Chapter5-1

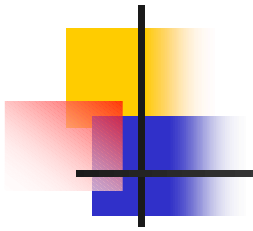
The Processor: Datapath and Control (Single-cycle implementation)

臺大電機系

吳安宇教授

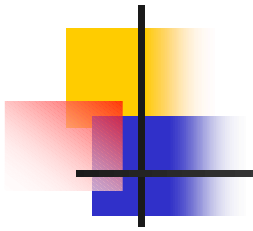
V1. 10/27/2004

V2. 11/03/2004



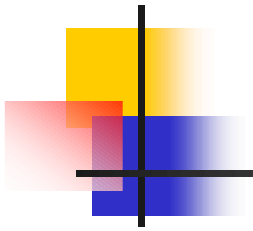
outline

- 5.1 Introduction
- 5.2 Logic Design Conventions
- 5.3 Building a Datapath
- 5.4 A Simple Implementation Scheme



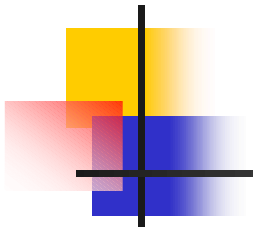
Introduction

- 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

- 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***



Introduction

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

  (1) Memory-reference instructions:
  use ALU to calculate “effective address”
  - (ex) lw $t0 offset($s5) -- compute ***offset + $s5***

  (2) Arithmetic-logical instructions:
  use ALU for opcode execution – ***add, sub, and, or***

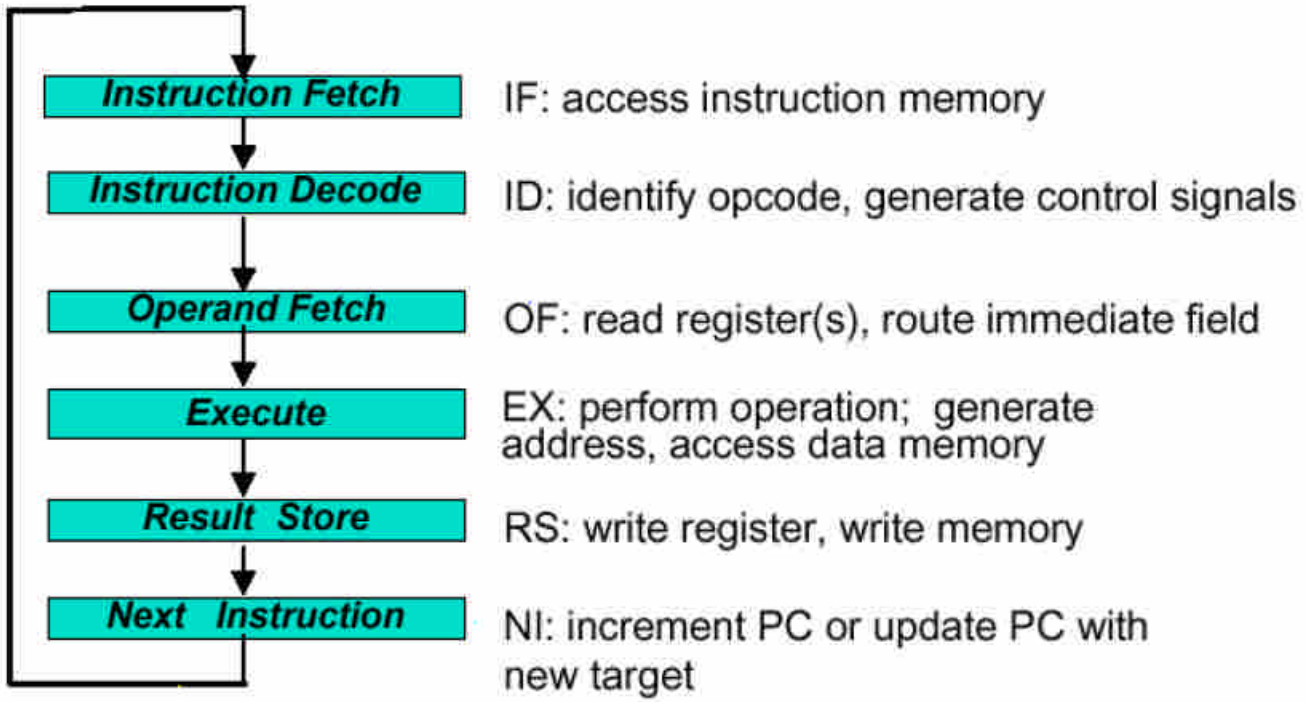
  (3) Branch instructions:
  use ALU for comparison – bne/slt $s1, $s2 ***($s1-$s2, and check sign of the results)***

Introduction

- 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 (change of PC)

Introduction

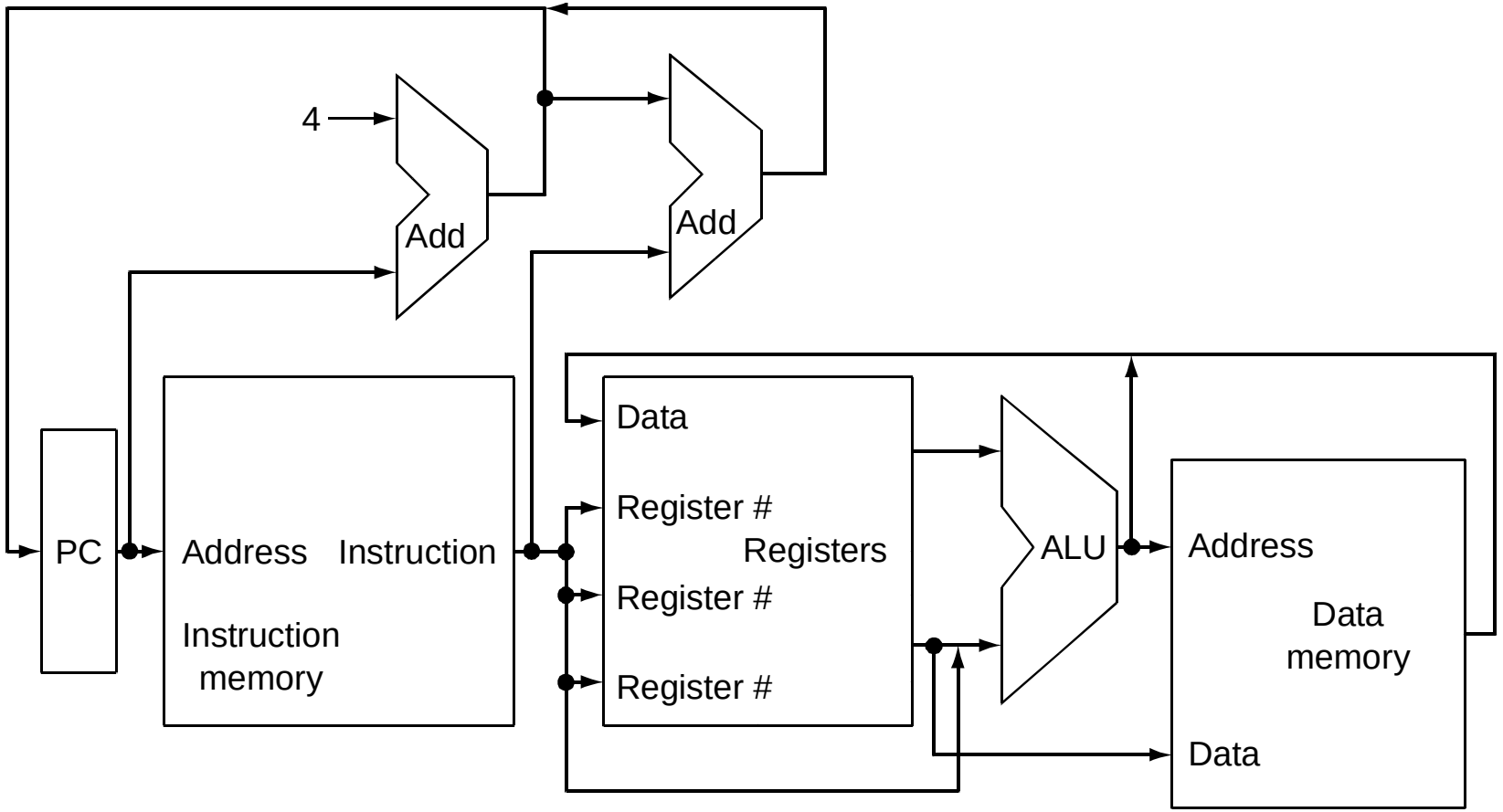
- Typical instruction execution

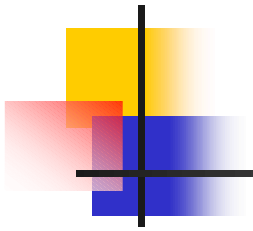


Note that each step does not necessarily correspond to a clock cycle. These only describe the basic flow of instruction execution. The details vary with instruction type.

Introduction

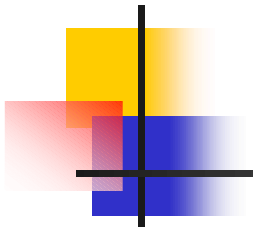
- An abstract view of the implementation of the MIPS subset showing the major functional units and the major connections between them.





outline

- 5.1 Introduction
- 5.2 Logic Design Conventions
- 5.3 Building a Datapath
- 5.4 A Simple Implementation Scheme

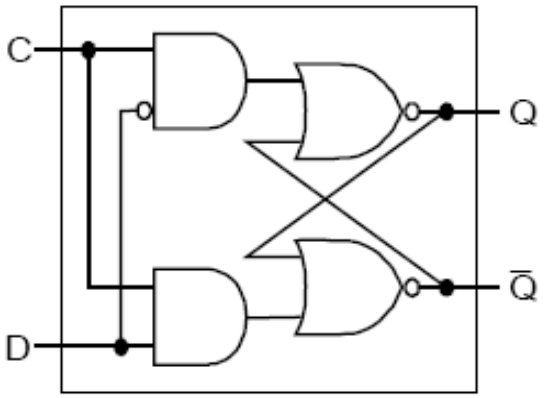


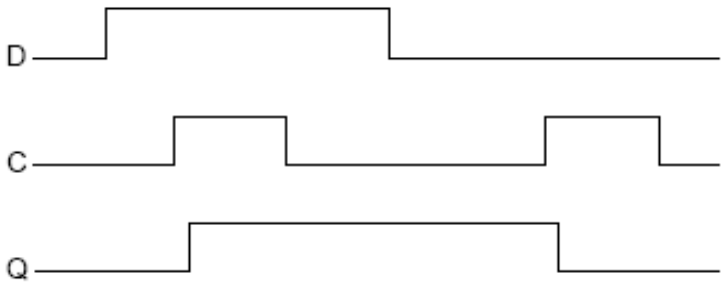
Logic Design Conventions

- Latch v.s. Flip-flop
  - Output is equal to the stored value inside the element. (don't need to ask for permission to look at the value)
  - Change of state (value) is based on the clock.
  - Latch: whenever the inputs change, and the clock is asserted.
  - Flip-flop: state changes only on a clock edge.

