Lecture 10: Datapath Control; Multicycle

- Organizational
  - We're still grading the exams; hopefully done on Thursday
  - Hand out HW #3, due a week from today
  - Partial solutions to HW #3 available on Thursday
- Last Time
  - Datapath organization
- Today
  - Datapath Control
  - Multicycle Machine
  - Introduction to Pipelining (if we have time)

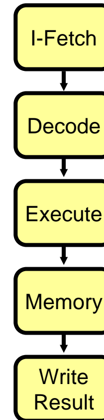
©2004 Morgan Kaufmann Publishers

Datapath Review

©2004 Morgan Kaufmann Publishers

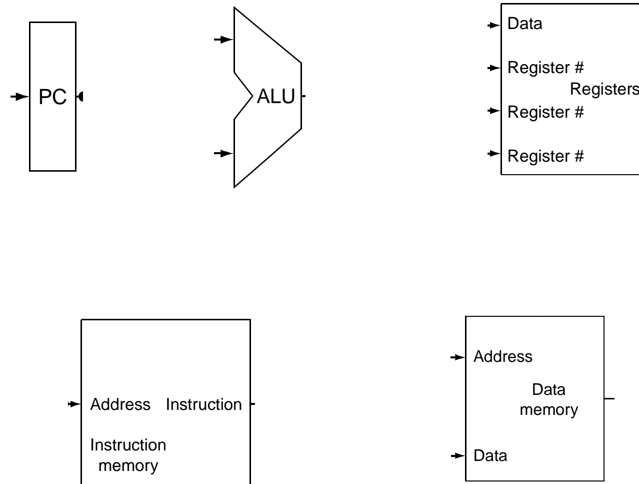
Instruction Execution

- 5 basic steps
  - fetch instruction (F)
  - decode instruction and read registers (R)
  - execute (X)
  - access memory (M)
  - store result (W)



