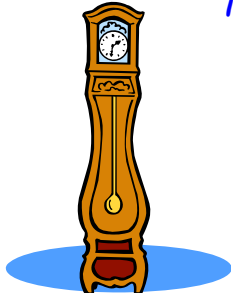


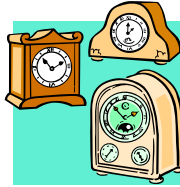
## Multiple Clock Cycle CPU



CSE 141

or

*Breaking Up Is Hard To Do*



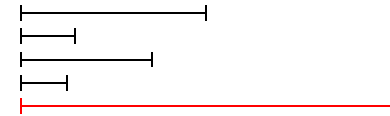
Dean Tullsen

## Why a Multiple Clock Cycle CPU?

- the problem => single-cycle cpu has a cycle time long enough to complete the **longest** instruction in the machine
- the solution => break up execution into smaller tasks, each task taking a cycle, different instructions requiring different numbers of cycles or tasks
- other advantages => reuse of functional units (e.g., alu, memory)

- $ET = IC * CPI * CT$

CSE 141



Dean Tullsen

## Breaking Execution Into Clock Cycles

- We will have five execution steps (not all instructions use all five)
  - fetch
  - decode & register fetch
  - execute
  - memory access
  - write-back
- We will use Register-Transfer-Language (RTL) to describe these steps

CSE 141

Dean Tullsen

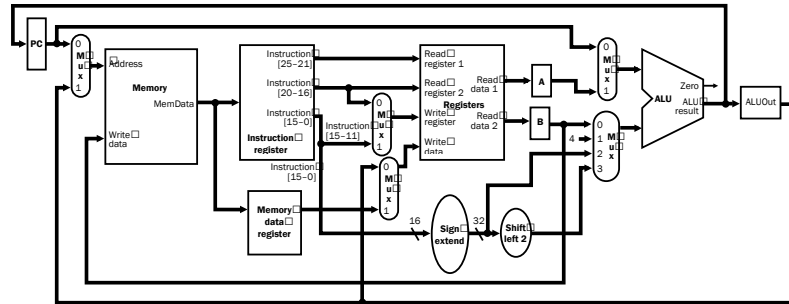
## Breaking Execution Into Clock Cycles

- Introduces extra registers when:
  - signal is **computed** in one clock cycle and **used** in another, AND
  - the inputs to the functional block that outputs this signal can **change** before the signal is written into a state element.
- Significantly complicates control. **Why?**
- The goal is to **balance** the amount of work done each cycle.

CSE 141

Dean Tullsen

## Multicycle datapath



CSE 141

Dean Tullsen

## 1. Fetch

$$IR = \text{Mem}[PC]$$

$$PC = PC + 4$$

(may not be final value of PC)

CSE 141

Dean Tullsen

## 2. Instruction Decode and Register Fetch

$$A = \text{Reg}[IR[25-21]]$$

$$B = \text{Reg}[IR[20-16]]$$

$$\text{ALUOut} = PC + (\text{sign-extend}(IR[15-0]) \ll 2)$$

- compute target before we know if it will be used (may not be branch, branch may not be taken)
- *ALUOut* is a new state element (temp register)
- everything up to this point must be *Instruction-independent*, because we still haven't decoded the instruction.
- everything instruction (opcode)-dependent from here on.

CSE 141

Dean Tullsen

## 3. Execution, memory address computation, or branch completion

- Memory reference (load or store)
 
$$\text{ALUOut} = A + \text{sign-extend}(IR[15-0])$$
- R-type
 
$$\text{ALUOut} = A \text{ op } B$$
- Branch
 
$$\text{if } (A == B) \text{ PC} = \text{ALUOut}$$

At this point, Branch is complete, and we start over; others require more cycles.

CSE 141

Dean Tullsen

## 4. Memory access or R-type completion

- Memory reference
  - load
    - $MDR = Mem[ALUout]$
  - store
    - $Mem[ALUout] = B$
- R-type
  - $Reg[IR[15-11]] = ALUout$

