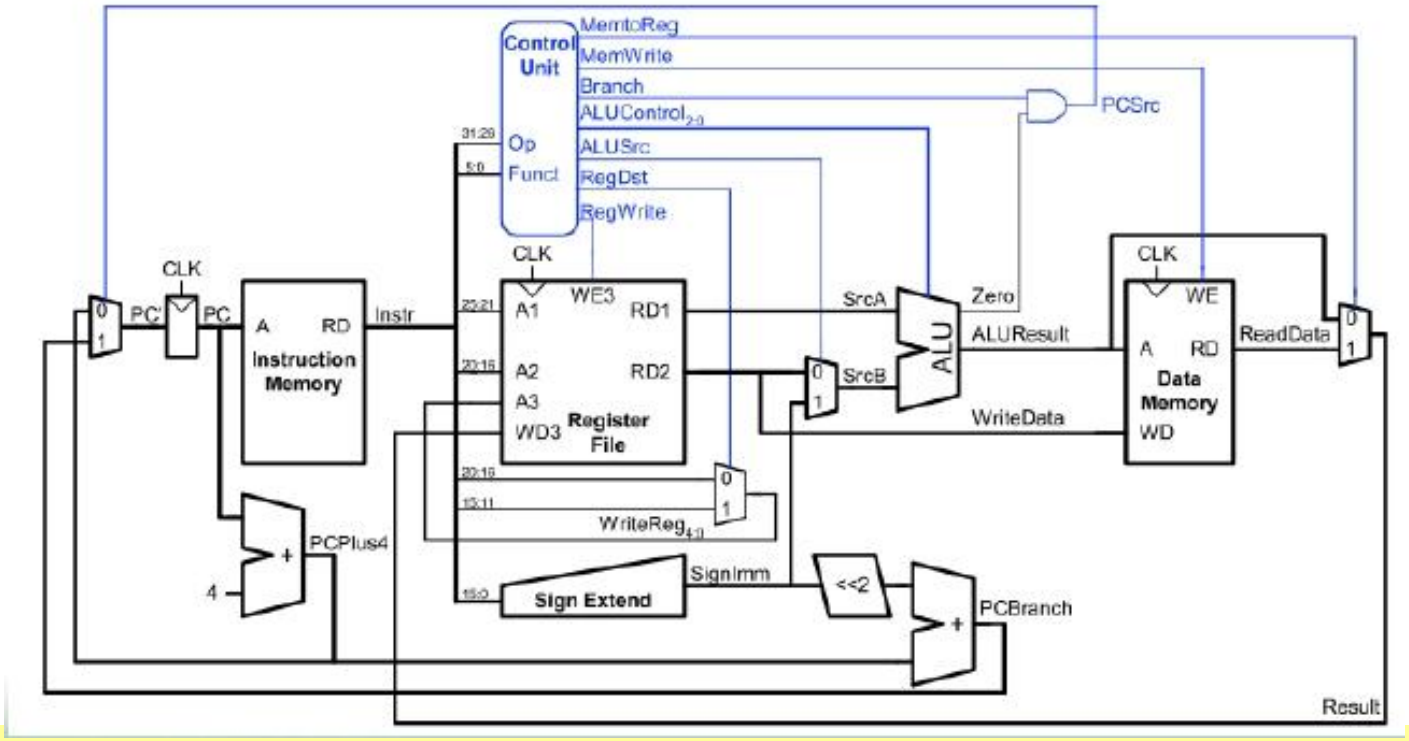
op	rs	rt	constant address
6位	5位	5位	16位

J 型指令

op	constant address
6位	26位

3.4.2 Control Main Decode

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01



控制信号定义

控制信号名称	1	0
RegDst	表示写寄存器的地址来自指令[15:11]	来自指令[20:16]
Jump	表示PC的值来自伪直接寻址	来自另一个复用器
Branch	表示下一级复用器的输入来PC相对寻址加法器	来自PC+4
MemToReg	表示写寄存器数据来自存储器数据总线	来自ALU结果
ALUSrc	表示ALU的第二个数据源来自指令[15:0]	来自读寄存器2
RegWrite	将写寄存器数据存入写寄存器地址中	无操作
MemWrite	将写数据总线上的数据写入内存地址单元	无操作
MemRead	将内存单元的内容输出到读数据总线上	无操作

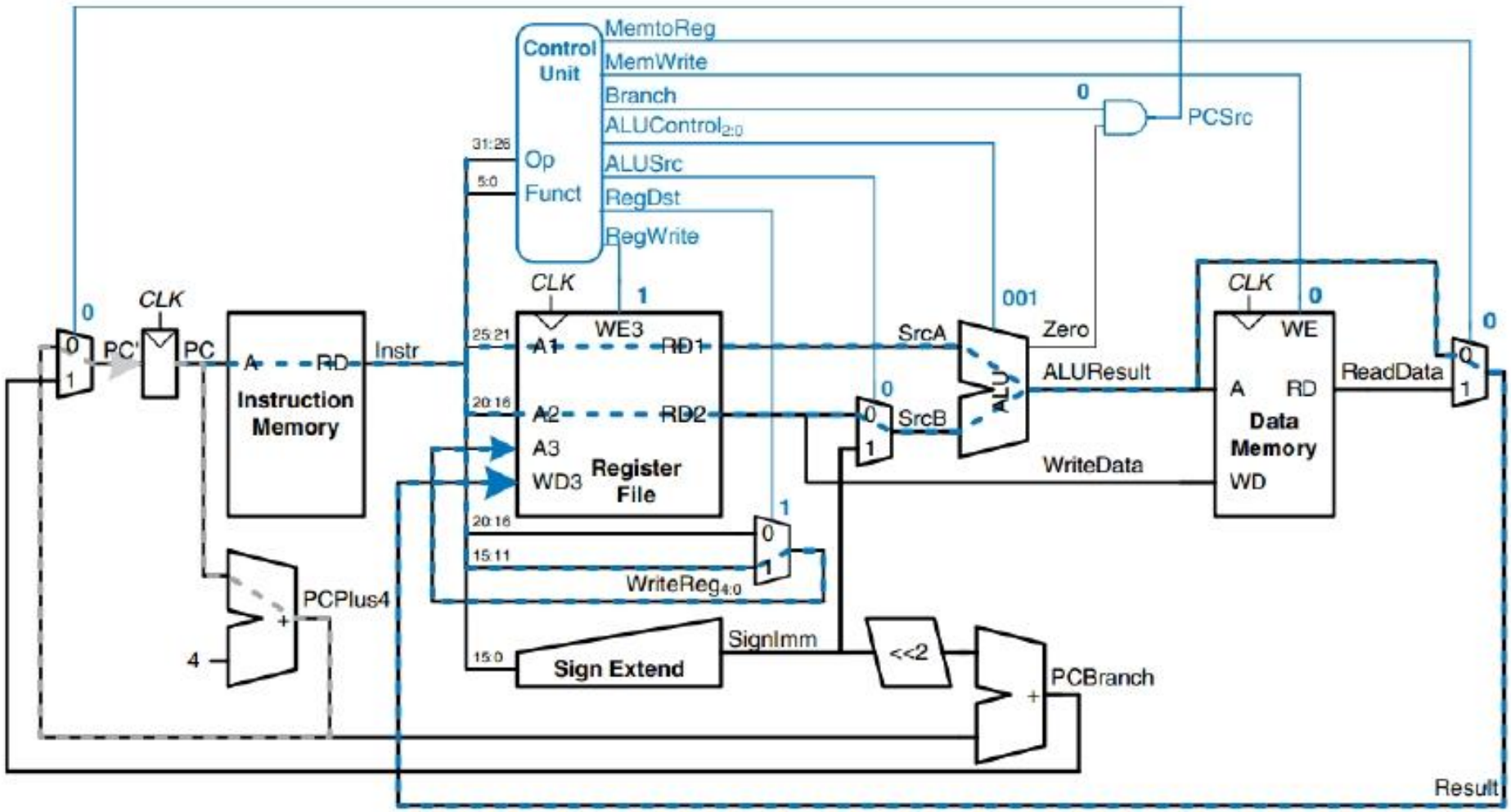
控制信号与指令之间的关系

指令	RegDst	Jump	Branch	MemToReg	ALUSrc	RegWrite	MemWrite	MemRead	ALUOp[1:0]
R型	1	0	0	0	0	1	0	0	1 0
lw	0	0	0	1	1	1	0	1	0 0
sw	X	0	0	X	1	0	1	0	0 0
beq	X	0	1	X	0	0	0	0	0 1
j	X	1	0	X	X	0	0	0	X x

