# Lecture 8:
# Control

COS / ELE 375

Computer Architecture and Organization

Princeton University
Fall 2015

Prof. David August

## Datapath and Control

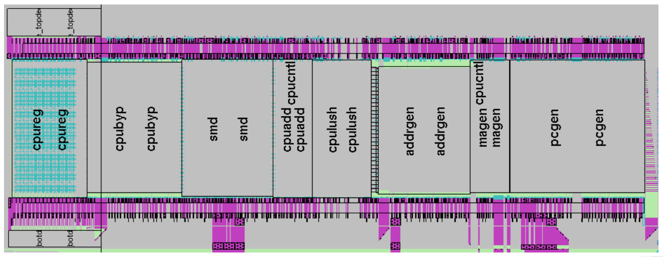
<u>Datapath</u>

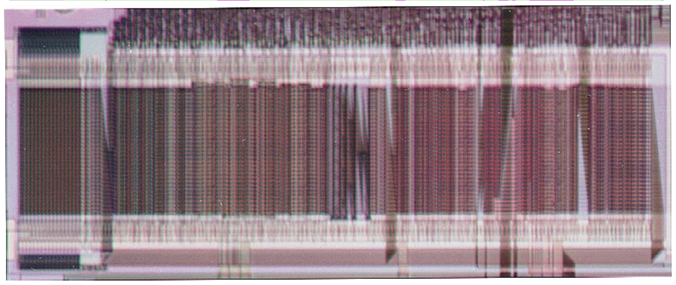
The collection of state elements, computation elements, and interconnections that together provide a conduit for the flow and transformation of data in the processor during execution. - DIA

<u>Control</u>

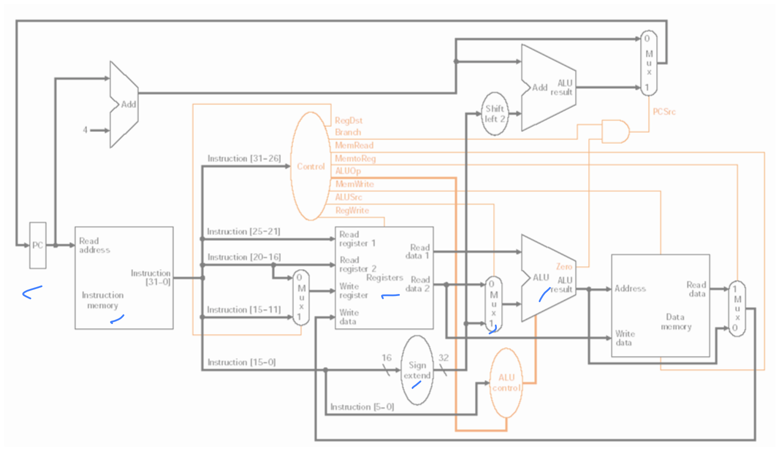
The component of the processor that commands the datapath, memory, and I/O devices according to the instructions of the program. – P&H

## A Real MIPS Datapath



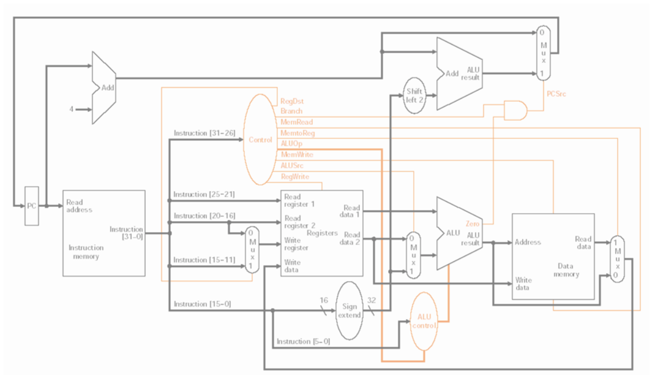


## Datapath with Control



## Datapath with Control

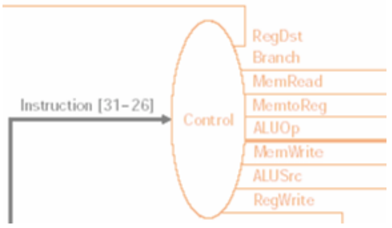
- Memory reference: `lw, sw`
- Arithmetic/logical: `add, sub, and, or, slt`
- Control flow: `beq, j (see book)`

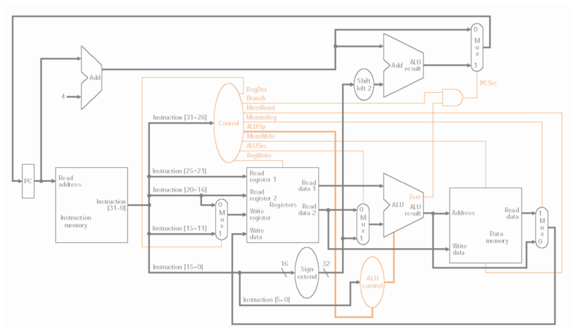


## The Control Unit

- Generates Control Signals

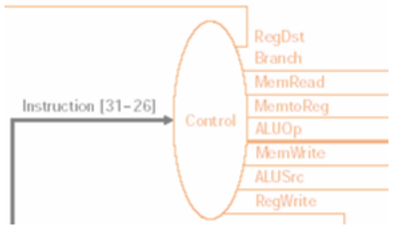
  RegDst, Branch, MemRead, MemtoReg, ALUOp, MemWrite, ALUSrc, RegWrite





