

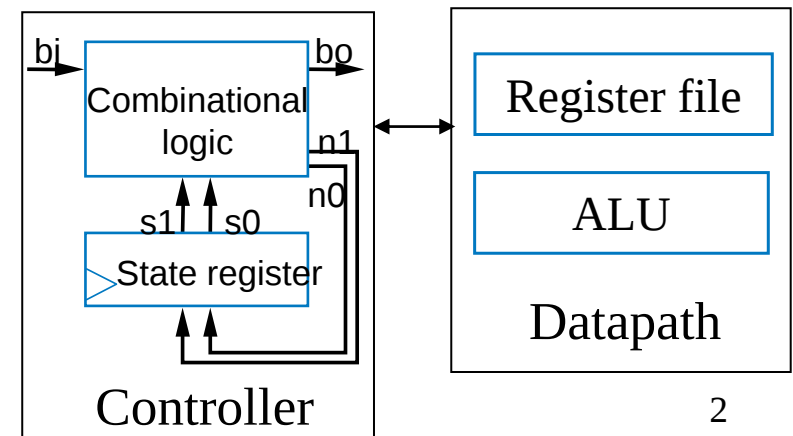
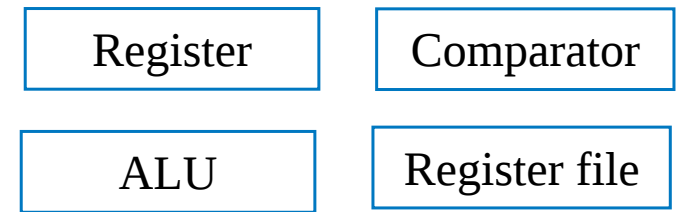
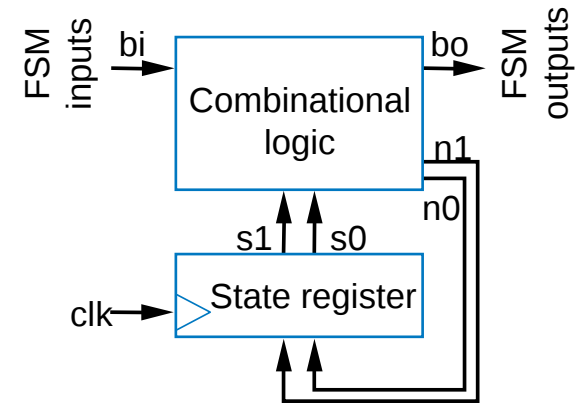


Digital Design

Chapter 5: Register-Transfer Level (RTL) Design

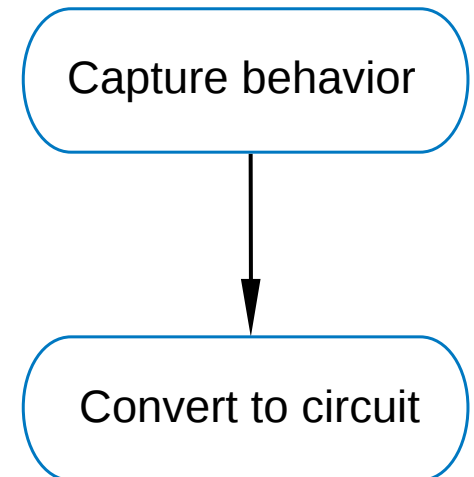
Introduction

- Chapter 3: Controllers
 - Control input/output: single bit (or just a few) representing event or state
 - Finite-state machine describes behavior; implemented as state register and combinational logic
- Chapter 4: Datapath components
 - Data input/output: Multiple bits collectively representing single entity
 - Datapath components included registers, adders, ALU, comparators, register files, etc.
- This chapter: custom **processors**
 - Processor: Controller and datapath components working together to implement an algorithm



RTL Design: Capture Behavior, Convert to Circuit

- Recall
 - Chapter 2: Combinational Logic Design
 - First step: Capture behavior (using equation or truth table)
 - Remaining steps: Convert to circuit
 - Chapter 3: Sequential Logic Design
 - First step: Capture behavior (using FSM)
 - Remaining steps: Convert to circuit
- RTL Design (the method for creating custom processors)
 - First step: Capture behavior (using high-level state machine, to be introduced)
 - Remaining steps: Convert to circuit



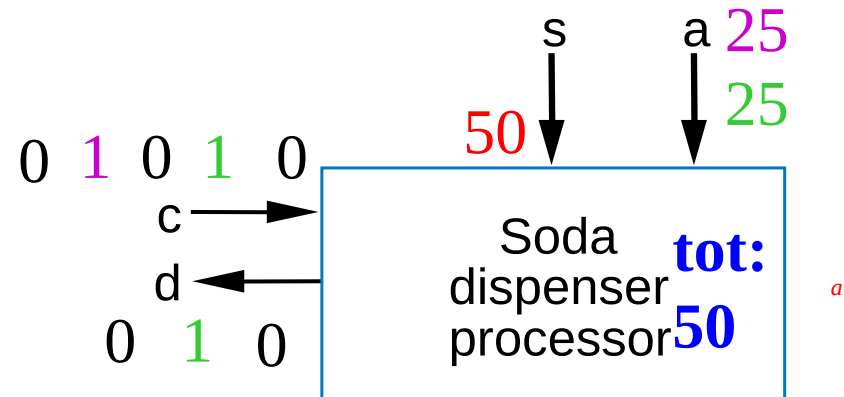
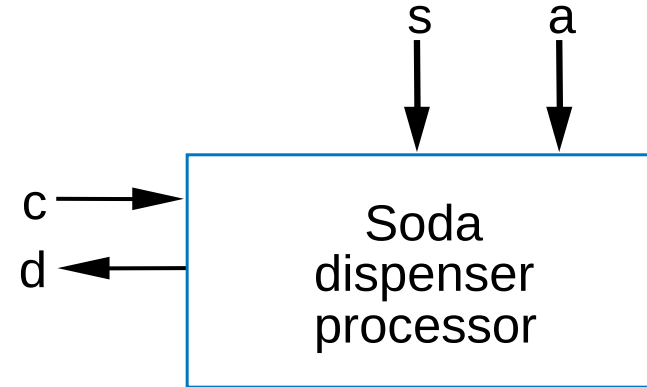
RTL Design Method

Step	Description
Step 1 <i>Capture a high-level state machine</i>	Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is "high-level" because the transition conditions and the state actions are more than just Boolean operations on bit inputs and outputs.
Step 2 <i>Create a datapath</i>	Create a datapath to carry out the data operations of the high-level state machine.
Step 3 <i>Connect the datapath to a controller</i>	Connect the datapath to a controller block. Connect external Boolean inputs and outputs to the controller block.
Step 4 <i>Derive the controller's FSM</i>	Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath.



RTL Design Method: “Preview” Example

- Soda dispenser
 - c : bit input, 1 when coin deposited
 - a : 8-bit input having value of deposited coin
 - s : 8-bit input having cost of a soda
 - d : bit output, processor sets to 1 when total value of deposited coins equals or exceeds cost of a soda

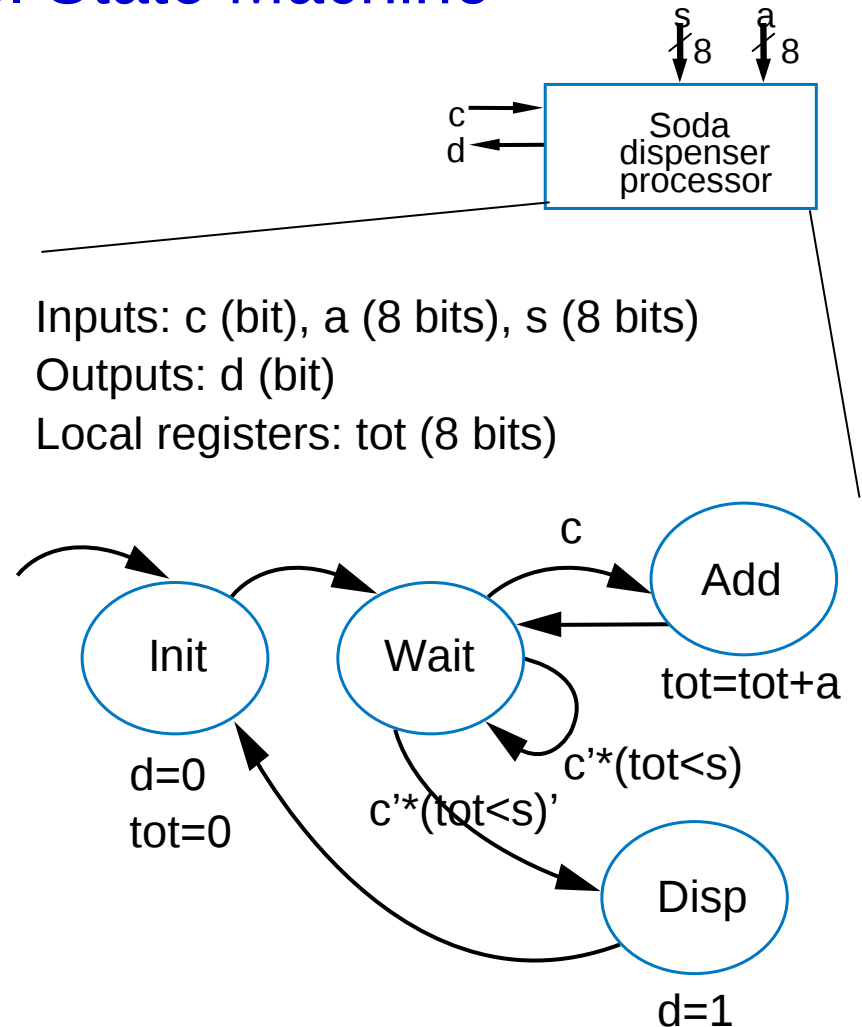


How can we precisely describe this processor's behavior?



