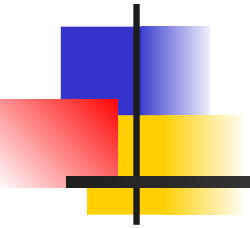
COMPUTER ARCHITECTURE



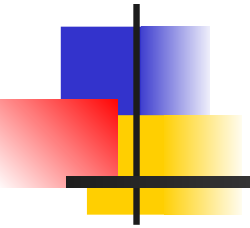
Processor: Multi-Cycle Datapath & Control

(Based on text: David A. Patterson & John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, 3rd Ed., Morgan Kaufmann, 2007)

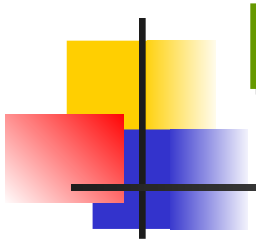
COURSE CONTENTS

- Introduction
- Instructions
- Computer Arithmetic
- Performance
- **Processor: Datapath**
- **Processor: Control**
- Pipelining Techniques
- Memory
- Input/Output Devices

PROCESSOR: DATAPATH & CONTROL



- Multi-Cycle Datapath
- Multi-Cycle Control
- Additional Registers and Multiplexers

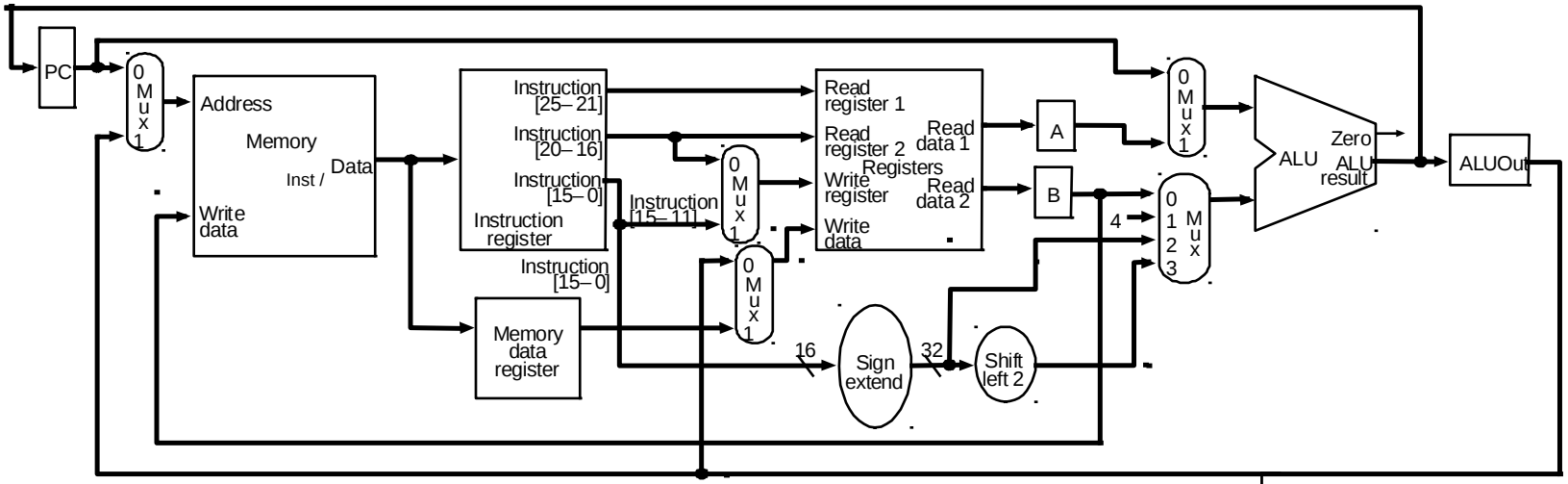


Multicycle Approach

- Break up an instruction into steps, each **step** takes a **cycle**:
  - **balance** the amount of work to be done
  - restrict each cycle to use only **one** major functional unit
  - Different instructions take **different** number of cycles to complete
- At the **end** of a **cycle**:
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” **registers** for such **temporal** storage
- **Reusing** functional units (reduces hardware cost):
  - Use **ALU** to compute address/result and to increment PC
  - Use **memory** for both instructions and data

