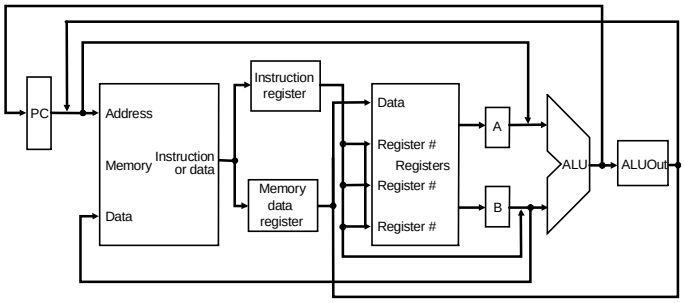
Where we are headed

- **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:**

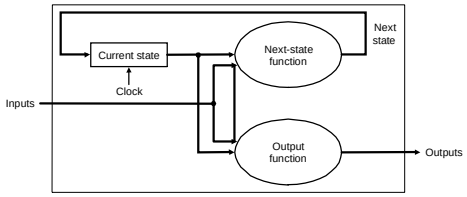


Multicycle Approach

- **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 solely by instruction**
  - **e.g., what should the ALU do for a "subtract" instruction?**
- **We'll use a finite state machine for control**

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)**

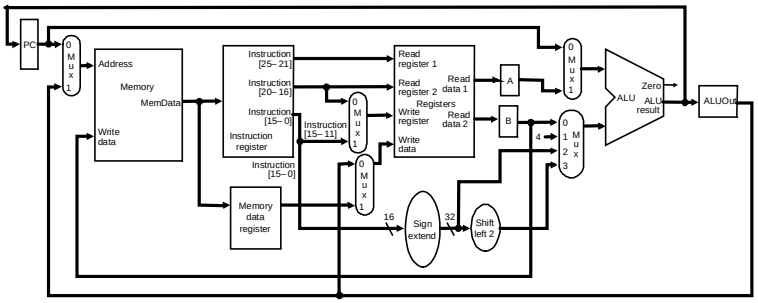
Review: finite state machines

- **Example:**

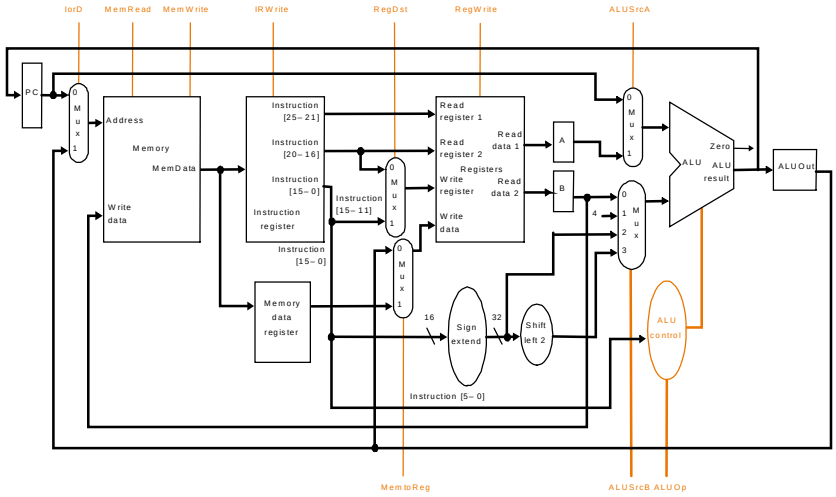
B. 21 ***A friend would like you to build an "electronic eye" for use as a fake security device. The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which, if asserted, indicate that a light should be on. Only one light is on at a time, and the light "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.***