## The Control Unit

- Generates Control Signals
- Uses Op Field [31-26]



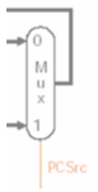
|  | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|---|
| **R:** | op | rs | rt | rd | shamt | funct |
| **I:** | op | rs | rt | address / immediate | | |
| **J:** | op | target address | | | | |

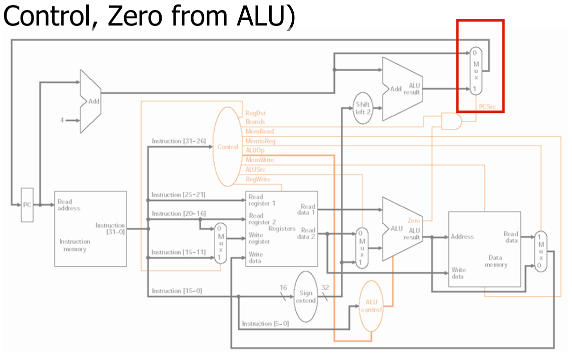
## Control Signals

PCSrc

- True: PC = SignExt($Imm_{16}$) << 2 + PC + 4
- False: PC = PC + 4
- PCSrc = Branch & Zero

  (Branch from Control, Zero from ALU)

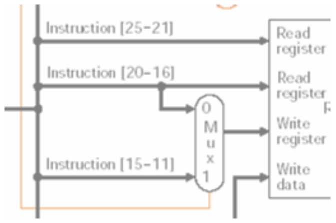


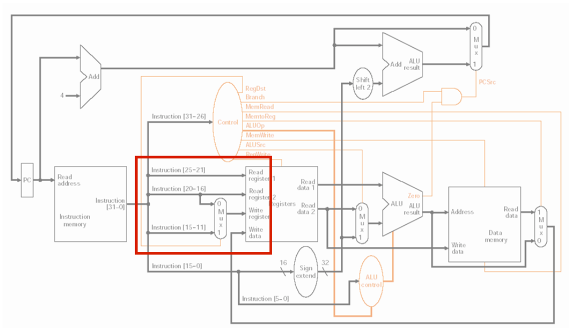


## Control Signals

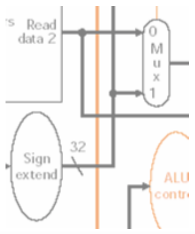
RegDst

- True: WriteReg = Inst[15-11]
- False: WriteReg = Inst[20-16]



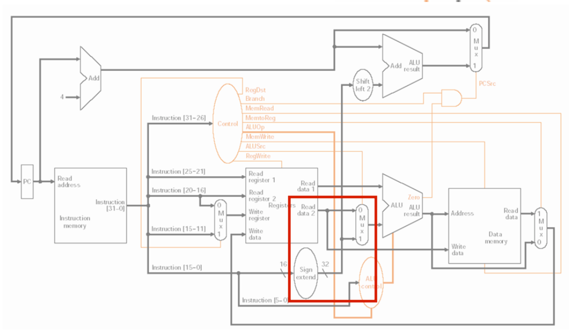


## Control Signals



ALUSrc

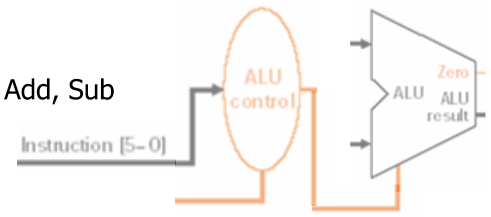
- True: SignExt($\text{Imm}_{16}$)
- False: RegisterData2



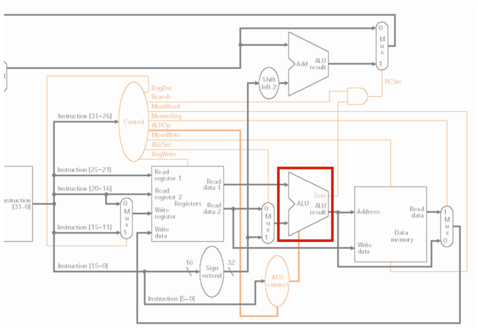
## Control Signals

ALUControl

- Values: And, Or, Add, Sub
- Uses Funct Field



| ALUControl | Function |
|---|---|
| 000 | AND |
| 001 | OR |
| 010 | add |
| 110 | subtract |
| 111 | set on less than |