指令操作码与控制信号逻辑关系真值表

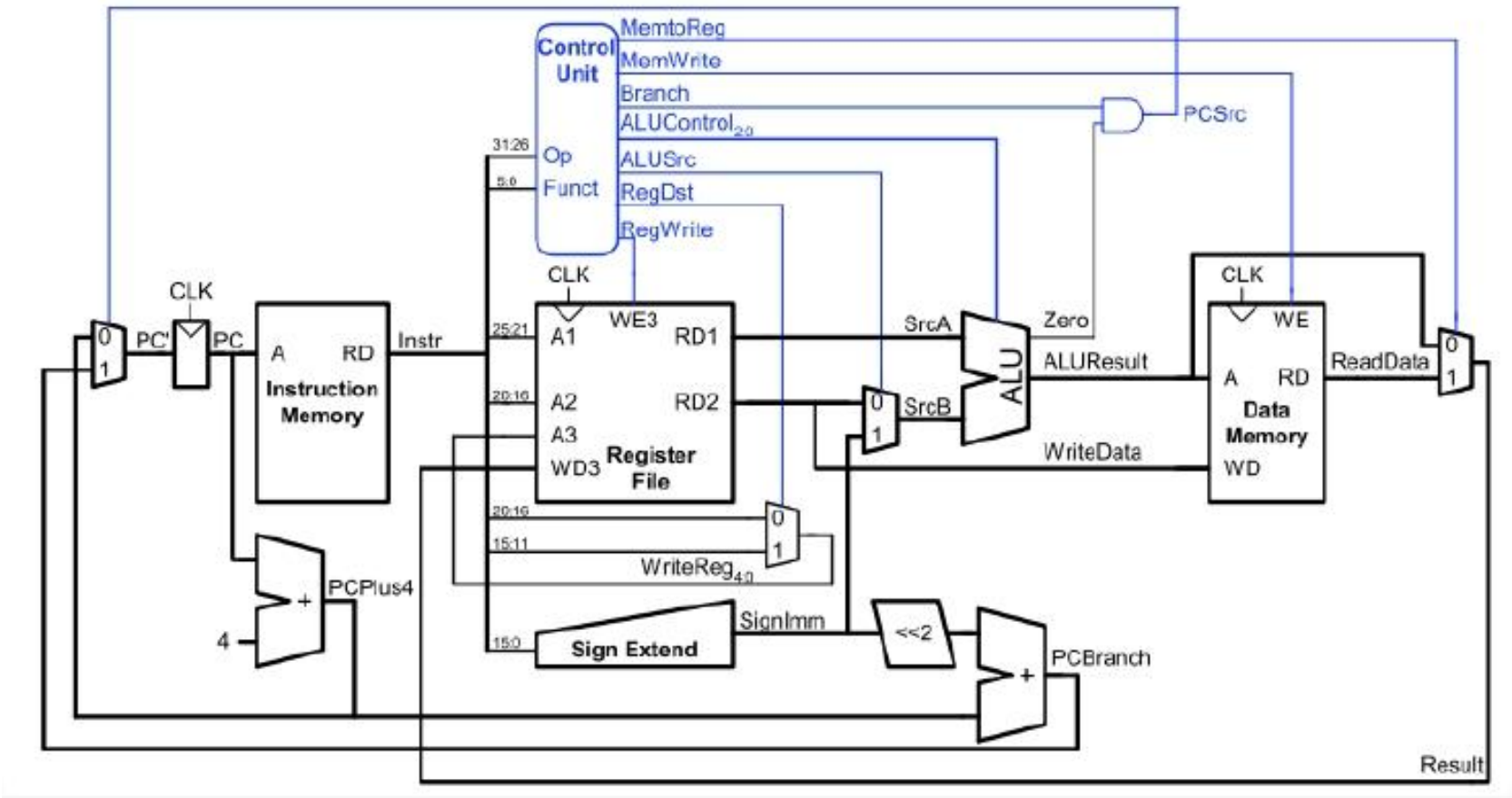
输入输出	信号名称	R型	lw	sw	beq	j
输入	op5	0	1	1	0	0
	op4	0	0	0	0	0
	op3	0	0	1	0	0
	op2	0	0	0	1	0
	op1	0	1	1	0	1
	op0	0	1	1	0	0
输出	ALUop1	1	0	0	0	X
	ALUop0	0	0	0	1	X
	RegDst	1	0	X	X	X
	ALUSrc	0	1	1	0	X
	RegWrite	1	1	0	0	0
	MemWrite	0	0	1	0	0
	MemRead	0	1	0	0	0
	Branch	0	0	0	1	0
	Jump	0	0	0	0	1
	MemToReg	0	1	X	X	X

Single-Cycle Datapath Example: or



Extended Functionality: addi

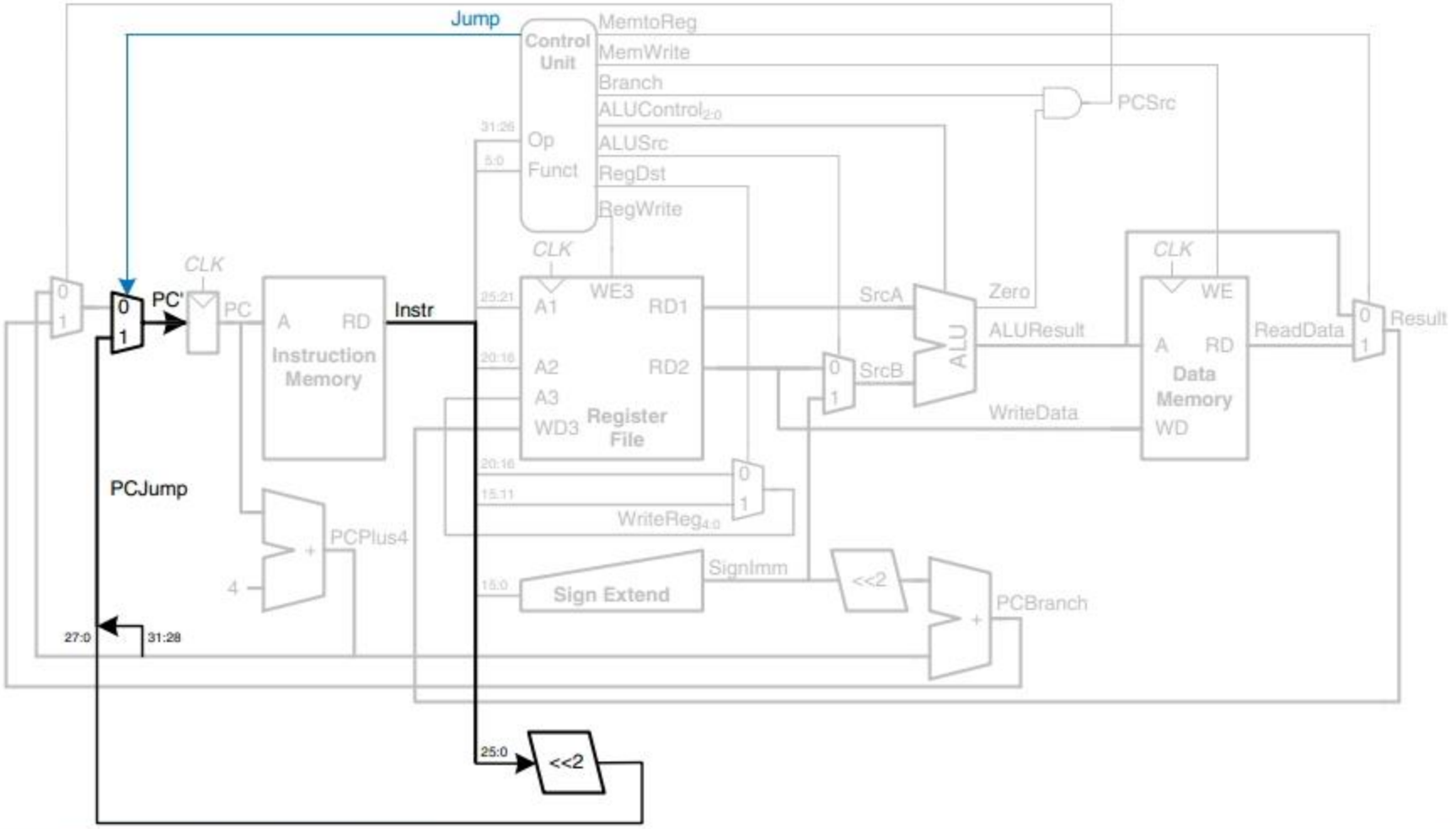
- No change to datapath



Extended Functionality: addi

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp
R-type	000000	1	1	0	0	0	0	10
lw	100011	1	0	1	0	0	1	00
sw	101011	0	X	1	0	1	X	00
beq	000100	0	X	0	1	0	X	01
addi	001000	1	0	1	0	0	0	00

Extended Functionality: j



Extended Functionality: j

Instruction	Opcode	RegWrite	RegDst	ALUSrc	Branch	MemWrite	MemtoReg	ALUOp	Jump
R-type	000000	1	1	0	0	0	0	10	0
lw	100011	1	0	1	0	0	1	00	0
sw	101011	0	X	1	0	1	X	00	0
beq	000100	0	X	0	1	0	X	01	0
addi	001000	1	0	1	0	0	0	00	0
j	000010	0	X	X	X	0	X	XX	1

Next . . .

