
Computer Aided Digital Systems Design - EE 4743/6743

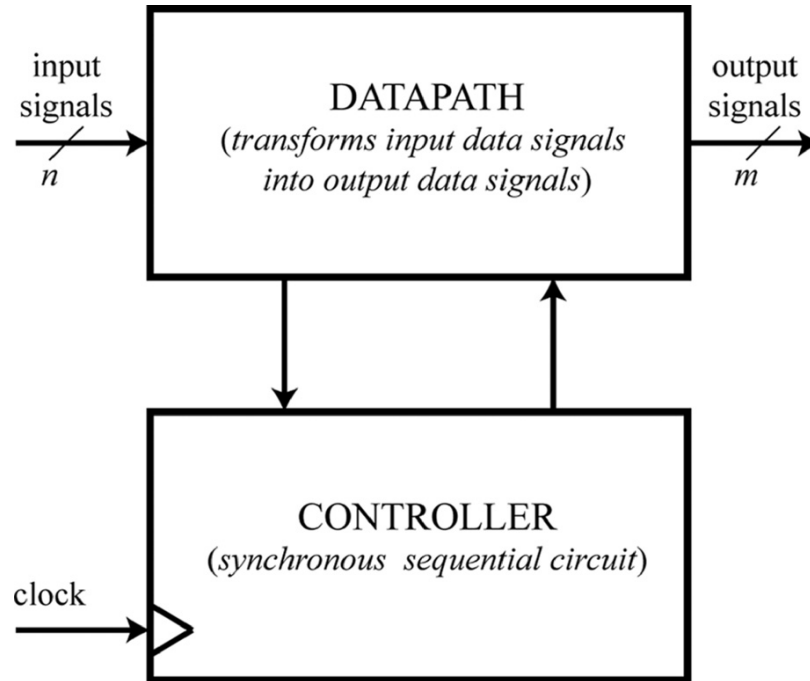
Sherif Abdelwahed

Datapath Design

**Department of Electrical and Compute Engineering
Mississippi State University**

Composition of Digital Systems

- Most digital systems can be partitioned into two types of modules:
 - **Datapath:** perform data-processing operations between registers
 - **Controller:** determine the sequence of those operations



Datapaths

- The datapath is the portion of the circuit that contains components that transform input data signals into output data signals.
 - Datapaths may be purely combinational or they may also contain synchronous components such as dedicated counters.
 - A typical datapath often consists of several components:
 - Registers
 - Memory units (LUT)
 - Arithmetic/logic units (ALU)
 - Comparators
 - Multiplexers
 - Tristate buses
-

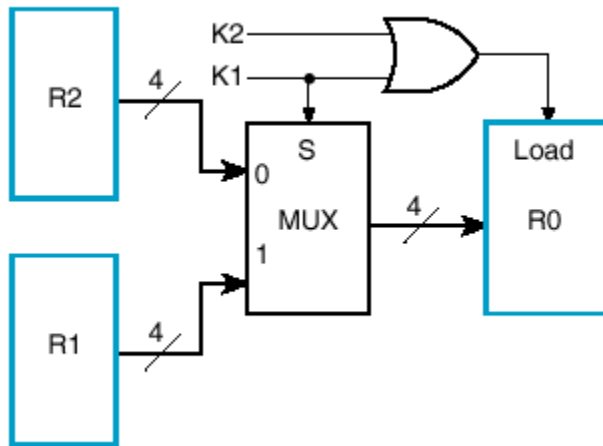
Controller

- Controller can be programmable or non-programmable
 - Programmable
 - Has a program counter which point to next instruction
 - Instructions are held in a RAM or ROM externally
 - Microprocessor is an example of programmable controller
 - Non-Programmable
 - Once designed, implements the same functionality
 - Another term is a “hardwired state machine” or “hardwired instructions”
 - **We will be focusing primarily on the non-programmable type in this course**
-

Data Transfer

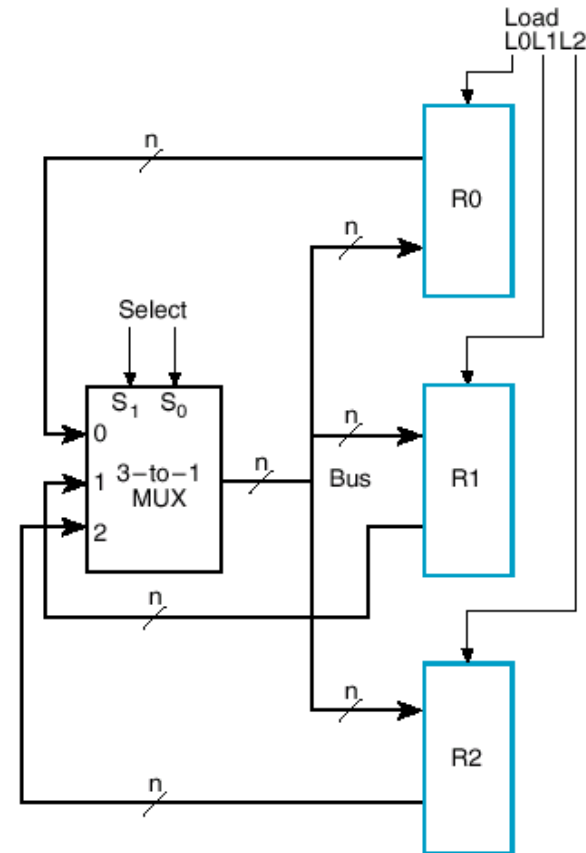
Multiplexer-based Transfer

- Used when a register receives data from two or more different sources at different times



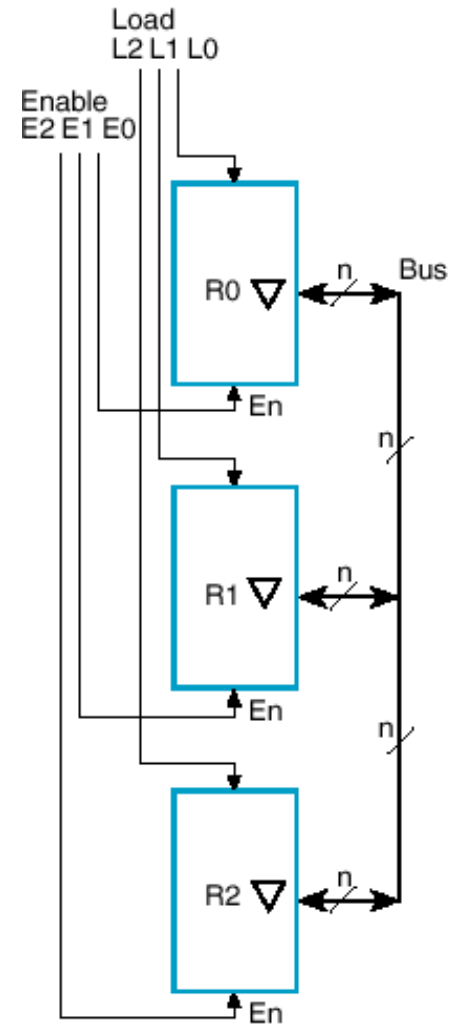
Bus-based Transfer

- Can have only one source but multiple destinations at a time



Tri-State Buses

- A bus can be constructed using three-state buffers
- Several three-state buffer outputs can be connected together
 - Avoid the high fan-in OR in multiplexers
 - Delay time and logic complexity can be reduced
- The signals can travel in two directions on a three-state bus
 - EN=1: output, EN=0: input
 - Simplify the interconnections