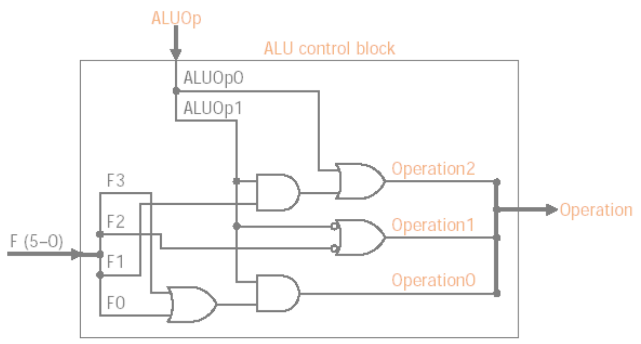


## ALUControl

| ALUOp | | Funct field | | | | | | Operation | |
|---|---|---|---|---|---|---|---|---|---|
| ALUOp1 | ALUOp0 | F5 | F4 | F3 | F2 | F1 | F0 | | |
| 0 | 0 | X | X | X | X | X | X | 010 | lw/sw |
| X | 1 | X | X | X | X | X | X | 110 | beq |
| 1 | X | X | X | 0 | 0 | 0 | 0 | 010 | R-Type |
| 1 | X | X | X | 0 | 0 | 1 | 0 | 110 | R-Type |
| 1 | X | X | X | 0 | 1 | 0 | 0 | 000 | R-Type |
| 1 | X | X | X | 0 | 1 | 0 | 1 | 001 | R-Type |
| 1 | X | X | X | 1 | 0 | 1 | 0 | 111 | R-Type |

| ALUControl | Function |
|---|---|
| 000 | AND |
| 001 | OR |
| 010 | add |
| 110 | subtract |
| 111 | set on less than |

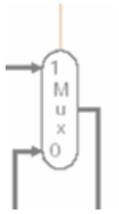
## ALU Control Combinational Implementation

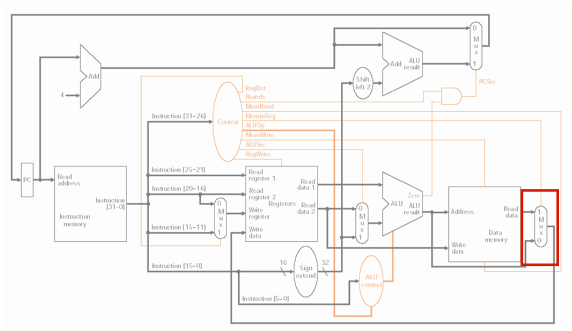


## Control Signals

MemtoReg

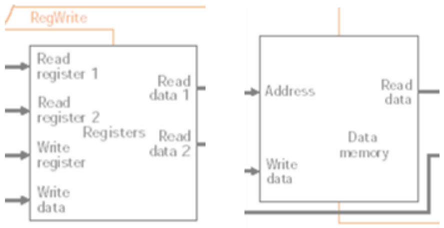
- True: RegisterWriteData = MemReadData
- False: RegisterWriteData = ALUResult

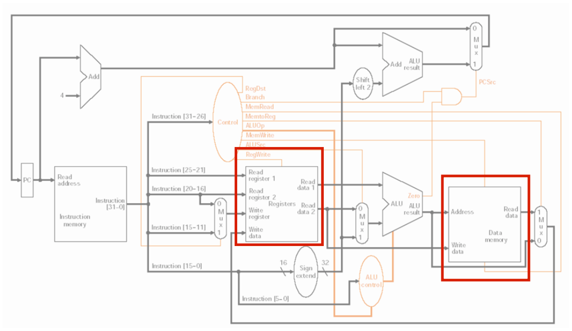




## Control Signals

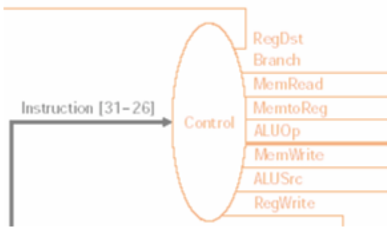
MemRead
MemWrite
RegWrite





## The Control Unit

- Generates Control Signals
- Uses Op Field [31-26]

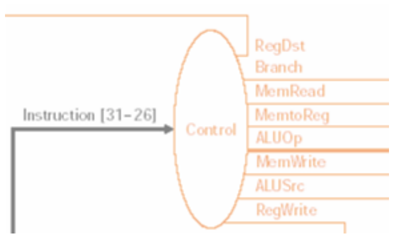


| | 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |
|---|---|---|---|---|---|---|
| **R:** | op | rs | rt | rd | shamt | funct |
| **I:** | op | rs | rt | address / immediate | | |
| **J:** | op | target address | | | | |

## Control Design: Inputs

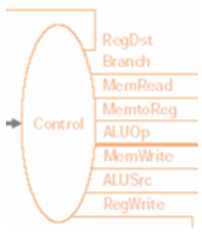
| Opcode | op5 | op4 | op3 | op2 | op1 | op0 | Value |
|---|---|---|---|---|---|---|---|
| R-Format | 0 | 0 | 0 | 0 | 0 | 0 | $0_{10}$ |
| lw | 1 | 0 | 0 | 0 | 1 | 1 | $35_{10}$ |
| sw | 1 | 0 | 1 | 0 | 1 | 1 | $43_{10}$ |
| beq | 0 | 0 | 0 | 1 | 0 | 0 | $4_{10}$ |

Where were these values chosen?