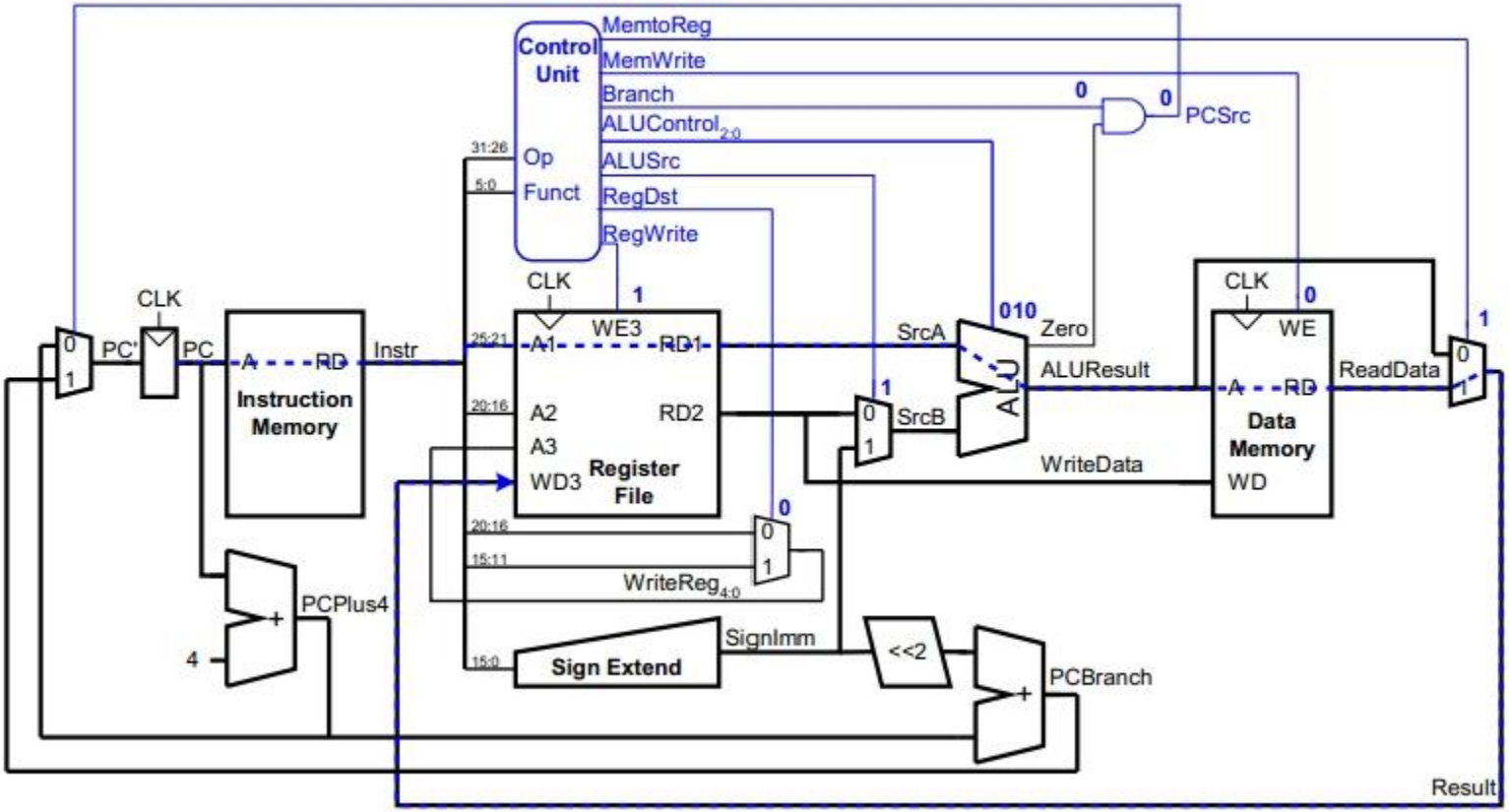
- Part-I:单周期处理器
  - Designing a Processor: Step-by-Step
  - Datapath Components and Clocking
  - 数据通路（Data Path）
  - 处理器性能与关键路径（Critical Path）
- Part-II :实验
  - Lab2 MIPS处理器实现 (实验教程，第6章)

Processor Performance

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

$\frac{\text{Seconds}}{\text{Program}}$	How long you have to wait
$\frac{\text{Instructions}}{\text{Program}}$	Number that must execute to complete the task
$\frac{\text{Clock Cycles}}{\text{Instruction}}$	CPI: Cycles per instruction
$\frac{\text{Seconds}}{\text{Clock Cycle}}$	The clock period (1/frequency)

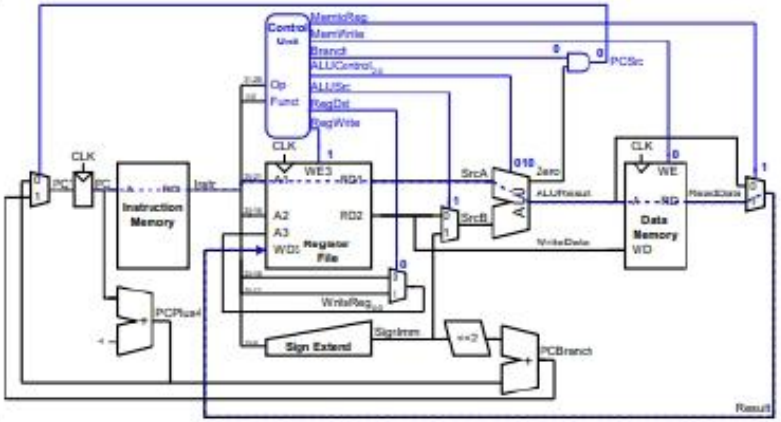
The Critical Path Here: Load from Memory



Instruction Memory to Register File to ALU to Data Memory to Register File

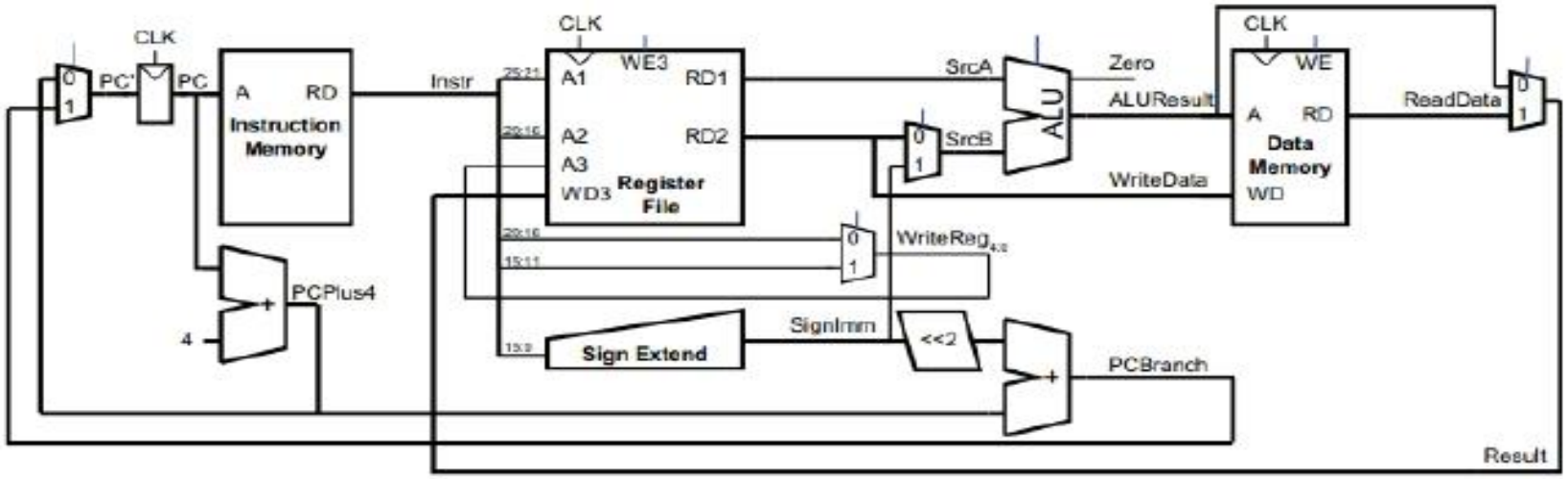
The Critical Path Dictates the Clock Period

