# Spiral 2-1

Datapath Components:
Counters
Adders
Design Example: Crosswalk Controller

## Spiral Content Mapping

| Spiral | Theory | Combinational Design | Sequential Design | System Level Design | Implementation and Tools | Project |
|---|---|---|---|---|---|---|
| 1 | • Performance metrics (latency vs. throughput)<br>• Boolean Algebra<br>• Canonical Representations | • Decoders and muxes<br>• Synthesis with min/maxterms<br>• Synthesis with Karnaugh Maps | • Edge-triggered flip-flops<br>• Registers (with enables) | • Encoded State machine design | • Structural Verilog HDL<br>• CMOS gate implementation<br>• Fabrication process | |
| 2 | • Shannon's Theorem | • Synthesis with muxes & memory<br>• Adder and comparator design | • Bistables, latches, and Flip-flops<br>• Counters<br>• Memories | • One-hot state machine design<br>• Control and datapath decomposition | • MOS Theory<br>• Capacitance, delay and sizing<br>• Memory constructs | |
| 3 | | | | • HW/SW partitioning<br>• Bus interfacing<br>• Single-cycle CPU | • Power and other logic families<br>• EDA design process | |

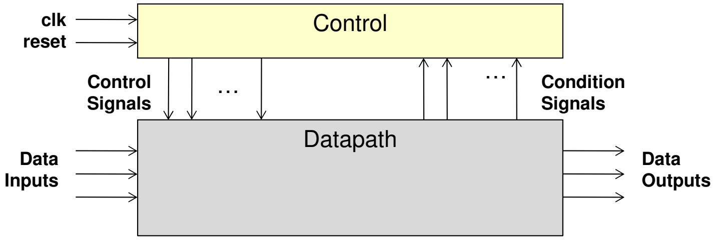
## Learning Outcomes

- I understand the control inputs to counters
- I can design logic to control the inputs of counters to create a desired count sequence
- I understand how smaller adder blocks can be combined to form larger ones
- I can build larger arithmetic circuits from smaller building blocks
- I understand the timing and control input differences between asynchronous and synchronous memories

# DATAPATH COMPONENTS

# Digital System Design

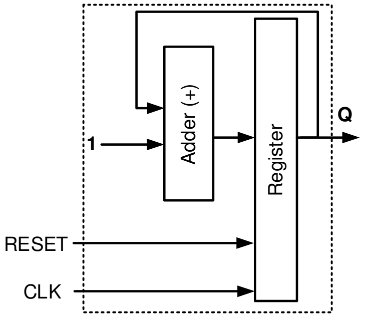
- ___________ **(CU)** and ___________ Unit **(DPU)** paradigm
  - Separate logic into datapath elements that operate on data and control elements that generate control signals for datapath elements
  - Datapath: ___________, _________, comparators, ________, registers (shift, with enables, etc.), memories, FIFO's
  - Control Unit: _______________/sequencers



# COUNTERS

# Counters

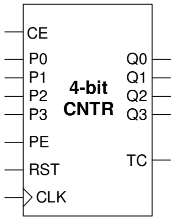
- Count (Add 1 to Q) at each clock edge
  - Up Counter: ____________
  - Can also build a down counter as well (_____________)
- Standard counter components include other features
  - Resets: Reset count to 0
  - Enables: Will not count at edge if EN=0
  - ______________ Inputs: Can initialize count to a value P (i.e. Q* = P rather than Q+1)



How would you design the adder block above for a 4-bit counter?

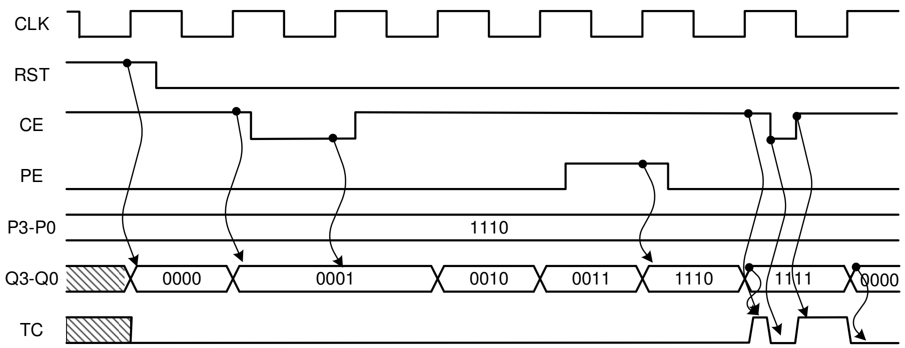
# Sample 4-bit Counter

- 4-bit Up Counter
  - RST: a synchronous reset input
  - PE and $P_i$ inputs: loads Q with P when PE is active
  - CE: Count Enable
    - Must be active for the counter to count up
  - TC (Terminal Count) output
    - Active when Q=1111 AND counter is enabled
    - TC = ________________
      - _________ output
    - Indicates that on the next edge it will roll over to 0000



| CLK | RST | PE | CE | Q* |
|---|---|---|---|---|
| 0,1 | | | | |
| ↑ | | | | |
| ↑ | | | | |
| ↑ | | | | |
| ↑ | | | | |

