Chapter4-1

The Processor: Datapath and Control (Single-cycle implementation)

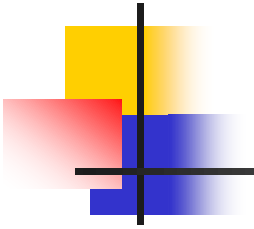
臺大電機系

吳安宇教授

outline

- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a Datapath
- 4.4 A Simple Implementation Scheme

Introduction



- Show key issues in creating datapaths and designing controls.

- Design and implement the MIPS instructions including:
  (1) memory-reference instructions: lw, sw
  (2) arithmetic-logical instructions: add, sub, and, or, slt
  (3) branch instructions: beq, j

- Guideline in hardware implementation:
  (1) *Make the common case fast*
  (2) *Simplicity favors regularity*

Overview of the implementation (1/3)

- For every instruction, the first two steps are the same:
  - **Fetch:** Send the Program Counter (PC) to the memory that contains the code (Instruction Fetch)
  - **Read registers:** Use fields of the instructions to select the registers to read.
    - Load/Store : Read one register
    - Others : Read two registers (R-type)
    - ***lw $s1, 200($s2)***
    - ***add $t0, $s1, $s2***

Overview of the implementation (2/3)

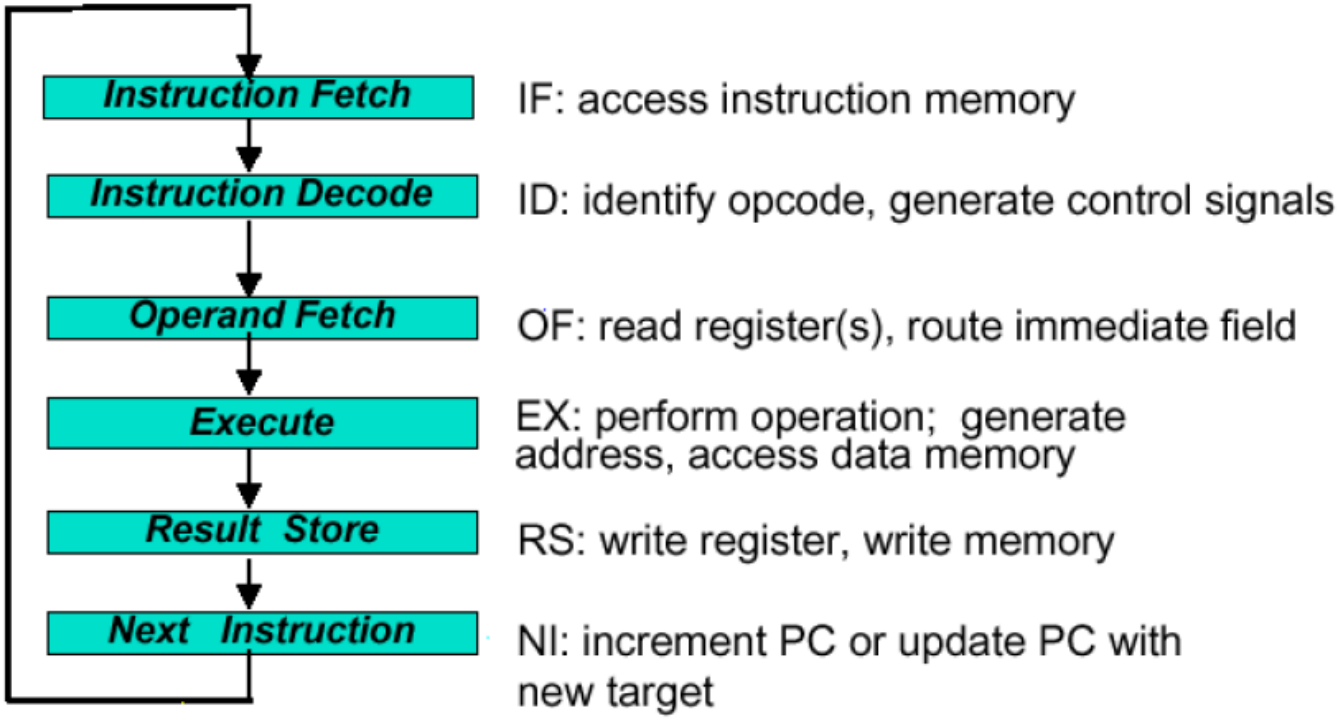
- Common actions for three instruction types:
  (all instructions use ALU after reading registers)

  (1) Memory-reference instructions:
  use ALU to calculate "effective address"
  e.g., lw $t0 offset($s5) → compute ***offset + $s5***
  (2) Arithmetic-logical instructions:
  use ALU for opcode execution → ***add, sub, or, and***
  (3) Branch instructions:
  use ALU for comparison – bne/slt $s1, $s2
  → ***$s1-$s2, and check sign bit of the results***

Overview of the implementation (2/3)

- After using ALU:
  1) *Memory-reference instructions:* need to access the memory containing the data to complete a "load" operation, or "store" a word to that memory location.
  2) *Arithmetic-logical instructions:* write the result of the ALU back into a destination register.
  3) *Branch instructions:* need to change the next instruction address based on the comparison (i.e., change the value of PC)

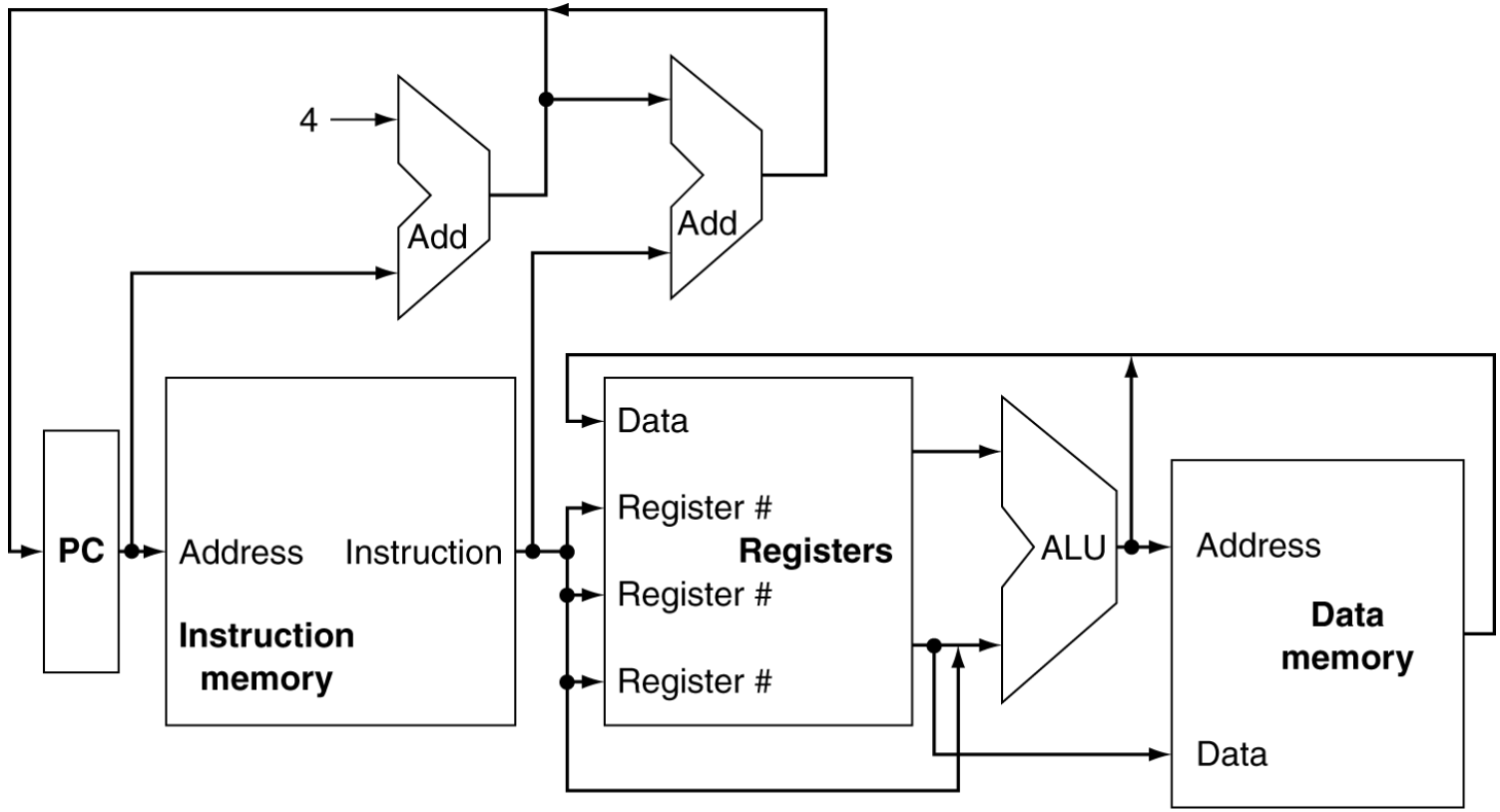
Typical Instruction Execution



Note that each step does not necessarily correspond to a clock cycle. These only describe the basic flow of instruction execution. The details vary with instruction type.

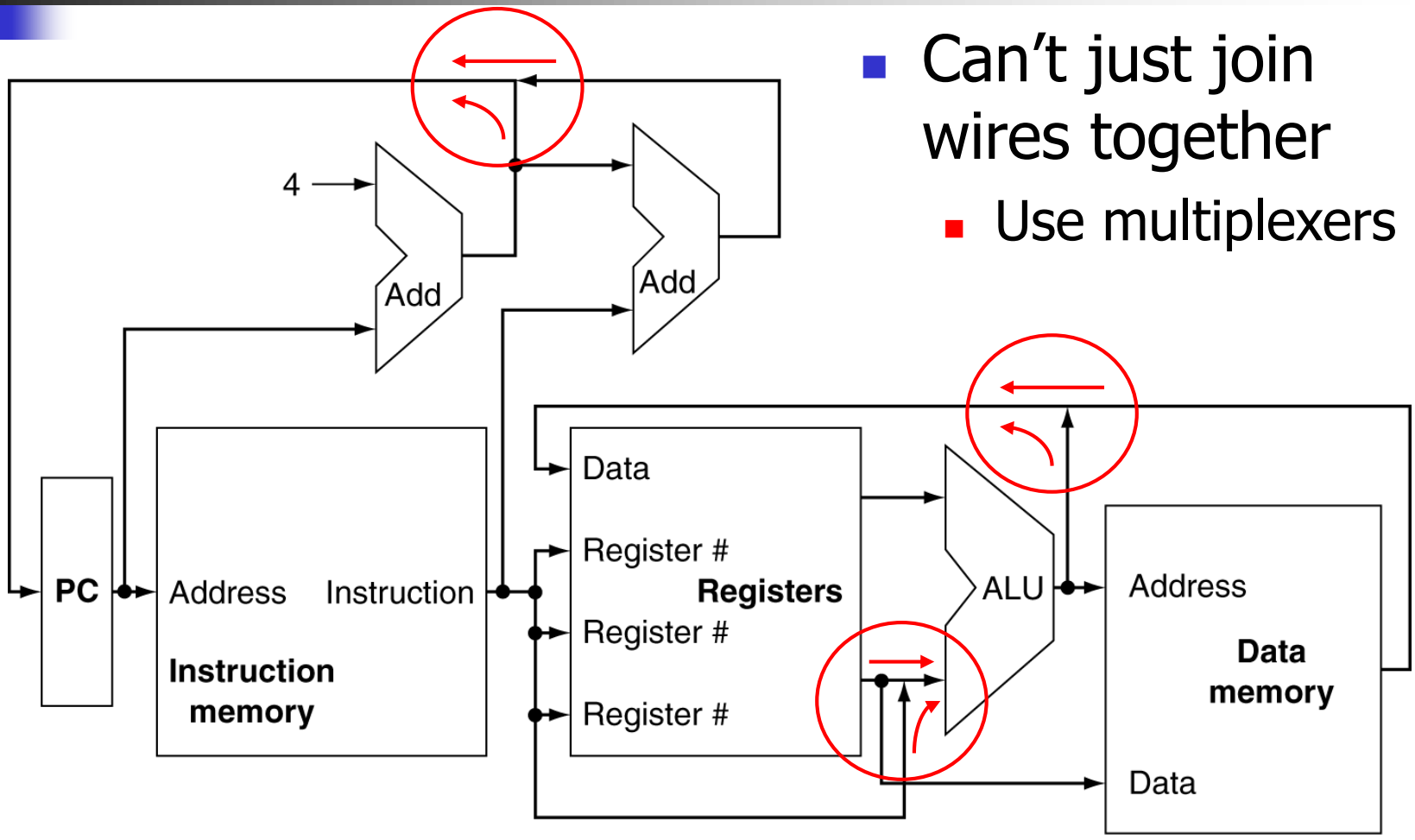
Abstract View of MIPS CPU Implementation (1/3)

- An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.

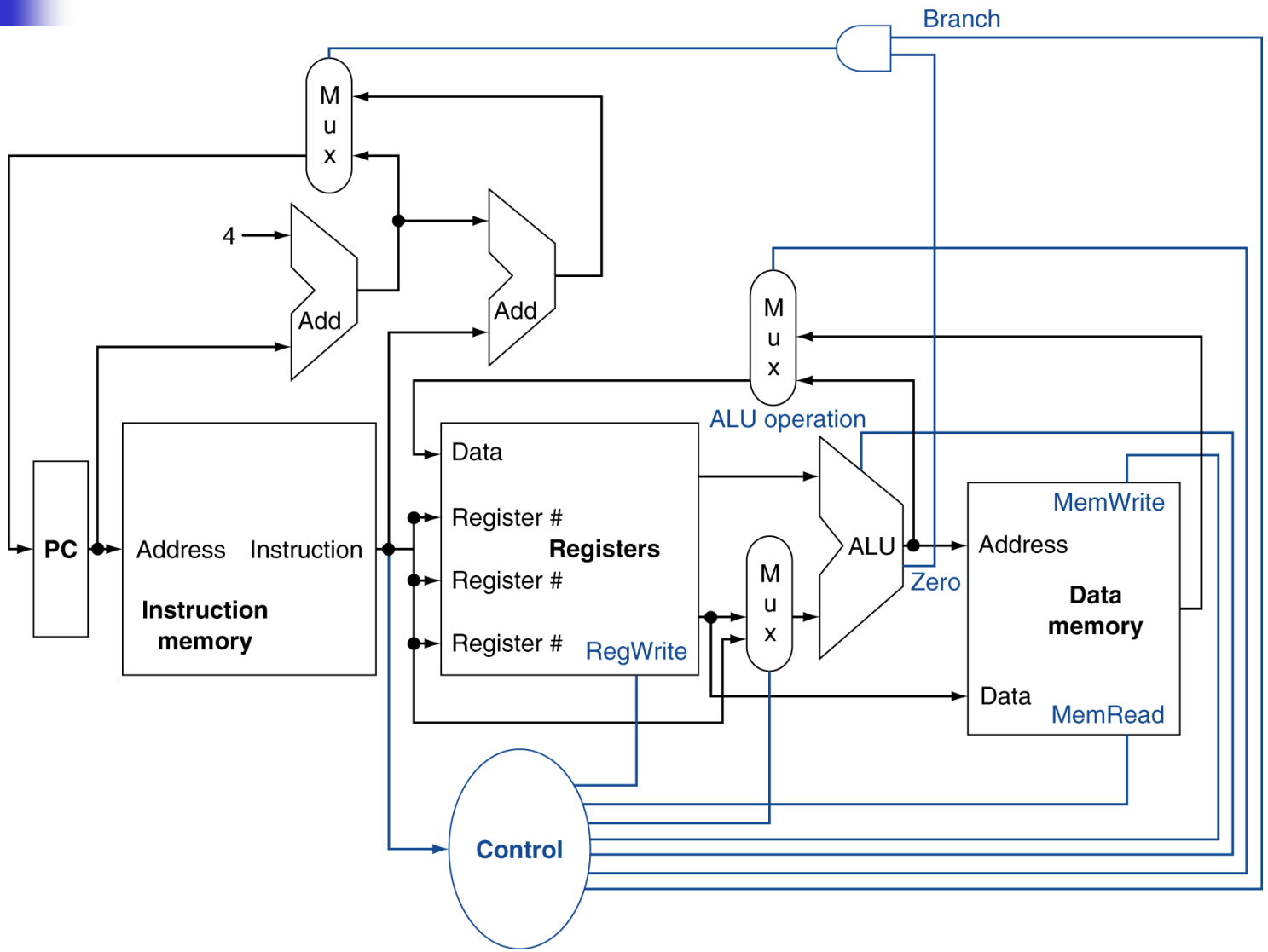


Multiplexers (2/3)

- Can't just join wires together
  - Use multiplexers



Control Signals (3/3)

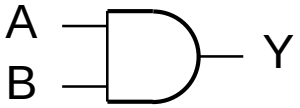


Outline

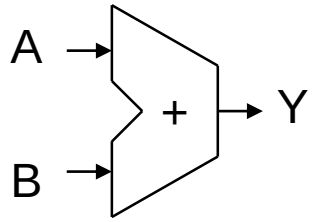
- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a Datapath
- 4.4 A Simple Implementation Scheme

Combinational Elements

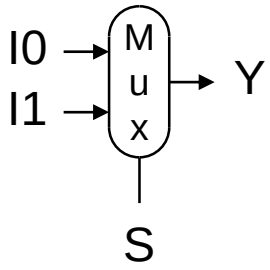
- AND-gate
  - Y = A & B



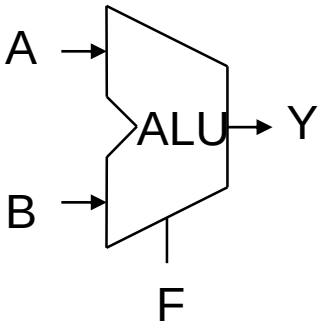
- Adder
  - Y = A + B



- Multiplexer
  - Y = S ? I1 : I0

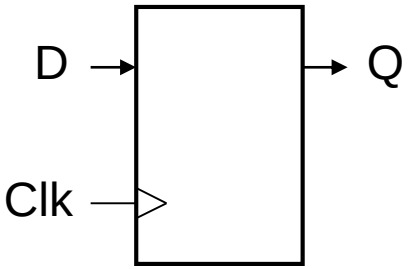


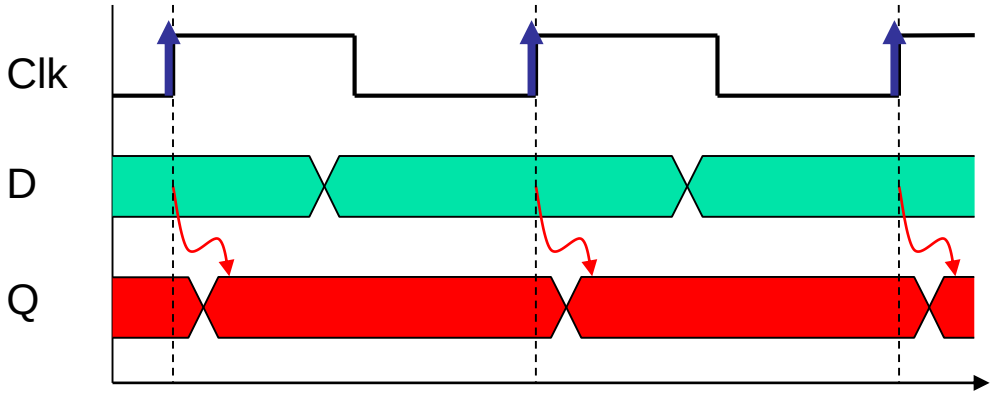
- Arithmetic/Logic Unit
  - Y = F(A, B)



Sequential Elements

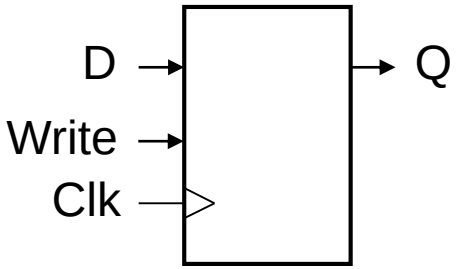
- Register: stores data in a circuit
  - Uses a clock signal to determine when to update the stored value
  - Edge-triggered: update when ***Clk*** changes from 0 to 1

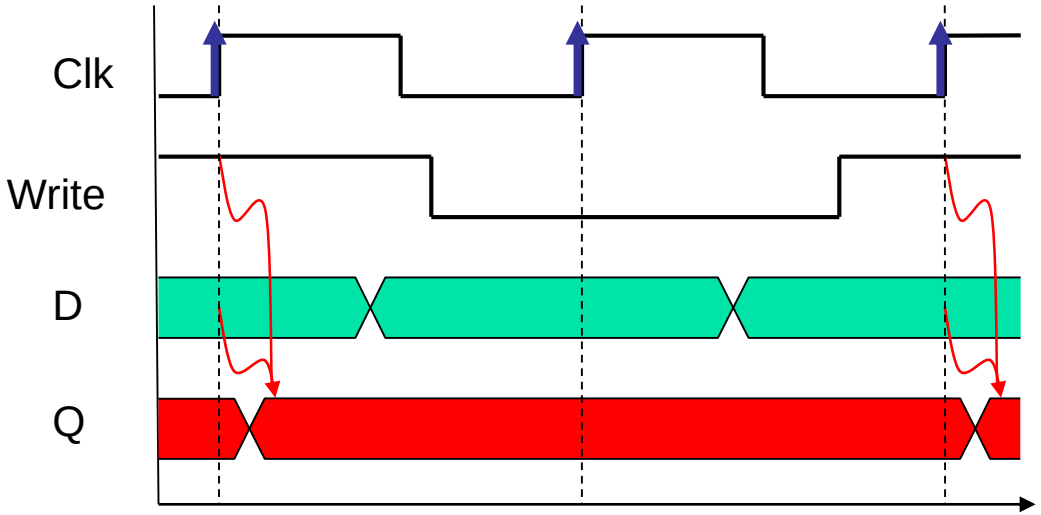




Sequential Elements

- Register with write control
  - Only updates on clock edge when write control input is 1
  - Used when stored value is required later