Preview Example: Step 1 -- Capture High-Level State Machine

- Declare local register *tot*
- **Init** state: Set $d=0$, $tot=0$
- **Wait** state: wait for coin
 - If see coin, go to **Add** state
- **Add** state: Update total value:
 $tot = tot + a$
 - Remember, *a* is present coin's value
 - Go back to **Wait** state
- In **Wait** state, if $tot \geq s$, go to **Disp**(ense) state
- **Disp** state: Set $d=1$ (dispense soda)
 - Return to **Init** state

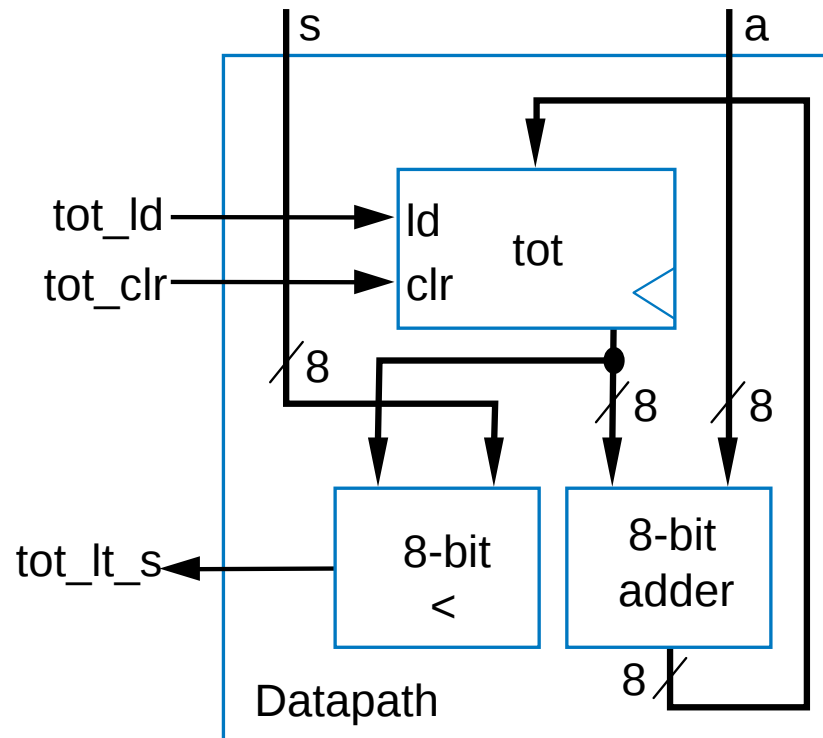
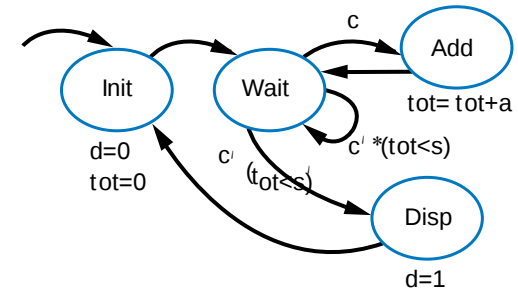


Preview Example:

Step 2 -- Create Datapath

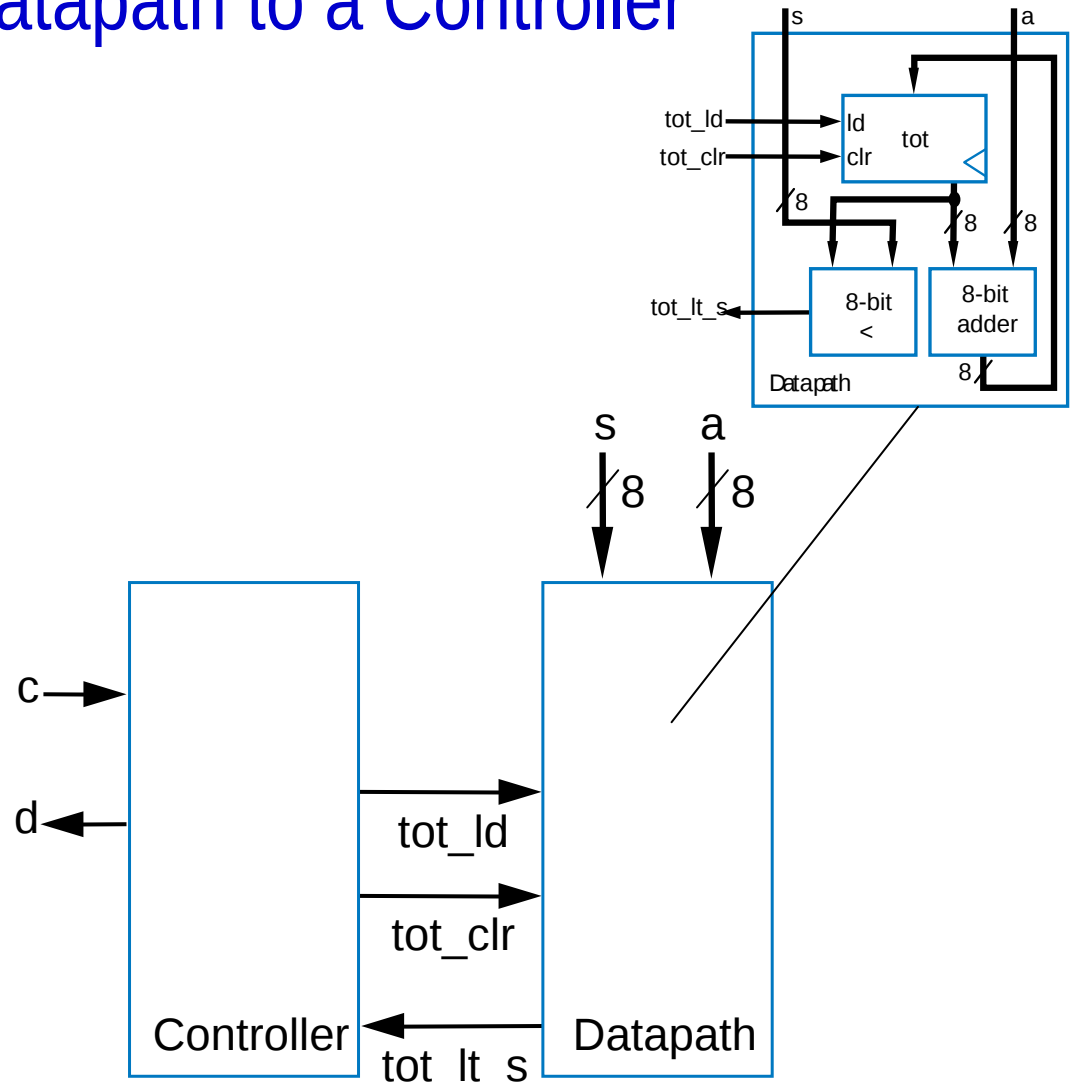
- Need *tot* register
- Need 8-bit comparator to compare *s* and *a*
- Need 8-bit adder to perform $tot = tot + a$
- Wire the components as needed for above
- Create control input/outputs, give them names

Inputs : *c* (bit), *a* (8 bits), *s* (8 bits)
 Outputs : *d* (bit)
 Local registers: *tot* (8 bits)



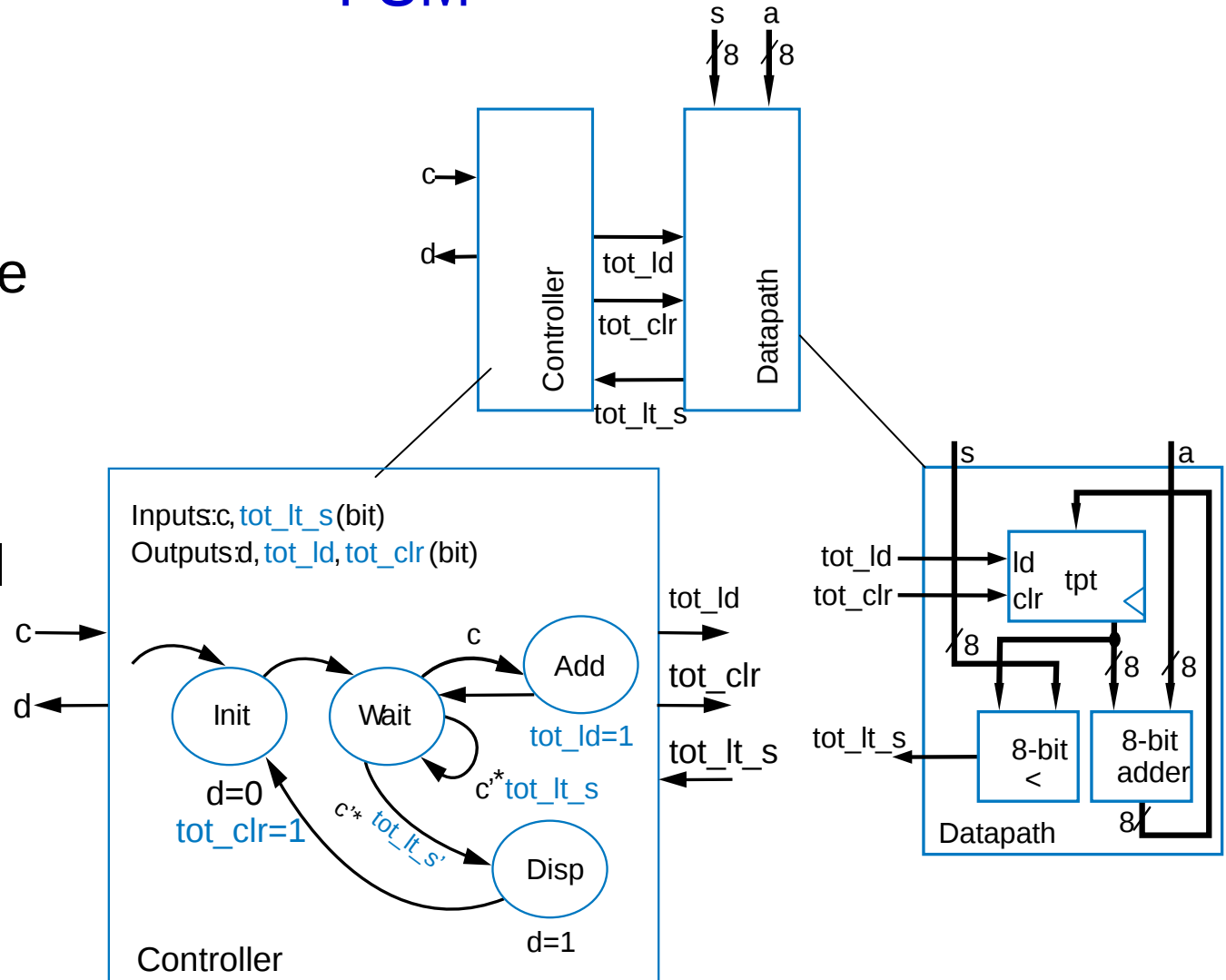
Preview Example: Step 3 – Connect Datapath to a Controller

- Controller's inputs
 - External input c (coin detected)
 - Input from datapath comparator's output, which we named tot_lt_s
- Controller's outputs
 - External output d (dispense soda)
 - Outputs to datapath to load and clear the tot register



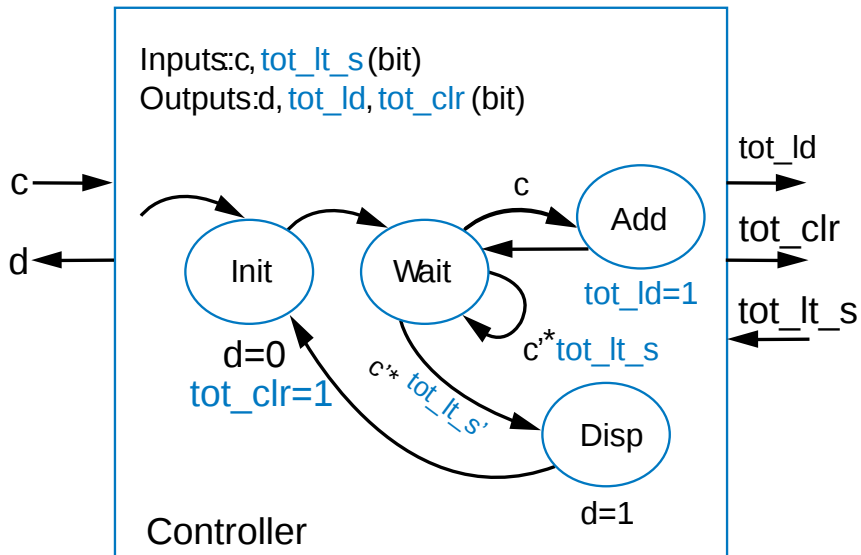
Preview Example: Step 4 – Derive the Controller's FSM

- Same states and arcs as high-level state machine
- But set/read datapath control signals for all datapath operations and conditions



Preview Example: Completing the Design

- Implement the FSM as a state register and logic
 - As in Ch3
 - Table shown on right



	s1	s0	c	tot_lt_s	n1	n0	d	tot_ld	tot_clr
Init	0	0	0	0	0	1	0	0	1
	0	0	0	1	0	1	0	0	1
	0	0	1	0	0	1	0	0	1
	0	0	1	1	0	1	0	0	1
Wait	0	1	0	0	1	1	0	0	0
	0	1	0	1	0	1	0	0	0
	0	1	1	0	1	0	0	0	0
	0	1	1	1	1	0	0	0	0
Add	1	0	0	0	0	1	0	1	0
			...			...			
Disp	1	1	0	0	0	0	1	0	0
			...			...			