*R-type is complete, store is complete.*

## 5. Memory Write-Back

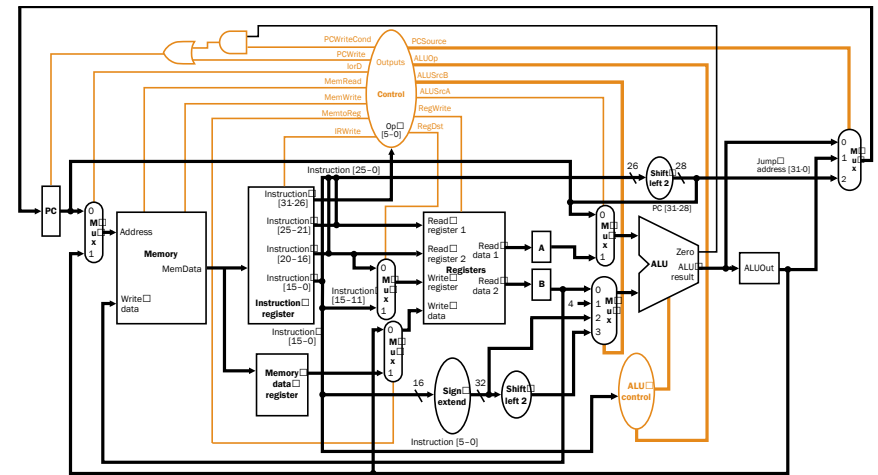
$Reg[IR[20-16]] = MDR$

*load is complete*

## Summary of execution steps

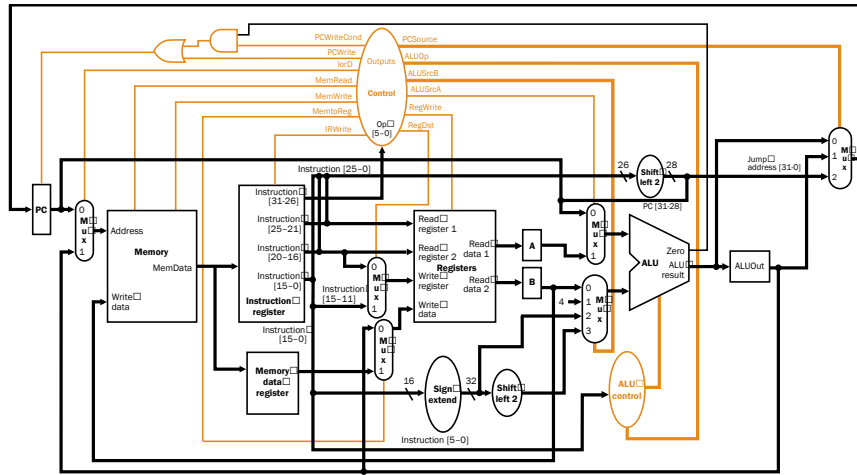
Step	R-type	Memory	Branch
Instruction Fetch		IR = Mem[PC] PC = PC + 4	
Instruction Decode/ register fetch		A = Reg[IR[25-21]] B = Reg[IR[20-16]] ALUout = PC + (sign-extend(IR[15-0]) << 2)	
Execution, address computation, branch completion	ALUout = A op B	ALUout = A + sign- extend(IR[15-0])	if (A==B) then PC=ALUout
Memory access or R- type completion	Reg[IR[15-11]] = ALUout	memory-data = Mem[ALUout] or Mem[ALUout]= B	
Write-back		Reg[IR[20-16]] = memory-data	

## Complete Multicycle Datapath



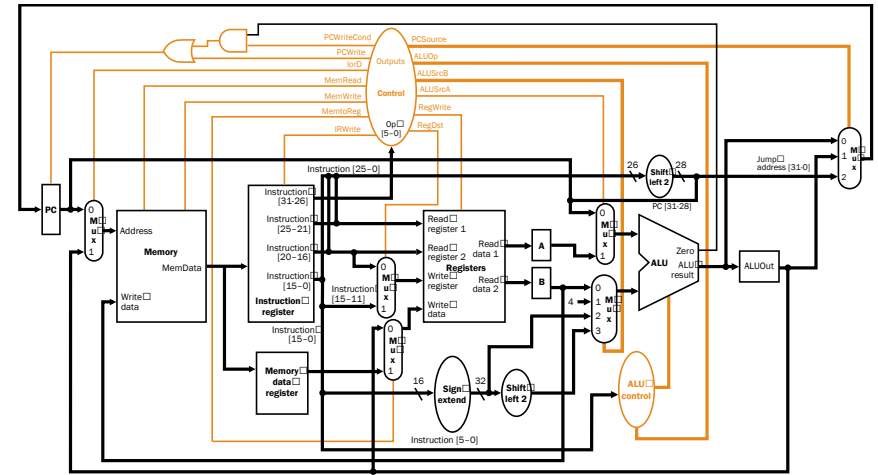
(support for what instruction just got added?)