Logic Design Conventions

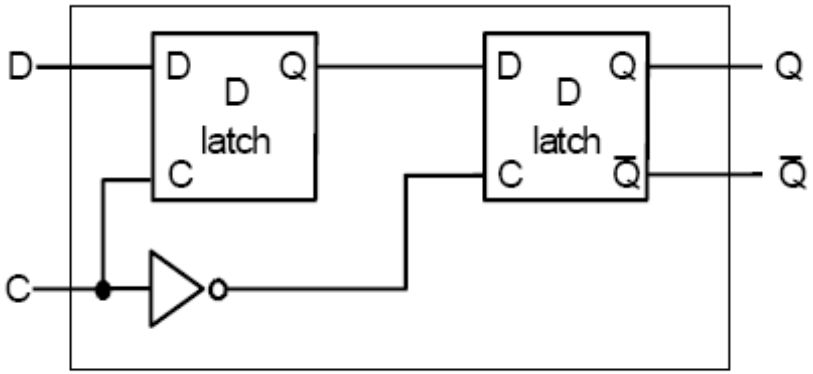
- D-latch
  - Two inputs:
    - the data value to be stored (D)
    - the clock signal (C) indicating when to read & store D
  - Two outputs:
    - the value of the internal state (Q) and it's complement

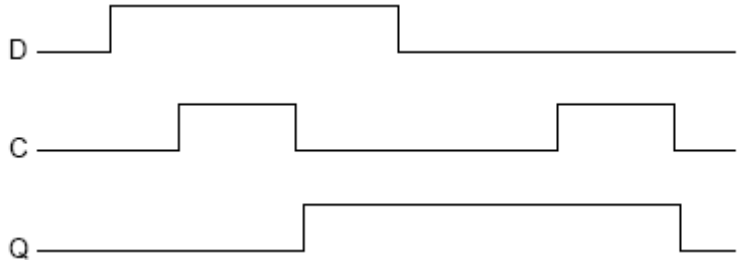




Logic Design Conventions

- D flip-flop (rising/falling edge)
  - Output changes only on the clock edge
  - (ex) the below is a D flip-flop with a falling-edge trigger





Logic Design Conventions

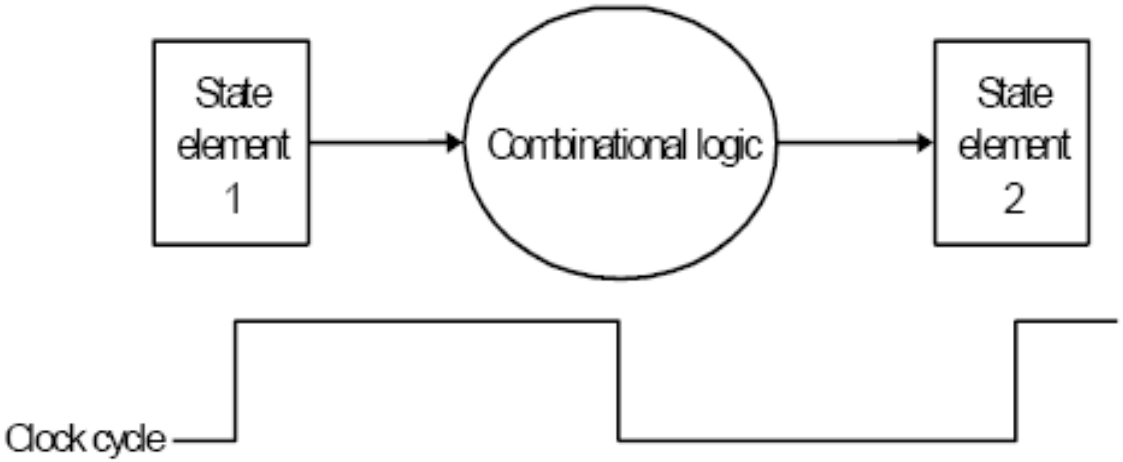
- The function units in the MIPS implementation consist of two different types of logic elements:
  - (1) combinational element :

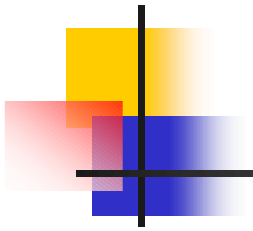
    The outputs depend only on the current inputs.
  - (2) state element :

    It contains state if it has some internal storage.
- Clocking methodology: defines when signals can be read and when they can be written.
- Control signal: used for multiplexer selection or for directing the operation of a functional unit; contrasts with a data signal, which contains information that us operated on by a functional unit.

Logic Design Conventions

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



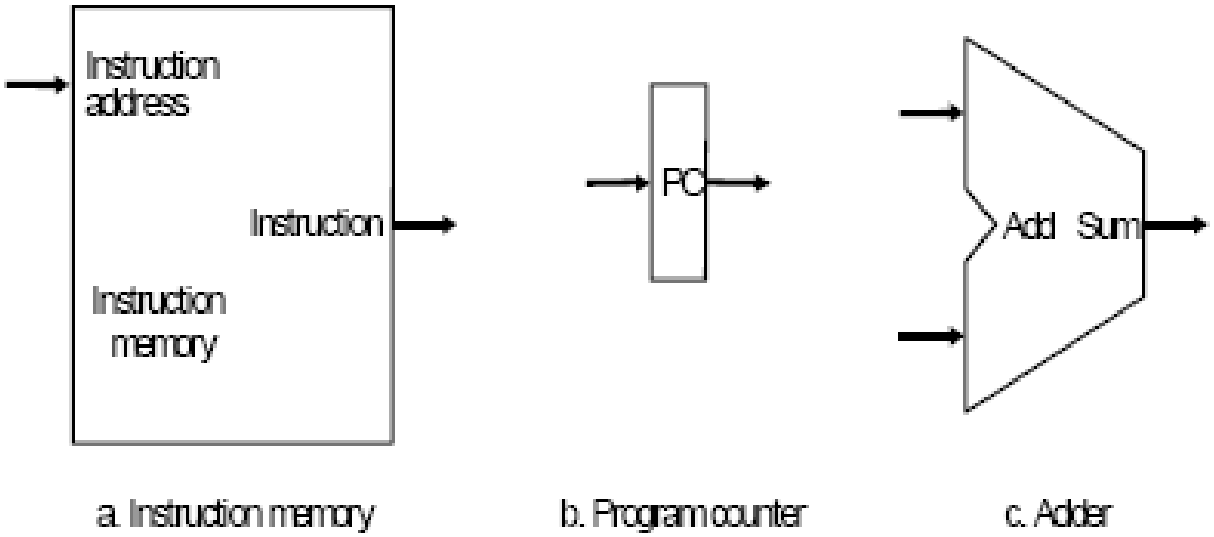


Outline

- 5.1 Introduction
- 5.2 Logic Design Conventions
- 5.3 Building a Datapath
- 5.4 A Simple Implementation Scheme