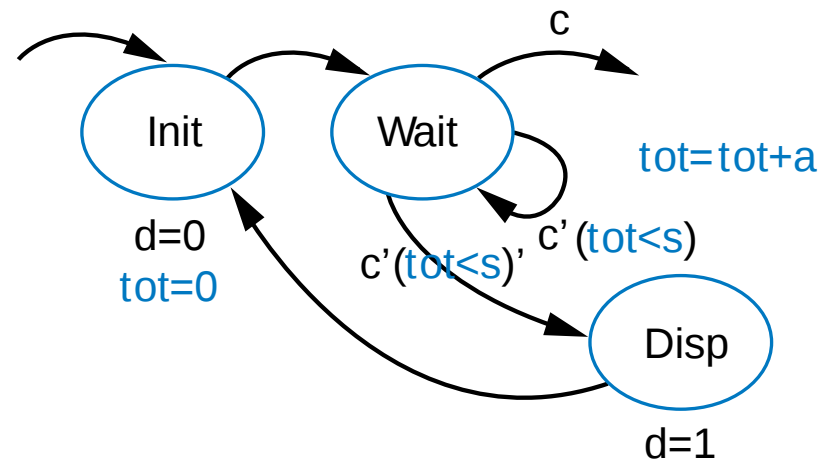
Step 1: Create a High-Level State Machine

- Let's consider each step of the RTL design process in more detail
- Step 1
 - Soda dispenser example
 - Not an FSM because:
 - Multi-bit (data) inputs a and s
 - Local register tot
 - Data operations $tot=0$, $tot<s$, $tot=tot+a$.
 - Useful high-level state machine:
 - Data types beyond just bits
 - Local registers
 - Arithmetic equations/expressions

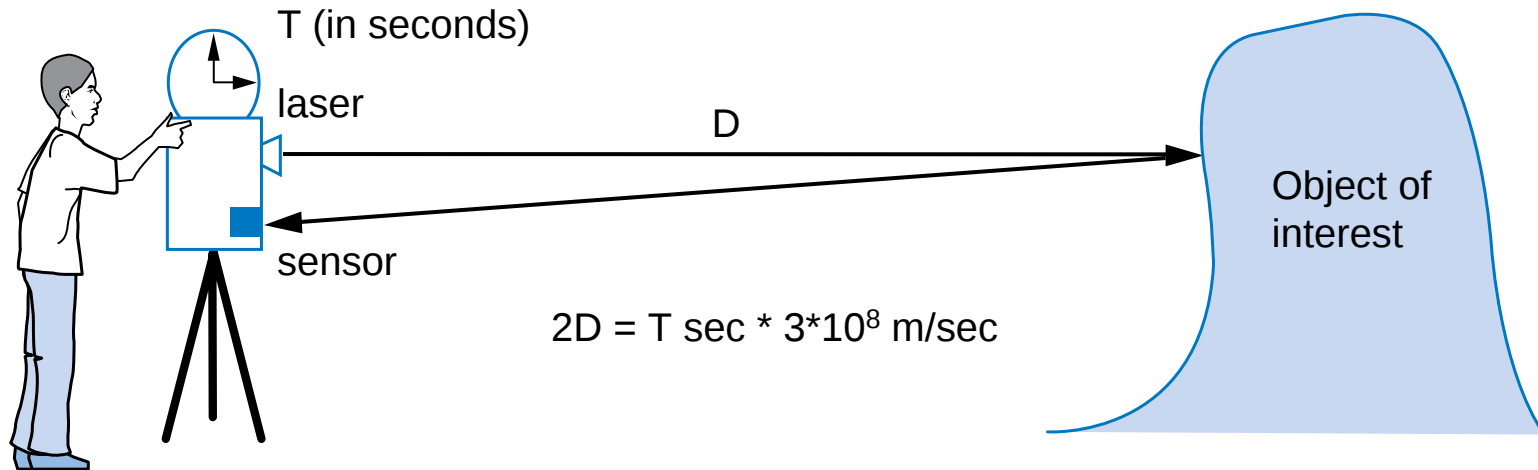
Inputs: c (bit), a (8 bits), s (8 bits)

Outputs: d (bit)

Local registers: tot (8 bits)



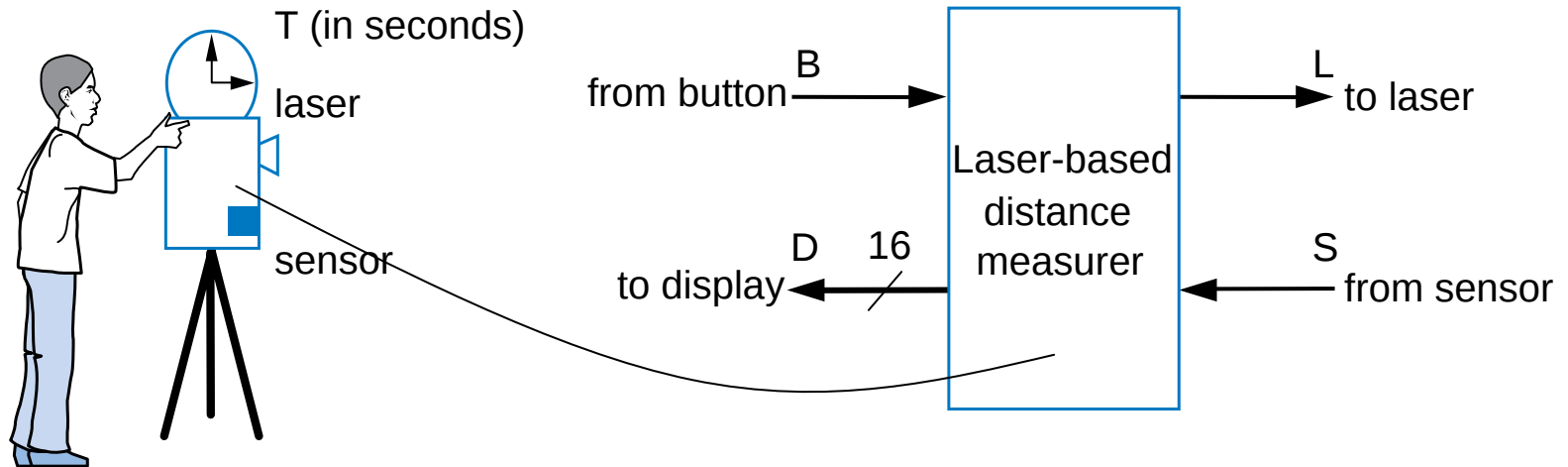
Step 1 Example: Laser-Based Distance Measurer



- Example of how to create a high-level state machine to describe desired processor behavior
- Laser-based distance measurement – pulse laser, measure time T to sense reflection
 - Laser light travels at speed of light, $3 \cdot 10^8 \text{ m/sec}$
 - Distance is thus $D = T \text{ sec} * 3 \cdot 10^8 \text{ m/sec} / 2$



Step 1 Example: Laser-Based Distance Measurer



- Inputs/outputs

- B : bit input, from button to begin measurement
- L : bit output, activates laser
- S : bit input, senses laser reflection
- D : 16-bit output, displays computed distance



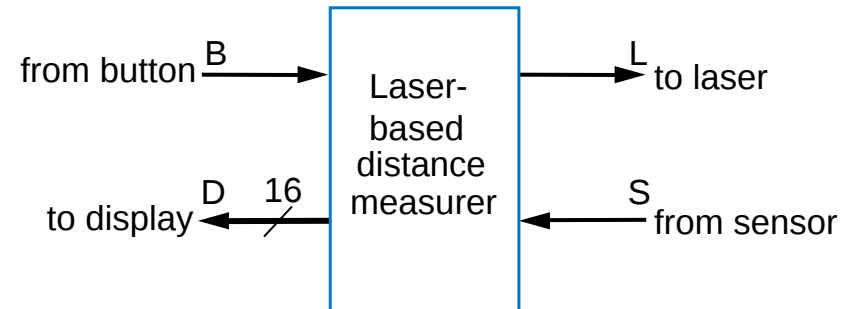
Step 1 Example: Laser-Based Distance Measurer

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)



a

L = 0 (laser off)
D = 0 (distance = 0)

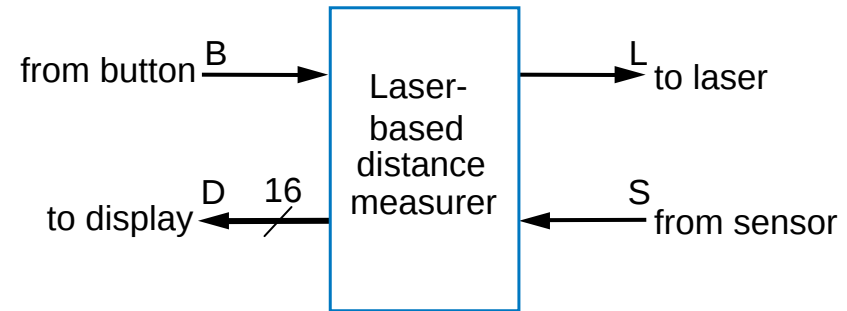
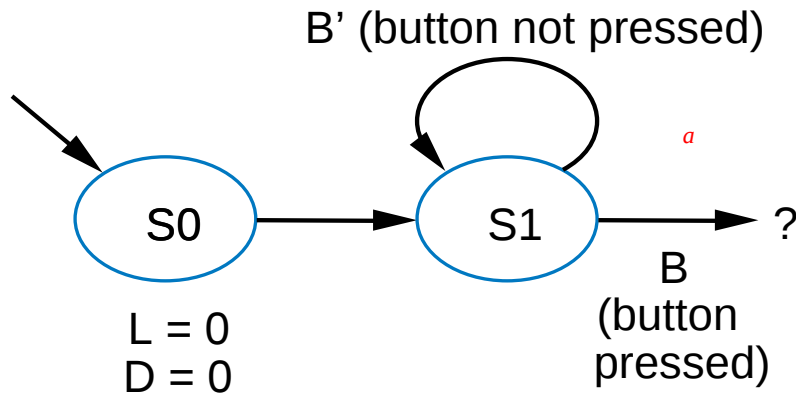