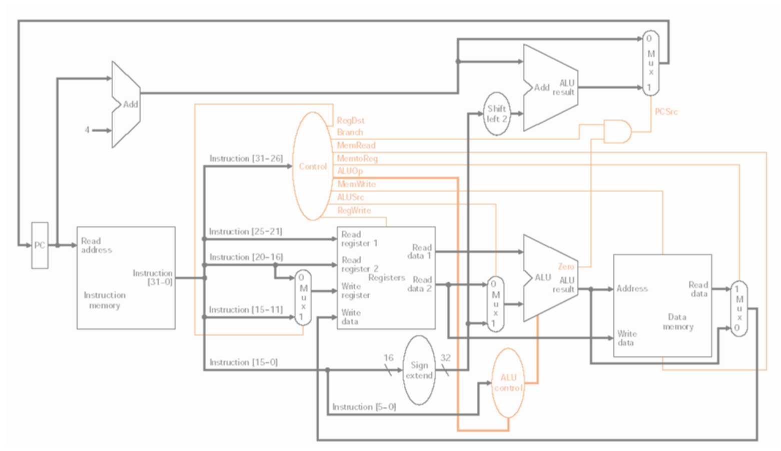
## Control Design: Outputs

| Signal | R-Format | lw | sw | beq |
|---|---|---|---|---|
| RegDst | | | | |
| ALUSrc | | | | |
| MemtoReg | | | | |
| RegWrite | | | | |
| MemRead | | | | |
| MemWrite | | | | |
| Branch | | | | |
| ALUOp1 | | | | |
| ALUOp2 | | | | |

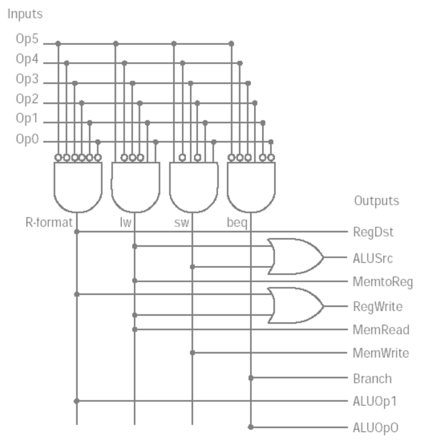


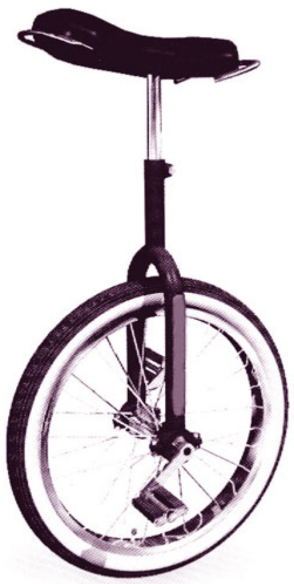
You fill it out with: 0, 1, X (don’t care) - check with book

