Spiral 2-3

Negative Logic

One-hot State Assignment

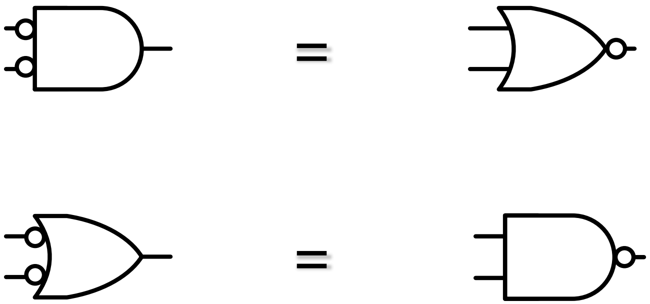
System Design Examples

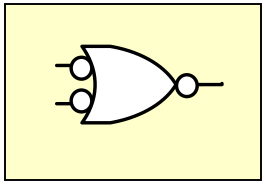
Learning Outcomes

- I understand the active-low signal convention and how to interface circuits that use both active-high and active-low signals
- I can take any state diagram and create a corresponding state machine using one-hot implementation by using one FF per state and creating the D-input circuit by converting each incoming transition arrow to a state into a logic gate and OR-ing them together
- I understand how to decompose an algorithm into states for each step and appropriate datapath units for each operator

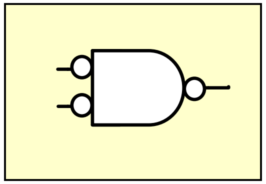
NEGATIVE (ACTIVE-LO) LOGIC

DeMorgan Equivalents





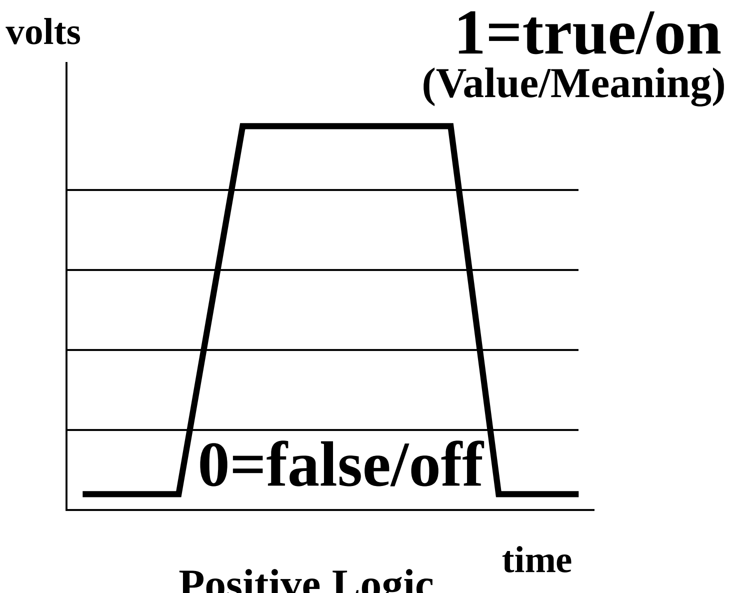
=



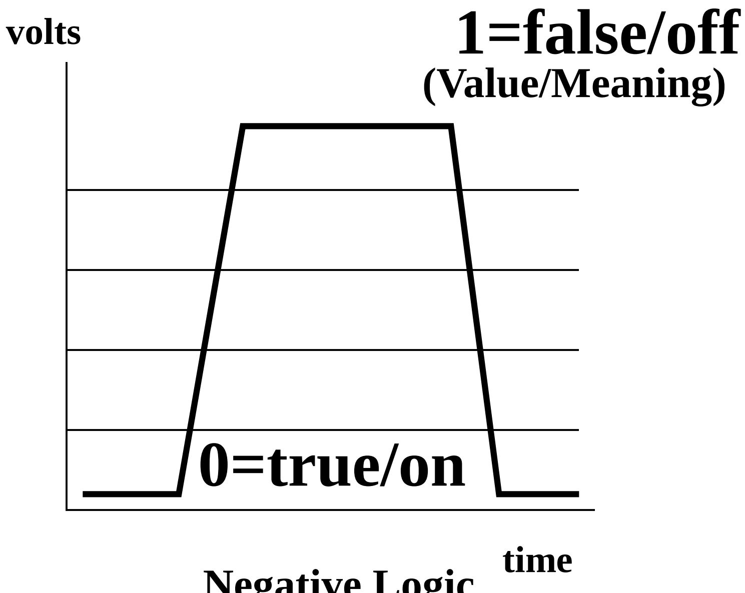
=

Negative Logic

- Recall it is up to us humans to assign meaning to the two voltage levels
  - Thus, far we've used (unknowingly) the positive logic convention where 1 means true and 0 means false
  - In negative logic 0 means true and 1 means false



Positive Logic Convention

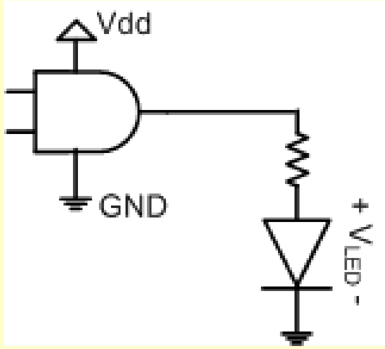


Negative Logic Convention

Why Active-low

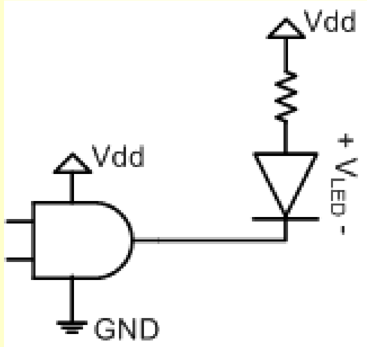
- Some digital circuits are better at “sinking” (draining/sucking) electric current than “sourcing” (producing) current

Active-hi output



LED is on when gate outputs '1'

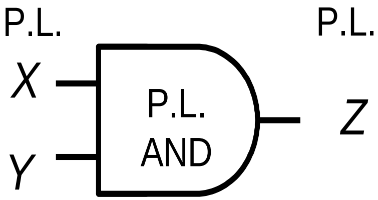
Active-low output



LED is on when gate outputs '0'

Negative Logic 'AND' Function

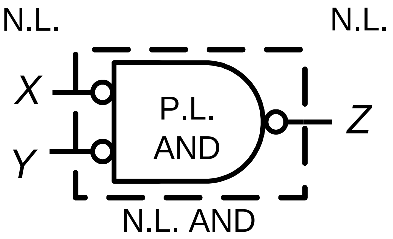
Traditional P.L. AND



X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

Traditional AND gate functionality assumes positive logic convention

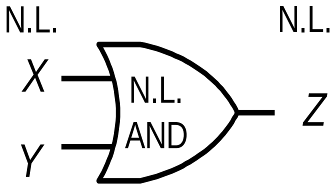
N.L. AND function



X	Y	Z
1	1	1
1	0	1
0	1	1
0	0	0

Given negative logic signals, we can invert to positive logic, perform the AND operation, then convert back to negative logic

N.L. AND = P.L. OR

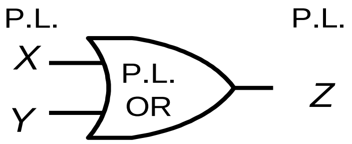


X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

However, we then see that an OR gate implements the negative logic 'AND' function

Negative Logic ‘OR’ Function

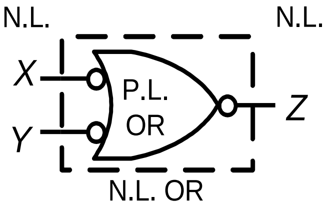
Traditional P.L. OR



X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	1

Traditional OR gate functionality assumes positive logic convention

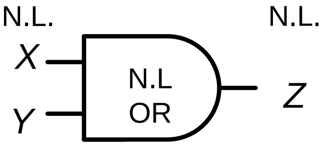
N.L. OR function



X	Y	Z
1	1	1
1	0	0
0	1	0
0	0	0

Given negative logic signals, we can invert to positive logic, perform the OR operation, then convert back to negative logic

N.L. OR = P.L. AND

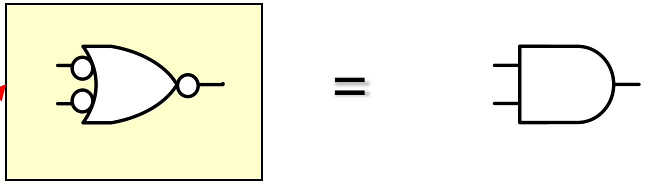


X	Y	Z
0	0	0
0	1	0
1	0	0
1	1	1

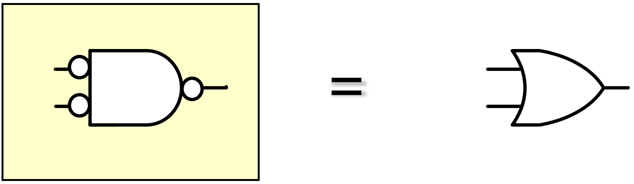
However, we then see that an AND gate implements the negative logic ‘OR’ function

Negative Logic

A negative logic OR function is equivalent to an AND gate



A negative logic AND function is equivalent to an OR gate



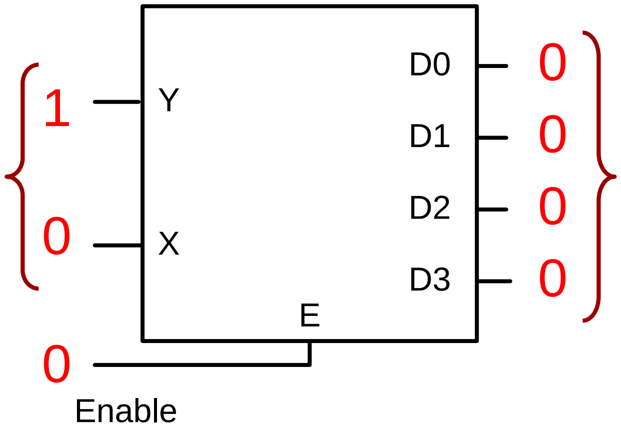
These are the preferred way of showing the N.L. functions because the inversion bubbles explicitly show where N.L. is being converted to P.L. and the basic gate schematics retain their meaning (when we see an AND gate we know we're doing some king of AND function with the bubbles indicating N.L.)

Active-hi vs. Active-low

- Active-hi convention
  - 1 = on/true/active
  - 0 = off/false/inactive
- Active-low convention
  - 0 = on/true/active
  - 1 = off/false/inactive
- To convert between conventions
  - INVERT!!!

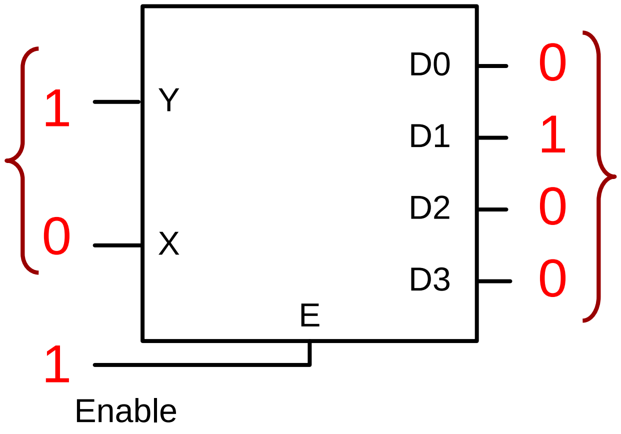
Enables

When E=0,
inputs is
ignored



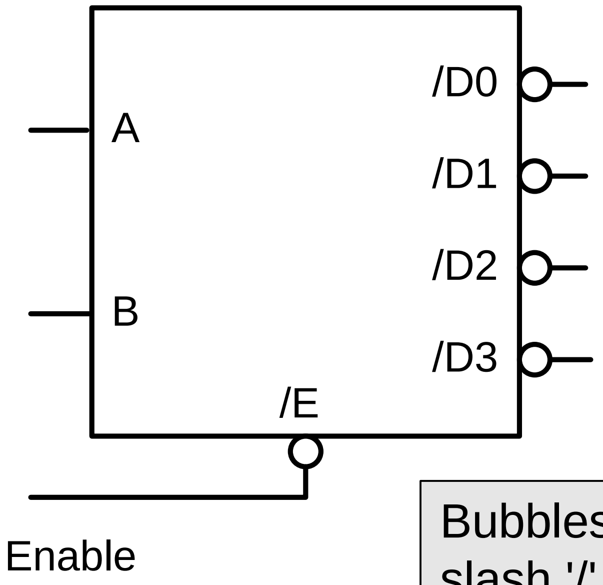
Since E=0,
all outputs = 0

When E=1,
inputs will cause the
appropriate output to
go active



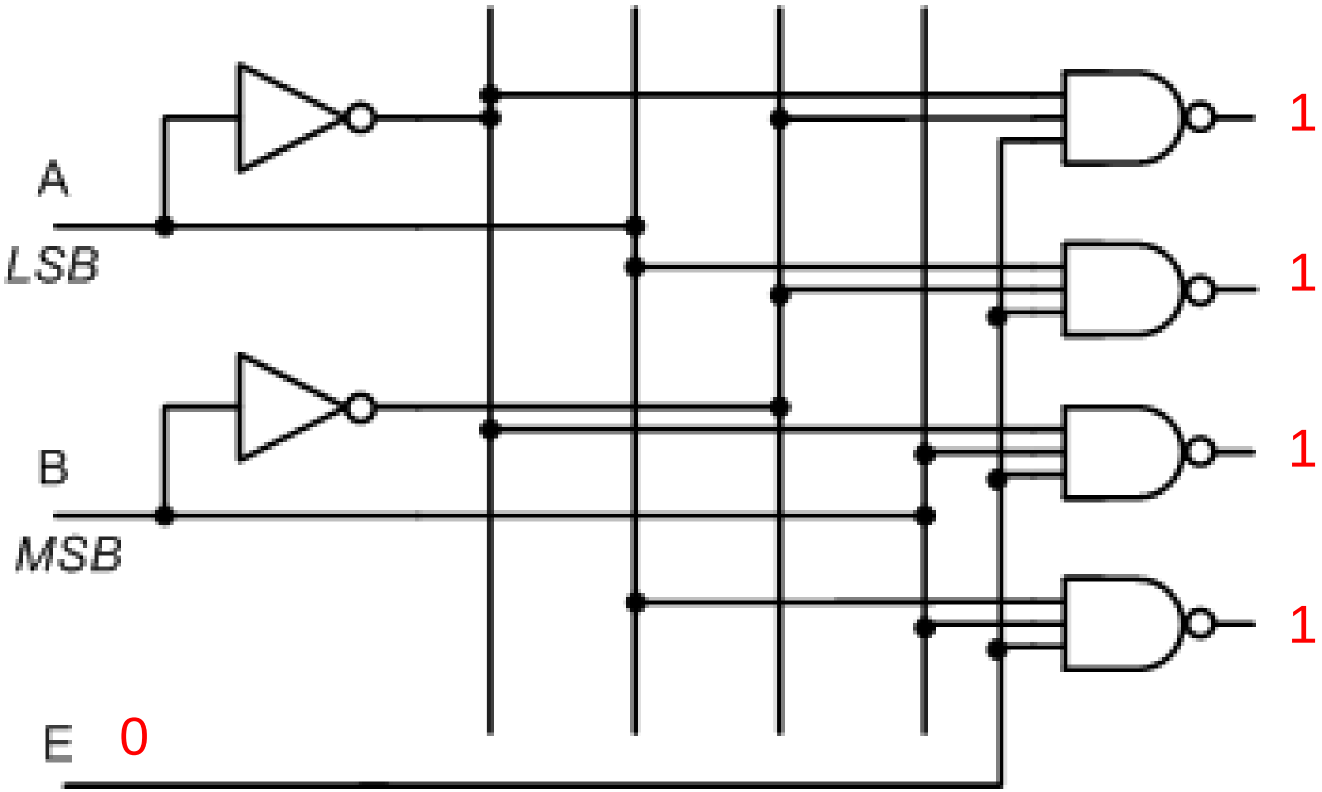
Since E=1,
outputs will
function normally

Decoder w/ Active Low Enable and Outputs



Bubbles and signals starting with a slash '/' indicate an active-low input or output…not an inverter…the inverters are actually in the logic diagram on the next pages…

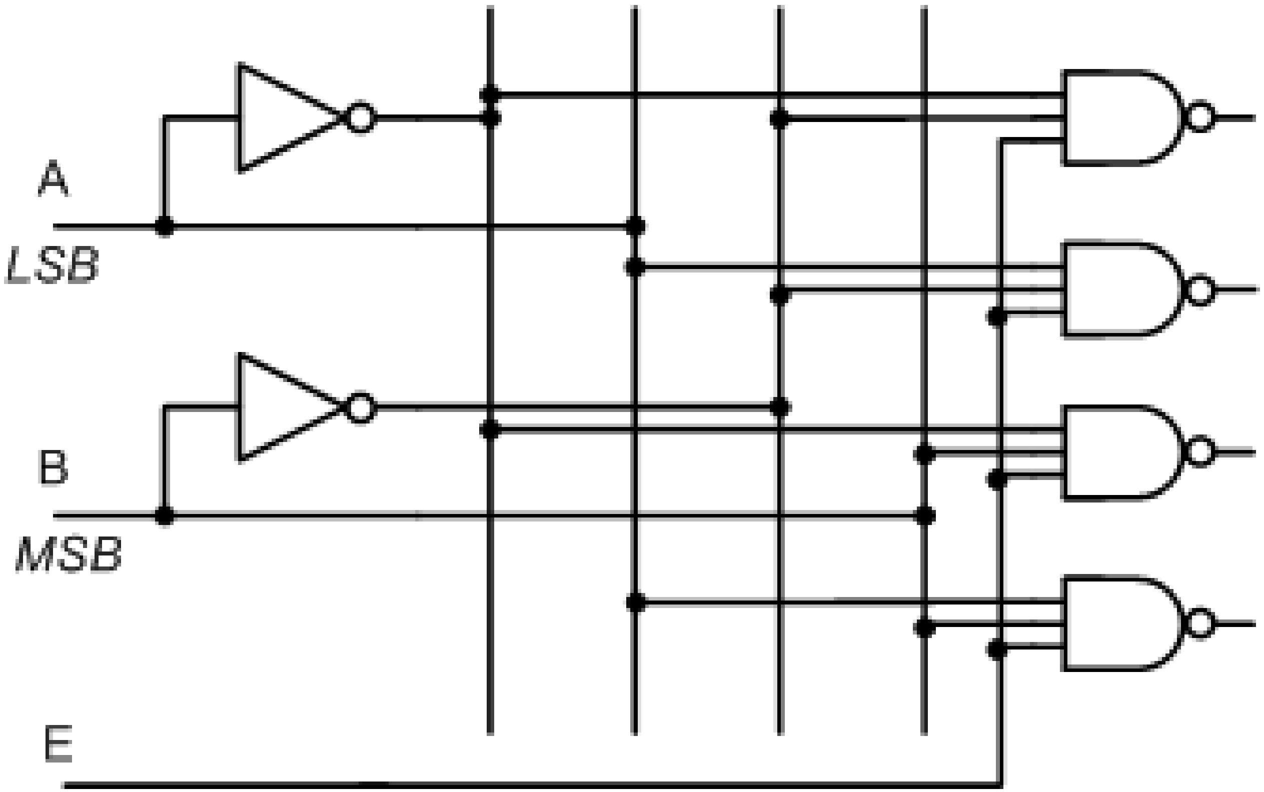
Active-Lo Outputs



When E=inactive (inactive means 0), Outputs turn off (off means 1)

When E=active (active means 1), Selected outputs turn on (on means 0)

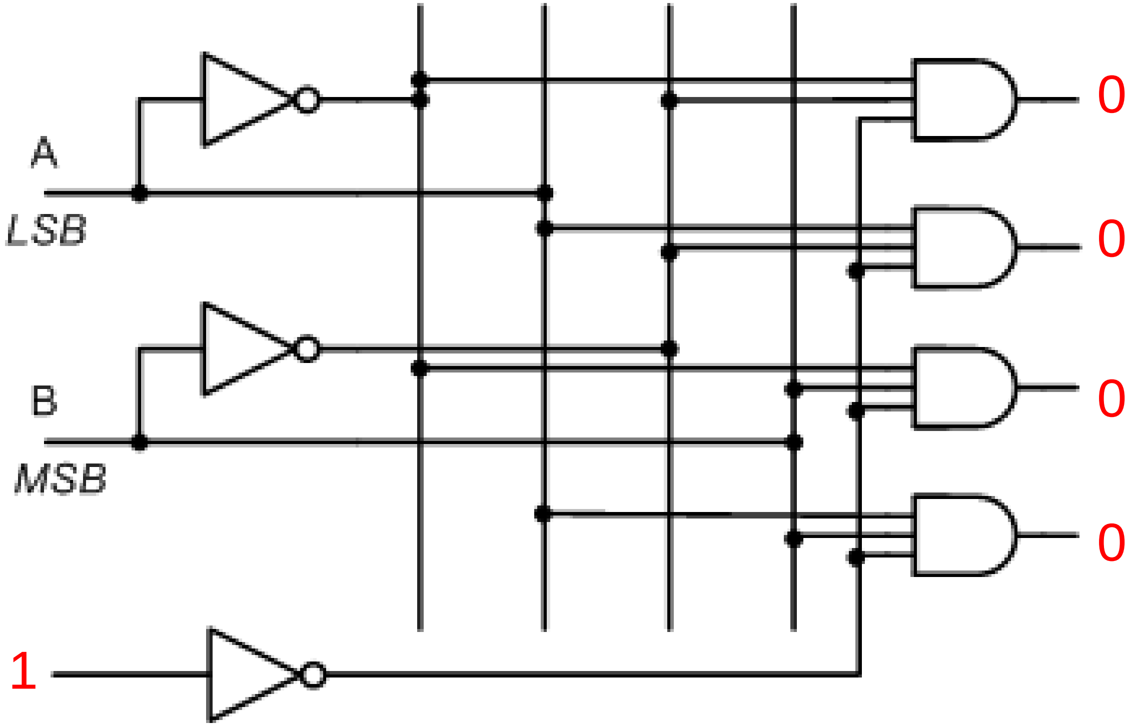
Active-Lo Outputs



When E=inactive (inactive means 0), Outputs turn off (off means 1)

When E=active (active means 1), Selected outputs turn on (on means 0)