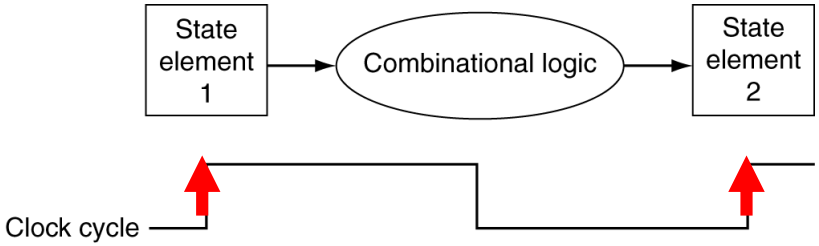


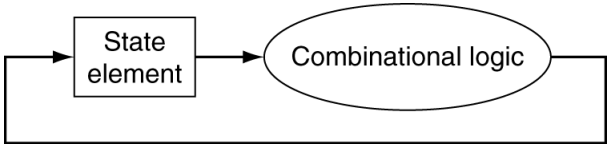


Clocking Methodology

- Combinational logic transforms data during clock cycles
  - An **Edge-triggered** methodology
  - Between clock edges
  - Input from state elements, output to state element
  - Longest delay determines clock period

  - Typical execution:
    - Read contents of some state elements,
    - Send values through some combinational logic
    - Write results to one or more state elements



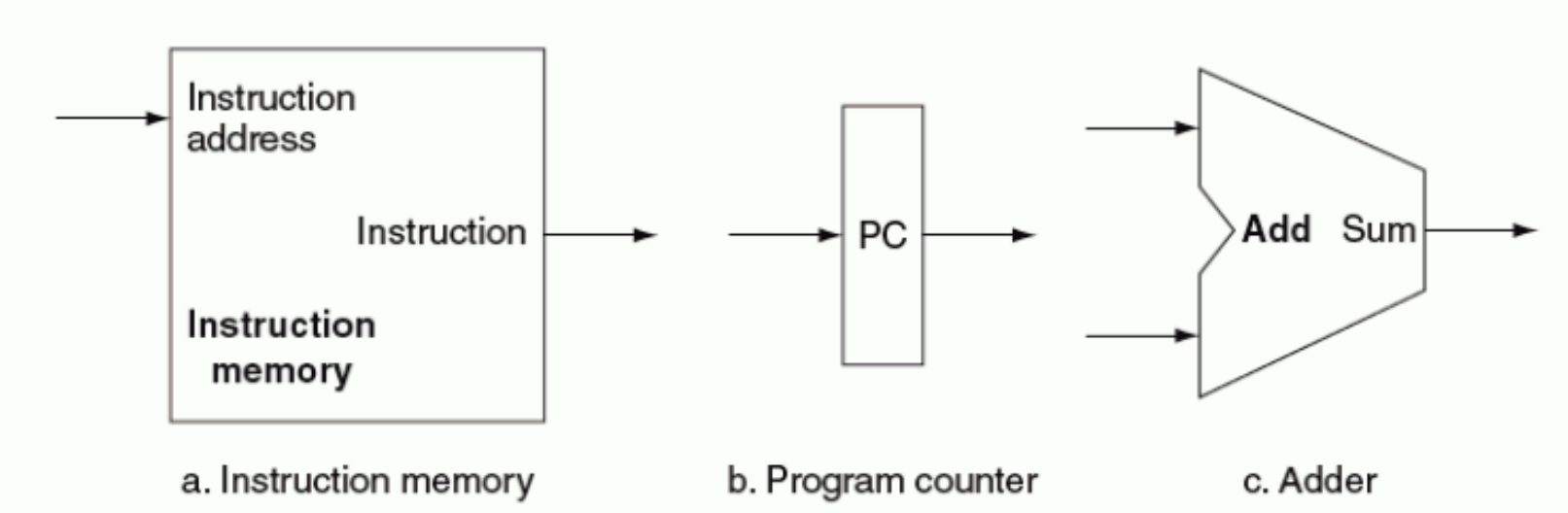


Outline

- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a Datapath
- 4.4 A Simple Implementation Scheme

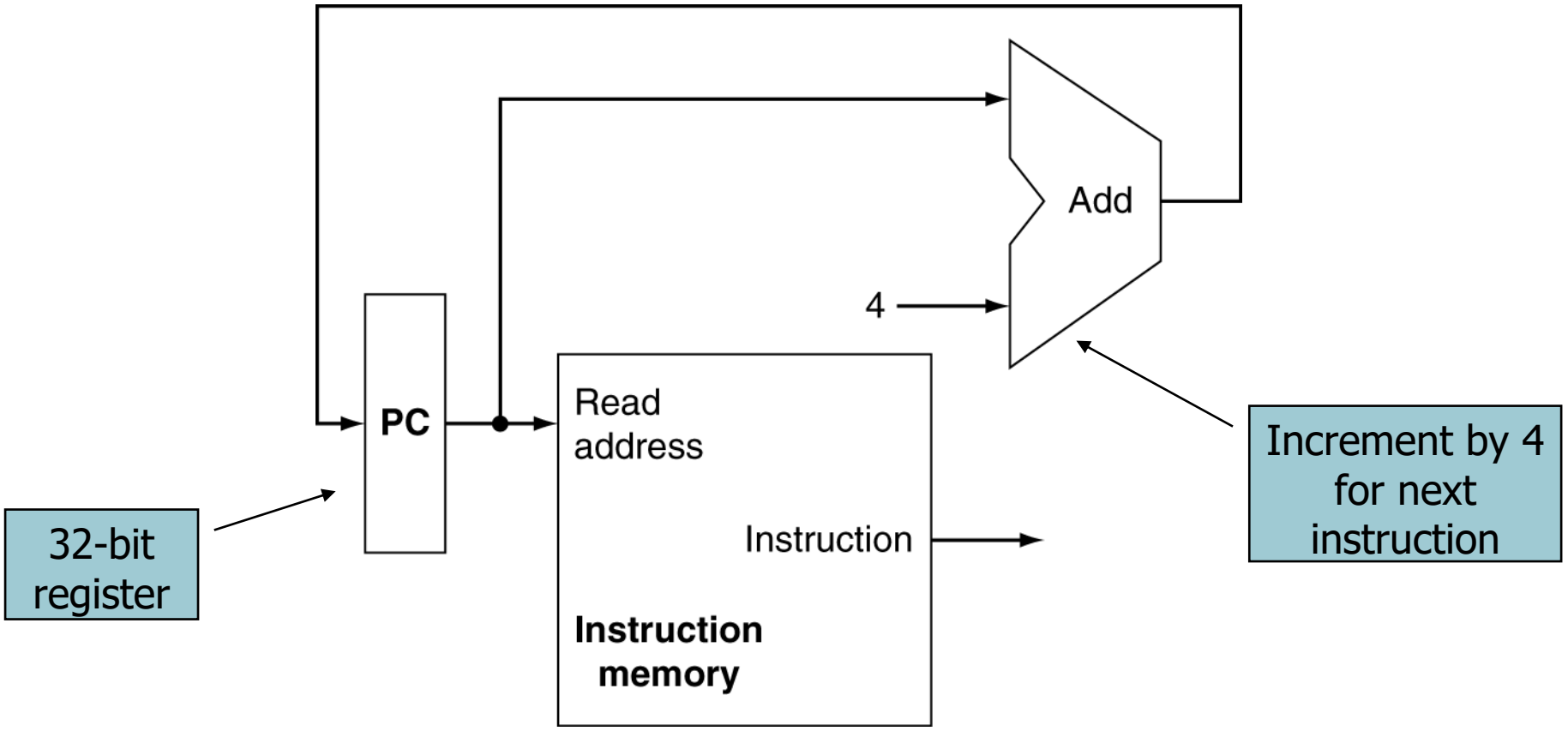
Building a Datapath

- Basic elements for “access” instructions:
  - (a) Instruction Memory (IM) unit
  - (b) Program Counter (PC): increase by 4 each time
  - (c) Adder: to perform “increase by 4”



Building a Datapath for PC

- A portion of datapath used for fetching instructions and incrementing the program counter



Building a Datapath for R-type

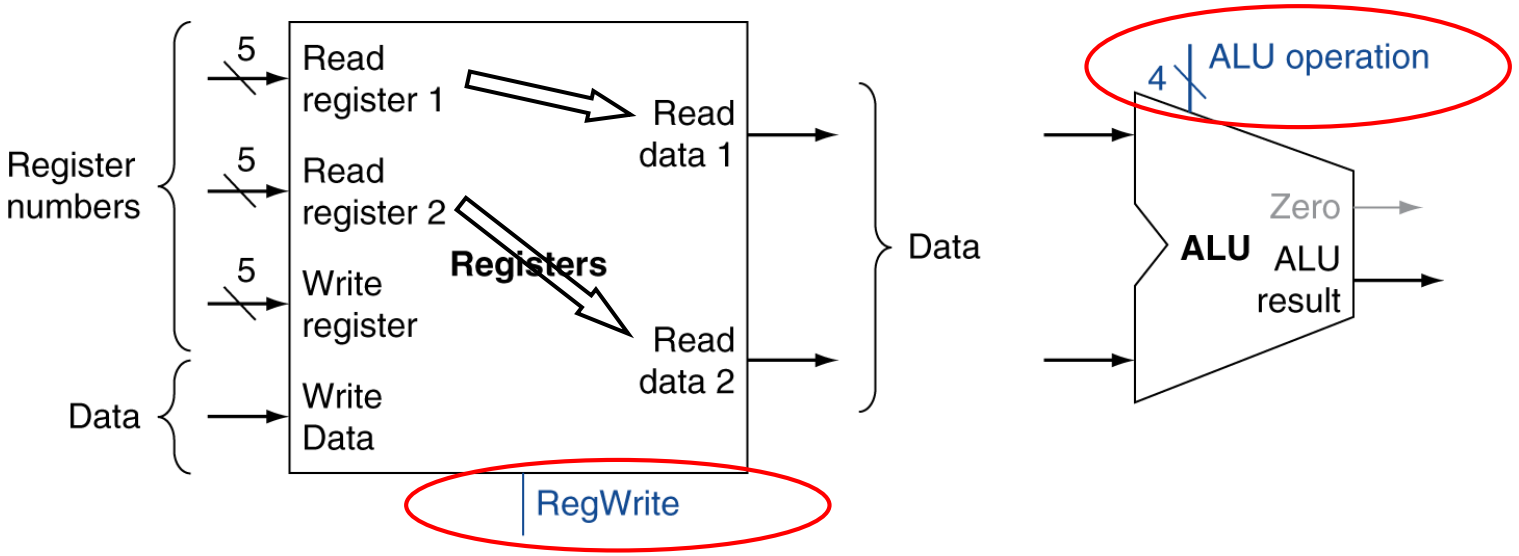
- Basic operations for R-type instructions:
  - Function:
    1. Read two registers
    2. Perform an ALU operation on the contents of registers
    3. Write the result back into the destination register
  - Read operation:
    1. Input to the “register file” to specify the indices of the TWO registers to be read.
    2. Two outputs of the register contents.
  - Write operation:
    1. An input to the “register file” to specify the index of the register to be written.
    2. An input (from ALU output) to supply the data to be written into the specified register.

Building a Datapath

- Elements which we need:

  (a) **Register file**: a collection of registers in which any register can be read or written by specifying the index of the register in the file.

  (b) **ALU (32 bits):** operate on the values read from the registers.



Review of Instruction Format

Field	0	rs	rt	rd	shamt	funct
Bit positions	31:26	25:21	20:16	15:11	10:6	5:0

a. R-type instruction

Field	35 or 43	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

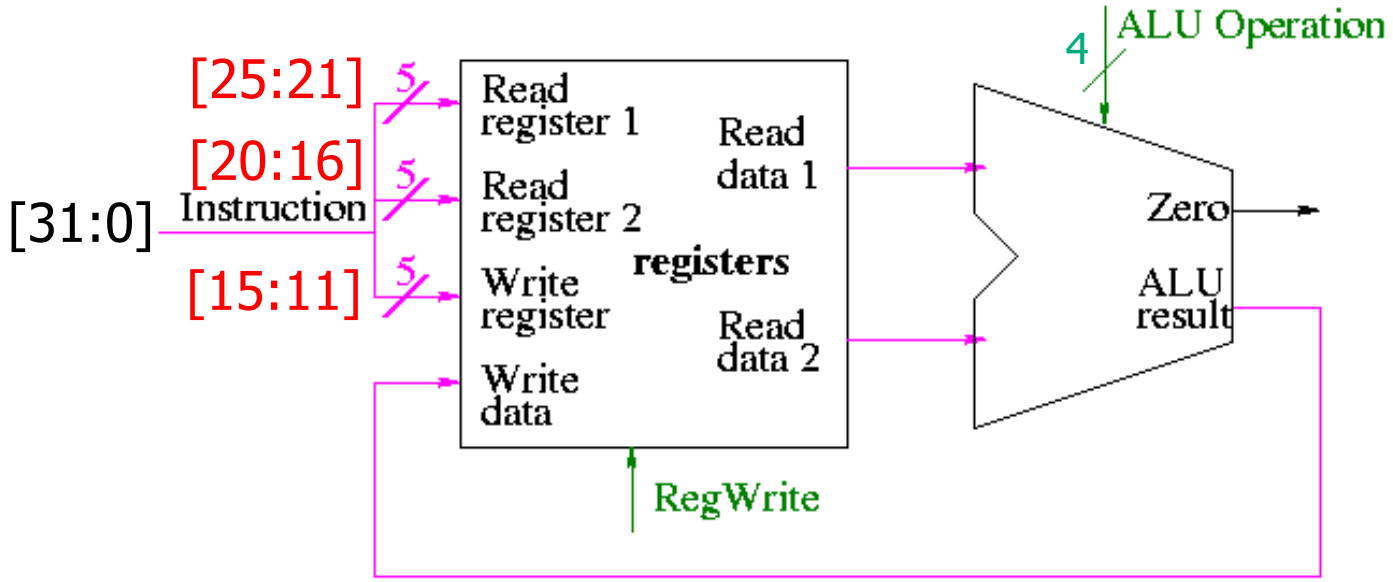
b. Load or store instruction

Field	4	rs	rt	address
Bit positions	31:26	25:21	20:16	15:0

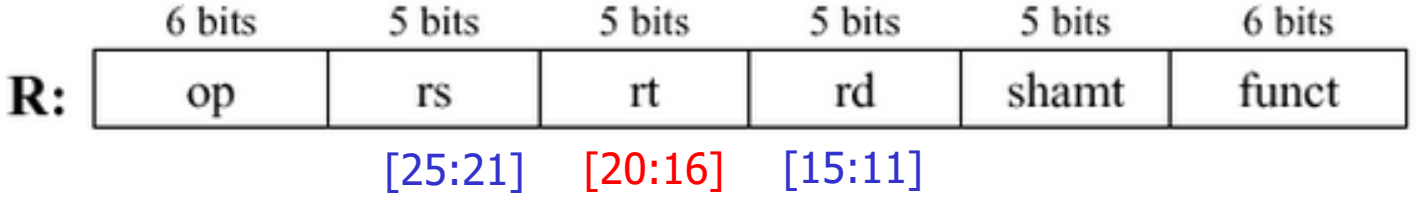
c. Branch instruction

FIGURE 4.14 The three instruction classes (R-type, load and store, and branch) use two different instruction formats.

Datapath for R-type instructions

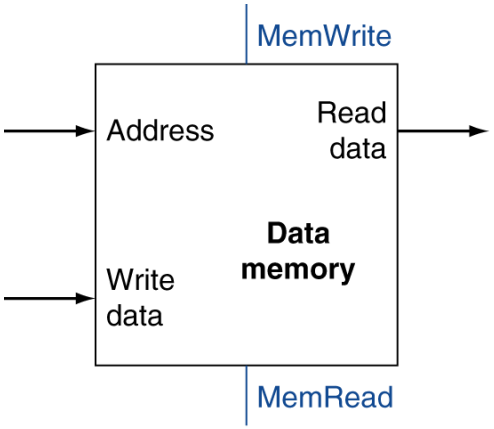


add $rd, $rs, $rt

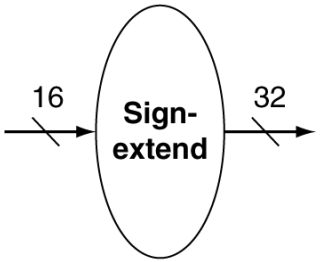


Building a Datapath for I-type

- Basic elements for load/store instructions:
  1. Data memory unit: read/write data
  2. Sign-extend unit: sign-extend the 16-bit offset field in the instruction to a 32-bit signed value.
  3. Register file
  4. ALU (add "reg" + "offset" to computer the mem address)



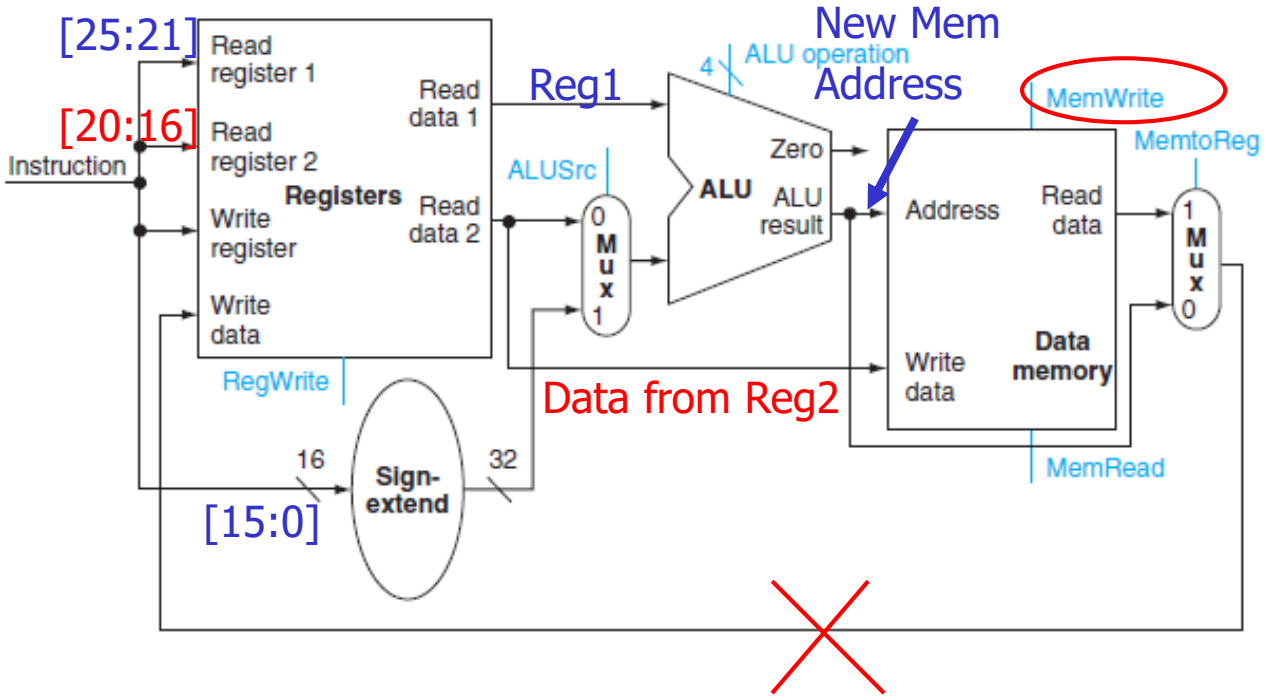
a. Data memory unit



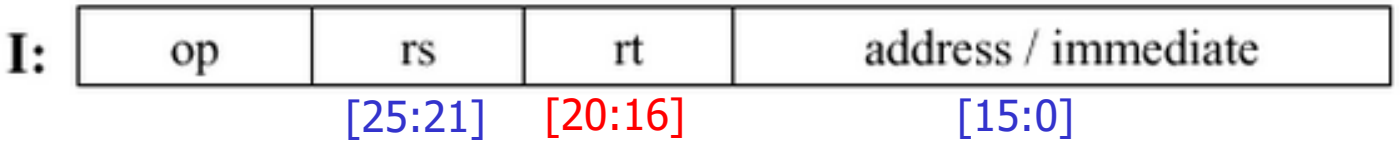
b. Sign extension unit

-- (3) & (4) are just shown as the previous slide.