## Datapath with Control

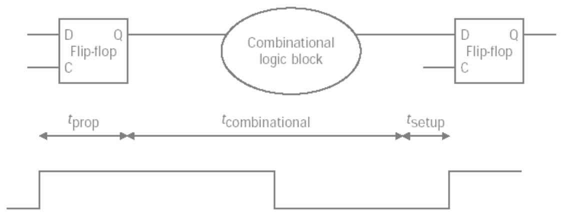


## Combinational Implementation





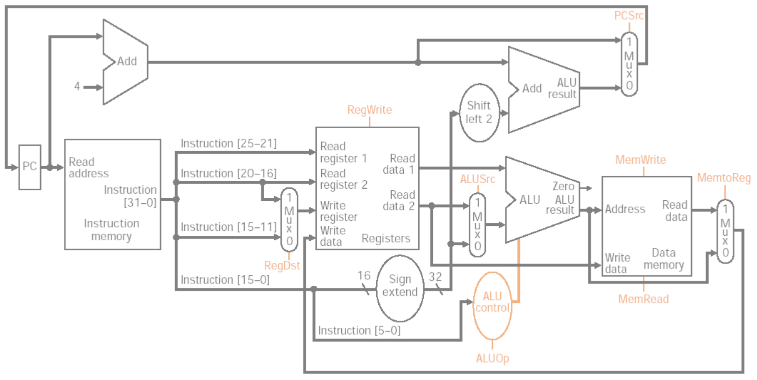
## Single Cycle Implementation



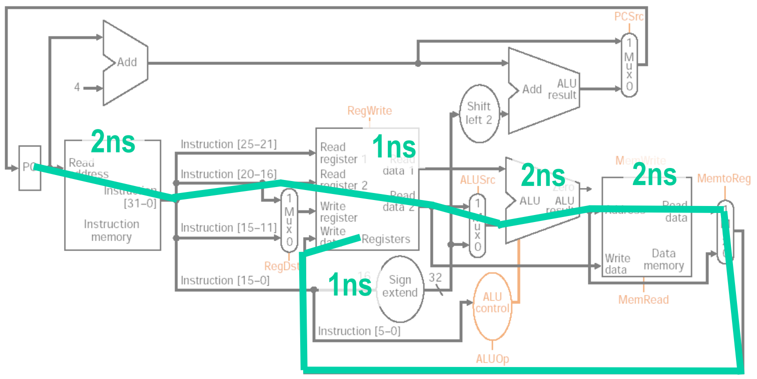
- All of the logic is combinational
- We wait for everything to settle down
  - ALU might not produce “right answer” right away
  - Use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path

## Single Cycle Implementation

Calculate cycle time assuming negligible delays except:
memory (2ns), ALU/adders (2ns), register file access (1ns)



## Single Cycle Implementation



Load Word is longest running instruction
(prove to yourself by computing time for sw, br, R-type)

memory (2ns), ALU/adders (2ns), register file access (1ns)

2ns + 1ns + 2ns + 2ns + 1ns = 8ns

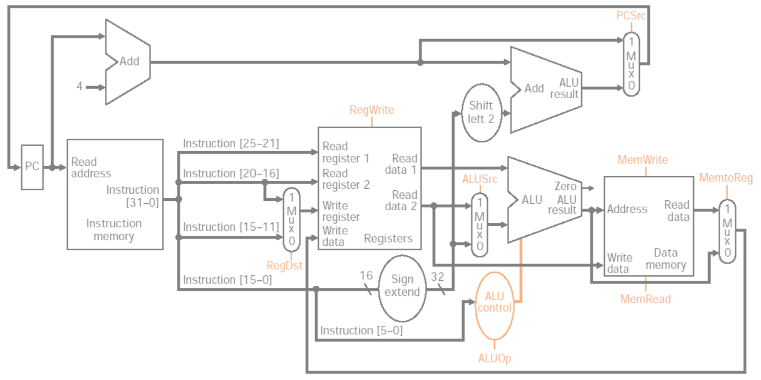
## Magic Moment!

At this point, you should be able to design a computer!

1. Analyze instruction set for datapath requirements
2. Select set of datapath components
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control signals
5. Assemble the control logic
6. Set clock rate

## What's Wrong with Single Cycle?

Calculate cycle time assuming negligible delays except:
memory (2ns), ALU/adders (2ns), register file access (1ns)

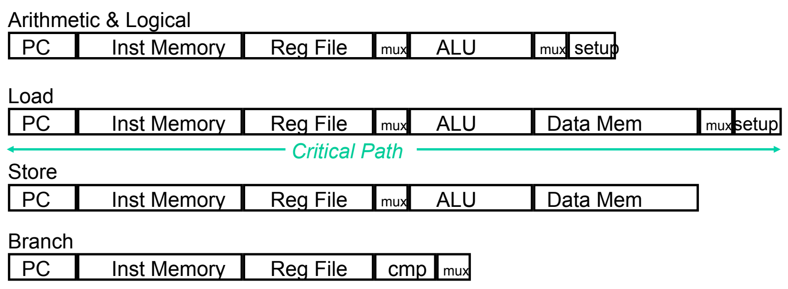


## What's Wrong with Single Cycle?

memory (2ns), ALU/adders (2ns), register file access (1ns)

| Inst. | Inst. Mem | Reg. Read | ALU | Data Mem | Reg. Write | Total |
|---|---|---|---|---|---|---|
| ALU | 2 | 1 | 2 | | 1 | 6 |
| lw | 2 | 1 | 2 | 2 | 1 | 8 |
| sw | 2 | 1 | 2 | 2 | | 7 |
| br | 2 | 1 | 2 | | | 5 |

## What's Wrong with Single Cycle?


Arithmetic & Logical
PC | Inst Memory | Reg File | mux | ALU | mux | setup
Load
PC | Inst Memory | Reg File | mux | ALU | Data Mem | mux | setup
Critical Path
Store
PC | Inst Memory | Reg File | mux | ALU | Data Mem
Branch
PC | Inst Memory | Reg File | cmp | mux

- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not so nice as our idealized memory
  (cannot always get the job done in fixed amount of time)

## How Bad?

- Assume: 100 instructions executed
  - 25% of instructions are loads (8ns),
  - 10% of instructions are stores (7ns),
  - 45% of instructions are adds (6ns), and
  - 20% of instructions are branches (5ns).
- Single-cycle execution:

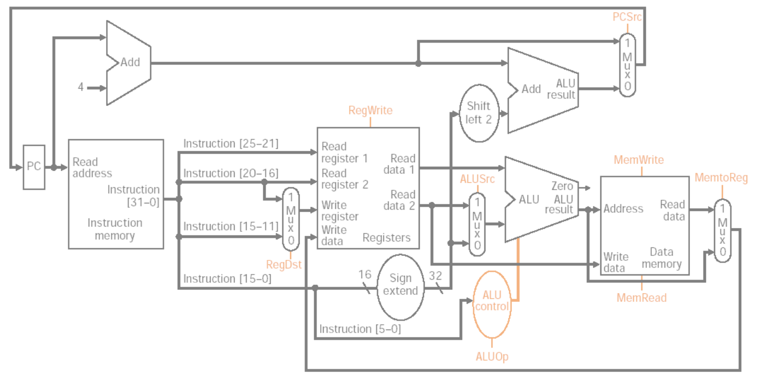
  100 * 8ns = 800 ns
- Optimal execution:

  25*8ns + 10*7ns + 45*6ns + 20*5ns = 640 ns

  Speedup = 800/640 = 1.25

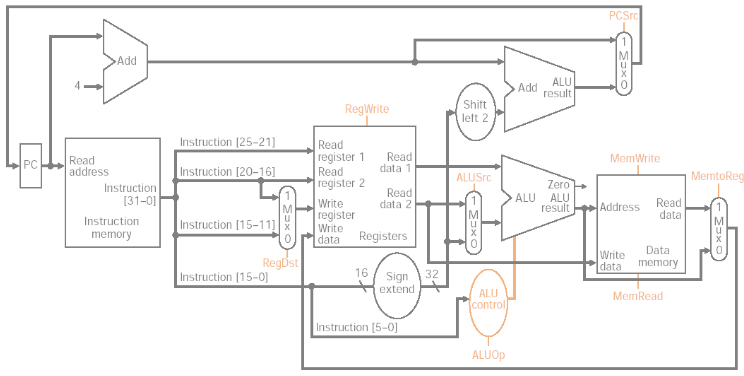
## Other Problems

- Instruction and Data Memory are the same
- Some units hold value long after job is complete
  - Could reuse ALU for example
  - Underutilized resources (wasteful of area/power)



## Other Problems

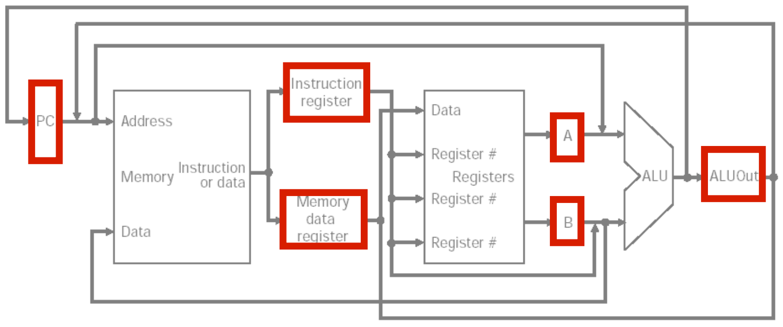
- What about floating point or other VERY LONG instructions?



## Multicycle Approach

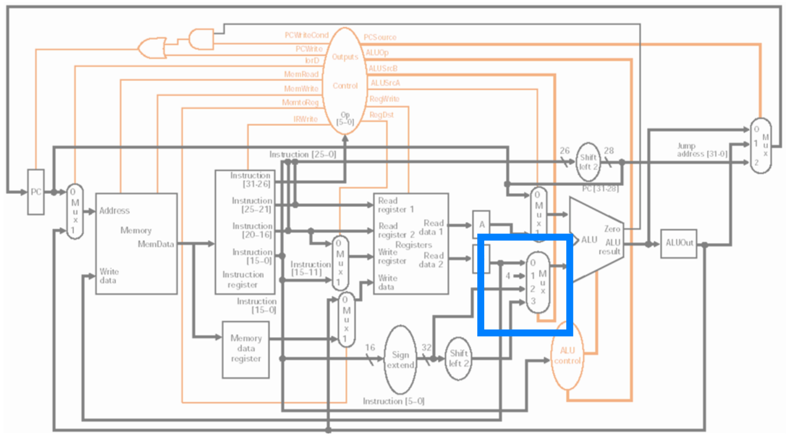
- Break up the instructions into steps, one per cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional “internal” registers to hold values between cycles

## Multicycle Approach



Architectural vs. Microarchitectural State?
Register File?