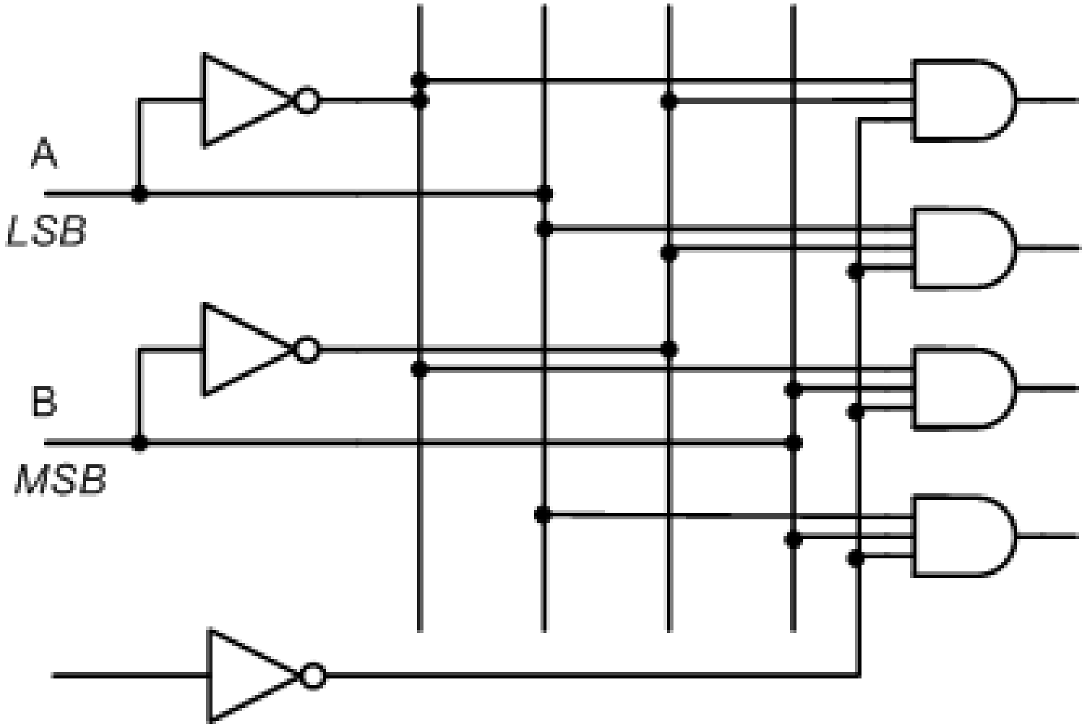
Active-Lo Enable



When E=inactive (inactive means 1), Outputs turn off (off means 0)

When E=active (active means 0), Selected outputs turn on (on means 1)

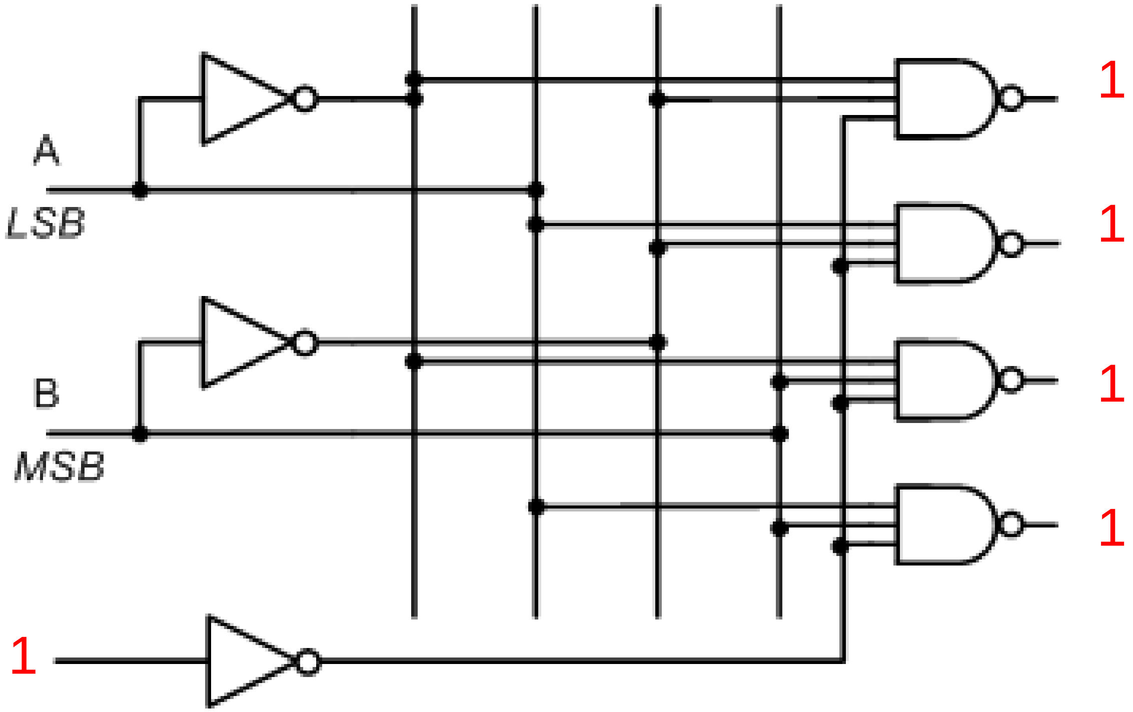
Active-Lo Enable



When E=inactive (inactive means 1), Outputs turn off (off means 0)

When E=active (active means 0), Selected outputs turn on (on means 1)

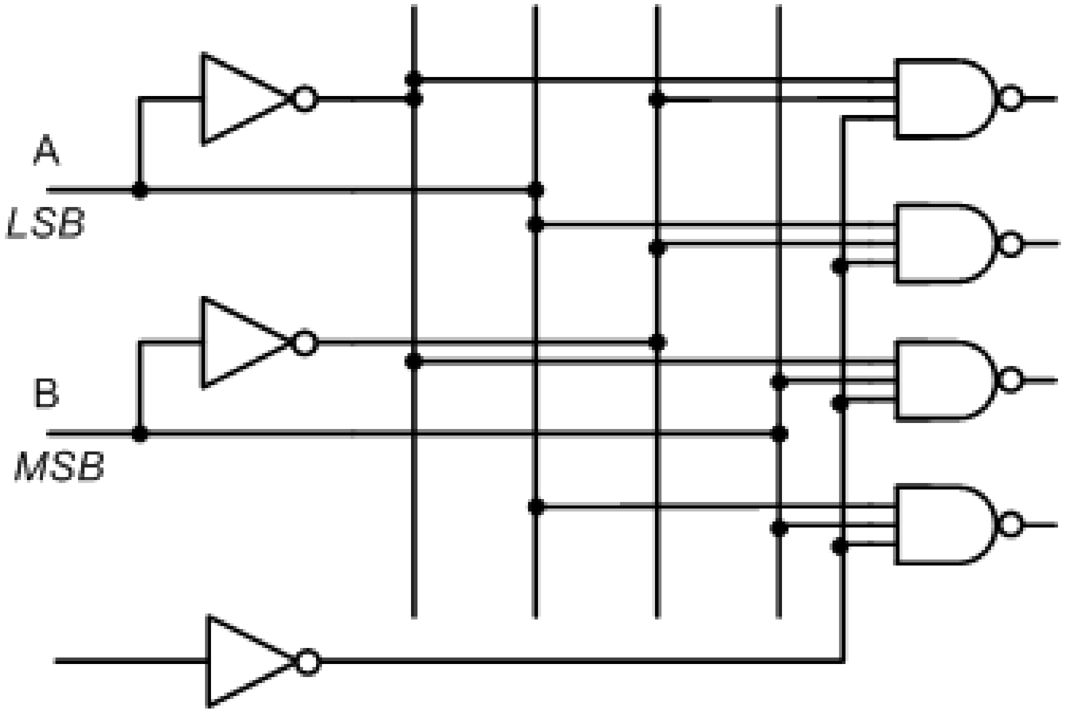
Active-Lo Enable



When E=inactive (inactive means 1), Outputs turn off (off means 1)

When E=active (active means 0), Selected outputs turn on (on means 0)

Active-Lo Enable

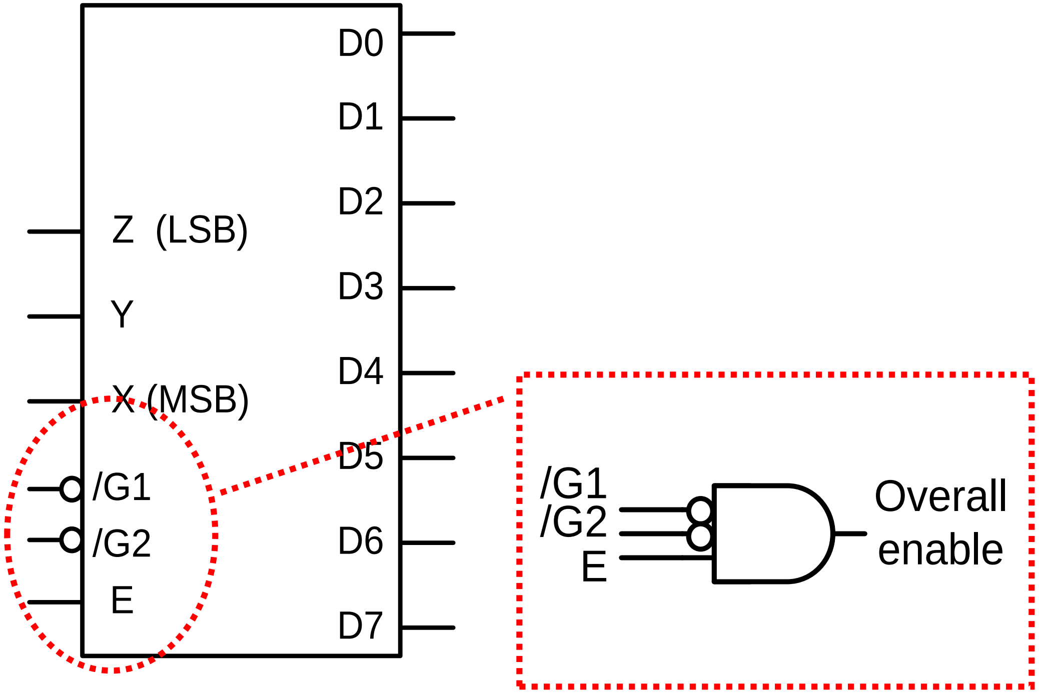


When E=inactive (inactive means 1), Outputs turn off (off means 1)

When E=active (active means 0), Selected outputs turn on (on means 0)

Decoder w/ Multiple Enables

- When a decoder has multiple enables, all enables ***must be active*** for the decoder to be enabled



3 Enables

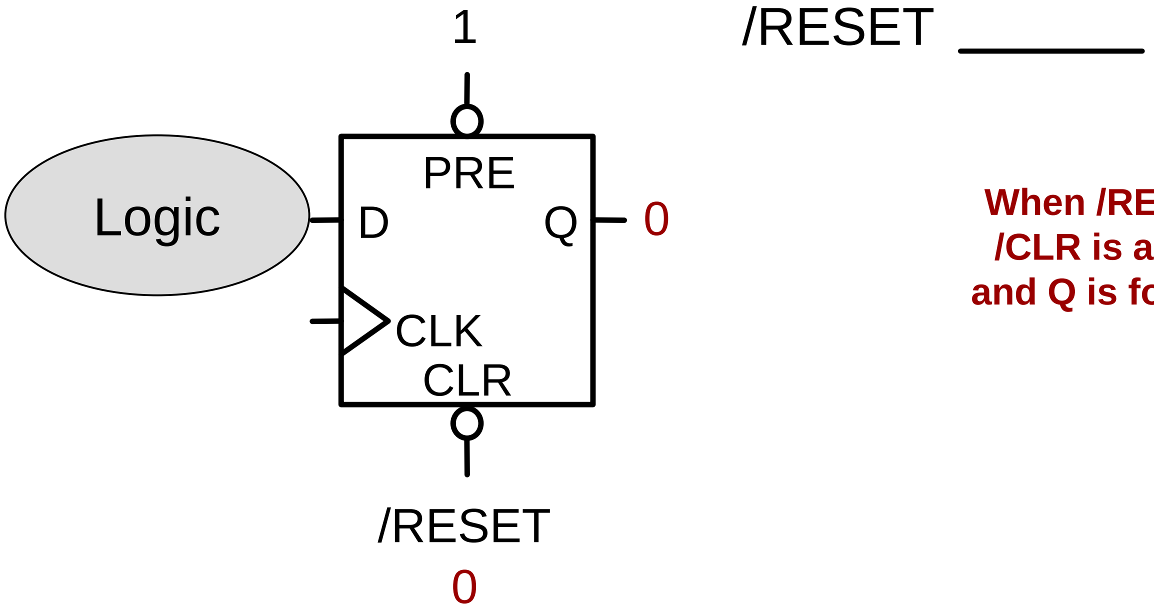
/G1 must equal 0

/G2 must equal 0

and E must equal 1

Active Low CLR and PRESET

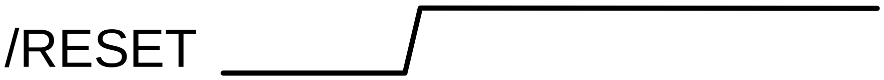
- The reset signal might also be active low (0 = Reset, 1 = Normal operations)
- FFs can be made with active low /CLR & /PRE

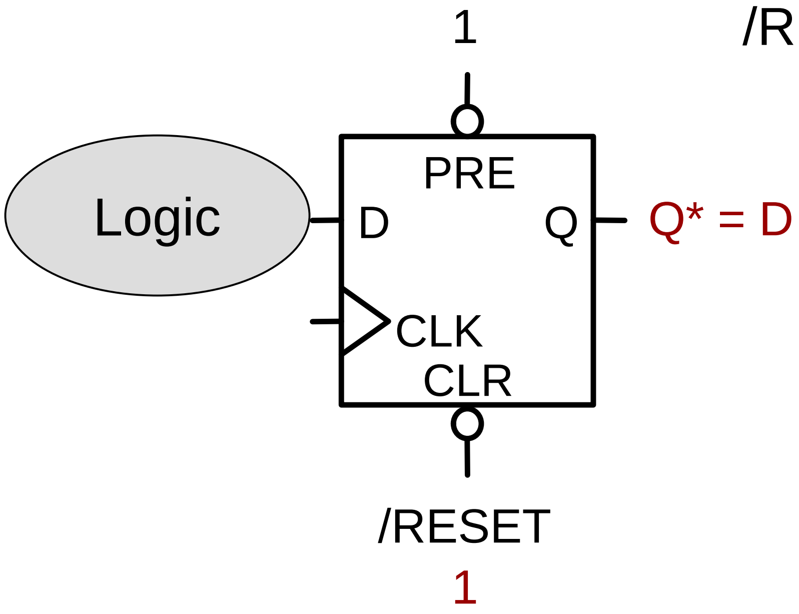


When /RESET = 0, /CLR is activated and Q is forced to 0

Active Low CLR and PRESET

- Need to be able to initialize Q to a known value (0 or 1)

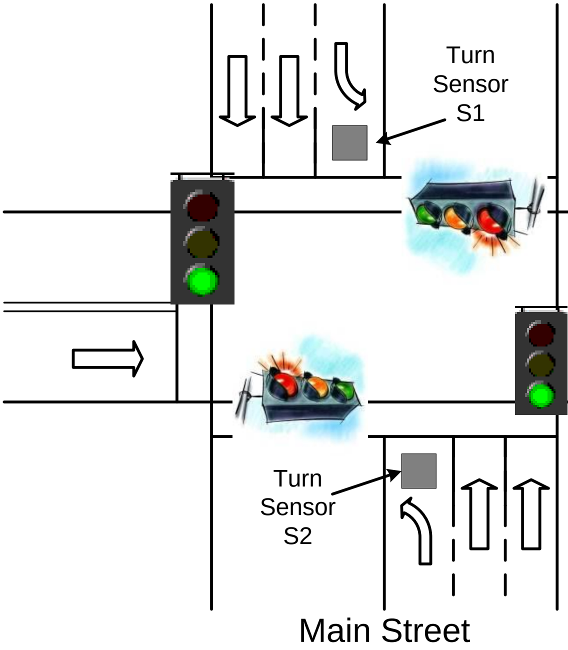




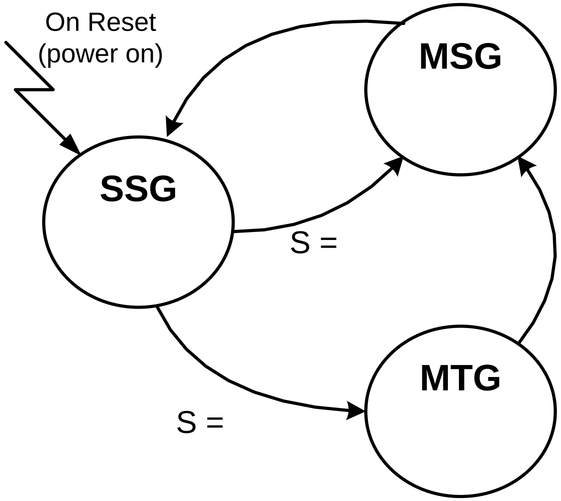
When /RESET = 1, /CLR is inactive and Q looks at D at each clock edge

ONE-HOT STATE ASSIGNMENT

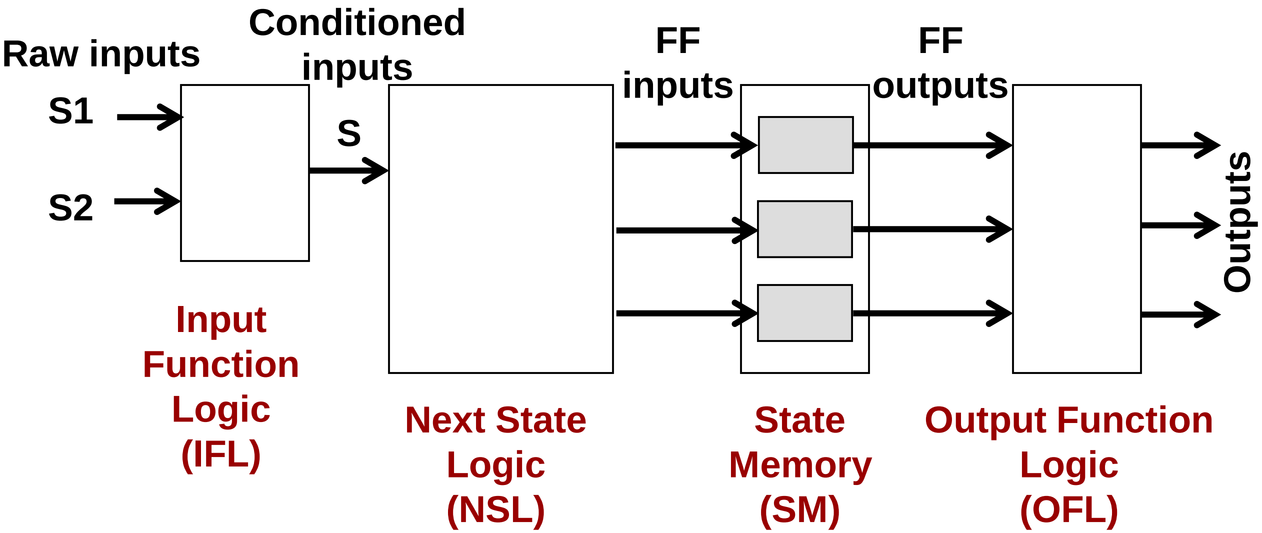
Digital System Representation



Overall sensor output
S = S1 + S2



State Diagram



Encoded State Assignment Review

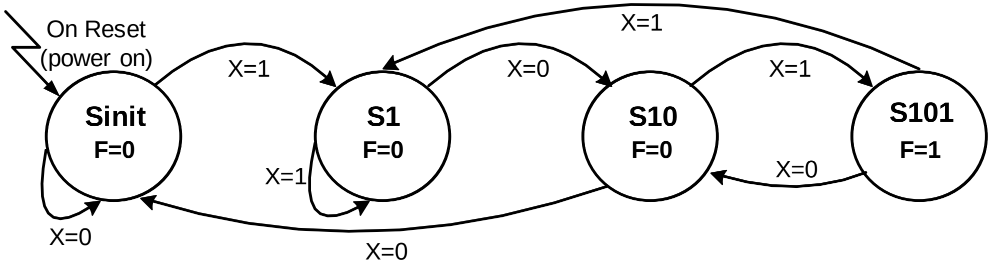
State Diagrams

1. States
2. Transition Conditions
3. Outputs

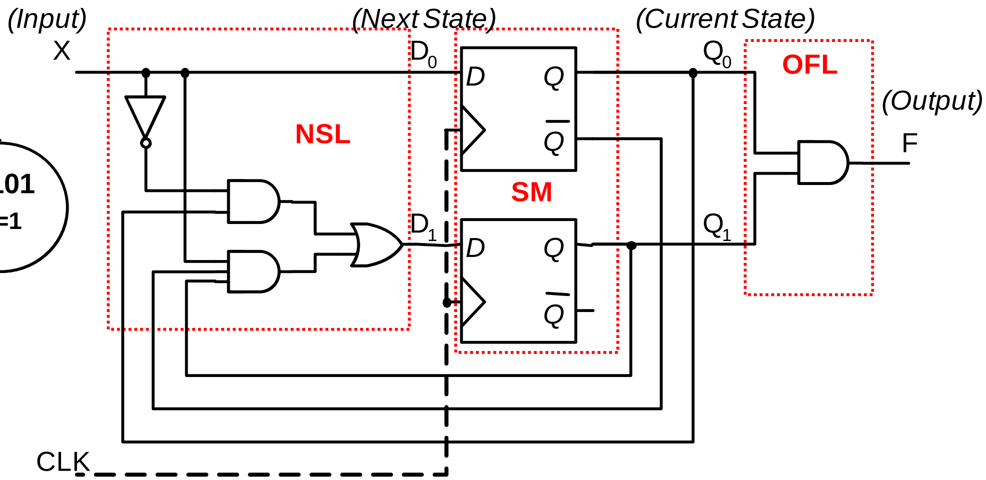
State Machine

1. State Memory => FF's
   - n-FF's => 2^n states
2. Next State Logic (NSL) + Input Function Logic (IFL)
   - combinational logic for FF inputs
3. Output Function Logic (OFL)
   - MOORE: f(state)
   - MEALY: f(state + inputs)

State Machines require sequential logic to remember the current state (w/ just combo logic we could only look at the current value of X, but now we can take 4 separate actions when X=0)