Modeling Three-State Registers

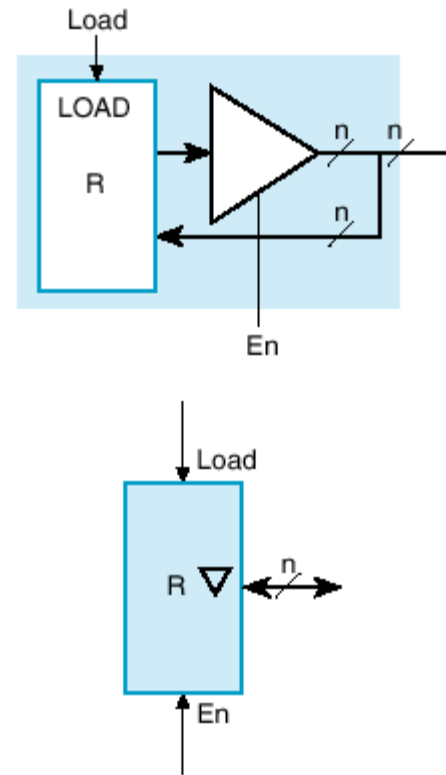
- To declare a bidirectional data port, `inout` type is used instead of input or output

```
module TriReg(CLK, Rst, EN, Load, Data);
  input CLK, Rst, EN, Load;
  inout Data;

  reg int_data, Data;

  always @(posedge CLK) begin
    if (Rst) int_data = 0;
    else if (Load) int_data = Data;
  end

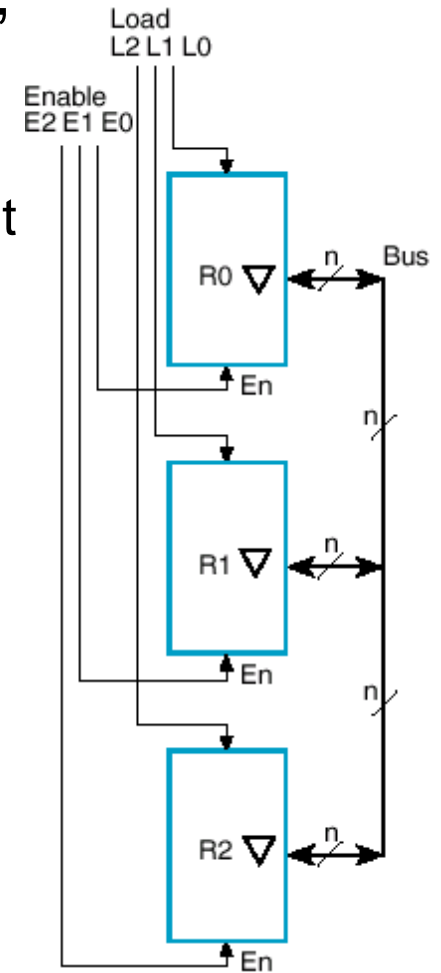
  always @(int_data or EN) begin
    if (EN) Data = int_data;
    else Data = 1`bz;
  end
endmodule
```



Modeling Tristate Bus

- To model the behavior of a three-state bus, `tri` type is used instead of `wire`
 - `tri`: has the same properties of `wire` but indicates more than one drivers may connect to it

```
module TriBus(CLK, Rst, E2, E1, E0, L2, L1, L0);  
  input CLK, Rst, E2, E1, E0, L2, L1, L0;  
  tri databus;  
  
  TriReg R0(CLK, Rst, E0, L0, databus);  
  TriReg R1(CLK, Rst, E1, L1, databus);  
  TriReg R2(CLK, Rst, E2, L2, databus);  
  
Endmodule
```

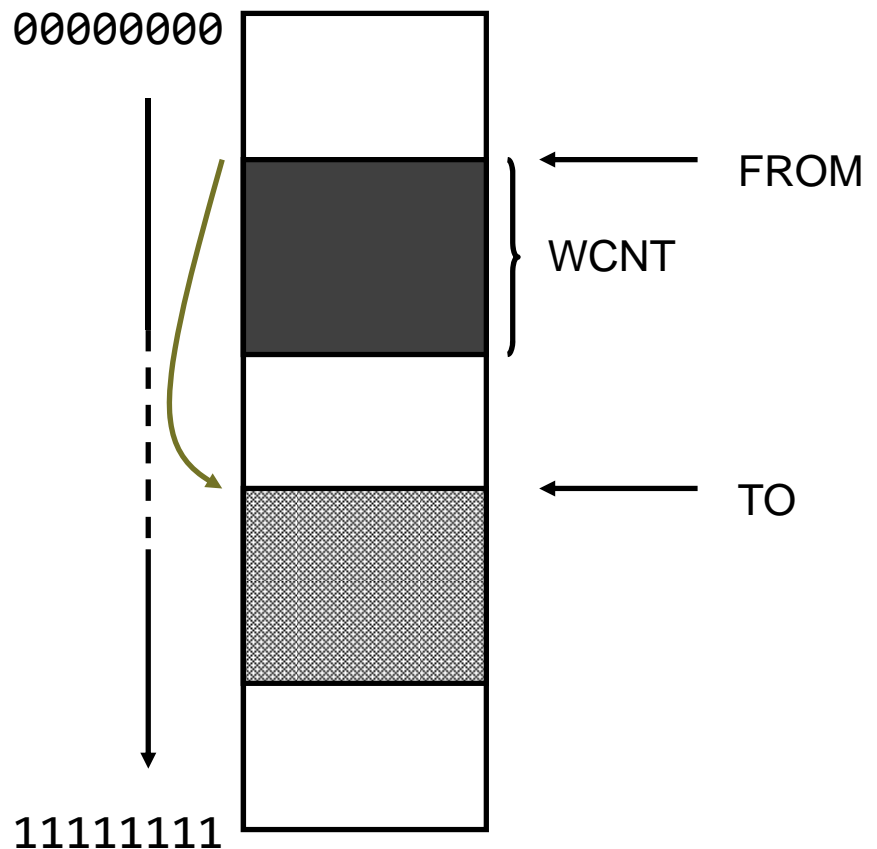


Design Process

- Layout in plain English what you want to accomplish
 - Design your datapath
 - Design your FSM controller (ASM chart)
 - Implement your datapath
 - Implement your FSM
 - Debug
 - Revise
-

Design Case Study

- Create a synchronous RAM block that has a block transfer capability
- Preserve normal RAM operation
- When 'xfer' input asserted,
 - assert BUSY output
 - transfer "WCNT" # of words
 - Copy from address "FROM"
 - Write into address "TO"
- Values of WCNT, FROM, TO are loaded before transfer operation started
- RAM size: 64 x 8



Normal RAM Interface

- Inputs

- clk
- we - write enable for RAM
- DIN[7..0] Data bus to RAM
- Addr[5..0] Address bus to RAM.