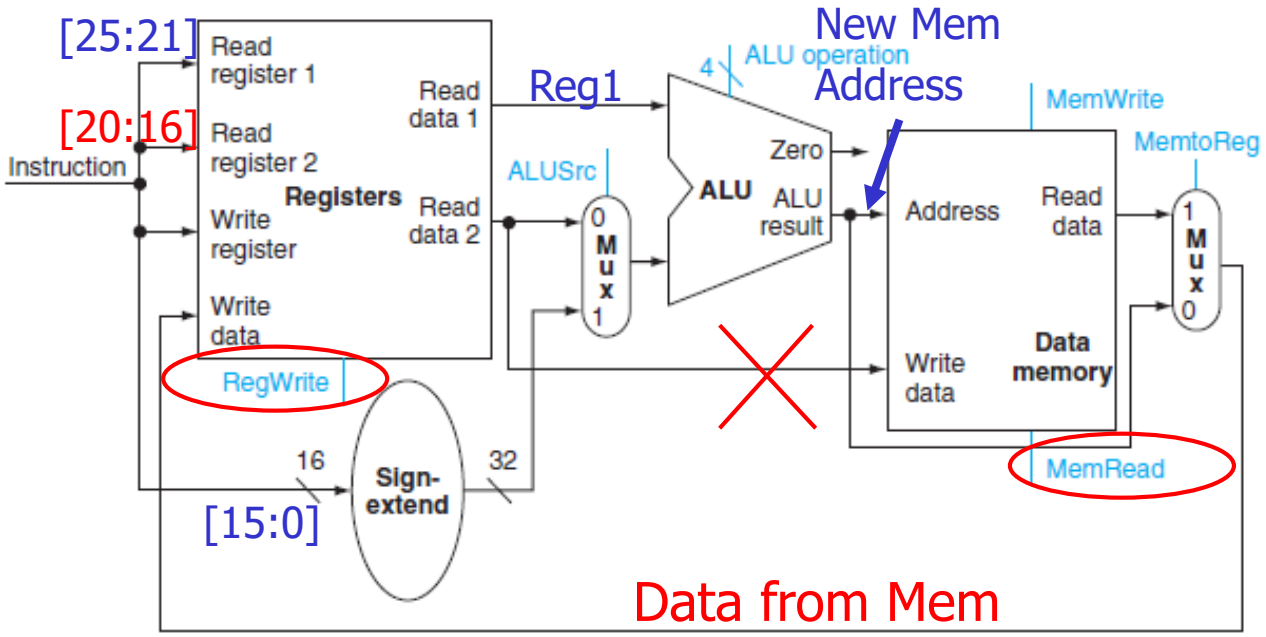
Datapath for **sw** instructions



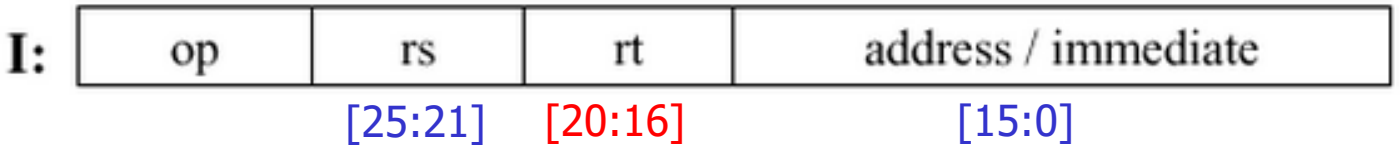
sw $rt, 100($rs)



Datapath for **lw** instructions



lw $rt, 100($rs)



Branch Instructions

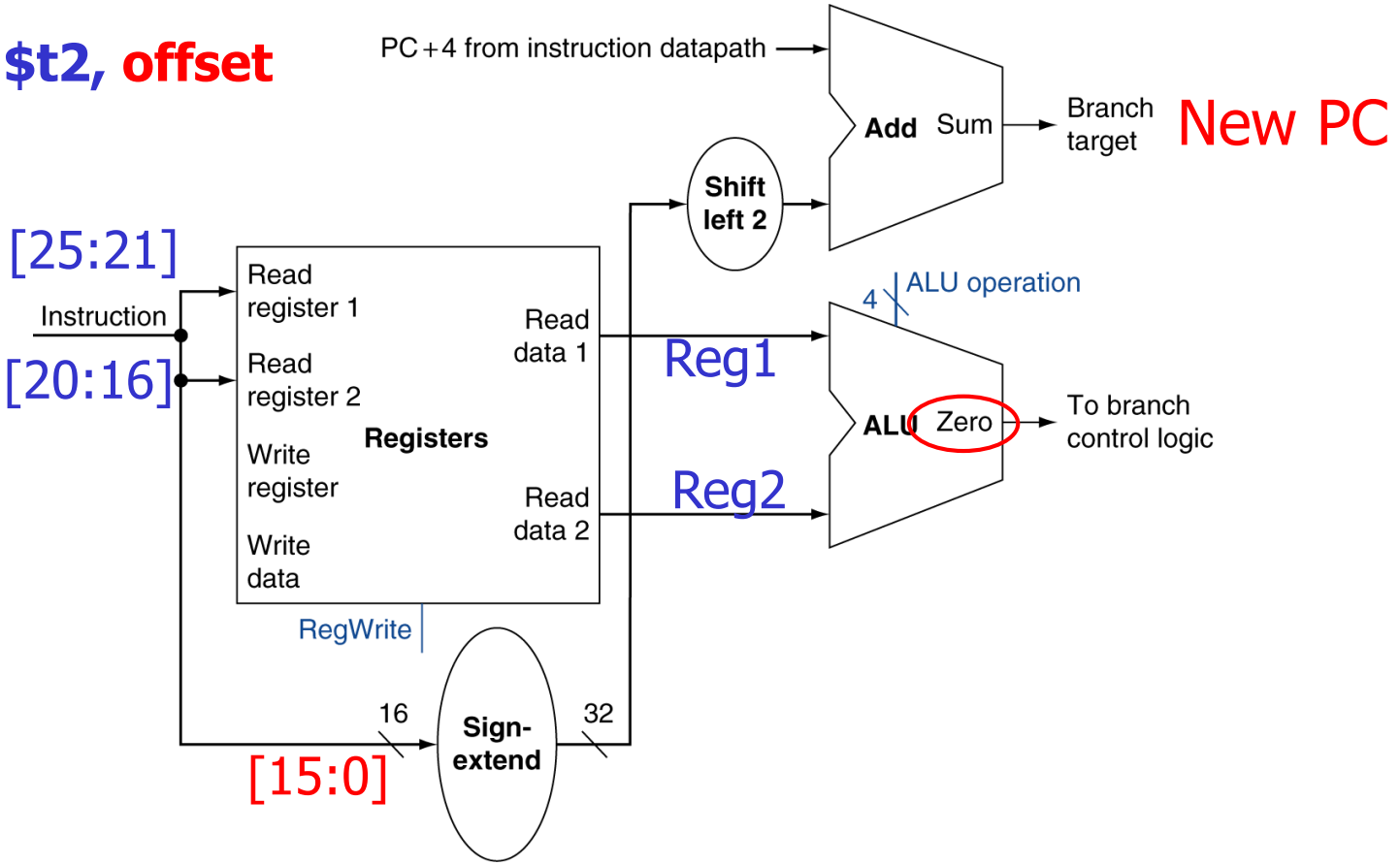
- (ex) beq $t1, $t2, offset

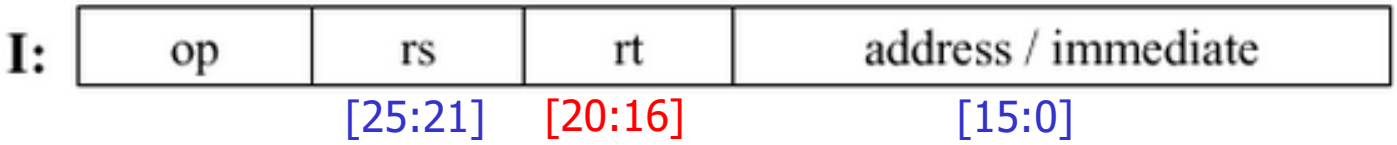
```
        # if ($t1==$t2)
                goto  (PC+4+offset)      // PC ← PC+4+offset
          else
                execute next instruction  //  PC ← PC+4
```

- Note:
  (1) The offset field is shifted left 2 bits so that it's a "word offset".
  (2) Branch is taken (taken branch): when the condition is true, the branch target address becomes the new PC.
  (3) Branch isn't taken (untaken branch): the incremented PC (PC+4) replaces the current PC, just as for normal instruction.
- Operations:
  (1) Compute the branch target address.
  (2) Compare the contents of the two registers.

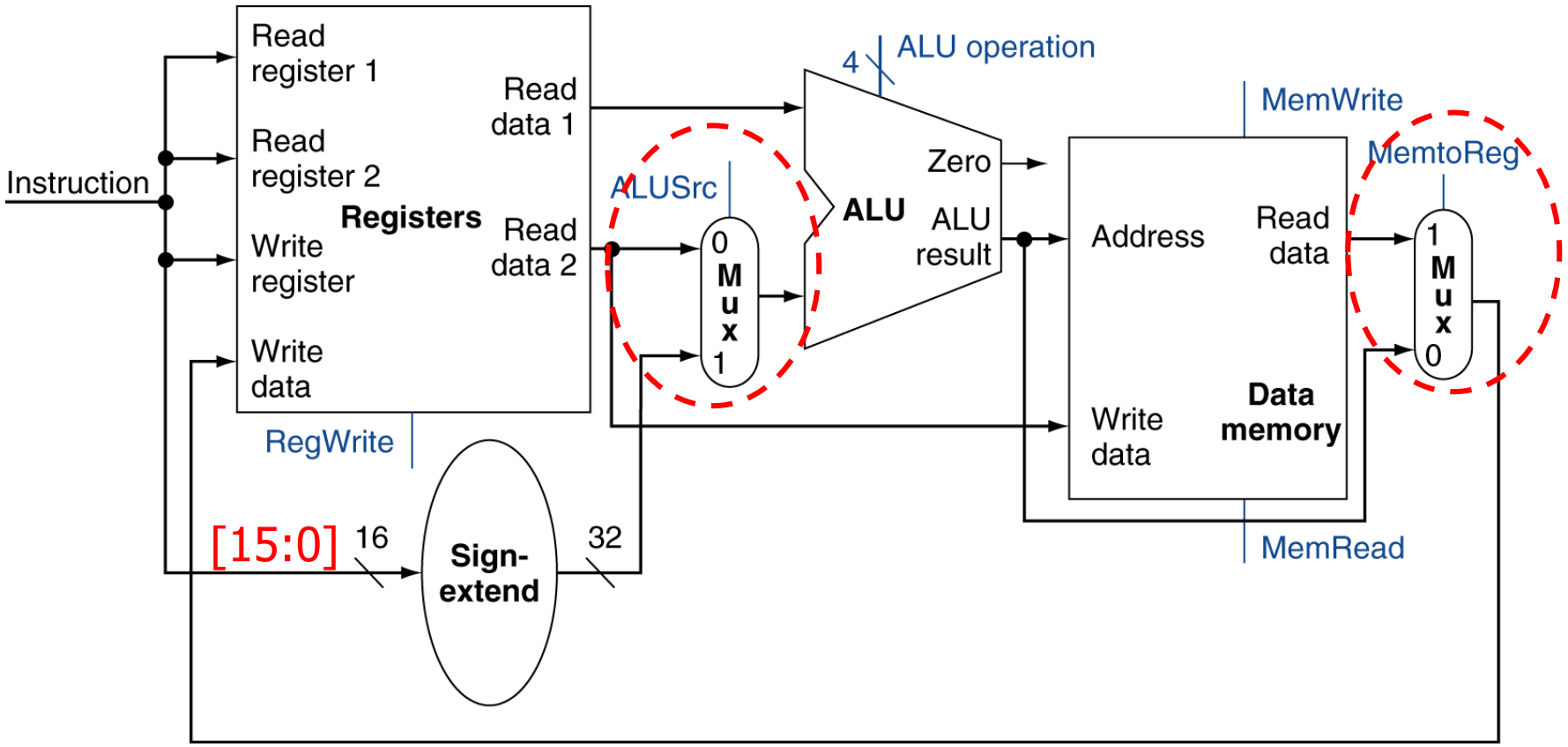
Datapath for "beq" Instructions

- **beq $t1, $t2, offset**

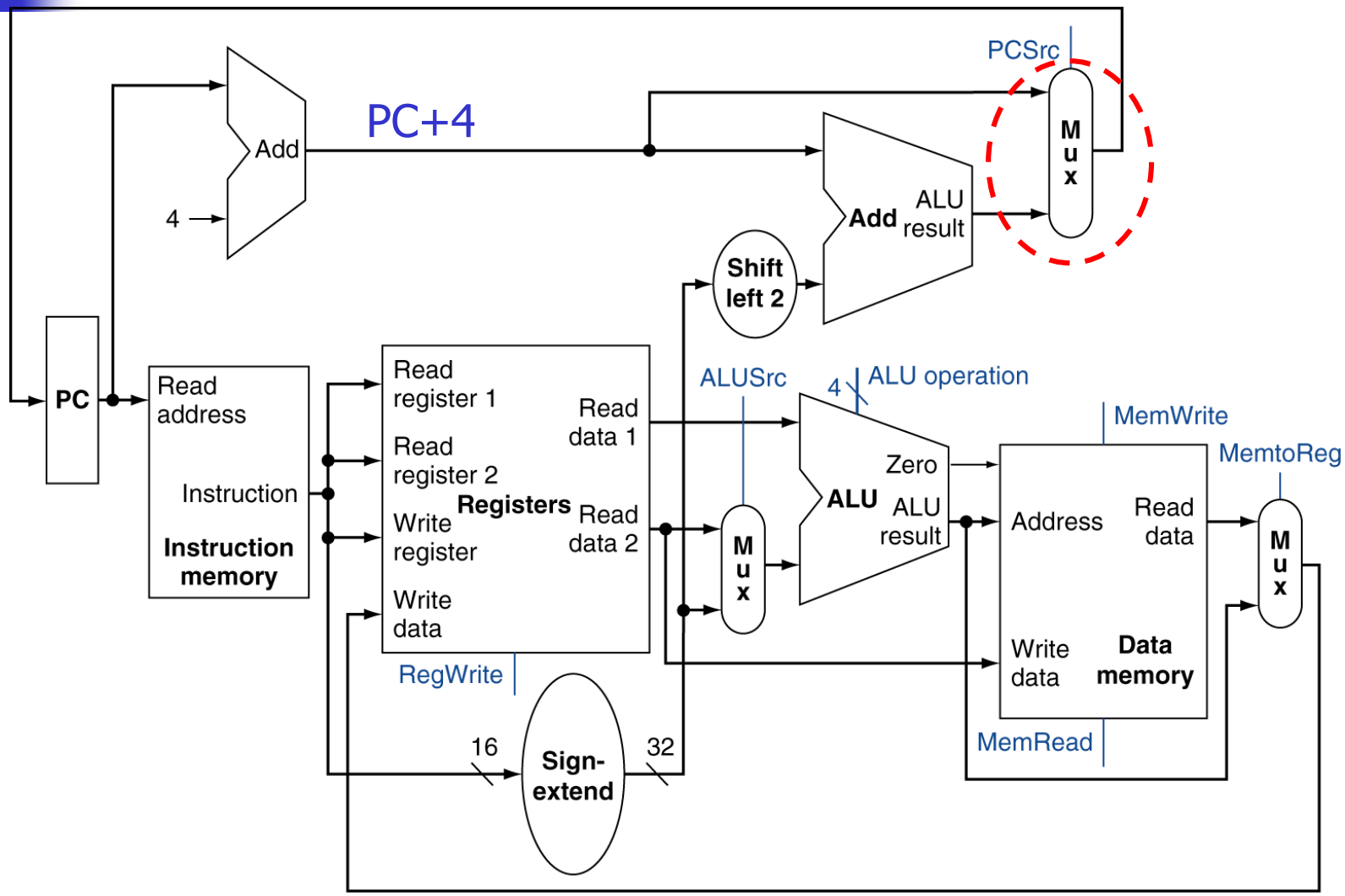




Datapath for both Memory and R-type Instructions



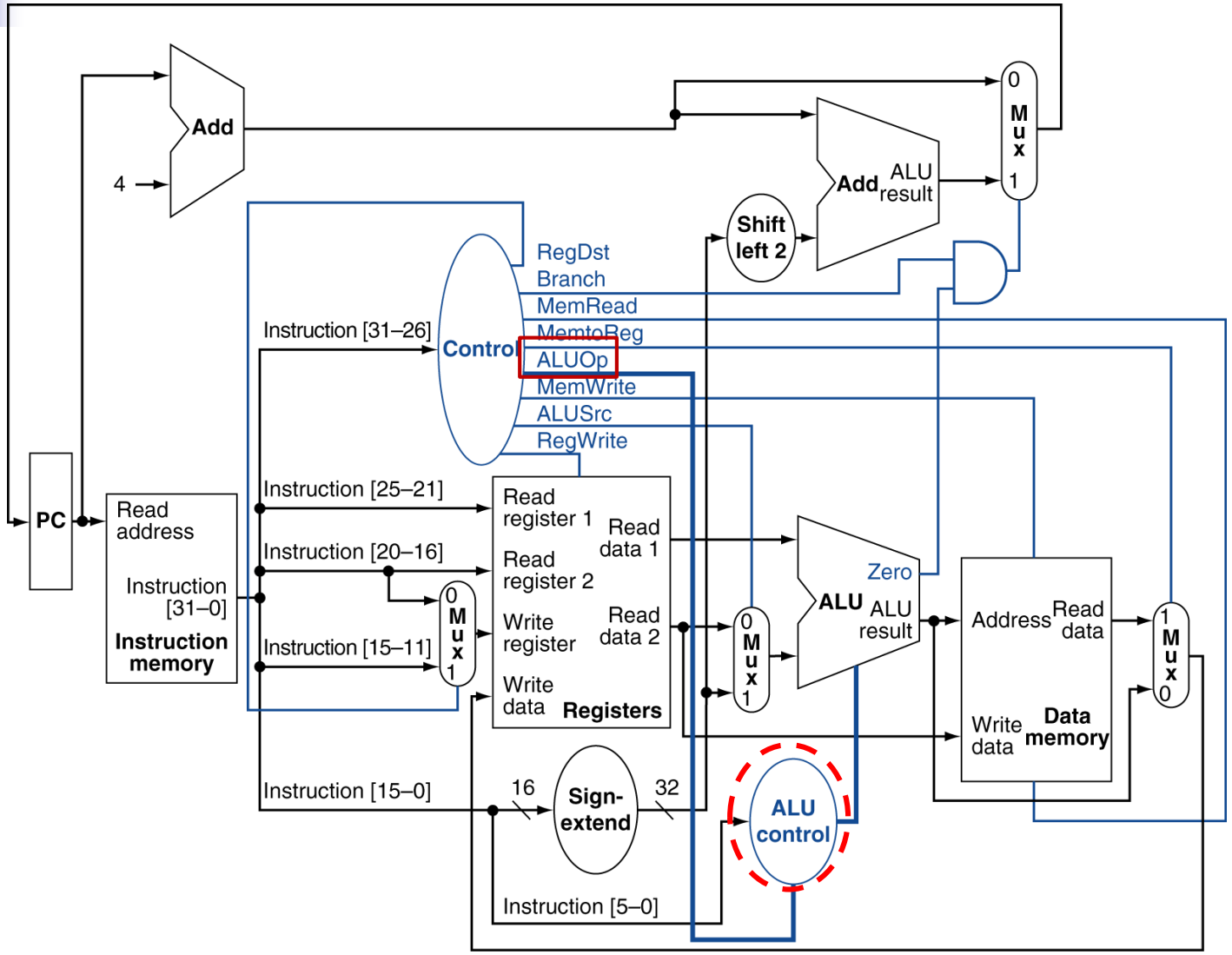
Simple Datapath for All three types of Instructions



Outline

- 4.1 Introduction
- 4.2 Logic Design Conventions
- 4.3 Building a Datapath
- 4.4 A Simple Implementation Scheme

Basic Datapath with Control Signals



Design of ALU control unit

- Depending on the instruction type, the ALU will perform
  - lw/sw: compute the memory address by addition
  - R-type (add, sub, AND,OR, slt): depending on the value of the 6-bit function field
  - Branch (beq): subtraction (R1-R2)

- ALU control signals:

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

ALU control for each type of instruction

- Assume 2-bit ALUOp derived from opcode
  - Combinational logic derives ALU control

Instruction opcode	ALUOp	Instruction operation	Funct field	Desired ALU action	ALU control input
LW	00	load word	xxxxxx	add	0010
SW		store word	xxxxxx	add	0010
Branch equal	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	100000	add	0010
R-type		subtract	100010	subtract	0110
R-type		AND	100100	and	0000
R-type		OR	100101	or	0001
R-type		set on less than	101010	set on less than	0111

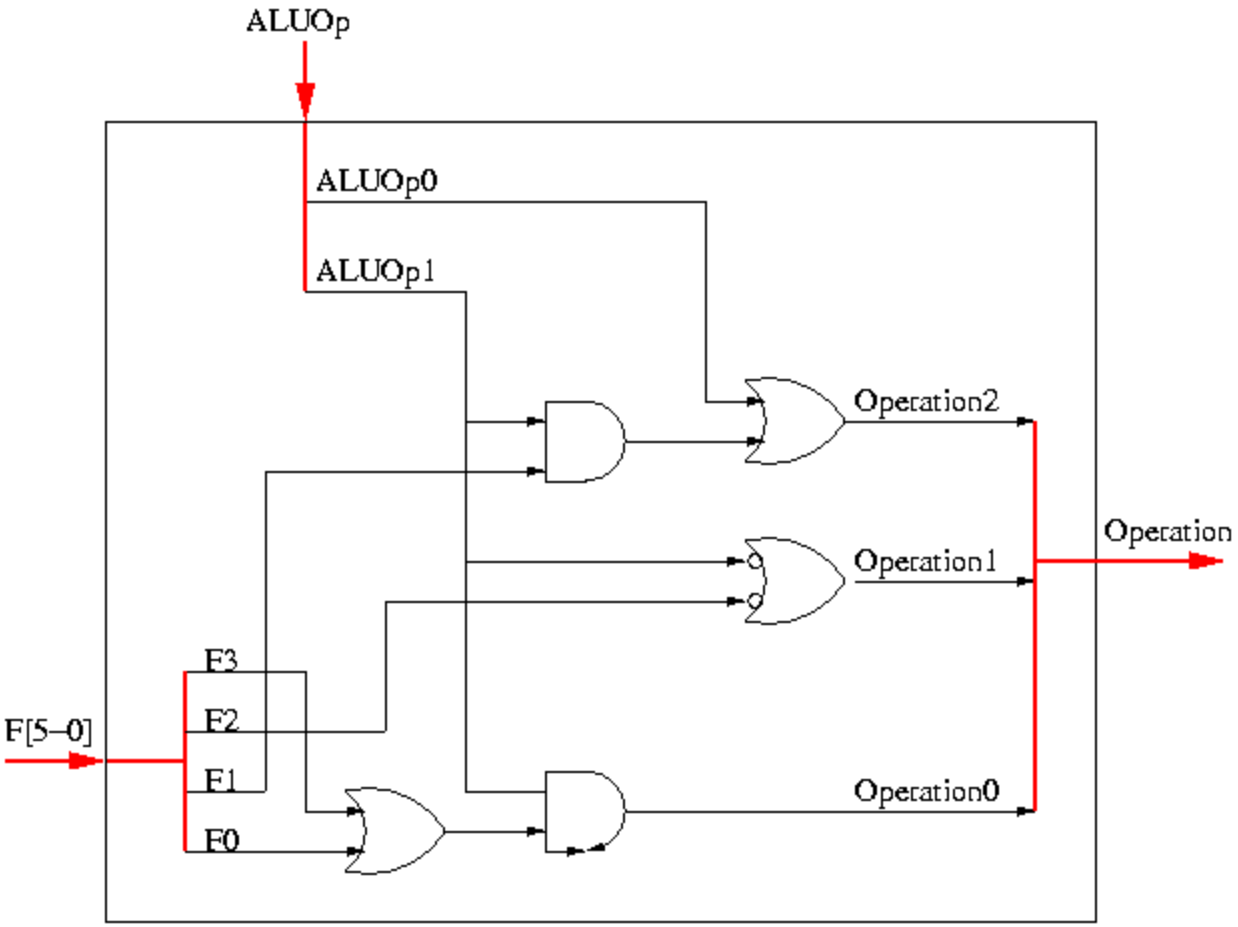
A Simple Implementation Scheme

- The truth table for the three ALU control bits (called Operation)