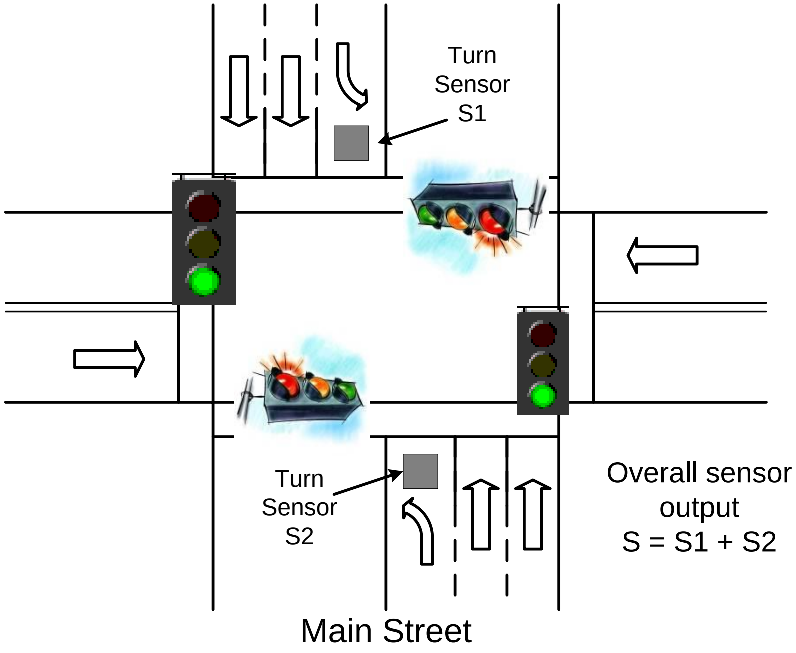


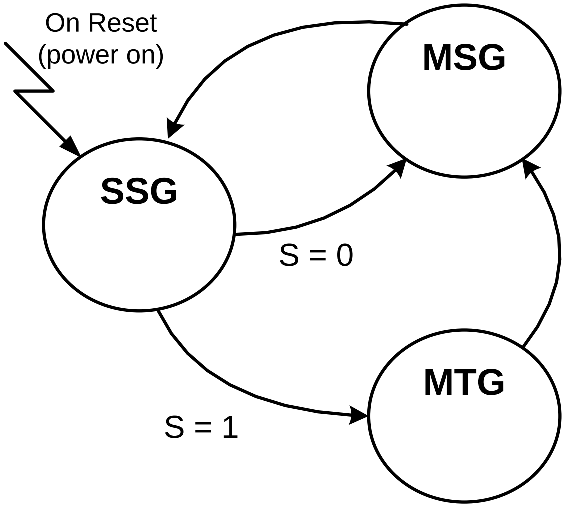
State Diagram for "101" Sequence Detector



State Assignment

- Design of the traffic light controller with main turn arrow
- Represent states with some binary code, but what kind?
  - Encoded: 3 States => 2 bit code: 00=SSG, 01=MSG, 10=MTG
  - One-hot: Separate FF per state: 100=SSG, 010=MSG, 001=MTG

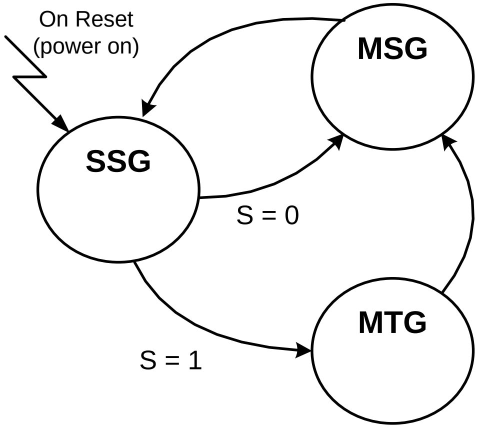




State Diagram

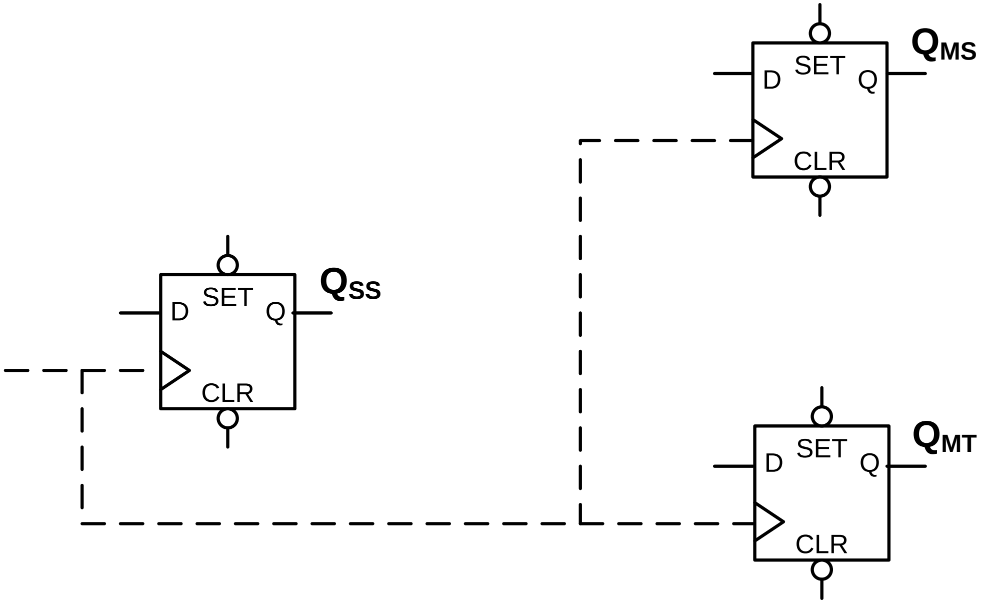
NSL Implementation in 1-Hot Method

- In one-hot assignment, NSL is designed by simple observation
- For each state, examine each incoming transition
  - Each incoming arrow will be one case in our logic
  - We can just OR each condition together
- Describe each transition as a combination of what state it originates from & any associated conditions
- Ex. Two arrows converge on MS: “Q_{MS} should be ‘1’ on the next clock when…
  - Current state is MT ***…OR…***
  - Current state is SS **AND** S=0



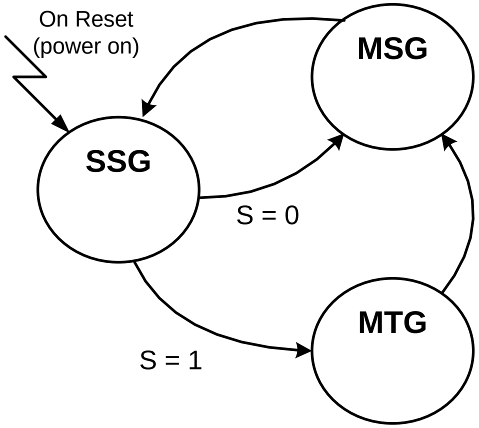
	Q_{SS}	Q_{MT}	Q_{MS}
SS	1	0	0
MT	0	1	0
MS	0	0	1

One-hot State Assignment



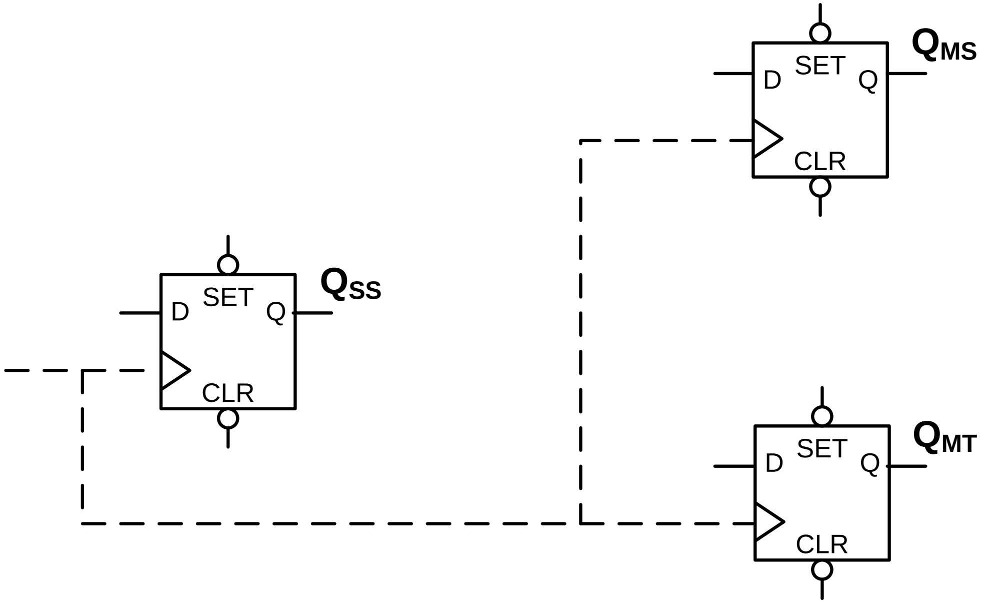
NSL Implementation in 1-Hot Method

- Two arrows converge on MS: "Q_{MS} should be '1' on the next clock when…
  - Current state is MT ***…OR…***
  - Current stat is SS **AND** S=0
- **$Q^*_{MS} = D_{MS} = Q_{MT} + Q_{SS} \bullet S'$**
- **$Q^*_{MT} = D_{MT} =$**
- **$Q^*_{SS} = D_{SS} =$**
- **What about initial state? Preset the appropriate flop.**



	Q_{SS}	Q_{MT}	Q_{MS}
SS	1	0	0
MT	0	1	0
MS	0	0	1

One-hot State Assignment



Array Multiplier (Combinational)

Add and Shift Method (Sequential)

MULTIPLICATION TECHNIQUES

Multiplication Techniques

- A multiplier unit can be
  - Purely Combinational: Each partial product is produced in parallel and fed into an array of adders to generate the product
  - Sequential and Combinational: Produce and add 1 partial product at a time (per cycle)

Combinational Multiplier Analysis

- Large Area due to (n-1) m-bit adders
  - n-1 because the first adder adds the first two partial products and then each adder afterwards adds one more partial product
- Propagation delay is in two dimensions
  - proportional to m+n

Sequential Multiplier

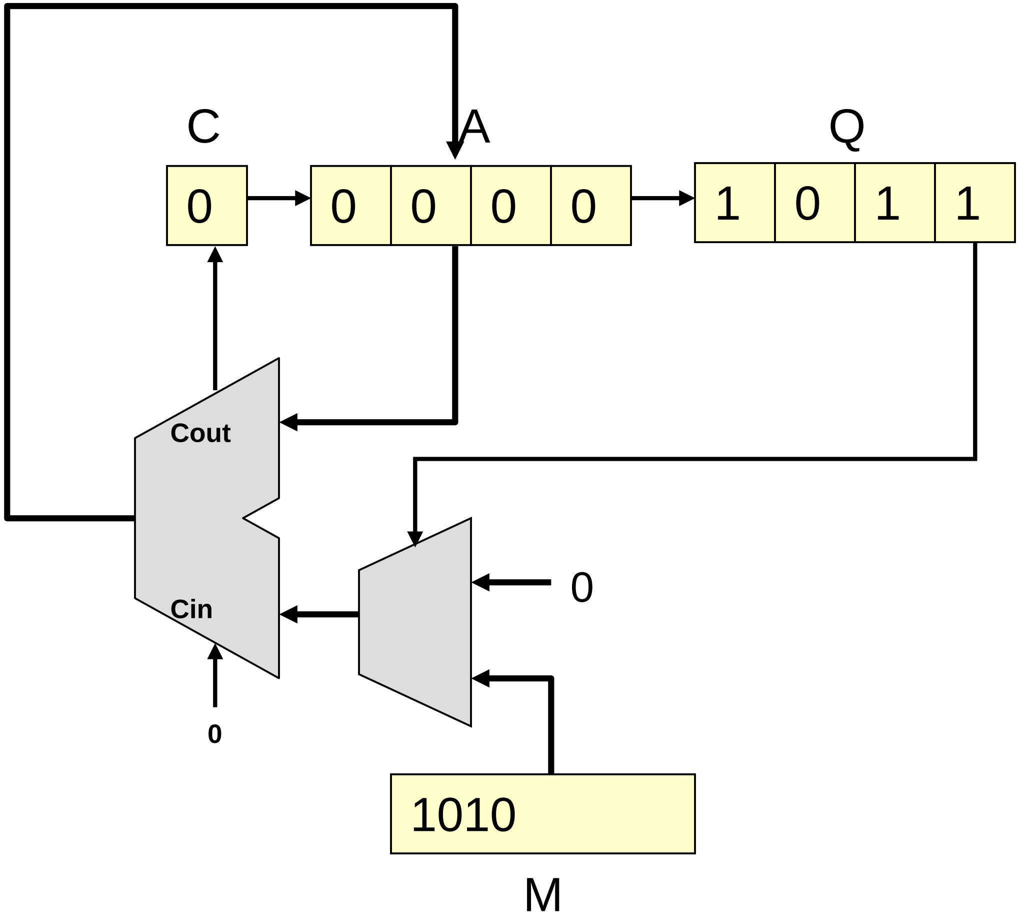
- Use 1 adder to add a single partial product per clock cycle keeping a running sum

Add and Shift Method

- Sequential algorithm
- n-bit * n-bit multiply
- Adds 1 partial product per clock
- Shift running sum 1-bit right each clock
- Three *n*-bit Registers, 1 Adder
- At start:
  - M = Multiplicand
  - Q = Multiplier
  - A = Answer => initialized to 0
- After completion
  - A and Q concatenate to form 2*n*-bit answer

Add and Shift Hardware

```
    1010  =  M
*   1011  =  Q
```



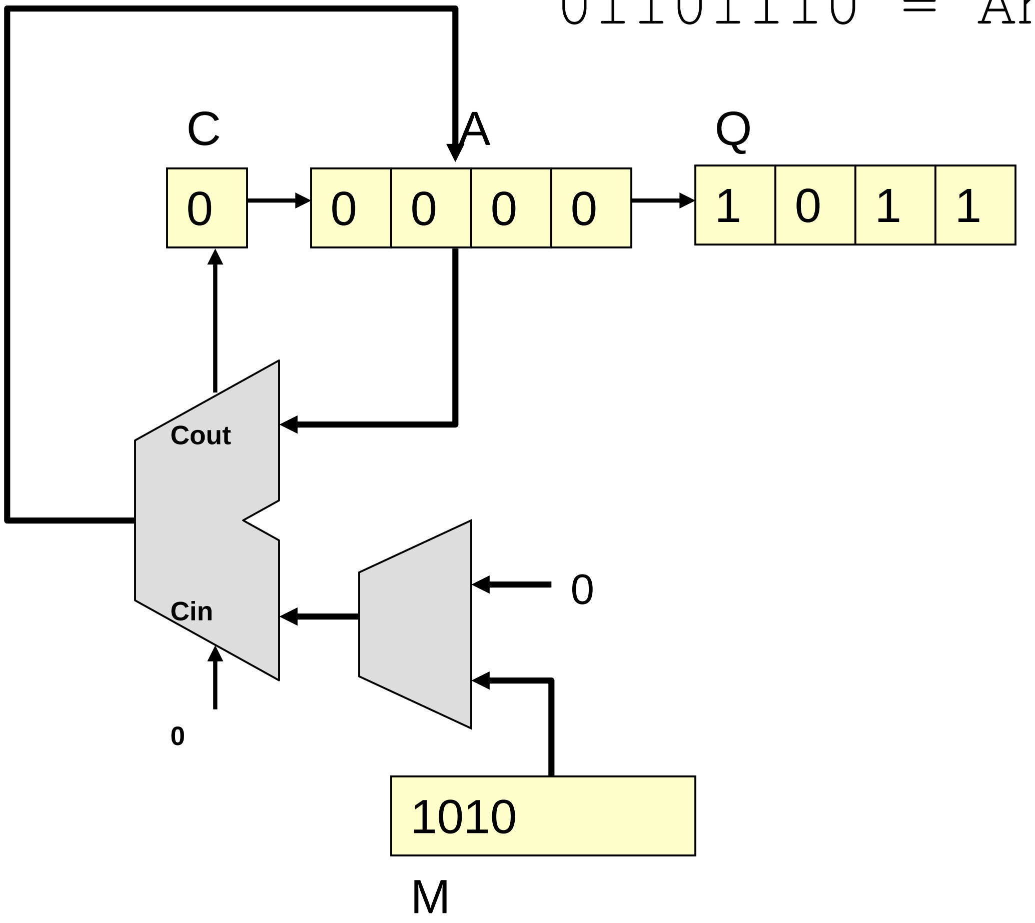
Add and Shift Algorithm

- C=0, A=0
- Repeat the following n-times
  - If Q[0] = 0, A = A+0
    Else if Q[0] = 1, A= A+M
  - Shift right 1-bit (0→C→A→Q)

$$\begin{array}{r} 1010 \\ \underline{*\ \ 1011} \end{array}$$

Add and Shift Multiplication

```
    1010  =  M
*   1011  =  Q
--------------
01101110  =  Ans
```



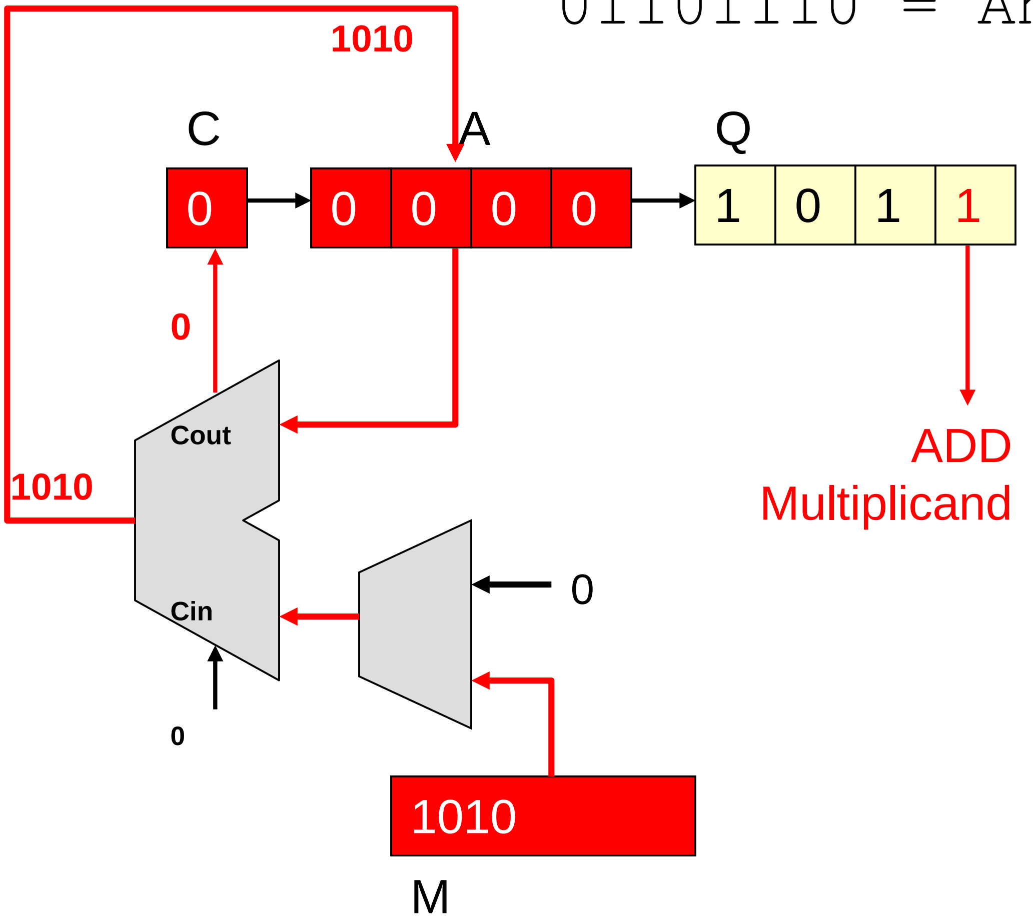
M = 1010

C	A	Q
0	0000	1011

Add and Shift Multiplication

```
    1010  =  M
*   1011  =  Q
------------------
01101110  =  Ans
```

```
     1010
  *  1011
  --------
  +  1010
  --------
     1010
```



ADD
Multiplicand

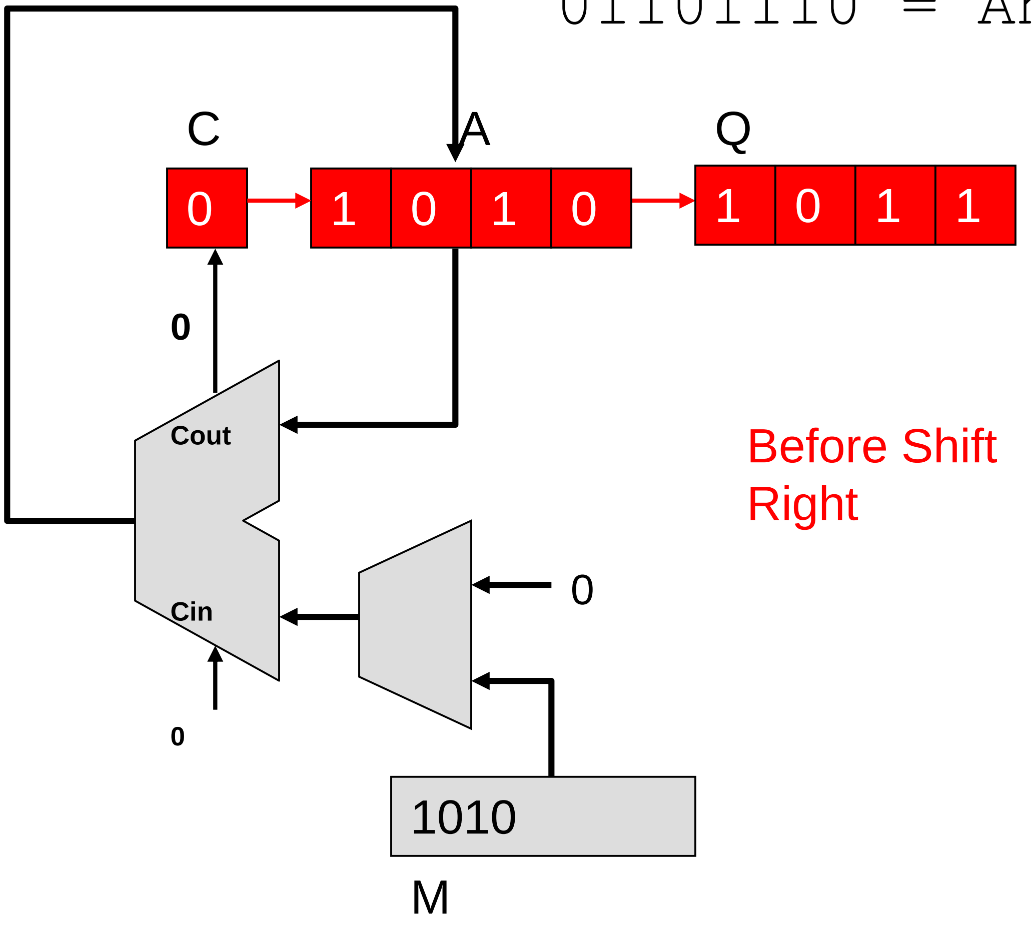
M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add

Add and Shift Multiplication

```
    1010  =  M
*   1011  =  Q
--------------
01101110  =  Ans
```

```
   1010
*  1011
-------
+  1010
-------
   1010
```