Multi-Cycle Datapath: Additional Registers

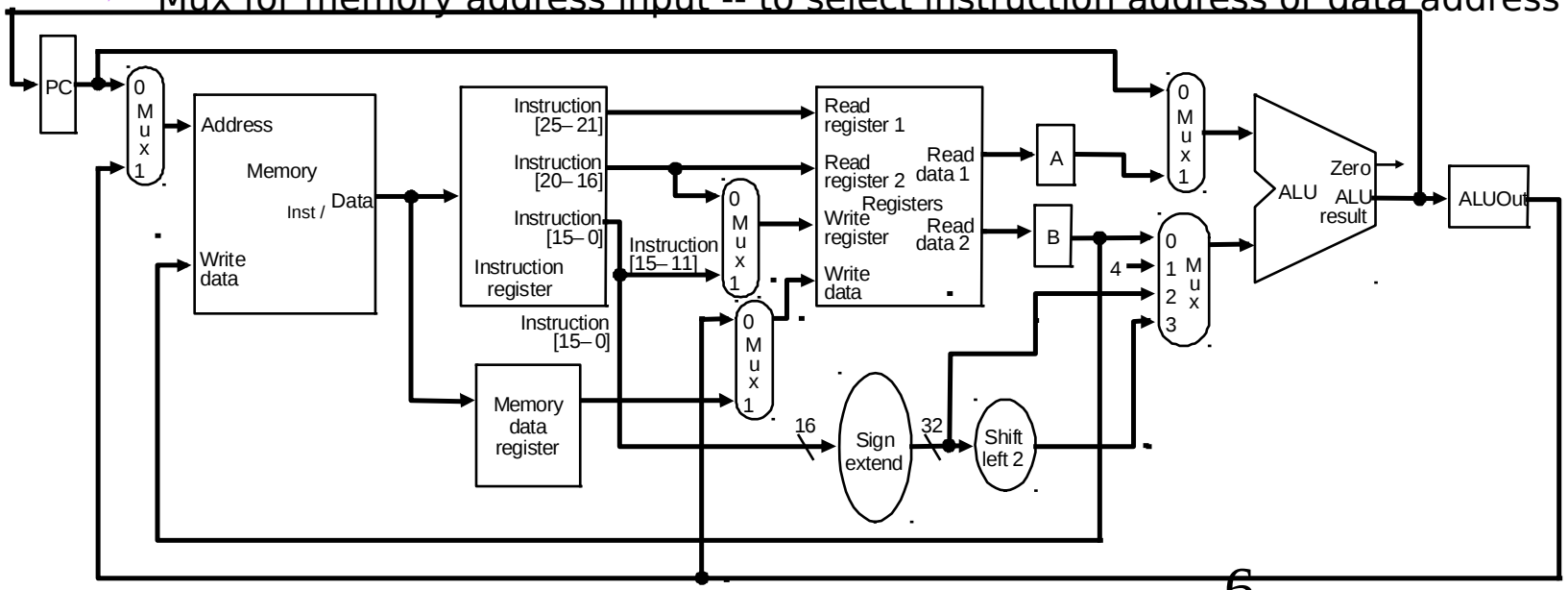
- Additional “internal registers”:
  - Instruction register (**IR**) -- to hold current instruction
  - Memory data register (**MDR**) -- to hold data read from memory
  - A register (**A**) & B register (**B**) -- to hold register operand values from register files
  - ALUOut register (**ALUOut**) -- to hold output of ALU, also serves as memory address register (MAR)
- All registers except IR hold data only between a pair of **adjacent** cycles and thus do **not** need **write** control signals; IR holds instructions till **end** of instruction, hence **needs** a **write** control signal