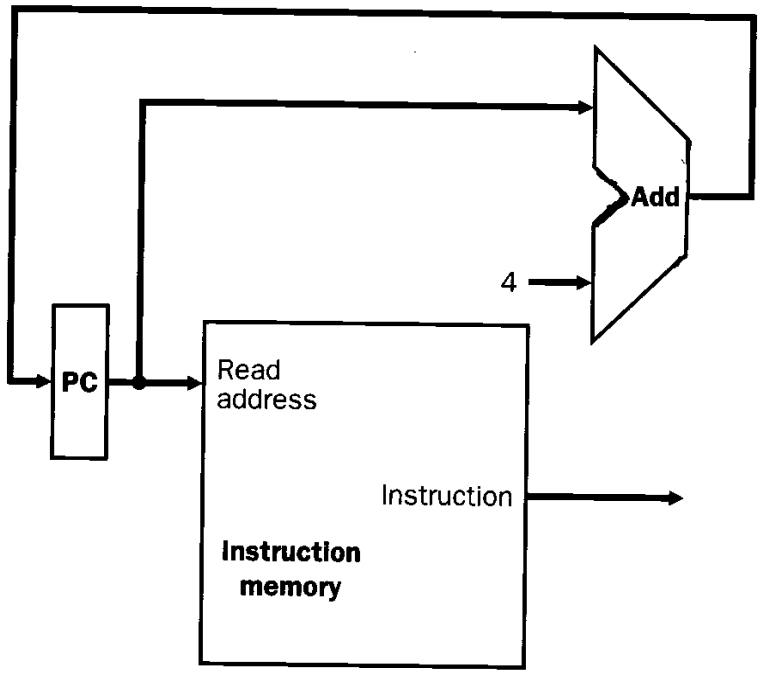
Building a Datapath

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

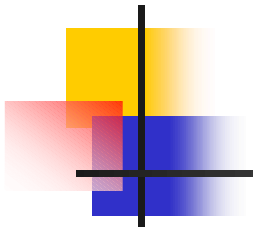


Building a Datapath

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



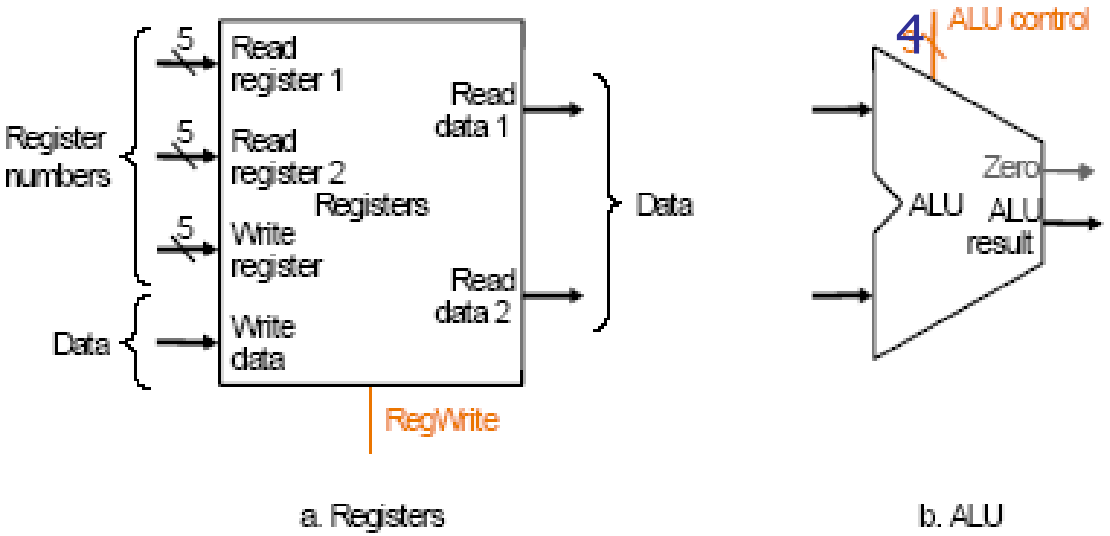
Building a Datapath



- Basic elements 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) an input to the “register file” to specify the index of the registers to be read.
    - (2) an output of the register contents.
  - Write operation:
    - (1) an input to the “register file” to specify the index of the registers to be written.
    - (2) an input to supply the data to be written into the specified register.

Building a Datapath

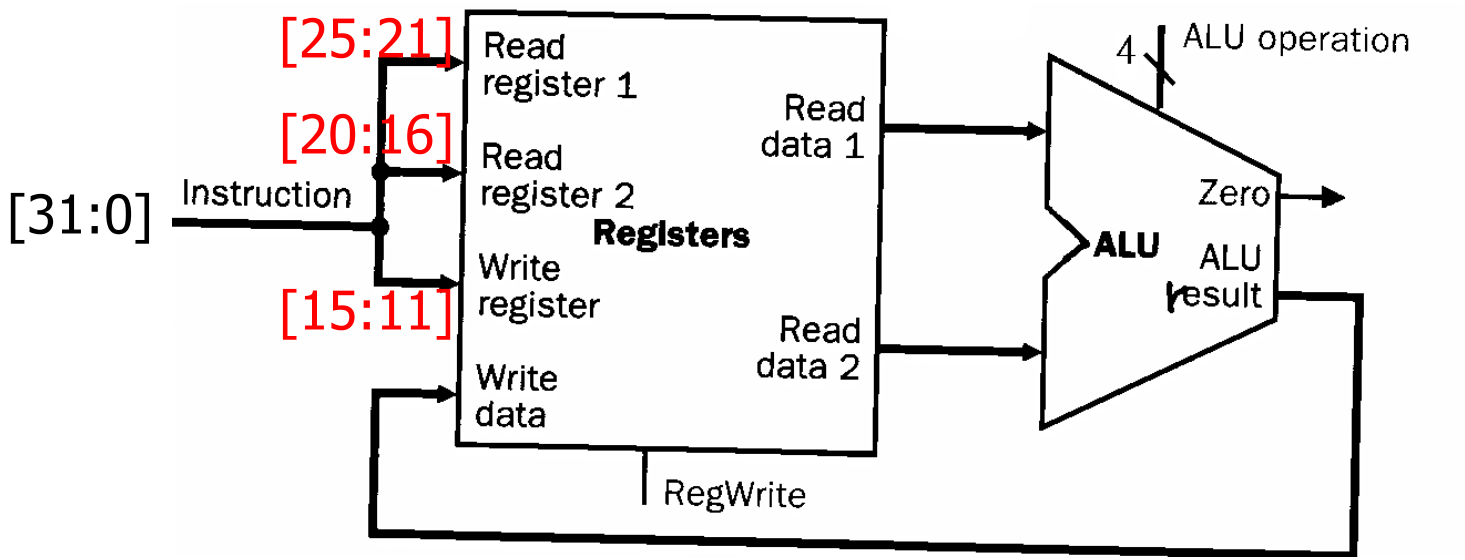
- Elements which we need:
  - (a) Register file: a collection of registers in which any register can be read or written by specifying the index of the register in the file.
  - (b) ALU (32 bits): operate on the values read from the registers.



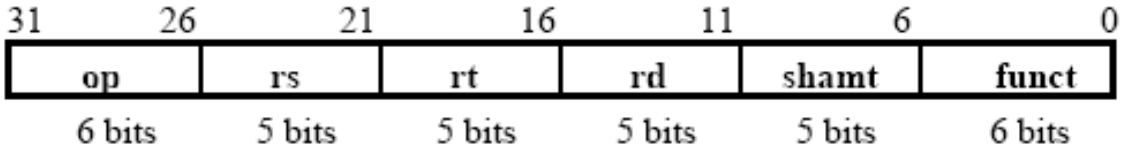
a. Registers

b. ALU

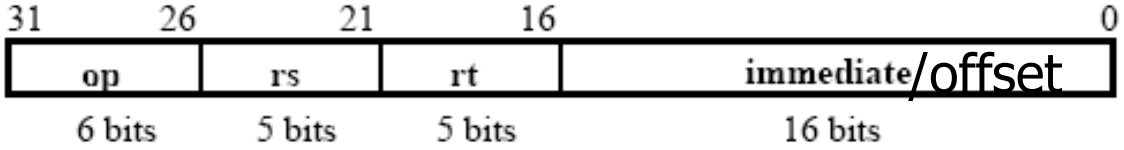
The datapath for R-type instructions



- **R-type:**

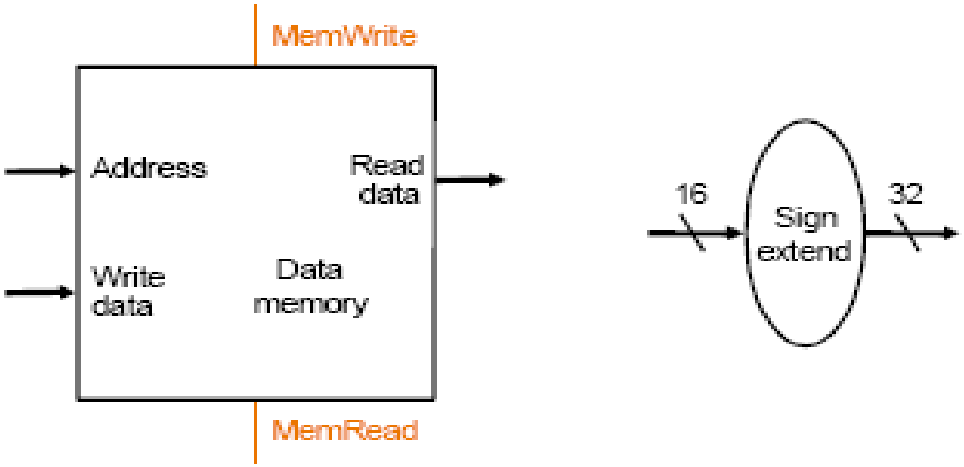


- **I-type:**



Building a Datapath

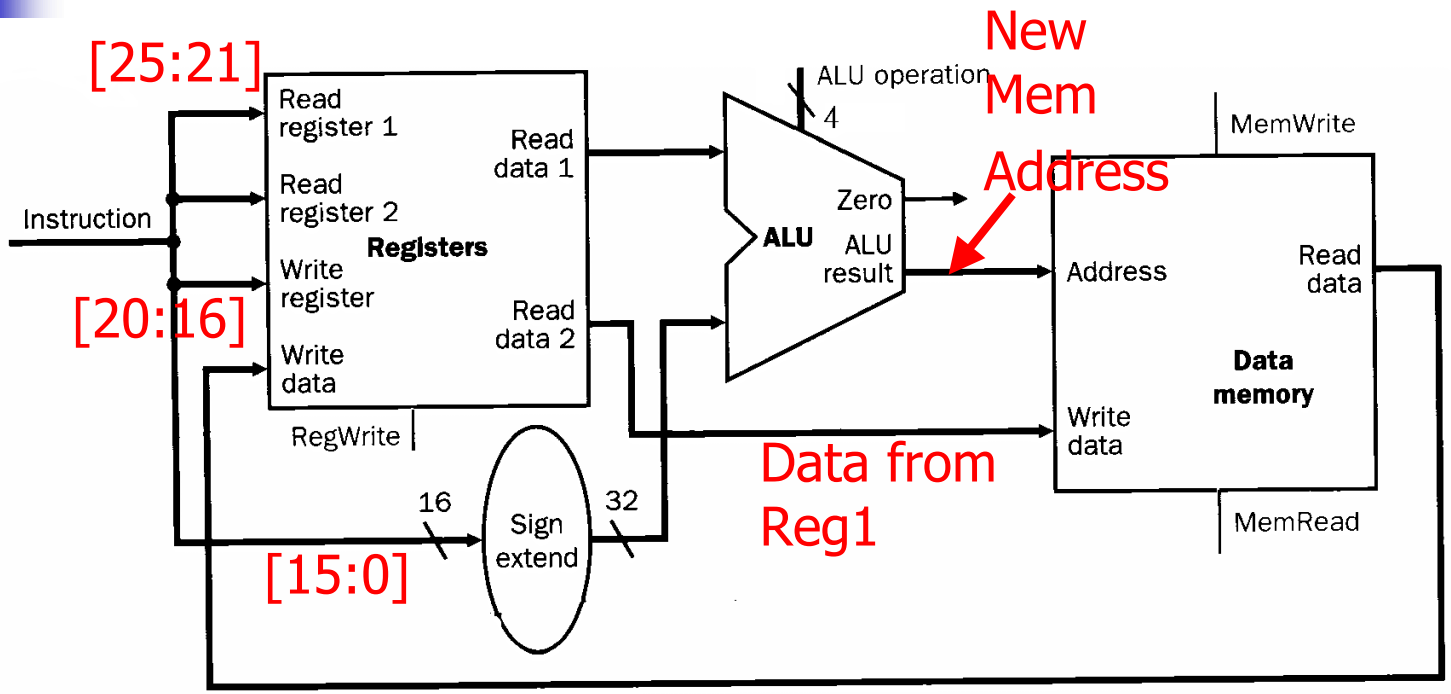
- Basic elements for load/store instructions:
  - (a) Data memory unit: read/write data
  - (b) Sign-extend unit: sign-extend the 16-bit offset field in the instruction to a 32-bit signed value.
  - (c) Register file
  - (d) ALU (add “reg” + “offset” to computer the mem address)



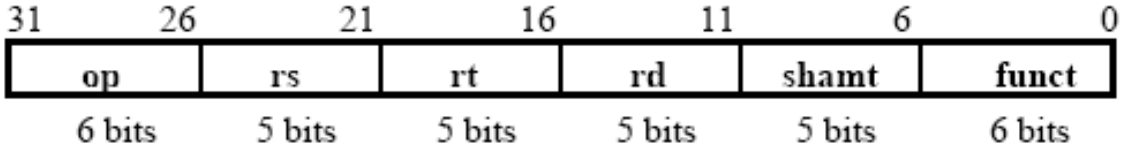
a. Data memory unit b. Sign-extension unit

-- (c) & (d) are just shown as the previous slide.