- Outputs

- DOUT[7..0]
-

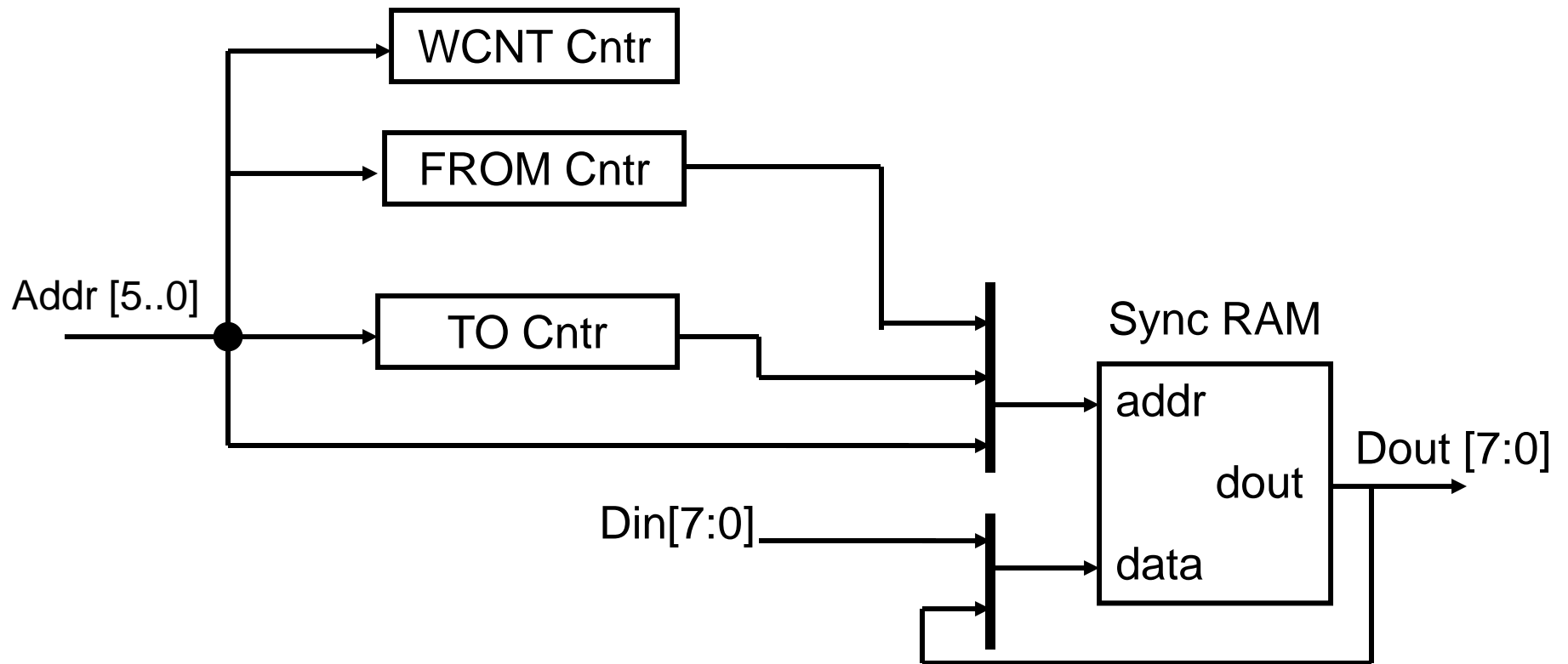
Datapath Components Needed

- Need the RAM
 - Counters for values
 - Word Count
 - To
 - From
 - Muxes
-

How do we put it together?

- Start with how the RAM is typically set up
 - We know we'll need to direct the addresses stored in FROM, TO registers to the RAM's address
 - We know we'll need to copy data from the RAM back into the RAM
 - We know that we need to increment the addresses in both blocks until WCNT is reached
-

Datapath Block Diagram



Control lines not shown on datapath diagram

What Control Lines do we need from FSM?

- **Counters:** Load lines for WCNT, TO, FROM registers driven externally and not under FSM control.
 - Count enables for these counters need to be exercised by FSM.
 - WCTN will be configured to count DOWN, the TO, FROM counters will count UP.
 - **Mux Selects:** When doing transfer operation, counters will be driving RAM address lines and RAM input data line will be a feedback from the RAM output.
 - **RAM:** The WE of the RAM needs to be an OR of the external WE and a WE that is provided by the FSM.
-

FSM Interface

■ **Inputs:**

- Clk, Reset
- xfer – kicks off transfer operation
- cnt_words[5:0] – WCNT counter value (need to check this to see if finished)

■ **Outputs:**

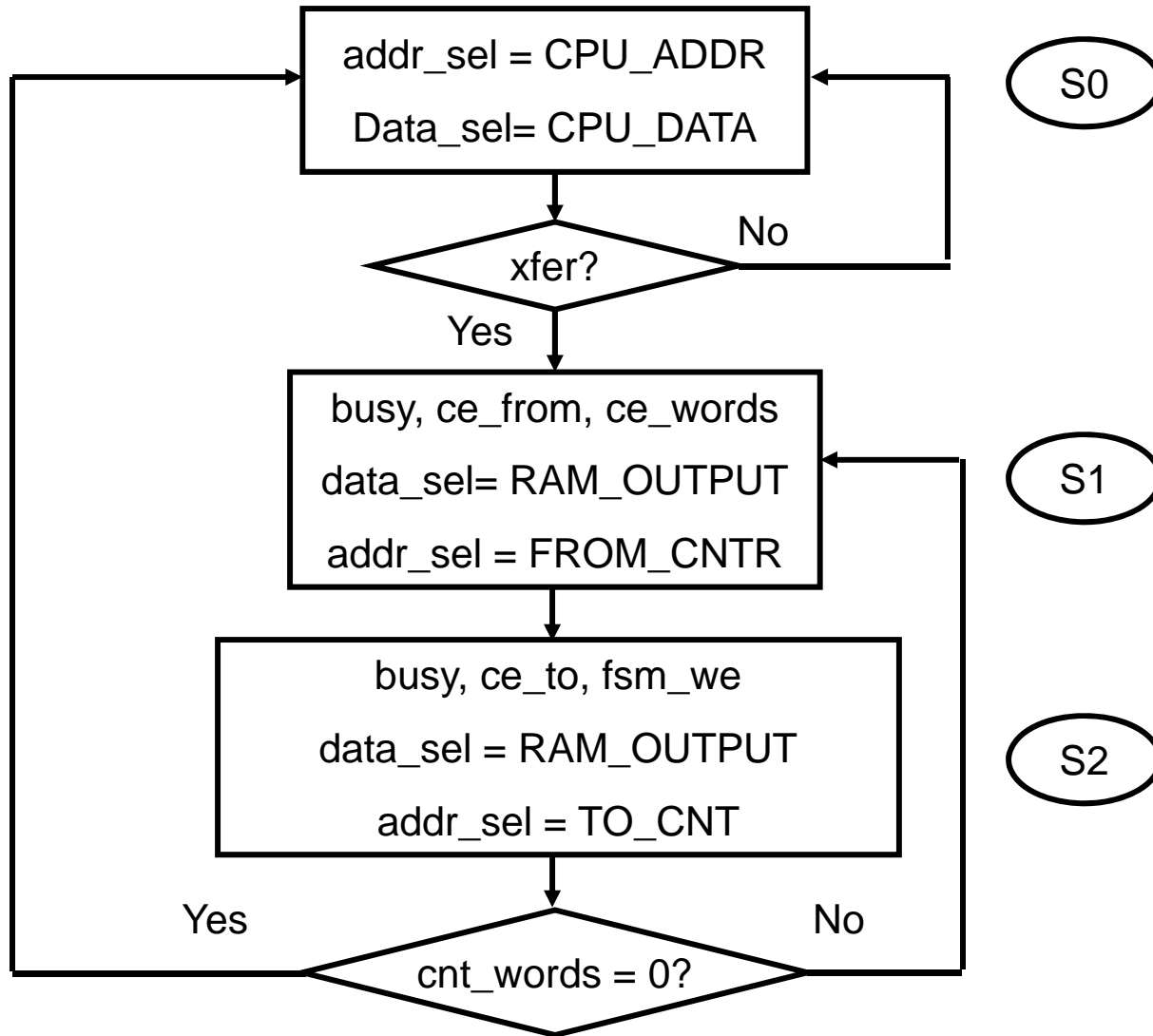
- busy – busy output
 - addr_sel[1:0] – mux select line for addr muxes
 - ce_from, ce_to, ce_words – count enables for FROM, TO, WCNT counters
 - fsm_we – WE to RAM
 - data_sel – mux select line for RAM input data mux
-

What operations do we need for FSM?

