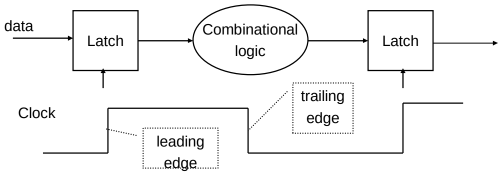
21 October

- Only 12 to go!
- Been to the Fair?
- Assignment 9 due 28th instead of 26th
- Today Control

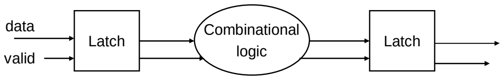
Synchronous Systems



On the leading edge of the clock, the input of a latch is transferred to the output and held.

We must be sure the combinational logic has *settled* before the next leading clock edge.

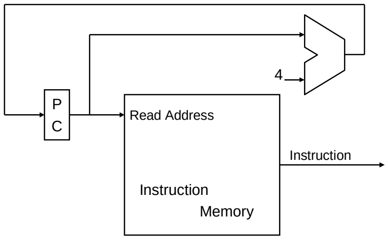
Asynchronous Systems



No clock!
The data carries a “valid” signal along with it
System goes at greatest possible speed.
Only “computes” when necessary.

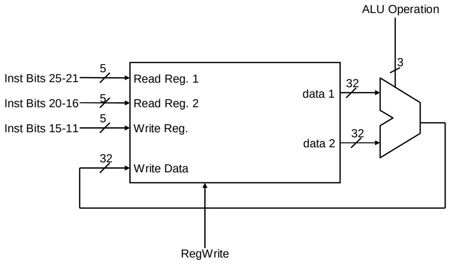
Everything we look at will be synchronous

Fetching Sequential Instructions

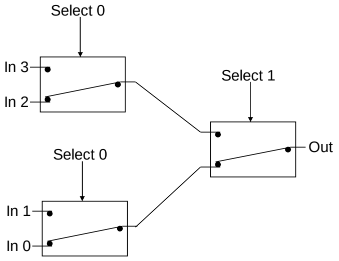


How about branch?

Datapath for R-type Instructions



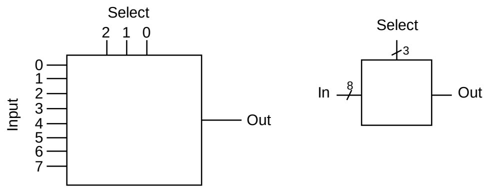
Fun with MUXes



Remember the MUX?

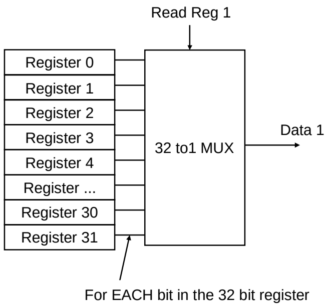
This will route 1 of 4 different 1 bit values to the output.

MUX Blocks



The select signal determines which of the inputs is connected to the output

Inside there is a 32 way MUX per bit



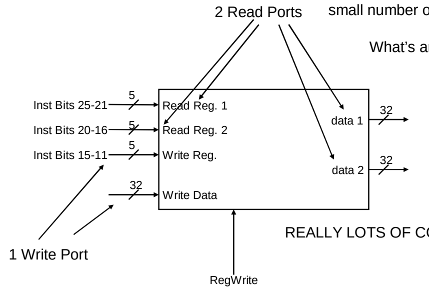
LOT'S OF CONNECTIONS!

And this is just one port!

Our Register File has 3 ports

This is one reason we have only a small number of registers

What's another reason?

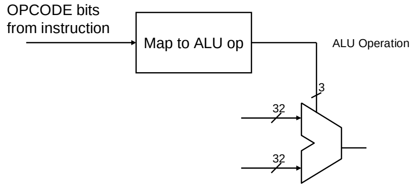


REALLY LOTS OF CONNECTIONS!

Implementing Logical Functions

Suppose we want to map M input bits to N output bits

For example, we need to take the OPCODE field from the instruction and determine what OPERATION to send to the ALU.

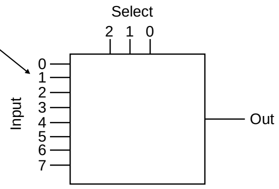


We can get 1 bit out with a MUX

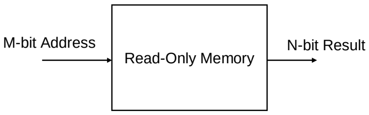
Put the INPUT HERE

Wire these to HIGH or LOW depending on the value you want OUT for that INPUT

For example, 3 input AND has INPUT7 wired HIGH and all the others wired LOW.