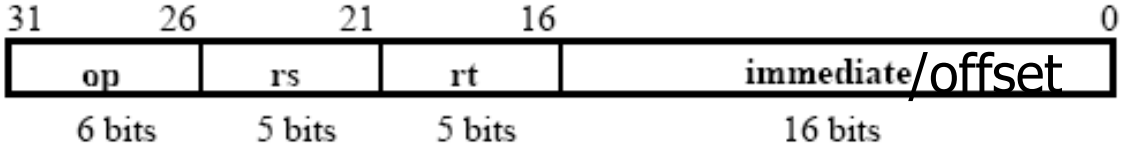
Datapath for load/store instructions



- **R-type:**



- **I-type:**



Branch Instructions

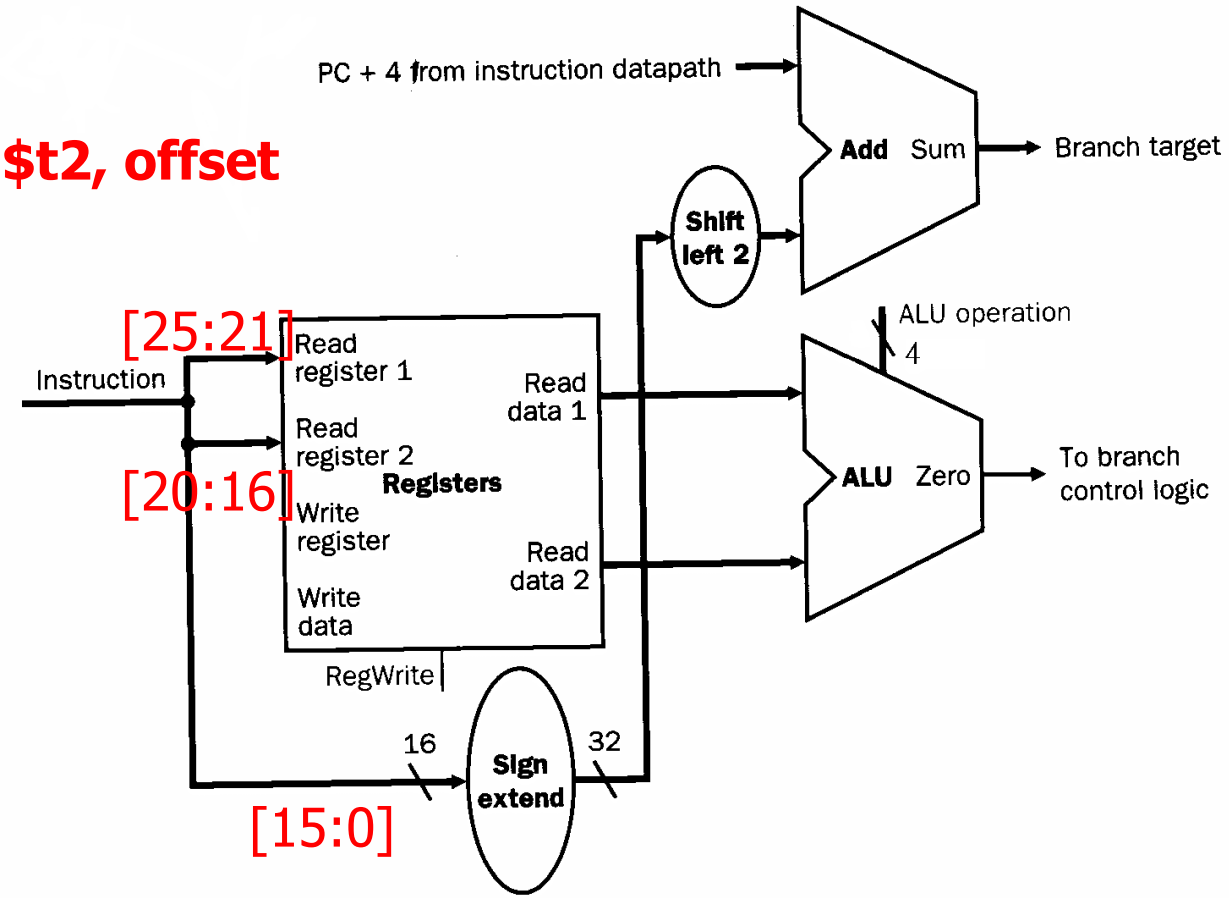
- (ex) beq $t1, $t2, offset

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

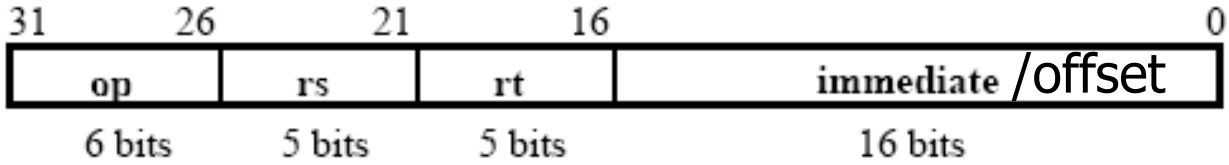
- Note:
  (1) The offset field is shifted left 2 bits so that it's a "word offset".
  (2) Branch is taken: when the condition is true, the branch target address becomes the new PC.
  (3) Branch isn't taken: the incremented PC (PC+4) replaces the current PC, just as for normal instruction.
- Operations: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