# Counters



SR=active at clock edge, thus Q=0

Q*=Q+1

Enable = off, thus Q holds

Q*=Q+1

Q*=Q+1

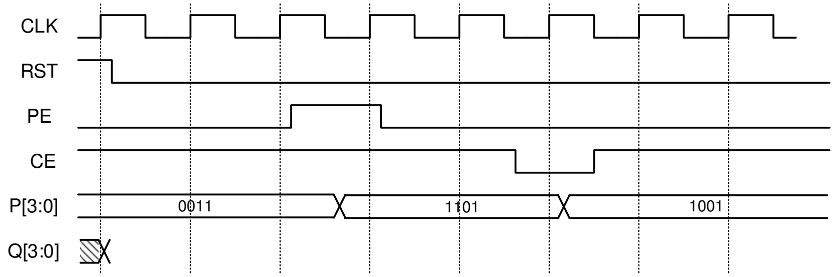
PE = active, thus Q=P

Q*=Q+1

Q*=Q+1

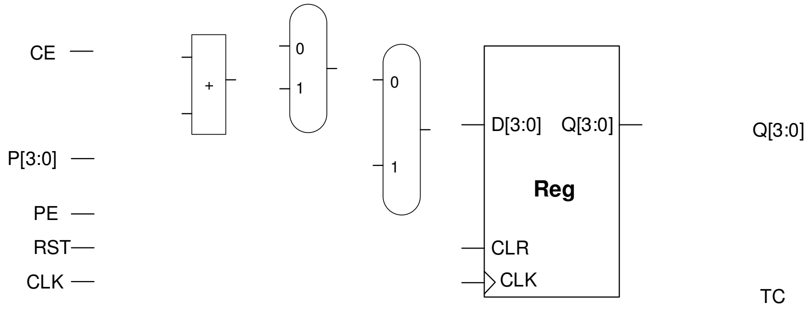
Mealy TC output: EN•Q3•Q2•Q1•Q0

# Counter Exercise

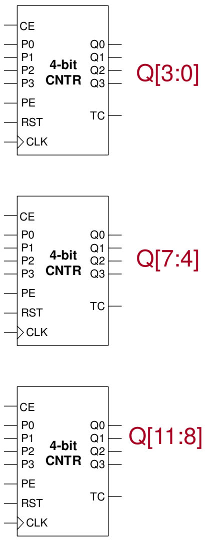


# Counter Design

- Sketch the design of the 4-bit counter presented on the previous slides



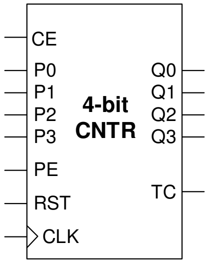
# Design a 12-bit Counter (Why TC?)



| Q9 | Q8 | Q7 | Q6 | Q5 | Q4 | Q3 | Q2 | Q1 | Q0 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| | | | | … | | | | | |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | | | | … | | | | | |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| | | | | … | | | | | |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |
| | | | | … | | | | | |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Counter Example (Using Parallel Inputs)

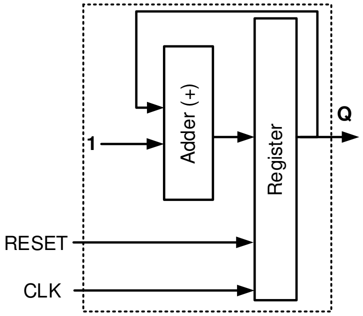
- Design a circuit that counts each clock cycle to produce the pattern 5, 6, 7, 8, 9, 5, 6, 7, 8, 9, 5…9, 5…9,…



# ADDERS

## Adder Intro

- So how would we build a circuit to add two numbers?
- Let's try to design a circuit that can add **ANY** two 4-bit numbers, X[3:0] and Y[3:0]
  - How many inputs?
  - Can we use K-Maps or sum of minterms, etc?



```
  0110 = X
+ 0111 = Y
  ----
  1101
```

## Adder Intro

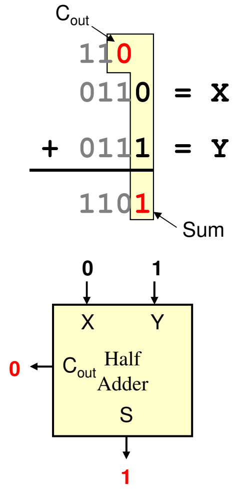
- **Idea**: Build a circuit that performs _____ column of addition and then use ________________ of those circuits to perform the overall 4-bit addition
- Let's start by designing a circuit that adds 2-bits: X and Y that are in the same column of addition

```
  0|1|1|0 = X
+ 0|1|1|1 = Y
  -------
  1|1|0|1
```