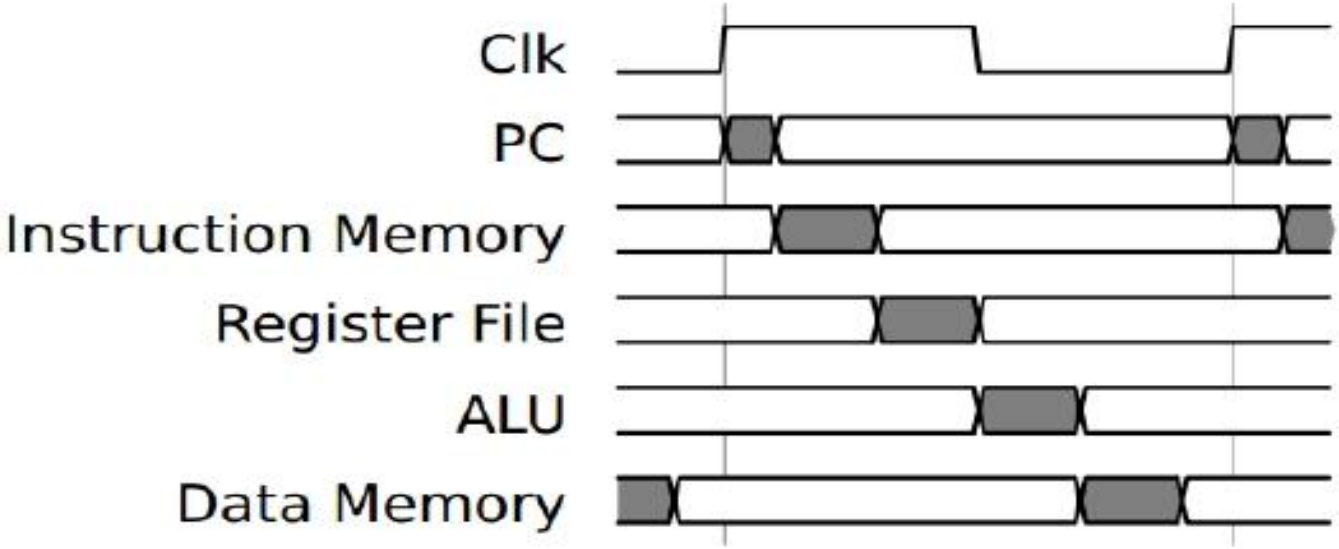
Element		Delay
Register clk-to-Q	$t_{pcq\text{-}PC}$	30 ps
Register setup	t_{setup}	20
Multiplexer	t_{mux}	25
ALU	t_{ALU}	200
Memory Read	t_{mem}	250
Register file read	t_{RFread}	150
Register file setup	$t_{RFsetup}$	20



$$
\begin{aligned}
T_C &= t_{pcq\text{-}PC} + t_{mem\text{-}I} + t_{RFread} + t_{ALU} + t_{mem\text{-}D} + t_{mux} + t_{RFsetup} \\
&= (30 + 250 + 150 + 200 + 250 + 25 + 20)\ \text{ps} \\
&= 925\ \text{ps} \\
&= 1.08\ \text{GHz}
\end{aligned}
$$

Single-Cycle Datapath Timing





Execution Time for the Single-Cycle Processor

- For a 100 billion-instruction task on our single-cycle processor with a 925 ps clock period,

$$\frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock Cycle}}$$

$$= 100 \times 10^9 \times 1 \times 925 \text{ ps}$$

$$= 92.5 \text{ seconds}$$

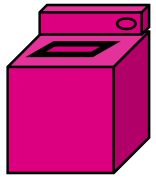
Q&A

Pipelined Processor Design

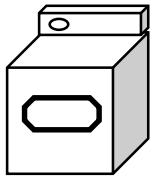
Pipelining Example

- Laundry Example: Three Stages

1. Wash dirty load of clothes

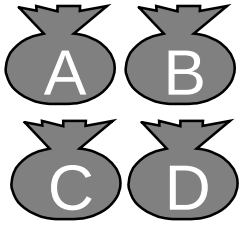


2. Dry wet clothes



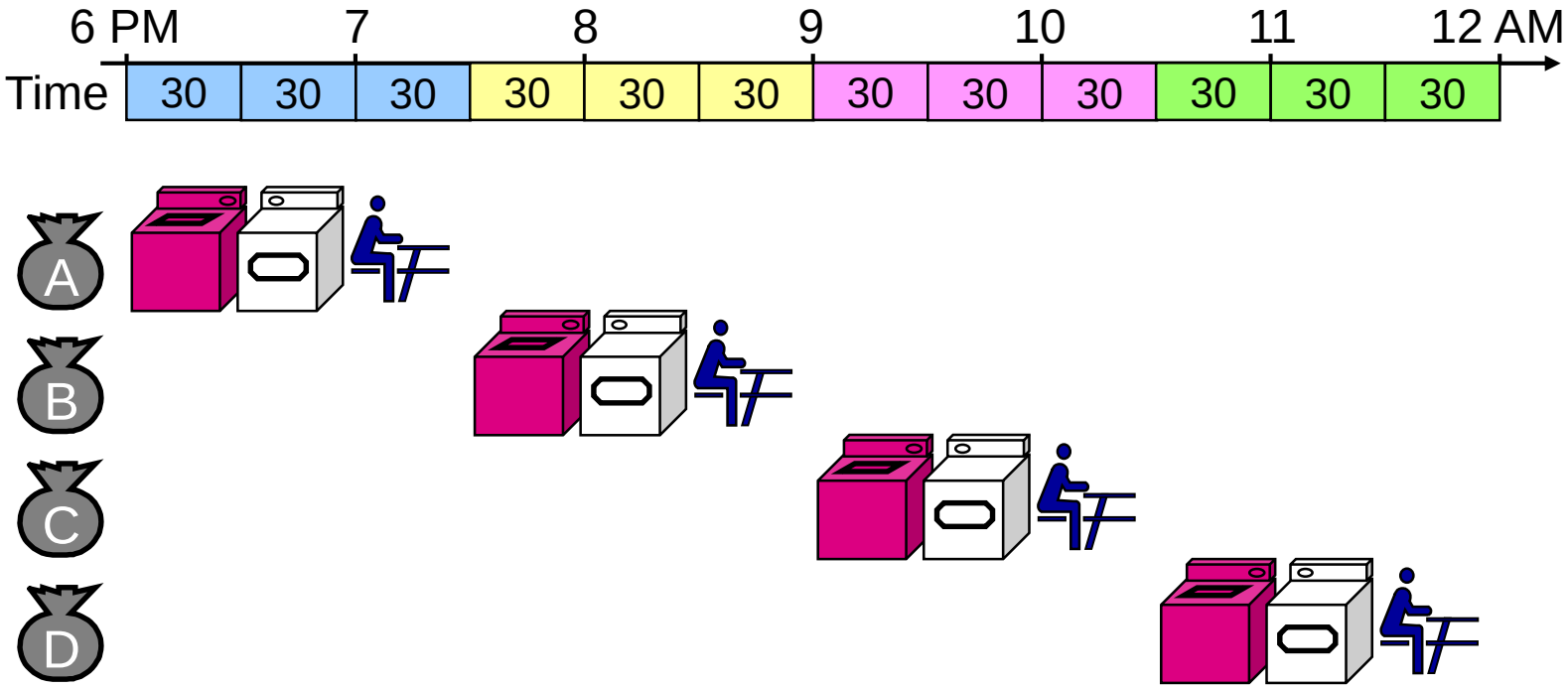
3. Fold and put clothes into drawers

- Each stage takes 30 minutes to complete



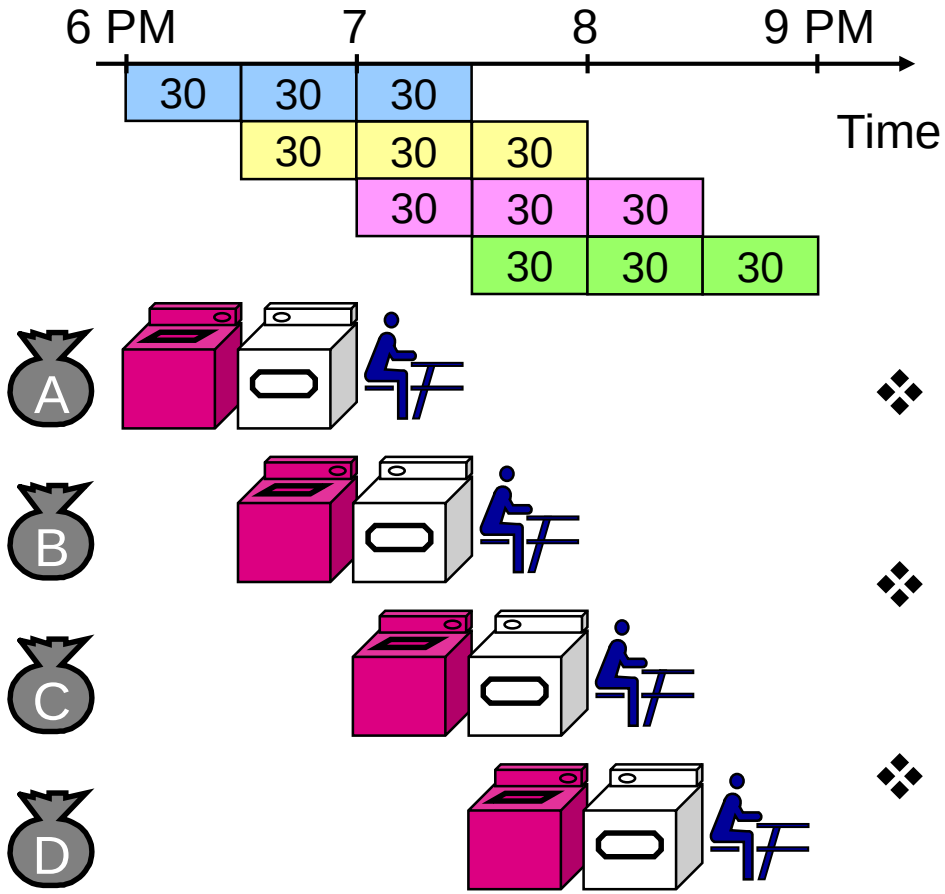
- Four loads of clothes to wash, dry, and fold

