# ALU Control

- **ALU's operation based on instruction type and function code**
  - e.g., what should the ALU do with any instruction
- **Example: lw $1, 100($2)**

| 35 | 2 | 1 | 100 |
|----|---|---|-----|

| op | rs | rt | 16 bit offset |
|----|----|----|---------------|

- **ALU control input**

```
000     AND
001     OR
010     add
110     subtract
111     set-on-less-than
```

- **Why is the code for subtract 110 and not 011?**

1

---

# ALU Control Information

- **Must describe hardware to compute 3-bit ALU conrol input**
  - **given instruction type**
    - **00 = lw, sw**
    - **01 = beq,**        ALUOp
    - **10 = arithmetic**   computed from instruction type
    - **11 = Jump**
  - **function code for arithmetic**
- **Control can be described using a truth table:**

| ALUOp | | Funct field | | | | | | Operation |
|-------|-------|----|----|----|----|----|----|-----------|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | |
| 0 | 0 | X | X | X | X | X | X | 010 |
| X | 1 | X | X | X | X | X | X | 110 |
| 1 | X | X | X | 0 | 0 | 0 | 0 | 010 |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 110 |
| 1 | X | X | X | 0 | 1 | 0 | 0 | 000 |
| 1 | X | X | X | 0 | 1 | 0 | 1 | 001 |
| 1 | X | X | X | 1 | 0 | 1 | 0 | 111 |

2

# Implementation of ALU Control

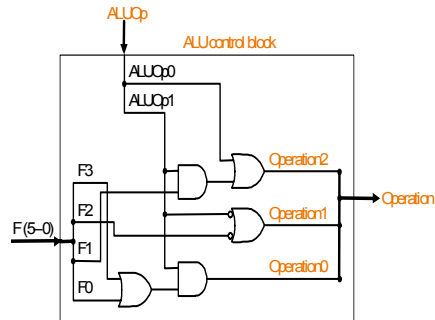- **Simple collection of gates to realize the truth tables**

For operation2 = 1

| ALU op1 | ALU op0 | F5 | F4 | F3 | F2 | F1 | F0 |
|---|---|---|---|---|---|---|---|
| X | 1 | X | X | X | X | X | X |
| 1 | X | X | X | X | X | 1 | X |

For operation1 = 1

| ALU op1 | ALU op0 | F5 | F4 | F3 | F2 | F1 | F0 |
|---|---|---|---|---|---|---|---|
| 0 | X | X | X | X | X | X | X |
| X | X | X | X | X | 0 | X | X |

For operation1 = 1

| ALU op1 | ALU op0 | F5 | F4 | F3 | F2 | F1 | F0 |
|---|---|---|---|---|---|---|---|
| 1 | X | X | X | X | X | X | 1 |
| 1 | X | X | X | 1 | X | X | X |



*How to handle X?*

3

---

# Implementation of Main Control

| Instruction | RegDst | ALUSrc | Memto- Reg | Reg Write | Mem Read | Mem Write | Branch | ALUOp1 | ALUp0 |
|---|---|---|---|---|---|---|---|---|---|
| 000000 R- | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 100000 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 |



*How to program PLA?*

4

# Datapath with Control and Jump Instruction



*Where are the changes?*

5

# Instruction Format
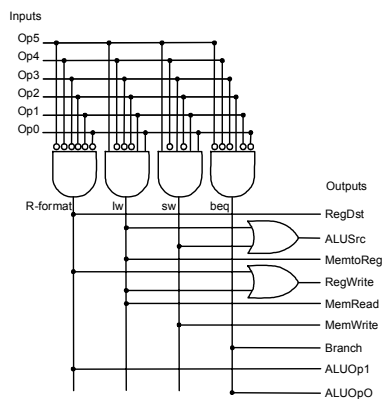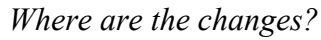
| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| LW | | REG 1 | | REG 2 | | LOAD ADDRESS | | | | OFFSET | |

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| SW | | REG 1 | | REG 2 | | STORE ADDRESS | | | | OFFSET | |

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| R-TYPE | | REG 1 | | REG 2 | | DST | | SHIFT AMOUNT | | ADD/AND/OR/SLT | |

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| BEQ/BNE | | REG 1 | | REG 2 | | BRANCH ADDRESS | | | | OFFSET | |

| 31 | 26 | 25 | 21 | 20 | 16 | 15 | 11 | 10 | 6 | 5 | 0 |
|----|----|----|----|----|----|----|----|----|----|----|----|
| JUMP | | JUMP | | | | | ADDRESS | | | | |