## 1. Instruction Fetch



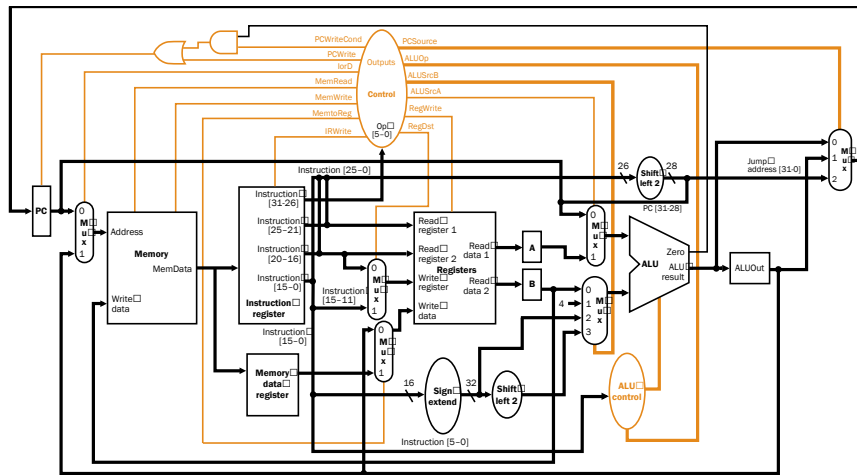
IR = Memory[PC]  
PC = PC + 4

## 2. Instruction Decode and Reg Fetch



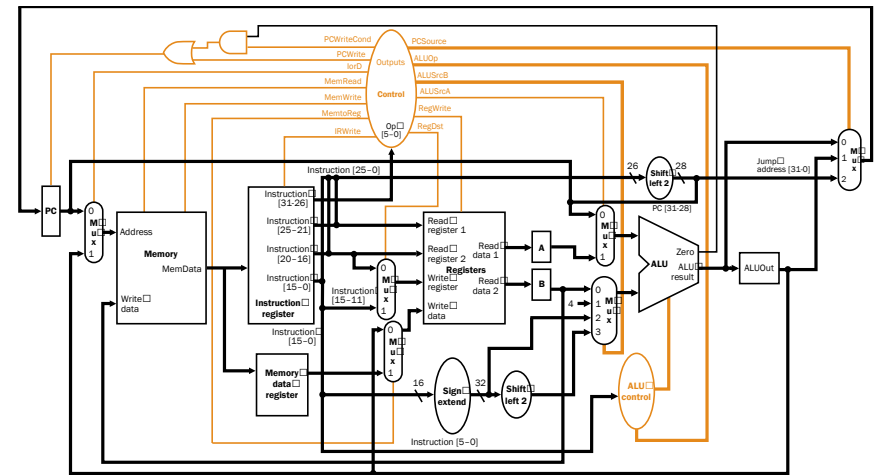
A = Register[IR[25-21]]  
B = Register[IR[20-16]]  
ALUOut = PC + (sign-extend (IR[15-0]) << 2)