Sequential Laundry (顺序洗衣)



- Sequential laundry takes 6 hours for 4 loads
- Intuitively, we can use pipelining to speed up laundry

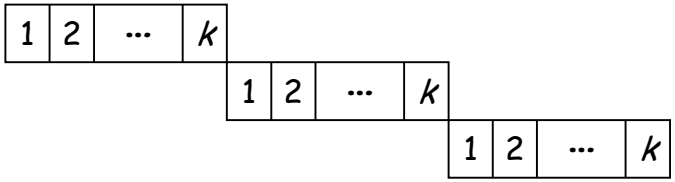
Pipelined Laundry: Start Load ASAP



- Pipelined laundry takes 3 hours for 4 loads
- Speedup factor is 2 for 4 loads
- Time to wash, dry, and fold one load is still the same (90 minutes)

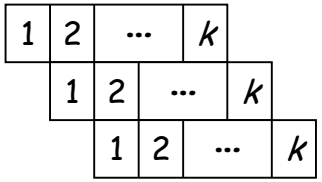
Serial Execution versus Pipelining

- Consider a task that can be divided into *k* subtasks
  - The *k* subtasks are executed on *k* different stages
  - Each subtask requires one time unit
  - The total execution time of the task is *k* time units
- Pipelining is to overlap the execution
  - The *k* stages work in parallel on *k* different tasks
  - Tasks enter/leave pipeline at the rate of one task per time unit



Without Pipelining

One completion every *k* time units

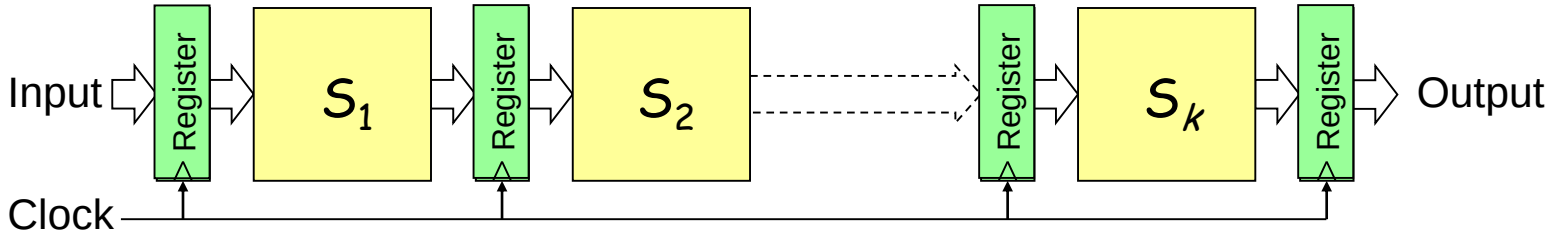


With Pipelining

One completion every 1 time unit

Synchronous Pipeline

- Uses clocked registers between stages
- Upon arrival of a clock edge …
  - All registers hold the results of previous stages simultaneously
- The pipeline stages are combinational logic circuits
- It is desirable to have balanced stages
  - Approximately equal delay in all stages
- Clock period is determined by the maximum stage delay

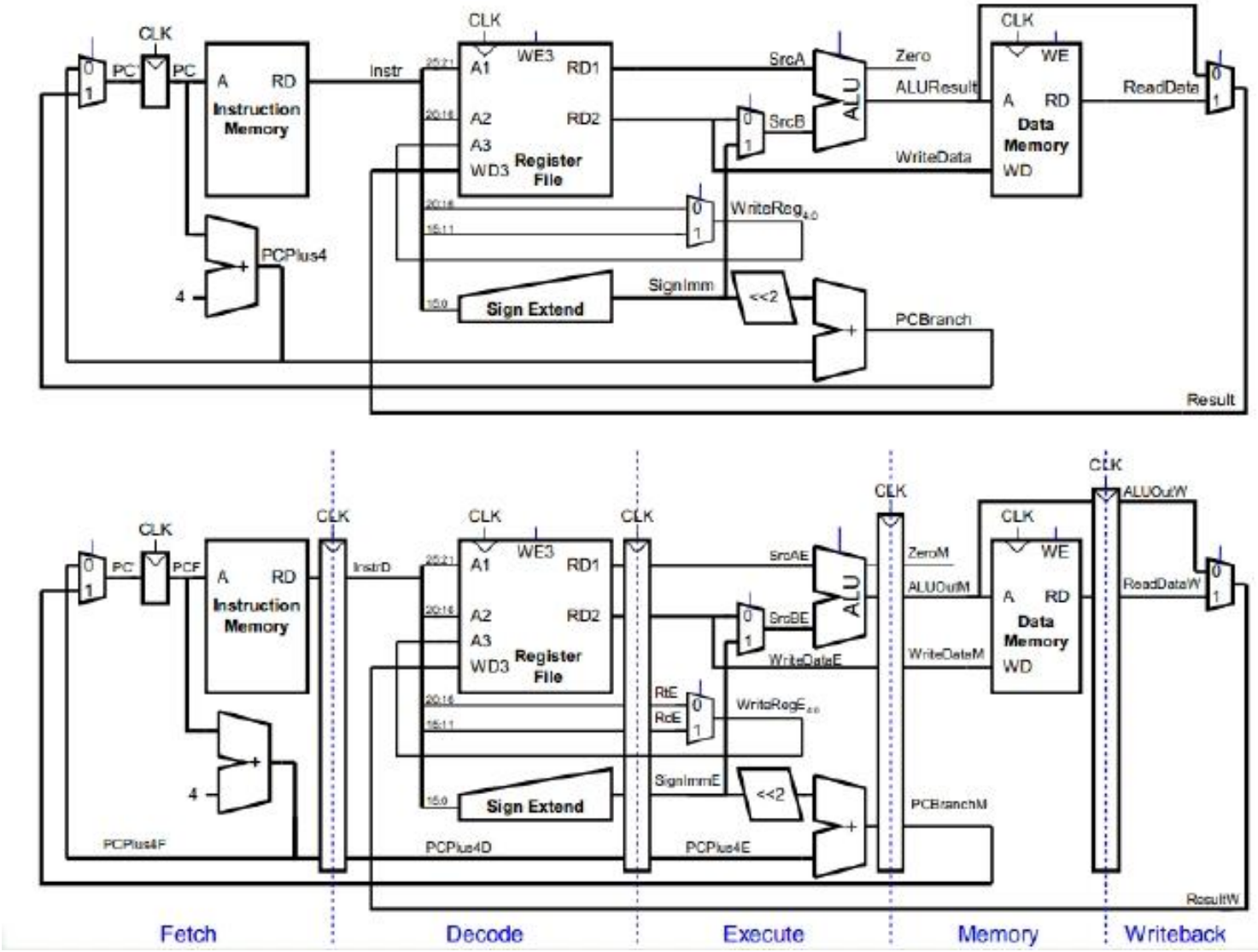


MIPS Processor Pipeline

- Five stages, one cycle per stage

1. IF: Instruction Fetch from instruction memory
2. ID: Instruction Decode, register read, and J/Br address
3. EX: Execute operation or calculate load/store address
4. MEM: Memory access for load and store
5. WB: Write Back result to register