- Step 1: Create high-level state machine
- Begin by declaring inputs and outputs
- Create initial state, name it **S0**
 - Initialize laser to off (L=0)
 - Initialize displayed distance to 0 (D=0)



Step 1 Example: Laser-Based Distance Measurer

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)



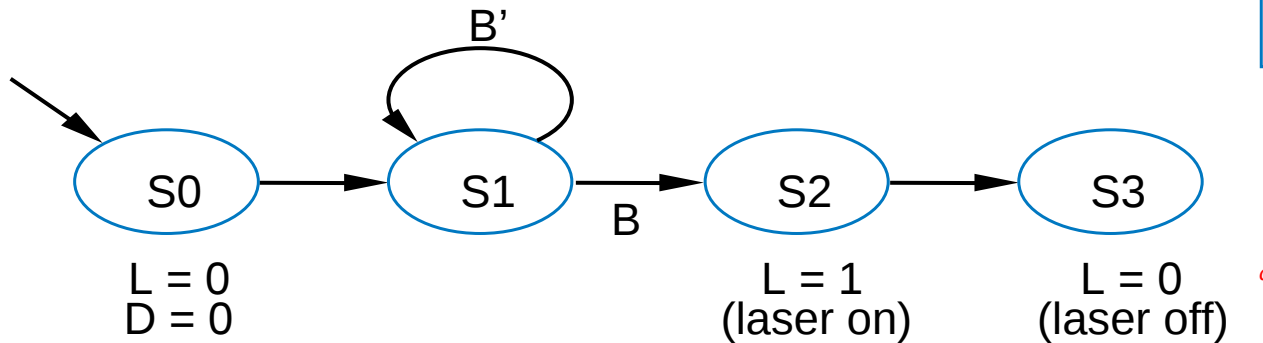
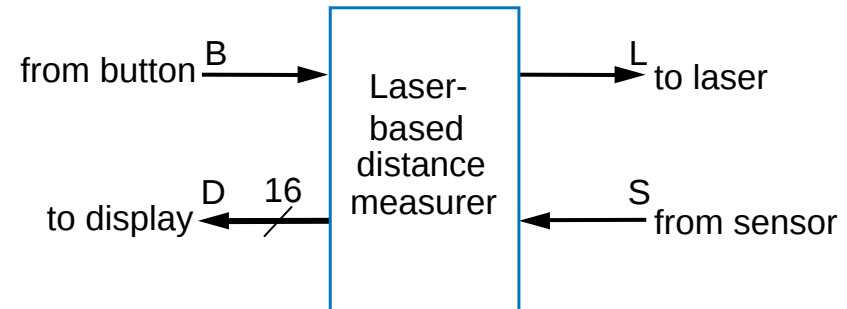
- Add another state, call **S1**, that waits for a button press
 - B' – stay in **S1**, keep waiting
 - B – go to a new state **S2**

Q: What should S2 do? **A: Turn on the laser**



Step 1 Example: Laser-Based Distance Measurer

Inputs: B, S (1 bit each)
Outputs: L (bit), D (16 bits)



- Add a state **S2** that turns on the laser (L=1)
- Then turn off laser (L=0) in a state **S3**

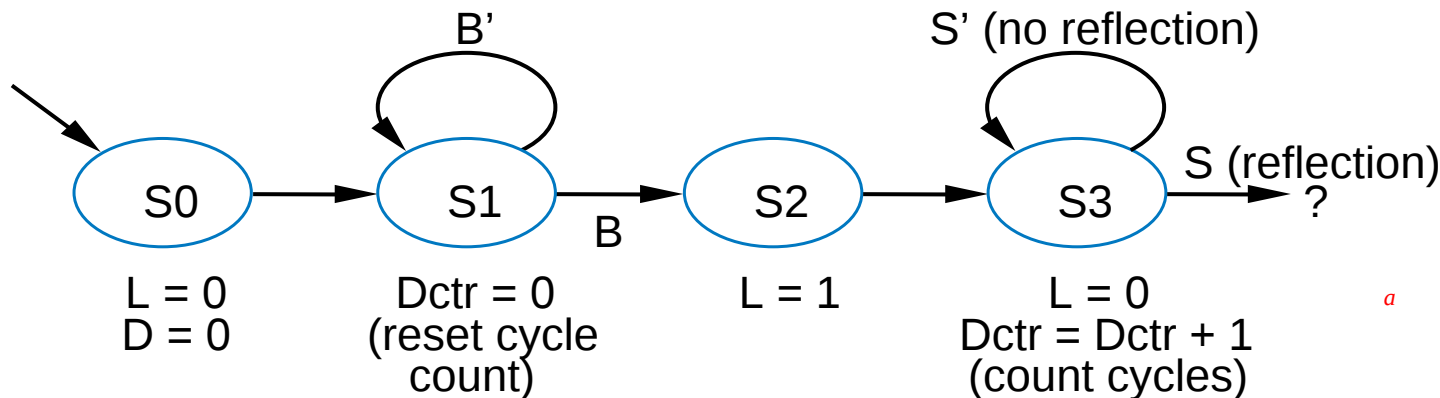
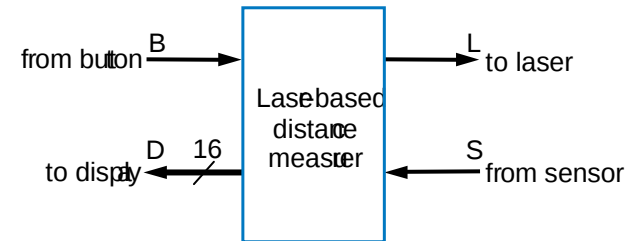
Q: What do next? A: Start timer, wait to sense reflection



Step 1 Example: Laser-Based Distance Measurer

Inputs: B, S (1 bit each) Outputs: L (bit), D (16 bits)

Local Registers: Dctr (16 bits)

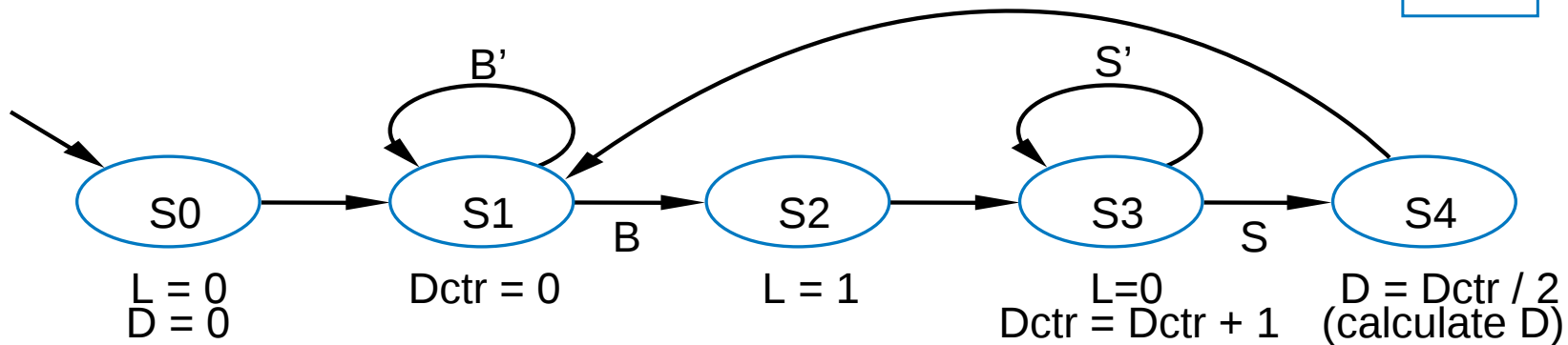
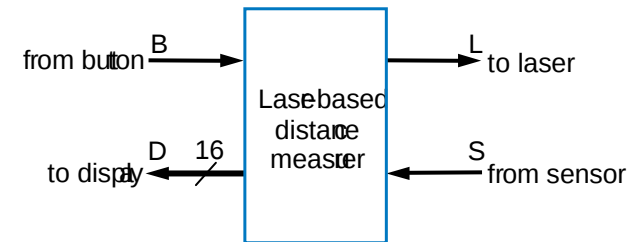


- Stay in **S3** until sense reflection (S)
- To measure time, count cycles for which we are in **S3**
 - To count, declare local register *Dctr*
 - Increment *Dctr* each cycle in **S3**
 - Initialize *Dctr* to 0 in **S1**. **S2** would have been O.K. too



Step 1 Example: Laser-Based Distance Measurer

Inputs: B, S (1 bit each) Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)



- Once reflection detected (S), go to new state **S4**
 - Calculate distance
 - Assuming clock frequency is 3×10^8 , *Dctr* holds number of meters, so $D = Dctr / 2$
- After **S4**, go back to **S1** to wait for button again