Before Shift Right

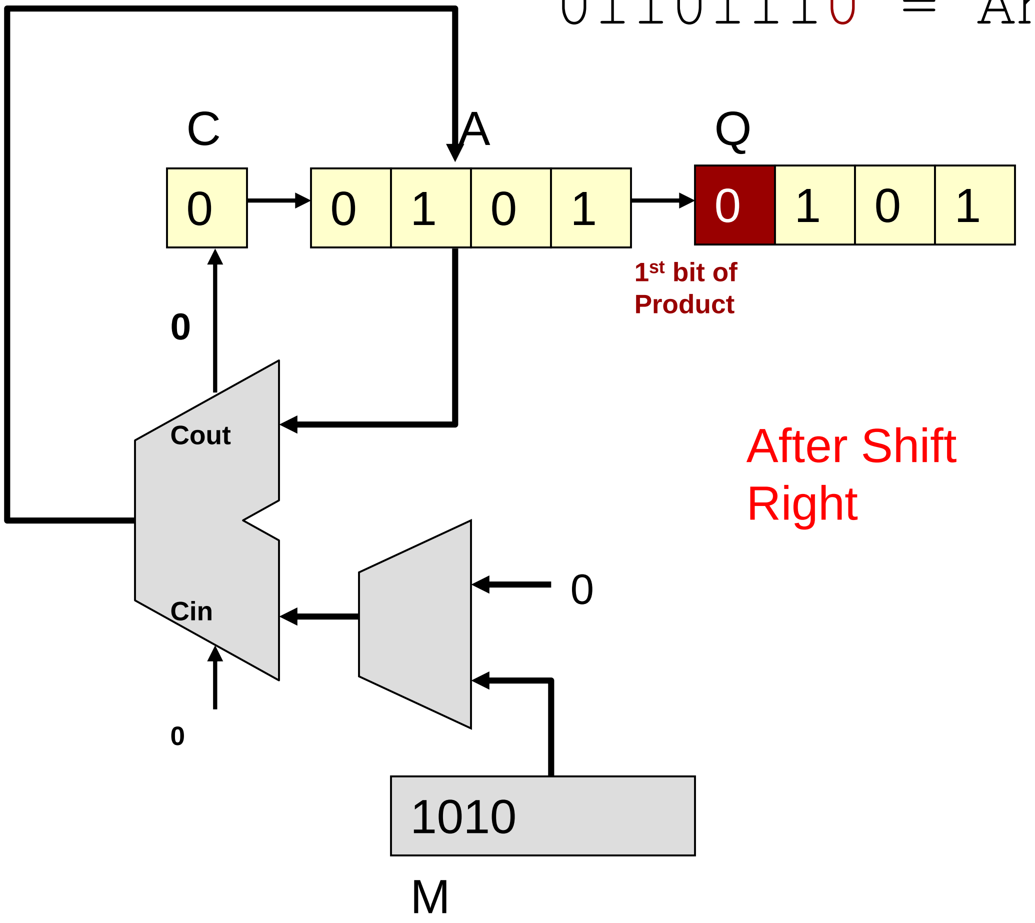
M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift

Add and Shift Multiplication

```
    1010  =  M
*   1011  =  Q
01101110  =  Ans
```

```
   1010
*  1011
+  1010
   1010
```



After Shift Right

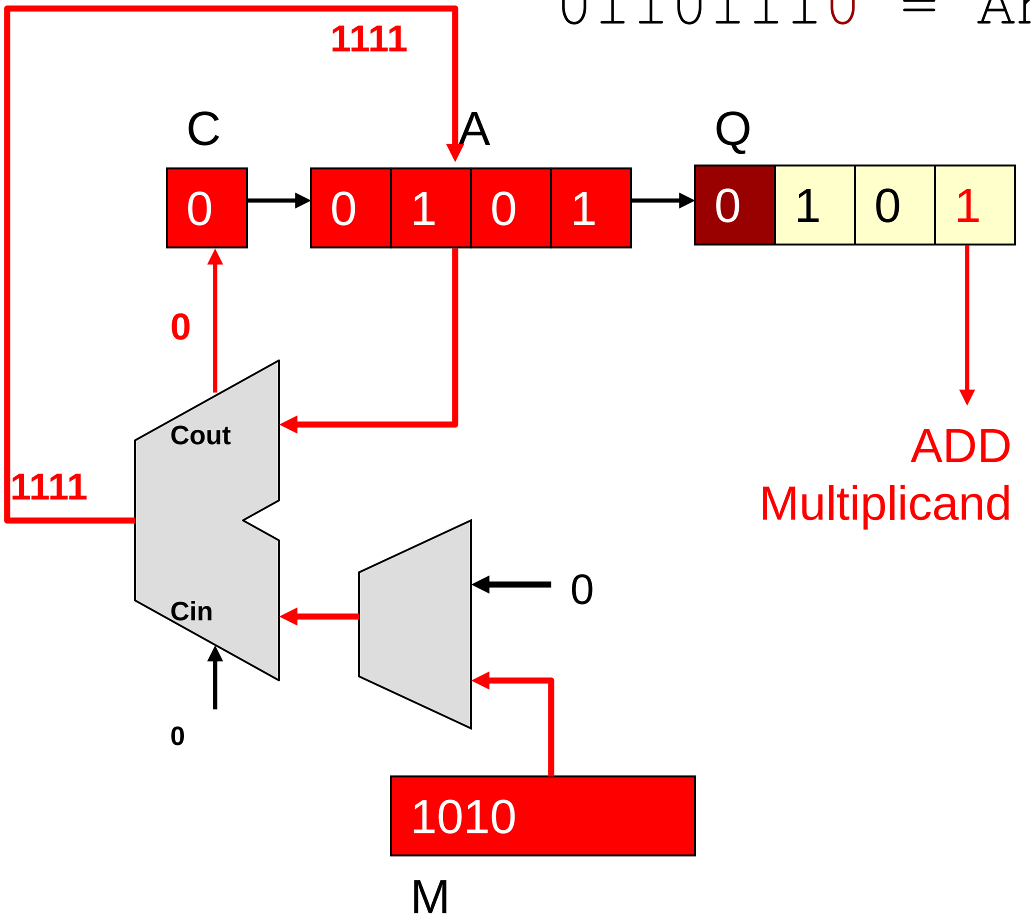
M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift

Add and Shift Multiplication

```
      1010   =  M
  *   1011   =  Q
  01101110   =  Ans
```

```
   1010
 * 1011
 ------
   1010
 ------
   1010
+ 1010-
-------
 011110
```



ADD Multiplicand

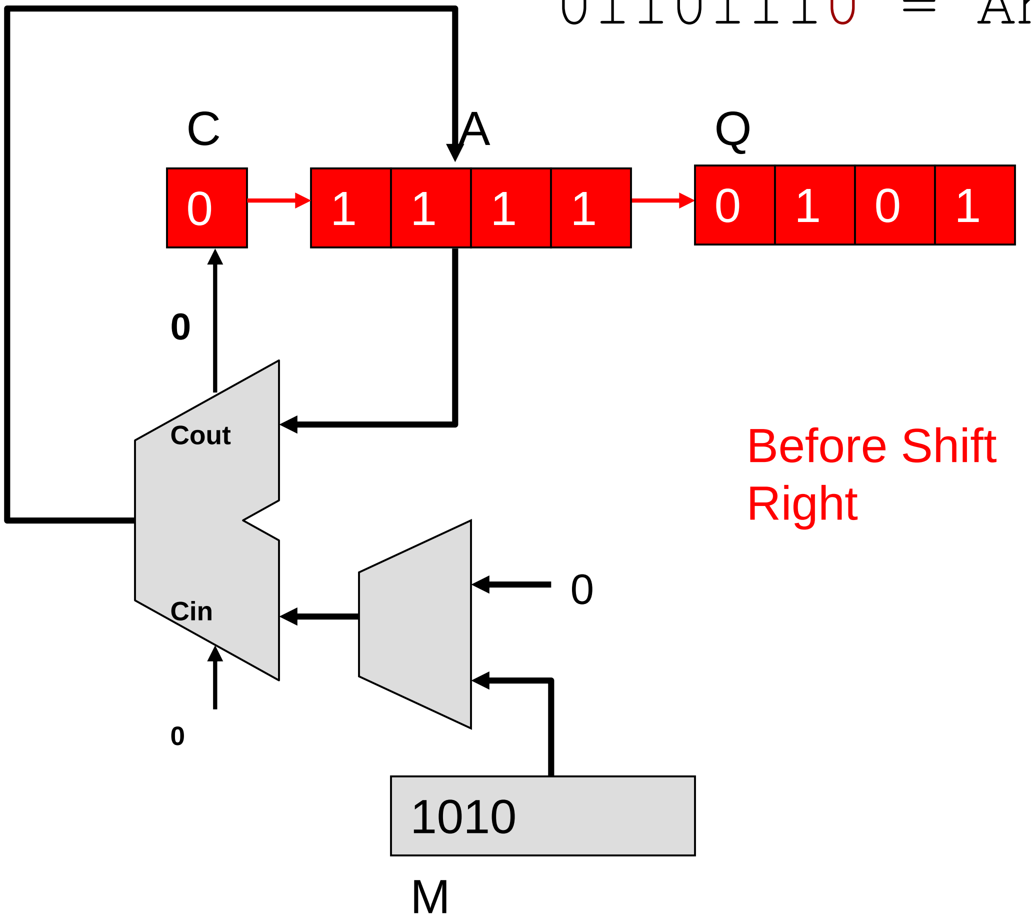
M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add

Add and Shift Multiplication

```
    1010 = M
*   1011 = Q
------------
01101110 = Ans
```

```
    1010
  * 1011
  ------
    1010
  ------
    1010
 + 1010-
 -------
  011110
```



Before Shift Right

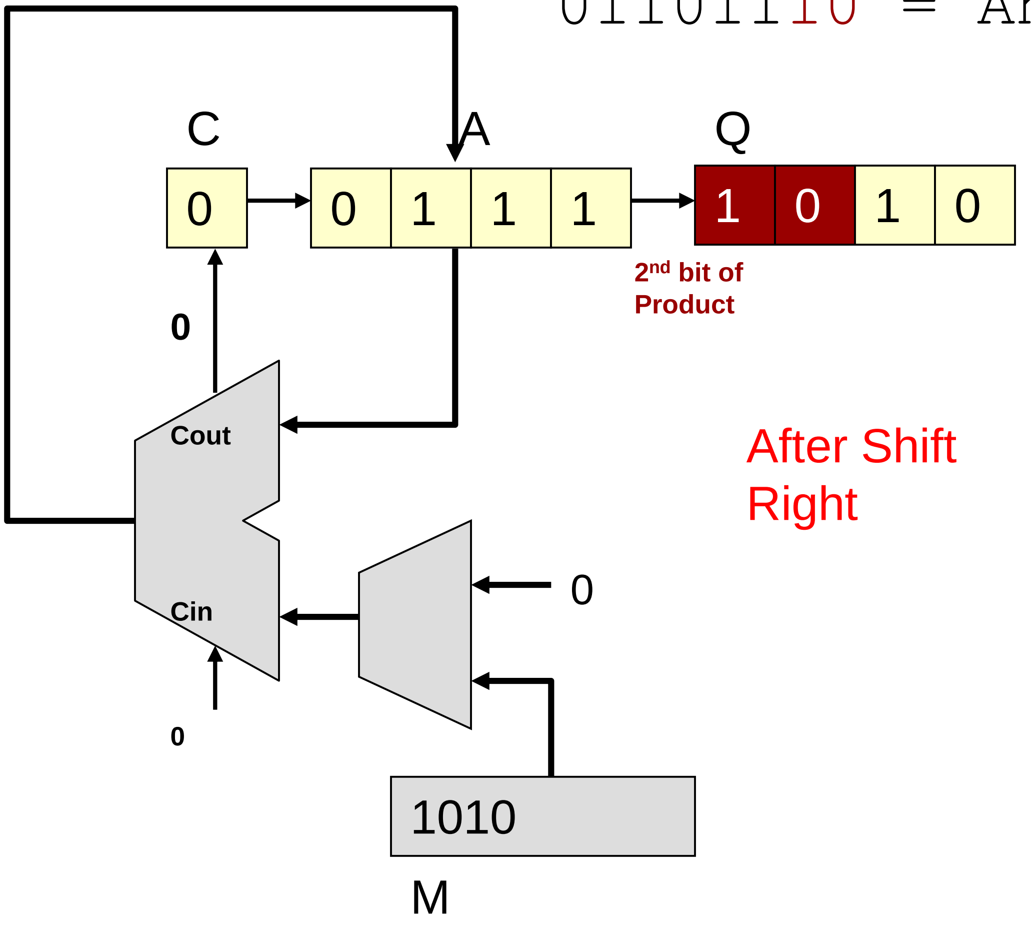
M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add
0	0111	1010	Shift

Add and Shift Multiplication

```
    1010 = M
 *  1011 = Q
--------
01101110 = Ans
```

```
   1010
 * 1011
 ------
   1010
 ------
   1010
 + 1010-
-------
 011110
```



2nd bit of Product

After Shift Right

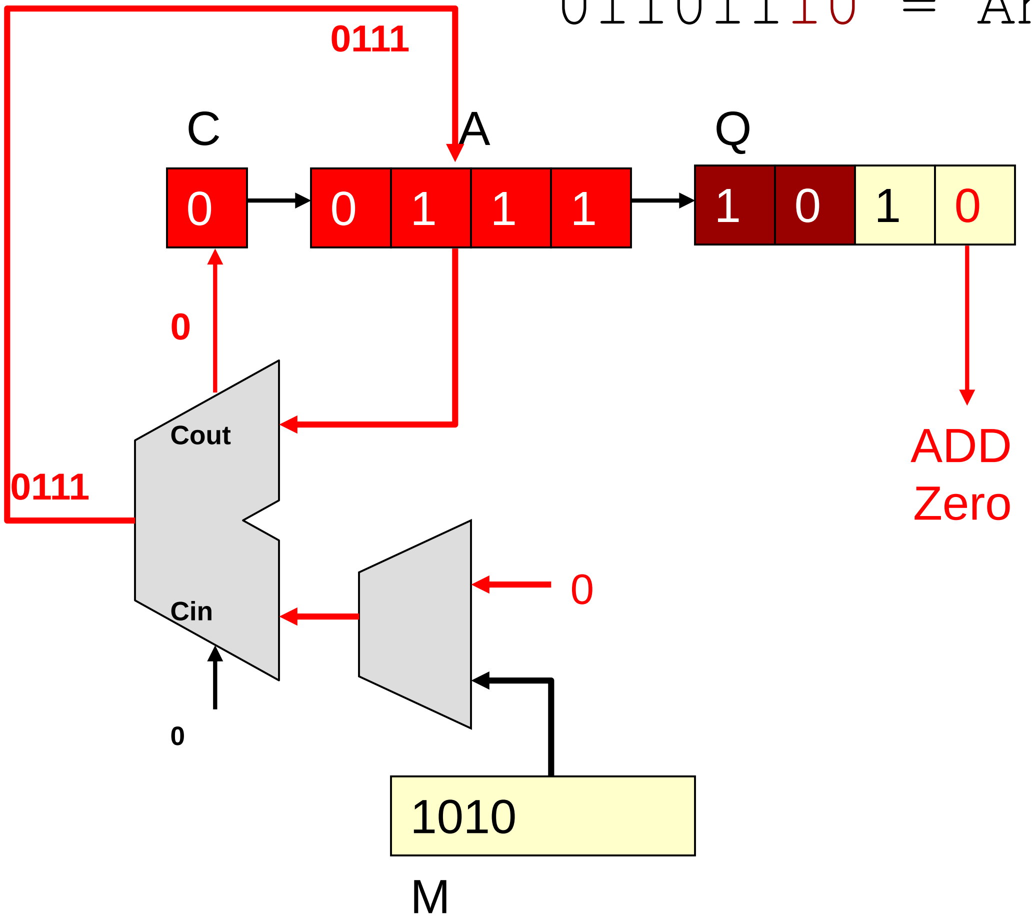
M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add
0	0111	1010	Shift

Add and Shift Multiplication

```
    1010  =  M
 *  1011  =  Q
 ------------
01101110  =  Ans
```

```
     1010
  *  1011
  -------
  +  1010
  -------
     1010
  + 1010-
  -------
   011110
 + 0000--
  -------
  0011110
```



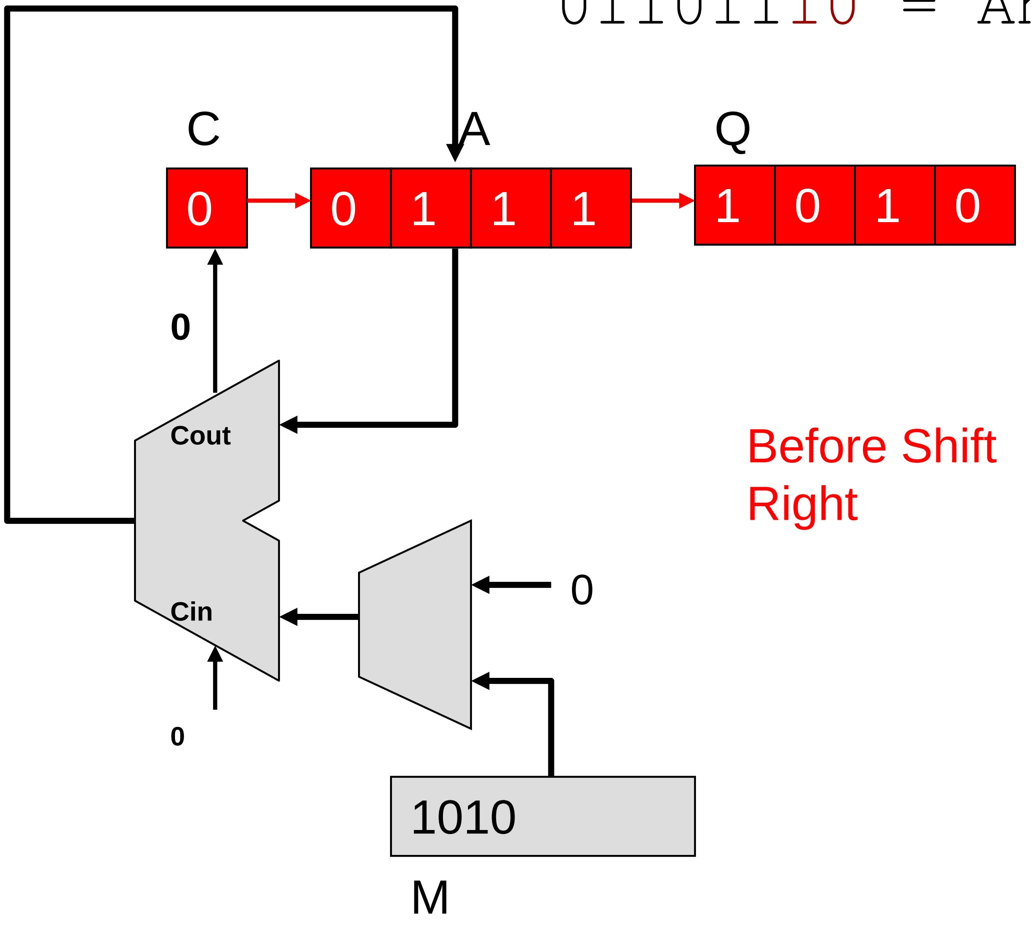
M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add
0	0111	1010	Shift
0	0111	1010	No Add

Add and Shift Multiplication

```
    1010  =  M
*   1011  =  Q
--------
01101110  =  Ans
```

```
    1010
  * 1011
--------
  + 1010
--------
    1010
 + 1010-
--------
  011110
+ 0000--
--------
 0011110
```



Before Shift Right

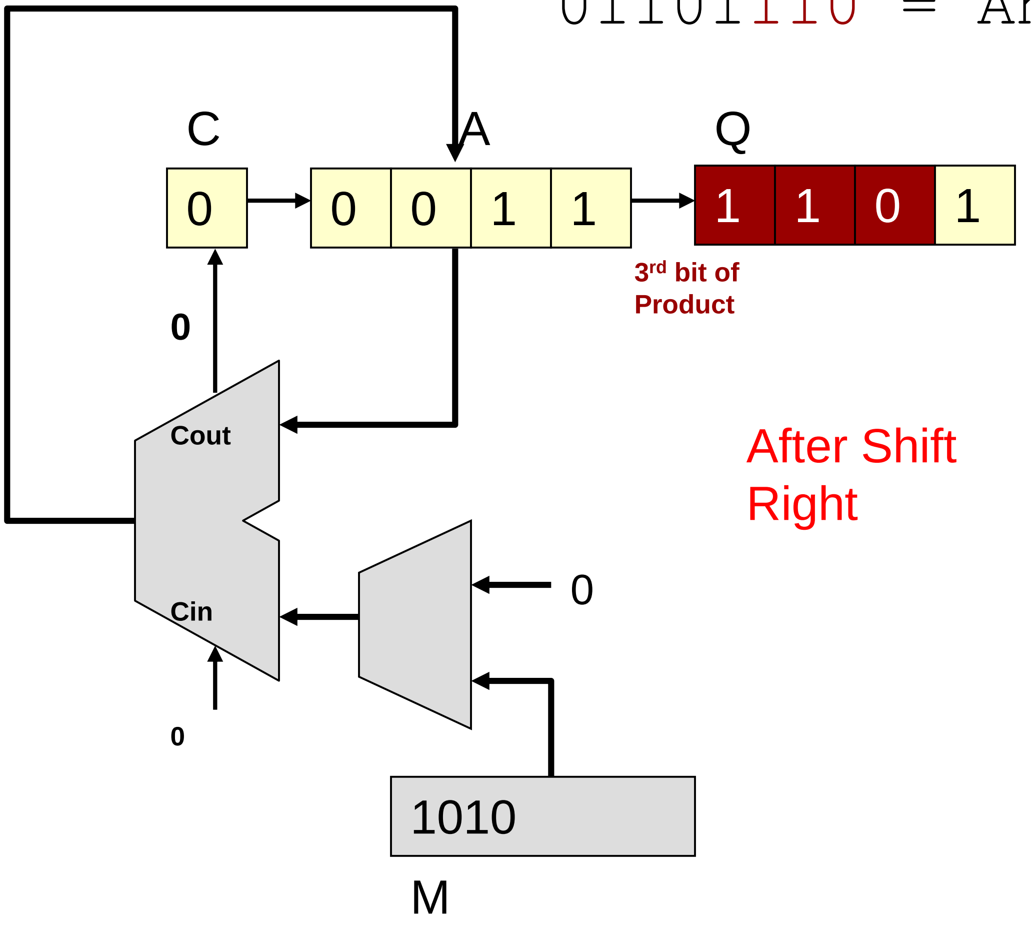
M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add
0	0111	1010	Shift
0	0111	1010	No Add
0	0011	1101	Shift

Add and Shift Multiplication

```
       1010  = M
   *   1011  = Q
   ------------
   01101110  = Ans
```

```
     1010
   * 1011
   ------
   + 1010
   ------
     1010
  + 1010-
  -------
   011110
 + 0000--
 --------
  0011110
```