Note: we ignore jump inst here

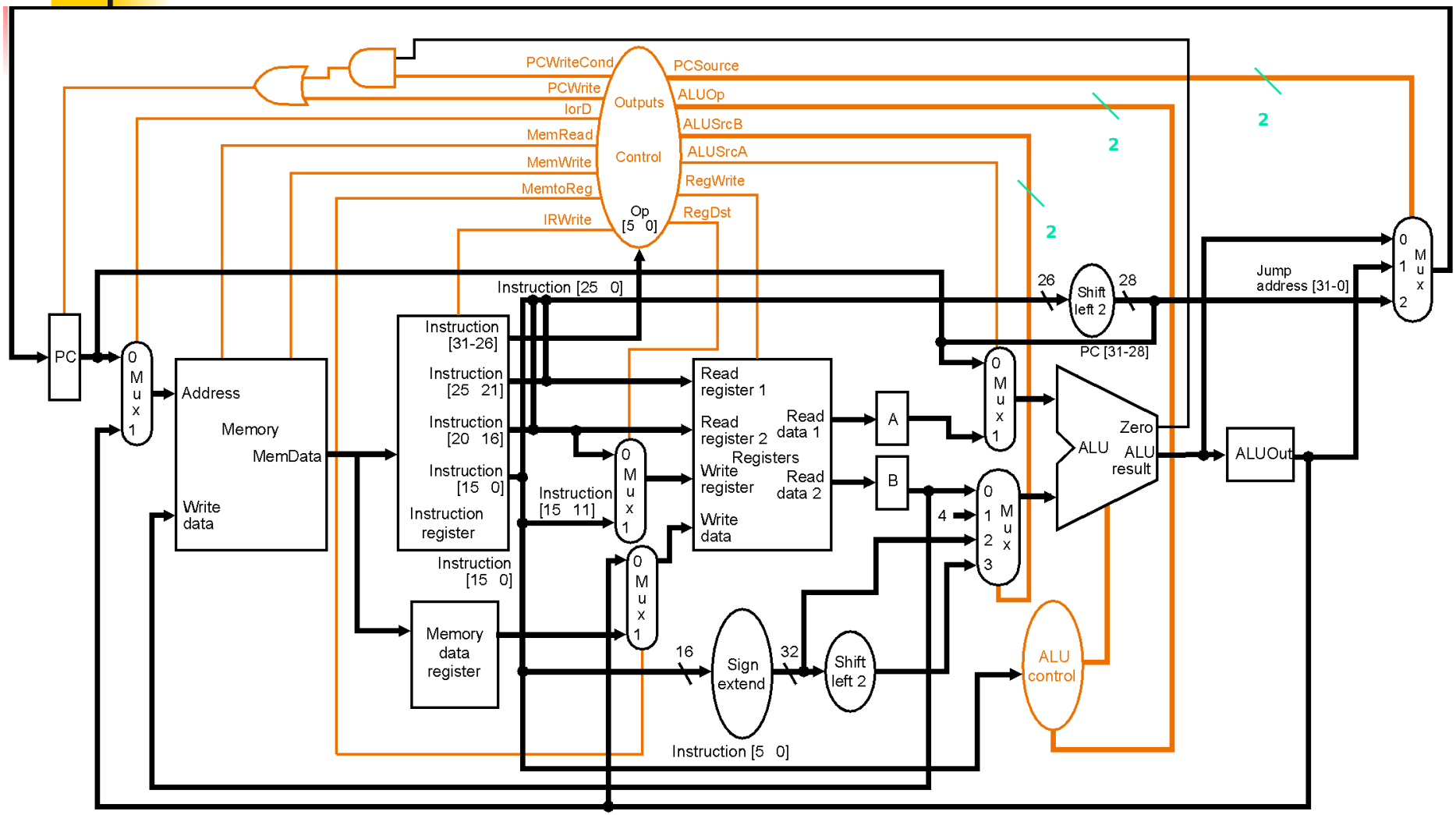
Multicycle Datapath: Additional Multiplexors

- Additional multiplexors:
  - Mux for first ALU input -- to select A or PC (since we use ALU for both address/result computation & PC increment)
  - Bigger mux for second ALU input -- due to two additional inputs: 4 (for normal PC increment) and the sign-extended & shifted offset field (in branch address computation)
  - Mux for memory address input -- to select instruction address or data address



Note: we ignore jump inst here

Multi-Cycle Datapath & Control



Note the reason for each control signal; also note that we have included the jump instruction

Control Signals for Multi-Cycle Datapath

- Note:
  - three possible sources for value to be written into PC (controlled by PCSource): (1) regular increment of PC, (2) conditional branch target from ALUOut, (3) unconditional jump (lower 26 bits of instruction in IR shifted left by 2 and concatenated with upper 4 bits of the incremented PC)
  - two PC write control signals: (1) PCWrite (for unconditional jump), & (2) PCWriteCond (for “zero” signal to cause a PC write if asserted during beq inst.)
  - since memory is used for both inst. & data, need IorD to select appropriate addresses
  - IRWrite needed for IR so that instruction is written to IR (IRWrite = 1) during the first cycle of the instruction and to ensure that IR not be overwritten by another instruction during the later cycles of the current instruction execution (by keeping IRWrite = 0)
  - other control signals

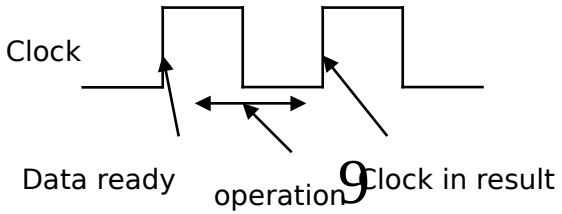
Breaking the Instruction into 3 - 5 Execution Steps

1. Instruction Fetch (All instructions)
2. Instruction Decode (All instructions), Register Fetch & Branch Address Computation (in advance, just in case)
3. ALU (R-type) execution, Memory Address Computation, or Branch Completion (Instruction dependent)
4. Memory Access or R-type Instruction Completion (Instruction dependent)
5. Memory Read Completion (only for lw)

At end of every clock cycle, needed data must be stored into register(s) or memory location(s).