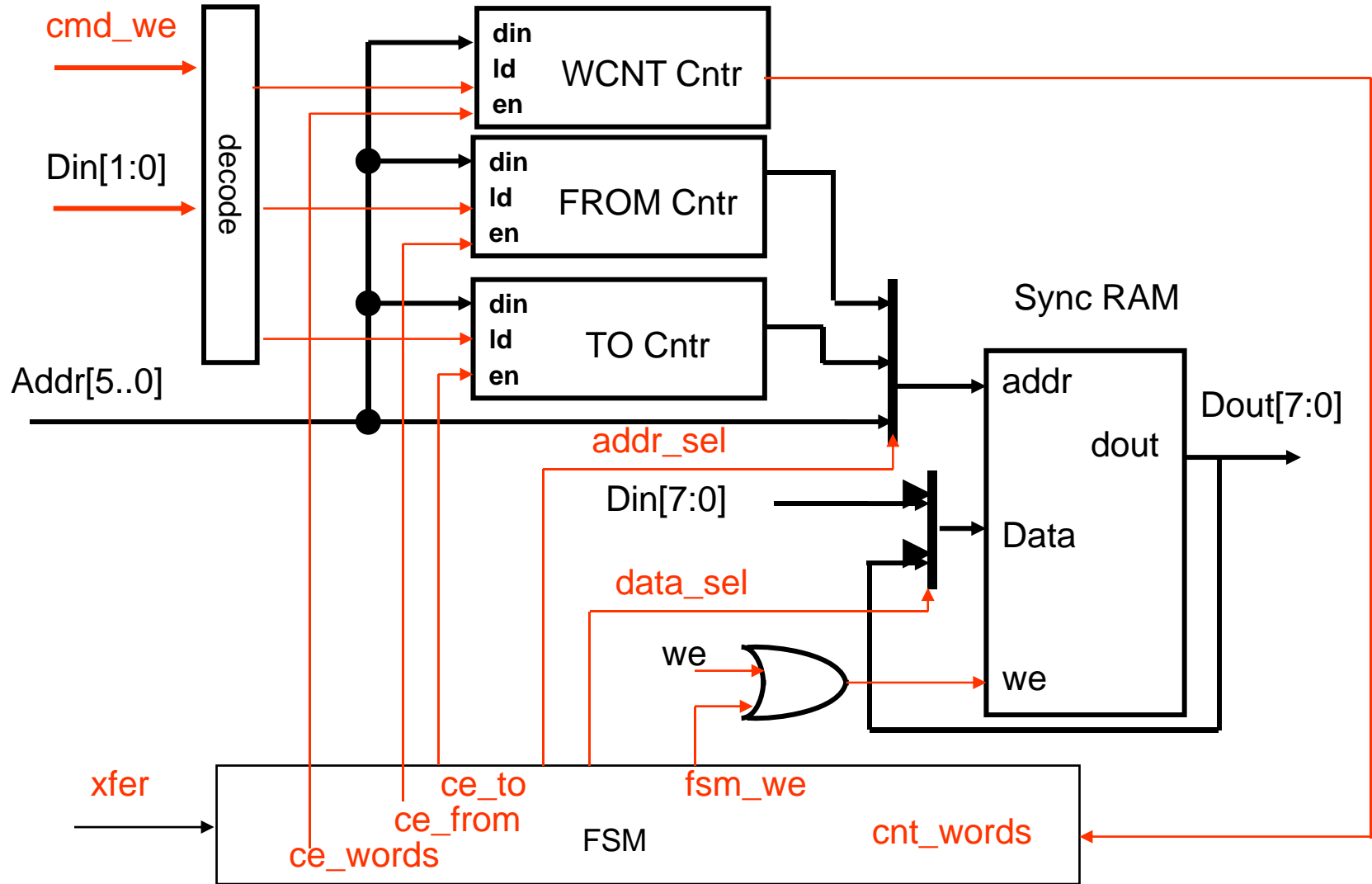
- Wait for transfer command (FSM simply waits for 'xfer' input to be asserted).
- Read a value from RAM using FROM counter address; increment the FROM counter and decrement the WCNT counter
- Write data value to RAM via TO address counter; increment the TO counter. Loop to read state unless WCNT counter value = 0.

Three DISTINCT operations, need three STATES in FSM.
Cannot do both a Read and Write in the same clock cycle.

Controller ASM

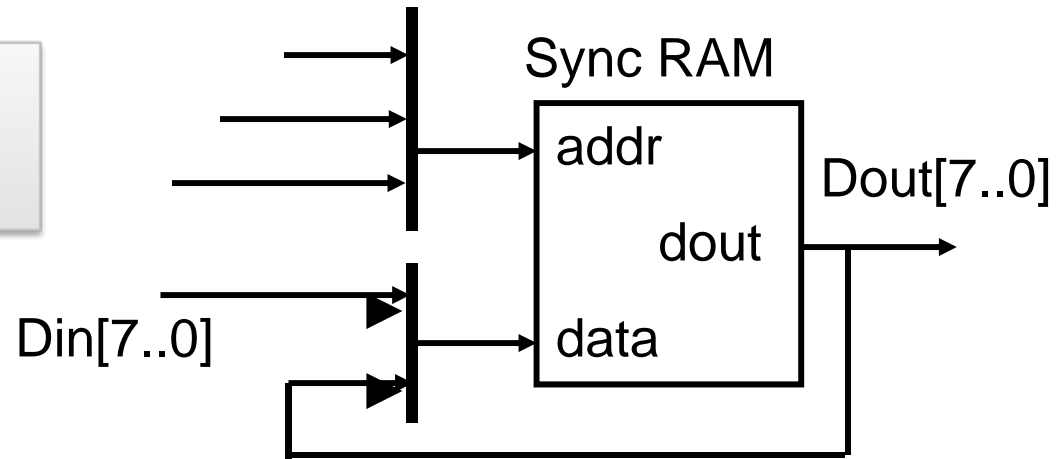


Datapath and Control



Comments

Why does the feedback path from Dout to Din work correctly?



- It only works because this is a sync RAM that has addr, input data, and control registered.
- The DOUT value are the memory contents corresponding to the address from the last clock edge (state S1, 'to' address).
- The current clock edge (state S2) will latch both this data and current address ('from' counter value) for the write operation.
- WOULD NOT WORK IF ASYNC RAM.

Design Implementation

- Once the datapath block diagram and ASM chart is done, the 'design' work is done. What is left is implementation.
 - Decide what parts of the datapath will be implemented in Verilog, what parts using pre-defined modules
 - The FSM control will certainly be done in Verilog
 - Misc registers can be easily included in the Verilog FSM code as well instead of using separate datapath components.
 - Do the schematic of the datapath FIRST!
 - Sometime just hooking up the datapath elements will expose a flaw in your reasoning.
-

Design Implementation

- After Datapath is finished, do FSM Verilog code
 - ALWAYS bring the FSM state value out as an external output for debugging purposes!!!
 - Should be able to write FSM code from ASM chart
 - DEBUG - take a systematic approach
 - Your design will NOT WORK the first time - be prepared to debug.
 - Attach external pins to as many internal nets as possible so that you can observe the internal net values
 - Debug your design ONE state at a time. Do not test the next state until the current state works as expected.
-

Design Implementation

- Until you get more confident with Verilog, should use as many pre-defined modules as you can
 - Can easily examine input/outputs in the simulator which makes it easier to debug
 - Always use a VERY LONG clock cycle to start out with so that you do not encounter timing problems
 - To be absolutely safe, make external inputs change on the falling edge if your internal logic is rising edge triggered (this gives you 1/2 clock of setup time).
-