ALUOp		Funct field						ALU Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
x	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	1111

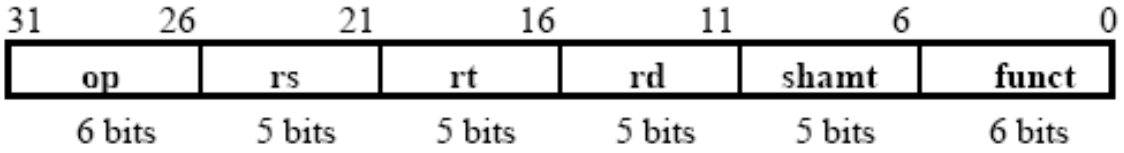
Simplify the ALU Control Design

- ALU control logic (overall)

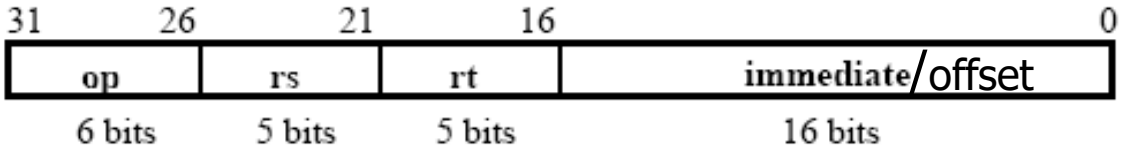


Designing the main control unit

- The two instruction classes
  - R-type:



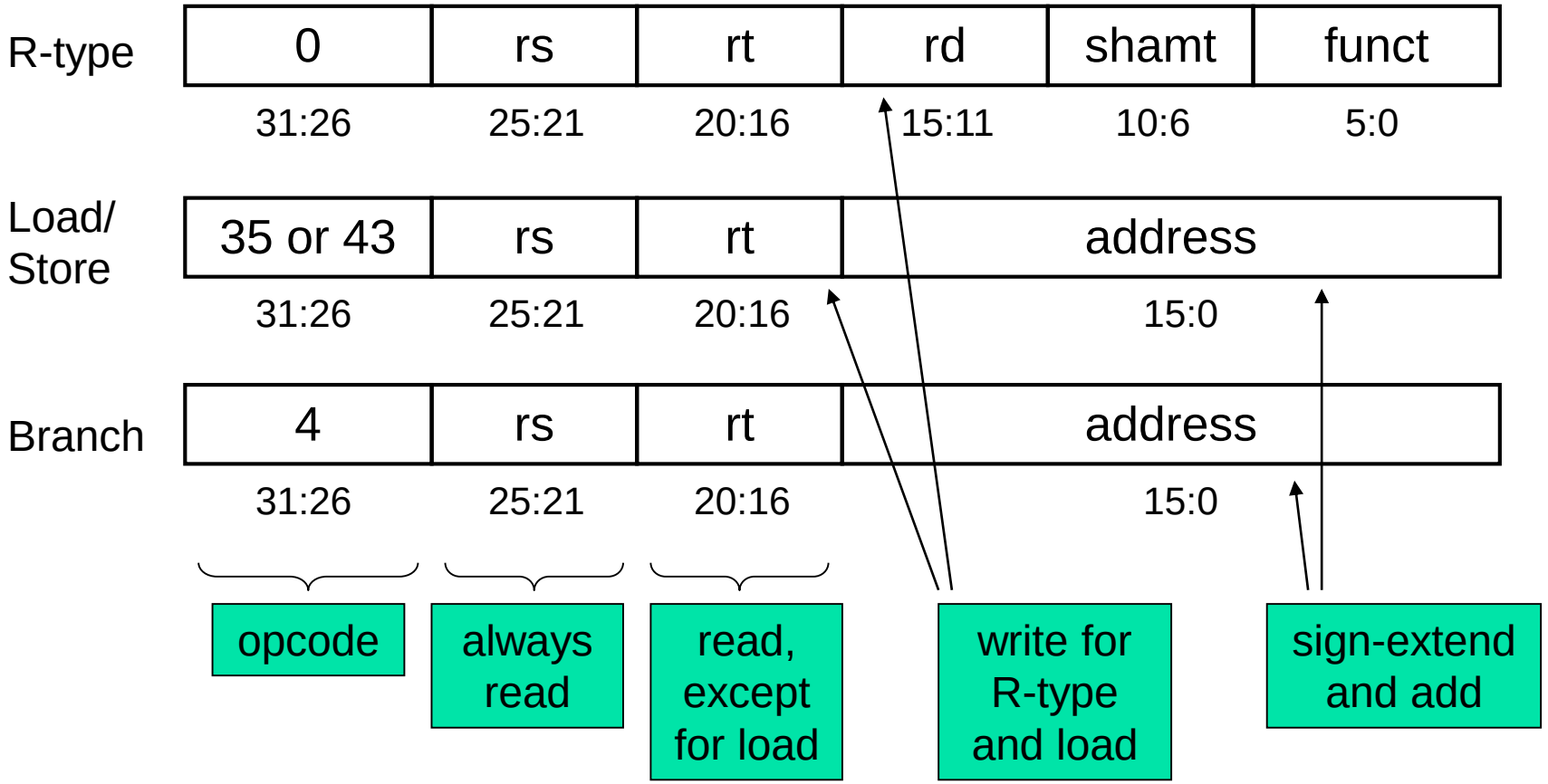
  - I-type:



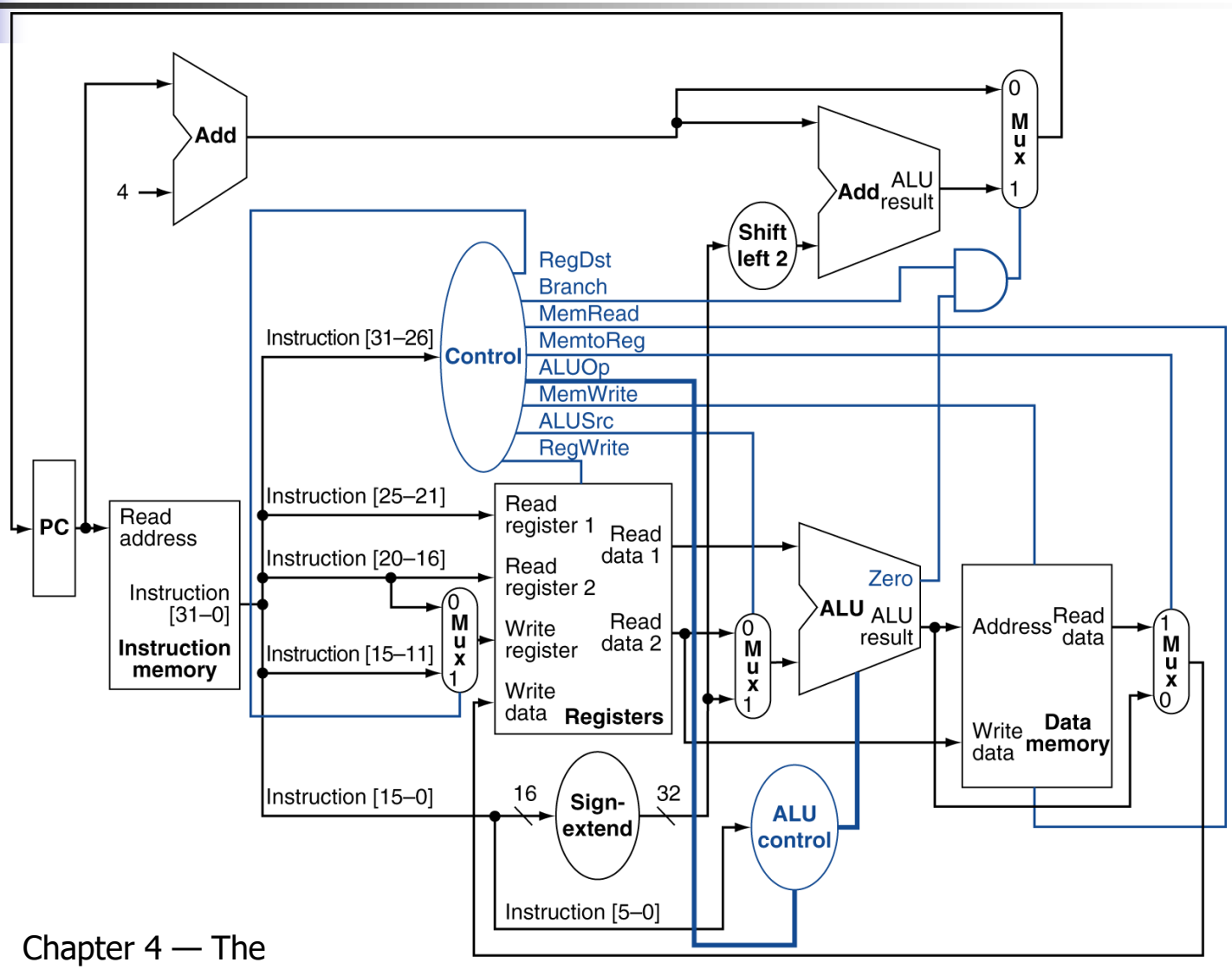
- Observations:
  - op field: opcode (bit[31:26], which is called Op[5:0].
  - The two registers to be read are specified by rs & rt (for R-type, beq).
  - Base register (for lw, sw) is rs.
  - 16-bit offset (for lw, sw, beq) is bit[15:0] (also immediate values)
  - The destination register is in one of the two places:
    - lw : rt, bit[20:16]
    - R-type : rd, bit[15:11]

The Main Control Unit

- Control signals derived from instruction



Datapath With Control



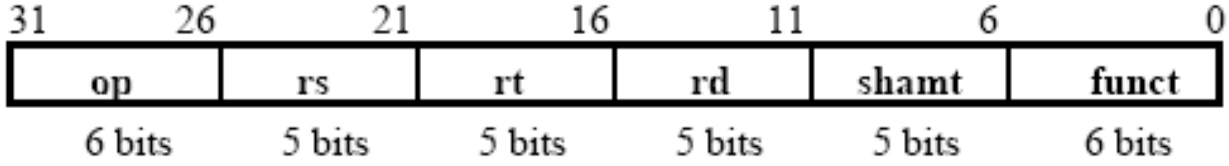
Effect of the 7 control signals

Signal name	Effect when deasserted(0)	Effect when asserted(1)
RegDst	The register destination number for the Write register comes from the rf field(bits20-16).	The register destination number for the Write register comes from the rd field(bits15-11).
RegWrite	None	The register on the Write register input is written with the value on the Write data input.
ALUSrc	The second ALU operand comes from the second register file output (Read data 2).	The second ALU operand is the sign-extend, lower 16 bits of the instruction.
PCSrc	The PC is replaced by the output of the adder that computes the value of PC + 4.	The PC is replaced by the output of the adder that computes the branch target.
MemRead	None	Data memory contents designated by the address input are put on the Read data output.
MemWrite	None	Data memory contents designated by the address input are replaced by the value on the Write data input.
MemtoReg	The value fed to the register Write data input comes from the ALU.	The value fed to the register Write data input comes from the data memory.

Operation for R-type instruction

- The 4 steps of the operation for R-type instruction

- R-type:



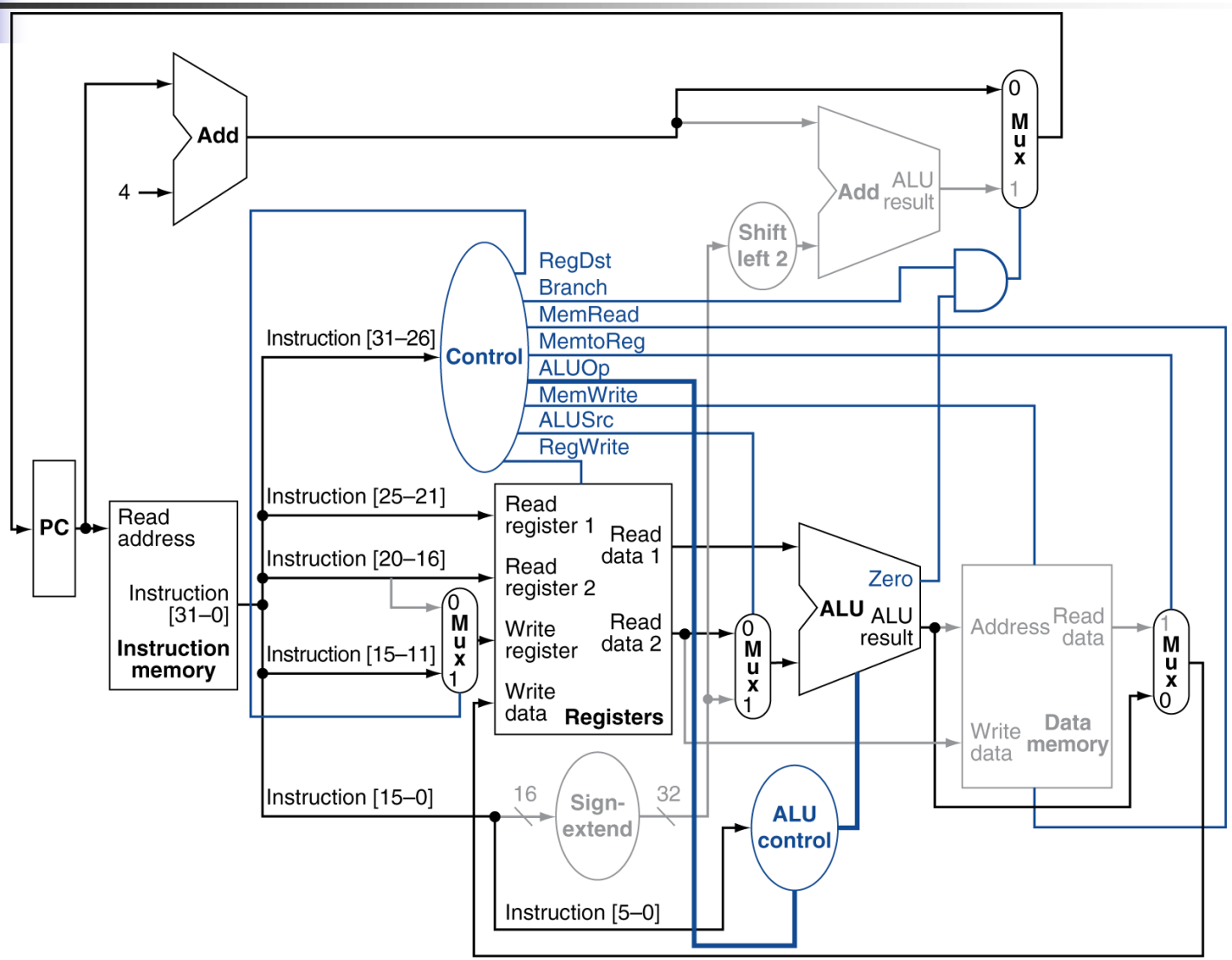
add $t1, $t2, $t3

- Fetch instruction and increment PC

  (Instr=Memory[PC] ; PC = PC + 4)

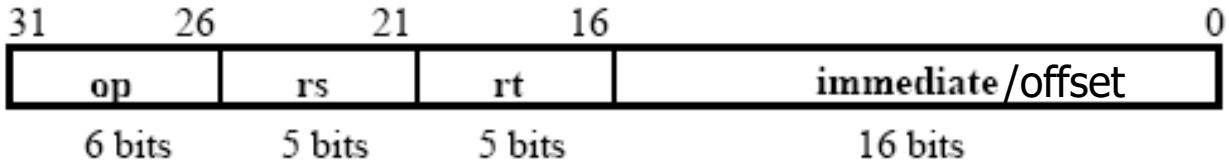
- Read registers (Reg1=Reg[rs], Reg2=Reg[rt])
- Run the ALU operation (Result = Reg1 ALUop Reg2)
- Store the result into Register File (Reg[rd] = Result)

R-Type Instruction



Operation for “load” instruction

- The 5 steps of the operation for “load” instruction
  - I-type:

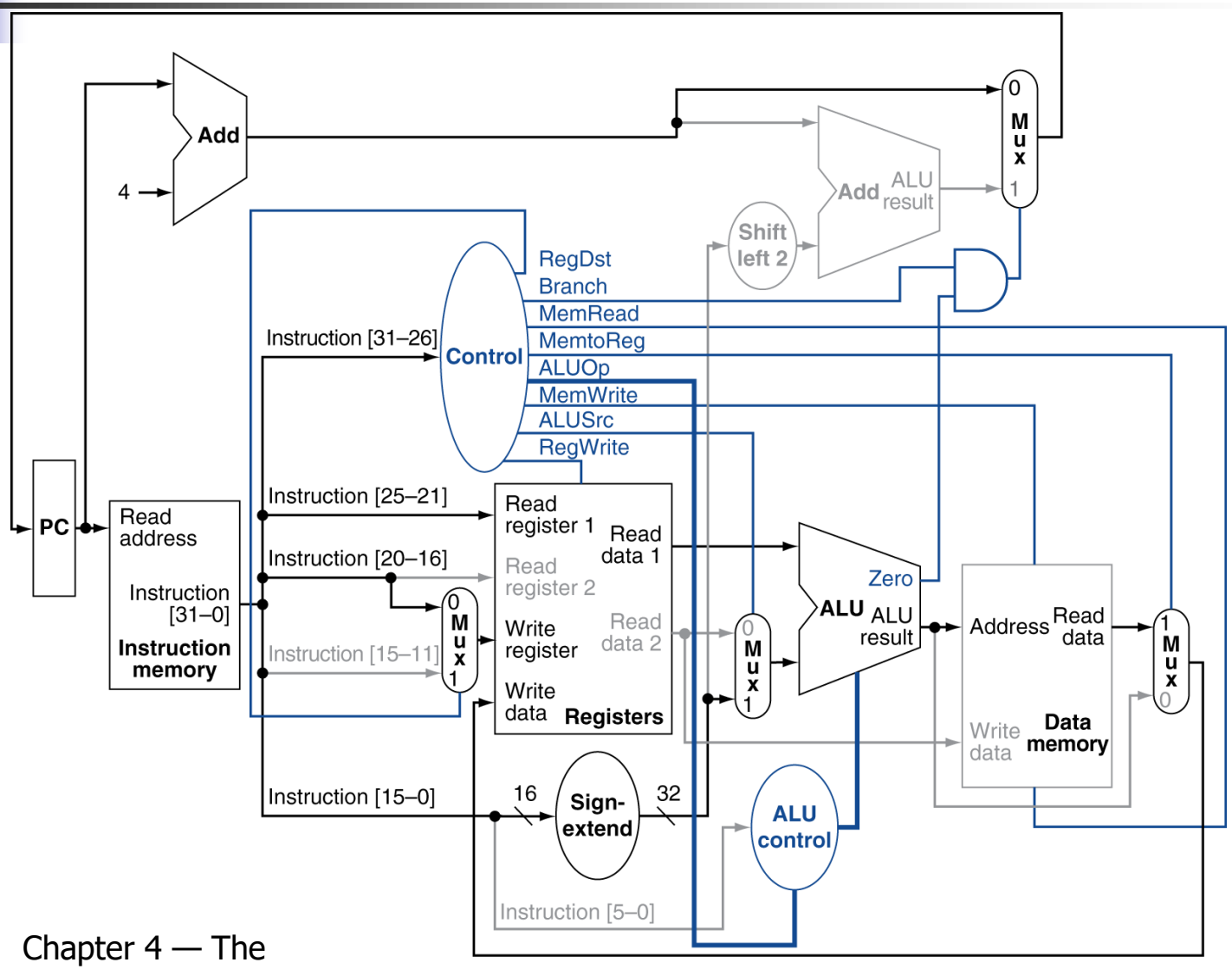


lw $t1, offset($t2)

- Fetch instruction and increment PC

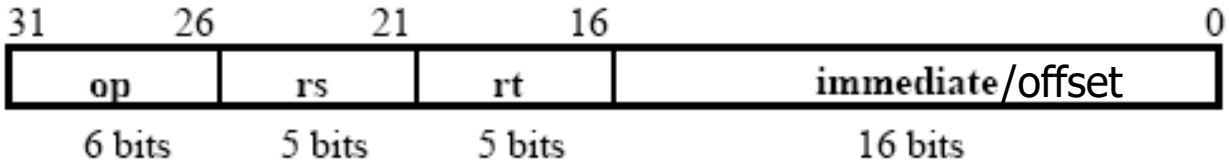
  (Instr=Memory[PC] ; PC = PC + 4)
- Read registers (temp = Reg[rs] , only one register is read)
- Address computing (Result = temp + sign-extend(Instr[15-0]))
- Load data from memory (Data = Memory[Result])
- Store data into Register File (Reg[rt] = Data)

Load Instruction



Operation for “store” instruction

- The 4 steps of the operation for “store” instruction
  - I-type:



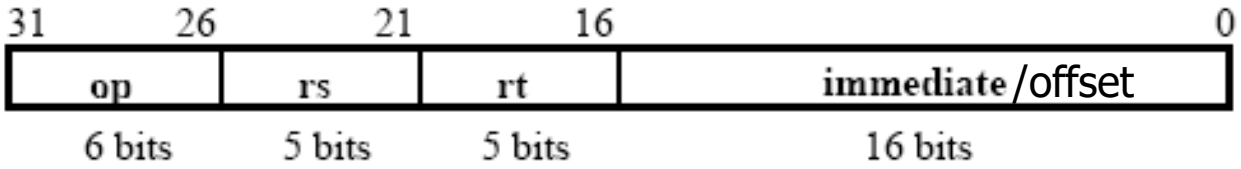
sw $t1, offset($t2)