Datapath for “beq” Instructions

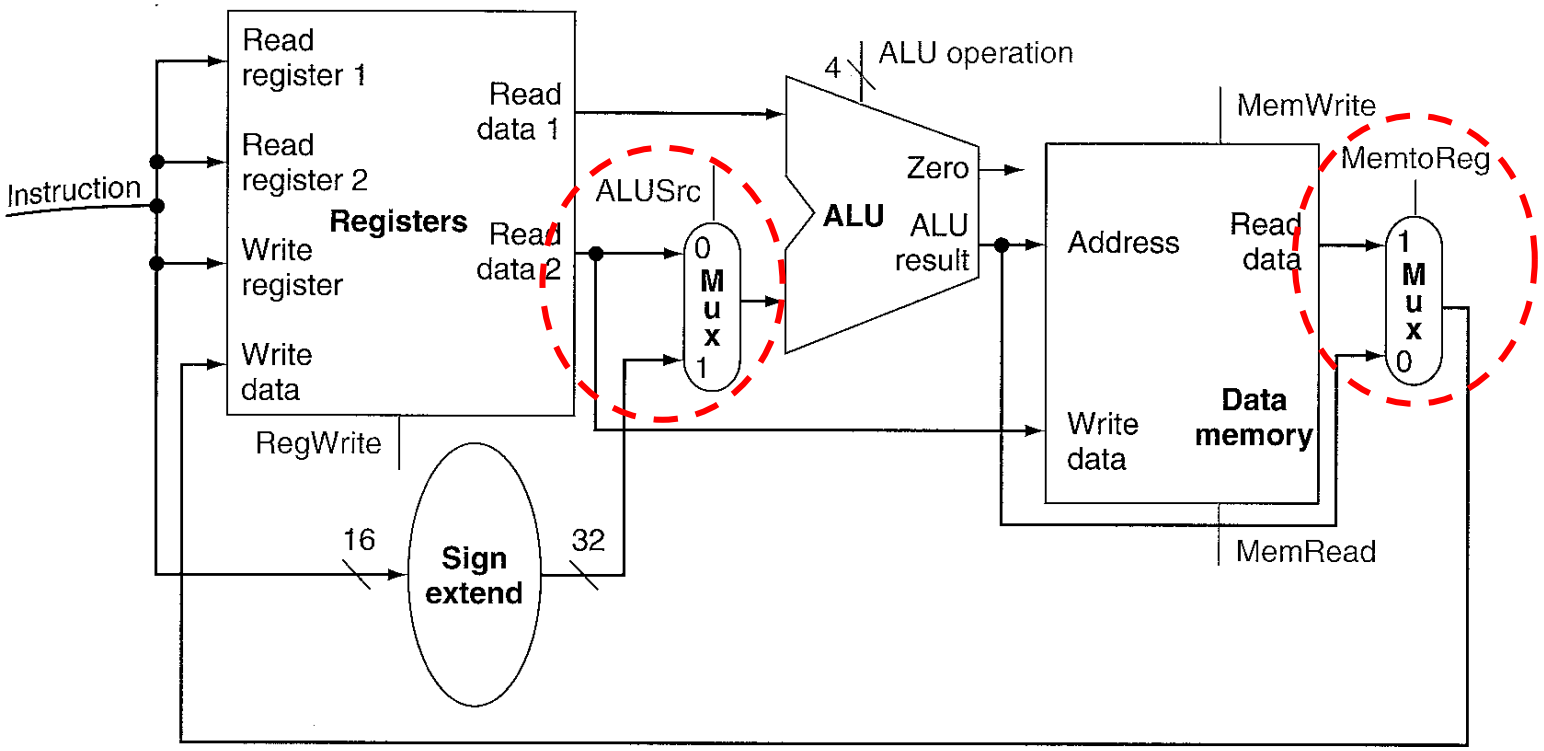
- **beq $t1, $t2, offset**



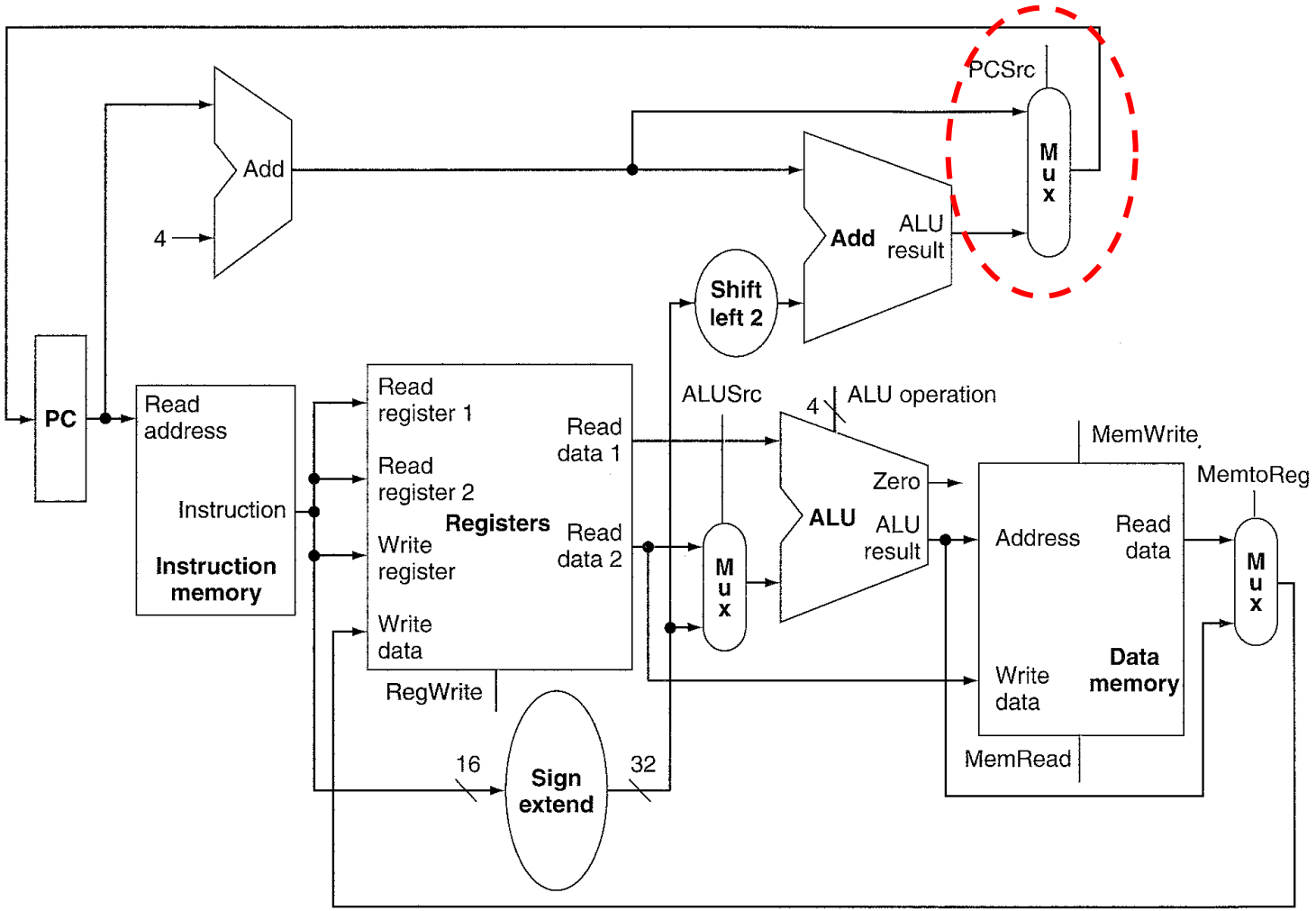
- I-type:

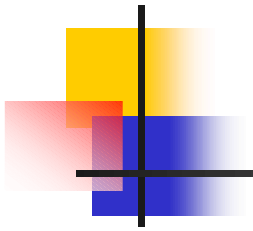


Datapath for both Memory and R-type Instructions (Fig. 5.10)



Simple Datapath for All three types of Instructions (Fig. 5.11)

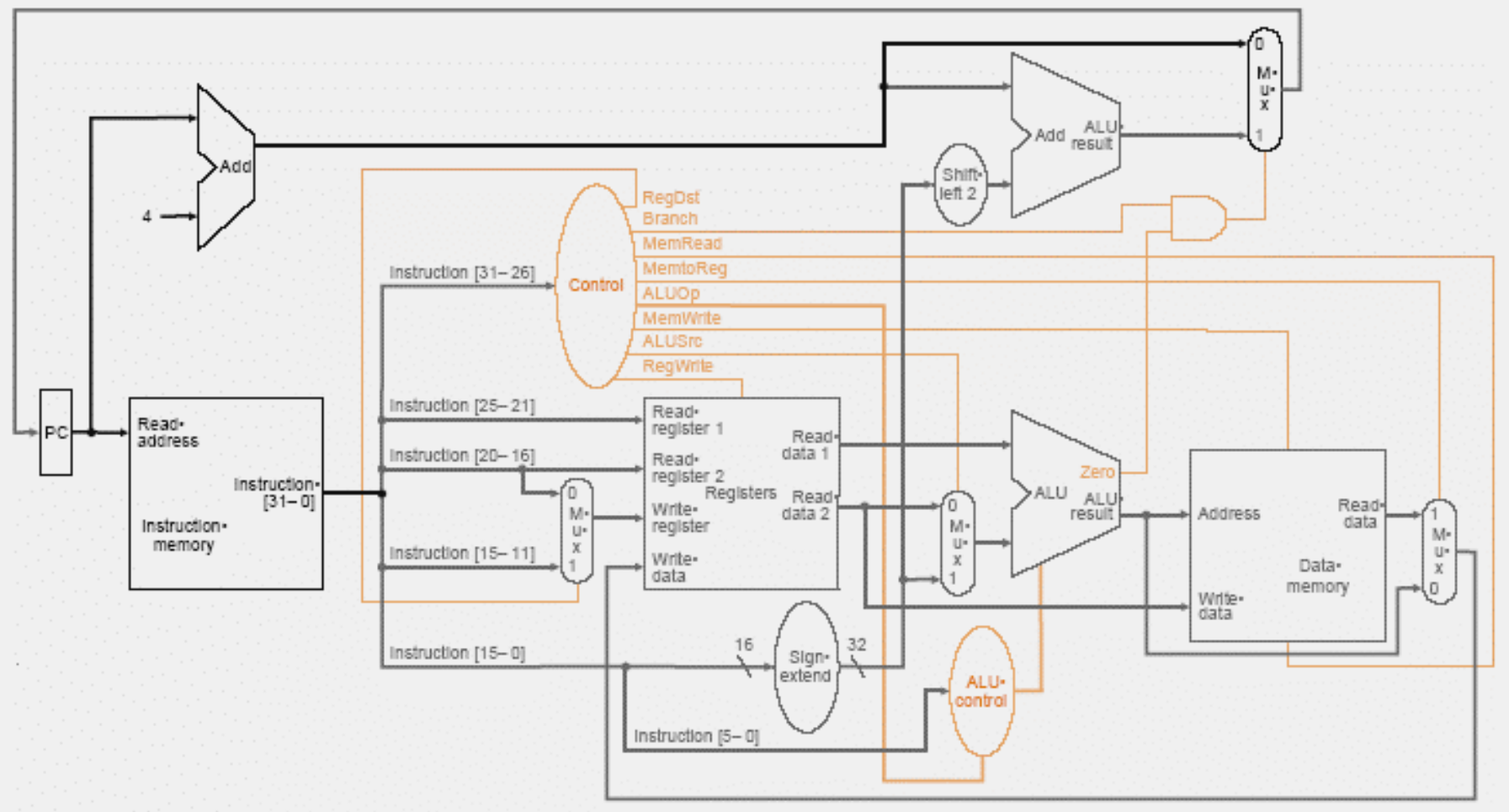




Outline

- 5.1 Introduction
- 5.2 Logic Design Conventions
- 5.3 Building a Datapath
- 5.4 A Simple Implementation Scheme

Basic datapath with control



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

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

A Simple Implementation Scheme

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

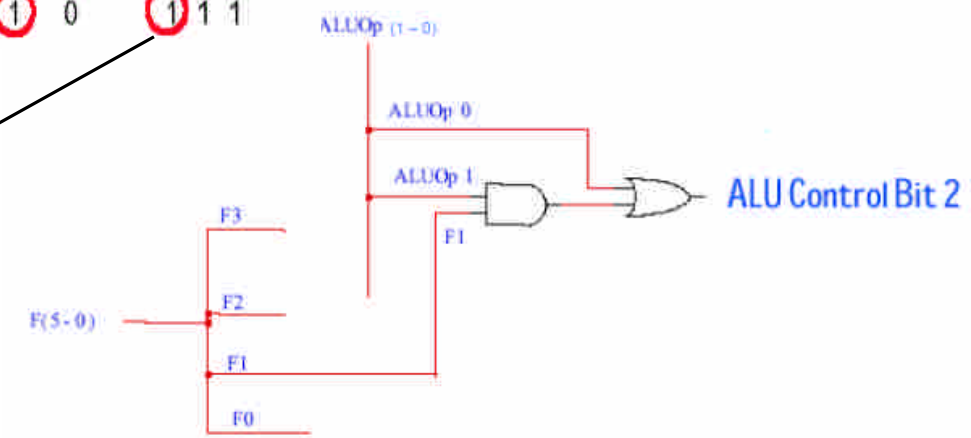
ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
x	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	1111

Simplify the ALU Control

- The ALU control bits are generated by ALUOp bits and Function code bits
  - Because ALU control bit3 is always 0, omit it.
  - When is ALU control bit2 == 1

ALUOp		Function code						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	ALU Control
0	0	X	X	X	X	X	X	0 1 0
X	1	X	X	X	X	X	X	1 1 0
1	X	X	X	0	0	0	0	0 1 0
1	X	X	X	0	0	1	0	1 1 0
1	X	X	X	0	1	0	1	0 0 1
1	X	X	X	1	0	1	0	1 1 1