Each step (can be several parallel operations) is 1 clock cycle --> Instructions take 3 to 5 cycles!

Events during a cycle, e.g.:



Step 1: Instruction Fetch

- Use PC to get instruction (from memory) and put it in the Instruction Register
- Increment of the PC by 4 and put the result back in the PC
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR <= Memory[PC];
PC <= PC + 4;
```

- *Which control signals need to be asserted?*
  - IorD = 0, MemRead = 1, IRWrite = 1
  - ALUSrcA = 0, ALUSrcB = 01, ALUOp = 00, PCWrite = 1, PCSource = 00

- *Why can instruction read & PC update be in the same step? Look at state element timing*

- *What is the advantage of updating the PC now?*

Step 2: Instruction Decode, Reg. Fetch, & Branch Addr. Comp.

- In this step, we **decode** the instruction in IR (the opcode enters control unit in order to generate control signals). **In parallel**, we can
- Read registers rs and rt, *just in case* we need them
- Compute the branch address, *just in case* the instruction is a branch beq
- RTL:

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```

- Control signals:
  - ALUSrcA = 0, ALUSrcB = 11, ALUOp = 00 (add)
  - Note: no explicit control signals needed to write A, B, & ALUOut. They are written by clock transitions automatically at **end of step**

Step 3: Instruction Dependent Operation

- One of four functions, **based on** instruction type:

- Memory address computation (for lw, sw):

```
ALUOut <= A + sign-extend(IR[15:0]);
```

  - Control signals: ALUSrcA = 1, ALUSrcB = 10, ALUOp = 00
- ALU (R-type):

```
ALUOut <= A op B;
```

  - Control signals: ALUSrcA = 1, ALUSrcB = 00, ALUOp = 10
- Conditional branch:

```
if (A==B) PC <= ALUOut;
```

  - Control signals: ALUSrcA = 1, ALUSrcB = 00, ALUOp = 01 (Sub), PCSource = 01, PCWriteCond = 1 (to enable zero to write PC if 1)

What is the content of ALUOut during this step? Immediately after this step?

- Jump:

```
PC <= PC[31:28] || (IR[25:0]<<2);
```

  - Control signals: PCSource = 10, PCWrite = 1

- ✔ *Note: Conditional branch & jump instructions completed at this step!*

Step 4: Memory Access or ALU (R-type) Instruction Completion

- For lw or sw instructions (access memory):

```
MDR <= Memory[ALUOut];
       or
Memory[ALUOut] <= B;
```

- Control signals (for lw): IorD = 1 (to select ALUOut as address), MemRead = 1, note that no write signal needed for writing to MDR, it is written by clock transition automatically at end of step
- Control signals (for sw): IorD = 1 (to select ALUOut as address), MemWrite = 1

- For ALU (R-type) instructions (write result to register):

```
Reg[IR[15:11]] <= ALUOut;
```

- Control signals: RegDst = 1 (to select register address), MemtoReg = 0, RegWrite = 1

- *The write actually takes place at the **end** of the cycle on the clock*

Step 5: Memory Read Completion

- For lw instruction only (write data from MDR to register):

```
Reg[IR[20:16]]<= MDR;
```

- Control signals: RegDst = 0 (to select register address), MemtoReg = 1, RegWrite = 1

- ✔ *Note: lw instruction completed at this step!*

Summary of Execution Steps

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	A <= Reg [IR[25:21]]			
decode/register fetch	B <= Reg [IR[20:16]]			
/branch addr comp	ALUOut <= PC + (sign-extend (IR[15:0]) << 2)			
Execution, address	ALUOut <= A op B	ALUOut <= A + sign-extend	if (A ==B) then	PC <= PC [31:28]
computation, branch/		(IR[15:0])	PC <= ALUOut	ll (IR[25:0]<<2)
jump completion				
Memory access or R-type	Reg [IR[15:11]] <=	Load: MDR <= Memory[ALUOut]		
completion	ALUOut	or		
		Store: Memory [ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Some instructions take shorter number of cycles, therefore next instructions can start earlier.

Hence, compare to single-cycle implementation where all instructions take same amount of time, multi-cycle implementation is faster!

Multi-cycle implementation also reduces hardware cost (reduces adders & memory, increases number of registers & muxes)

Simple Questions

- How many cycles will it take to execute this code?

```
            lw $t2, 0($t3)
            lw $t3, 4($t3)
            beq $t2, $t3, Label        #assume not
            add $t5, $t2, $t3
            sw $t5, 8($t3)
Label:      ...
```

- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` takes place?