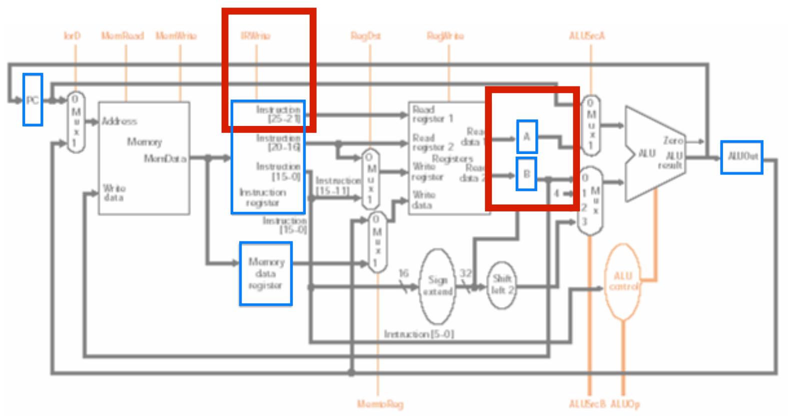
## Multicycle Control
## More Mux Magic



## Multicycle Control

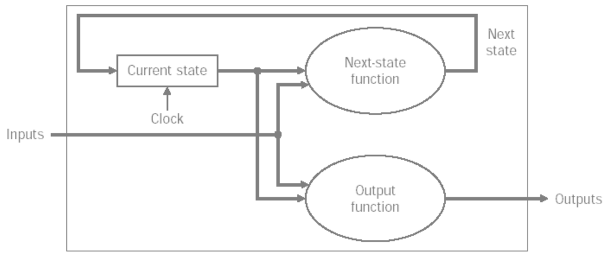
- Reuse datapath components
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Control signals not determined solely by instruction
  - What should the ALU do for a “subtract” instruction?

## Multicycle Control



## Use Finite State Machine For Control

- Set of states
- 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)
- To derive FSM, need steps of the instruction!

## Five Execution Steps

1. Instruction Fetch
2. Instruction Decode and Register Fetch
3. Execution, Memory Address Computation, or Branch Completion
4. Memory Access or R-type instruction completion
5. Write-Back Step

*INSTRUCTIONS TAKE FROM 3 - 5 STEPS/CYCLES!*

## Step 1: Instruction Fetch

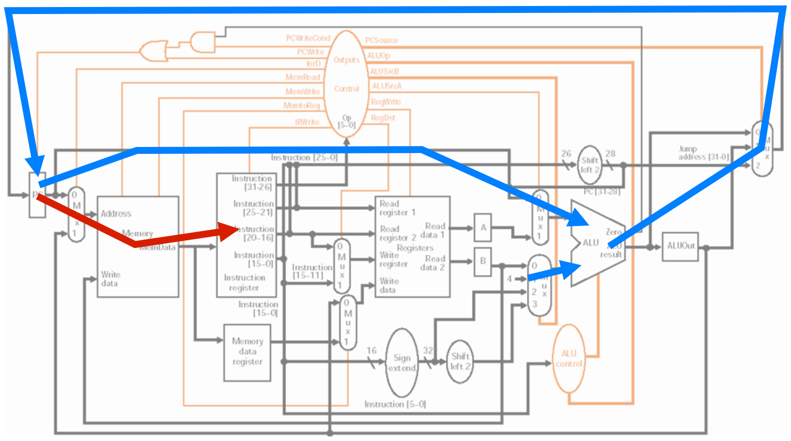
- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

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

*Can we figure out the values of the control signals?*
*ALU? IR?*

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

## Step 1: Instruction Fetch



**IR = Memory[PC];**

**PC = PC + 4;**

## Step 2:
## Instruction Decode & Register Fetch

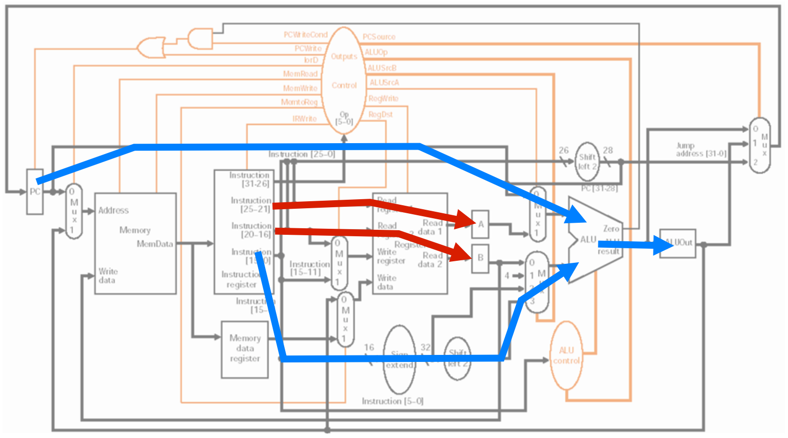
- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

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

*Note: Again, no control lines based on the instruction type (busy "decoding" it in control logic)*

*Reg? ALU? What if not branch instruction?*

## Step 2:



**A = Reg[IR[25-21]];**
**B = Reg[IR[20-16]];**
**ALUOut = PC + (SgnExt(IR[15-0]) << 2);**

## Step 3: (Instruction Dependent)

ALU performs a function 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;
```

*ALU Utilization rate to step 3?*

## 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;
```