3rd bit of Product

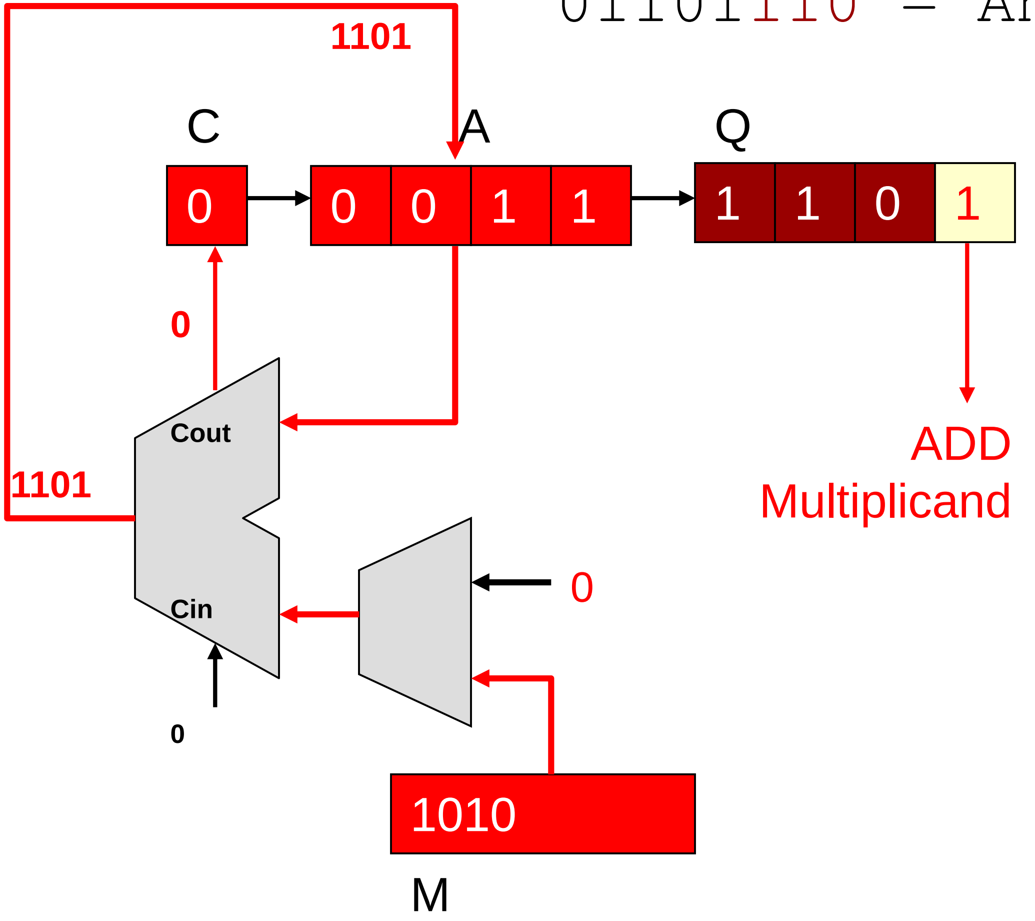
After Shift Right

M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add
0	0111	1010	Shift
0	0111	1010	No Add
0	0011	1101	Shift

Add and Shift Multiplication

```
    1010  =  M
*   1011  =  Q
-------------
01101110  =  Ans
```



```
      1010
    * 1011
  --------
    + 1010
  --------
      1010
   + 1010-
  --------
    011110
  + 0000--
  --------
   0011110
 + 1010---
  --------
  01101110
```

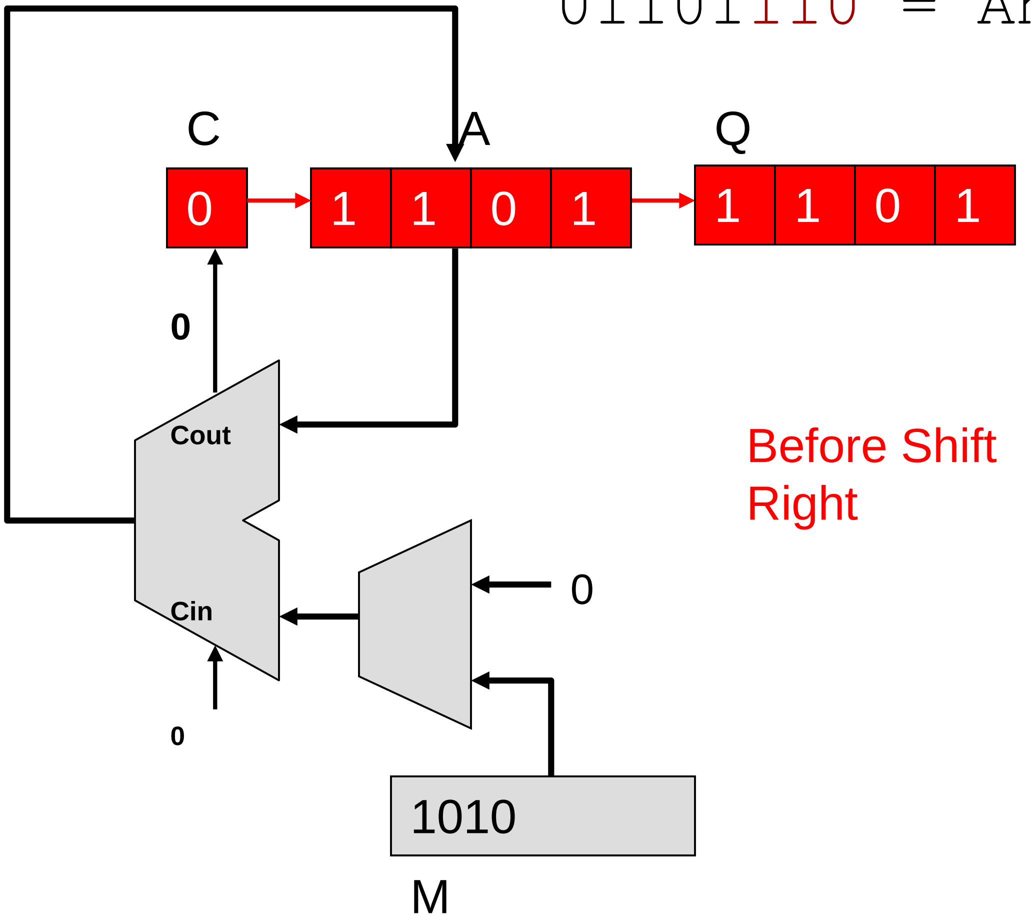
M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add
0	0111	1010	Shift
0	0111	1010	No Add
0	0011	1101	Shift
0	1101	1101	Add

Add and Shift Multiplication

```
    1010  =  M
*   1011  =  Q
--------------
01101110  =  Ans
```

```
      1010
    * 1011
  --------
    + 1010
  --------
      1010
   + 1010-
  --------
    011110
  +  0000--
  ---------
   0011110
 +  1010---
 ----------
  01101110
```



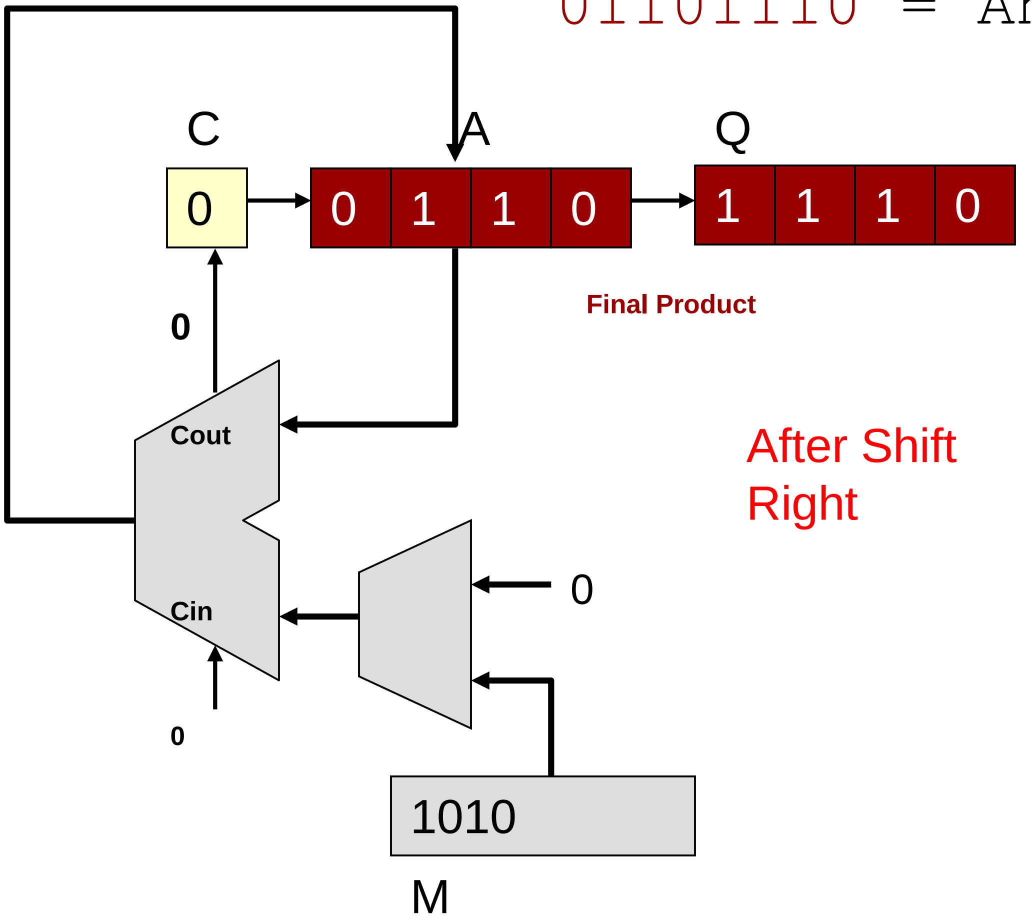
Before Shift Right

M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add
0	0111	1010	Shift
0	0111	1010	No Add
0	0011	1101	Shift
0	1101	1101	Add
0	0110	1110	Shift

Add and Shift Multiplication

```
        1010 = M
  *     1011 = Q
    01101110 = Ans
```



Final Product

After Shift Right

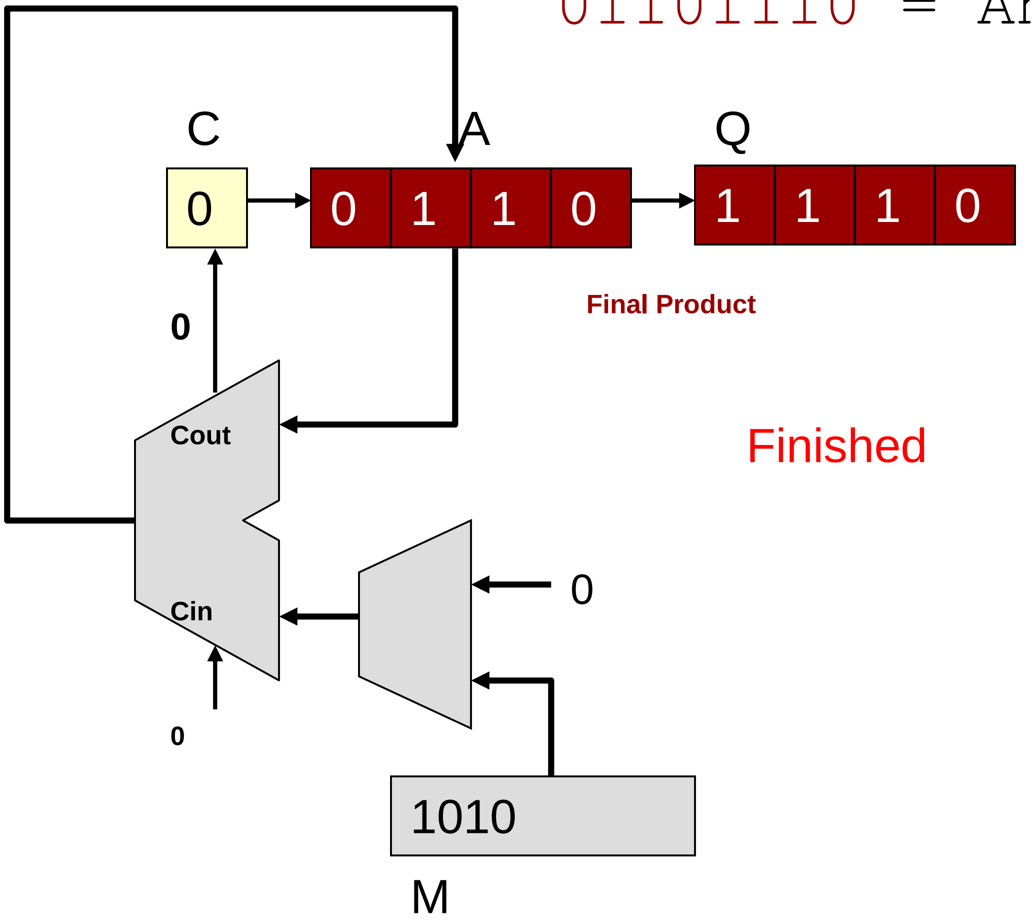
```
        1010
      * 1011
    --------
      + 1010
    --------
        1010
     + 1010-
    --------
      011110
    + 0000--
    --------
     0011110
   + 1010---
    --------
    01101110
```

M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add
0	0111	1010	Shift
0	0111	1010	No Add
0	0011	1101	Shift
0	1101	1101	Add
0	0110	1110	Shift

Add and Shift Multiplication

```
     1010  =  M
 *   1011  =  Q
 -------------
 01101110  =  Ans
```



Final Product

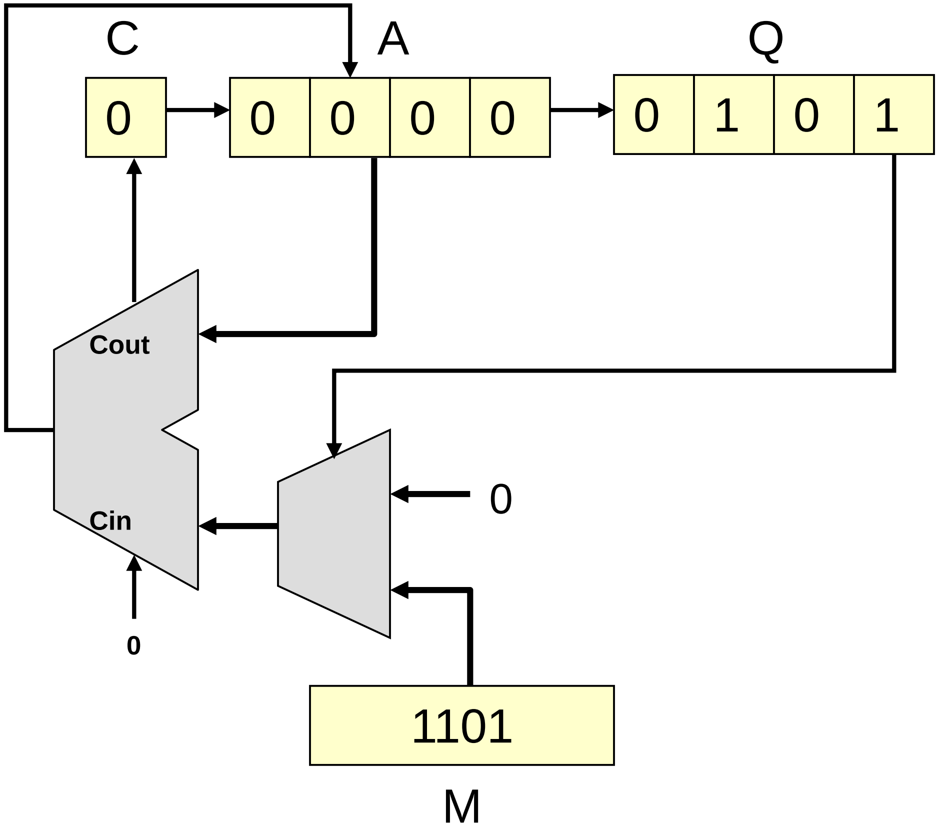
Finished

```
         1010
     *   1011
     --------
     +   1010
     --------
         1010
     +  1010-
     --------
       011110
     + 0000--
     --------
      0011110
     + 1010---
     --------
     01101110
```

M = 1010

C	A	Q	
0	0000	1011	
0	1010	1011	Add
0	0101	0101	Shift
0	1111	0101	Add
0	0111	1010	Shift
0	0111	1010	No Add
0	0011	1101	Shift
0	1101	1101	Add
0	0110	1110	Shift
	0110	1110	= 110_{10}

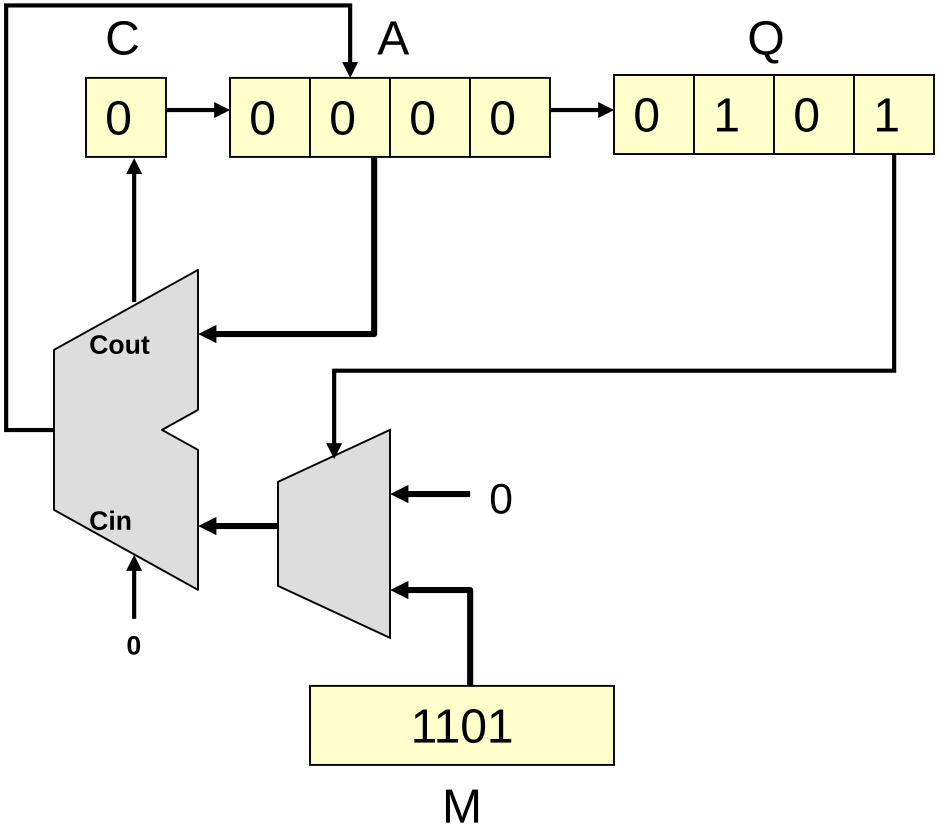
1101 * 0101 Example



C=0	M=1101
	A=0000
		Q=0101	Description

0	1101	0101	A=A+M
			Shift Right C,A,Q
			A=A+0
			Shift Right C,A,Q
			A=A+M
			Shift Right C,A,Q
			A=A+0
			Shift Right C,A,Q

1101 * 0101 Example



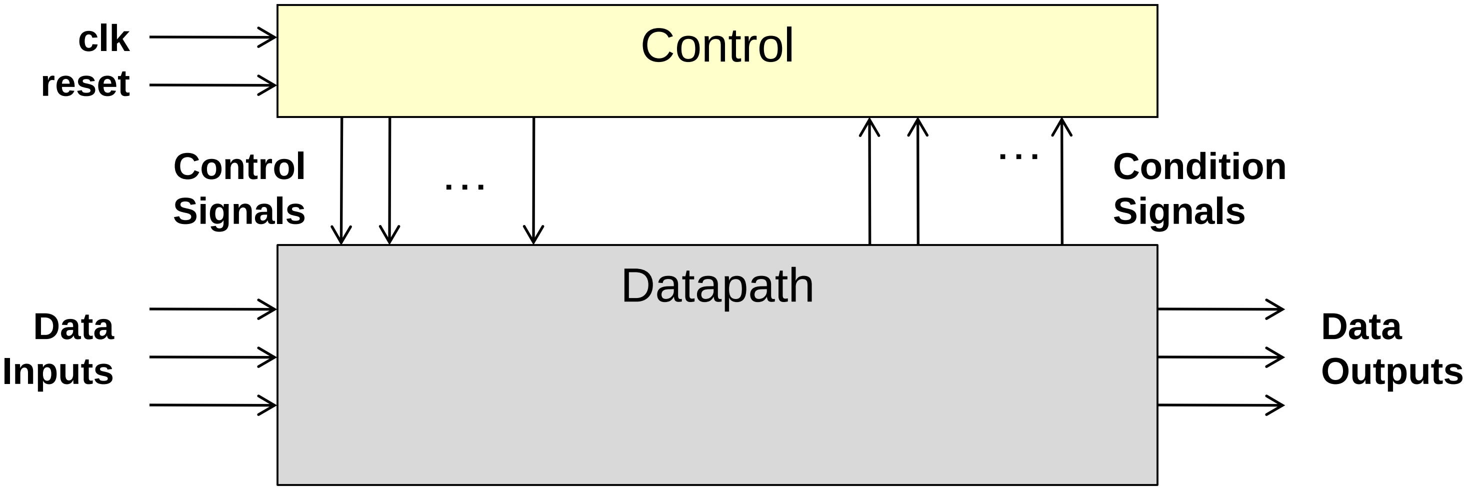
	M=1101		
C=0	**A=0000**	**Q=0101**	**Description**
0	1101	0101	A=A+M
0	0110	1010	Shift Right C,A,Q
0	0110	1010	A=A+0
0	0011	0101	Shift Right C,A,Q
1	0000	0101	A=A+M
0	1000	0010	Shift Right C,A,Q
0	1000	0010	A=A+0
0	0100	0001	Shift Right C,A,Q

Sequential Multiplier Analysis

- Pros:
  - Smaller Area due to the use of only 1 adder
- Cons:
  - Slow to execute (2 cycles per bit of the multiplier)