## 3. Execution (R-type)



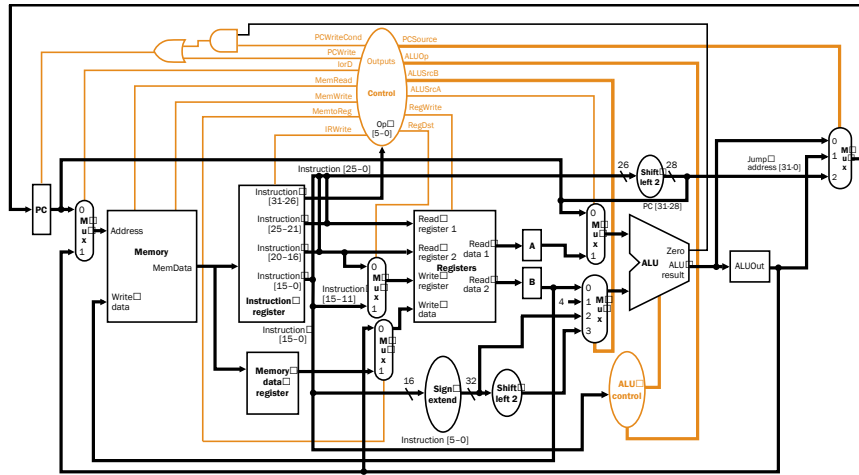
ALUout = A op B

## 4. R-type Completion



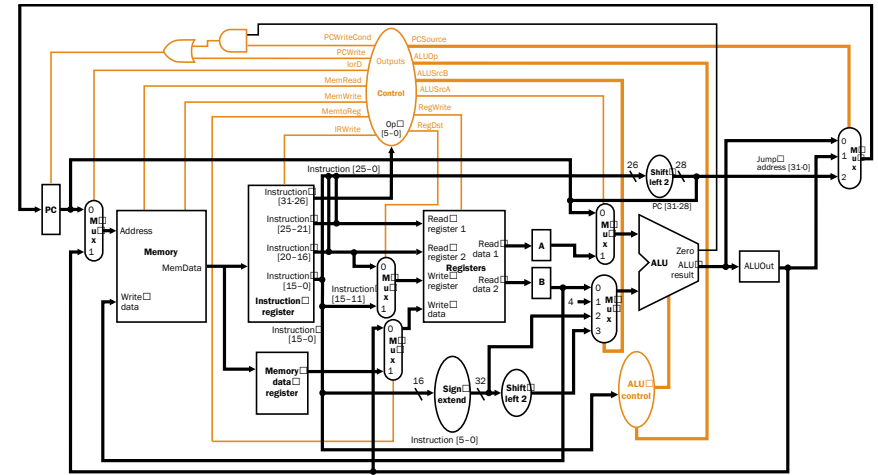
Reg[IR[15-11]] = ALUout

### 3. Branch Completion



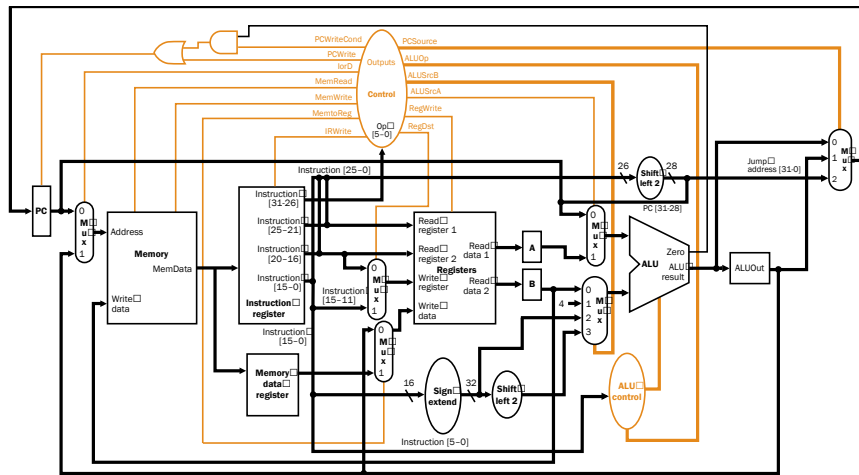
if (A == B) PC = ALUOut

### 3. Memory Address Computation



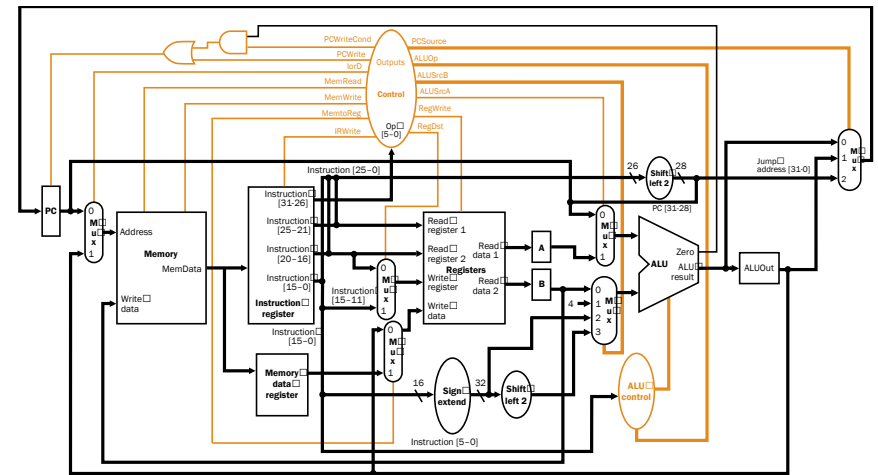
ALUout = A + sign-extend(IR[15:0])

### 4. Memory Access (Load)



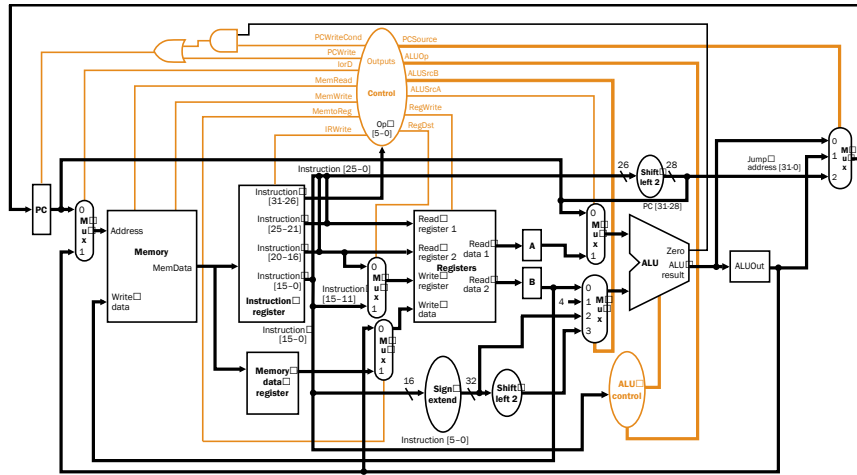
memory-data = Memory[ALUOut]