- Fetch instruction and increment PC

  (Instr=Memory[PC] ; PC = PC + 4)
- Read two registers (Reg1=Reg[rs], Reg2=Reg[rt])
- Address computing (Result = Reg1 + sign-extend(Instr[15-0]))
- Store data into memory (Memory[Result] = Reg2)

Operation for “beq” instruction

- The 3 steps of the operation for “beq” instruction
  - I-type:



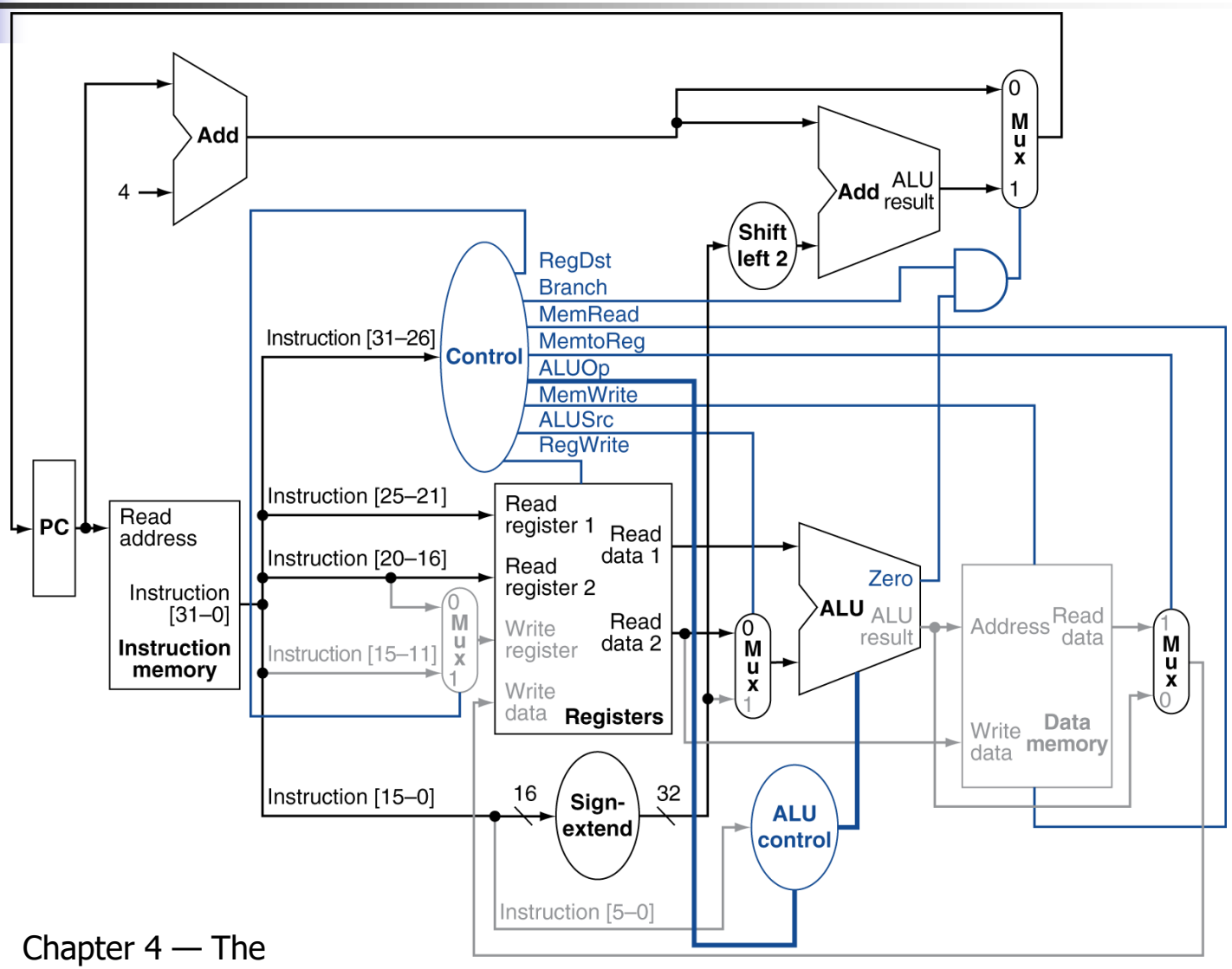
beq $t1, $t2, offset

- Fetch instruction and increment PC

  (Instr=Memory[PC] ; PC = PC + 4)
- Read two registers (Reg1=Reg[rs], Reg2=Reg[rt])
  - Compute **branch target address** (Result = PC + (sign-extend (Instr[15-0] << 2)))
  - Run the ALU operation (Result = Reg1 minus Reg2)
- Observe “zero” to branch or not

  (If zero==1, then PC = Result. Otherwise, PC unchanged)

Branch-on-Equal Instruction



Control Unit Design

- The setting of the control lines is completed by the “opcode” field (op[5:0]) of the instruction.

Op<5-0>	Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
000000	R-format	1	0	0	1	0	0	0	1	0
100011	lw	0	1	1	1	1	0	0	0	0
101011	sw	X	1	X	0	0	1	0	0	0
000100	beq	X	0	X	0	0	0	1	0	1

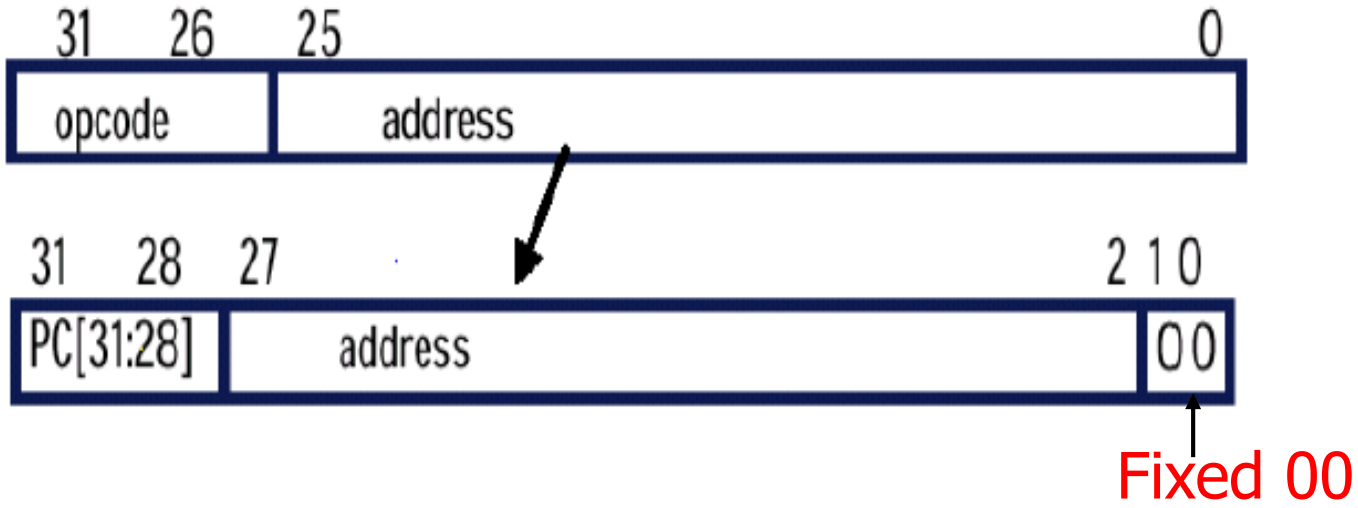
Note this table can be further simplified. (e.g. Branch is the same as ALUop0)

Finalizing the control signals

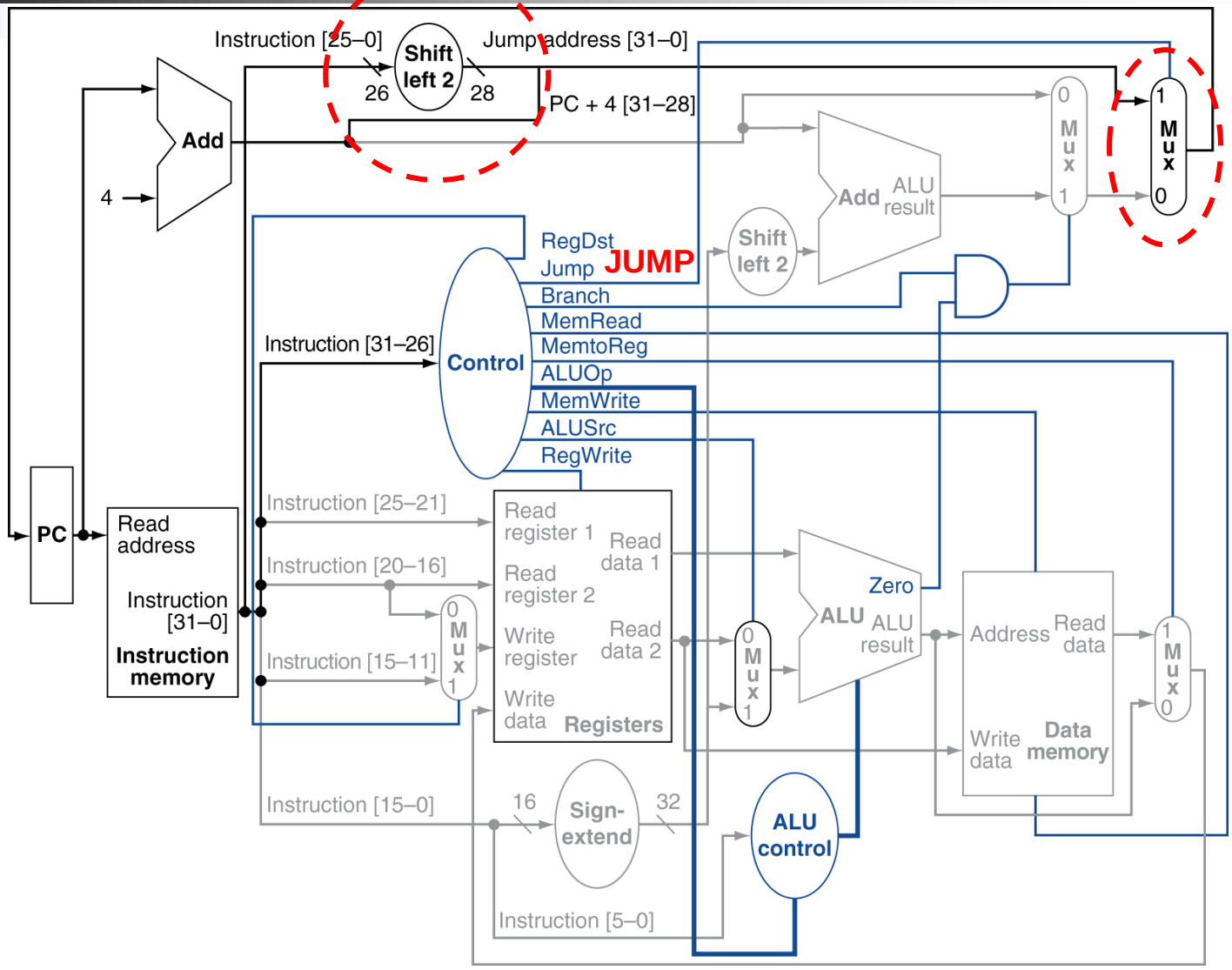
Input/output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	x	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

Datapath for "Jump"

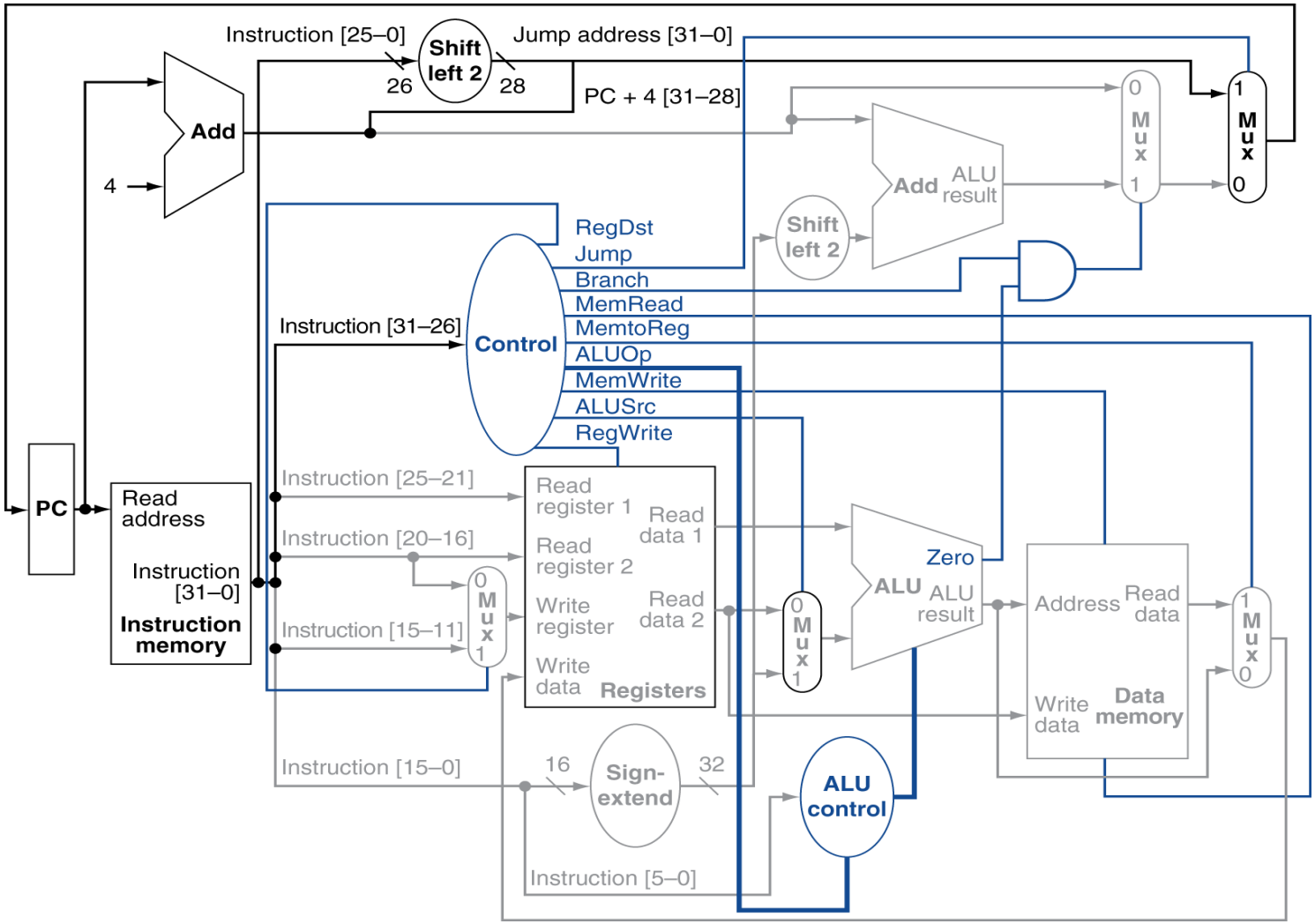
- "Jump" operation: (opcode = 000010)
  - Replace a portion of the PC(bit 27-0) with the lower 26 bits of the instruction shifted left by 2 bits.
  - The shift operation is accomplished by simple concatenating "00" to the jump offest.



Implementing “Jumps”

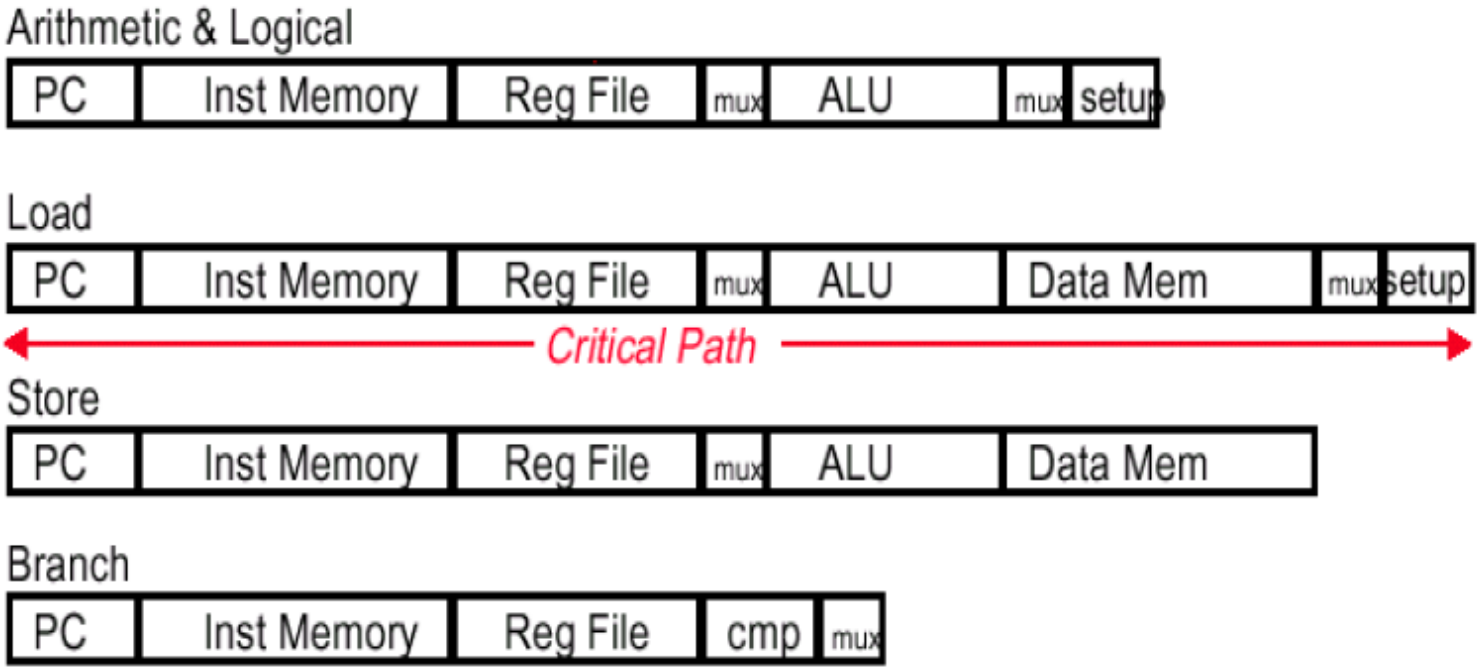


Single-cycle MIPS Implementation with 4 Instructions



Single-cycle implementation

- Why a single-cycle implementation isn't used today?



  - Long cycle time for each instruction (load takes longest time)
  - All instructions take as much time as the slowest one

Performance of single-cycle implementation

- Example:
  - Assumption:
    - Memory units : 200 ps
    - ALU and adders : 100 ps
    - Register file (read / write) : 50 ps
    - Multiplexers, control unit, PC accesses, sign extension unit, and wires have no delay.
    - The **instruction mix**: 25% loads, 10% stores, 45% ALU instructions, 15% branches, 5% jumps.
  - Problem: which one would be faster and by how much?

    (1) Fixed clock cycle

    (2) Variable-length clock cycle

Performance of single-cycle implementation

- Answer:
  - The critical path for the different instruction classes:

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Memory access	Register access
Store word	Instruction fetch	Register access	ALU	Memory access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

  - Compute the require length for each instruction class:

Instruction class	Instruction memory	Register read	ALU operation	Data memory	Register write	Total
R-type	200	50	100	0	50	400 ps
Load word	200	50	100	200	50	600 ps
Store word	200	50	100	200		550 ps
Branch	200	50	100	0		350 ps
Jump	200					200 ps

Performance of single-cycle implementation

- Calculation equations:
  - CPU execution time = instruction count * CPI * clock cycle time
  - Assume CPI=1, CPU execution time = instruction count * clock cycle time
- Calculate CPU execution time :

  (1) fixed clock cycle : 600 ps

  (2) variable-length clock cycle :

  600*25% + 550*10% + 400*45% + 350*15% + 200* 5%
  =447.5 ps

  -- The one with variable-length clock cycle is faster.