# Addition – Half Adders

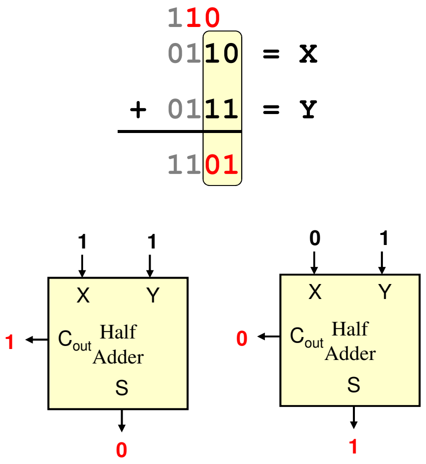
- Addition is done in columns
  - Inputs are the bit of X, Y
  - Outputs are the Sum Bit and Carry-Out ($C_{out}$)
- Design a Half-Adder (HA) circuit that takes in X and Y and outputs S and $C_{out}$

| X | Y | $C_{out}$ | S |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |



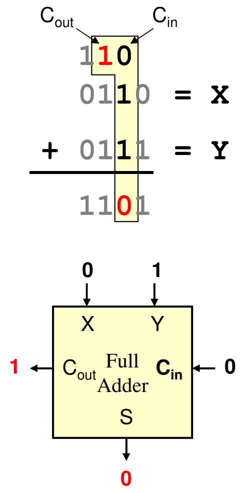
# Addition – Half Adders

- We'd like to use one adder circuit for each column of addition
- Problem:
  - No place for __________ of last adder circuit
- Solution
  - Redesign adder circuit to include an ___________ _____________



# Addition – Full Adders

- Add a Carry-In input($C_{in}$)
- New circuit is called a Full Adder (FA)
- Design the internal circuitry on the next slide



# Addition – Full Adders

- Find the minimal 2-level implementations for Cout and S…

| X | Y | $C_{in}$ | $C_{out}$ | S |
|---|---|---|---|---|
| 0 | 0 | 0 | | |
| 0 | 0 | 1 | | |
| 0 | 1 | 0 | | |
| 0 | 1 | 1 | | |
| 1 | 0 | 0 | | |
| 1 | 0 | 1 | | |
| 1 | 1 | 0 | | |
| 1 | 1 | 1 | | |

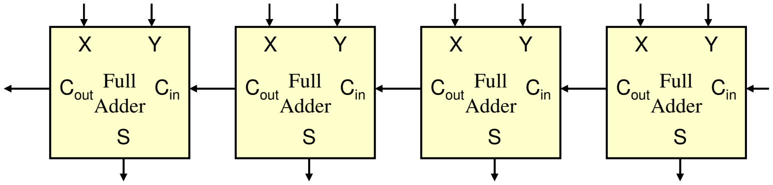
# Full Adder Logic

- S = ____________________
  - Recall: _____ is defined as true when ODD number of inputs are true…exactly when the sum bit should be 1
- Cout = _____________________
  - Carry when sum is 2 or more (i.e. when at least 2 inputs are 1)
  - Circuit is just checking all combinations of 2 inputs

# Addition – Full Adders

- Use 1 Full Adder for each column of addition

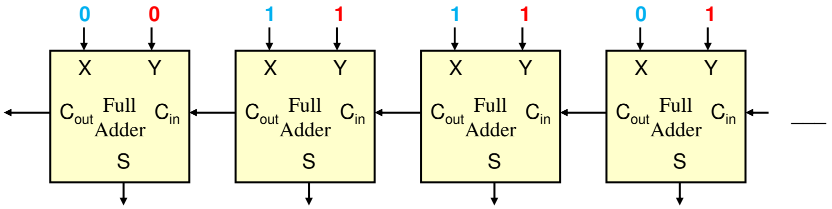
```
   0110
+  0111
```



# Addition – Full Adders

- Connect bits of bottom number to Y inputs

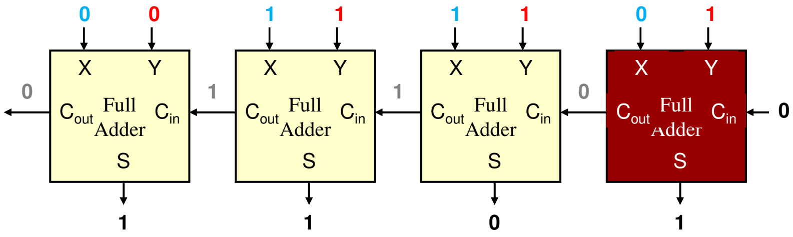
```
   0110  = X
+  0111  = Y
```



# Addition – Full Adders

- Use 1 Full Adder for each column of addition

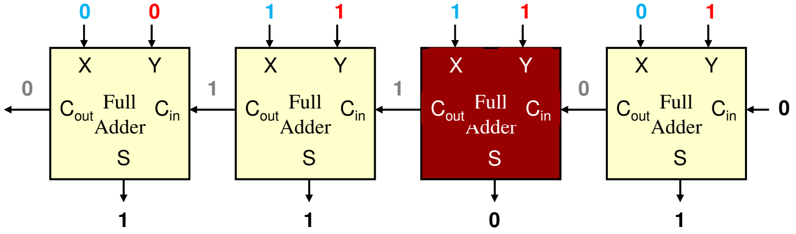
```
  01100
   0110  = X
+  0111  = Y
   1101
```