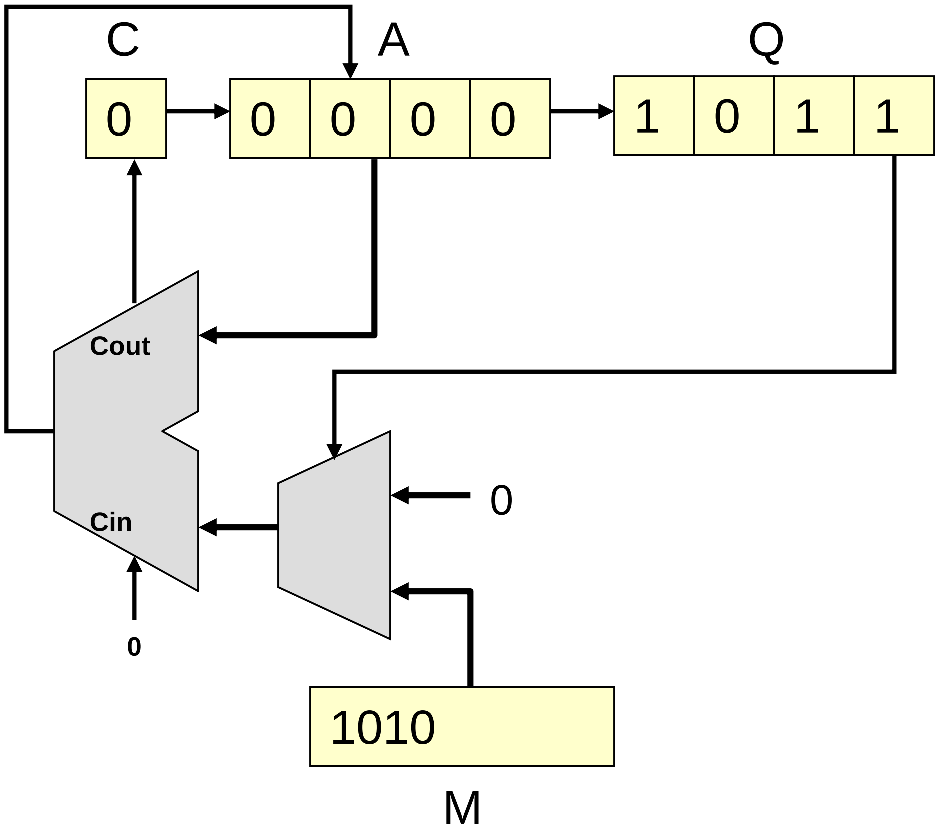
Digital System Design

- Control and Datapath Unit paradigm
  - Separate logic into datapath elements that operate on data and control elements that generate control signals for datapath elements
  - Datapath: Adders, muxes, comparators, counters, registers (w/ enables)
  - Control Unit: State machines/sequencers



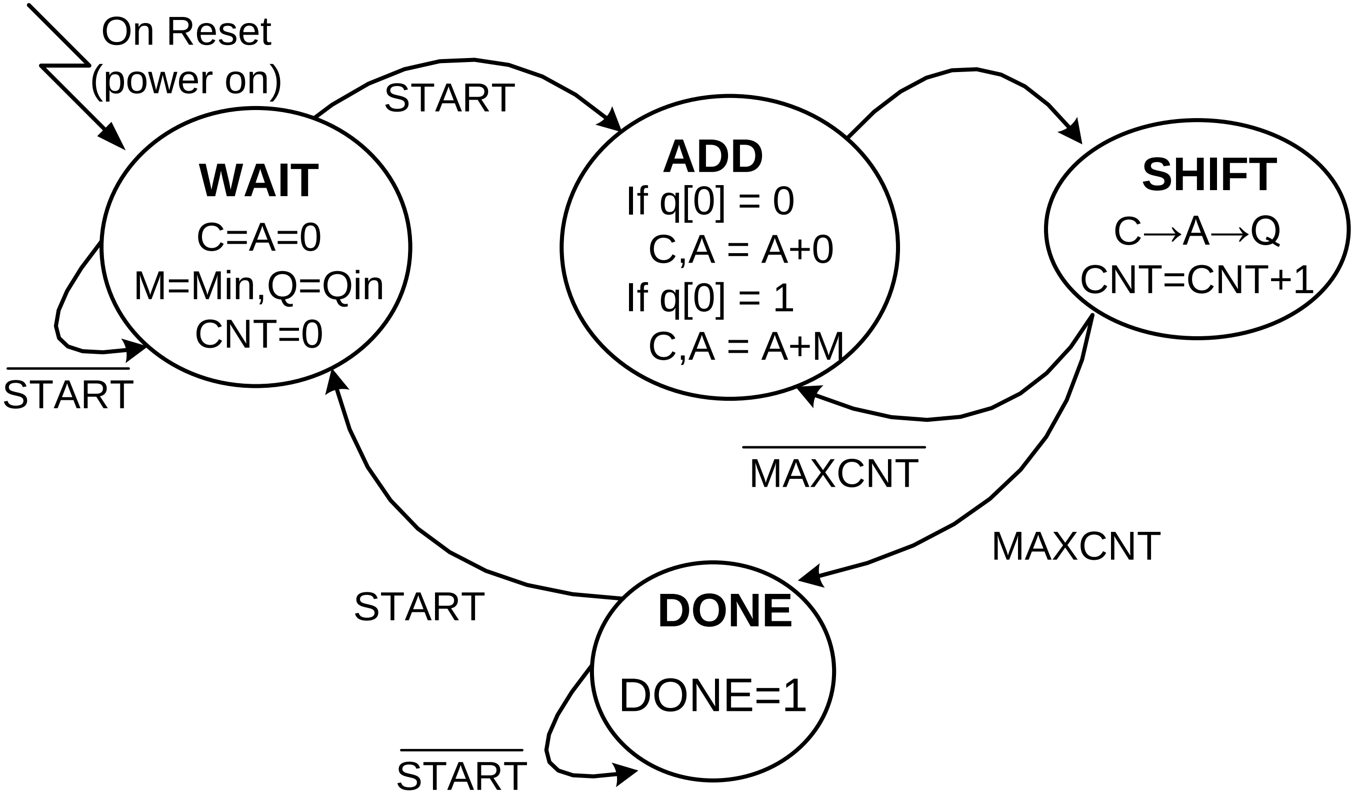
Let's Practice our Design Skills

- Break design into control and datapath
  - This is the datapath
  - 1 Adder
  - 2-to-1 mux
  - 2 shift registers (A/Q)
  - 1 normal reg (M)
  - 1 FF w/ Enable (C)



State Machine Control

- From our high level datapath we can arrive at a high-level state diagram

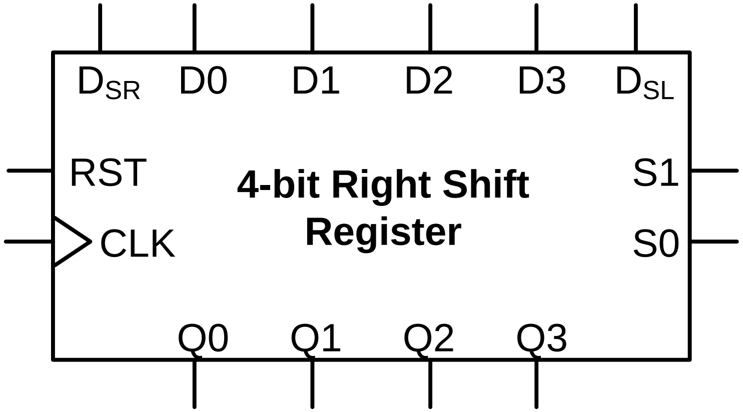


Refining our Design

- But now we need to refine our design to actual components, specific control bits, etc.

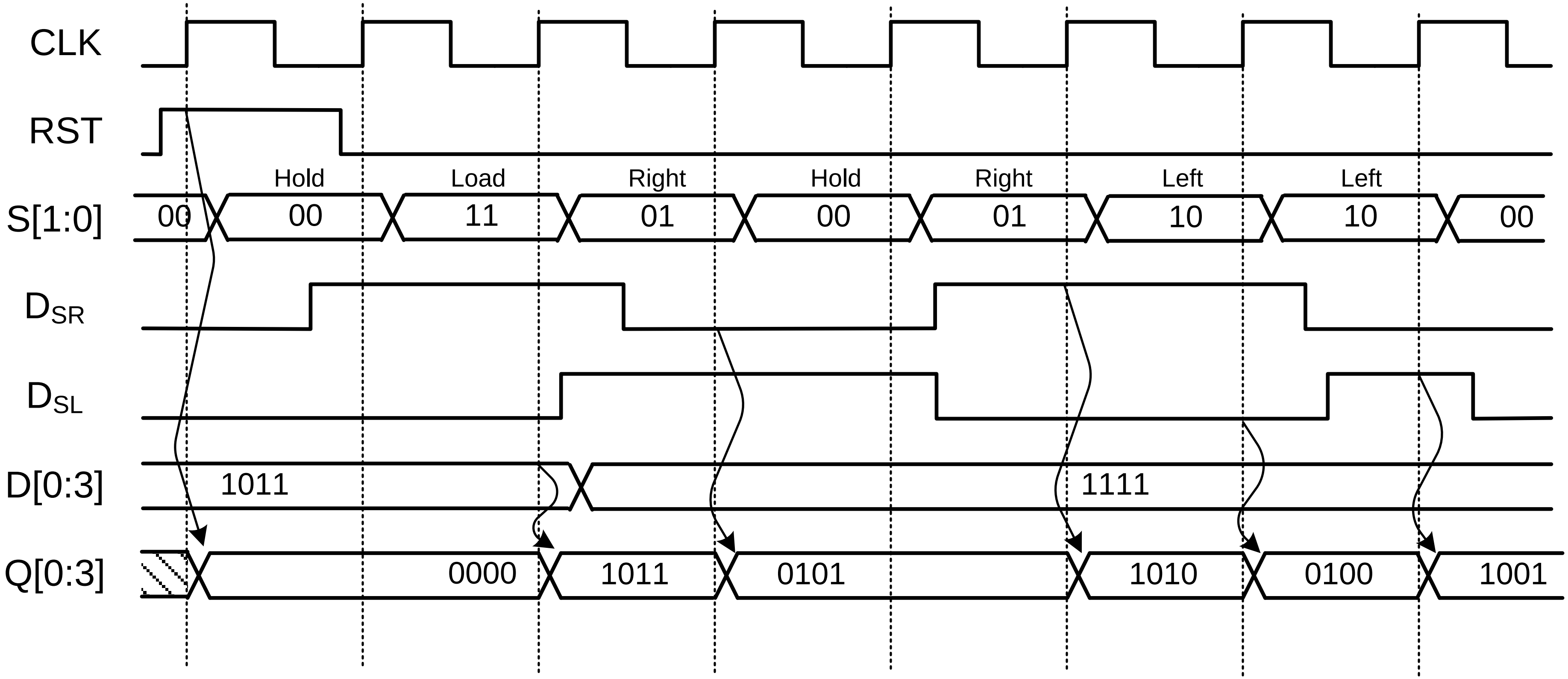
Sample Shift Register

- Shift registers come in many flavors, we'll just look at one example
- 4-bit Bi-directional Shift Register
  - RST: synchronous reset
  - S[1:0]: Hold, Right Shift, Left Shift, or Load
  - DSL and DSR
    - Data to shift in from left or right



CLK	ACLR	S1	S0	Q*[3:0]	(case)
0,1	X	X	X	Q[3:0]	
⇈	1	X	X	0000	Reset
⇈	0	0	0	Q[3:0]	Hold
⇈	0	0	1	D_{SR},Q[3:1]	Right
⇈	0	1	0	Q[2:0],D_{SL}	Left
⇈	0	1	1	D[0:3]	Load

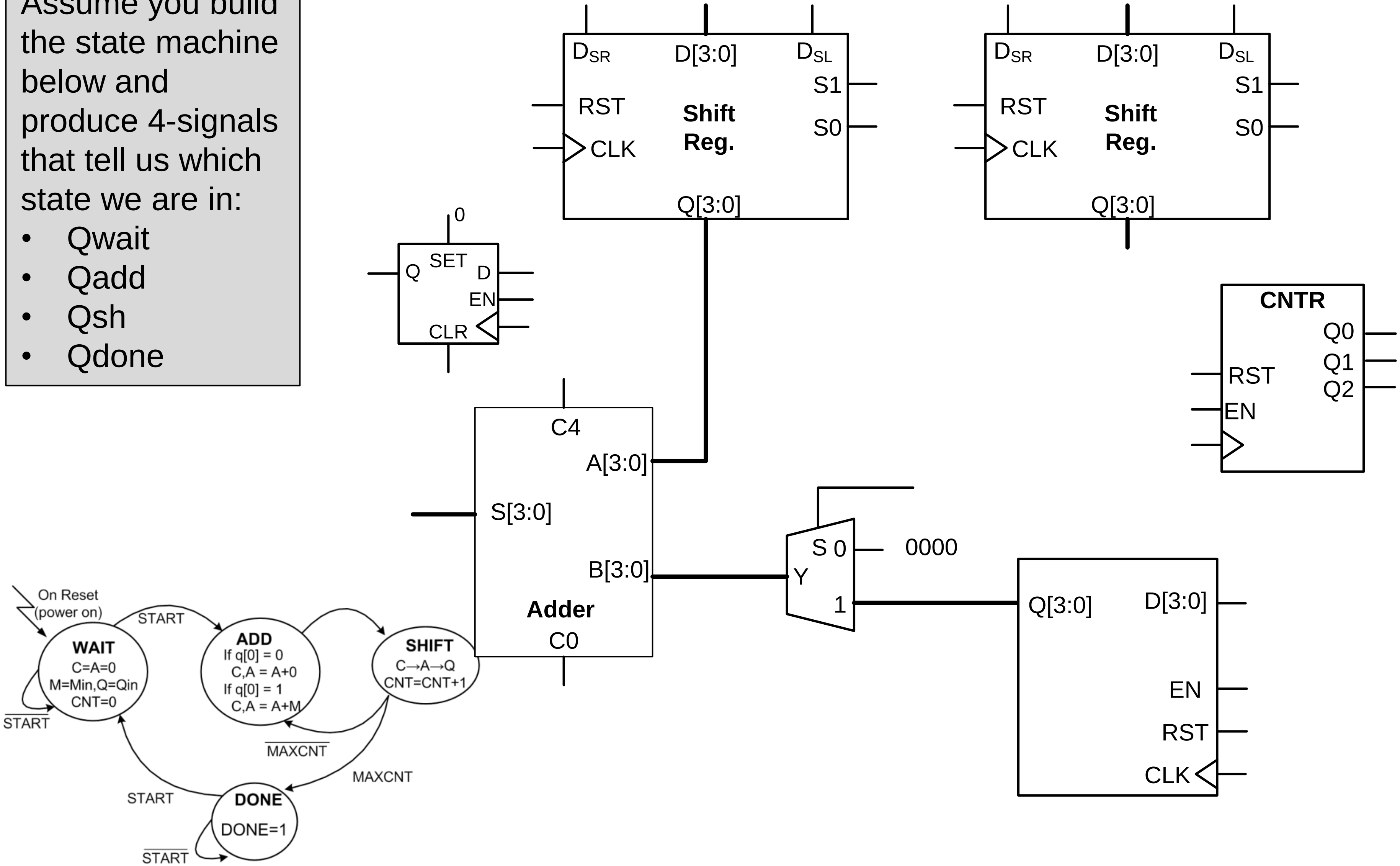
Shift Registers



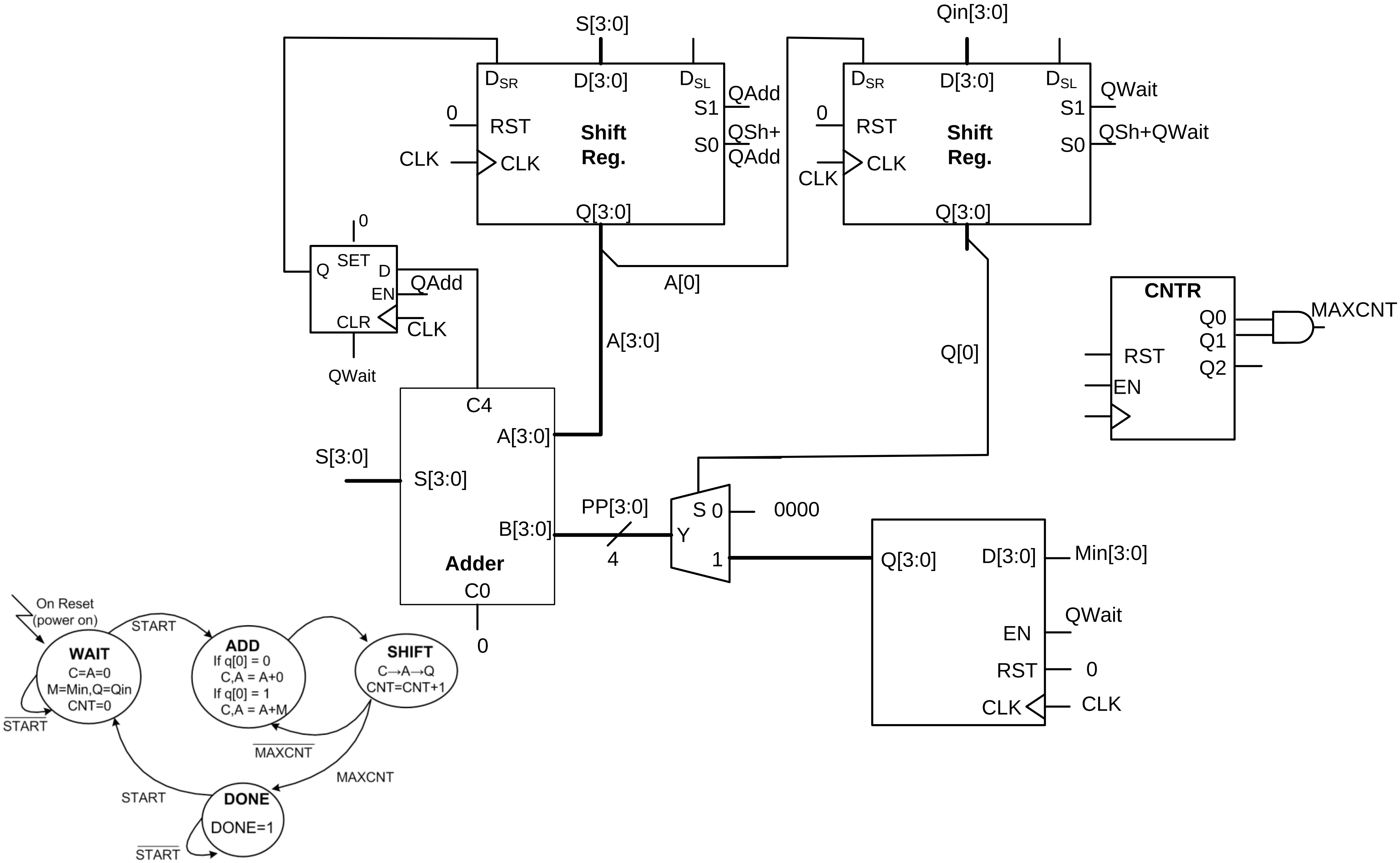
Complete the DataPath

Assume you build the state machine below and produce 4-signals that tell us which state we are in:

- Qwait
- Qadd
- Qsh
- Qdone



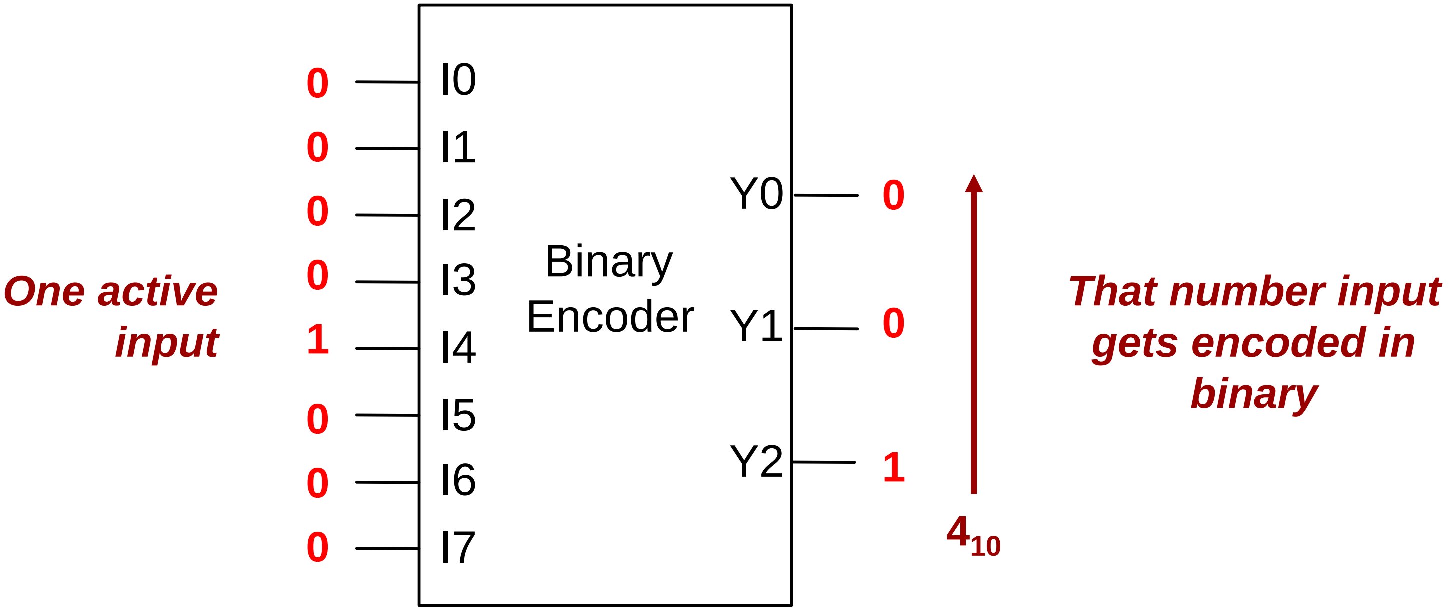
Complete the DataPath



SIMPLE & PRIORITY ENCODERS

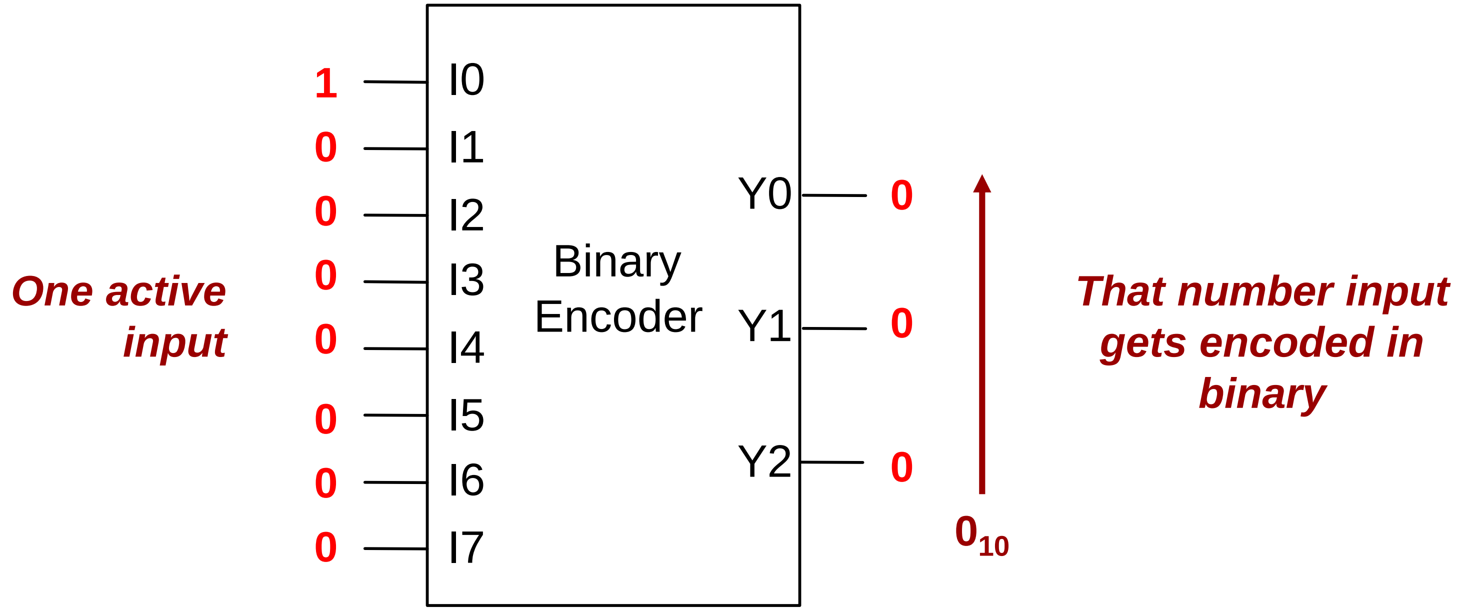
Encoders

- Another common datapath component
- Opposite function of decoders
- Takes in 2^n inputs and produces an n-bit number



Encoders

- Assumption: Only one input will be active at a time



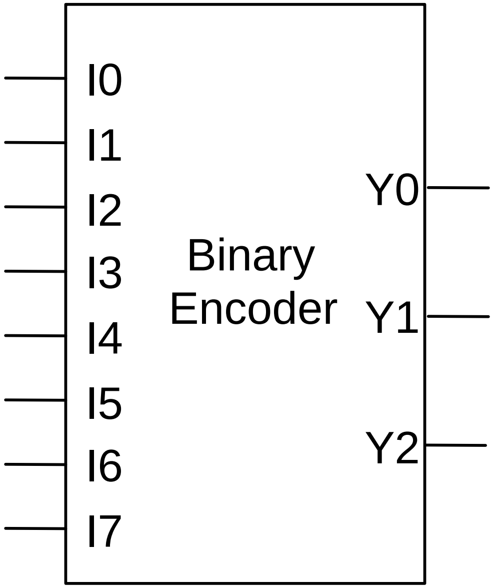
One active input

That number input gets encoded in binary

Encoders

- What's inside an encoder?