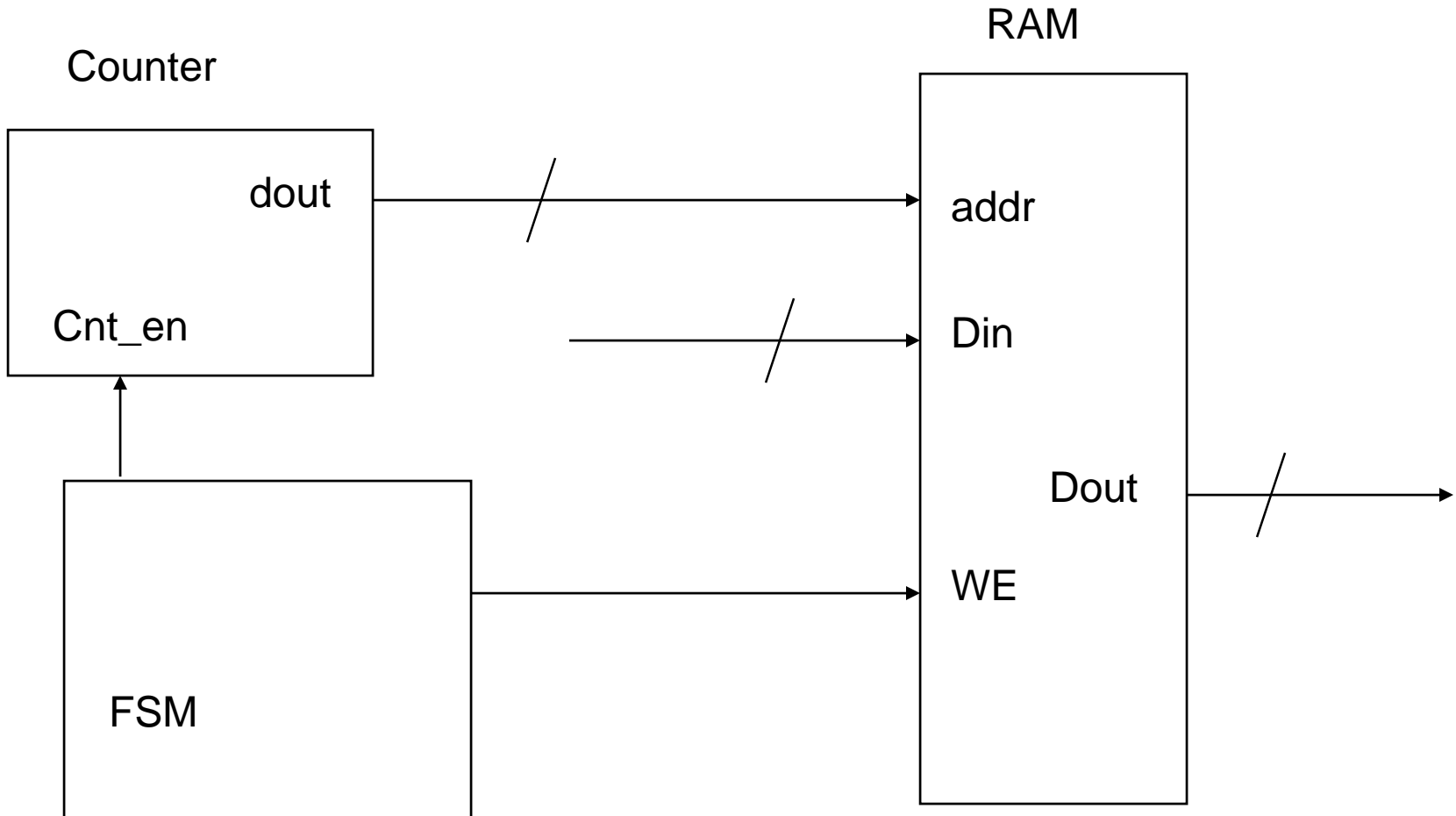
Tips and Tricks

- Do everything on paper first
 - Don't "hack"
 - If it's too complex – divide it into modules
 - Simpler circuits are easier to debug
 - Make your datapath dumb and your FSM smart
 - A bug/change in the datapath means you'll need to recheck your FSM
 - Changing the FSM is easy
-

Synchronous vs. Asynchronous RAM

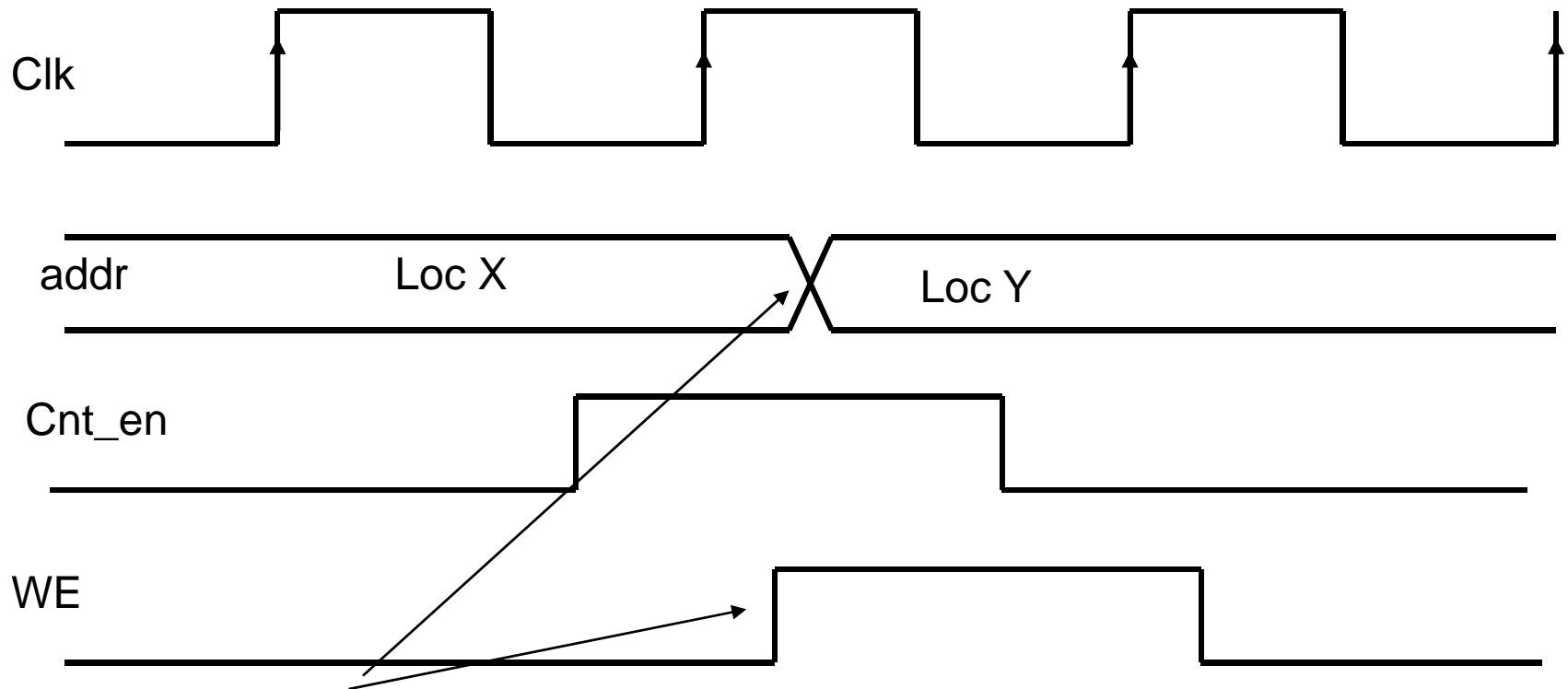
- Asynchronous RAM looks like a combinational element
 - No Clock
 - Data available after propagation delay from address
 - Address **MUST BE** stable while WE (write enable) is high so that only **ONE** location is written too. Data must also be stable during write cycle.
 - Synchronous RAM has a clock input and will latch input data and control lines (address, data)
-