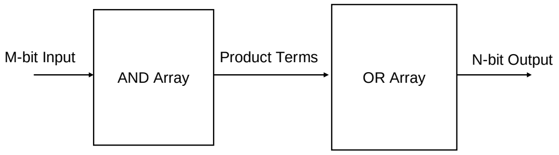


Or use a ROM



Or use a PLA

Programmable Logic Array



Think of the SUM of PRODUCTS form.

The AND Array generates the products of various input bits

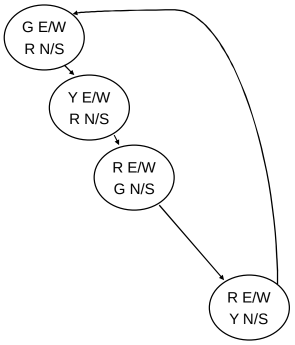
The OR Array combines the products into various outputs

Finite State Machines

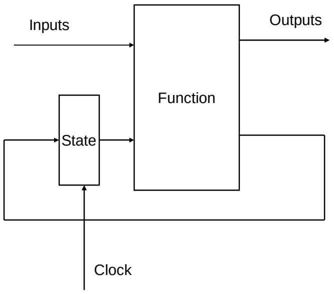
- •A set of STATES
- •A set of INPUTS
- •A set of OUTPUTS
- •A function to map the STATE and the INPUT into the next STATE and an OUTPUT

Remember “Shoots and Ladders”?

Traffic Light Controller



Implementing a FSM



Recognizing Numbers

Recognize the regular expression for floating point numbers

[\t]* [-+]?[0-9]*(. [0-9]*)? (e[-+]?[0–9]+)?

Examples:
+123.456e23
.456
1.5e-10
-123

“a” matches itself

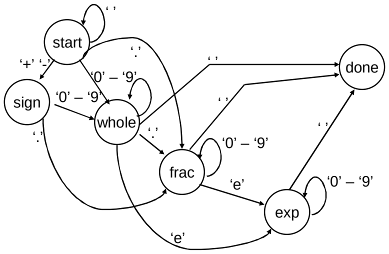
“[abc]” matches one of a, b, or c

“[a-z]” matches one of a, b, c, d, ..., x, y, or z

“0*” matches zero or more 0’s (“”, “0”, “00”, “0000”)

“Z?” matches zero or 1 Z’s

FSM Diagram



FSM Table

IN : STATE → NEW STATE

‘ ’ : start → start

‘0’ | ‘1’ | ... | ‘9’ : start → whole

‘+’ | ‘-’ : start → sign

‘.’ : start → frac

‘0’ | ‘1’ | ... | ‘9’ : sign → whole

‘.’ : sign → frac

‘0’ | ‘1’ | ... | ‘9’ : whole → whole

‘.’ : whole → frac

‘ ’ : whole → done

‘e’ : whole → exp

‘e’ : frac → exp

‘0’ | ‘1’ | ... | ‘9’ : frac → frac

‘ ’ : frac → done

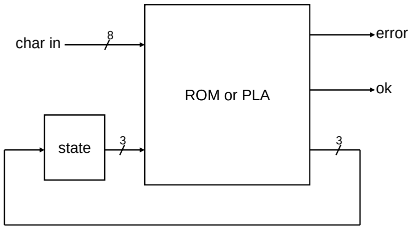
‘0’ | ‘1’ | ... | ‘9’ : exp → exp

‘ ’ : exp → done

STATE ASSIGNMENTS

```
start = 0 = 000
sign  = 1 = 001
whole = 2 = 010
frac  = 3 = 011
exp   = 4 = 100
done  = 5 = 101
error = 6 = 110
```

FSM Implementation



Our PLA has:

- 11 inputs
- 5 outputs

FSM Take Home

- With ***JUST*** a register and some logic, we can implement complicated sequential functions like recognizing a FP number.
- This is useful in its own right for compilers, input routines, etc.
- The reason we’re looking at it here is to see how designers implement the complicated sequences of events required to implement instructions
- Think of the OP-CODE as playing the role of the input character in the recognizer. The character AND the state determine the next state (and action).