Step 2: Create a Datapath

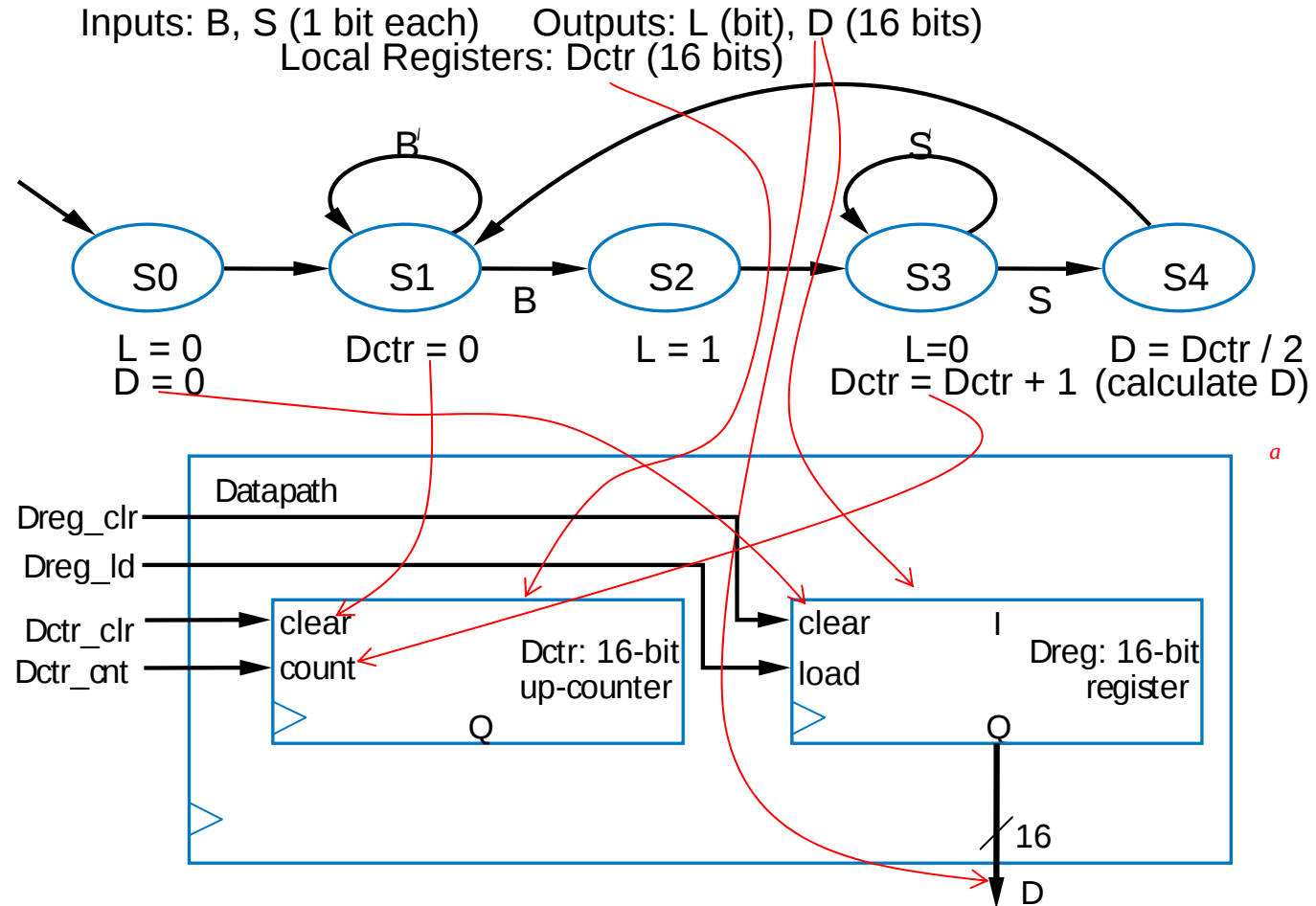
- Datapath must
 - Implement data storage
 - Implement data computations
- Look at high-level state machine, do three substeps
 - (a) Make data inputs/outputs be datapath inputs/outputs
 - (b) Instantiate declared registers into the datapath (also instantiate a register for each data output)
 - (c) Examine every state and transition, and instantiate datapath components and connections to implement any data computations

Instantiate: to introduce a new component into a design.



Step 2 Example: Laser-Based Distance Measurer

- Make data inputs/outputs be datapath inputs/outputs
- Instantiate declared registers into the datapath (also instantiate a register for each data output)
- Examine every state and transition, and instantiate datapath components and connections to implement any data computations

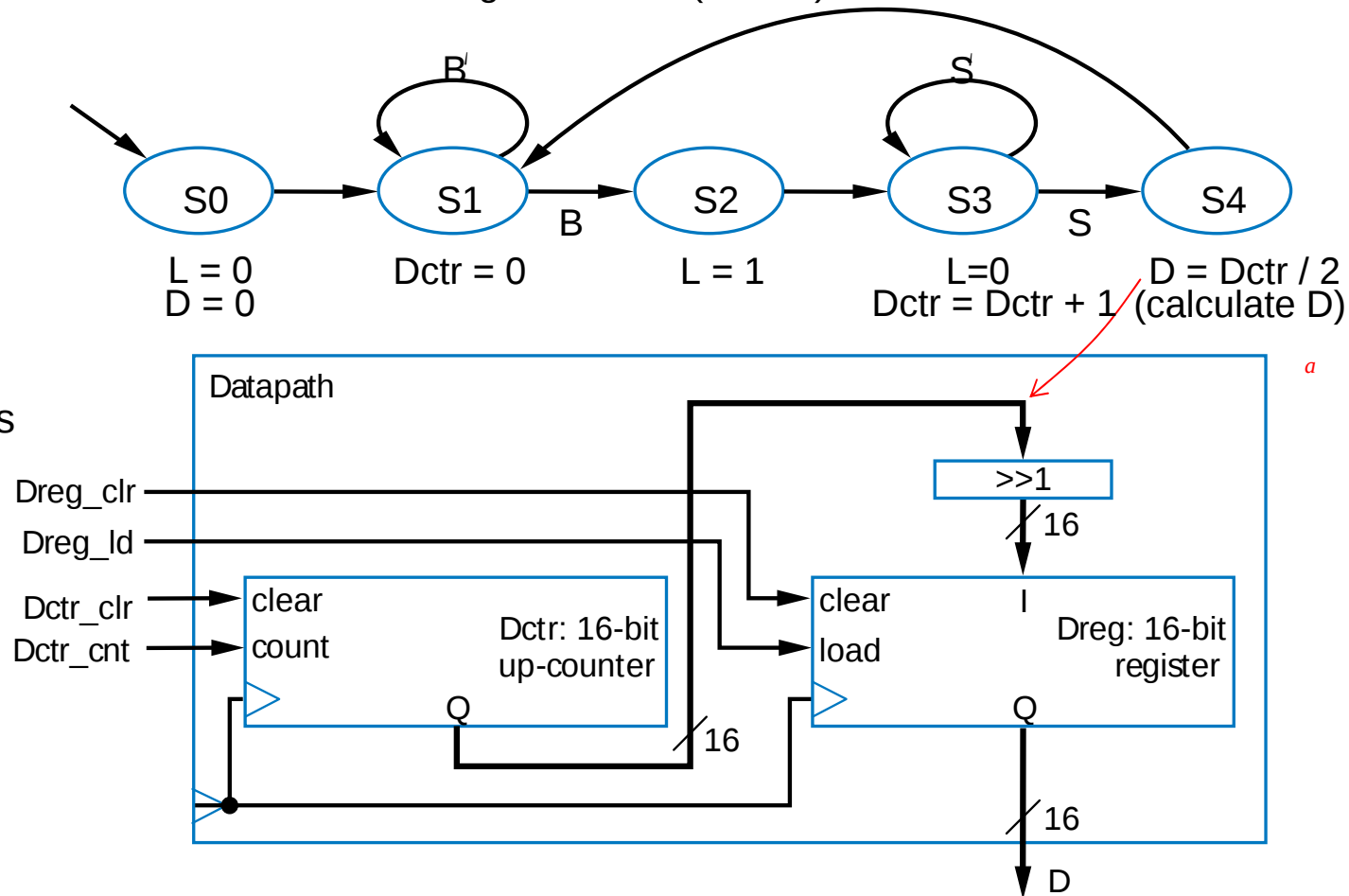


Step 2 Example: Laser-Based Distance Measurer

(c) (continued)

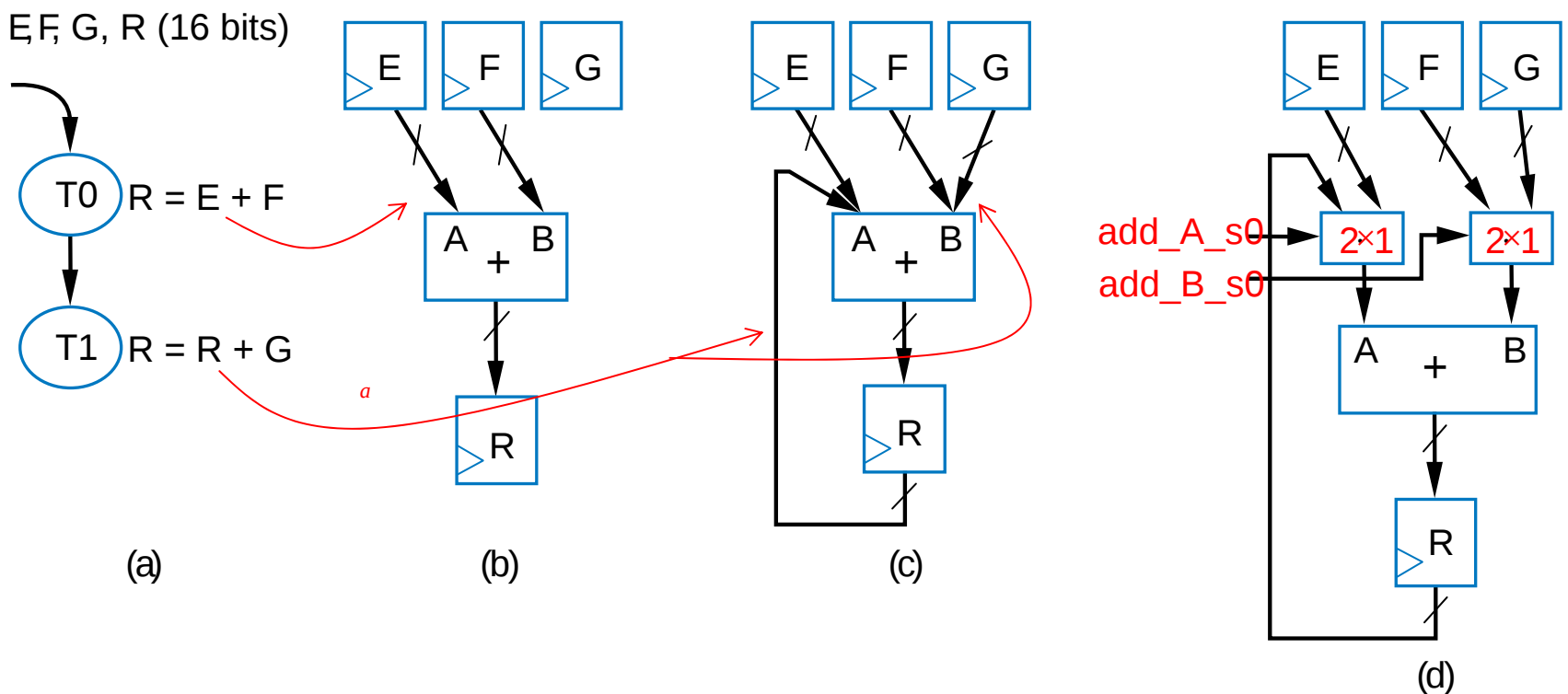
Examine every state and transition, and instantiate datapath components and connections to implement any data computations

Inputs: B, S (1 bit each) Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)