*How does the format help performance?*

6

# Timing: Single Cycle Implementation

- **Calculate cycle time assuming negligible delays except:**
  - **memory (2ns), ALU and adders (2ns), register file access (1ns)**
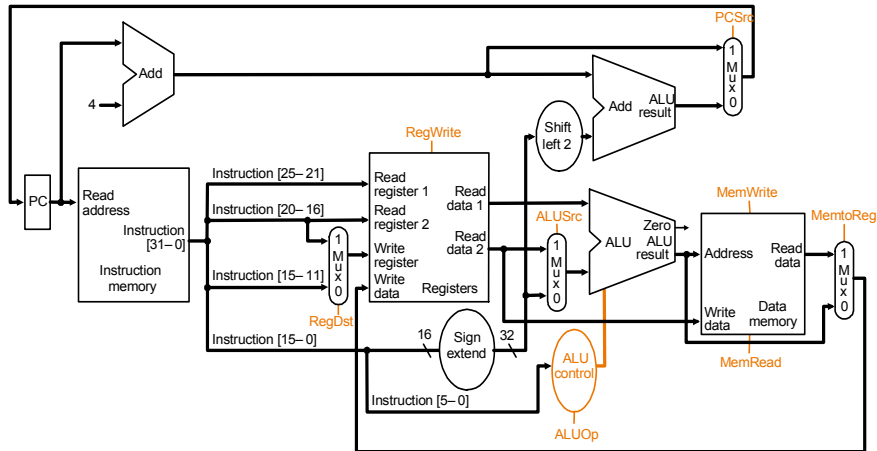
# Where we are headed

- **Single Cycle Problems:**
  - **what if we had a more complicated instruction like floating point?**
  - **wasteful of area**
- **One Solution:**
  - **use a "smaller" cycle time**
  - **have different instructions take different numbers of cycles**
  - **a "multicycle" datapath:**
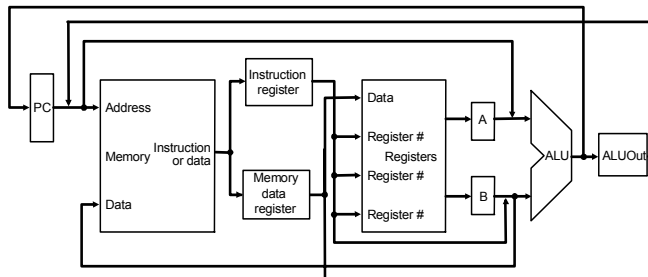
## Operation for Each Instruction

| LW: | SW: | R-Type: | BR-Type: | JMP-Type: |
|---|---|---|---|---|
| 1. READ INST | 1. READ INST | 1. READ INST | 1. READ INST | 1. READ INST |
| 2. READ REG 1<br>*READ REG 2* | 2. READ REG 1<br>READ REG 2 | 2. READ REG 1<br>READ REG 2 | 2. READ REG 1<br>READ REG 2 | 2. |
| 3. ADD REG 1 +<br>OFFSET | 3. ADD REG 1 +<br>OFFSET | 3. OPERATE on<br>REG 1 / REG 2 | 3. SUB REG 2<br>from REG 1 | 3. |
| 4. READ MEM | 4. WRITE MEM | 4. | 4. | 4. |
| 5. WRITE REG2 | 5. | 5. WRITE DST | 5. | 5. |

9

## Multicycle Approach

- **We will be reusing functional units**
  - **Break up the instruction execution in smaller steps**
  - **Each functional unit is used for a specific purpose in one cycle**
  - **Balance the work load**
  - **ALU used to compute address and to increment PC**
  - **Memory used for instruction and data**
- **At the end of cycle, store results to be used again**
  - **Need additional registers**
- **Our control signals will not be determined solely by instruction**
  - **e.g., what should the ALU do for a "subtract" instruction?**
- **We'll use a finite state machine for control**

10

# Review:  finite state machines

- **Finite state machines:**
  - **a set of states and**
  - **next state function (determined by current state and the input)**
  - **output function (determined by current state and possibly input)**
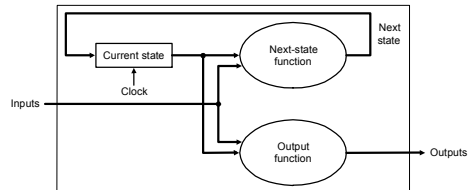


  - **We'll use a Moore machine (output based only on current state)**

# Review:  finite state machines

- **Example:**

  **B. 21** *A friend would like you to build an "electronic eye" for use as a fake security device.  The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which, if asserted, indicate that a light should be on.  Only one light is on at a time, and the light "moves" from left to right and then from right to left, thus scaring away thieves who believe that the device is monitoring their activity.  Draw the graphical representation for the finite state machine used to specify the electronic eye.  Note that the rate of the eye's movement will be controlled by the clock speed (which should not be too great) and that there are essentially no inputs.*

## Multi-Cycle DataPath Operation



13

## 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!*

14

## 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?*

---

## 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)**

# 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;

17

# 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*

18

# Step 5: Write-back step

- `Reg[IR[20-16]]= MDR;`

*What about all the other instructions?*

---

# Summary:

| Step name | Action for R-type instructions | Action for memory-reference instructions | Action for branches | Action for jumps |
|---|---|---|---|---|
| Instruction fetch | | IR = Memory[PC]<br>PC = PC + 4 | | |
| Instruction decode/register fetch | | A = Reg [IR[25-21]]<br>B = Reg [IR[20-16]]<br>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) then PC = ALUOut | PC = PC [31-28] II (IR[25-0]<<2) |
| Memory access or R-type completion | Reg [IR[15-11]] = ALUOut | Load: MDR = Memory[ALUOut]<br>or<br>Store: Memory [ALUOut] = B | | |
| Memory read completion | | Load: Reg[IR[20-16]] = MDR | | |