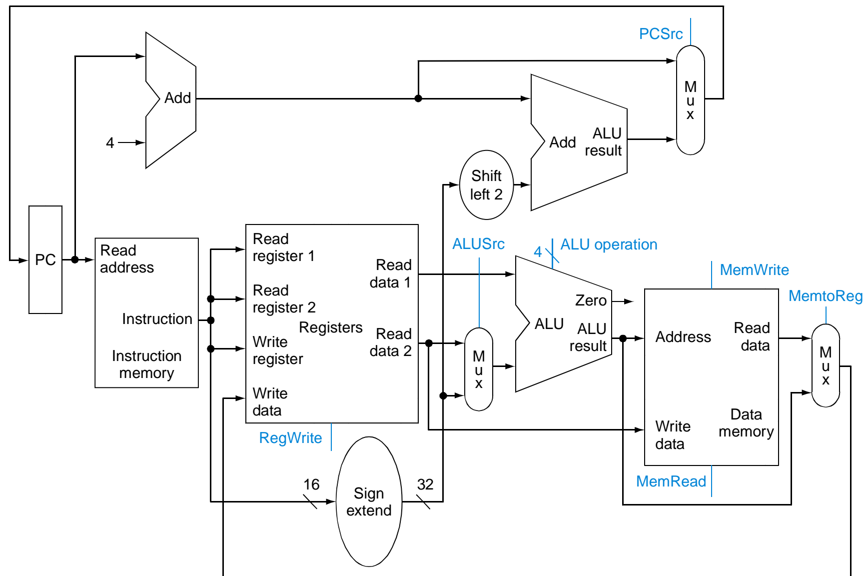
©2004 Morgan Kaufmann Publishers

Pieces we'll need

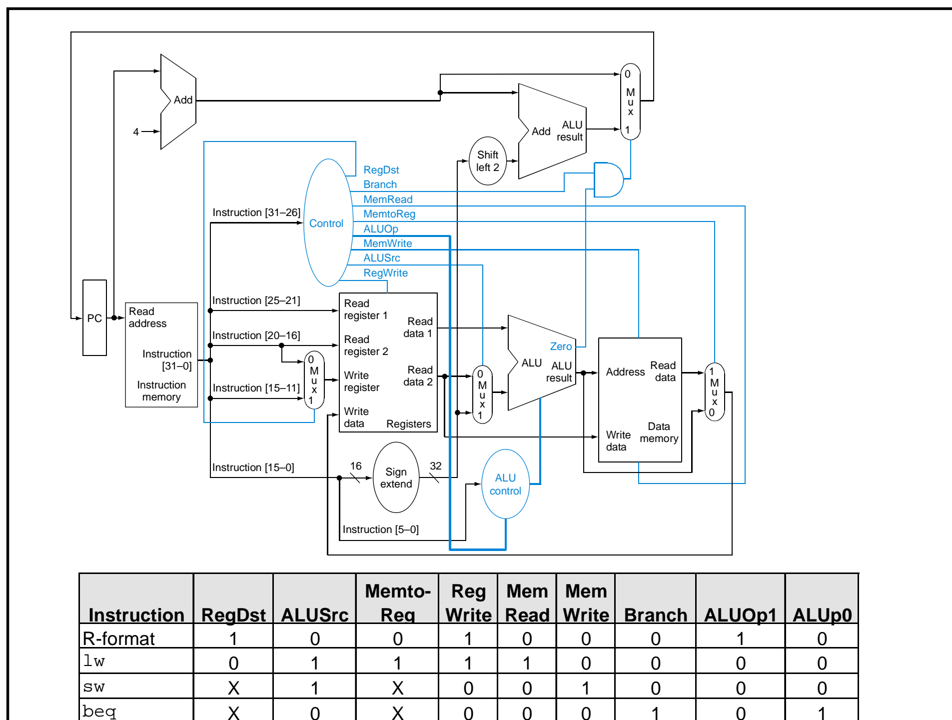


©2004 Morgan Kaufmann Publishers

Single-cycle MIPS datapath

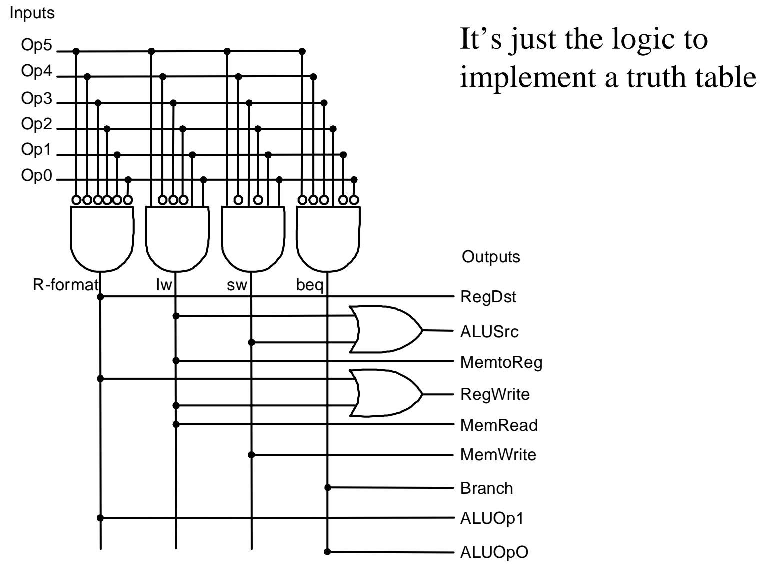


©2004 Morgan Kaufmann Publishers



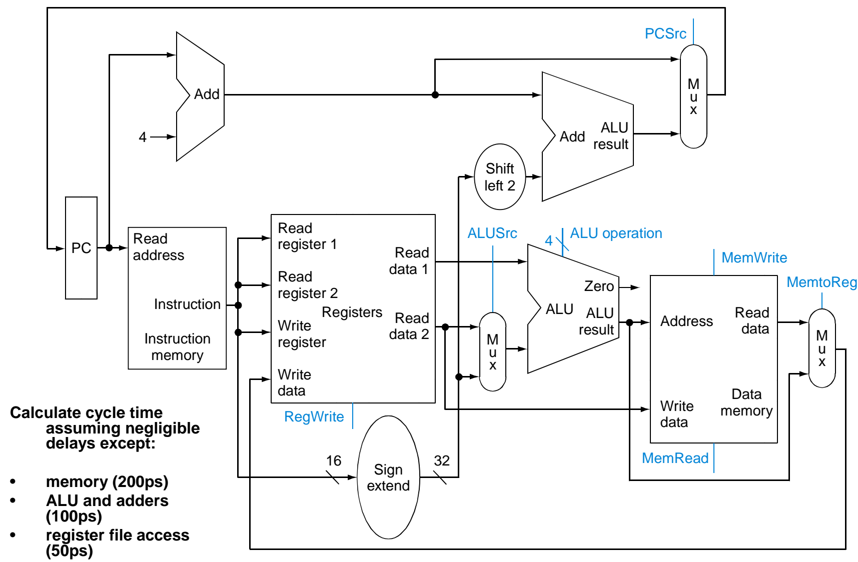
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Control is built with combinational logic



It's just the logic to implement a truth table

Single Cycle Implementation

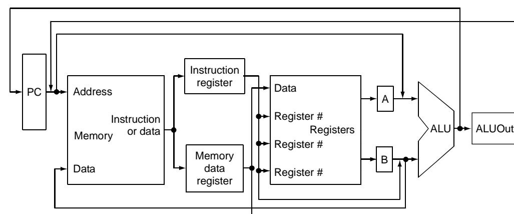


Calculate cycle time assuming negligible delays except:

- **memory (200ps)**
- **ALU and adders (100ps)**
- **register file access (50ps)**

Where we are headed

- Single Cycle Problems:
  - what if we want to reuse hardware (e.g. ALU/Adder) rather than having two copies?
- One Solution:
  - use a "smaller" cycle time
  - have different instructions take different numbers of cycles
  - a "multicycle" datapath:



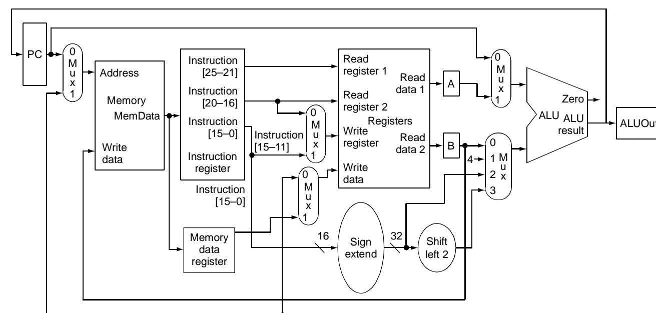
Multicycle Datapath

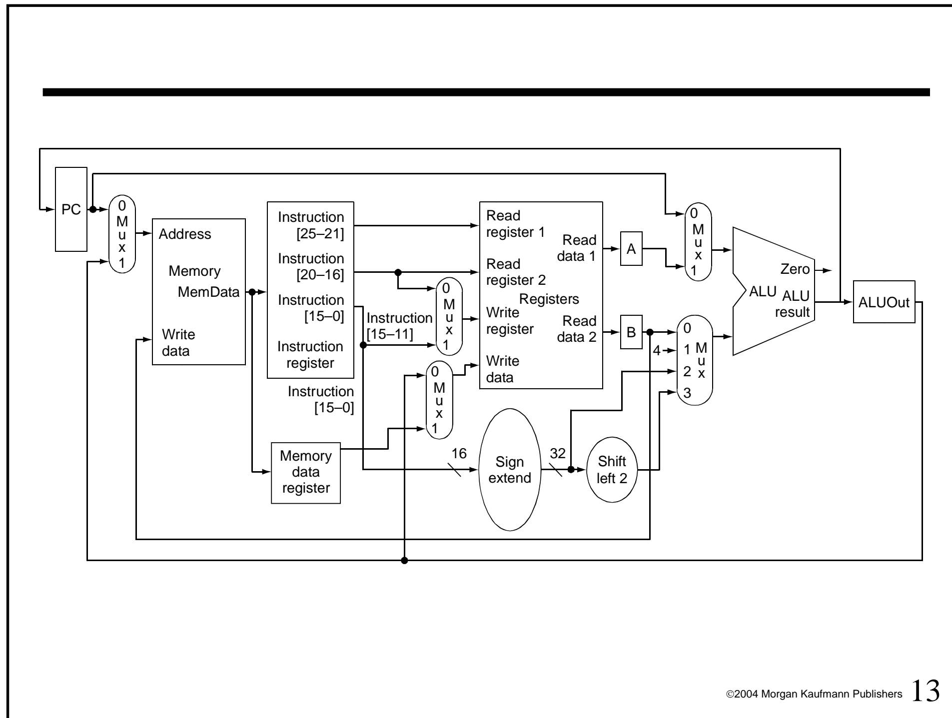
Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a “subtract” instruction?
- We’ll use a finite state machine for control