# Addition – Full Adders

- Use 1 Full Adder for each column of addition

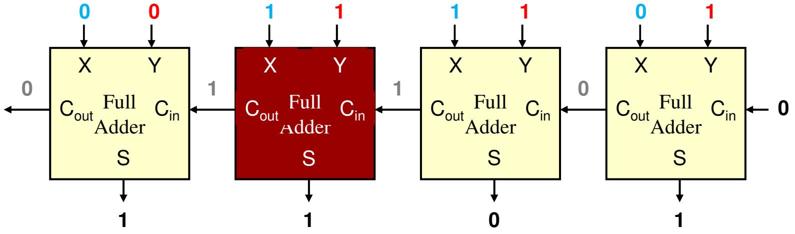
```
  01100
   0110 = X
 + 0111 = Y
   1101
```



# Addition – Full Adders

- Use 1 Full Adder for each column of addition

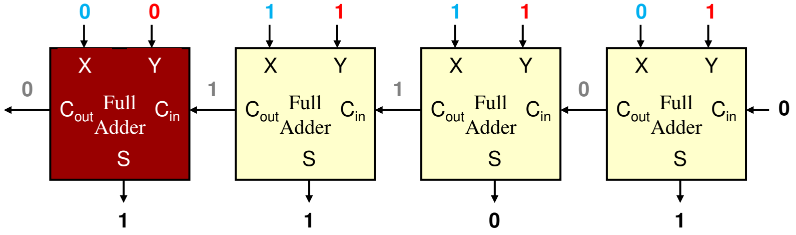
```
  01100
   0110 = X
 + 0111 = Y
   1101
```



# Addition – Full Adders

- Use 1 Full Adder for each column of addition

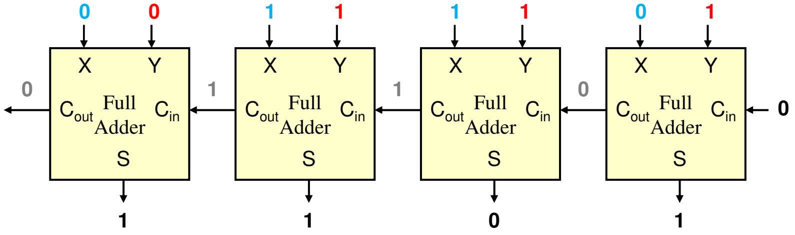
```
  01100
   0110 = X
 + 0111 = Y
   1101
```



# Addition – Full Adders

- Use 1 Full Adder for each column of addition

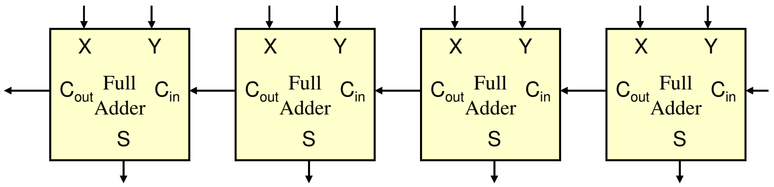
```
  01100
   0110 = X
 + 0111 = Y
   1101
```



# Performing Subtraction w/ Adders

- To subtract
  - Flip bits of Y
  - Add 1

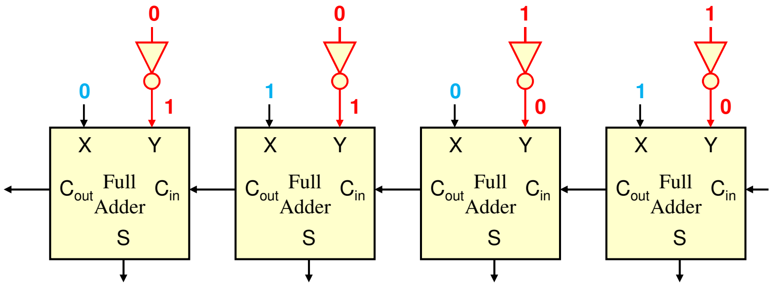
$$\begin{array}{rrl} & 0101 & = X \\ - & 0011 & = Y \\ \hline & 0010 & \end{array} \quad \Rightarrow \quad \begin{array}{rr} & 0101 \\ + & 1100 \\ & 1 \\ \hline & 0010 \end{array}$$



