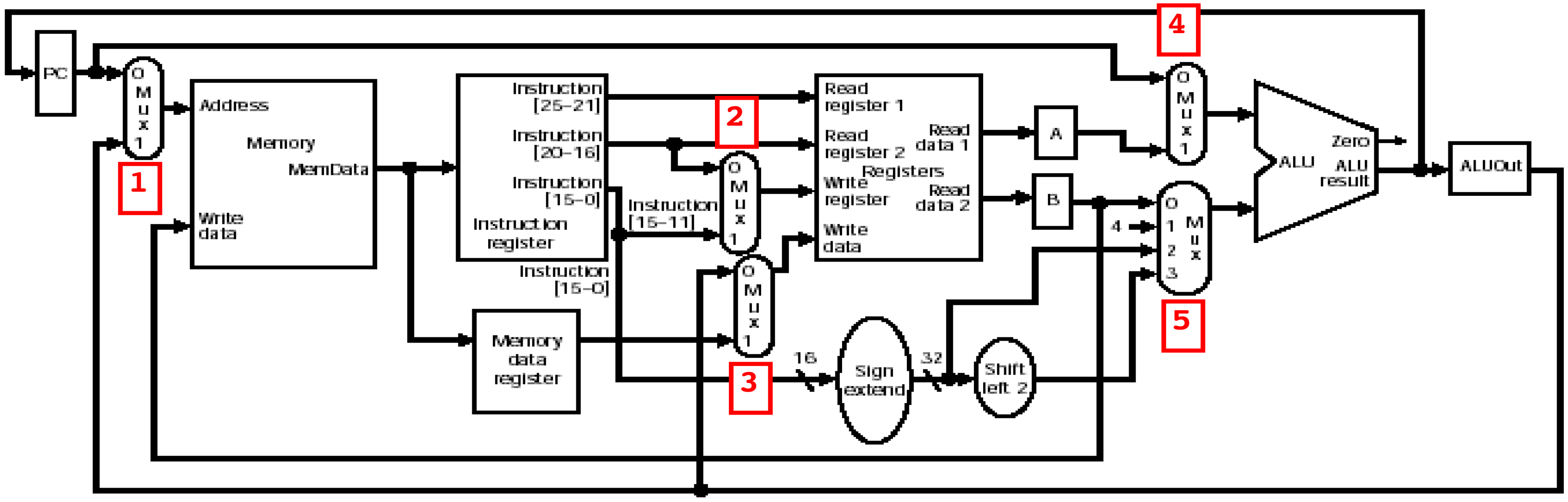
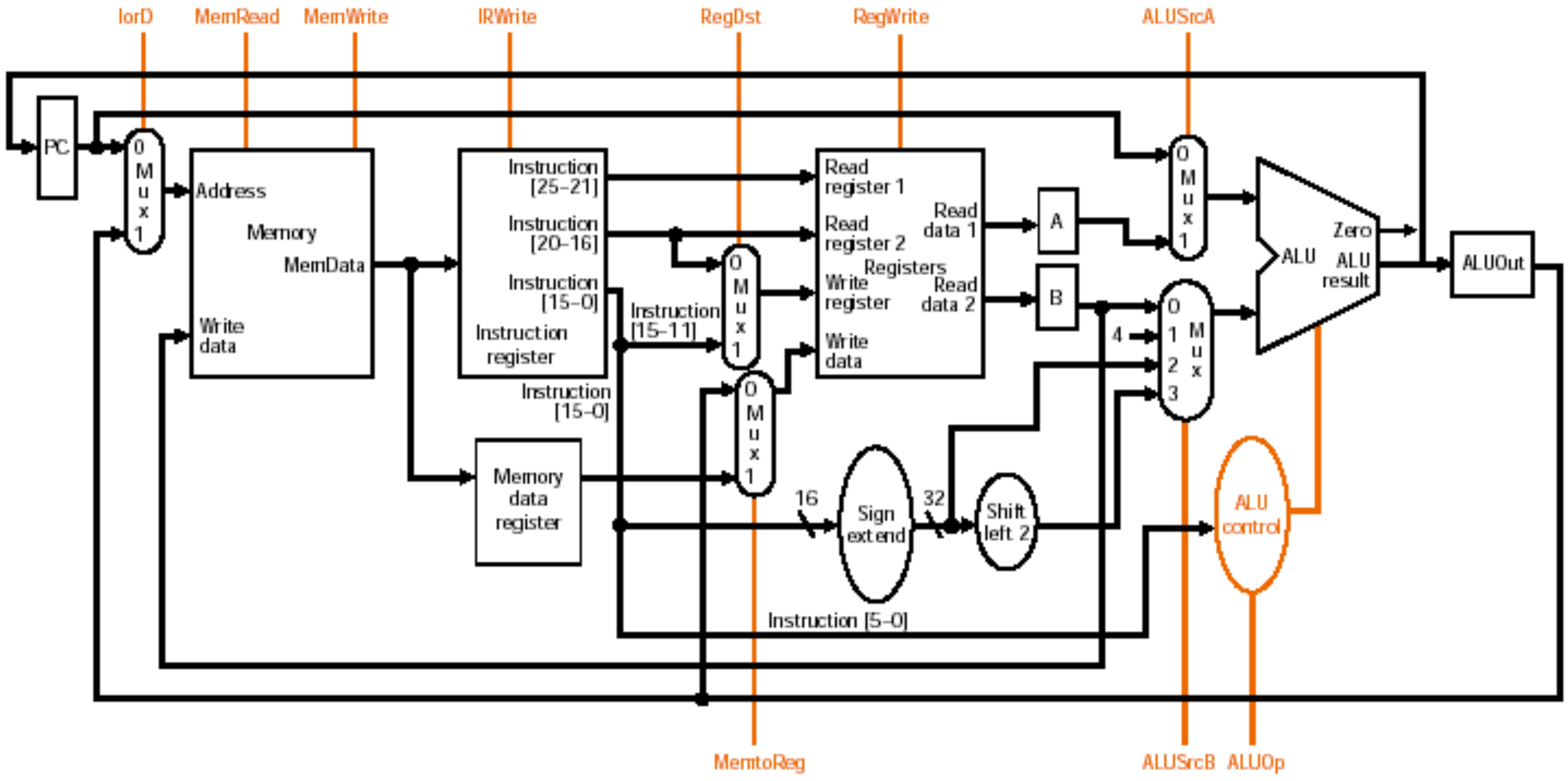