Multicycle Approach

- Break up the instructions 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
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers





Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum (“op”) of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

```
Reg[Memory[PC][15:11]] <=  Reg[Memory[PC][25:21]] op
                            Reg[Memory[PC][20:16]]
```

  - In order to accomplish this we must break up the instruction. (kind of like introducing variables when programming)

Breaking down an instruction

- **ISA definition of arithmetic:**

```
Reg[Memory[PC][15:11]] <= Reg[Memory[PC][25:21]]  op
                          Reg[Memory[PC][20:16]]
```

- **Could break down to:**
  - `IR <= Memory[PC]`
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]`
  - `ALUOut <= A op B`
  - `Reg[IR[20:16]] <= ALUOut`

- **We forgot an important part of the definition of arithmetic!**
  - `PC <= PC + 4`

©2004 Morgan Kaufmann Publishers

Idea behind multicycle approach

- **We define each instruction from the ISA perspective (do this!)**
- **Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)**
- **Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)**
- **Finally try and pack as much work into each step**
  (avoid unnecessary cycles)
  while also trying to share steps where possible
  (minimizes control, helps to simplify solution)
- **Result: Our book's multicycle Implementation!**

©2004 Morgan Kaufmann Publishers

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

©2004 Morgan Kaufmann Publishers

Step 1: Instruction Fetch

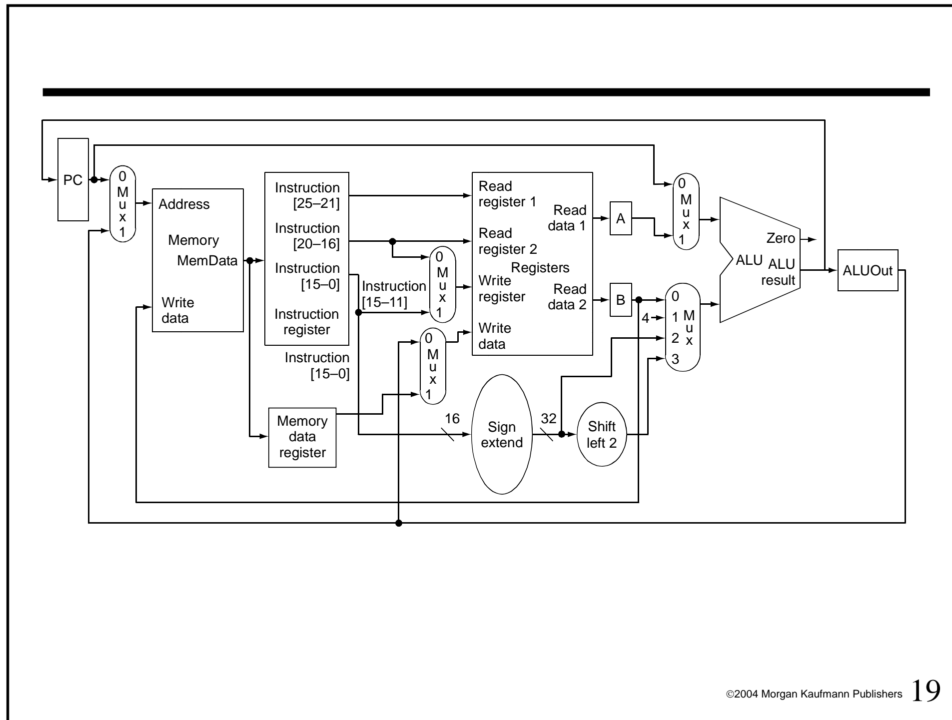
- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