# Performing Subtraction w/ Adders

- To subtract
  - Flip bits of Y
  - Add 1

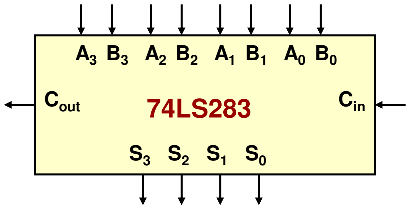
$$\begin{array}{rrl} & 0101 & = X \\ - & 0011 & = Y \\ \hline & 0010 & \end{array} \quad \Rightarrow \quad \begin{array}{rr} & 0101 \\ + & 1100 \\ & 1 \\ \hline & 0010 \end{array}$$



# 4-bit Adders

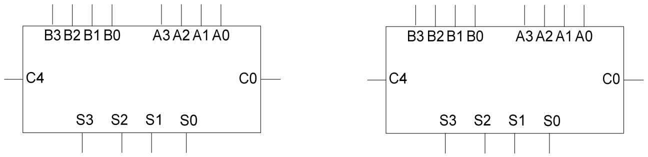
- 74LS283 chip implements a 4-bit adder

$$\begin{array}{rrl} & A_3A_2A_1A_0 & = A \\ + & B_3B_2B_1B_0 & = B \\ \hline & S_4S_3S_2S_1S_0 & = S \end{array}$$



# Building an 8-bit Adder

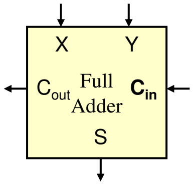
- Use (2) 4-bit adders to build an 8-bit adder to add X=X[7:0] and Y= Y[7:0] and produce a sum, S=[7:0] and a carry-out, C8.
  - Make sure you understand the difference between system labels (actual signal names from the top level design) and device labels (placeholder names for the signals inside each block).



# EXERCISES

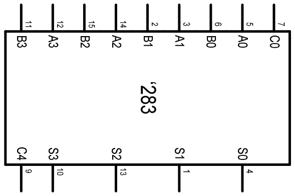
# Adding Many Bits

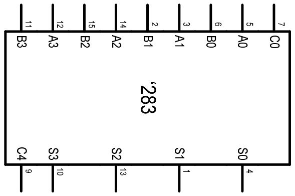
- You know that an FA adds X + Y + Ci
- Use FA and/or HA components to add 4 individual bits:
  A + B + C + D



# Adding 3 Numbers

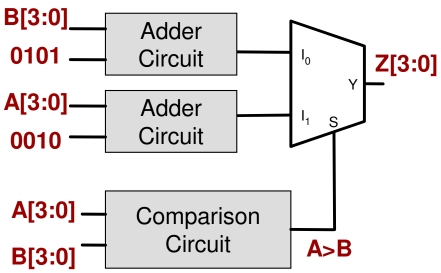
- Add X[3:0] + Y[3:0] + Z[3:0] to produce F[?:0] using the adders shown plus any FA and HA components you need





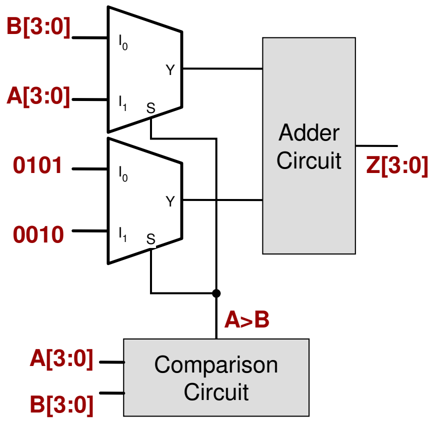
# Mapping Algorithms to HW

- Wherever an if..then..else statement is used usually requires a mux
  - if(A[3:0] > B[3:0])
    - Z = A+2
  - else
    - Z = B+5



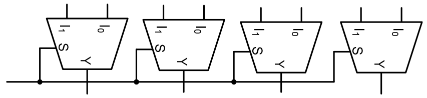
# Mapping Algorithms to HW

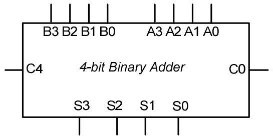
- Wherever an if..then..else statement is used usually requires a mux
  - if(A[3:0] > B[3:0])
    - Z = A+2
  - else
    - Z = B+5



# Adder / Subtractor

- If sub/~add = 1
  - Z = X[3:0]-Y[3:0]
- Else
  - Z = X[3:0]+Y[3:0]

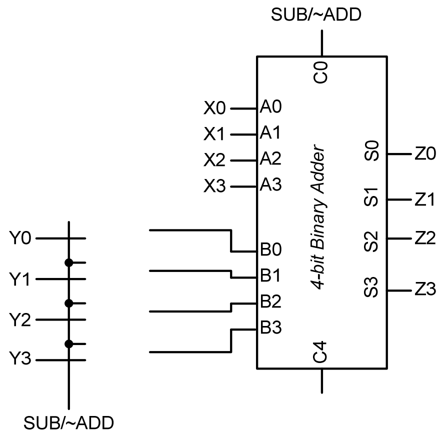




# Adder / Subtractor

- If sub/~add = 1
  - Z = X[3:0]-Y[3:0]
- Else
  - Z = X[3:0]+Y[3:0]