Using Counters to Drive Address



Memory Design Issue

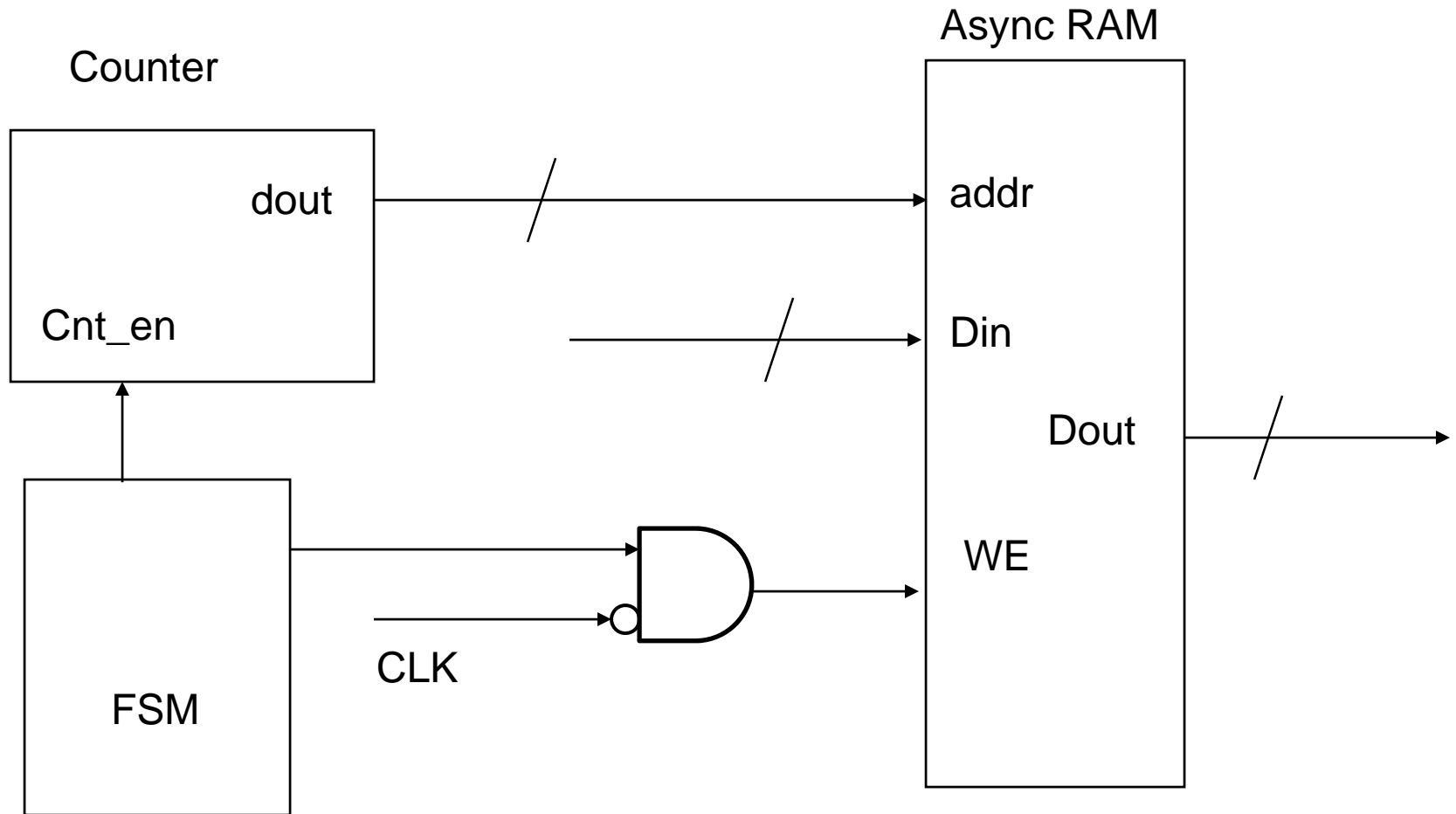
If WE and Address change on the same Clock Cycle, it may write multiple locations in Async RAM



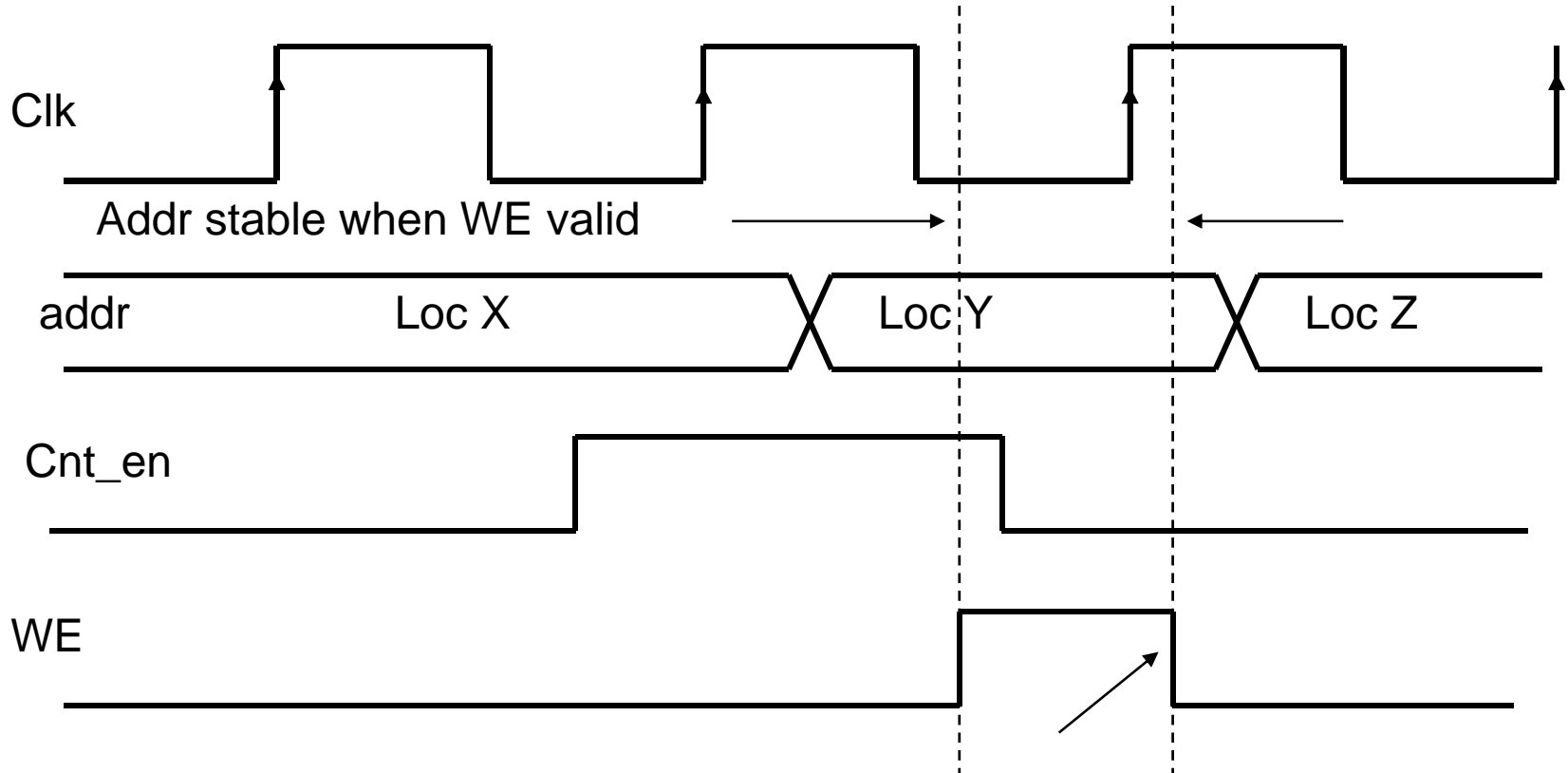
Delays can cause WE to change before or after address. If before, then can write to both Loc X and Loc Y