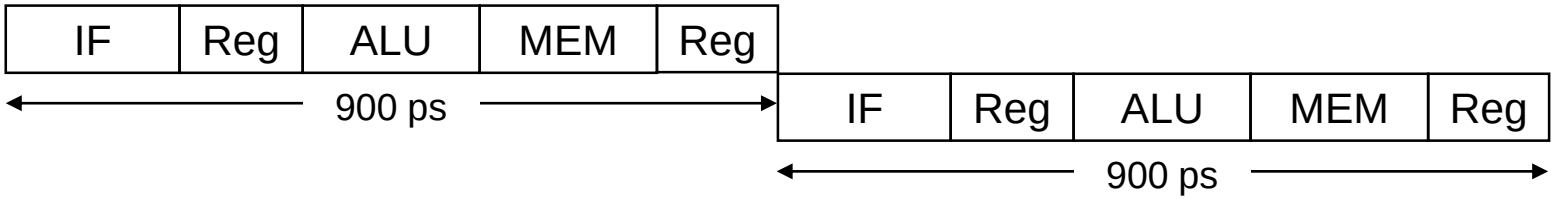
Single-Cycle vs. Pipelined Datapath



Single-Cycle vs Pipelined Performance

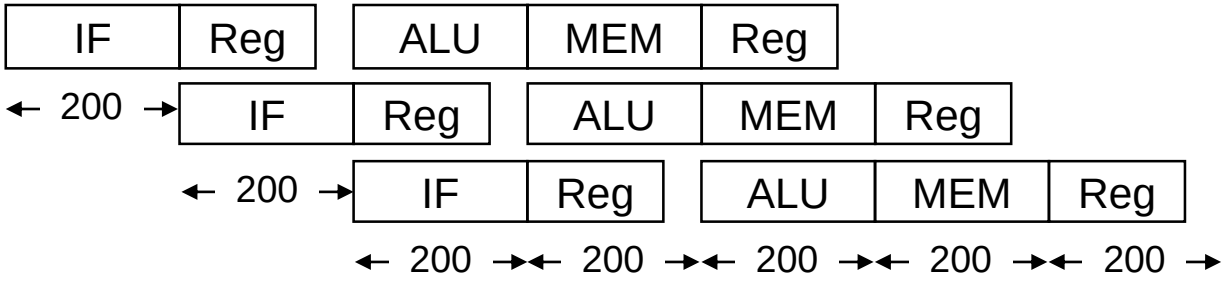
- Consider a 5-stage instruction execution in which …
  - Instruction fetch = ALU operation = Data memory access = 200 ps
  - Register read = register write = 150 ps
- What is the clock cycle of the single-cycle processor?
- What is the clock cycle of the pipelined processor?
- What is the speedup factor of pipelined execution?
- Solution

  Single-Cycle Clock = 200+150+200+200+150 = 900 ps



Single-Cycle versus Pipelined – cont'd

- Pipelined clock cycle = max(200, 150) = 200 ps



- CPI for pipelined execution = 1
  - One instruction completes each cycle (ignoring pipeline fill)
- Speedup of pipelined execution = 900 ps / 200 ps = 4.5
  - Instruction count and CPI are equal in both cases
- Speedup factor is less than 5 (number of pipeline stage)
  - Because the pipeline stages are not balanced

Pipeline Performance Summary

- Pipelining doesn't improve latency of a single instruction
- However, it improves throughput of entire workload
  - Instructions are initiated and completed at a higher rate
- In a *k*-stage pipeline, *k* instructions operate in parallel
  - Overlapped execution using multiple hardware resources
  - Potential speedup = number of pipeline stages *k*
  - Unbalanced lengths of pipeline stages reduces speedup
- Pipeline rate is limited by slowest pipeline stage
- Unbalanced lengths of pipeline stages reduces speedup
- Also, time to fill and drain pipeline reduces speedup

Pipeline Hazards

- Hazards: situations that would cause incorrect execution
  - If next instruction were launched during its designated clock cycle

1. Structural hazards
   - Caused by resource contention
   - Using same resource by two instructions during the same cycle
2. Data hazards
   - An instruction may compute a result needed by next instruction
   - Hardware can detect dependencies(依赖关系) between instructions
3. Control hazards
   - Caused by instructions that change control flow (branches/jumps)
   - Delays in changing the flow of control

- Hazards complicate pipeline control and limit performance

(冒险使得流水线控制复杂化，并限制了性能)

超标量技术

微处理器集成多个ALU、多个译码器和多条流水线，以并行处理的方式来提高性能

假设处理器有一个整数部件和一个浮点部件，处理器至多能发出两条指令，一条是整数类型的指令，包括整数算术逻辑运算，存储器访问操作和转移指令；另一条必须是浮点类型的指令

异常处理机制

处理异常事件的机制

异常种类	来源	MIPS处理器命名
I/O设备	外部	中断
用户程序唤醒操作系统	内部	异常
计算结果溢出	内部	异常
未定义的指令（非法指令）	内部	异常
硬件出错	两者	异常或中断