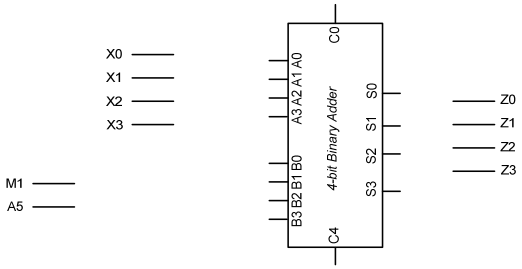
| SUB/ ~ADD | Yi | Bi |
|---|---|---|
| 0 | 0 | |
| 0 | 1 | |
| 1 | 0 | |
| 1 | 1 | |



# Another Example

- Design a circuit that takes a 4-bit binary number, X, and two control signals, A5 and M1 and produces a 4-bit result, Z, such that:
- Z = X + 5, when A5,M1 = 1,0
- Z = X – 1, when A5,M1 = 0,1
- Z = X, when A5,M1 = 0,0

| | | 4-bit Adder Input | | | |
|---|---|---|---|---|---|
| A5 | M1 | B3 | B2 | B1 | B0 |
| 0 | 0 | | | | |
| 0 | 1 | | | | |
| 1 | 0 | | | | |
| 1 | 1 | d | d | d | d |



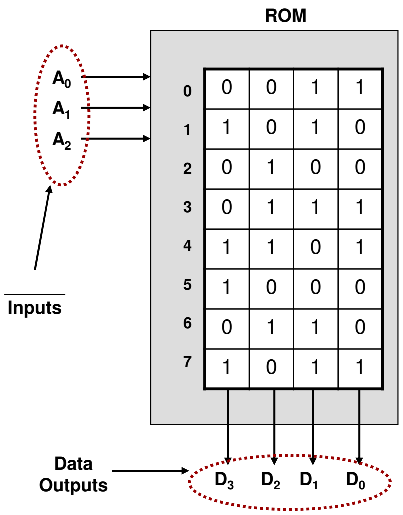
# ROMS AND MEMORIES

# Memories

- Memories store (write) and retrieve (read) data
  - Read-Only Memories (ROM's): Can only retrieve data (contents are initialized and then cannot be changed)
  - Read-Write Memories (RWM's): Can retrieve data and change the contents to store new data

# ROM's

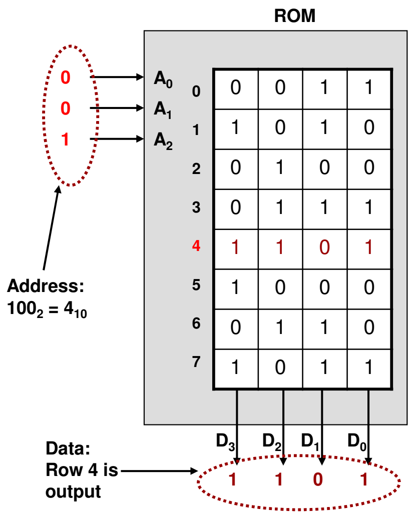
- Memories are just tables of data with rows and columns
- When data is _______, one entire ______ of data is read out
- The row to be read is selected by putting a binary number on the _________ inputs



| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 1 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 |
| 7 | 1 | 0 | 1 | 1 |

# ROM's

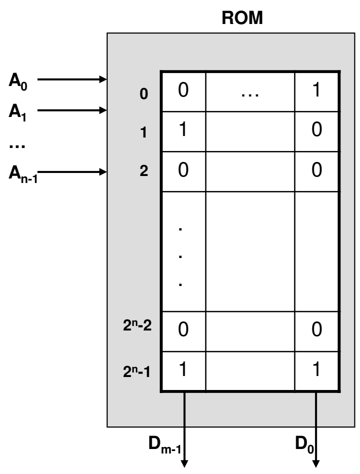
- Example
  - Address = 4 dec. = 100 bin. is provided as input
  - ROM outputs data in that row (1101 bin.)



| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 1 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 |
| 7 | 1 | 0 | 1 | 1 |

# Memory Dimensions

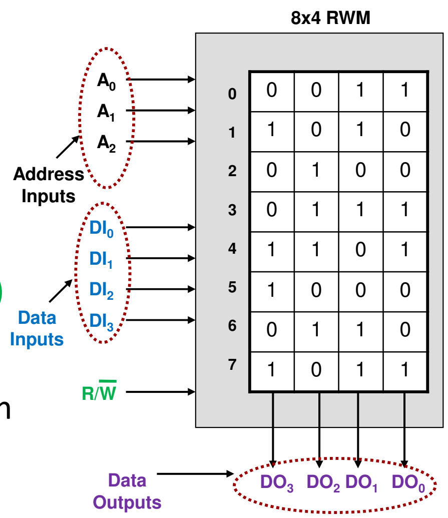
- Memories are named by their dimensions:
  - ________ x _________
- *n* rows and *m* columns => n x m ROM
- $2^n$ rows => n address bits (or k rows => $\log_2 k$ address bits)
- m cols. => m data outputs



|  |  |  |  |
|---|---|---|---|
| 0 | 0 | … | 1 |
| 1 | 1 |  | 0 |
| 2 | 0 |  | 0 |
|  | . . . |  |  |
| $2^n$-2 | 0 |  | 0 |
| $2^n$-1 | 1 |  | 1 |