Multicycle Approach

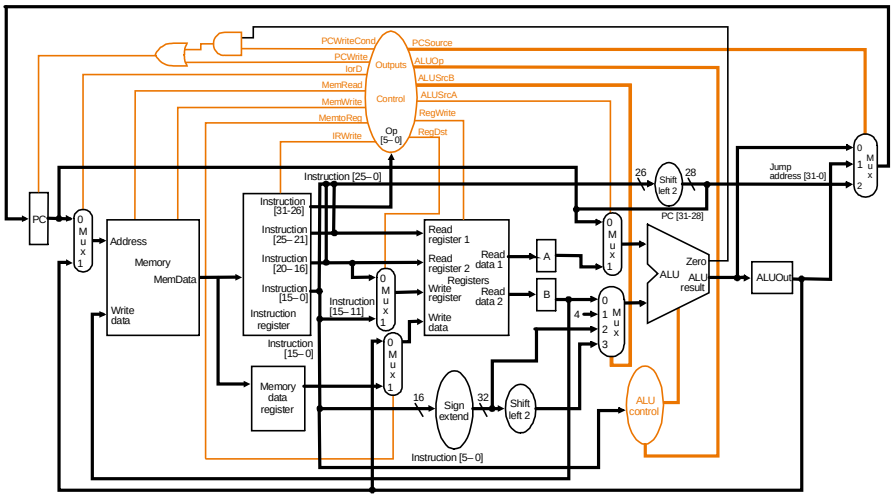
- **Break up the instructions into steps, each step takes a cycle**
  - **balance the amount of work to be done**
  - **restrict each cycle to use only one major functional unit**
- **At the end of a cycle**
  - **store values for use in later cycles (easiest thing to do)**
  - **introduce additional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



Multicycle Datapath with control signals



Multicycle Datapath & Control

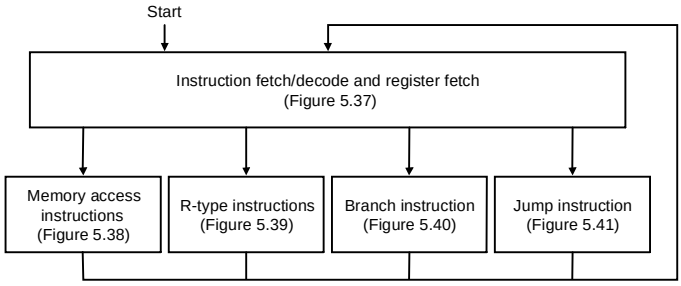


Five Execution Steps

- **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!

High level view of finite state machine control



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?

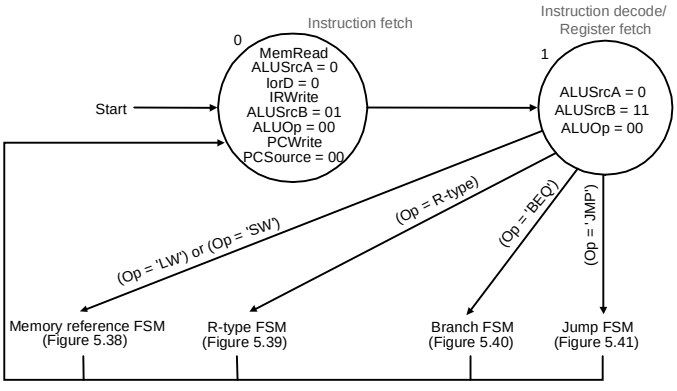
Step 2: Instruction Decode and Register Fetch

- **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)**

Instruction Fetch & Decode



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;
```


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;
```

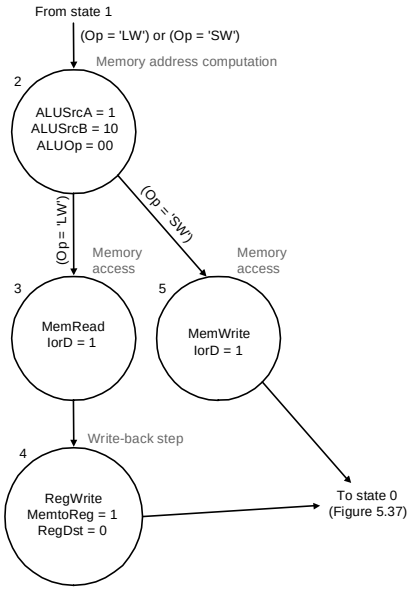
The write actually takes place at the end of the cycle on the edge

Write-back step

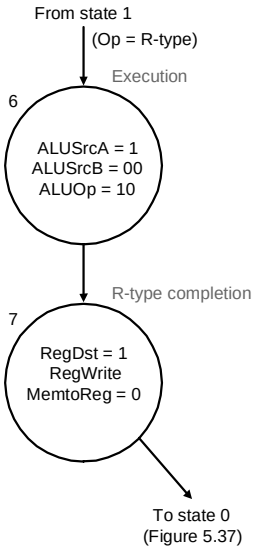
- **`Reg[IR[20-16]]= MDR;`**

What about all the other instructions?