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

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

©2004 Morgan Kaufmann Publishers

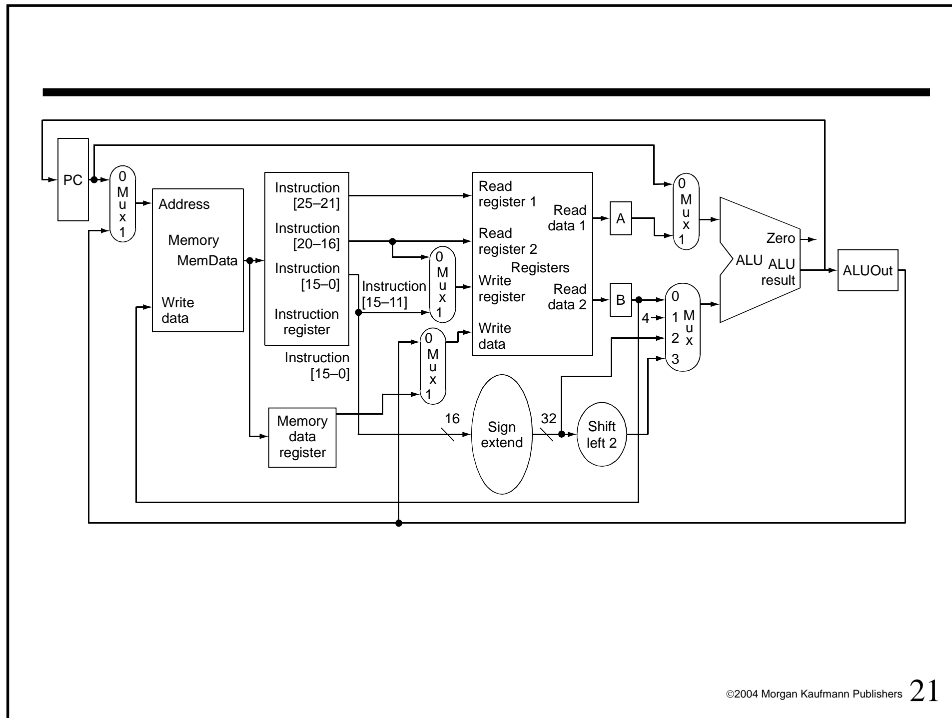


Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

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

- We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)



©2004 Morgan Kaufmann Publishers

Step 3 (instruction dependent)

- **ALU is performing one of three functions, based on instruction type**
- **Memory Reference:**