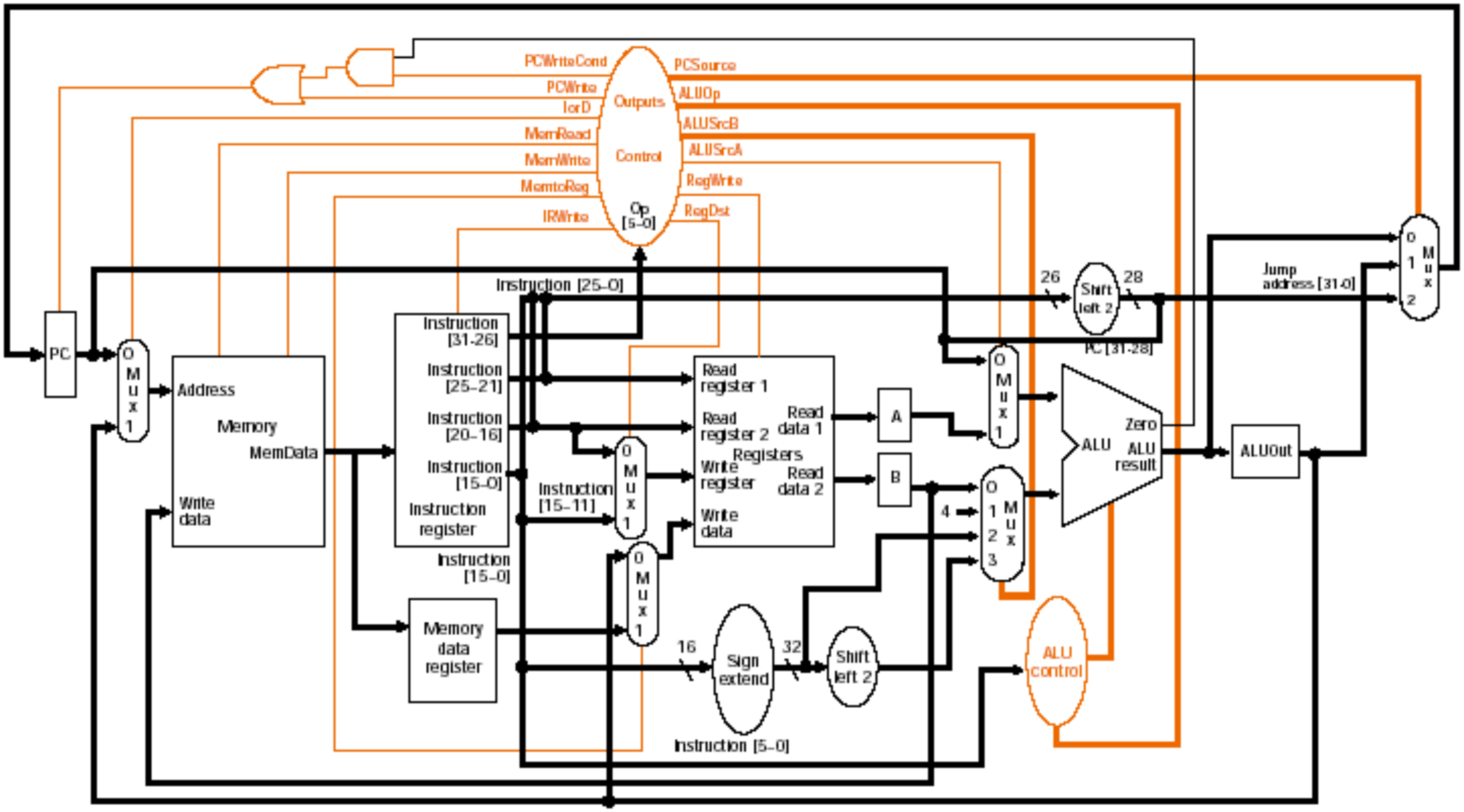
Muxes for Multi Cycle