ALU control bit2

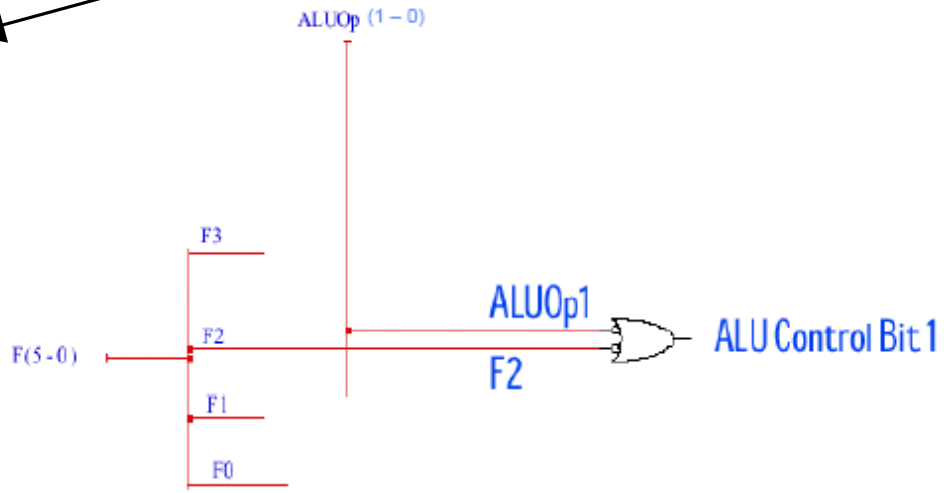


Simplify the ALU Control

- When is ALU control bit1 == 1

ALUOp		Function code						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	ALU Control
0	0	X	X	X	X	X	X	0 1 0
X = 0	1	X	X	X	X	X	X	1 1 0
1	X	X	X	0	0	0	0	0 1 0
1	X	X	X	0	0	1	0	1 1 0
1	X	X	X	0	1	0	1	0 0 1
1	X	X	X	1	0	1	0	1 1 1

ALU control bit1

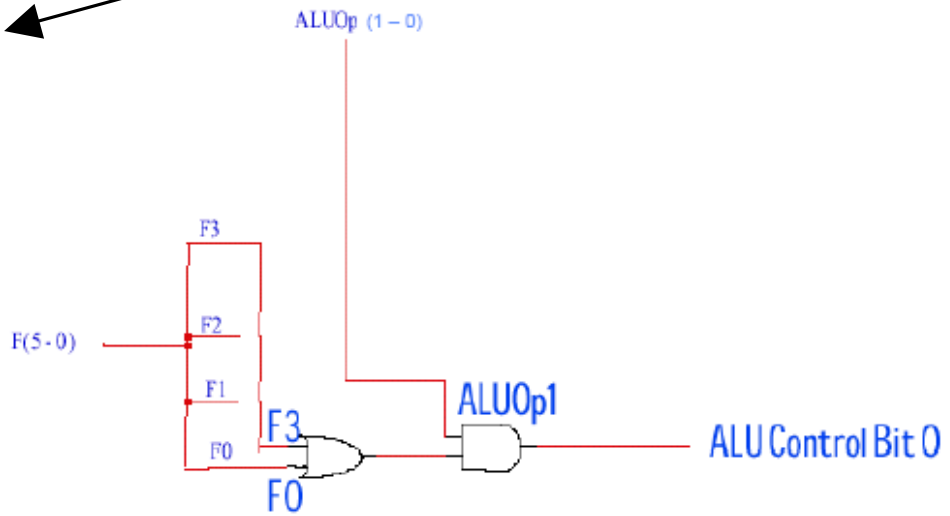


Simplify the ALU Control

- When is ALU control bit0 == 1

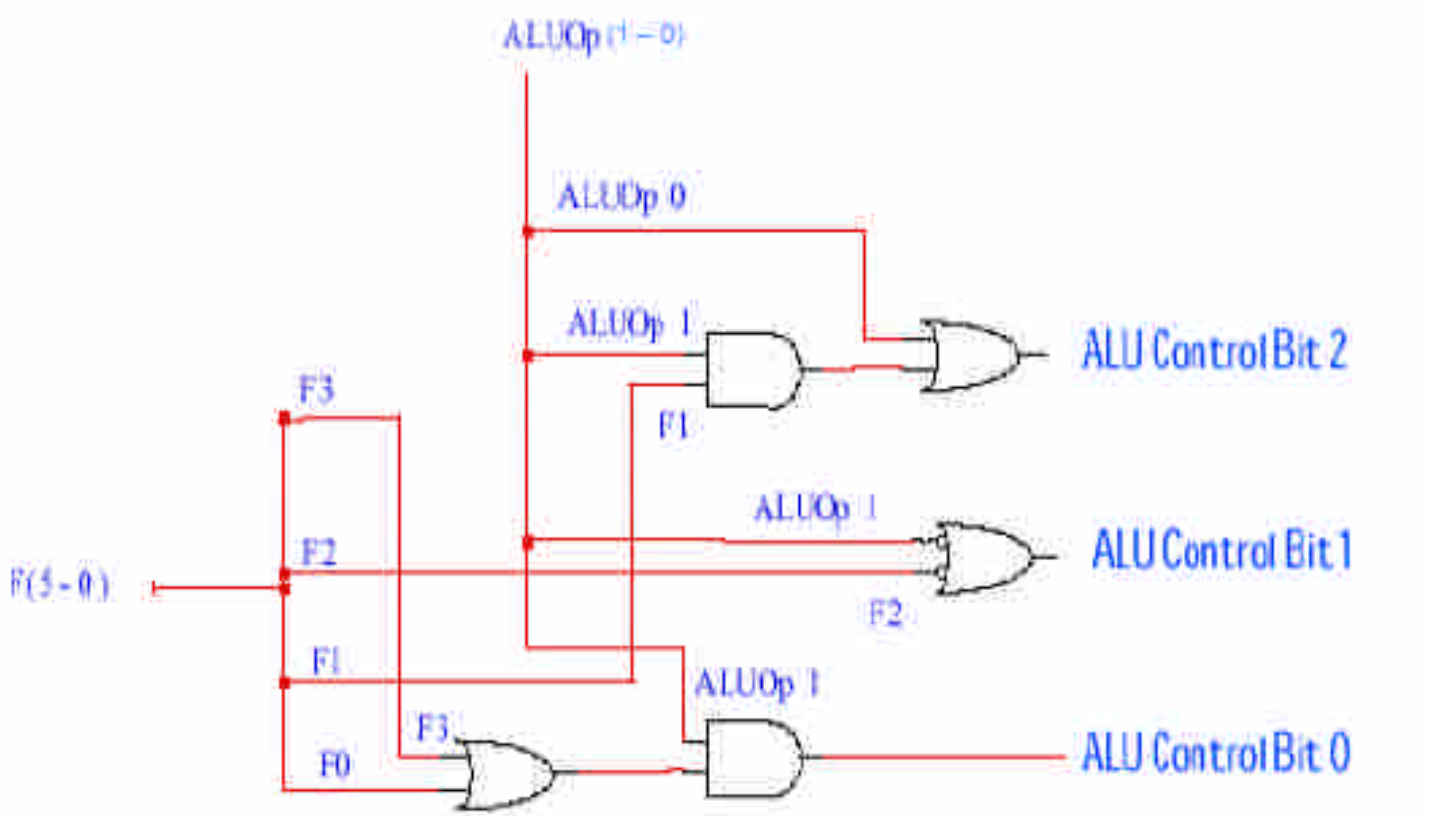
ALUOp		Function code						
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	ALU Control
0	0	X	X	X	X	X	X	0 1 0
X	1	X	X	X	X	X	X	1 1 0
1	X	X	X	0	0	0	0	0 1 0
1	X	X	X	0	0	1	0	1 1 0
1	X	X	X	0	1	0	1	0 0 1
1	X	X	X	1	0	1	0	1 1 1

ALU control bit0



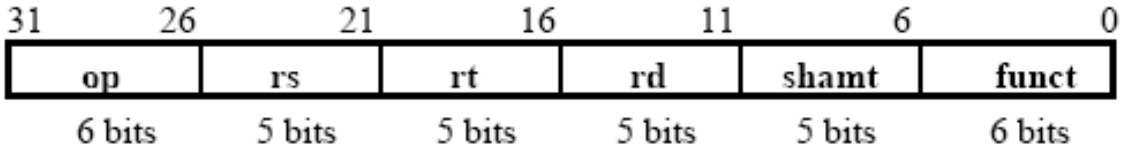
Simplify the ALU Control

- ALU control logic (overall)

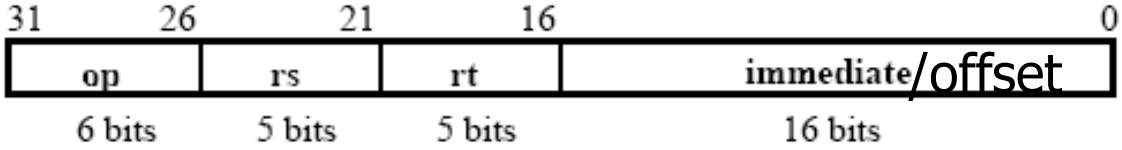


Designing the main control unit

- The two instruction classes
  - R-type:

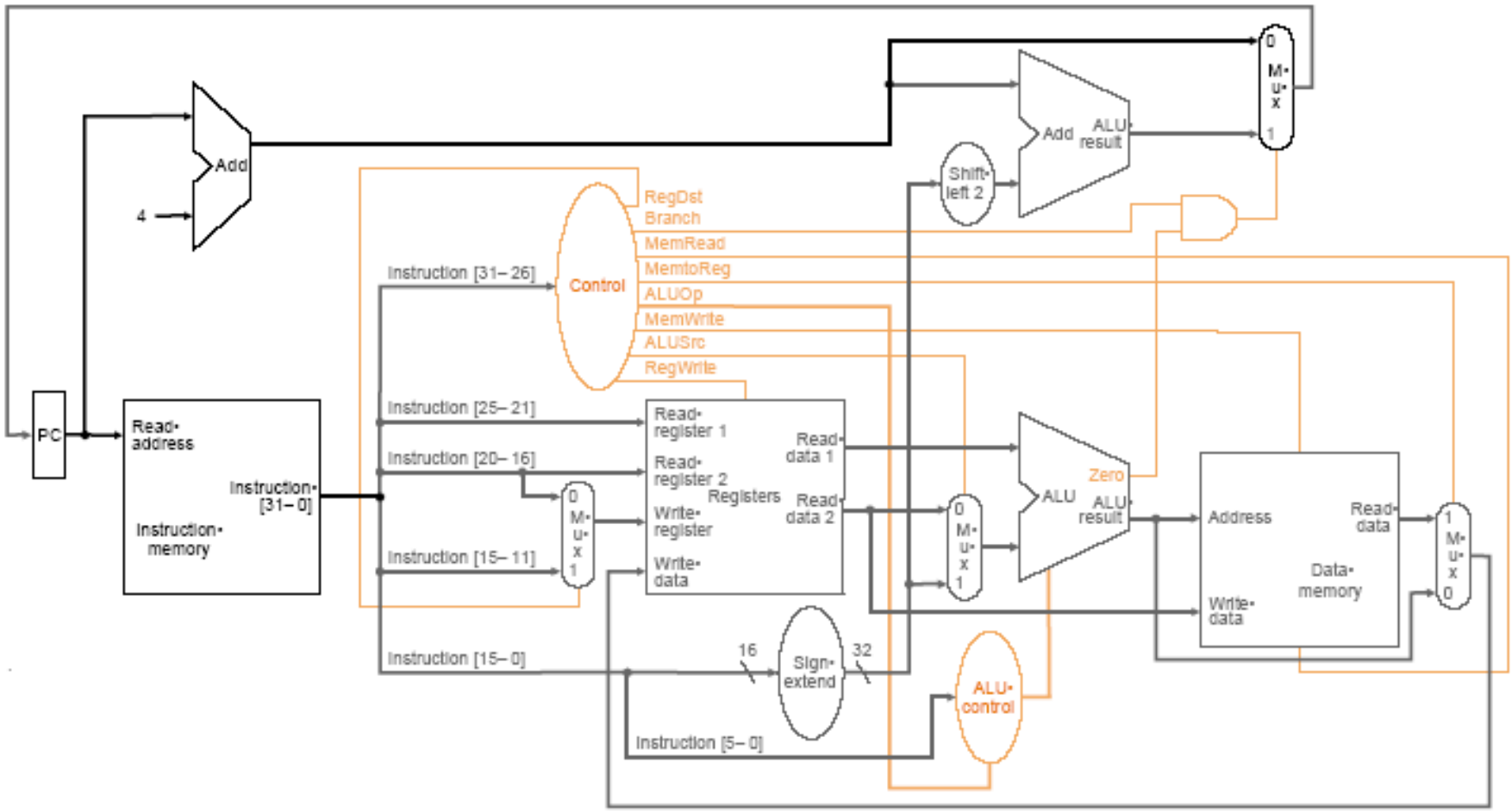


  - I-type:



- 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]

Simple datapath with the control unit



Control Unit Design

- The setting of the control lines is completed by the “opcode” field 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)

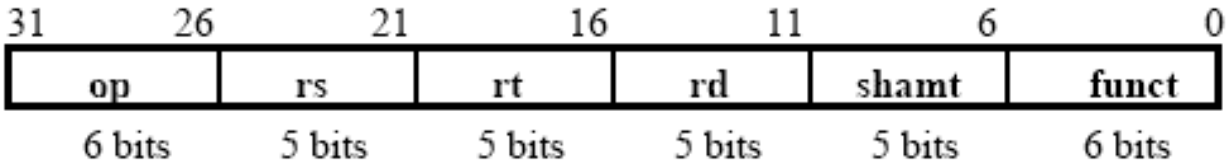
Effect of the 7 control signals

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:



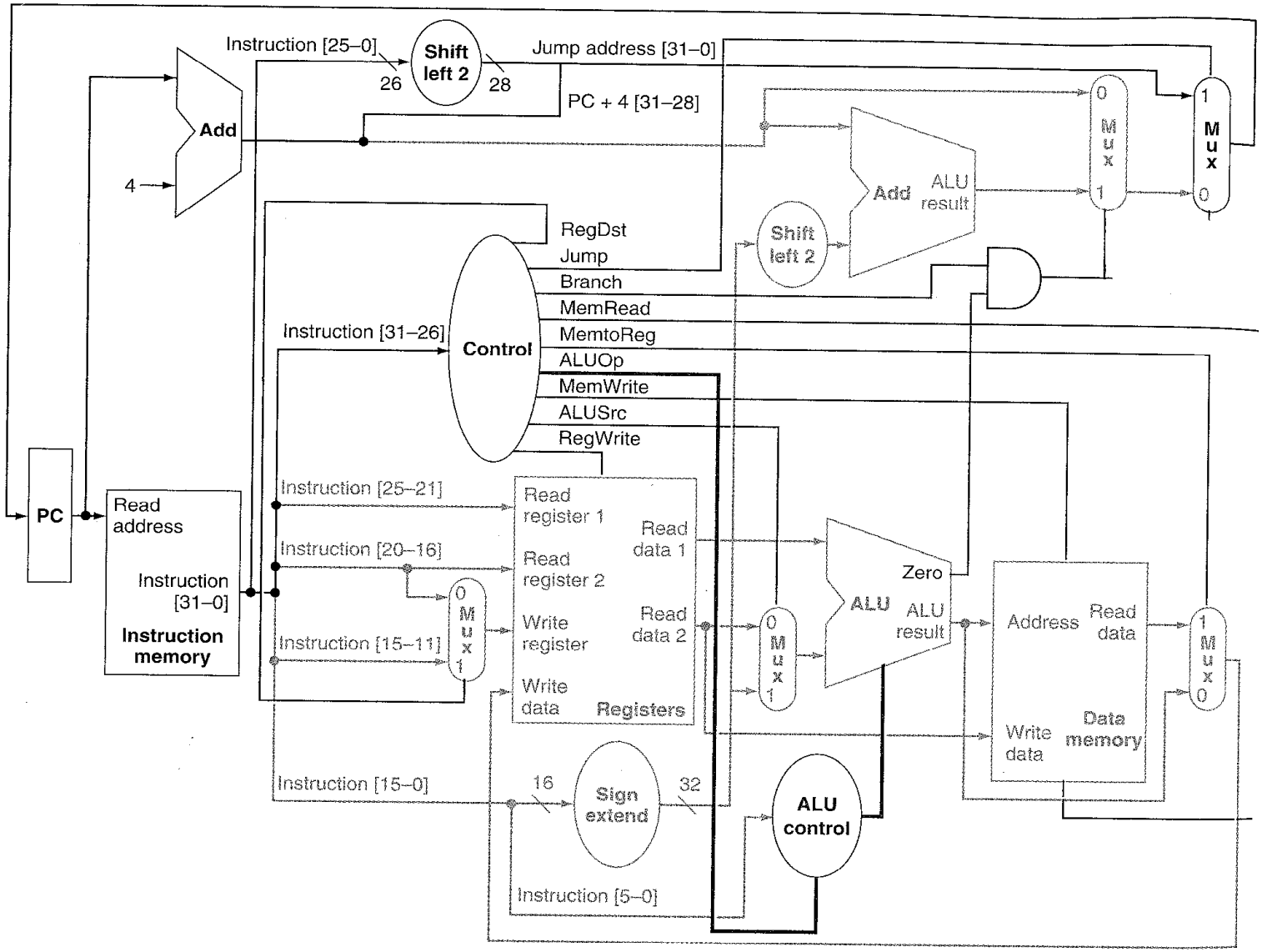
add $t1, $t2, $t3

- Fetch instruction and increment PC

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

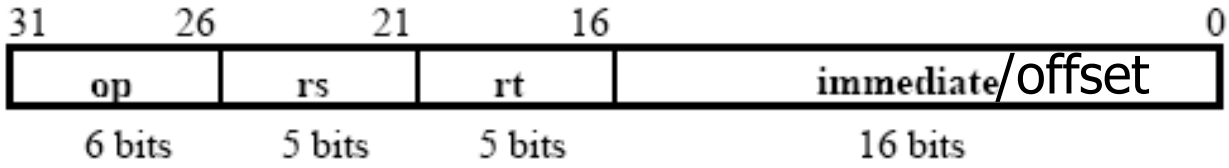
- Read registers (Reg1=Reg[rs], Reg2=Reg[rt])
- Run the ALU operation (Result = Reg1 ALUop Reg2)
- Store the result into Register File (Reg[rd] = Result)

Single-cycle MIPS with 4 instructions



Operation for “load” instruction

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



lw $t1, offset($t2)

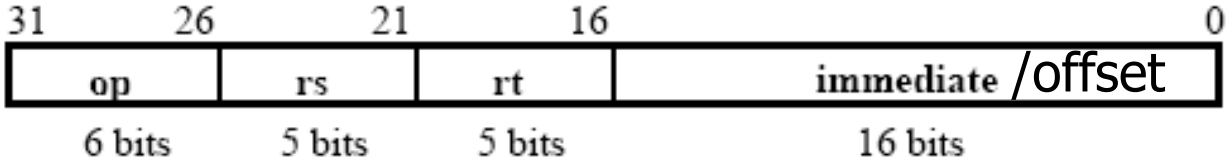
- Fetch instruction and increment PC

  (Instr=Memory[PC] ; PC = PC + 4)
- Read registers ($t2 = Reg[rs] , only one register is read)
- Address computing (Result = $t2 + sign-extend(Instr[15-0]))
- Load data from memory (Data = Memory[Result])
- Store data into Register File (Reg[rt] = Data)

Operation for “store” instruction

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

- I-type:



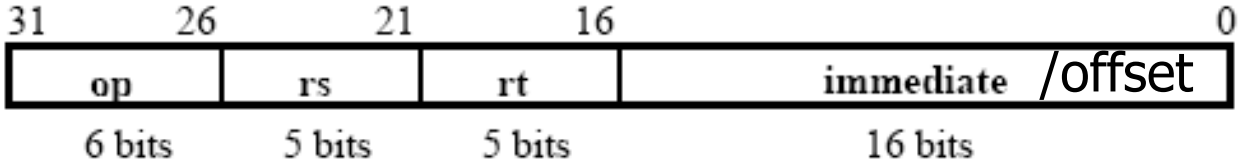
sw $t1, offset($t2)

- 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 4 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