异常处理系统需具备的功能

微处理器要能够实现异常处理需要完成以下几方面的功能：

记录异常发生的原因

记录程序断点处的指令在存储器中的地址

记录不同种类的异常处理程序在内存中的地址

建立异常种类与异常处理程序地址之间的对应关系。

异常事件识别机制

状态位法（MIPS)：

在微处理器中利用一个寄存器对每种异常事件确定一个标志位，当有异常事件发生时，寄存器中对应的位置1，一个32位的寄存器可以表示32种不同类型的异常事件

向量法(Intel）：

对不同类型的异常事件进行编码，这个编码叫中断类型码或异常类型码。

断点保存和返回

寄存器法（嵌入式）

在微处理器中设计一个寄存器EPC，当微处理器出现异常时，就将PC的值保存到EPC中。异常处理完之后，再把EPC的值赋给PC，这样就可以实现中断的返回

栈（PC）

微处理器直接将PC的值压入栈中，异常处理完之后，再从栈顶把值弹出来赋给PC

异常处理程序进入方式

1）专门的内存区域保存异常处理程序

在这块内存区域中为每个异常处理程序分配固定长度的空间如32个字节或8条指令长度的空间，而且针对每个异常事件其异常处理程序的存放地址是固定的。

2）仅提供一个异常处理程序存放地址

发生任何异常事件都首先转移到该地址执行总的异常处理，并在总异常处理程序中分析异常事件的原因，然后再根据异常的原因通过子程序调用的方式去执行相应的异常处理。

异常处理程序进入方式

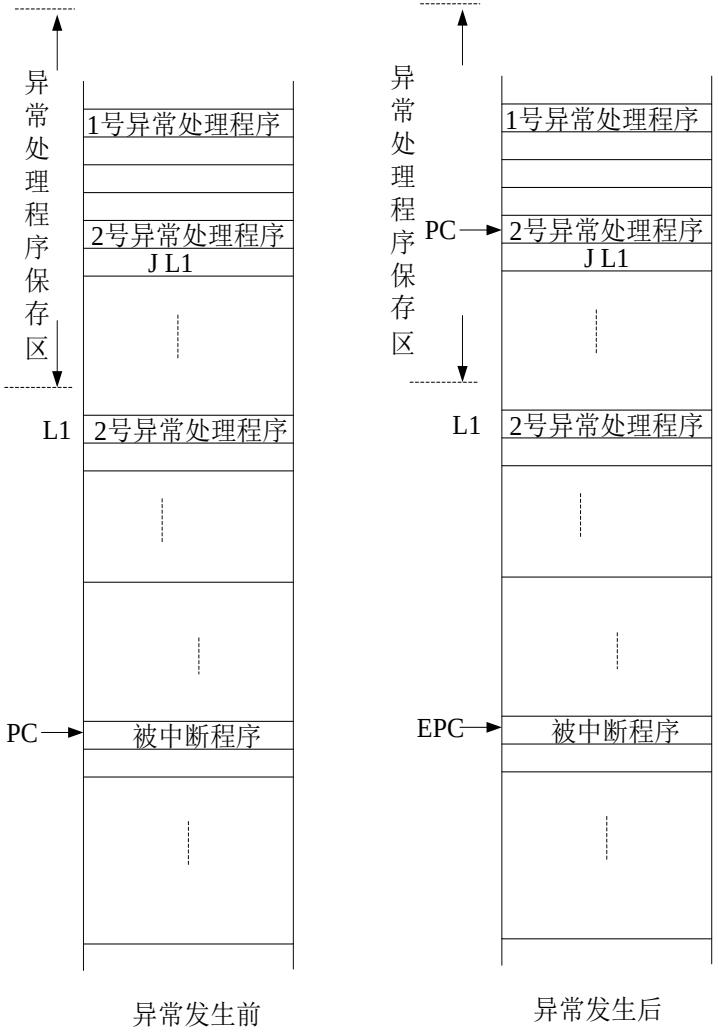
3）分配一块专门的内存区域保存异常处理程序的入口地址

异常处理程序的入口地址叫中断向量

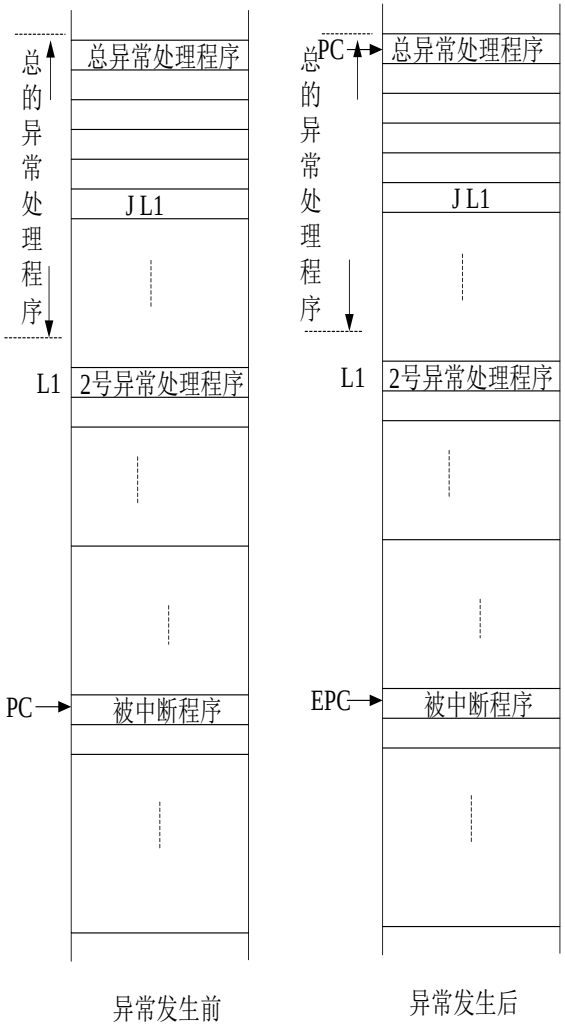
保存异常处理程序的入口地址的内存区域叫做中断向量表

异常处理程序可以存放在内存中的任意位置，只需要把该异常处理程序的入口地址保存到中断向量表中正确的地址中，当异常发生时，微处理器就可以通过中断向量表查找到中断服务程序的入口地址。