Memory Design Issue

Solution: Make WE valid only during 2nd half of clock cycle



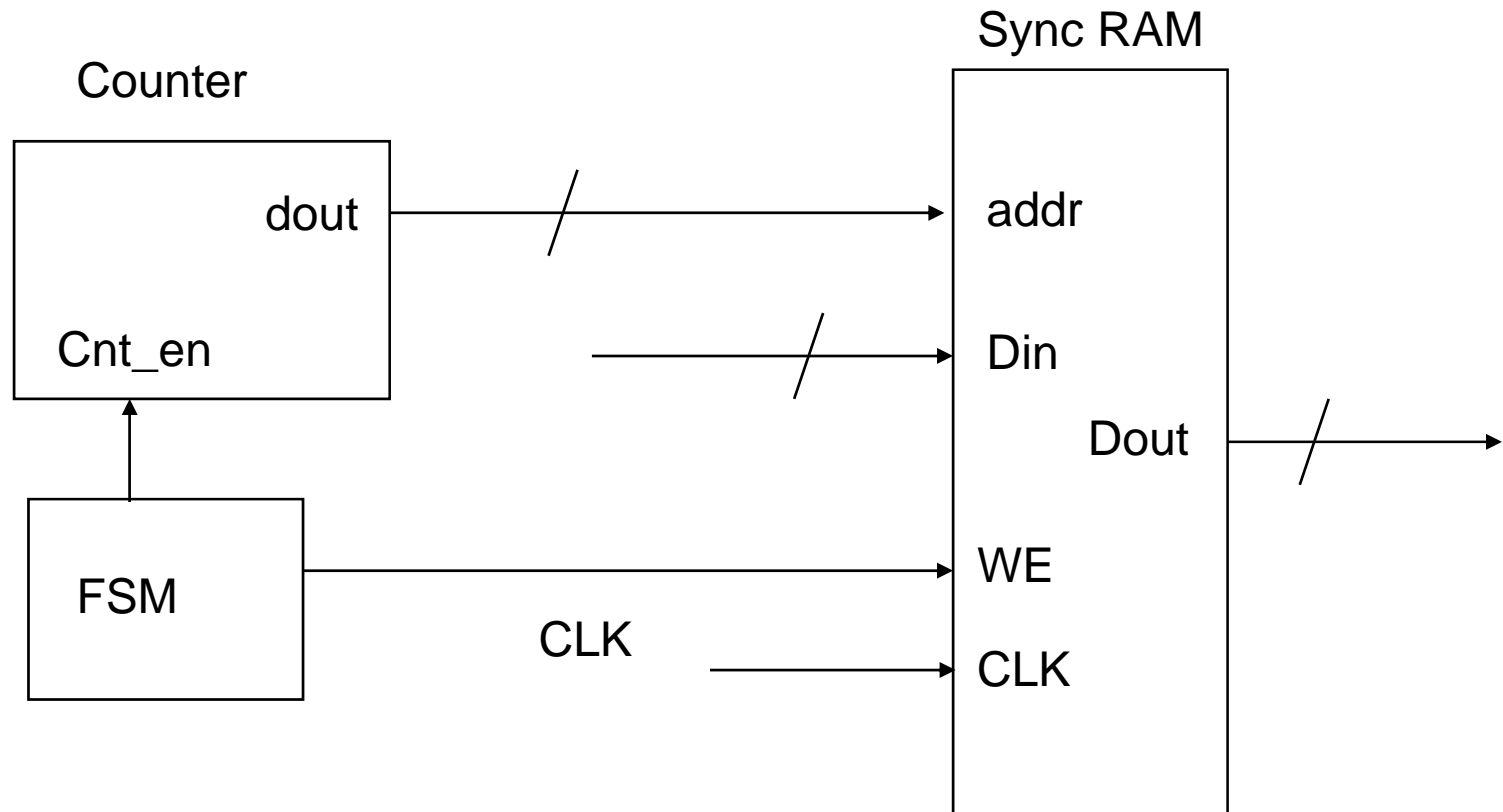
Memory Design Issue



WE must go invalid before address changes – assume delay through AND gate less than register delay of address value.

Synchronous RAM

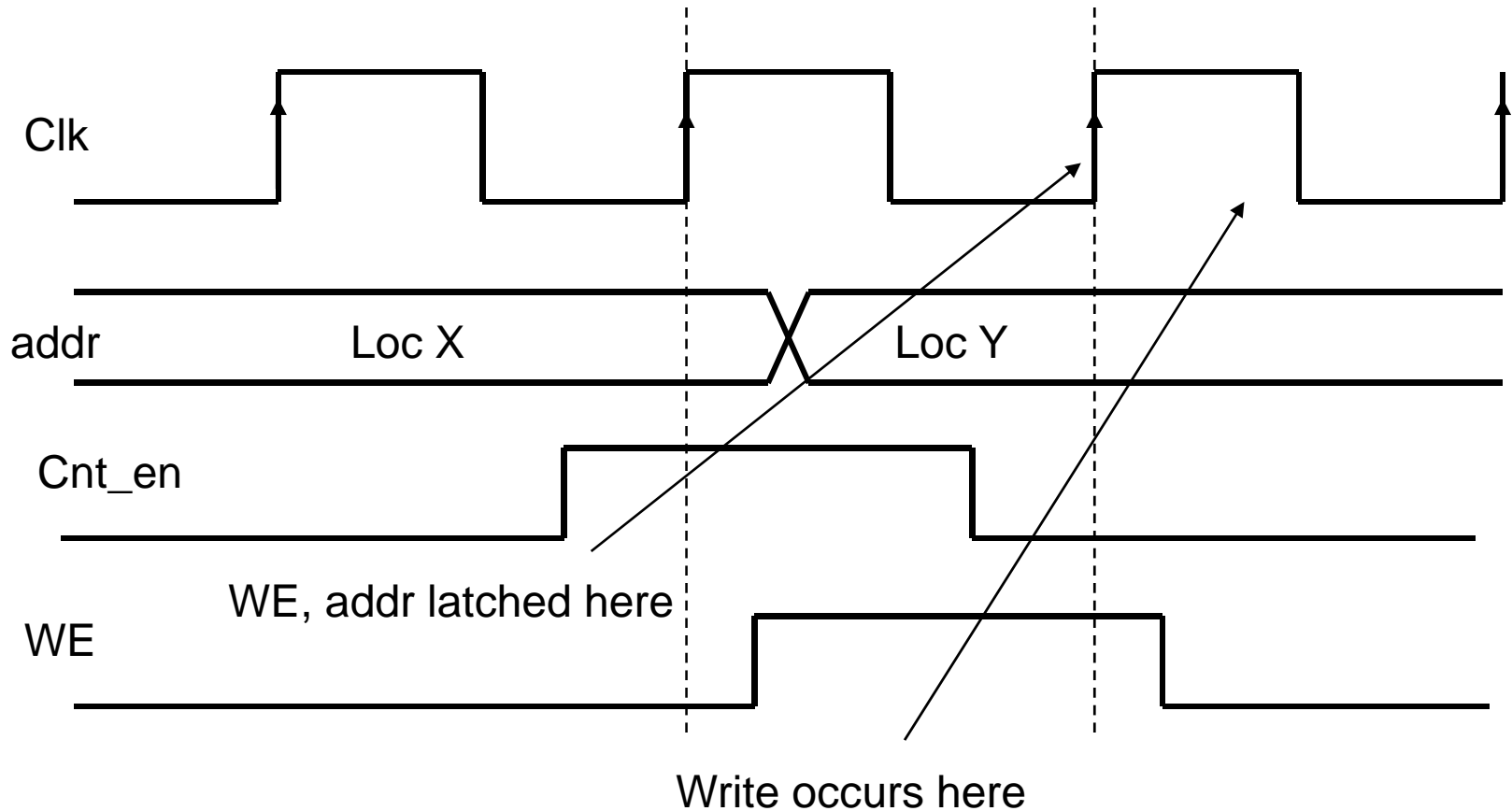
Sync Ram latches control, addr, input data. Can also latch output data



Registers used for latching input/output data are internal to RAM.