- Performance ratio:

$$\frac{\text{CPU clock cycle (fixed)}}{\text{CPU clock cycle (variable)}} = \frac{600}{447.5} = 1.34$$

Chapter4-2 (補充教材)

The Processor: Datapath and Control (Multi-cycle implementation)

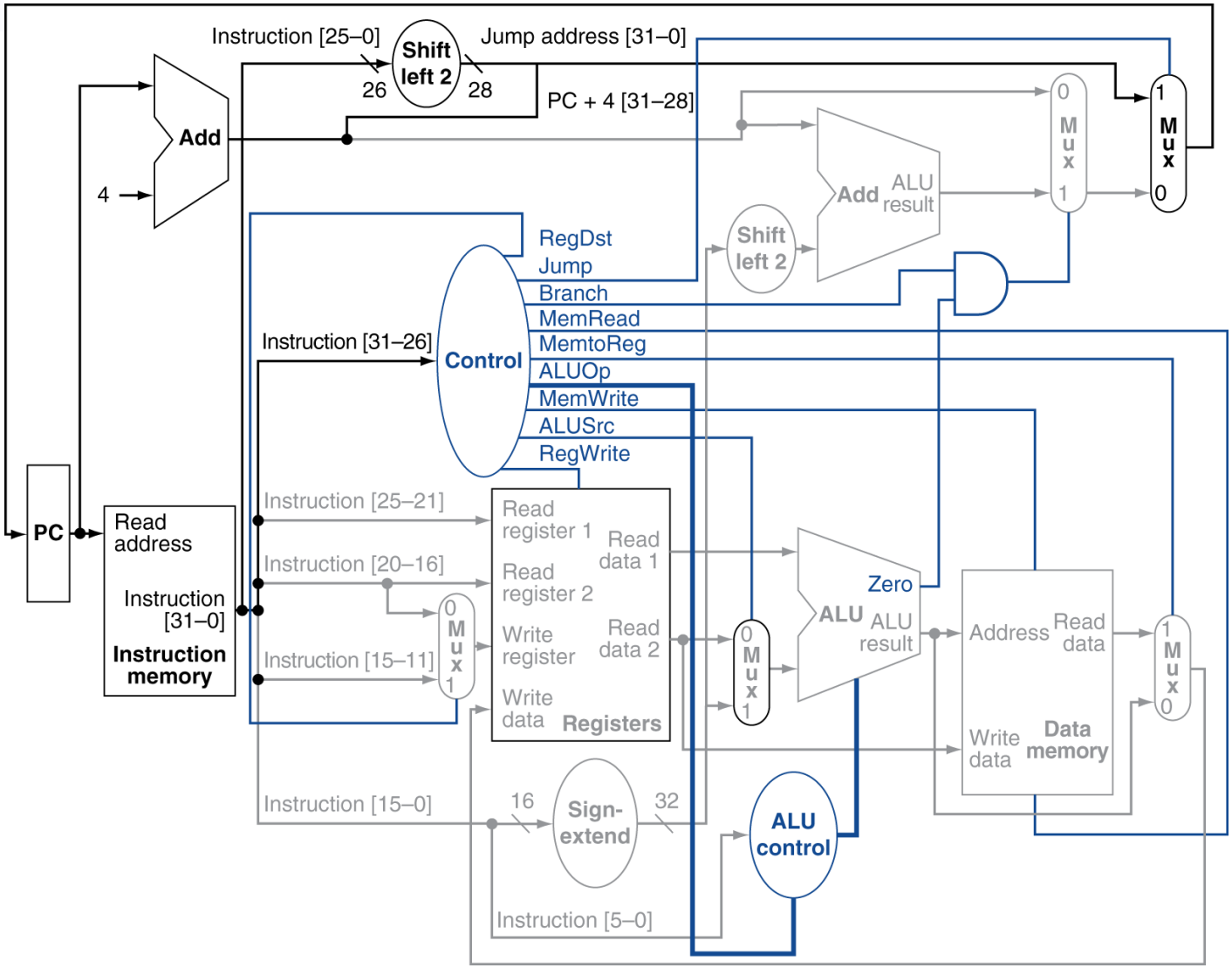
臺大電機系

吳安宇教授

Reference:

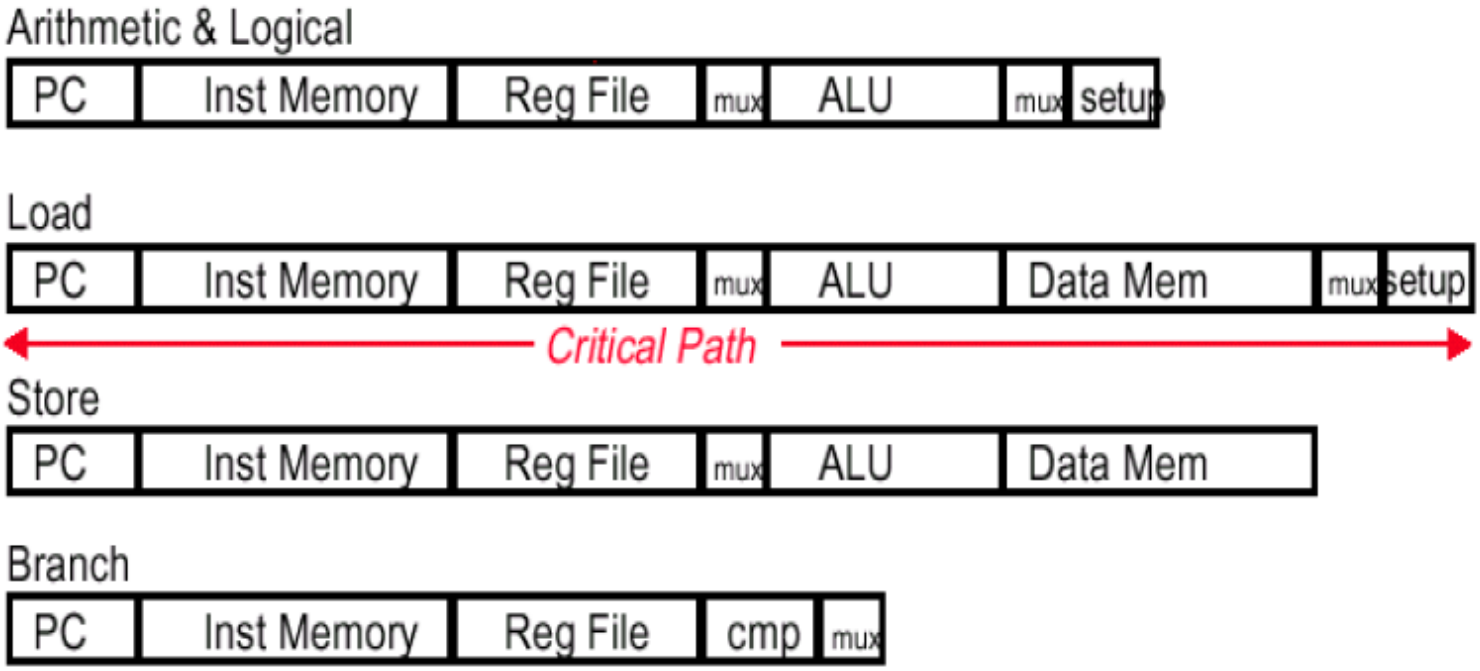
"Computer Organization and Design – The Hardware/Software Interface", ***3th Edition***

Review of Single-cycle Implementation



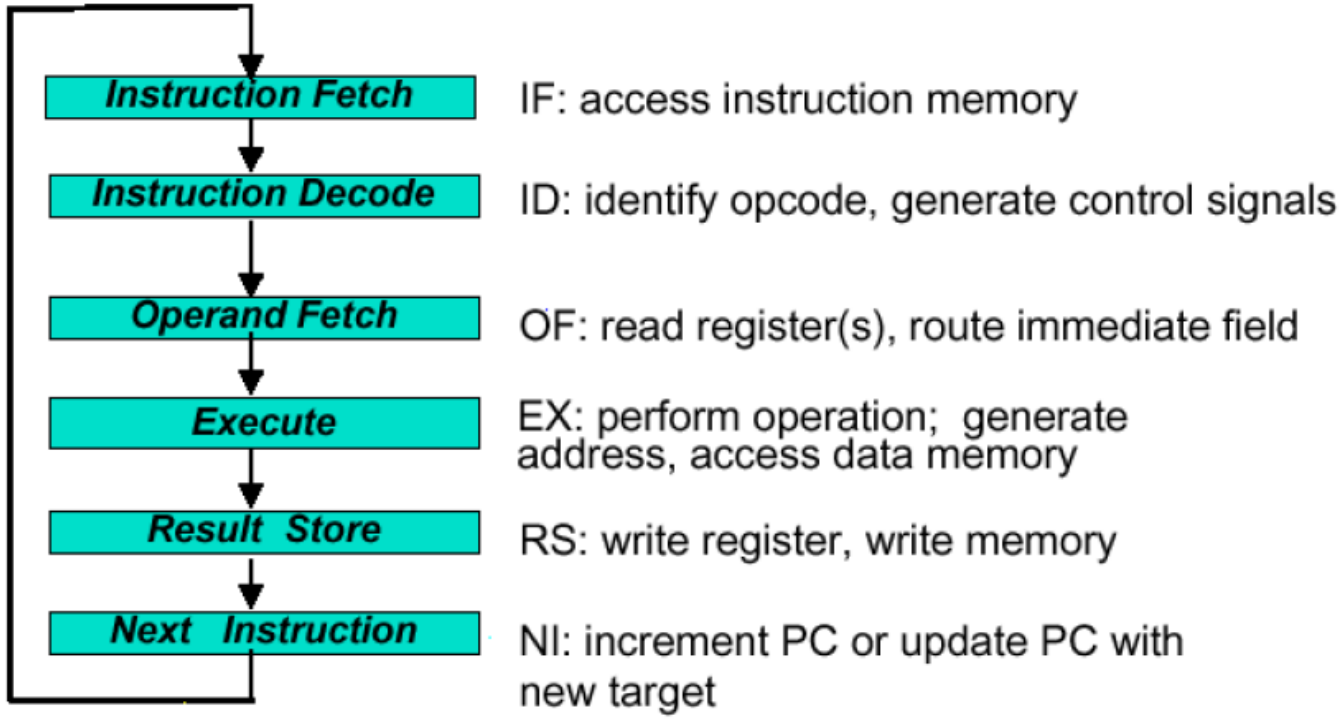
Single-cycle implementation

- Why a single-cycle implementation isn't used today?



  - Long cycle time for each instruction (load takes longest time)
  - All instructions take as much time as the slowest one

Typical Instruction Execution



Note that each step does not necessarily correspond to a clock cycle. These only describe the basic flow of instruction execution. The details vary with instruction type.

A multi-cycle Implementation

- Each step in the execution will take one clock cycle.
- Allow a function unit (e.g., ALU) to be used more than once per instruction, as long as it is used on different clock cycles.
- Advantage:
  - Allow instructions to take different numbers of clock cycles.
  - Share function units within the execution of a single instruction.
- (v) The difference between single-cycle & multi-cycle implementation:
  - A single memory unit is used for both instructions and data.
  - A register is used to save the instruction after it is read from memory. – It is called ***"Instruction Register (IR)"***.
  - A single ALU is used, rather than an ALU + two adders.

Multi-cycle Datapath

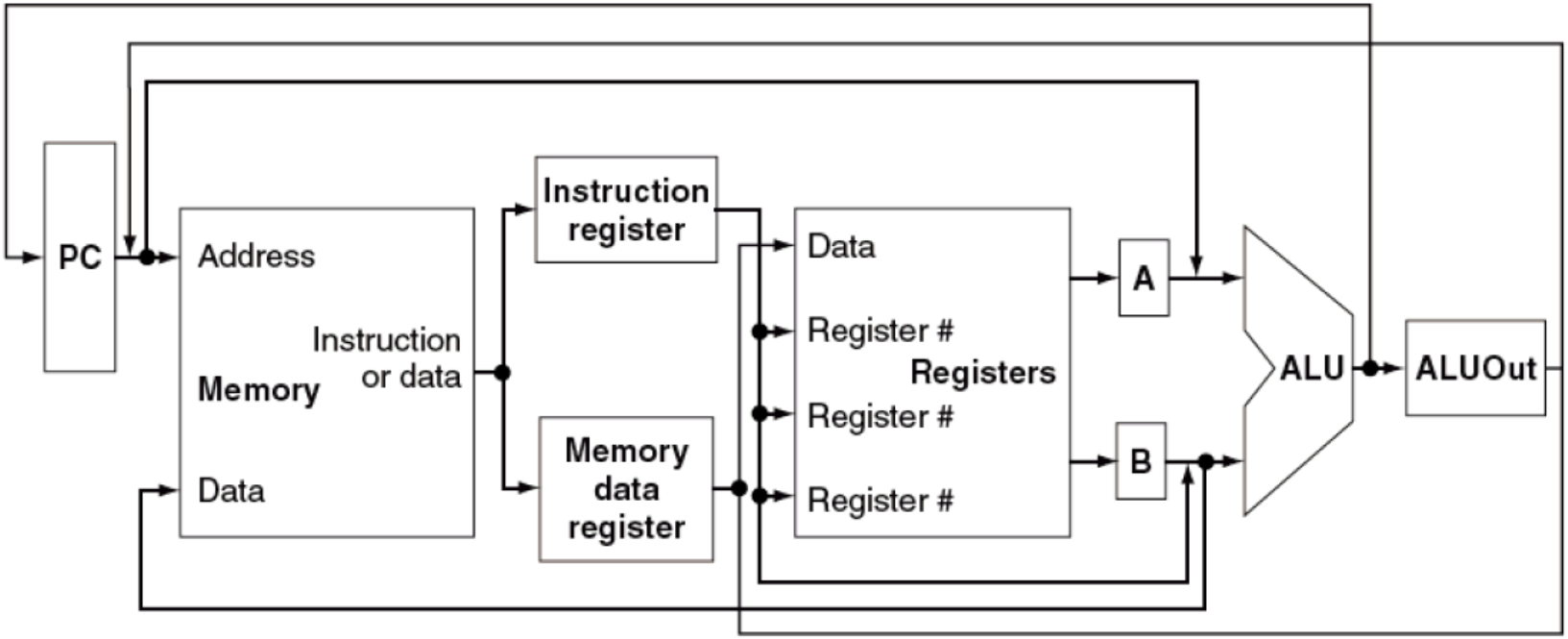


FIGURE 5.25 The high-level view of the multicycle datapath. This picture shows the key elements of the datapath: a shared memory unit, a single ALU shared among instructions, and the connections among these shared units. The use of shared functional units requires the addition or widening of multiplexors as well as new temporary registers that hold data between clock cycles of the same instruction. The additional registers are the Instruction register (IR), the Memory data register (MDR), A, B, and ALUOut.

Added Temporary Registers

- The ***Instruction Register (IR)*** and the ***Memory Data Register (MDR)*** are added to save the output of memory for an instruction read and a data read, respectively.
- Two separate registers are used, since both values are needed during the same clock cycle (the **IR** needs to hold the instruction until the end of execution of that instruction, and thus will require a write control signal)

- The ***A*** and ***B registers*** are used to hold the register operand values read from the register file.
- The **ALUOut register** holds the output of the ALU.

A multi-cycle Implementation

- Two sources for a memory address: a MUX to select
  - The PC (for instruction access)
  - ALUOut (calcuated address for data access, ***lw, sw***)
- A single ALU must accommodate all the inputs

- Two required changes to the datapath:
  - **An additional multiplexor**: choose between the A register and the PC.
  - **A four-way multiplexor**:
    - the B register
    - the constant 4
    - the sign-extended field
    - the sign-extended and shifted offset field (2bits)

Multi-cycle Datapath

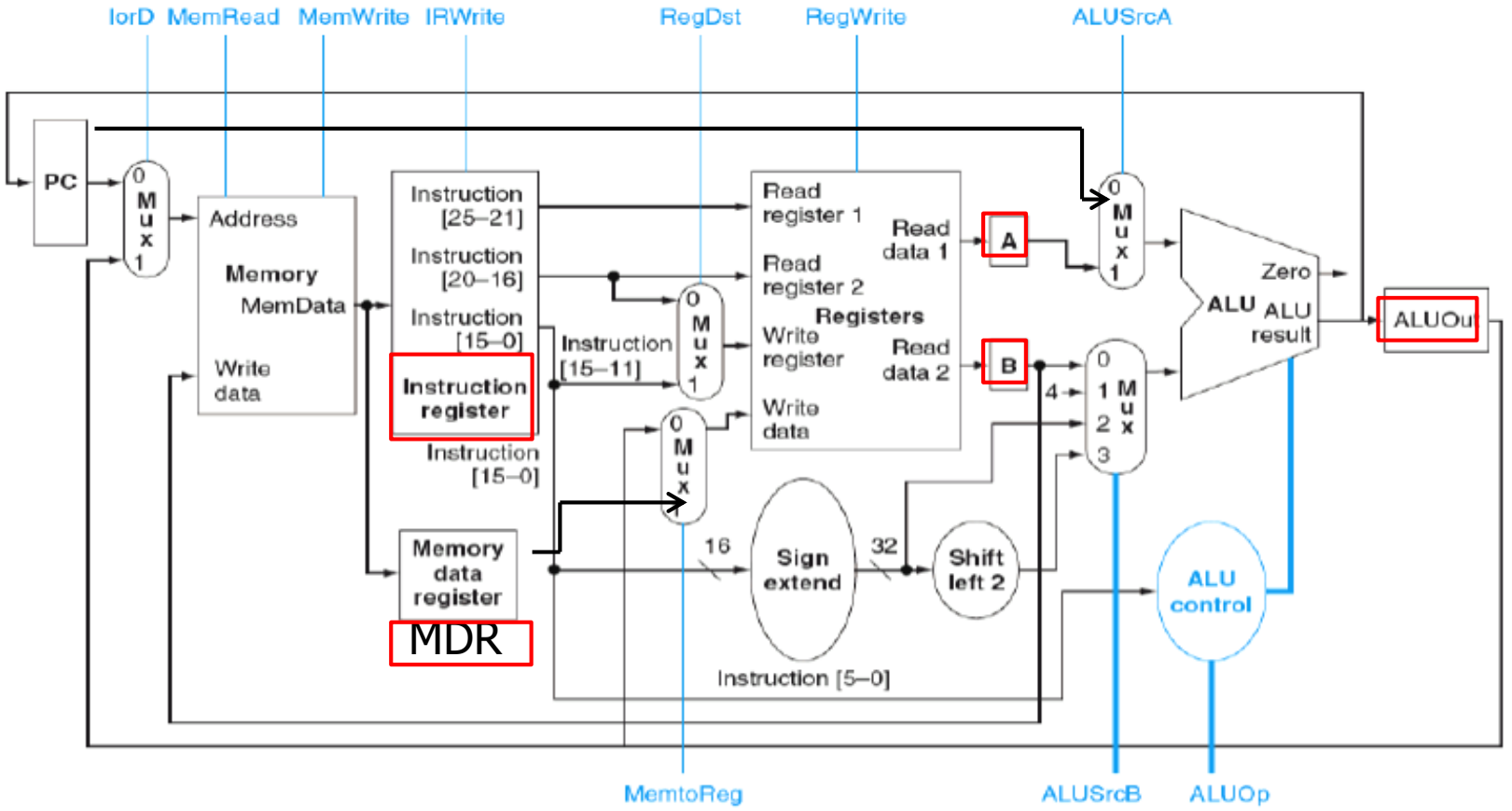


FIGURE 5.27 The multicycle datapath from Figure 5.26 with the control lines shown. The signals ALUOp and ALUSrcB are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register A nor B requires a write signal, since their contents are only read on the cycle immediately after it is written. The memory data register has been added to hold the data from a load when the data returns from memory. Data from a load returning from memory cannot be written directly into the register file since the clock cycle cannot accommodate the time required for both the memory access and the register file write. The MemRead signal has been moved to the top of the memory unit to simplify the figures. The full set of datapaths and control lines for branches will be added shortly.

Multi-cycle Datapath + Control signals

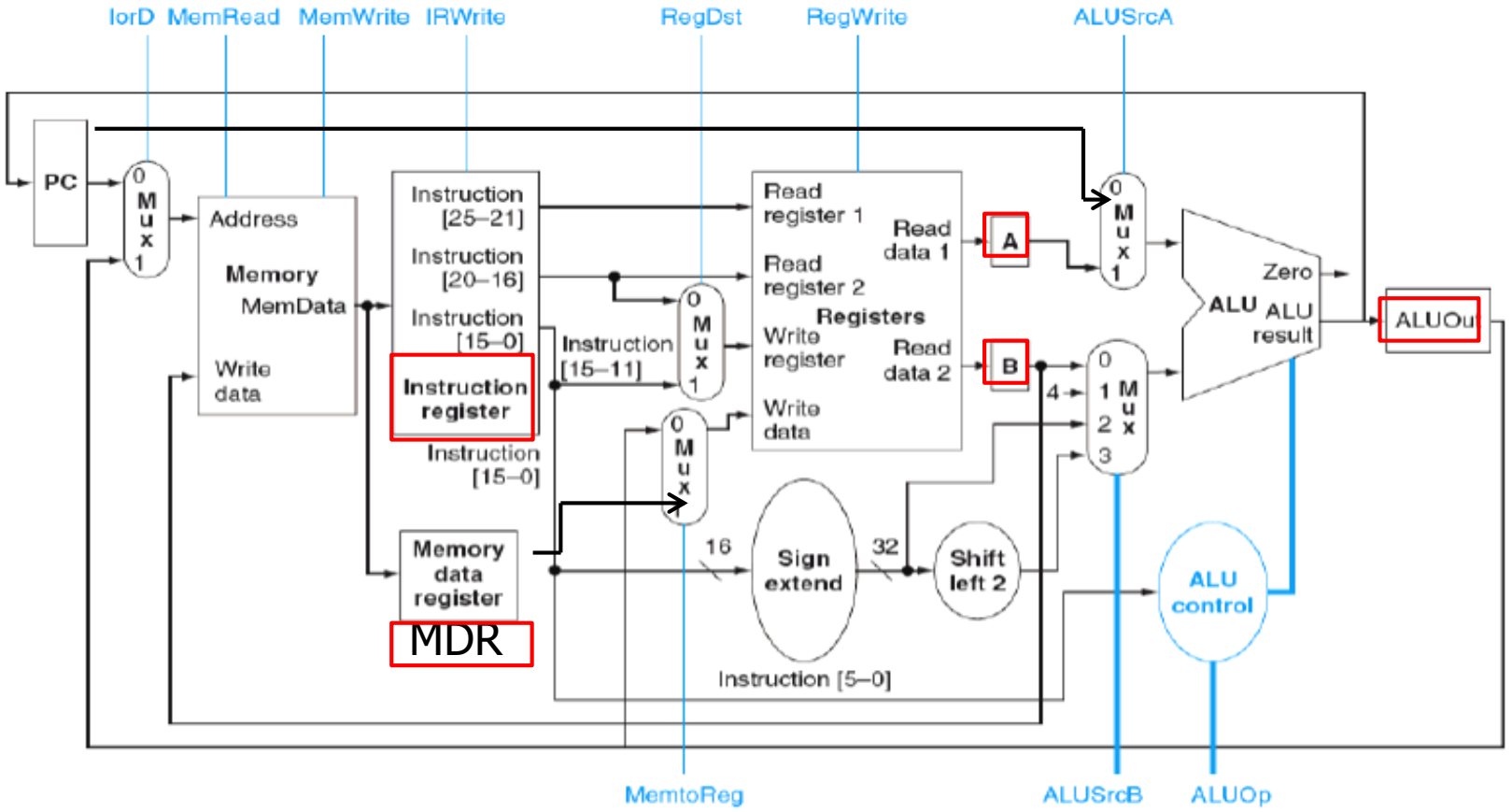


FIGURE 5.27 The multicycle datapath from Figure 5.26 with the control lines shown. The signals ALUOp and ALUSrcB are 2-bit control signals, while all the other control lines are 1-bit signals. Neither register A nor B requires a write signal, since their contents are only read on the cycle immediately after it is written. The memory data register has been added to hold the data from a load when the data returns from memory. Data from a load returning from memory cannot be written directly into the register file since the clock cycle cannot accommodate the time required for both the memory access and the register file write. The MemRead signal has been moved to the top of the memory unit to simplify the figures. The full set of datapaths and control lines for branches will be added shortly.