### 4. Memory Access (Store)



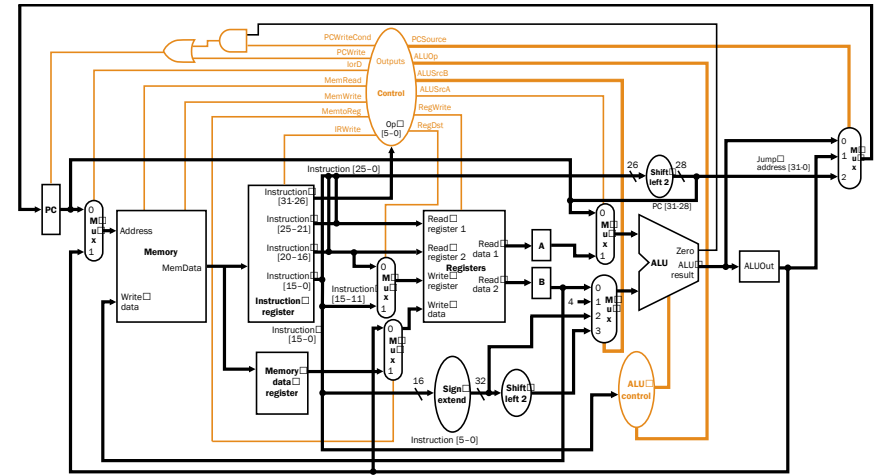
Memory[ALUOut] = B

### 5. Load Write-back



Reg[IR[20-16]] = memory-data

### 3. JMP Completion

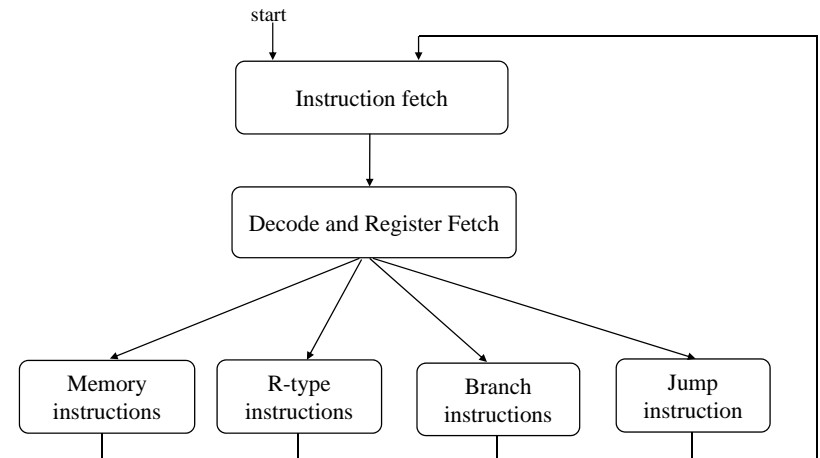


PC = PC[31-28] | (IR[25-0] << 2)

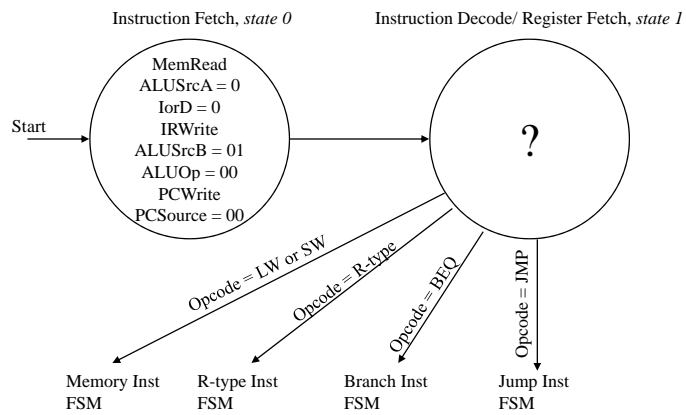
### Multicycle Control

- Single-cycle control used combinational logic
- Multi-cycle control uses ??
- FSM defines a succession of states, transitions between states (based on inputs), and outputs (based on state)
- First two states **same** for every instruction, next state **depends on opcode**