Synchronous RAM

Sync RAM latches address, WE

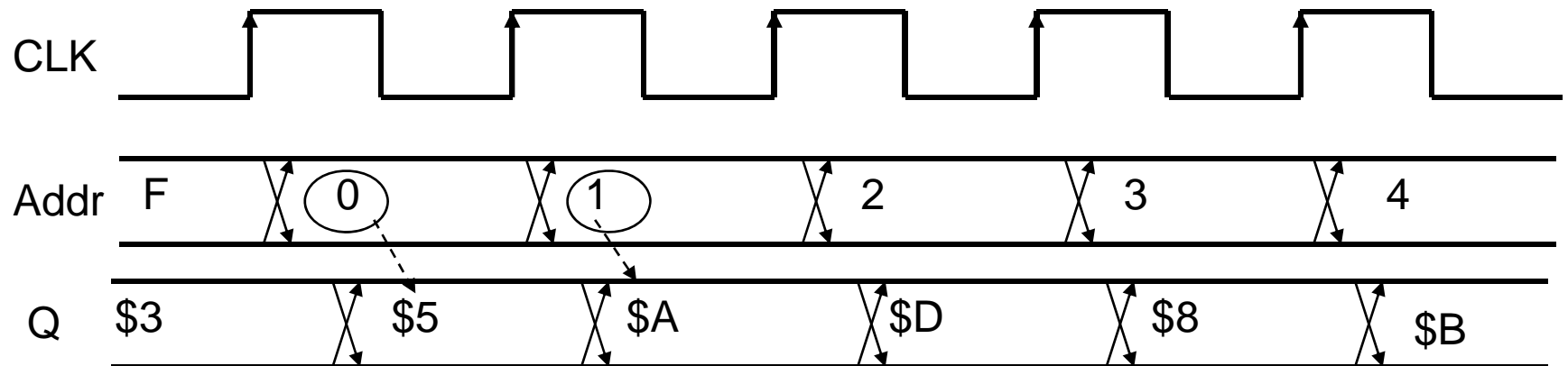


Synchronous RAM

- Will usually prefer Sync Ram
 - Sync RAM is easier to use from a timing perspective but adds latency to operations
 - If address coming from counter, then have an extra clock cycle of latency from when counter value is updated to when RAM data is available for that address
 - In our Lab exercises and class examples, will normally use synchronous RAM
 - Will not latch output data unless specifically needed
-

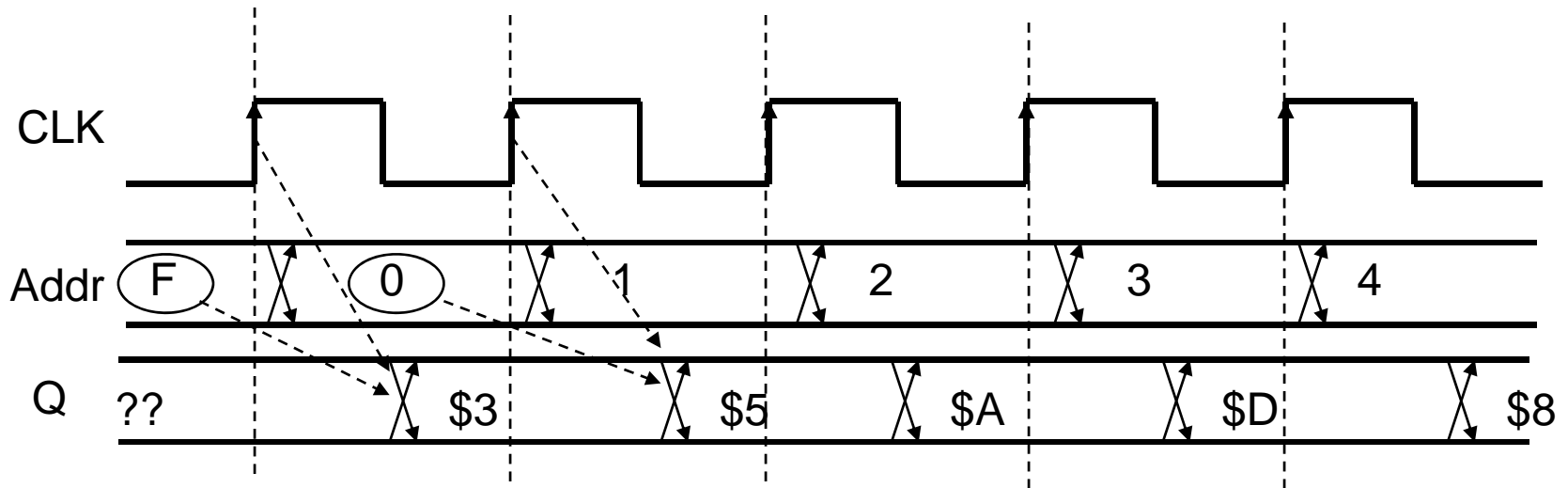
Asynchronous RAM Timing

- Asynchronous – no latching
- Address Control= “Unregistered”
- Input data = “Unregistered”
- Output data = “Unregistered”
- Assume 16x4 RAM
Contents: loc \$F= 3, \$0= 5, \$1= \$A, \$2= \$D, \$3 = \$8, \$4=\$B



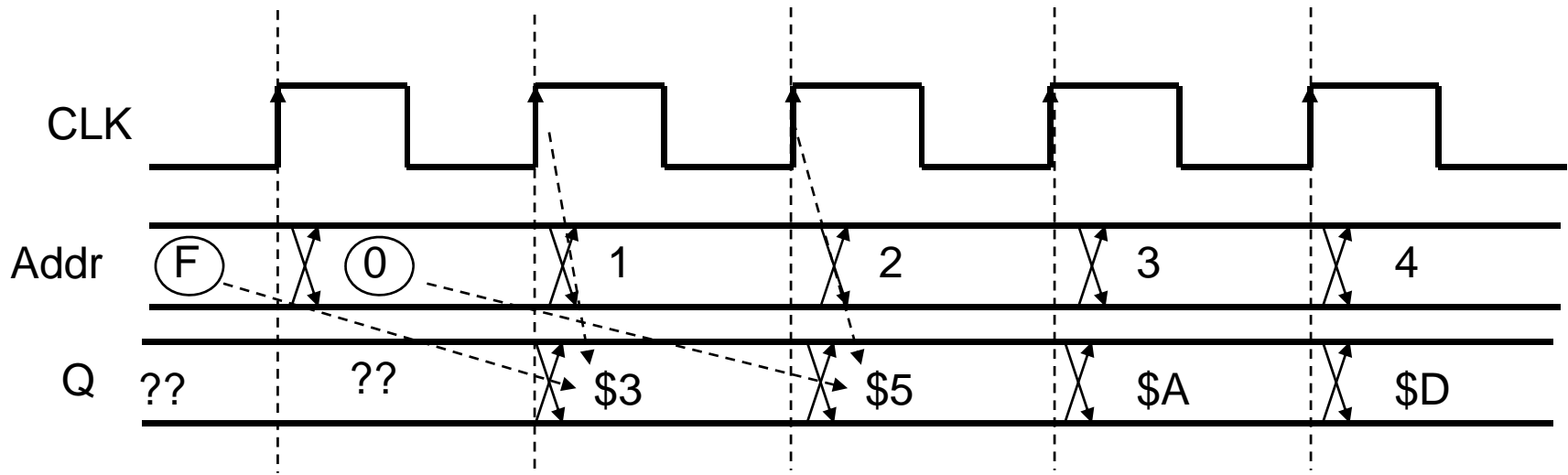
Synchronous RAM Timing

- Synchronous, latch input data and control
- Address Control= “Registered” (on ‘inclock’)
- Input data = “Registered” (on ‘inclock’)
- Output data = “Unregistered”
- Assume 16x4 RAM
Contents: loc \$F= 3, \$0= 5, \$1= \$A, \$2= \$D, \$3 = \$8, \$4=\$B



Synchronous RAM Timing

- Synchronous, latch input data, control, output data
- Address Control = “Registered” (on ‘inclock’)
- Input data = “Registered” (on ‘inclock’)
- Output data = “Registered” (on ‘outclock’)
- Assume 16x4 RAM
Contents: loc \$F= 3, \$0= 5, \$1= \$A, \$2= \$D, \$3 = \$8, \$4=\$B



Asynchronous vs. Synchronous Control

- Some modules have both synchronous and asynchronous control lines
 - Counter has 'aload' (asynchronous load), and 'sload' (synchronous load); 'aclr' and 'sclr' (async and sync clear)
 - Should always use a **Synchronous** control line if possible, especially if connected to a FSM output.
 - Any glitch on an asynchronous control line can trigger it
 - If using a FSM output for an asynchronous control, the output should come directly from a Flip-Flop output, NOT from combinational gating.
-