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	Y_2	Y_1	Y_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

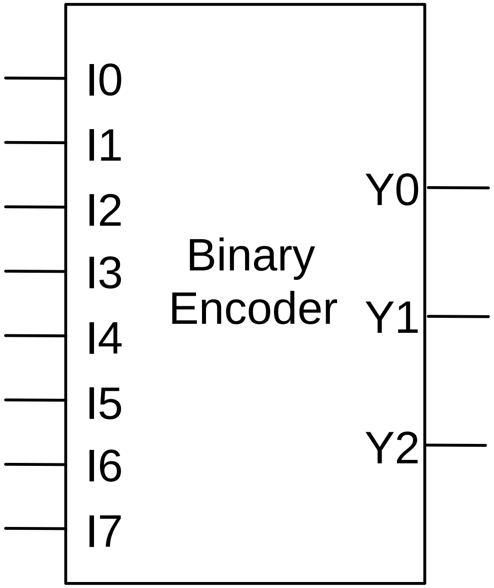


Deriving equations for Y_0, Y_1, Y_2 is made simpler because of the assumption that only 1 input can be active at a time. Rather than having 256 rows in our truth table we only have 8

Encoders

- What's inside an encoder?

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	Y_2	Y_1	Y_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1



Y_2 = 1 when I_4 = 1 or I_5 = 1 or I_6 = 1 or I_7 = 1…

Y_2 = I4 + I5 + I6 + I7

Encoders

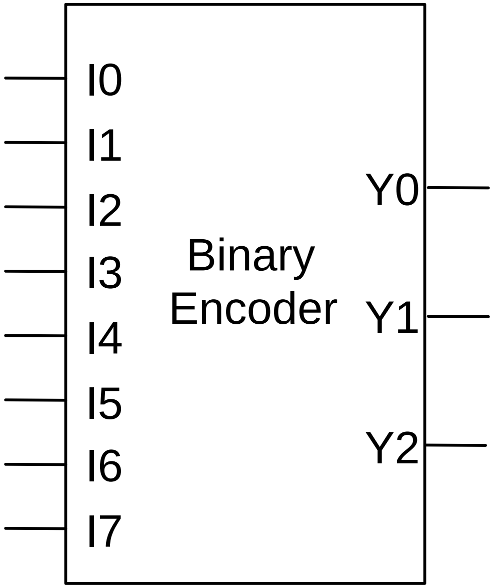
- What's inside an encoder?

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	Y_2	Y_1	Y_0
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Y_2 = I4 + I5 + I6 + I7

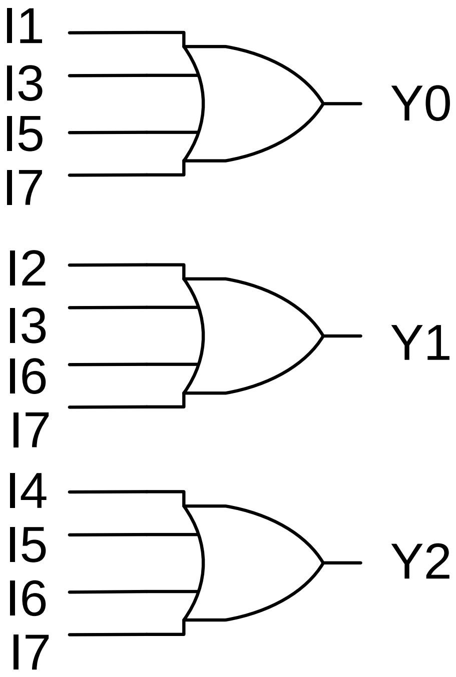
Y_1 = I2 + I3 + I6 + I7

Y_0 = I1 + I3 + I5 + I7



Encoders

- A simple binary encoder can be made with just OR gates

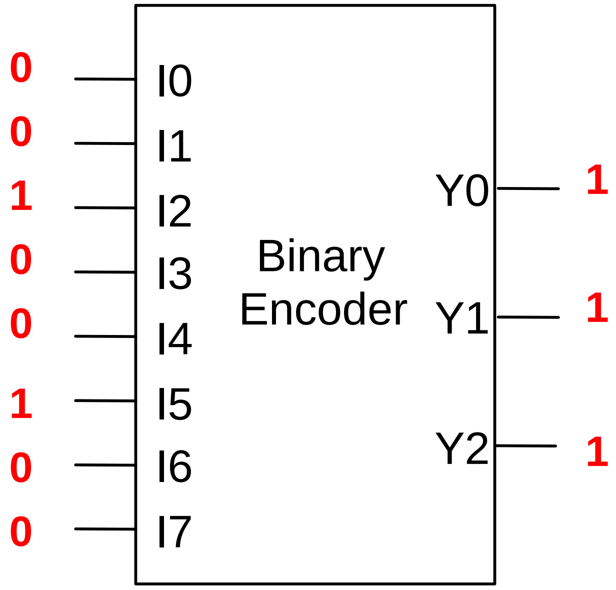


Problems

- There is a problem…
  - Our assumption is that only 1 input can be active at a time
  - What happens if 2 or more inputs are active or if 0 inputs are active

2 or More Active Inputs

- What if I5 and I2 are active at the same time?
  - Substitute values into equation
- Output will be ‘111’ = 7
- Output is neither 2 nor 5, it’s something different, 7



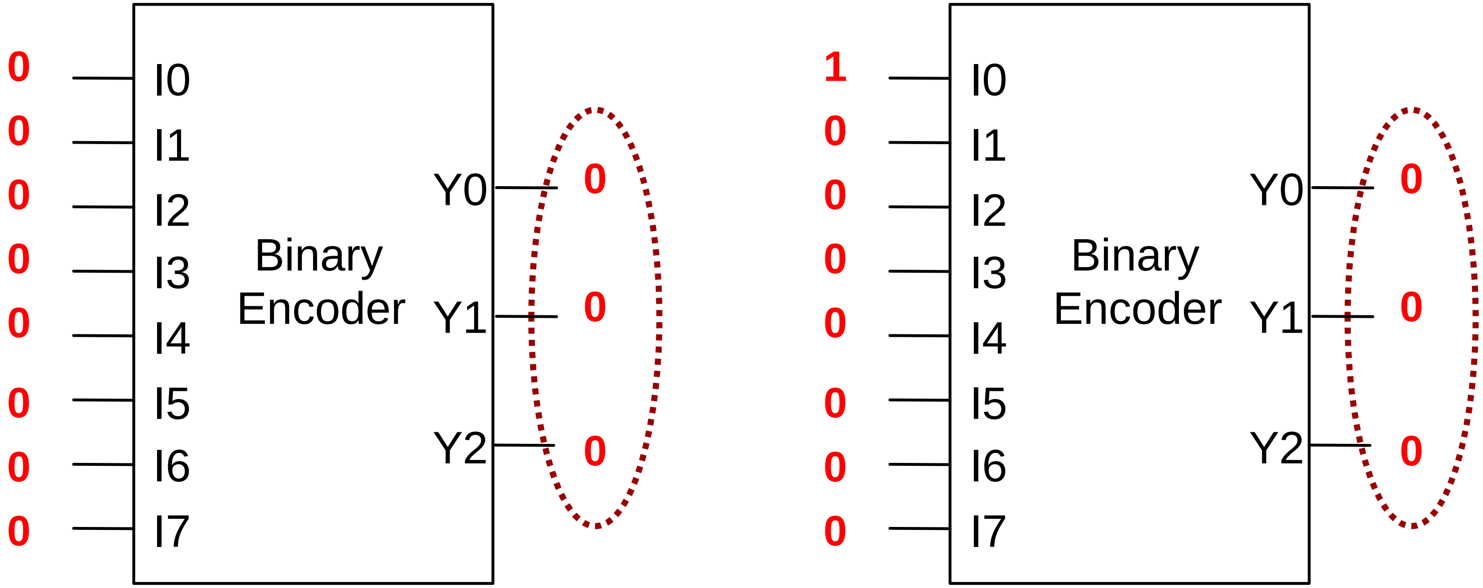
Y_2 = I4 + I5 + I6 + I7

Y_1 = I2 + I3 + I6 + I7

Y_0 = I1 + I3 + I5 + I7

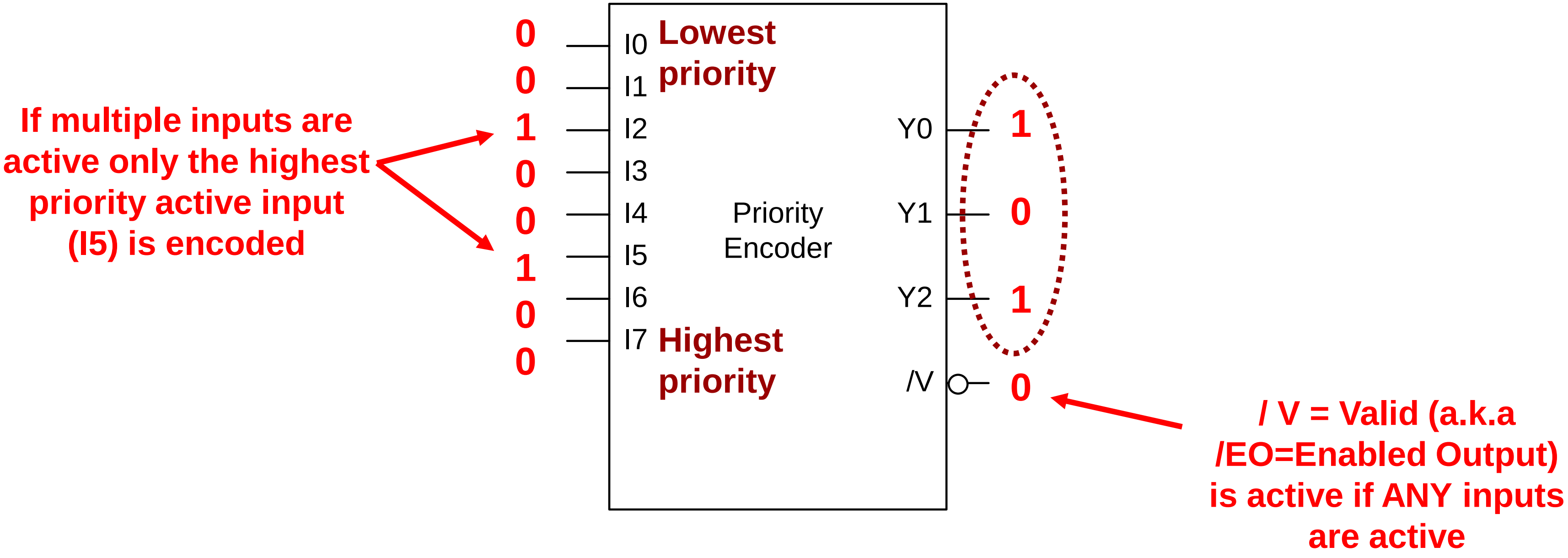
0 Active Inputs

- What if no inputs are active?
  - Substitute values into equation
- Output will be ‘000’ = 0
- Problem: ‘000’ means that input 0 was active
  - Can’t tell the difference between when ‘000’ means input 0 was active or no inputs was active



Priority Encoders

- Fix the 2 problems seen above
- Problem of more than 2 active inputs
  - Assign priority to inputs and only encode the highest priority active input
- Problem of zero active inputs
  - Create an extra output to indicate if any inputs are active
  - We will call this output the “Valid” output (/V)



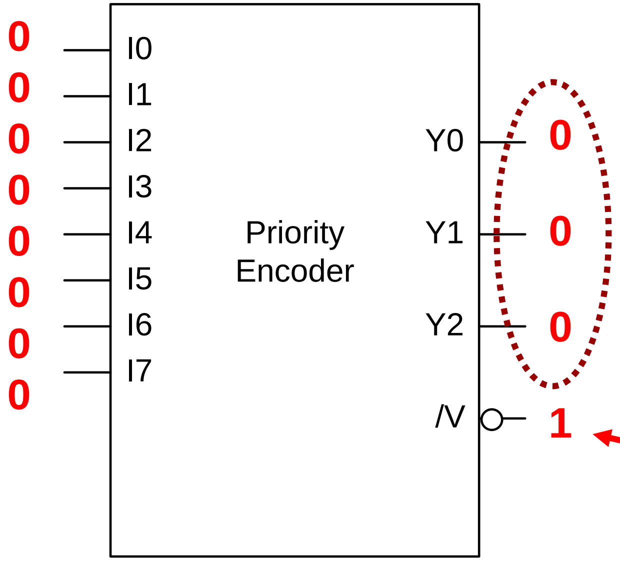
If multiple inputs are active only the highest priority active input (I5) is encoded

/ V = Valid (a.k.a /EO=Enabled Output) is active if ANY inputs are active

Priority Encoders

- Fix the 2 problems seen above
- Problem of more than 2 active inputs
  - Assign priority to inputs and only encode the highest priority active input
- Problem of zero active inputs
  - Create an extra output to indicate if any inputs are active

No inputs are active

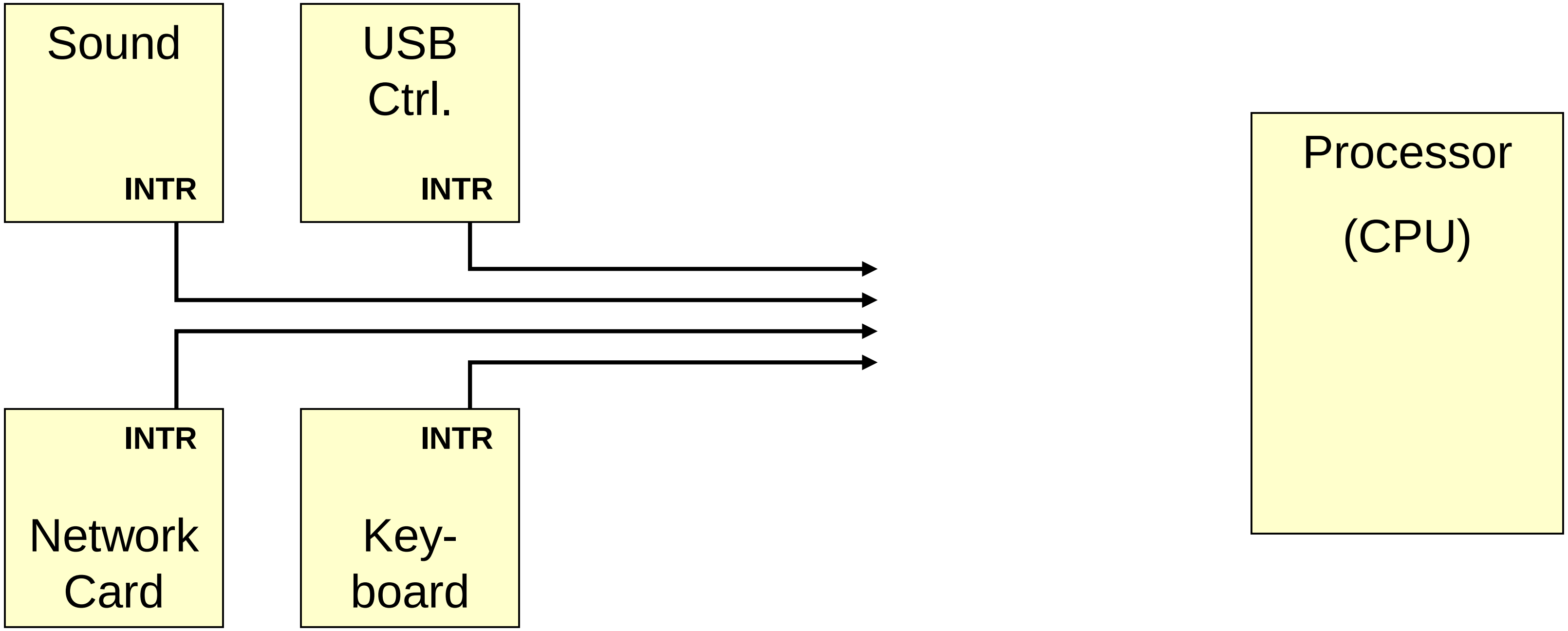


Output is still 000 but /V tells us that this is not because I0 was active

/ V = Valid (a.k.a /EO=Enabled Output) is inactive if no inputs are active

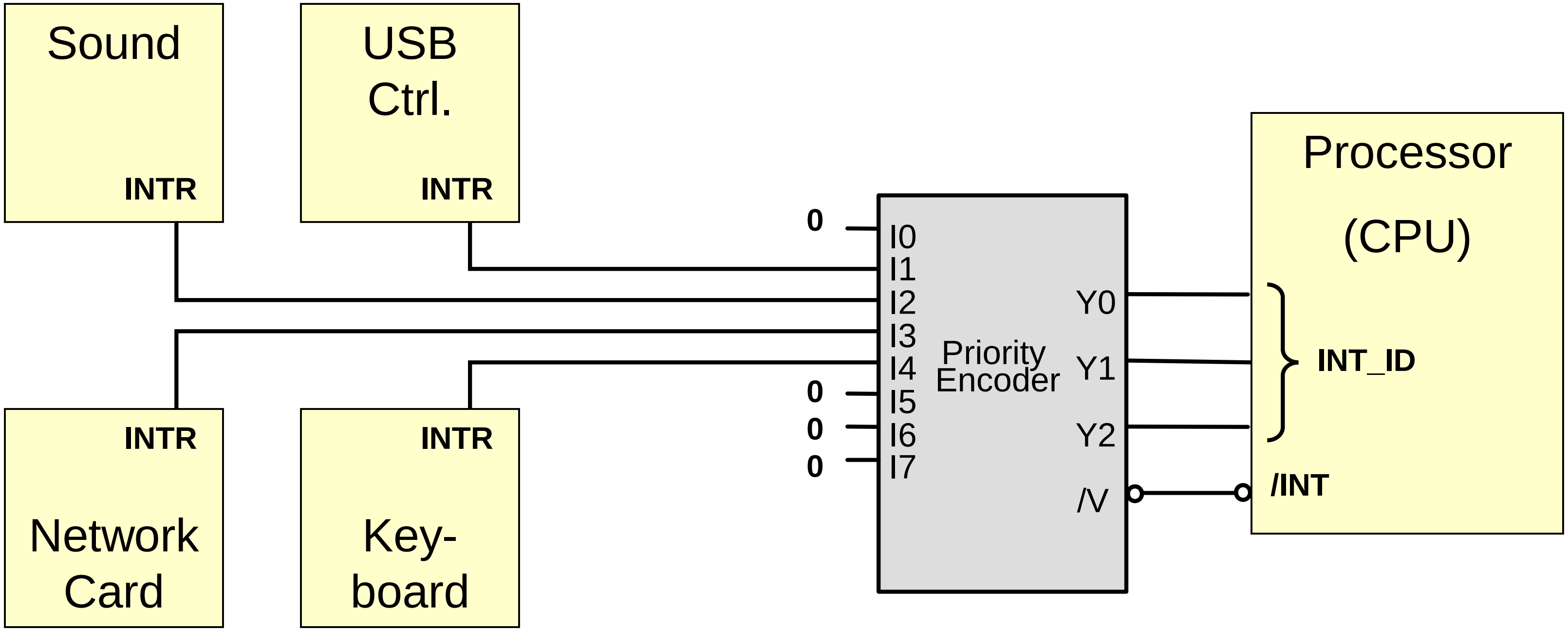
Encoder Application: Interrupts

- I/O Devices in a computer need to request attention from the CPU…they need to “interrupt” the processor
- CPU cannot have a dedicated line to each I/O device (too many inputs and outputs) plus it can only service one device at a time