Step 2 Example Showing Mux Use

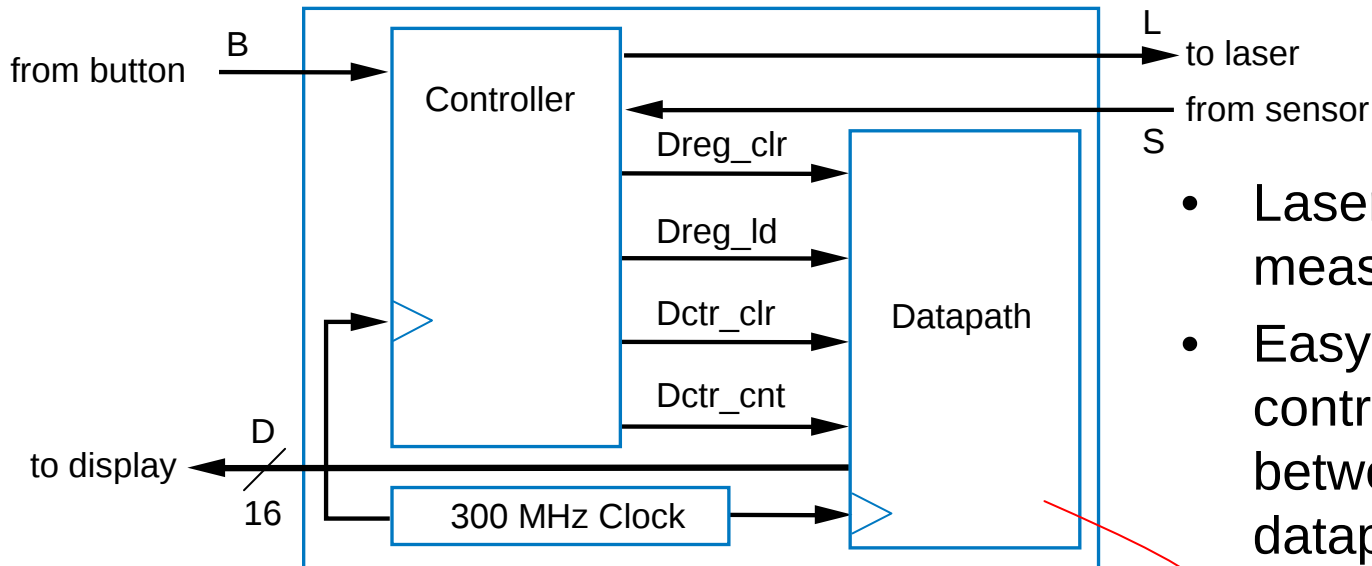
Local registers
E, F, G, R (16 bits)



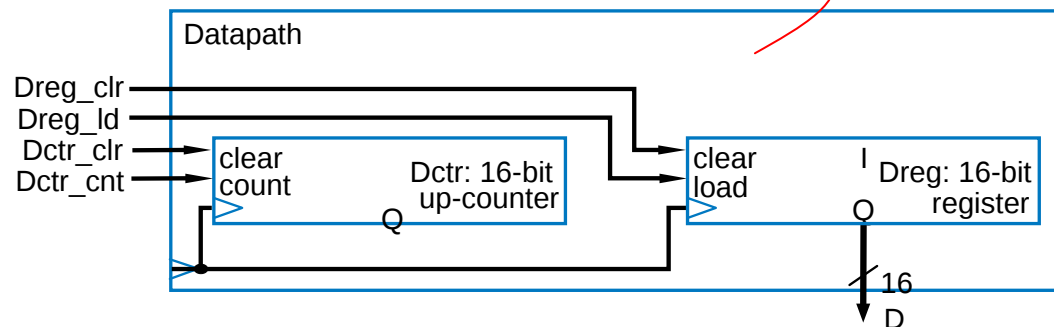
- Introduce mux when one component input can come from more than one source



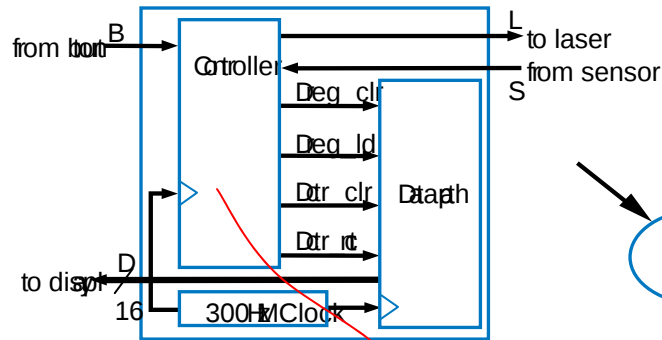
Step 3: Connecting the Datapath to a Controller



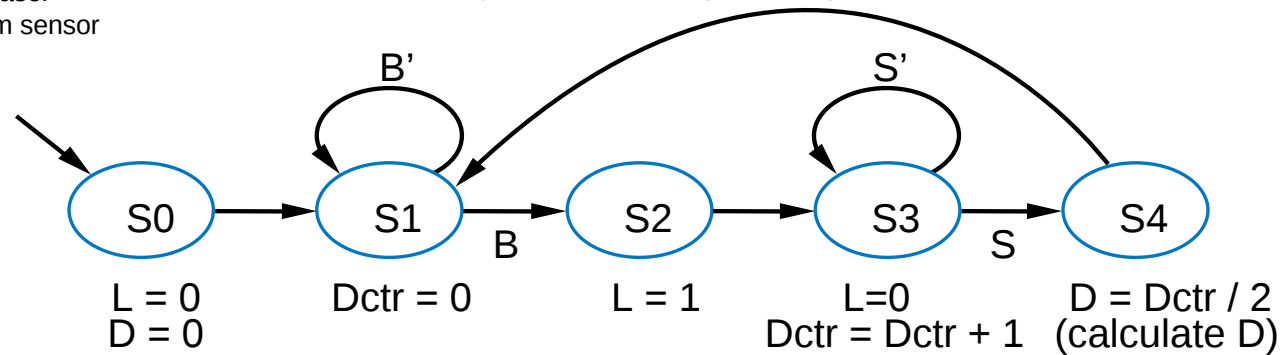
- Laser-based distance measurer example
- Easy – just connect all control signals between controller and datapath



Step 4: Deriving the Controller's FSM



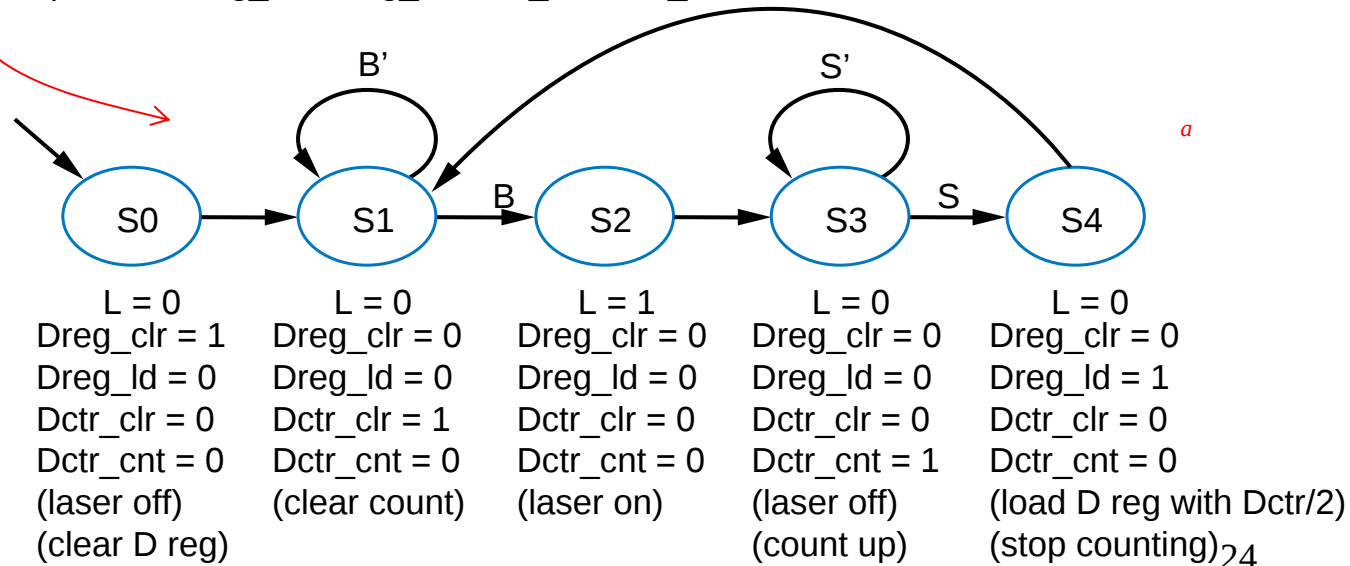
Inputs: B, S (1 bit each) Outputs: L (bit), D (16 bits)
Local Registers: Dctr (16 bits)



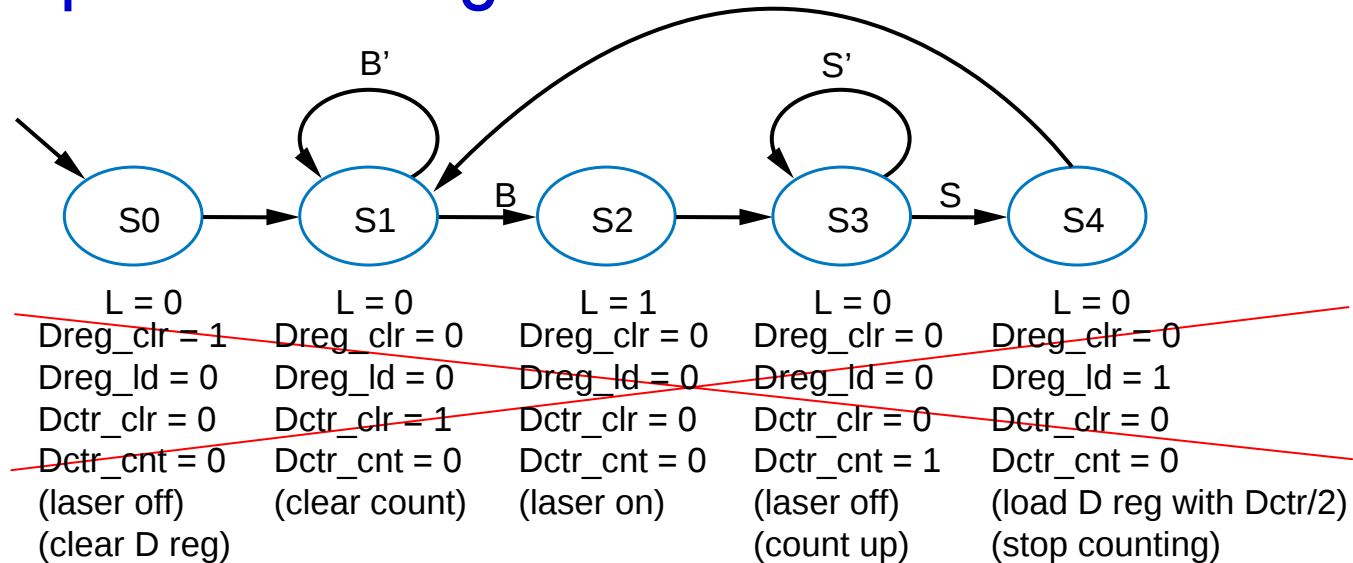
FSM has same structure as high-level state machine

- Inputs/outputs all bits now
- Replace data operations by bit operations using datapath