*Writes takes place at the end of the cycle on the edge*

## Step 5: Write-Back

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

*What about all the other instructions?*

## Summary Of Steps

| 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 | | |

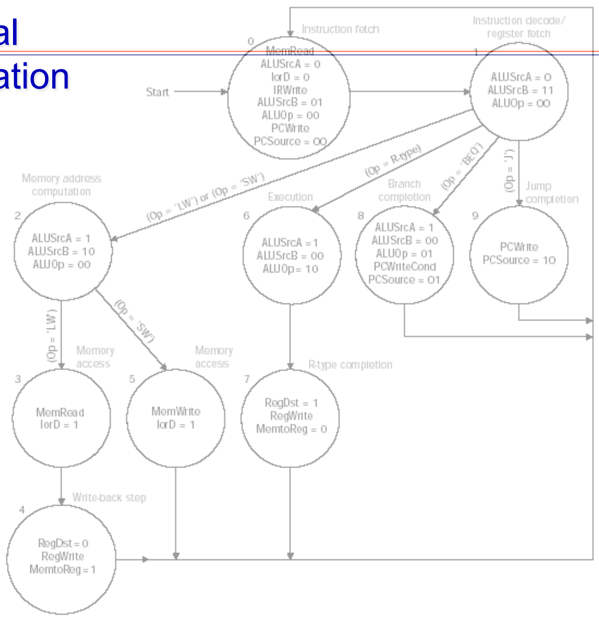
## Implementing the Control

- Values of control signals are dependent upon:
  1. instruction is being executed
  2. step is being performed

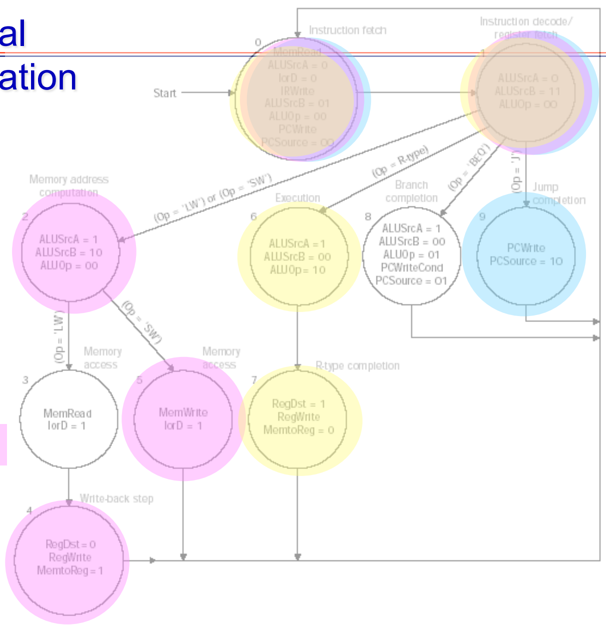
- Using steps, specify a finite state machine
  - specify the finite state machine graphically, or
  - use “microprogramming” (more on this next time)

- Implementation can be derived from specification

## Graphical Specification of FSM


Instruction fetch
Instruction decode/ register fetch
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
Start
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
Execution
Branch completion
Jump completion
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Write-back step
4
RegDst = 0 RegWrite MemtoReg = 1

## Graphical Specification of FSM


Instruction fetch
Instruction decode/ register fetch
0
MemRead ALUSrcA = 0 IorD = 0 IRWrite ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
Start
1
ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
(Op = 'LW') or (Op = 'SW')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
Memory address computation
Execution
Branch completion
Jump completion
2
ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
6
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 10
8
ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
9
PCWrite PCSource = 10
(Op = 'LW')
(Op = 'SW')
Memory access
Memory access
R-type completion
3
MemRead IorD = 1
5
MemWrite IorD = 1
7
RegDst = 1 RegWrite MemtoReg = 0
Write-back step
4
RegDst = 0 RegWrite MemtoReg = 1

Add r1, r2, r3

sw r1, 100(r2)

jr r31

## Performance Evaluation

- What is the average CPI?
  - state diagram gives CPI for each instruction type
  - workload gives frequency of each type

| Type | $CPI_i$ for type | Frequency | $CPI_i$ x $freqI_i$ |
|---|---|---|---|
| Arith/Logic | 4 | 40% | 1.6 |
| Load | 5 | 30% | 1.5 |
| Store | 4 | 10% | 0.4 |
| branch | 3 | 20% | 0.6 |
| | | Average CPI: | 4.1 |

## Single/Multi-Cycle Summary

$$CPU\,time = \frac{Instructions}{Program} X \frac{Cycles}{Instruction} X \frac{Seconds}{Cycle}$$

Single Cycle Datapath:
- CPI = 1!!
- Long cycle time ☹ (critical path based)

Multiple Cycle Datapath:
- Short cycle time!!
- CPI = 3-5 ☹

Can we achieve a CPI of 1 (on average)
with a clock cycle time
similar to the multiple cycle datapath?