### Multicycle Control FSM



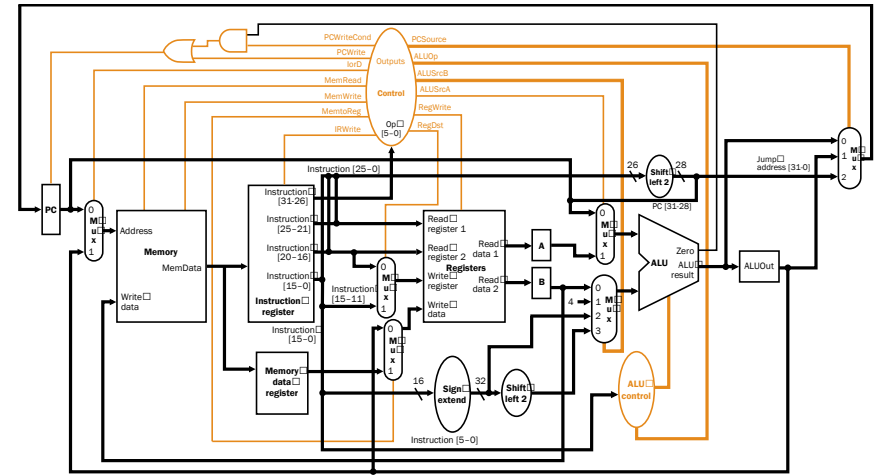
## First two states of the FSM



CSE 141

Dean Tullsen

## Instruction Decode and Reg Fetch

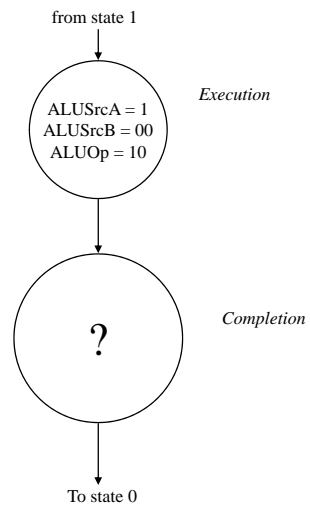


A = Register[IR[25-21]]

B = Register[IR[20-16]]

Target = PC + (sign-extend (IR[15-0]) << 2)

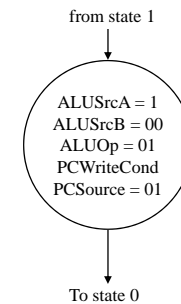
## R-type Instructions



CSE 141

Dean Tullsen

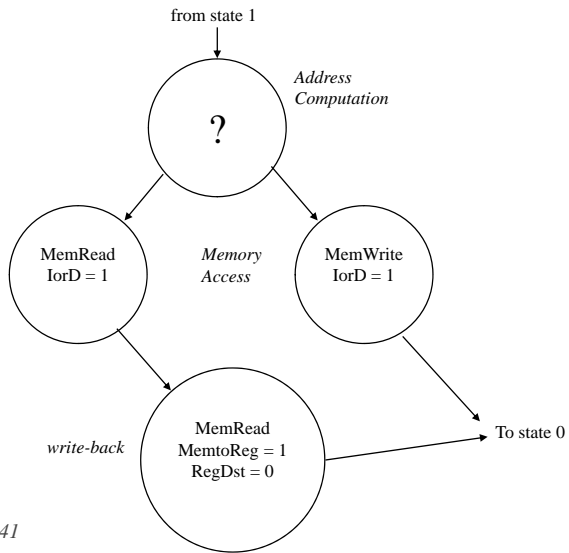
## BEQ Instruction



CSE 141

Dean Tullsen

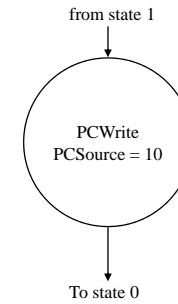
## Memory Instructions



CSE 141

Dean Tullsen

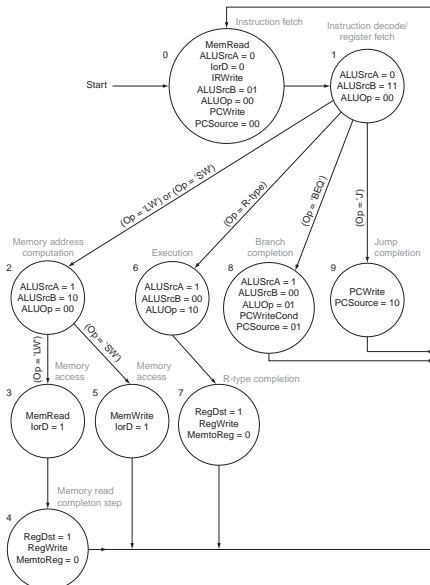
## JMP Instruction



CSE 141

Dean Tullsen

## The Whole FSM



CSE 141

Dean Tullsen

## Some Questions

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

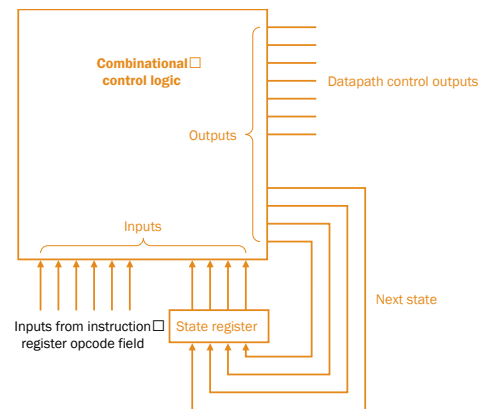
```
lw $t2, 0($t3)
lw $t3, 4($t3)
beq $t2, $t3, Label #assume not taken
add $t5, $t2, $t3
sw $t5, 8($t3)
Label: ...
```
- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of \$t2 and \$t3 take place?
- Assume 20% loads, 10% stores, 50% R-type, 20% branches, **what is the CPI?**