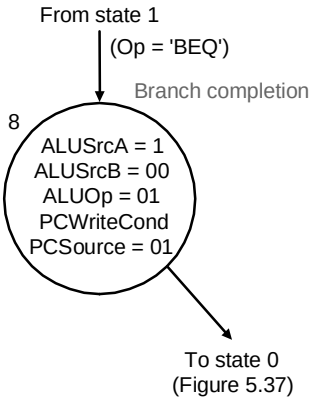
Finite state machine for memory access instructions



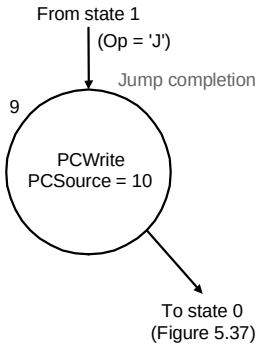
Finite state machine for R-format instructions



Finite state machine for branch instruction



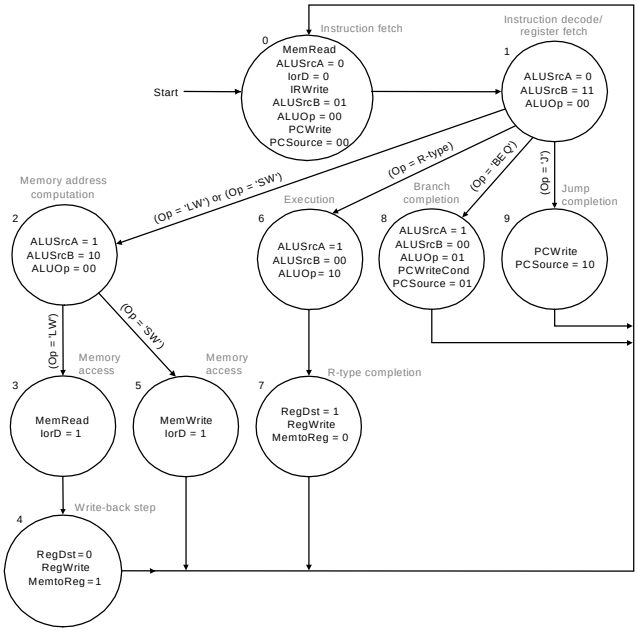
Finite State Machine for jump



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) 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] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Complete Finite State Machine



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?**

Exceptions

- **Hardest part of control is to implement exceptions & interrupts**

Type of event	From where?	MIPS terminology
I/O device request	External	Interrupt
Invoke the operating system from user program	Internal	Exception
Arithmetic overflow	Internal	Exception
Using an undefined instruction	Internal	Exception
Hardware malfunctions	Internal or External	Exception or interrupt

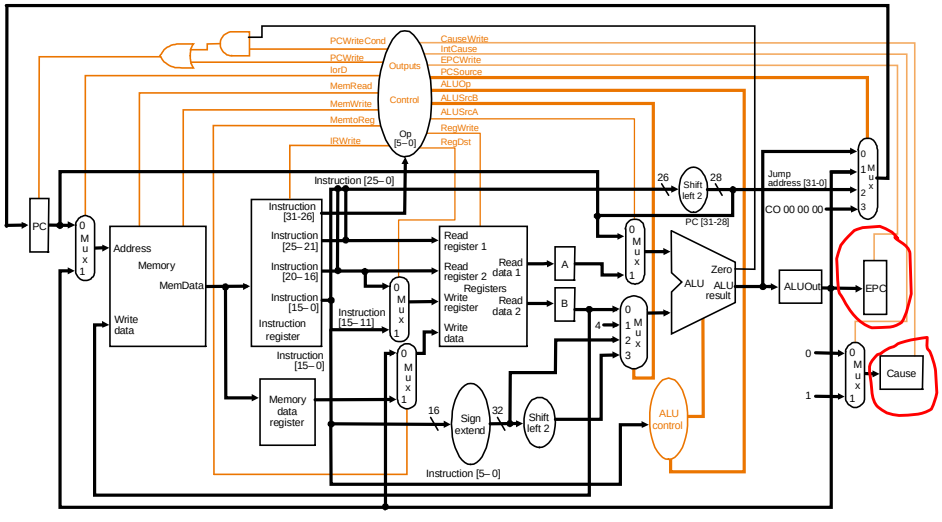
How are exceptions handled?