# RWM's

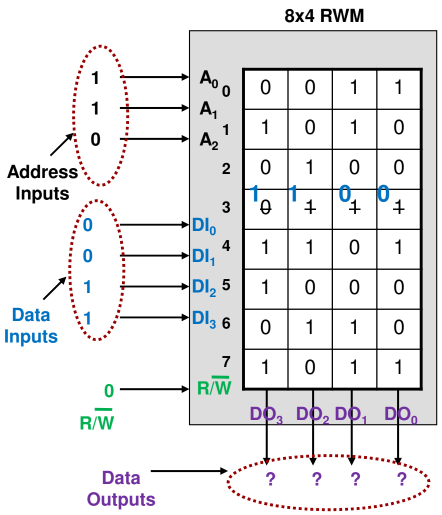
- Writable memories provide a set of data inputs for write data (as opposed to the data outputs for read data)
- A control signal $R/\overline{W}$ (1=_______ / 0 = ________) is provided to tell the memory what operation the user wants to perform



|  |  |  |  |  |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 1 |
| 4 | 1 | 1 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 |
| 7 | 1 | 0 | 1 | 1 |

# RWM's

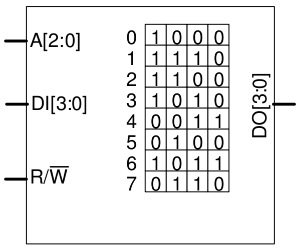
- Write example
  - Address = 3 dec. = 011 bin.
  - DI = 12 dec. = 1100 bin.
  - $R/\overline{W}$ = 0 => Write op.
- Data in row 3 is overwritten with the new value of 1100 bin.



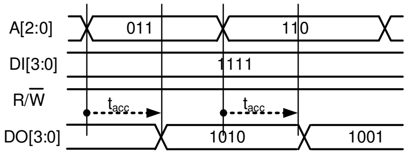
|  |  |  |  |  |
|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 1 ~~0~~ | 1 ~~1~~ | 0 ~~1~~ | 0 ~~1~~ |
| 4 | 1 | 1 | 0 | 1 |
| 5 | 1 | 0 | 0 | 0 |
| 6 | 0 | 1 | 1 | 0 |
| 7 | 1 | 0 | 1 | 1 |

# Asynchronous Memories

- Notice that there is ______________ signal with this memory
- Devices that do not use a clock signal are called "______________" devices
- For these memories, the address must be kept ________ and stable for at least $t_{acc}$ amount of time

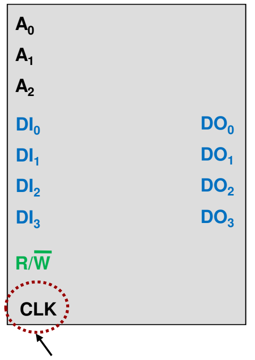


|  |  |  |  |  |
|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |
| 2 | 1 | 1 | 0 | 0 |
| 3 | 1 | 0 | 1 | 0 |
| 4 | 0 | 0 | 1 | 1 |
| 5 | 0 | 1 | 0 | 0 |
| 6 | 1 | 0 | 1 | 1 |
| 7 | 0 | 1 | 1 | 0 |



# Asynchronous vs. Synchronous Memories

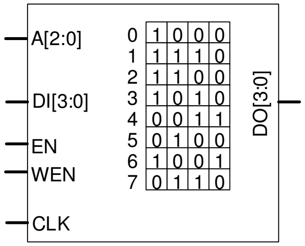
- Asynchronous memories use no CLK signal
  - For read: Address and R/W signal must be **held steady for a certain period of time** before DO outputs become valid
  - For write: Address, DI, and R/W signal must be **held steady for a certain period of time** before internal memory is updated
- Synchronous memories use a CLK signal
  - For read: Address and R/W signal will be **registered on the CLK edge** and then DO will become valid during that subsequence clock cycle
  - For write: Address, DI and R/W signals will be **registered on the CLK edge** and then the internal memory updated during the subsequent clock cycle

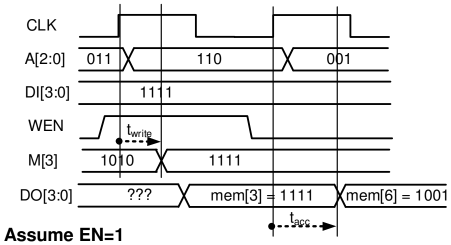


**Synchronous memories add a clock signal and the input values at a clock edge will only be processed during the subsequent clock cycle**