CSE 141

Dean Tullsen

## Finite State Machine for Control

- Implementation:

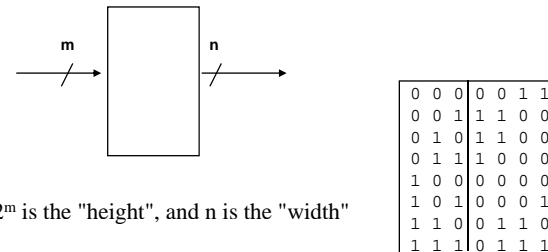


CSE 141

Dean Tullsen

## ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is  $m$ -bits, we can address  $2^m$  entries in the ROM.
  - our outputs are the bits of data that the address points to.



$2^m$  is the "height", and  $n$  is the "width"

CSE 141

Dean Tullsen

## ROM Implementation

- How many inputs are there?
  - 6 bits for opcode, 4 bits for state = 10 address lines (i.e.,  $2^{10} = 1024$  different addresses)
- How many outputs are there?
  - 16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is  $2^{10} \times 20 = 20K$  bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same
  - i.e., opcode is often ignored

CSE 141

Dean Tullsen

## Multicycle CPU Key Points

- Performance gain achieved from variable-length instructions
- $ET = IC * CPI * \text{cycle time}$
- Required very few new state elements
- More, and more complex, control signals
- Control requires **FSM**

CSE 141

Dean Tullsen

## Exceptions

or  
*Oops!*

## Exceptions

- There are two sources of non-sequential control flow in a processor
  - explicit branch and jump instructions
  - exceptions
- *Branches* are synchronous and deterministic
- *Exceptions* are typically asynchronous and non-deterministic
- Guess which is more difficult to handle?

(*control flow* refers to the movement of the program counter through memory)

## Exceptions and Interrupts

the terminology is not consistent, but we'll refer to

- *exceptions* as any unexpected change in control flow
- *interrupts* as any externally-caused exception

So then, what is:

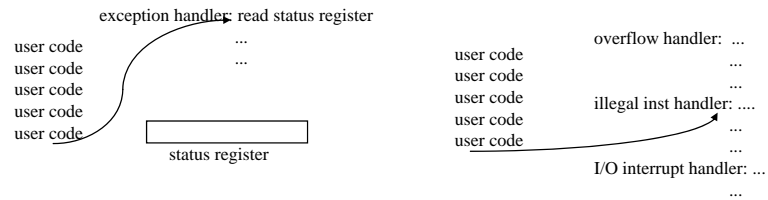
- arithmetic overflow
- divide by zero
- I/O device signals completion to CPU
- user program invokes the OS
- memory parity error
- illegal instruction
- timer signal

## For now...

- The machine we've been designing in class can generate two types of exceptions.
  - arithmetic overflow
  - illegal instruction
- On an exception, we need to
  - save the PC (invisible to user code)
  - record the nature of the exception/interrupt
  - transfer control to OS

## Handling exceptions

- PC saved in EPC (exception program counter), which the OS may read and store in kernel memory
- A status register, and a single exception handler may be used to record the exception and transfer control, or
- A *vectored interrupt* transfers control to a different location for each possible type of interrupt/exception