- **In our design, we will consider two types of exceptions**
  - **Arithmetic overflow**
  - **Execution of an undefined instruction**
- **Actions on exception**
  - **Save address of offending instruction in the Exception Program Counter (EPC)**
  - **Transfer control to the operating system at a pre-specified address (exception handler)**
    - OS then takes appropriate action

Exception handling

- **For the OS to take appropriate action, it must know the <u>reason for the exception</u>**
- **Two ways to communicate reason to OS**
  - **Have a Status register which holds a field that indicates the reason for the exception**
  - **Vectored interrupts**
    - Address to which control is transferred depends upon the cause of the exception
- **MIPS uses first method above; it has a register called <u>Cause</u> (in addition to the EPC register)**

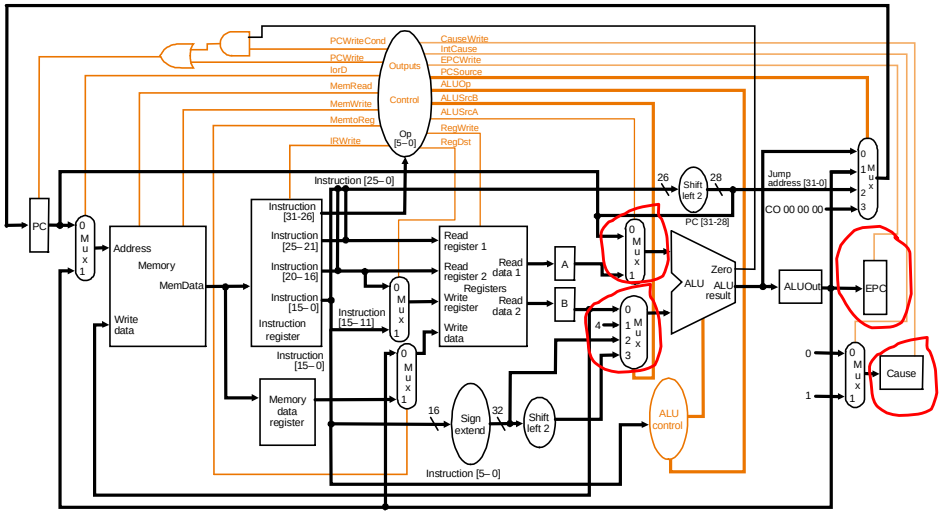
Datapath & Control with support for exceptions



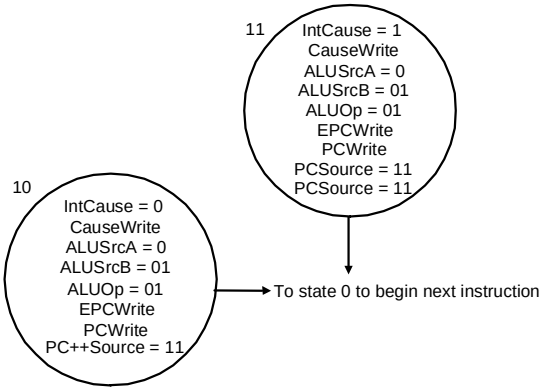
Exception Handling

- Datapath additions
  - EPC, Cause (for undefined instruction, Cause = 0, arithmetic overflow Cause = 1)
- Control Signals
  - EPCWrite, CauseWrite
  - IntCause (sets the Cause register)
  - PCSrc has to be modified so that one of its sources is the OS entry point
- Three steps
  1. Write Cause
  2. EPC = PC – 4 (Have to use ALU, so need to expand multiplexors for ALUSrcA and ALUSrcB
  3. Write PC

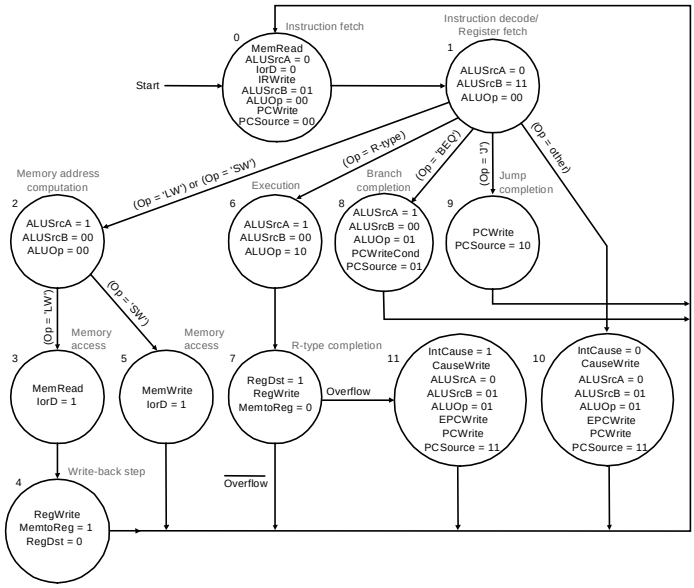
Datapath & Control with support for exceptions