Program Counter Control

- With the ***jump*** and ***branch*** instruction, there are 3 possible value to be written into the PC:
  - *Normal:* The output of the ALU: PC+4, which should be stored directly into the PC
  - *The register ALUOut:* the address of the ***branch target address***
  - *When the instruction is a **jump**:* The lower 26 bits of the IR shifted left by 2 and concatenated with the upper 4 bits of the incremented PC.
- PCWrite: causes an unconditional write of the PC
- PCWriteCond: causes a write of the PC if the branch condition is also true

Complete Datapath & Controls

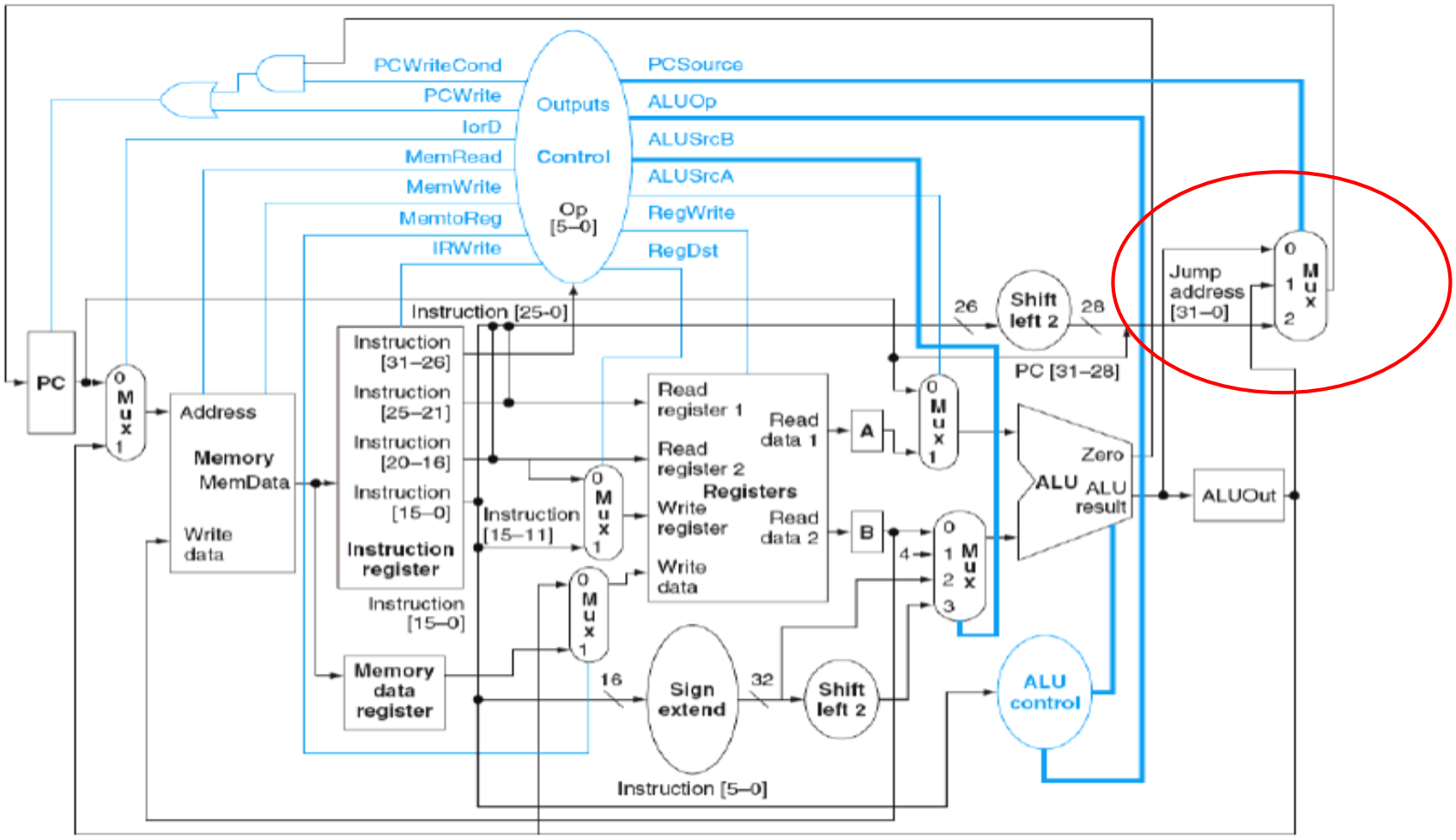


FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure 5.27 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 5.27 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken. Support for jumps is included.

Breaking the Instruction Execution into Clock Cycles

- The limitation of ***one ALU operation***, ***one memory access***, and ***one register file access*** determines what can fit in one step
- Breaking the Instruction Execution into Clock Cycles
  1. **Instruction fetch (IF)** step
  2. **Instruction decode (ID)** and register fetch step
  3. **Execute (EX),** memory address computation, or branch completion
  4. **Memory access (MEM)** or **R-type** instruction completion step
  5. Memory read completion step **(Write back, WB)**
- Each MIPS instruction needs 3 ~ 5 of these steps.

Instruction Fetch (IF) and Decode (ID)

1. Instruction fetch (IF) step
   - IR ← Memory [PC];
   - PC ← PC + 4;

2. Instruction decode (ID) and register fetch step
   - A ← Reg [IR [25:21]]; # get ***rs***
   - B ← Reg [IR [20:16]]; # get ***rt***
   - ALUOut ← PC + (sign-extend (IR[15:0]) << 2)

     # **pre-compute** branch target address

Execution (EX) cycle

3. Execute, memory address computation, or branch completion

- Memory reference:
  ALUOut ← **A** + sign-extend (IR[15:0]);
- R-type:
  ALUOut ← **A** op **B**;
- Branch:
  if (A==B) PC ← **ALUOut**; # use pre-computed PC
- Jump:
  PC ← {PC[31:28], (IR[25:0], 2'b00)};
  #{x, y} is the Verilog notation for **concatenation** of bit fields x and y

Instruction Completion Steps

4. Memory access or R-type instruction completion step

   - Memory reference:

     MDR ← Memory [ALUOut]; # for lw

     or

     Memory [ALUOut] ← B; # for sw
   - R-type:

     Reg [IR [15:11]] ← ALUOut; # completion of R-type

5. Memory read completion step

   Load: Reg[IR[20:16]] ← MDR; # for lw

Review: Complete Datapath & Controls

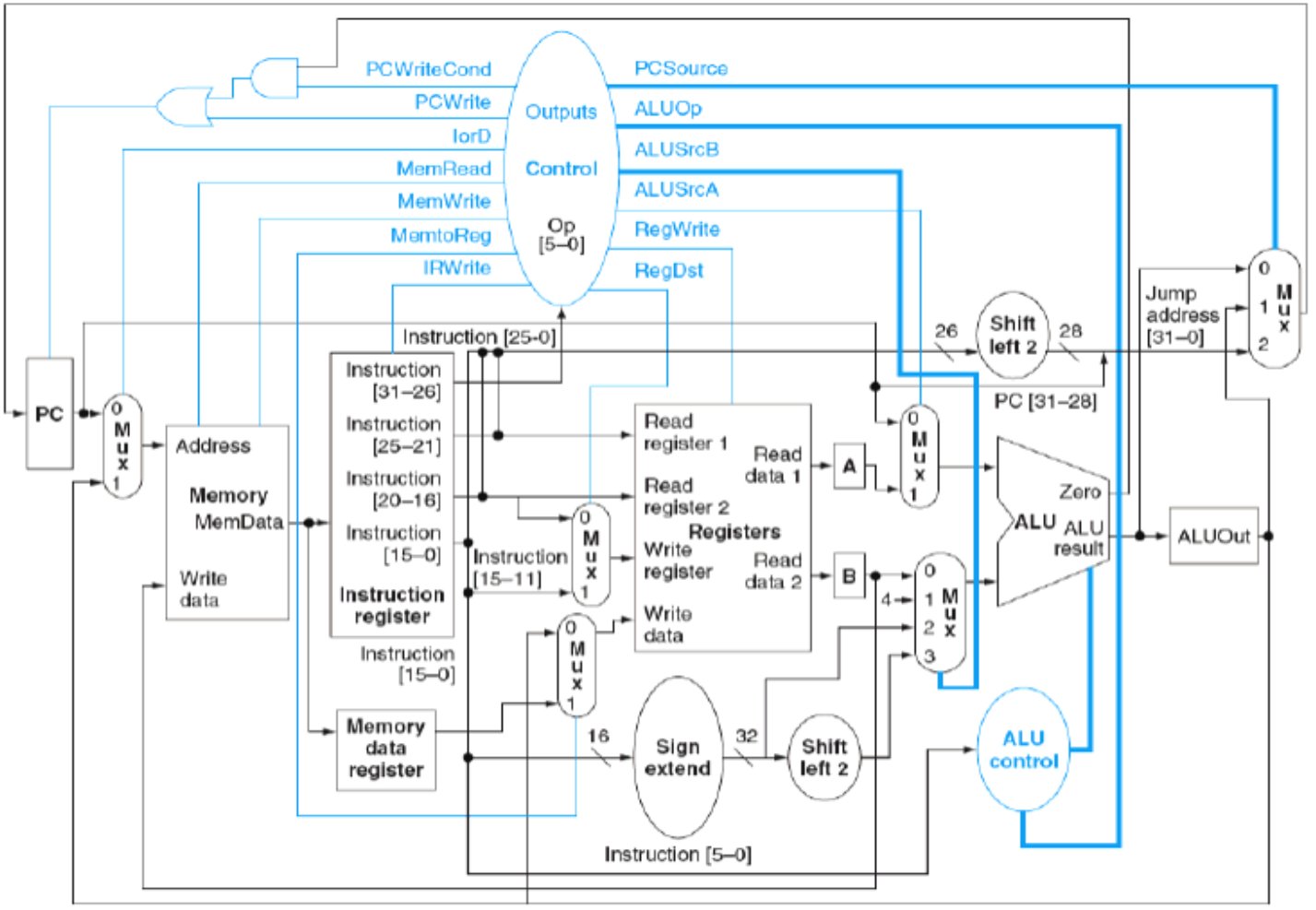


FIGURE 5.28 The complete datapath for the multicycle implementation together with the necessary control lines. The control lines of Figure 5.27 are attached to the control unit, and the control and datapath elements needed to effect changes to the PC are included. The major additions from Figure 5.27 include the multiplexor used to select the source of a new PC value; gates used to combine the PC write signals; and the control signals PCSource, PCWrite, and PCWriteCond. The PCWriteCond signal is used to decide whether a conditional branch should be taken. Support for jumps is included.

Actions of the control signals

Actions of the 1-bit control signals

Signal name	Effect when deasserted	Effect when asserted
RegDst	The register file destination number for the Write register comes from the rt field.	The register file destination number for the Write register comes from the rd field.
RegWrite	None.	The general-purpose register selected by the Write register number is written with the value of the Write data input.
ALUSrcA	The first ALU operand is the PC.	The first ALU operand comes from the A register.
MemRead	None.	Content of memory at the location specified by the Address input is put on Memory data output.
MemWrite	None.	Memory contents at the location specified by the Address input is replaced by value on Write data input.
MemtoReg	The value fed to the register file Write data input comes from ALUOut.	The value fed to the register file Write data input comes from the MDR.
IorD	The PC is used to supply the address to the memory unit.	ALUOut is used to supply the address to the memory unit.
IRWrite	None.	The output of the memory is written into the IR.
PCWrite	None.	The PC is written; the source is controlled by PCSource.
PCWriteCond	None.	The PC is written if the Zero output from the ALU is also active.

Priority: PCWrite > PCWriteCond

Actions of the 2-bit control signals

Actions of the 2-bit control signals

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

FIGURE 5.29 The action caused by the setting of each control signal in Figure 5.28 on page 323. The top table describes the 1-bit control signals, while the bottom table describes the 2-bit signals. Only those control lines that affect multiplexors have an action when they are deasserted. This information is similar to that in Figure 5.16 on page 306 for the single-cycle datapath, but adds several new control lines (IRWrite, PCWrite, PCWriteCond, ALUSrcB, and PCSource) and removes control lines that are no longer used or have been replaced (PCSrc, Branch, and Jump).

A multi-cycle Implementation

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

FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction class. The empty entries for the Memory access step or the Memory read completion step indicate that the particular instruction class takes fewer cycles. In a multicycle implementation, a new instruction will be started as soon as the current instruction completes, so these cycles are not idle or wasted. As mentioned earlier, the register file actually reads every cycle, but as long as the IR does not change, the values read from the register file are identical. In particular, the value read into register B during the Instruction decode stage, for a branch or R-type instruction, is the same as the value stored into B during the Execution stage and then used in the Memory access stage for a store word instruction.

CPI in a Multi-cycle CPU:

- CPI in the multi-cycle CPU:
  - 25% loads (1% load byte + 24% load word)
  - 10% stores (1% store byte + 9% store word)
  - 11% branches (6% beq, 5% bne)
  - 2% jumps (1% jal + 1% jr)
  - 52% ALU (all the rest)

- Loads: 5 (clock cycles)
- Stores: 4
- ALU instructions: 4
- Branches: 3
- Jumps: 3

- CPI = 0.25*5 + 0.10*4 + 0.52*4 + 0.11*3 + 0.02*3 = 4.12

- This CPI is better than the worst-case CPI of 5.0 when all the instructions take the same number of clock cycles.

Finite-state Machine Control

The high-level view of the finite state machine control

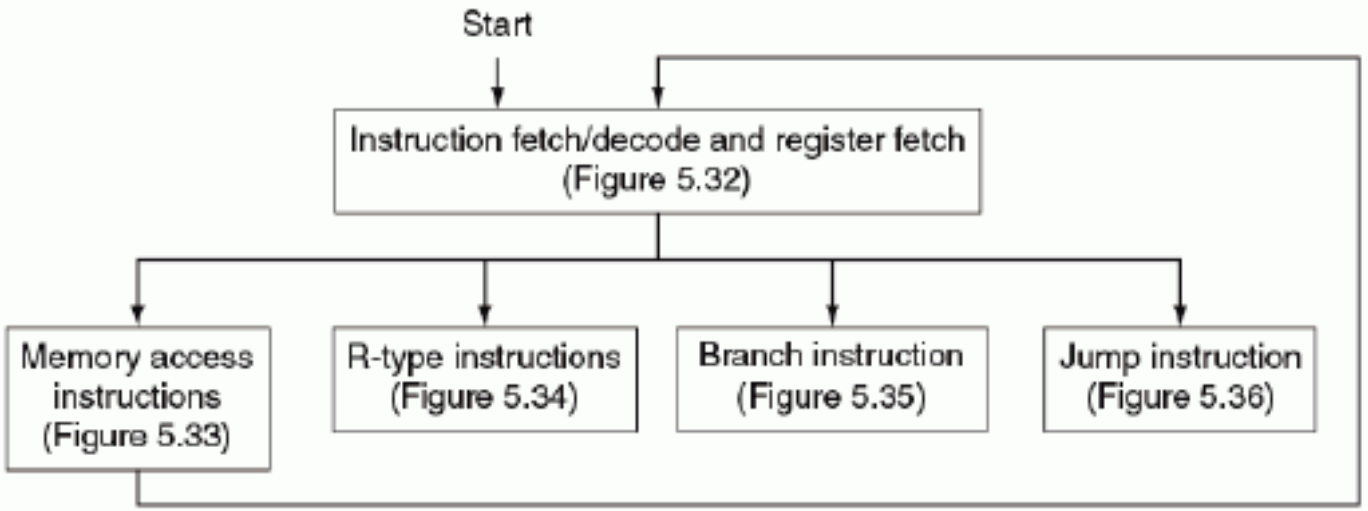
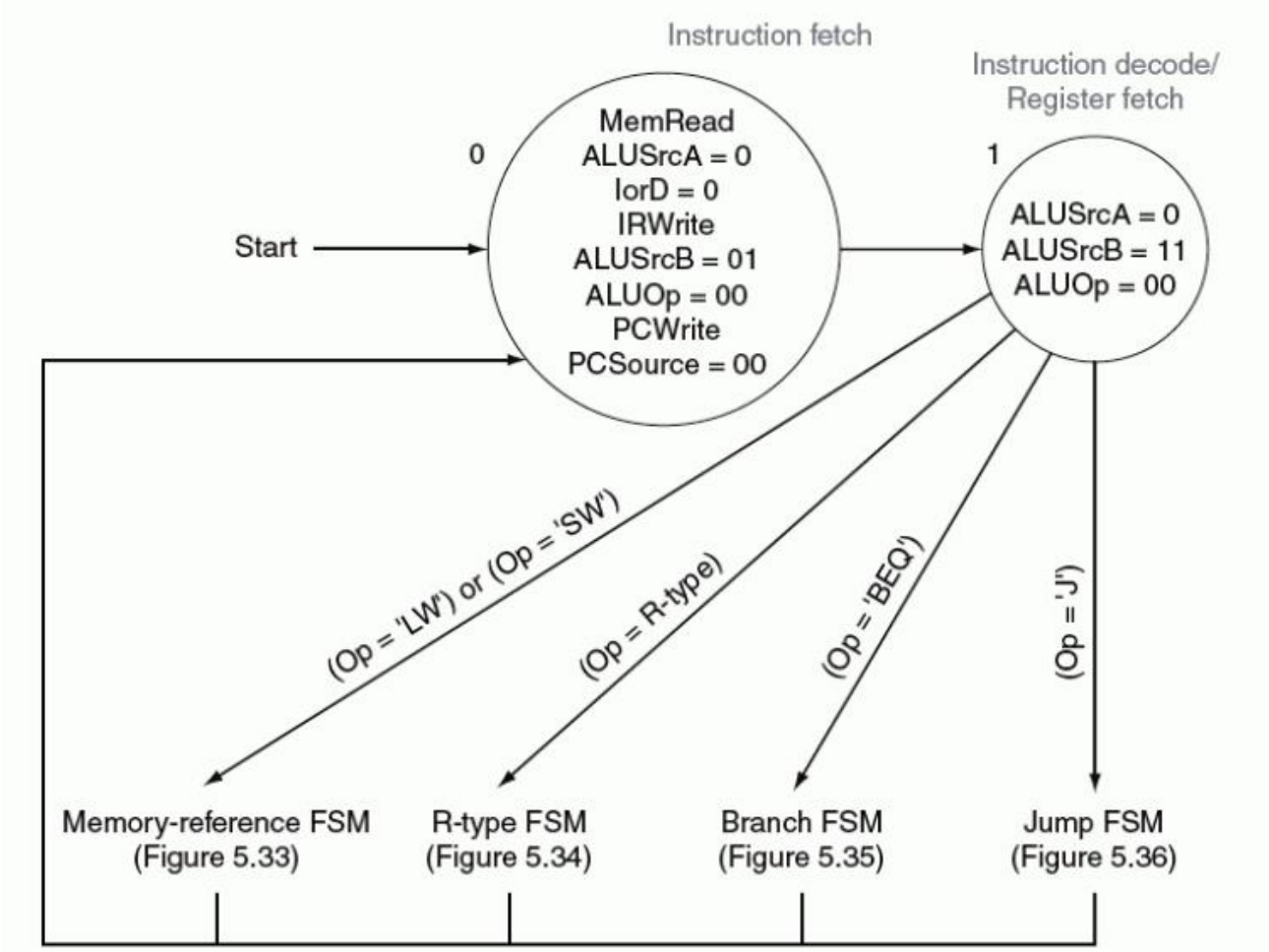
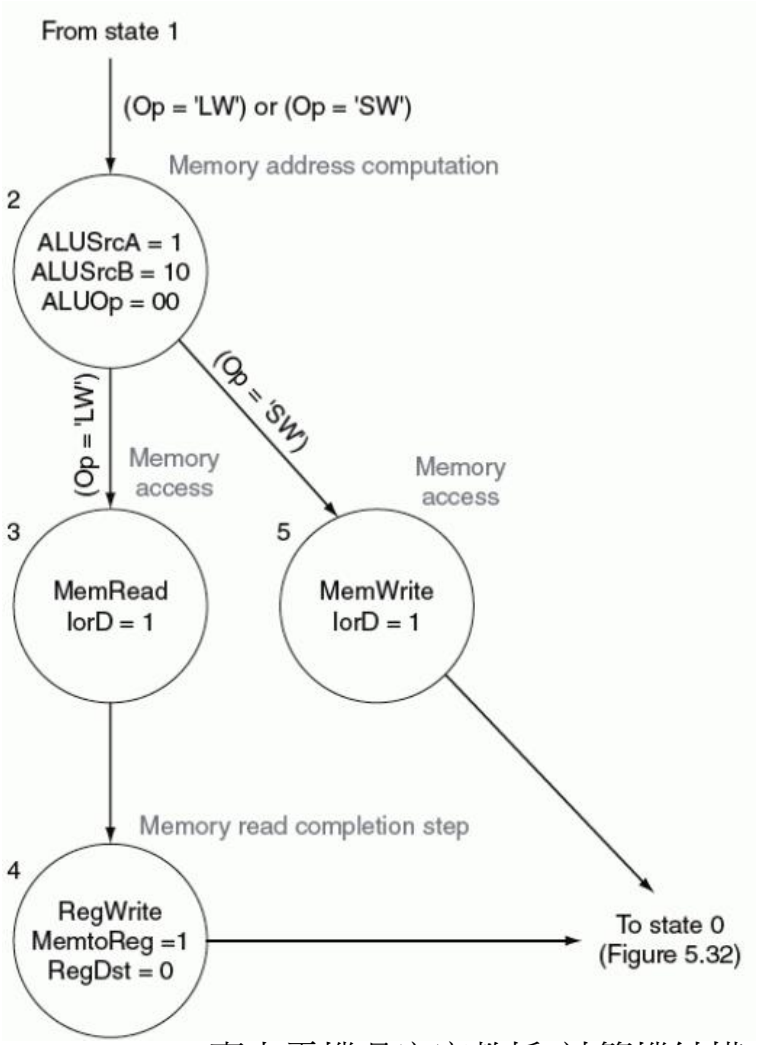


FIGURE 5.31 The high-level view of the finite state machine control. The first steps are independent of the instruction class; then a series of sequences that depend on the instruction opcode are used to complete each instruction class. After completing the actions needed for that instruction class, the control returns to fetch a new instruction. Each box in this figure may represent one to several states. The arc labeled *Start* marks the state in which to begin when the first instruction is to be fetched.