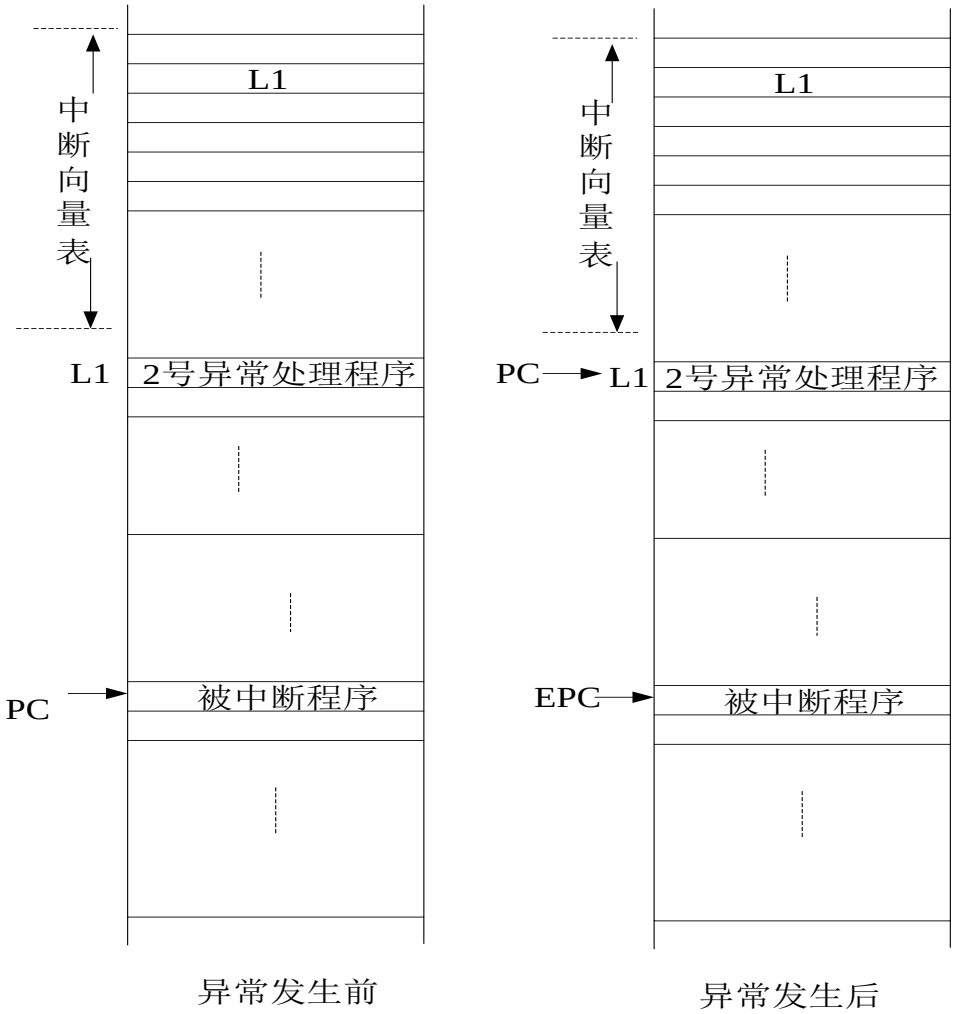
异常处理程序进入方式1）



异常处理程序进入方式2）

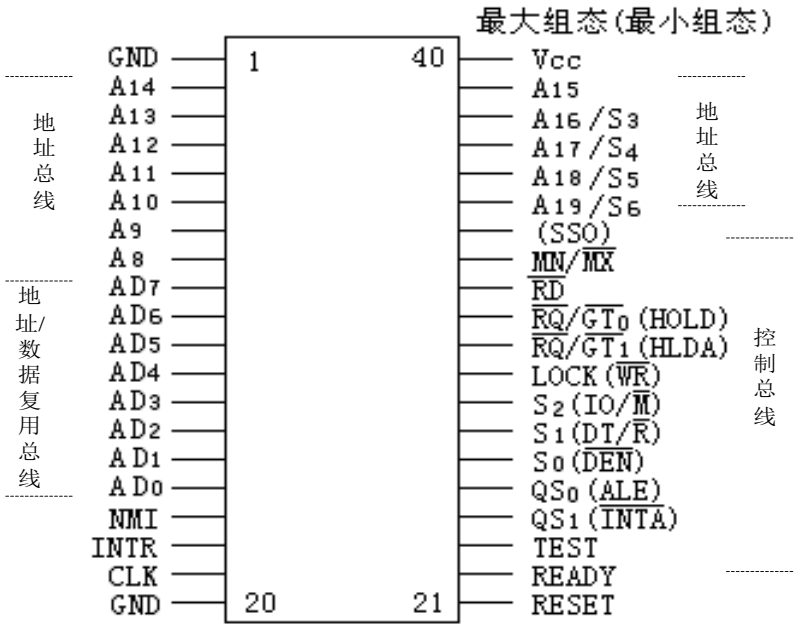


异常处理程序进入方式3）



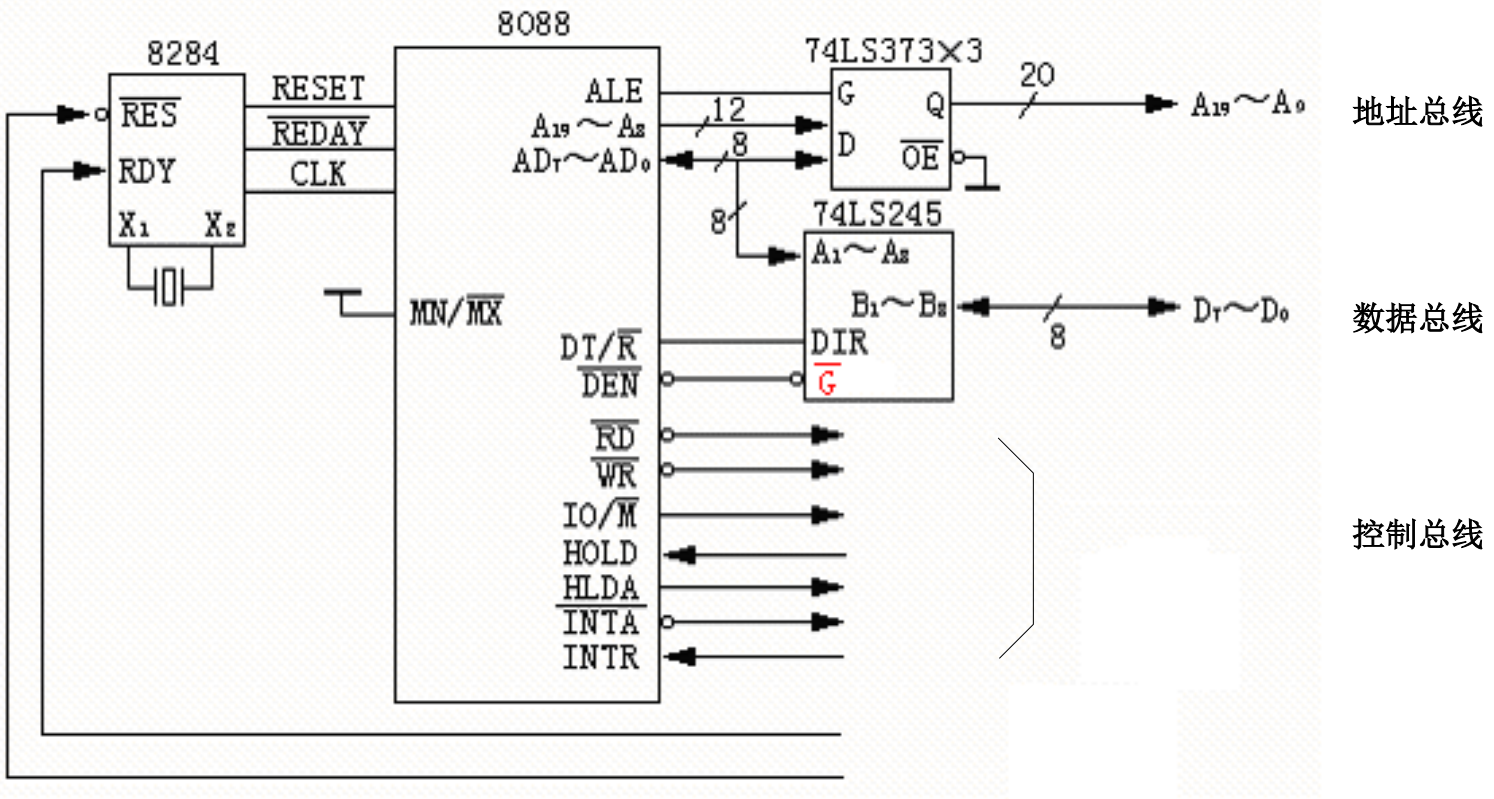
微处理器外部接口

微处理器为实现与计算机系统内的其他部件之间的信息交互必须提供地址、数据和控制总线

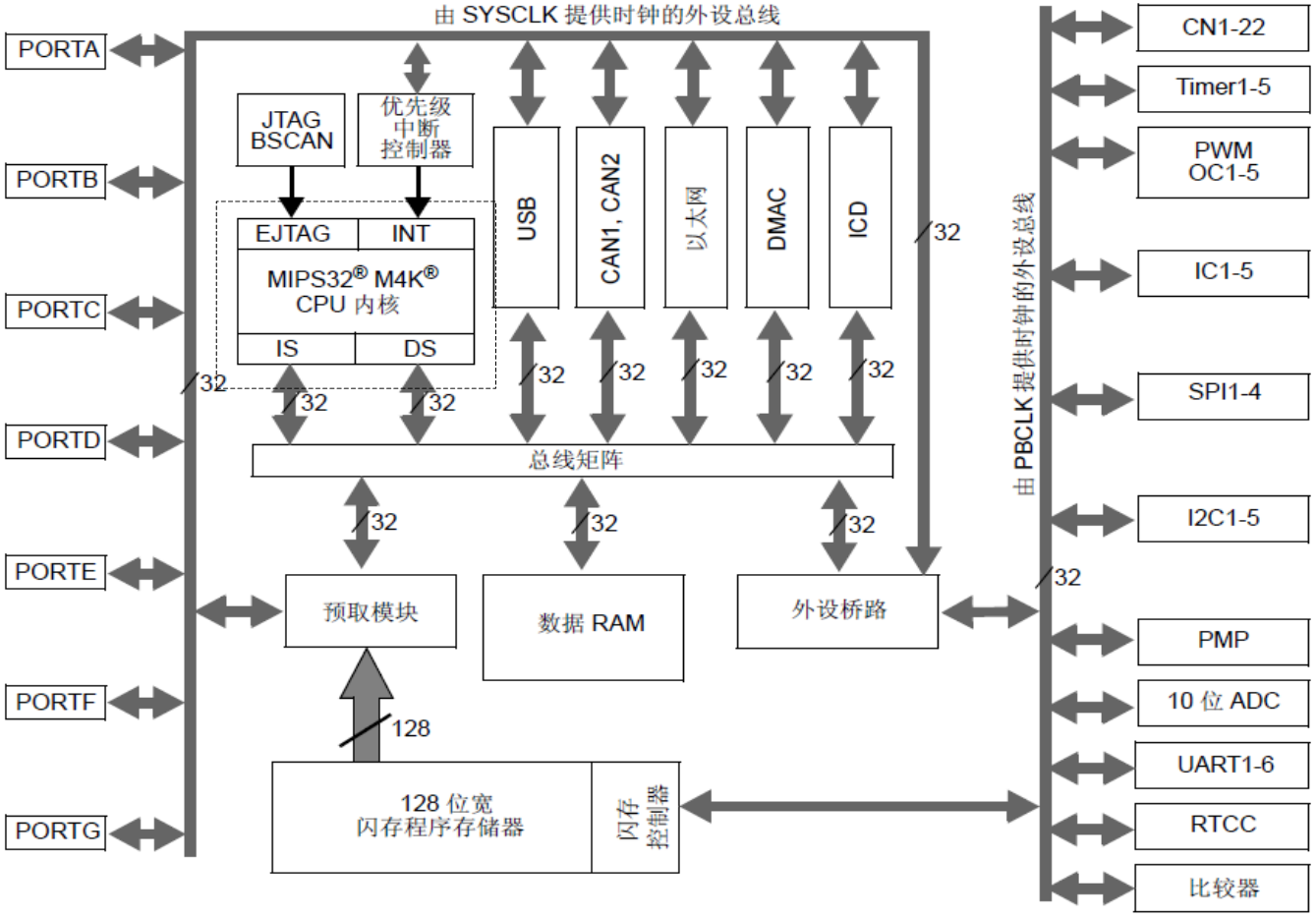


8088微处理器外部接口

8088微处理器最小组态系统总线接口电路



嵌入式芯片PIC32MX5XX/6XX/7XX系列框图



MicroBlaze微处理器简介

可以配置为支持大字节序或小字节序，

当采用PLB外部总线时为大字节序，

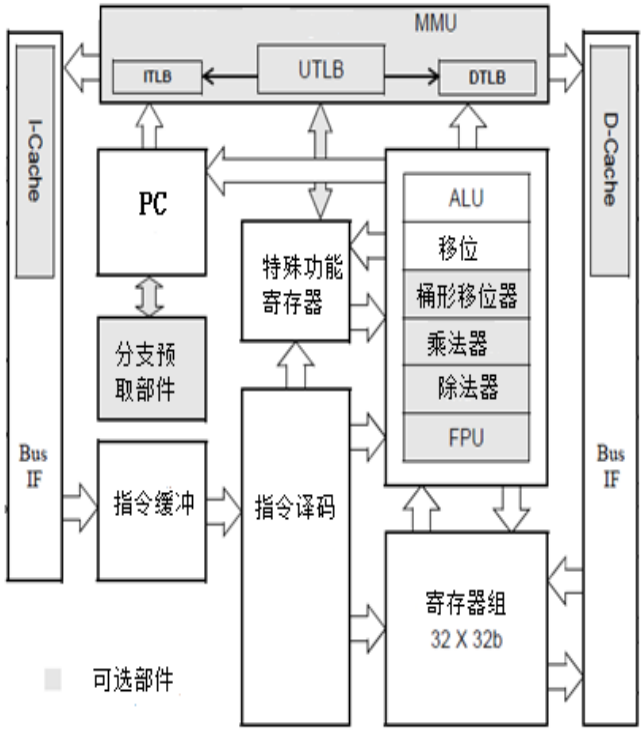
采用AXI4外部总线时为小字节序。

支持三级或五级流水线，

具有可选的指令和数据cache，

支持虚拟内存管理。

支持通过PLB、LMB、AXI总线与外围接口或部件相连。

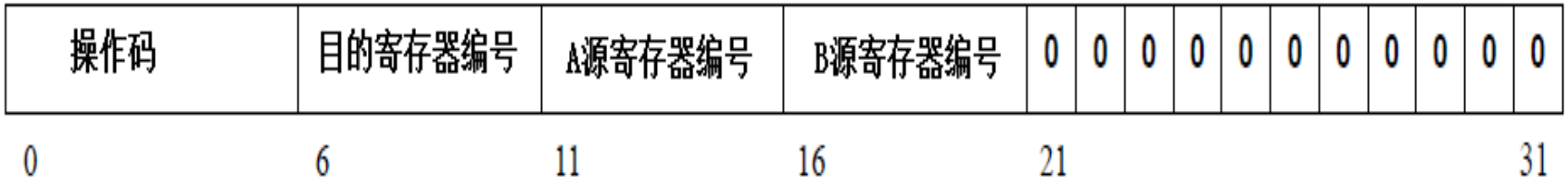


具有32个32位的通用寄存器，使用规则与MIPS微处理器的通用寄存器使用规则相同，命名为R0~R31。

R14，R15，R16，R17又用做异常返回地址寄存器，

还具有18个32位的特殊功能寄存器，包括PC，MSR等

MicroBlaze指令架构



作业

7