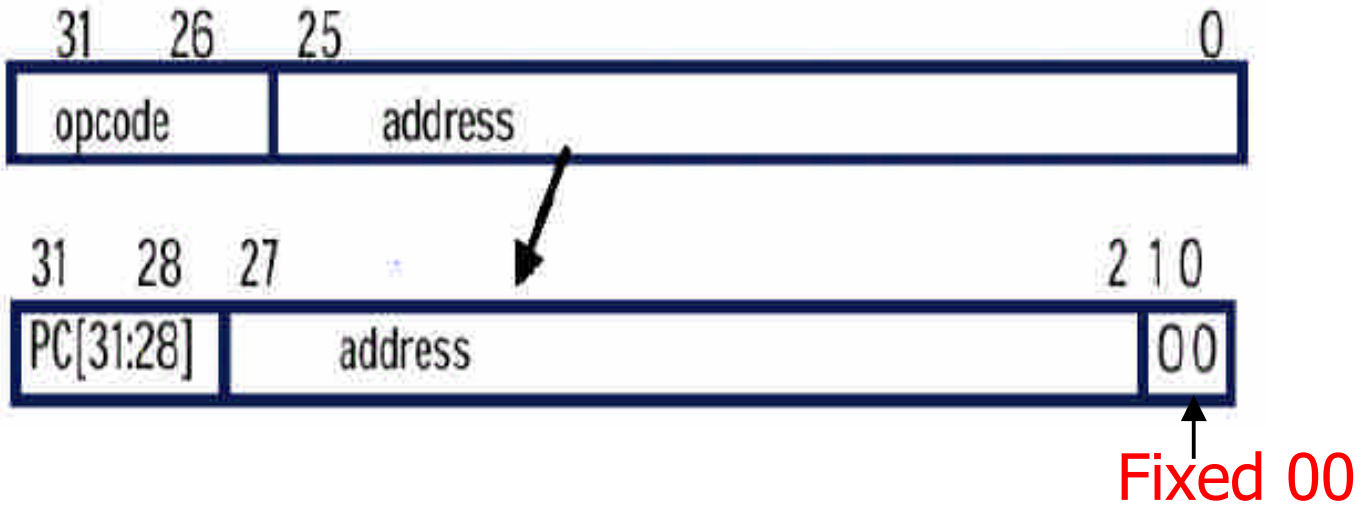
  (If zero==1, then PC = Result. Otherwise, PC unchanged)

Finalizing the control signals

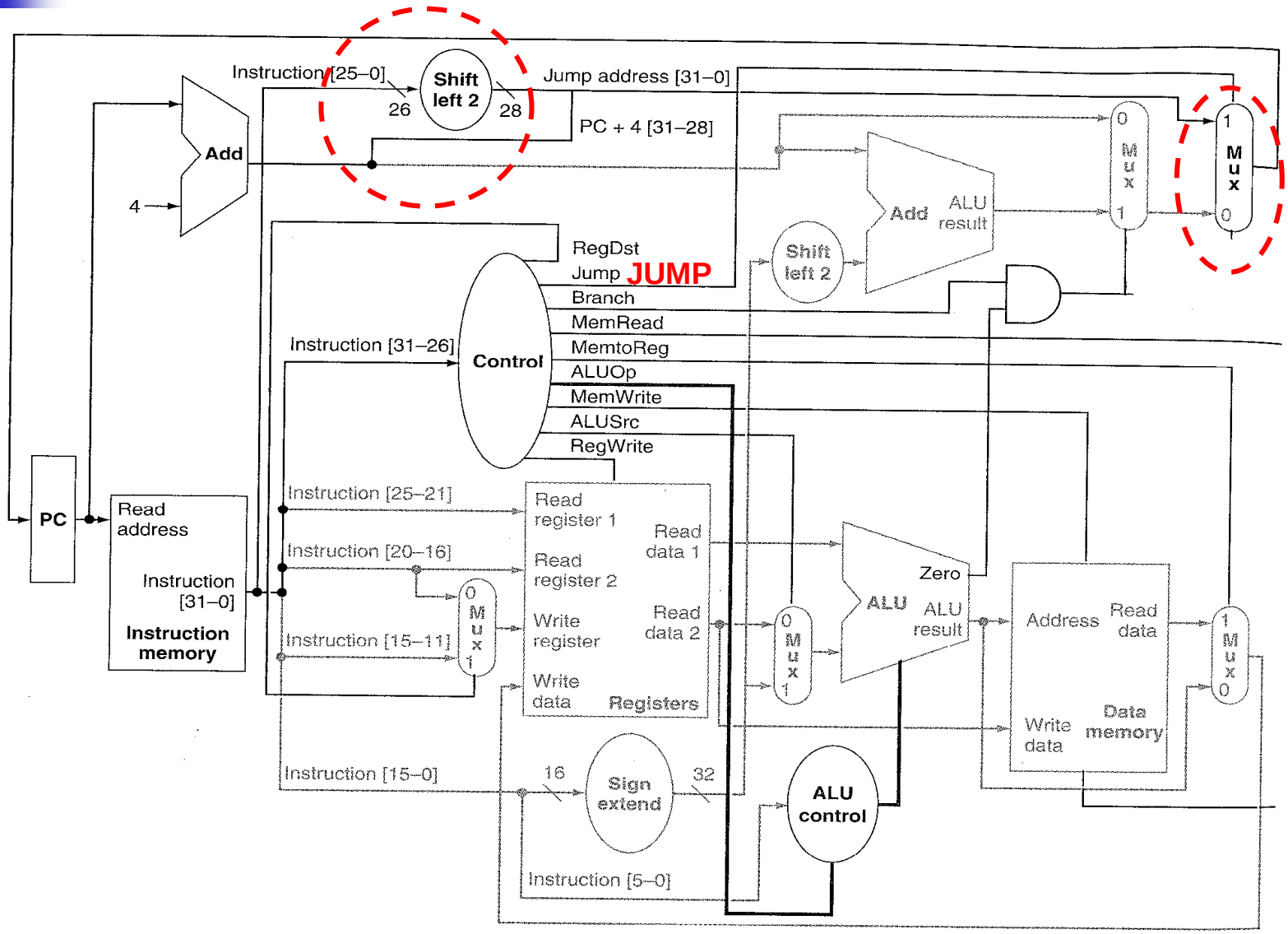
Input/output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
outputs	RegDst	1	0	x	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	x	x
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

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.

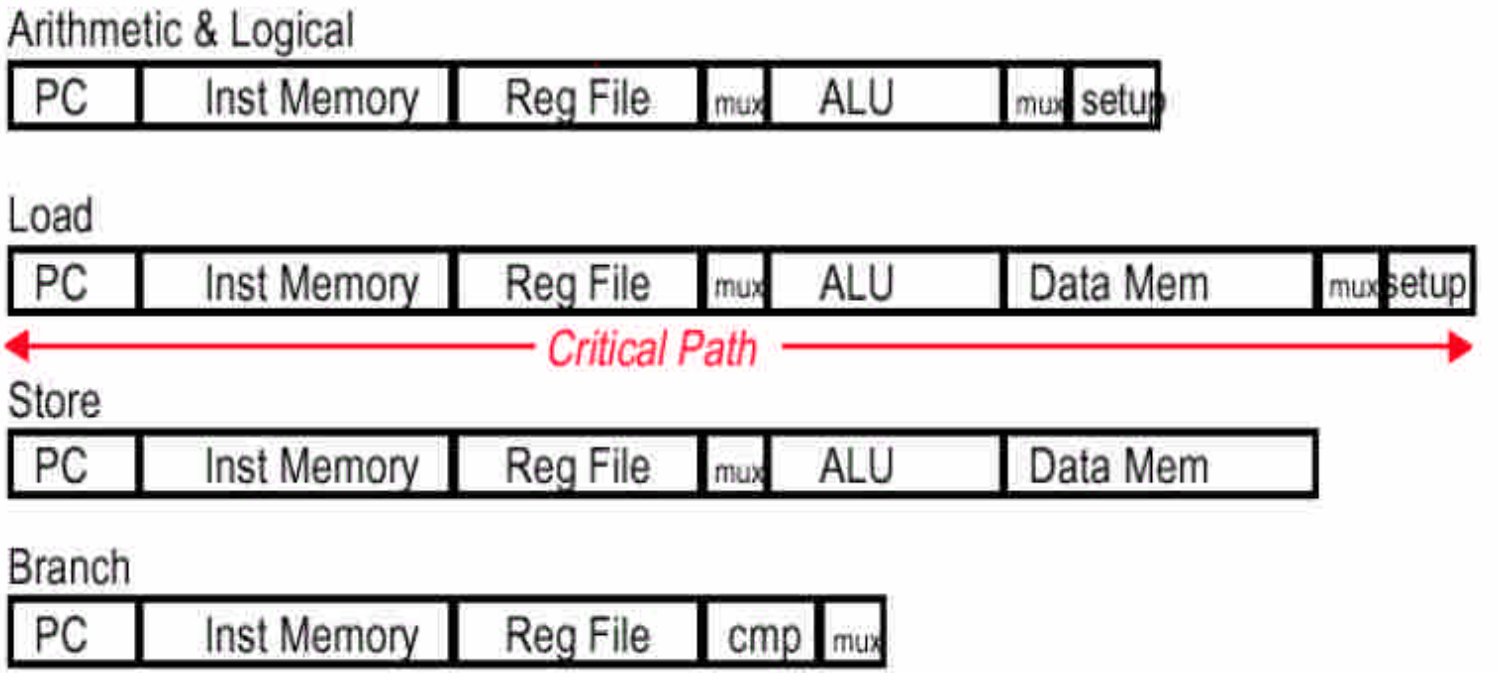


Implementing "Jumps"

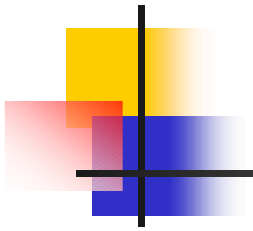


Single-cycle implementation

- 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 :

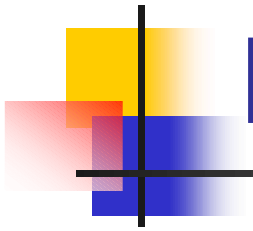
  (1) fixed clock cycle : 600 ps

  (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:

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



HW#5 and Mid-term Exam

- HW#5 (CPU survey proposal) due date: 11/5 (Friday by 2pm) to TA Shen (by E-mail). No late submissions

- Mid-term exam: Nov. 10, 2004, 10-12am (in classroom)
  - Closed book
  - Chapter 1.1 ~ Chapter 5.4