Inputs: B, S
Outputs: L, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_cnt

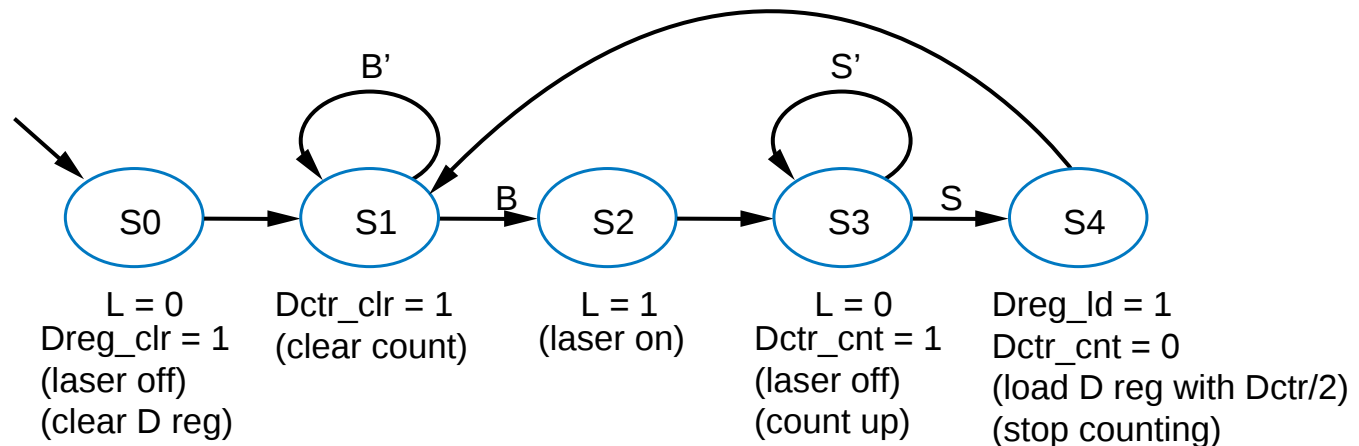


Step 4: Deriving the Controller's FSM

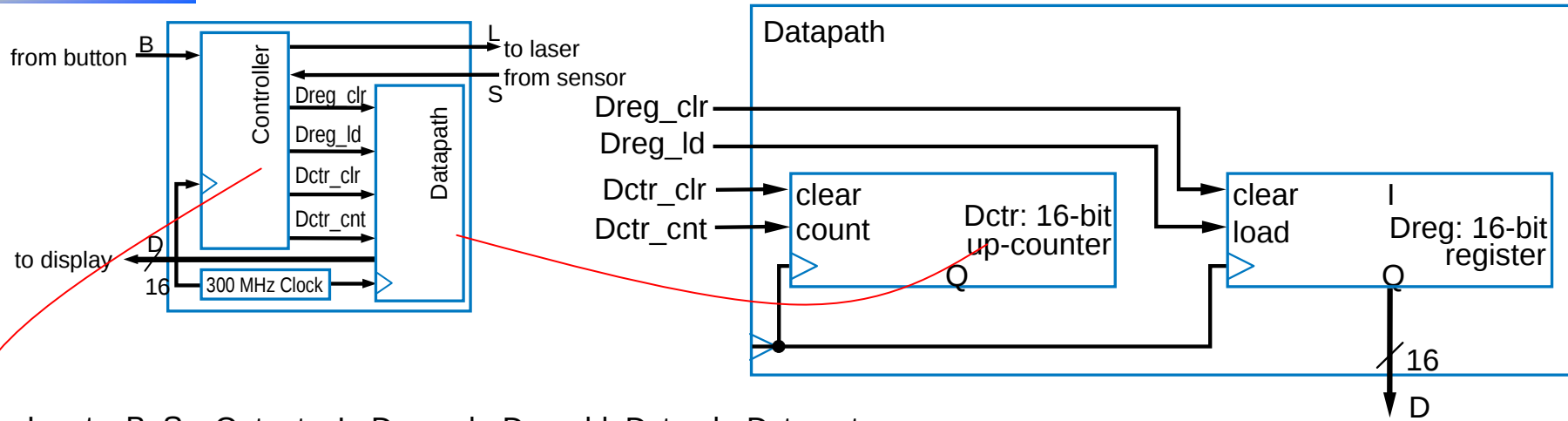


- Using shorthand of outputs not assigned implicitly assigned 0

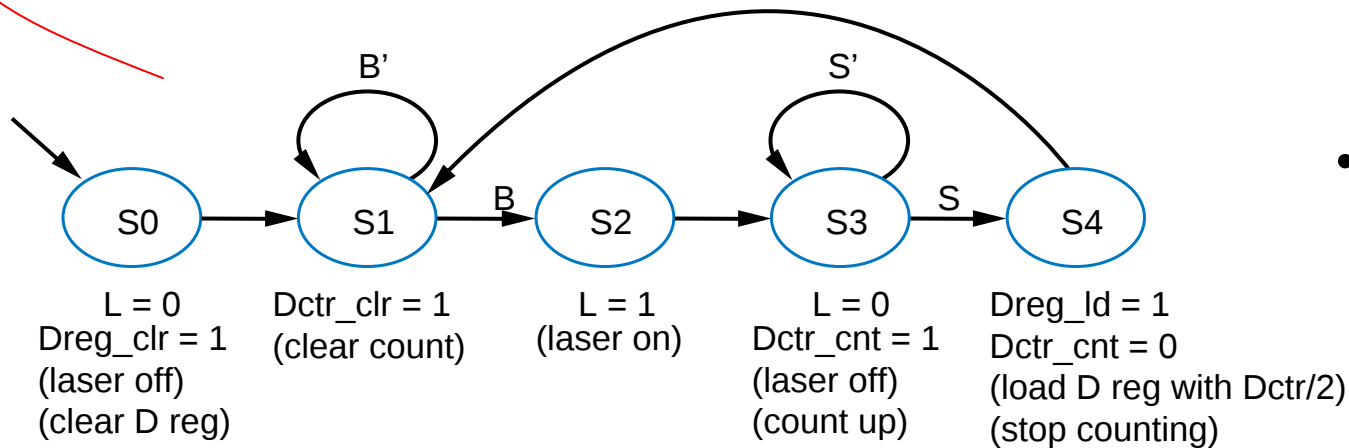
Inputs: B, S Outputs: L, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_cnt



Step 4



Inputs: B, S Outputs: L, Dreg_clr, Dreg_Id, Dctr_clr, Dctr_cnt

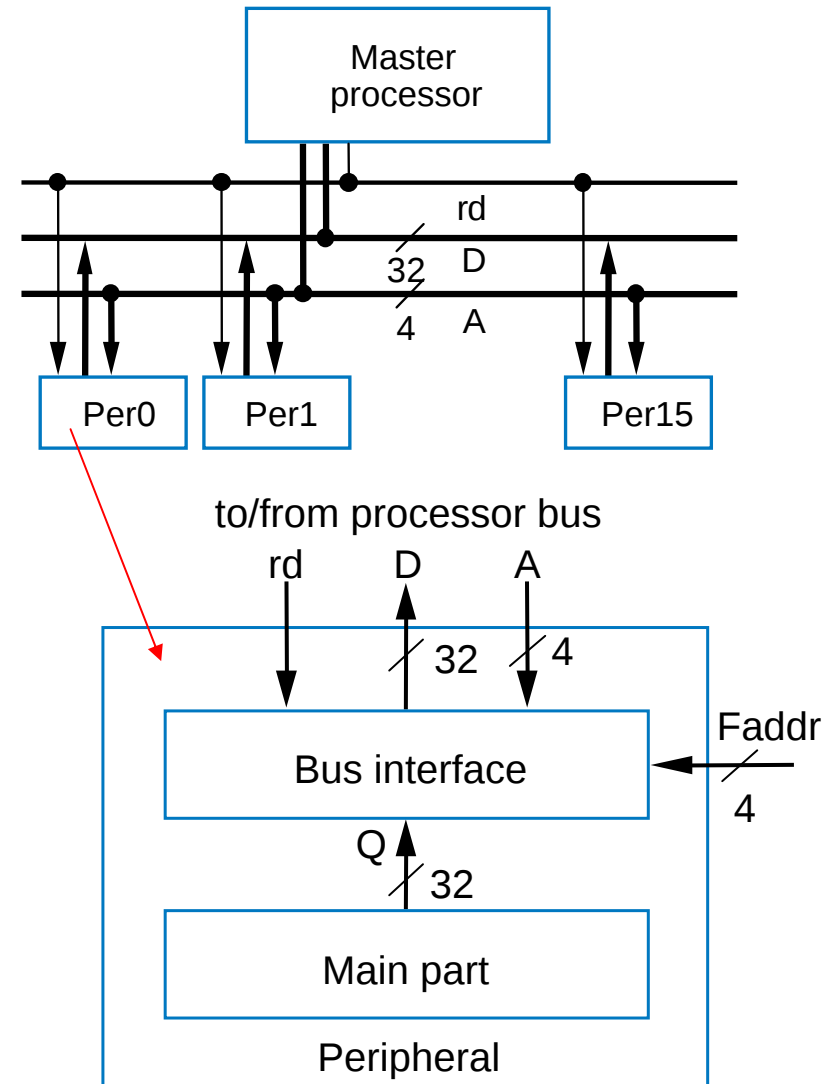


- Implement FSM as state register and logic (Ch3) to complete the design

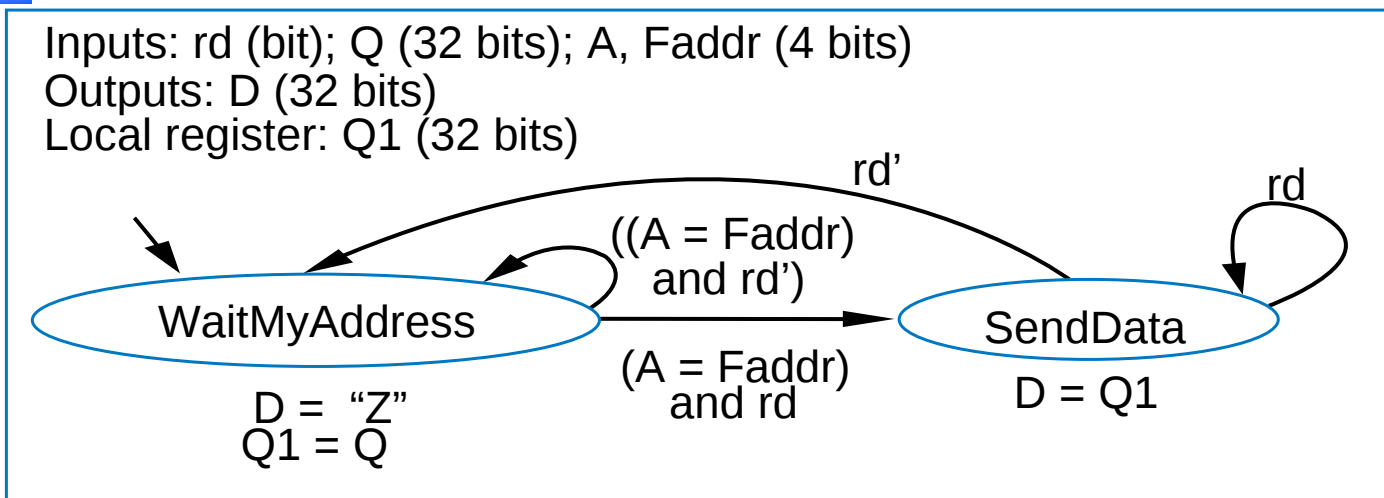


RTL Design Examples and Issues

- We'll use several more examples to illustrate RTL design
- Example: **Bus interface**
 - Master processor can read register from any peripheral
 - Each register has unique 4-bit address
 - Assume 1 register/periph.
 - Sets $rd=1$, $A=address$
 - Appropriate peripheral places register data on 32-bit D lines
 - Periph's address provided on $Faddr$ inputs (maybe from DIP switches, or another register)