# Synchronous Timing

- For **synchronous** memories the address must be valid and stable at ________________ but then may be changed
- EN = _______ enable (unless it is 1) the memory won't read or write
- WEN = _______________
  - 1 = Write / 0 = read

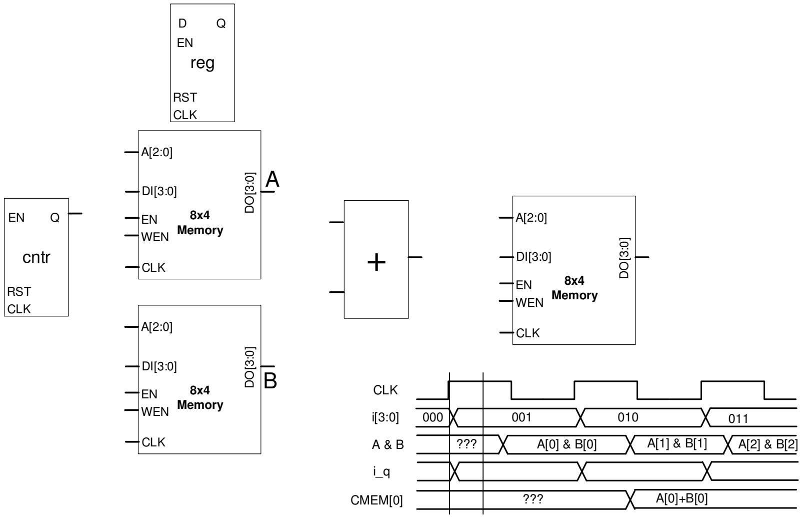




**Assume EN=1**

# Using Memories

- Add two 8 number arrays (C[i] = A[i] + B[i])

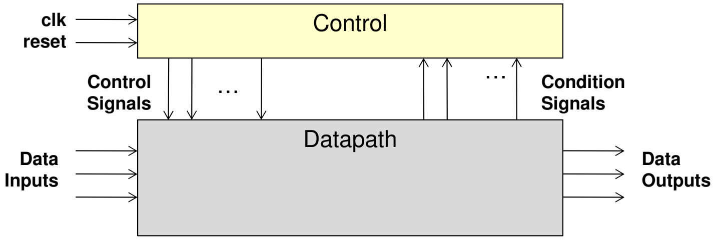


Crosswalk Controller

# SYSTEM DESIGN EXAMPLE

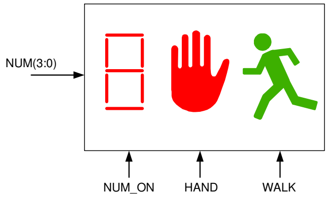
# Digital System Design

- 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



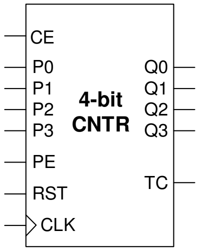
# Crosswalk Controller

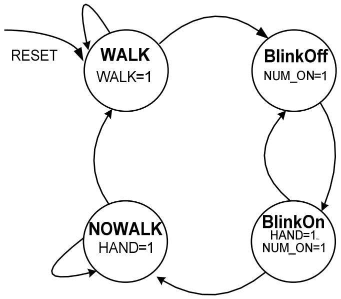
- Design a crosswalk controller to adhere to the following description
- 8 ticks of the clock in the WALK phase
- 8 ON/OFF BLINKING hand cycles (16 total ticks)
- Count 8 downto 1 on the NUM display while hand is blinking
- 16 cycles in the SOLID hand



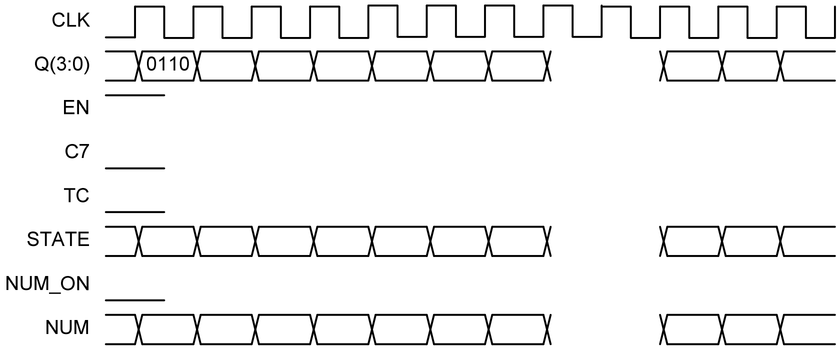
# Crosswalk State Machine

- Use a 4-bit counter to count cycles along with an additional gate or two…





# Crosswalk Controller Operation



# Summary

- You should now be able to build:
  - Registers (w/ Enables)
  - Counters
  - Adders