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

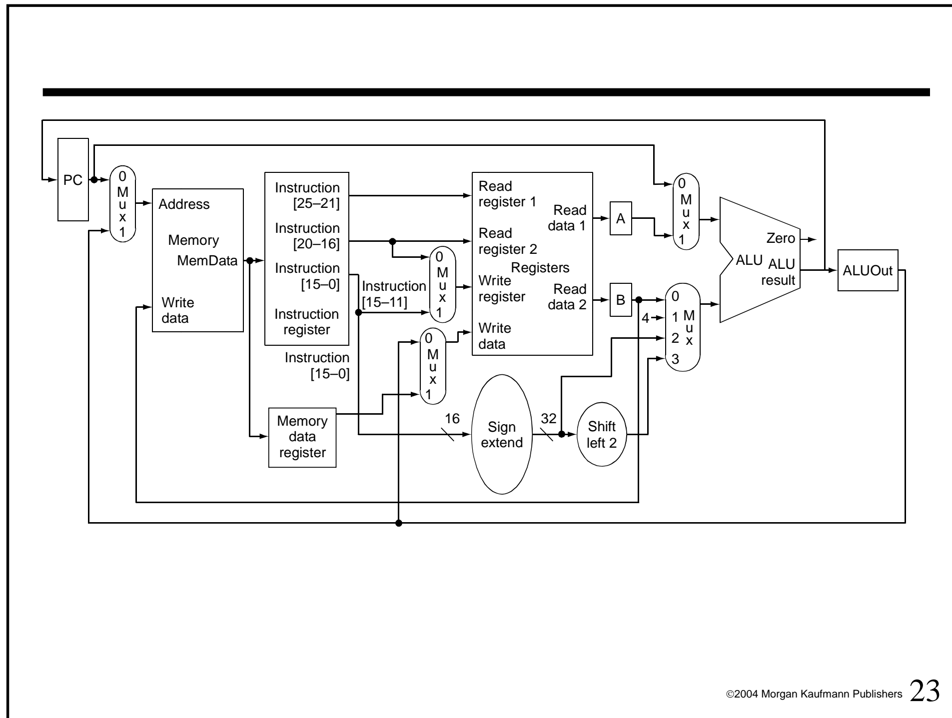
- **R-type:**

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

- **Branch:**

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

©2004 Morgan Kaufmann Publishers



Step 4 (R-type or memory-access)