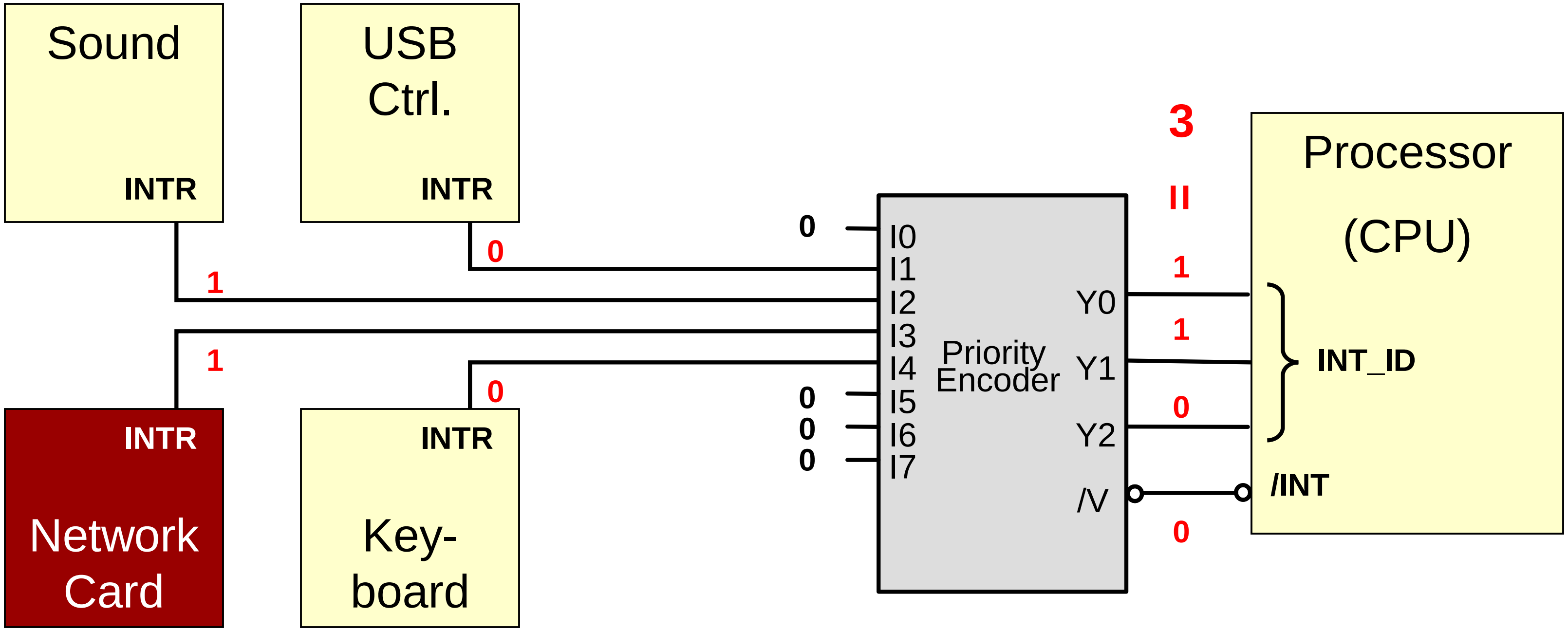
Encoder Application: Interrupts

- Solution: Priority Encoder
- /INT input of CPU indicates SOME device is requesting attention
- INT_ID inputs identify who is requesting attention



Encoder Application: Interrupts

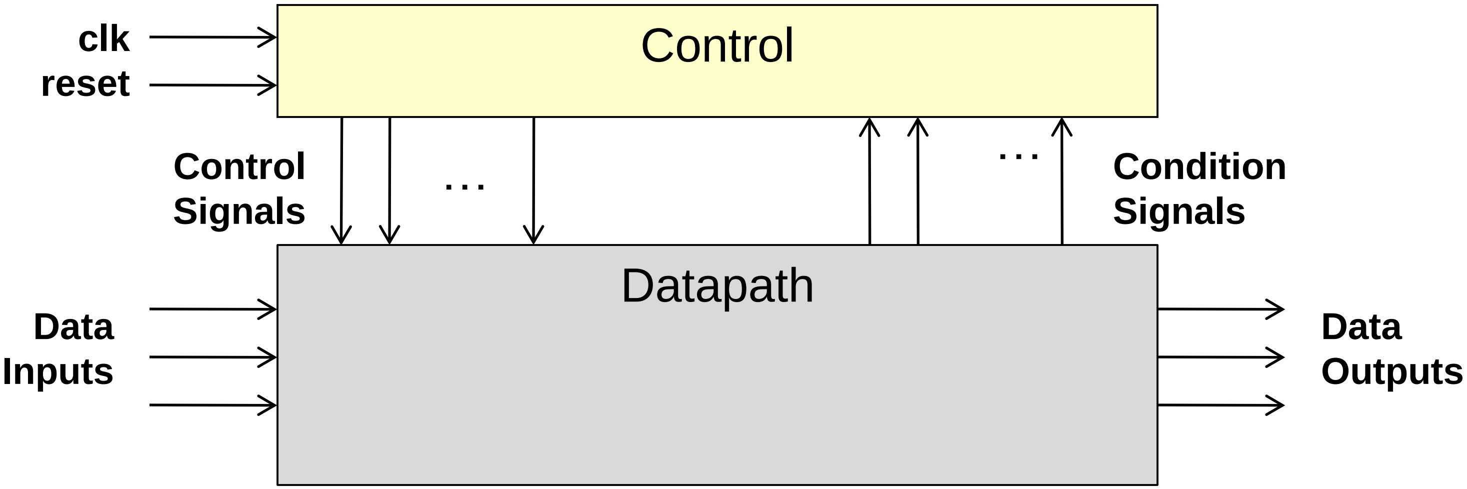
- Example: Sound and Network request interrupt at the same time
- Network is highest priority and is encoded
- After network is handled, sound will cause interrupt



VENDING MACHINE

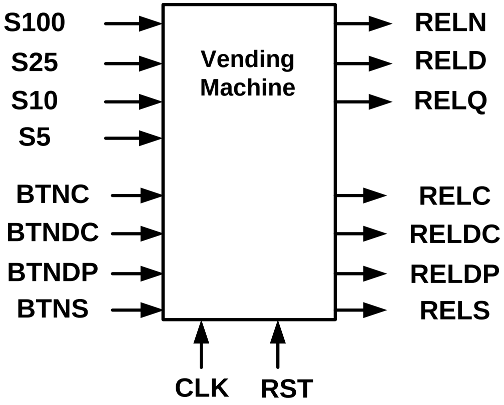
Digital System Design

- Control and Datapath Unit paradigm
  - Separate logic into datapath elements that operate on data and control elements that generate control signals for datapath elements
  - Datapath: Adders, muxes, comparators, counters, registers (w/ enables)
  - Control Unit: State machines/sequencers



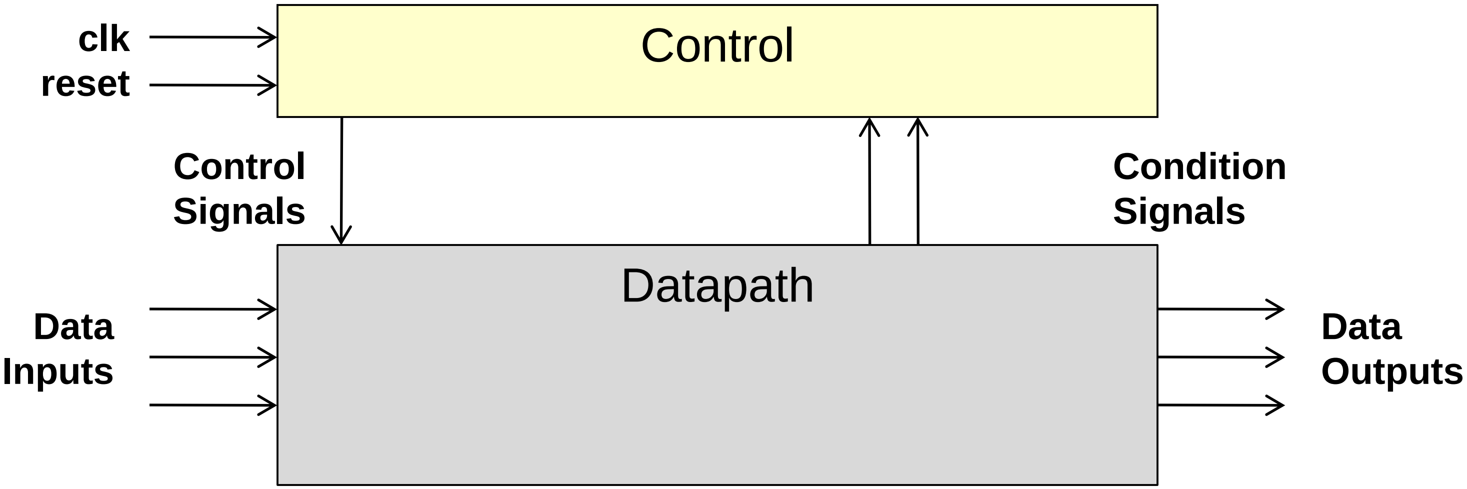
Vending Machine Controller

- Consider a vending machine that sells Coke, Diet Coke, Sprite and Dr. Pepper
  - Drinks cost $1
  - Sensors indicate (for 1 clock cycle) when a user has entered a nickel, dime, quarter, or dollar bill
  - Max. input amount is $2 (beyond that the machine is not responsible for counting)
  - Individual buttons for each drink allow the user to select their drink and if at least $1 has been entered, a release signal for each drink should be asserted
  - Making change will be considered in a future lab



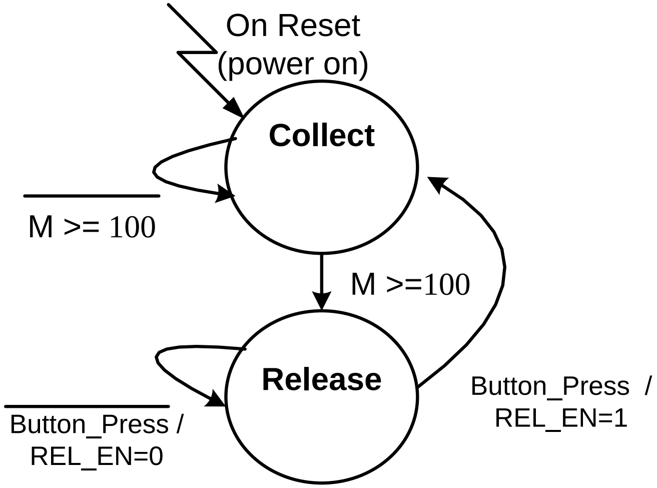
Digital System Design

- Control and Datapath Unit paradigm
  - Separate logic into datapath elements that operate on data and control elements that generate control signals for datapath elements
  - Datapath: Adders, muxes, comparators, counters, registers (w/ enables)
  - Control Unit: State machines/sequencers



Money Collection & Release FSM

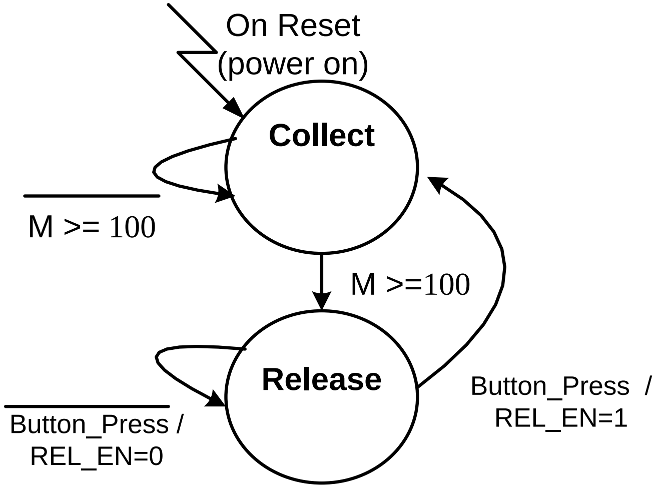
- Consider the state machine only for money collection and release signal generation



Pseudocode for collection algorithm:

Money Collection & Release FSM

- Consider the state machine only for money collection and release signal generation

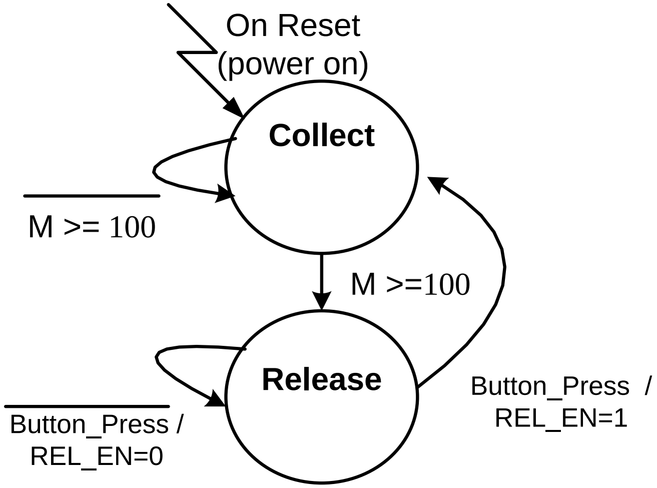


Pseudocode for collection algorithm:

```
Let M = 0
while( M < 200 )
  if S5 == 1 then M = M + 5
  if S10 == 1 then M = M + 10
  if S25 == 1 then M = M + 25
  if S100 == 1 then M = M + 100
```

Money Collection & Release FSM

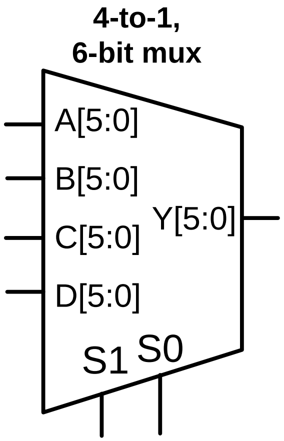
- Consider the state machine only for money collection and release signal generation

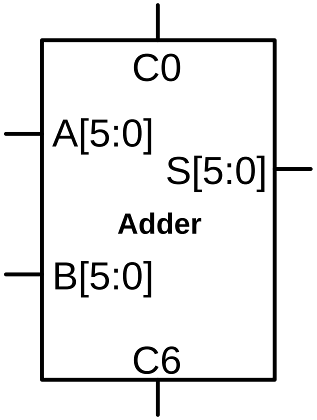

On Reset
(power on)
Collect
$\overline{\text{M >= 100}}$
M >=100
Release
$\overline{\text{Button\_Press}}$ / REL_EN=0
Button_Press / REL_EN=1

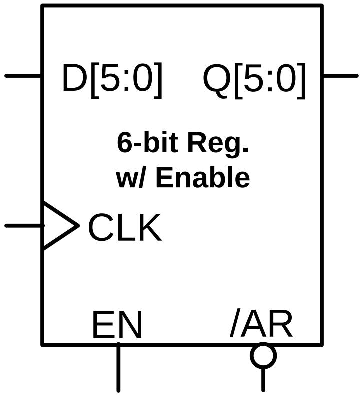
Pseudocode for collection algorithm:

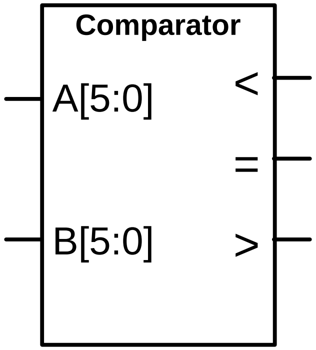
```
Let M = 0
while( state == COLLECT )
  if S5 == 1 then M = M + 1
  if S10 == 1 then M = M + 2
  if S25 == 1 then M = M + 5
  if S100 == 1 then M = M + 20
```

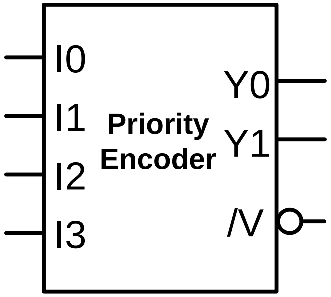
Money Collection Datapath





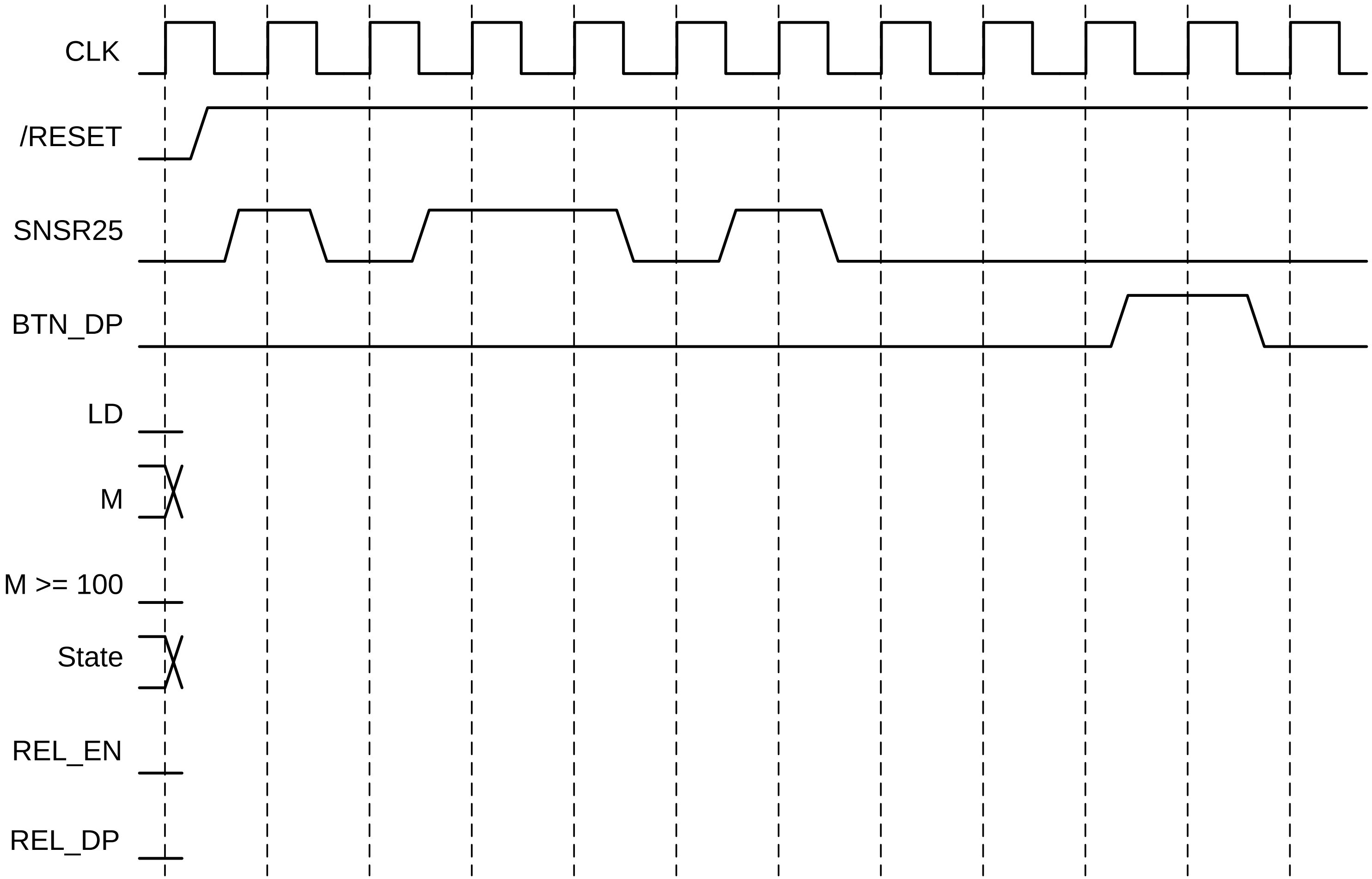




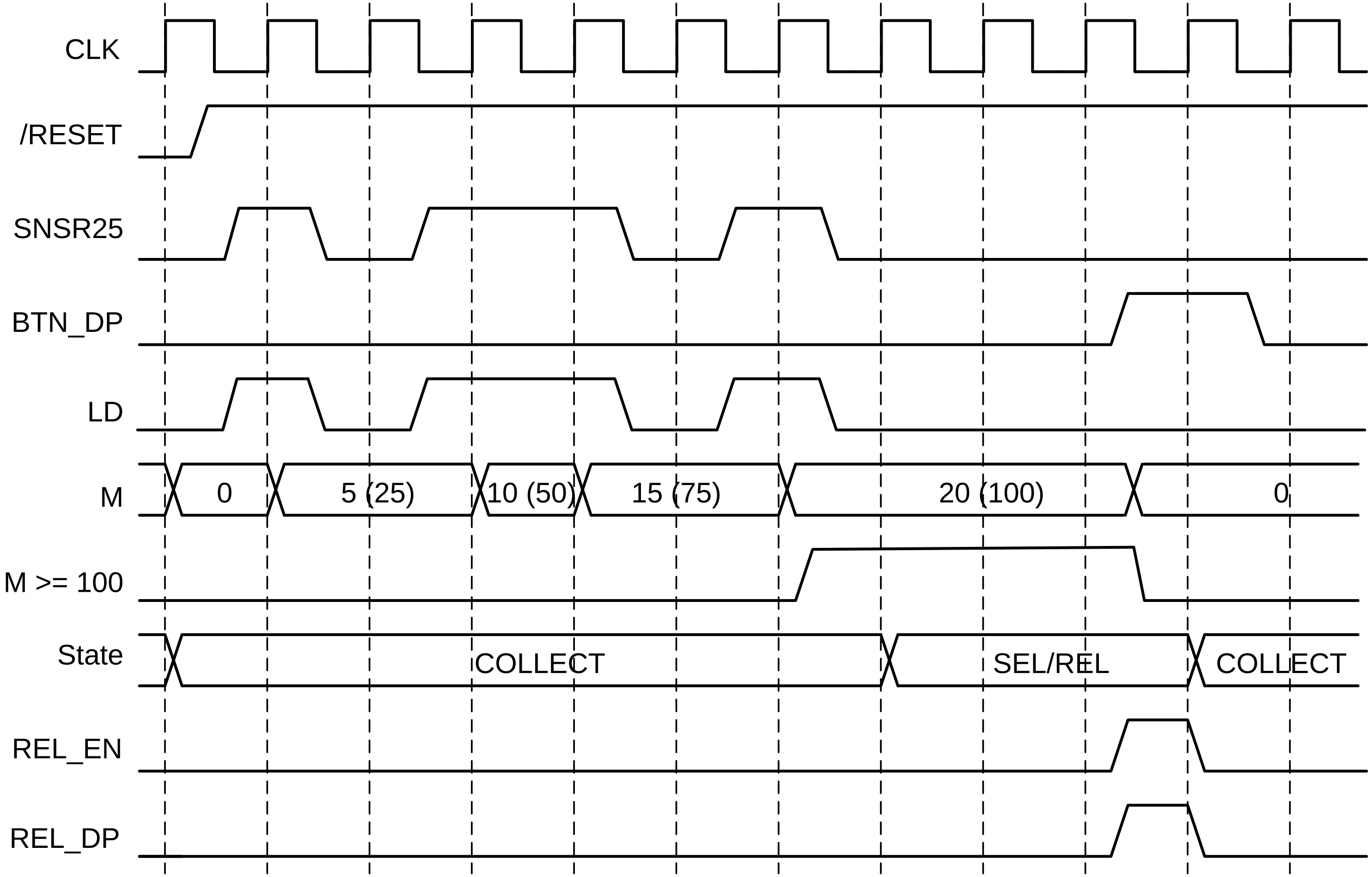


Money Release Datapath

Sample Operation Waveform



Sample Operation Solution


CLK
/RESET
SNSR25
BTN_DP
LD
M
0
5 (25)
10 (50)
15 (75)
20 (100)
0
M >= 100
State
COLLECT
SEL/REL
COLLECT
REL_EN
REL_DP