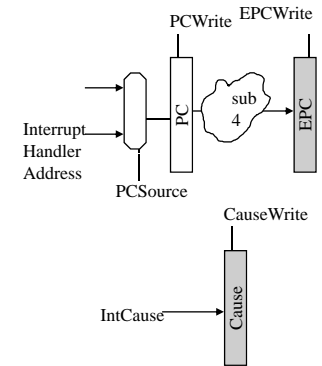


CSE 141

Dean Tullsen

## Supporting exceptions

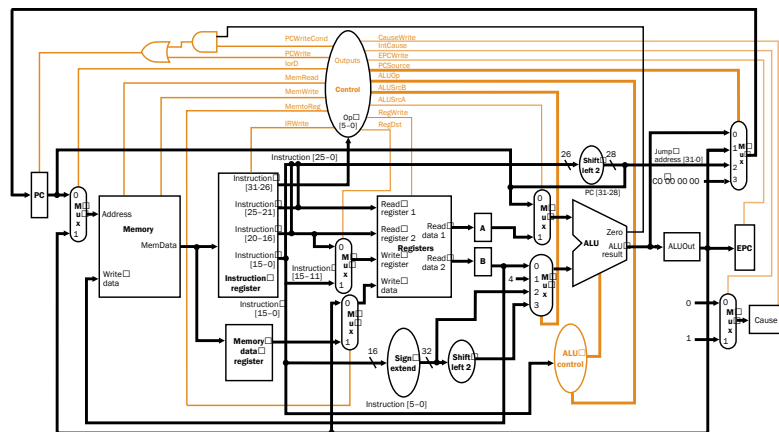
- For our MIPS-subset architecture, we will add two registers:
  - EPC: a 32-bit register to hold the user's PC
  - Cause: A register to record the cause of the exception
    - we'll assume undefined inst = 0, overflow = 1
- We will also add three control signals:
  - EPCWrite (will need to be able to subtract 4 from PC)
  - CauseWrite
  - IntCause
- We will extend PCSource multiplexor to be able to latch the interrupt handler address into the PC.



CSE 141

Dean Tullsen

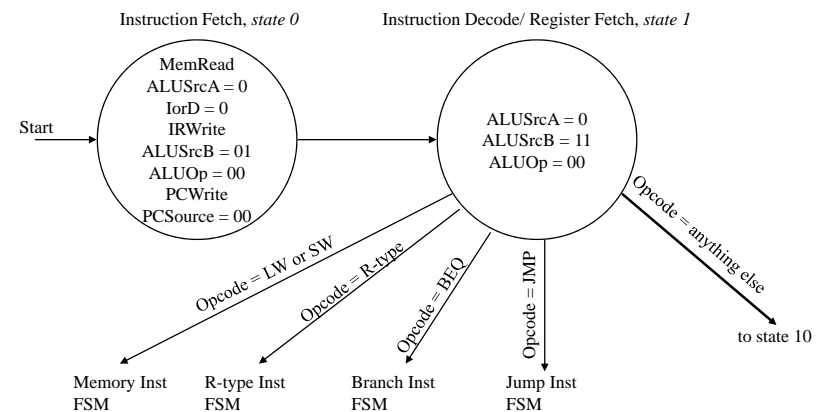
## Supporting exceptions in our DataPath



CSE 141

Dean Tullsen

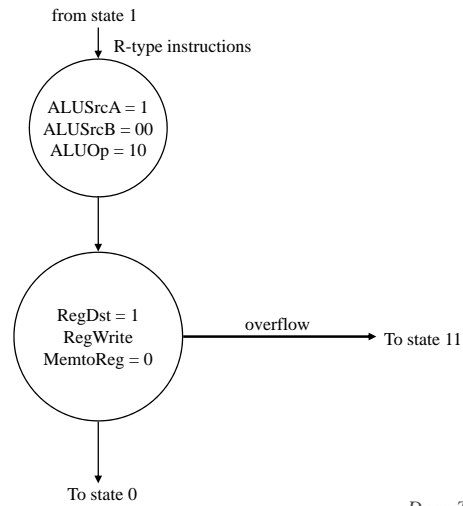
## Supporting exceptions in our FSM



CSE 141

Dean Tullsen

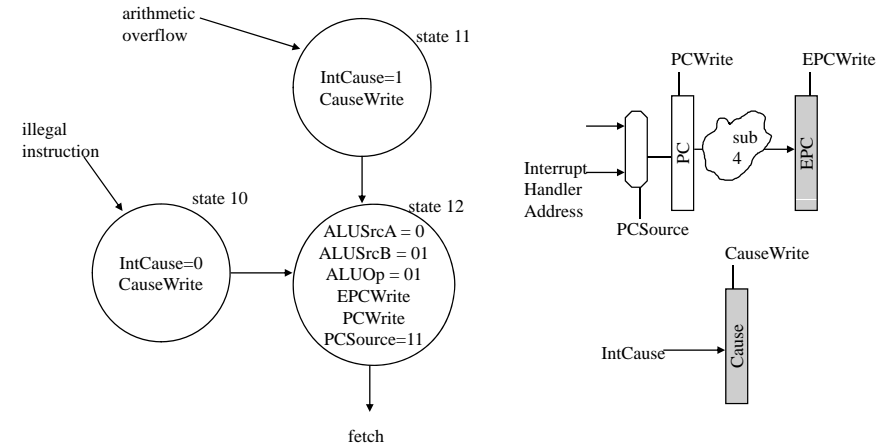
## Supporting exceptions in our FSM.



CSE 141

Dean Tullsen

## Supporting exceptions in our FSM



CSE 141

Dean Tullsen

## Key Point

- Exception-handling is difficult in the CPU, because the interactions between the executing instructions and the interrupt are complex and sometimes unpredictable.

CSE 141

Dean Tullsen