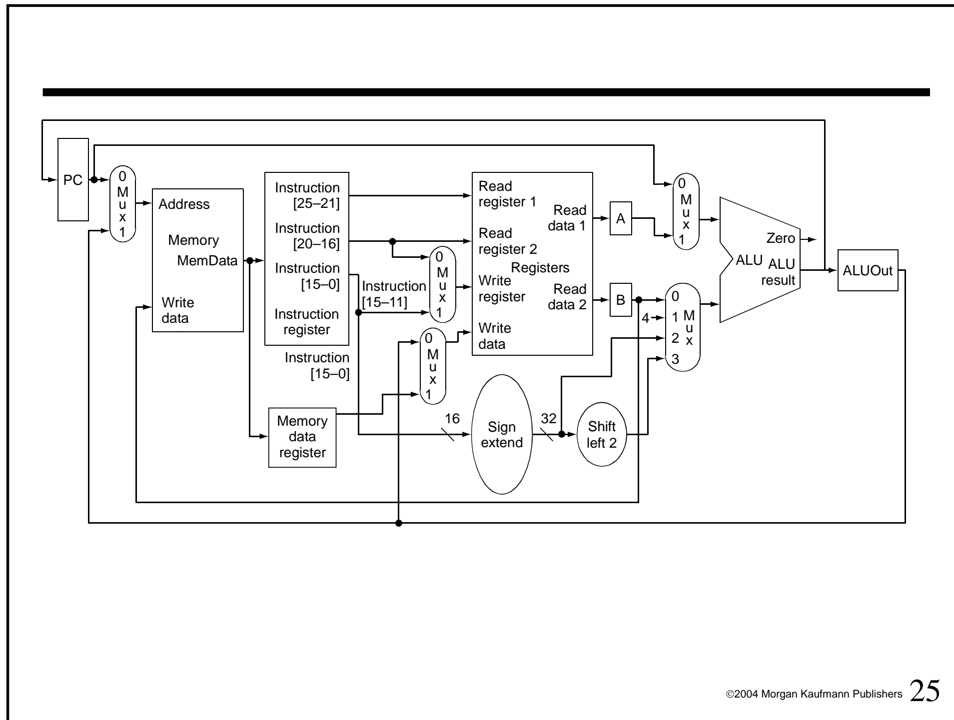
- **Loads and stores access memory**

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

- **R-type instructions finish**

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

The write actually takes place at the end of the cycle on the edge



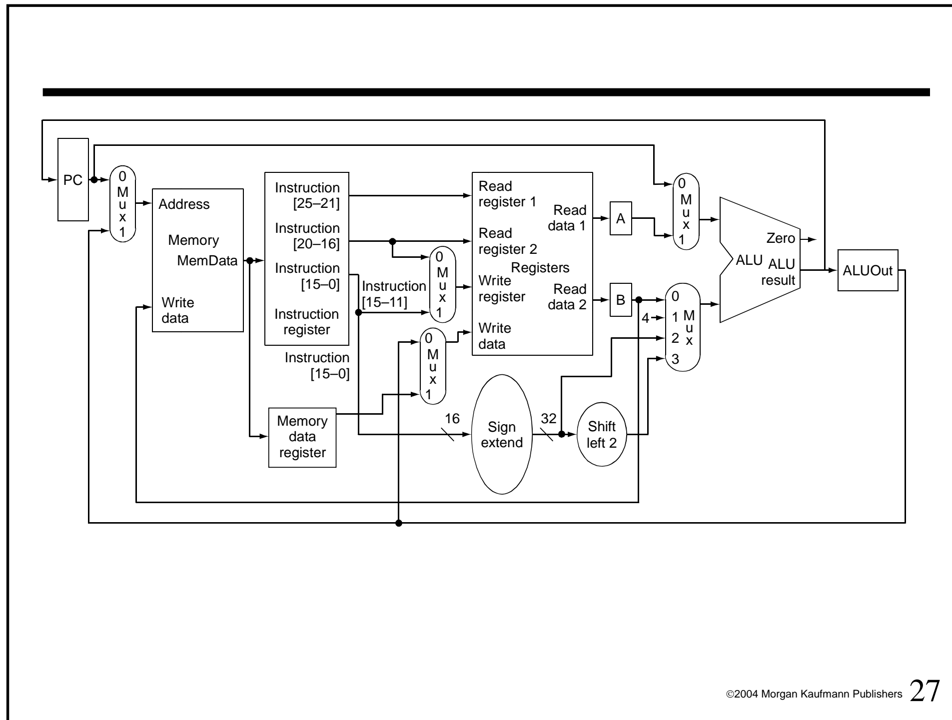
©2004 Morgan Kaufmann Publishers

Write-back step

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

Which instruction needs this?

©2004 Morgan Kaufmann Publishers



Summary:

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		

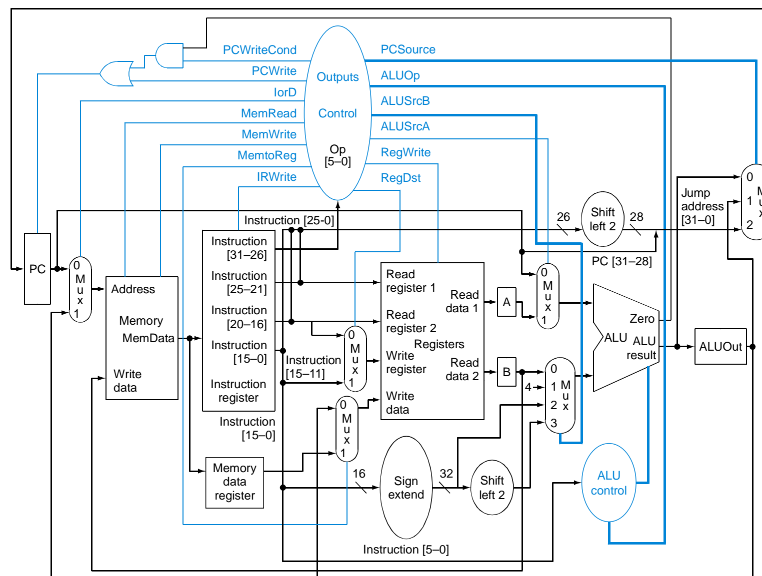
FIGURE 5.30 Summary of the steps taken to execute any instruction class. Instructions take from three to five execution steps. The first two steps are independent of the instruction class. After these steps, an instruction takes from one to three more cycles to complete, depending on the instruction 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.

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?**



Pipeline example: Doing Laundry

Summary

- Datapath control
- Multicycle machine

- Next Time
  - Pipelining

- Reading assignment – P&H 6.1 – 6.3