Control Signals for Multi Cycle





PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op [5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
PC
0 M u x 1
Address
Memory
MemData
Write data
Instruction [31-26]
Instruction [25–21]
Instruction [20–16]
Instruction [15–0]
Instruction register
Instruction [15–0]
Memory data register
Instruction [25–0]
Instruction [15–11]
Read register 1
Read register 2
Registers
Write register
Write data
Read data 1
Read data 2
A
B
4
0 1 2 3 M u x
26
Shift left 2
28
PC [31-28]
Jump address [31-0]
0 1 2 M u x
ALU
Zero
ALU result
ALUOut
16
Sign extend
32
Shift left 2
Instruction [5–0]
ALU control

Cycle 1: Instruction Fetch

Work to be done - IR = Memory[PC]; PC = PC + 4;

Control signals needed

MemRead, IRWrite asserted

IorD set to 0 to select PC as address source

Increment PC by 4:

ALUSrcA = 0: PC to ALU

ALUSrcB = 01: 4 to ALU

ALUOp = 00: add

Store PC back

PCSource = 00: ALU result

PCWrite = 1

The memory access and PC increment can occur in parallel. Why?

Because the PC value doesn't change until the next clock cycle!

Where else is the incremented PC value stored? ALUOut

Cycle 2: Instruction decode and register fetch

What do we know about the type of instruction so far? Nothing!

So, we can only perform operations which apply to all instructions, or do not conflict with the actual inst.

What can we do at this point?

Read the registers from the register file into A and B

Compute branch address using ALU and save in ALUOut

But, what if the instruction doesn't use 2 registers, or it isn't a branch?

No problem; we can simply use what we need once we know what kind of instruction we have

This is why having a regular instruction pattern is a good idea.

Operation:

A = Reg[IR[25-21]];

B = Reg[IR[20-16]];

ALUOut = PC + sign_extend (IR[15-0]) << 2;

What are the control signals to determine whether to write registers A and B?

There aren't any! ***We can read the register file and store A and B on EVERY clock cycle.***

Branch address computation:

ALUSrcA = 0: PC to ALU

ALUSrcB = 11: sign-extended/shifted immediate to ALU

ALUOp = 00: add

These operations occur in parallel.

Cycle 3: ALU operates on the operands, depending on class of instruction

Memory reference: (ALUOut = A + sign_extend (IR[15-0]);)
Operation: ALU creates memory address by adding operands
Control signals:

- ALUSrcA = 1: register A
- ALUSrcB = 10: sign-extension unit output
- ALUOp = 00: add

Arithmetic-logical operation (R-type): (ALUOut = A op B;)
Operation: ALU performs operation specified by function code on values in registers A, B
(Where did these operands come from? They were read from the register file on the previous cycle.)
Control signals:

- ALUSrcA = 1: register A
- ALUSrcB = 00: register B
- ALUOp = 10: use function code bits to determine ALU control

Cycle 3: ALU operates on the operands, depending on class of instruction

<u>Branch If (A == B) PC = ALUOut;</u>

Operation: ALU compares A and B. If equal, Zero output signal is set to cause branch, and PC is updated with branch address

Control signals

- ALUSrcA = 1: register A
- ALUSrcB = 00: register B
- ALUOp = 01: subtract
- PCWriteCond = 1: update PC if Zero signal is 1
- PCSource = 01: ALUOut

(What is in ALUOut, and how did it get there? It's the branch address calculated from the previous cycle, NOT the result of A – B. Why not? Because ALUOut is updated at the END of each cycle.)

Note that PC is actually updated twice if the branch is taken:

Output of the ALU in the previous cycle (instruction decode/register fetch),

From ALUOut if A and B are equal

Could this cause any problems? No, because only the last value of PC is used for the next instruction execution.

Cycle 4: Memory access or R-type instruction completion

Load or store: accesses memory

Arithmetic-logical operation writes result to register

<u>Memory reference MDR = Memory[ALUOut]; or Memory[ALUOut] = B;</u>

Operation: If operation is load, word from memory is put into MDR. If operation is store, memory location is written with value from register B.

(Where does memory address come from? It was computed by ALU in previous cycle.
Where does register B value come from? It was read from register file in step 3 and also in step 2.)

Control signals

MemRead = 1 (load) or
MemWrite = 1 (store)
IorD = 1: address from ALU, not PC
What about MDR? It's written on every clock cycle.

<u>Arithmetic-logical operation Reg[IR[15-11]] = ALUOut;</u>

Operation: ALUOut contents are stored in result register.

Control signals

RegDst = 1: use $rd field from IR for result register
RegWrite = 1: write the result register
MemtoReg = 0: write from ALUOut, not memory data

Cycle 5: Memory read completion

<u>Value read from memory is written back to register Reg[IR[20-16]] = MDR;</u>

Operation: Write the load data from MDR to target register $rt

Control signals

- MemtoReg = 1: write from MDR
- RegWrite = 1: write the result register
- RegDst = 0: use $rt field from IR for result register