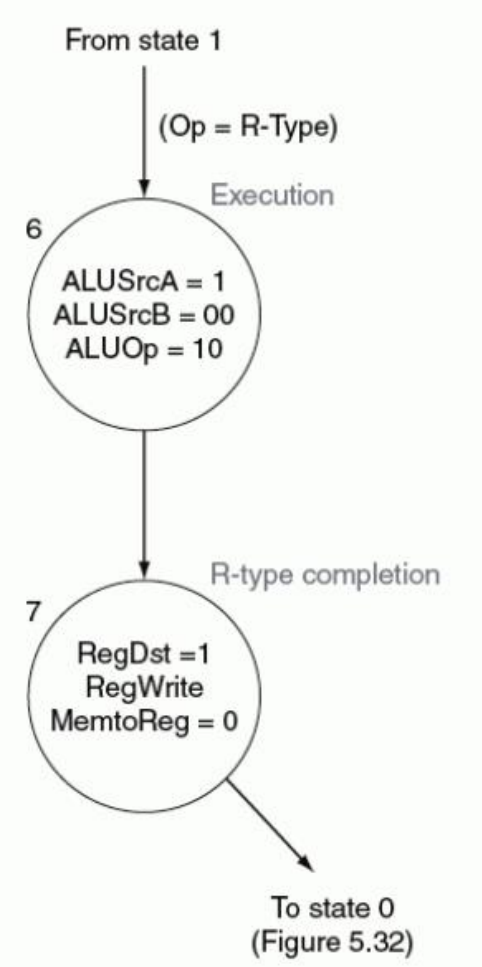
1. Instruction Fetch and Decode (Fig. 5.32)



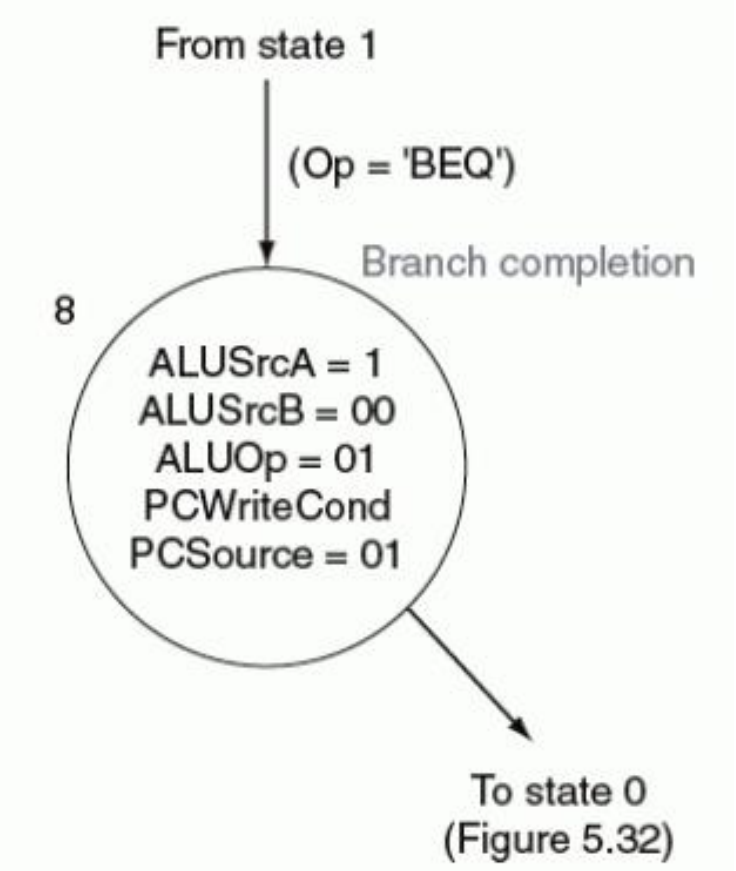
2. Memory-reference instructions (Fig. 5.33)



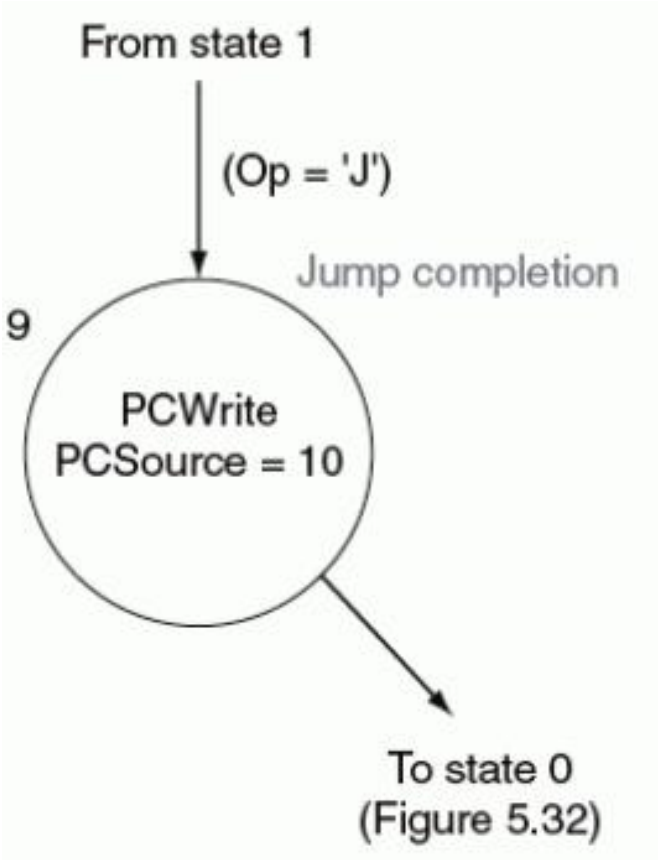
3. R-type (Fig. 5.34)



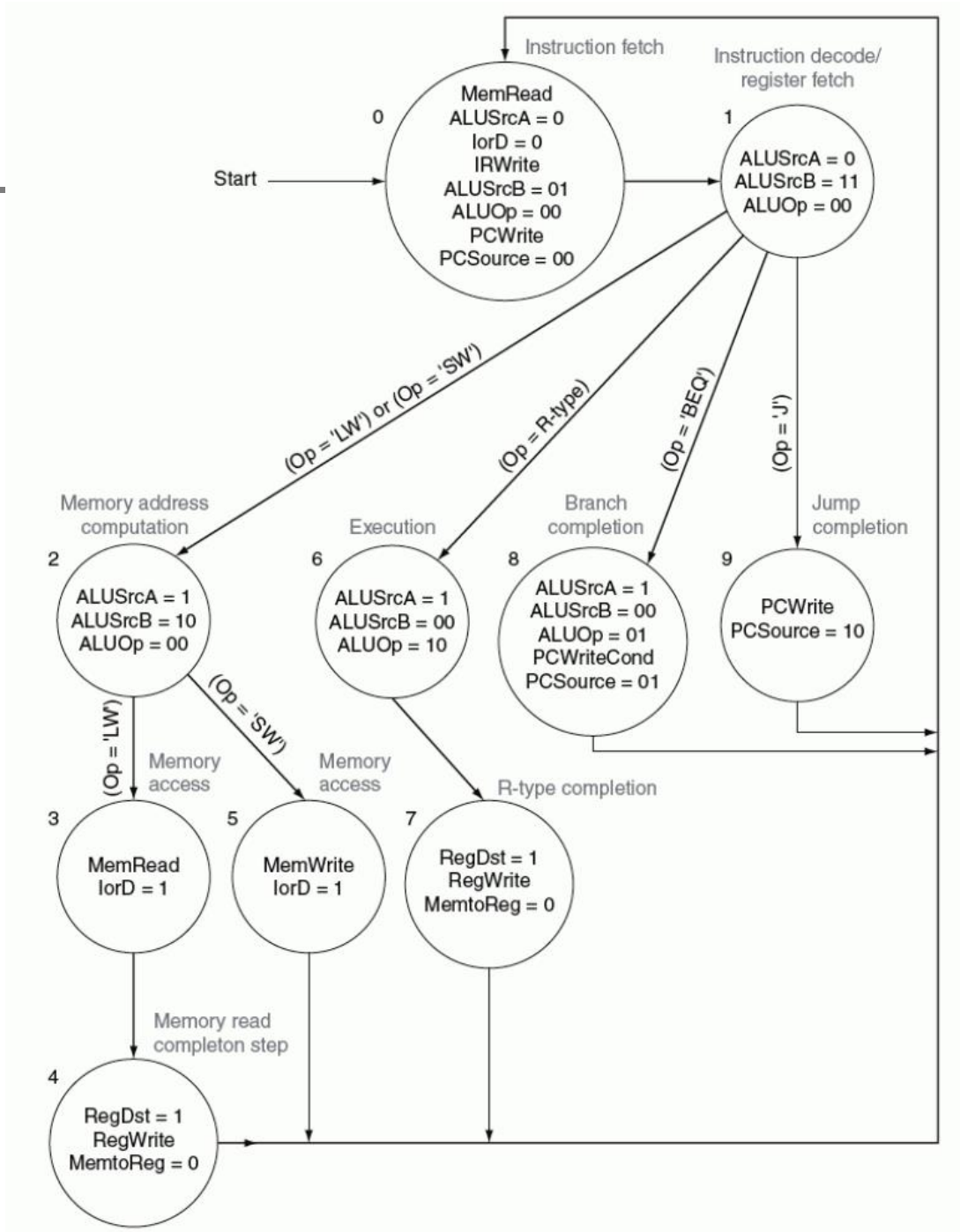
4. Branch (Fig. 5.35)



5. Jump (Fig. 5.36)



Complete State diagram for Control Signals



Techniques to Specify the Control

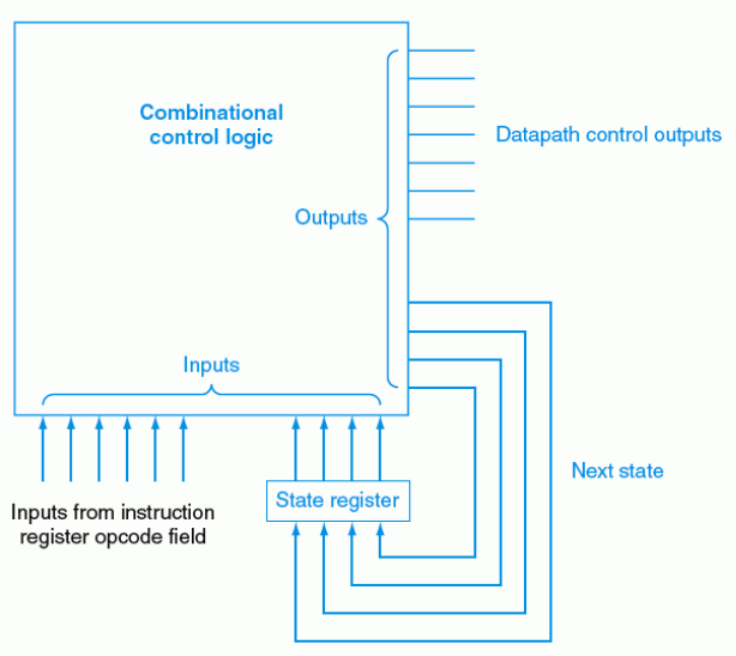
- Two different techniques to specify the control:
  1. Finite state machine (state diagram)
  2. Microprogramming (see Appendix CDROM)

Microprogram :

- A symbolic representation of control in the form of instructions, called **microinstructions**, that are executed on a simple micromachine.
- Store all control signals in a ROM (in binary format)
- Use uPC (micro program counter) to address the desired signals.

Implementation of State Diagram

A. Conventional way to implement the Control Unit



B. Use **Verilog/VHDL** to implement the State Diagram (most popular nowadays).

Interrupt and Exception (Chap.4.9)

- Interrupts were initially created to handle unexpected events like **arithmetic overflow** and to **signal requests for service from I/O devices.**
- Some events generated internally or externally:

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Either	Exception or interrupt

Interrupt and Exception (Chap.4.9)

- ***Exception:*** any unexpected change in control flow without distinguishing whether the cause is internal or external
- ***Interrupt:*** only when the event is externally caused
- We will only discuss how to handle ***an undefined instruction*** or an ***arithmetic overflow*** in this chapter.

- How exceptions are handled:
  - Save the address of the offending instruction in the ***Exception Program Counter (EPC)*** and transfer control to the operating system at some specified address.
  - Take some predefined action in response to an overflow, or stop the execution of the program and report an error (Execute ***Interrupt Service Routine, ISR***)
  - Terminate the program or may continue its execution, using the EPC to determine where to restart the execution of the program.

Interrupt Registers

Two main methods used to communicate the reason for an exception:

1. **Cause register: A status register** which holds a field that indicates the reason for the exception (used in MIPS architecture)
2. **Vectored interrupt:** An interrupt for which the address to which control is transferred is determined by the cause of the exception.

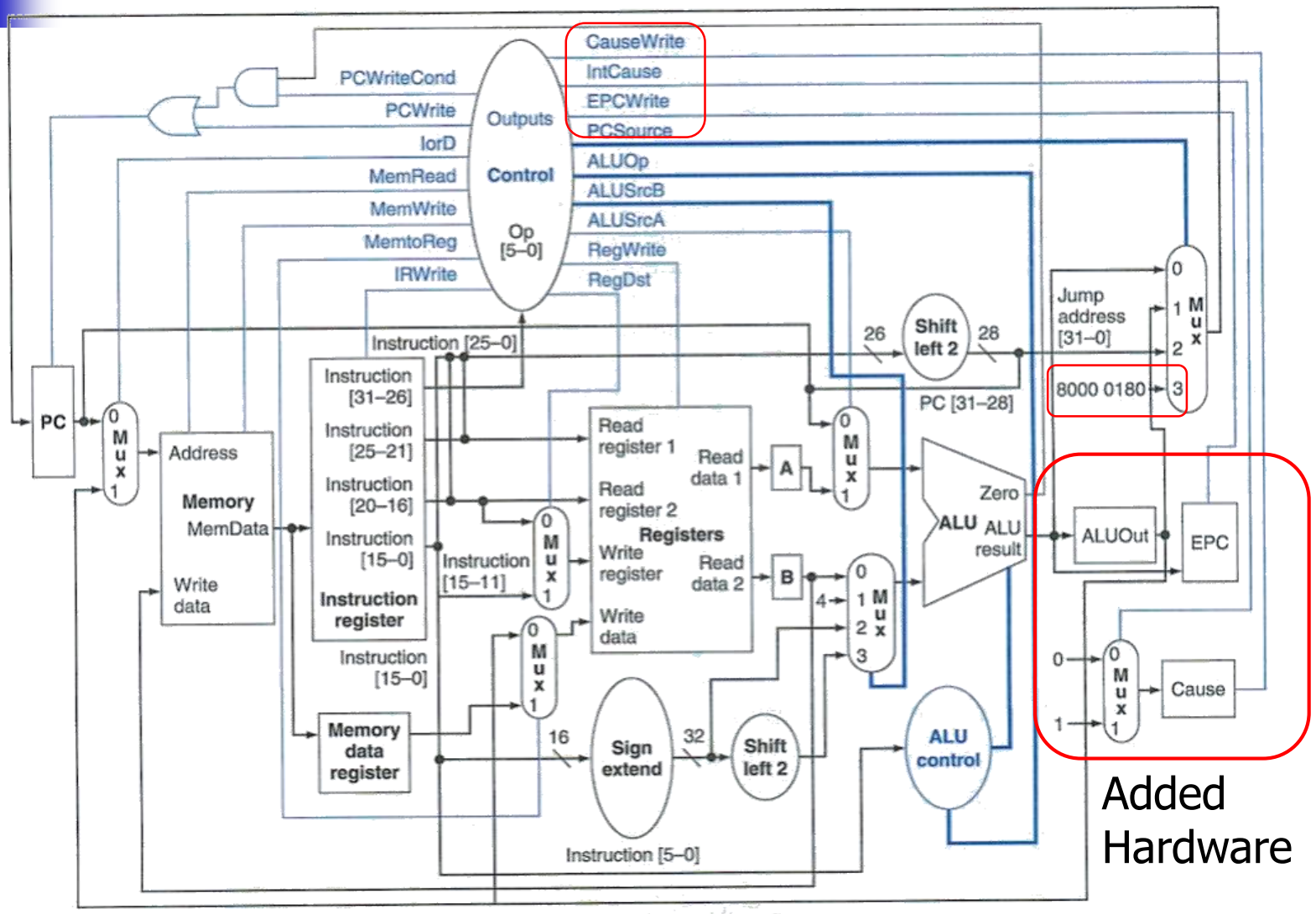
Exception type	Exception vector address (in hex)
Undefined instruction	8000 0000_{hex}
Arithmetic overflow	8000 0180_{hex}

- The operating system knows the reason for the exception by the address at which it is initiated.
- The address are separated by 32 bytes or 8 instructions, and the operating system must record the reason for the exception and may perform some limited processing in this sequence.

Support Exception in MIPS CPU

- For MIPS exception system
  - Two additional registers to the datapath:
    1. **EPC (exception program counter):** A 32-bit register used to hold the address of the affected instruction.
    2. **Cause:** A register used to record the cause of the exception. In the MIPS architecture, this register is 32 bits.
  - 3 Additional control signals:
    - *EPCWrite* (update the problem instruction address)
    - *CauseWrite* (update the Cause number)
    - *IntCause*
  - Change the 3-way multiplexor (controlled by PCSouse) to a 4-way multiplexor, with additional input wired to the constant value **$8000\ 0180_{hex}$** → **The Operating system entry point for exception handling subroutines**

Review of Multi-Cycle MIPS CPU



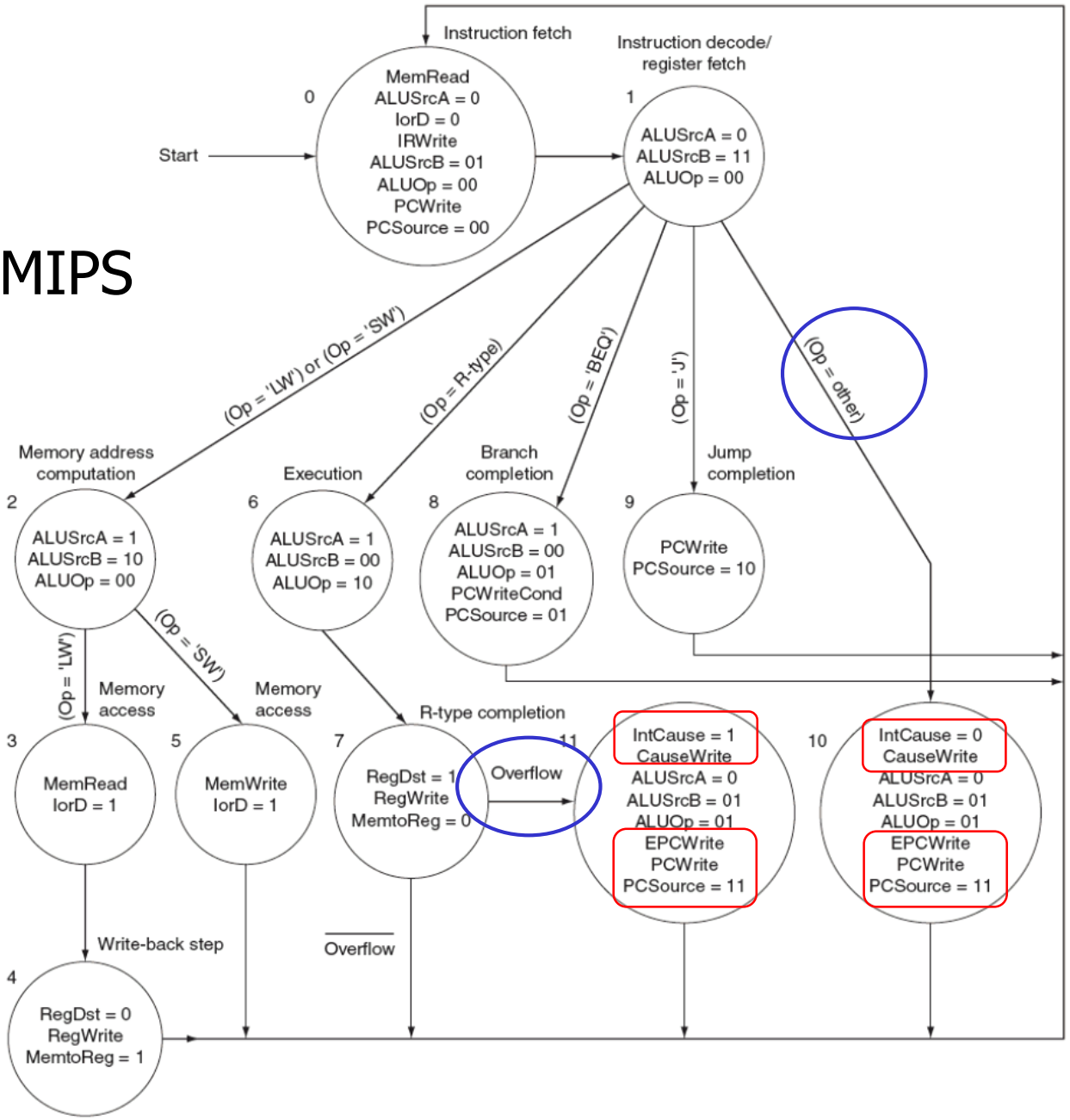
Handle Interrupt in MIPS control

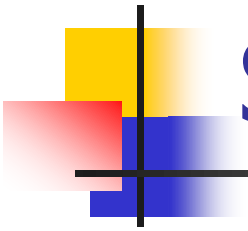
- Two new states (10 and 11) are added to Page 92.

1. Undefined instruction (10): This is detected when no next state is defined from state 1 for the op value.

2. Arithmetic overflow (11): The Overflow signal is used in the modified finite state machine to specify an additional possible next state(11) for state 7.

Complete State Diagram of the MIPS Controller (with Exception support)





Summary

- Single-cycle and multi-cycle RISC CPU designs are introduced.
- Adding new instructions is easy (*e.g.,* add **jr, jal, addi, subi, slt**) by adding/changing datapath units and control signals → final project.
- Good for complex Verilog programming (e.g., Digital System Design (DSD) Course)