States for handling exceptions



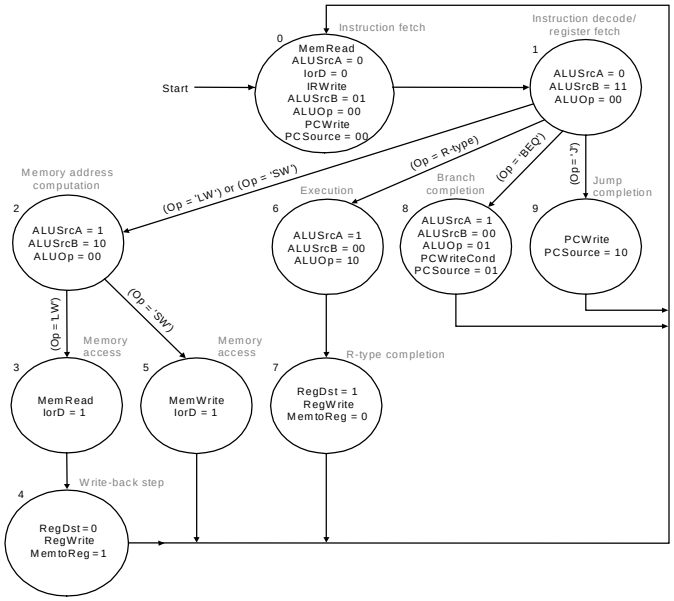
Complete FSM including support for exceptions



Implementing the Control

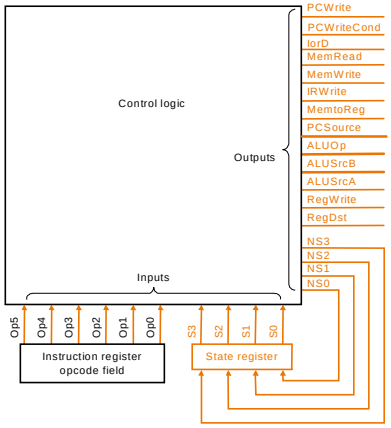
- **Value of control signals is dependent upon:**
  - **what instruction is being executed**
  - **which step is being performed**
- **Use the information we've acculumated to specify a finite state machine**
  - **specify the finite state machine graphically, or**
  - **use microprogramming**
- **Implementation can be derived from specification**

Graphical Specification of FSM



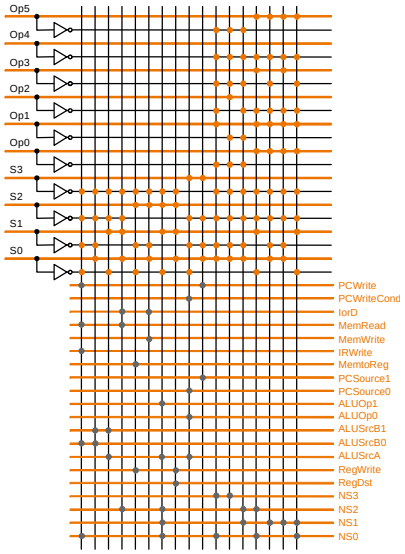
Finite State Machine for Control

- **Implementation:**



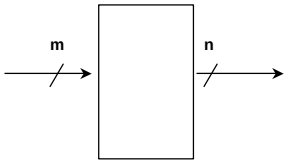
PLA Implementation

- **If I picked a horizontal or vertical line could you explain it?**



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.



0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

  - m is the "height", and n is the "width"

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} x 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

ROM vs PLA

- **Break up the table into two parts**
  - **— 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM**
  - **— 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM**
  - **— Total: 4.3K bits of ROM**
- **PLA is much smaller**
  - **— can share product terms**
  - **— only need entries that produce an active output**
  - **— can take into account don't cares**
- **Size is (#inputs ? #product-terms) + (#outputs ? #product-terms)**
  For this example = (10x17)+(20x17) = 460 PLA cells
- **PLA cells usually about the size of a ROM cell (slightly bigger)**

The Big Picture