RTL Example: Bus Interface

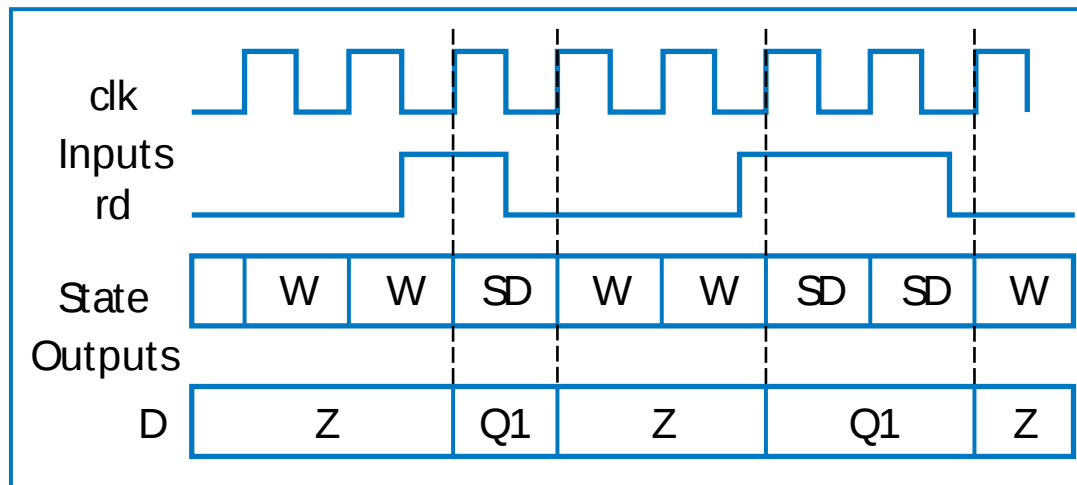
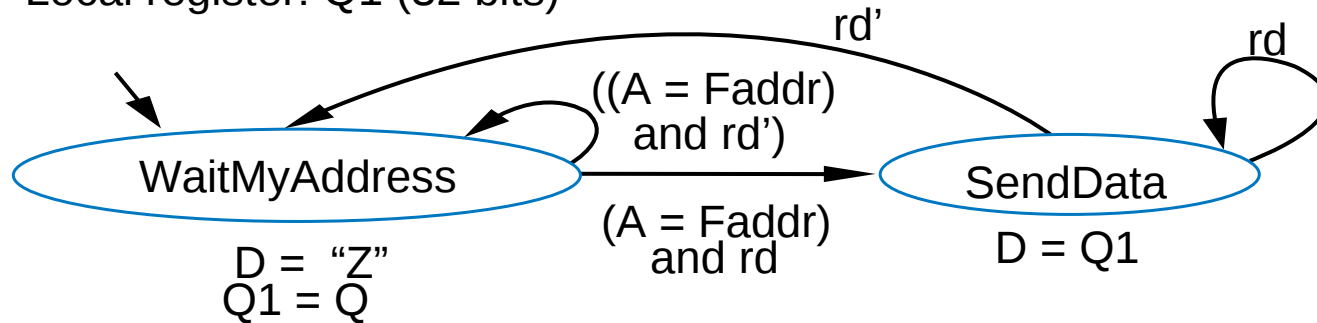


- Step 1: Create high-level state machine
 - State **WaitMyAddress**
 - Output “nothing” (“Z”) on D , store peripheral’s register value Q into local register $Q1$
 - Wait until this peripheral’s address is seen ($A = \text{Faddr}$) and $rd = 1$
 - State **SendData**
 - Output $Q1$ onto D , wait for $rd = 0$ (meaning main processor is done reading the D lines)



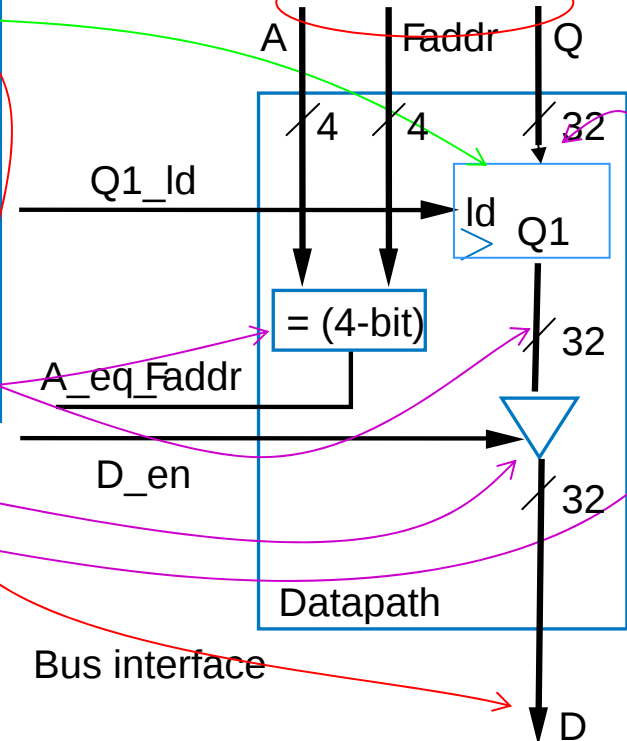
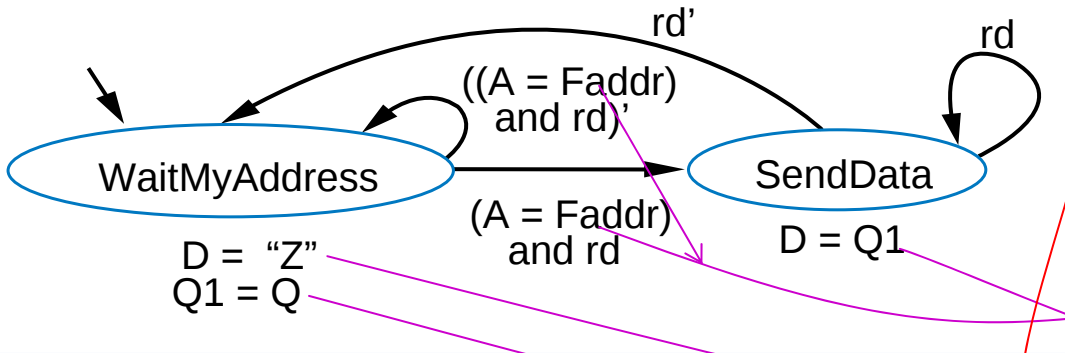
RTL Example: Bus Interface

Inputs: rd (bit); Q (32 bits); A, Faddr (4 bits)
 Outputs: D (32 bits)
 Local register: Q1 (32 bits)



RTL Example: Bus Interface

Inputs: rd (bit); Q (32 bits); A, Faddr (4 bits)
 Outputs: D (32 bits)
 Local register: Q1 (32 bits)



• Step 2: Create a datapath

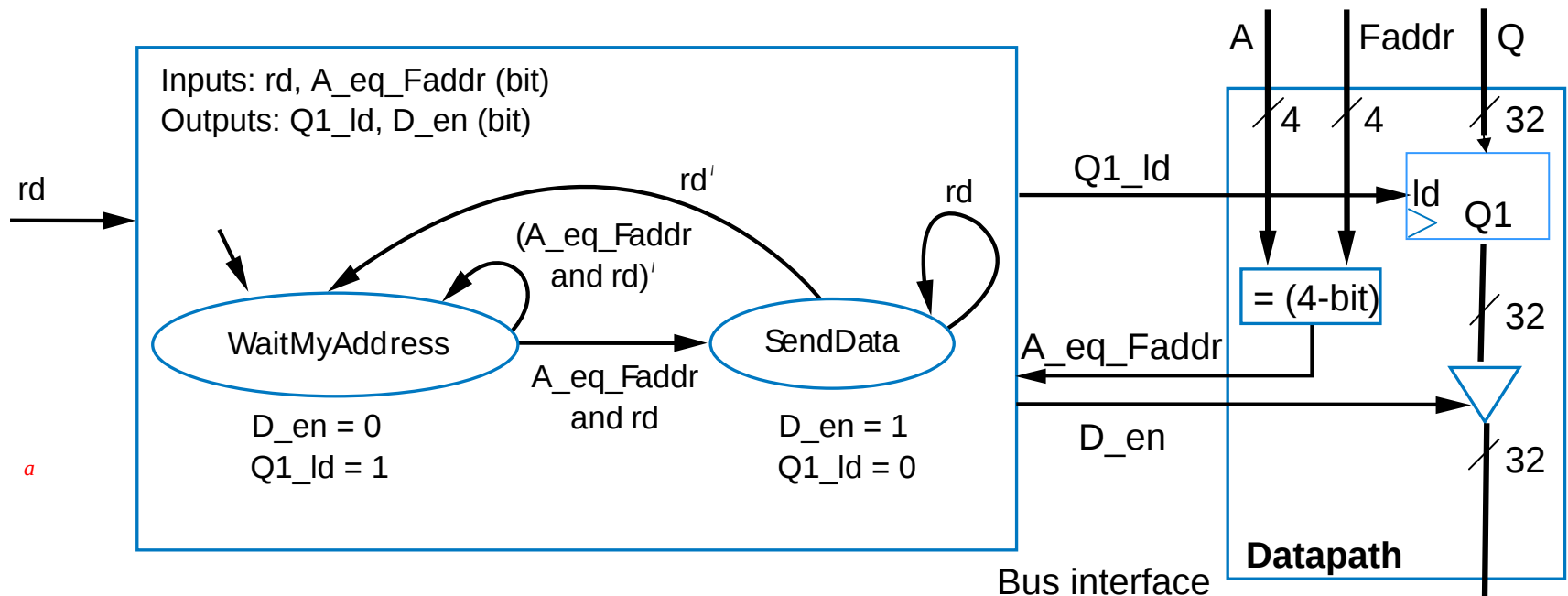
(a) Datapath inputs/outputs

(b) Instantiate declared registers

(c) Instantiate datapath components and connections



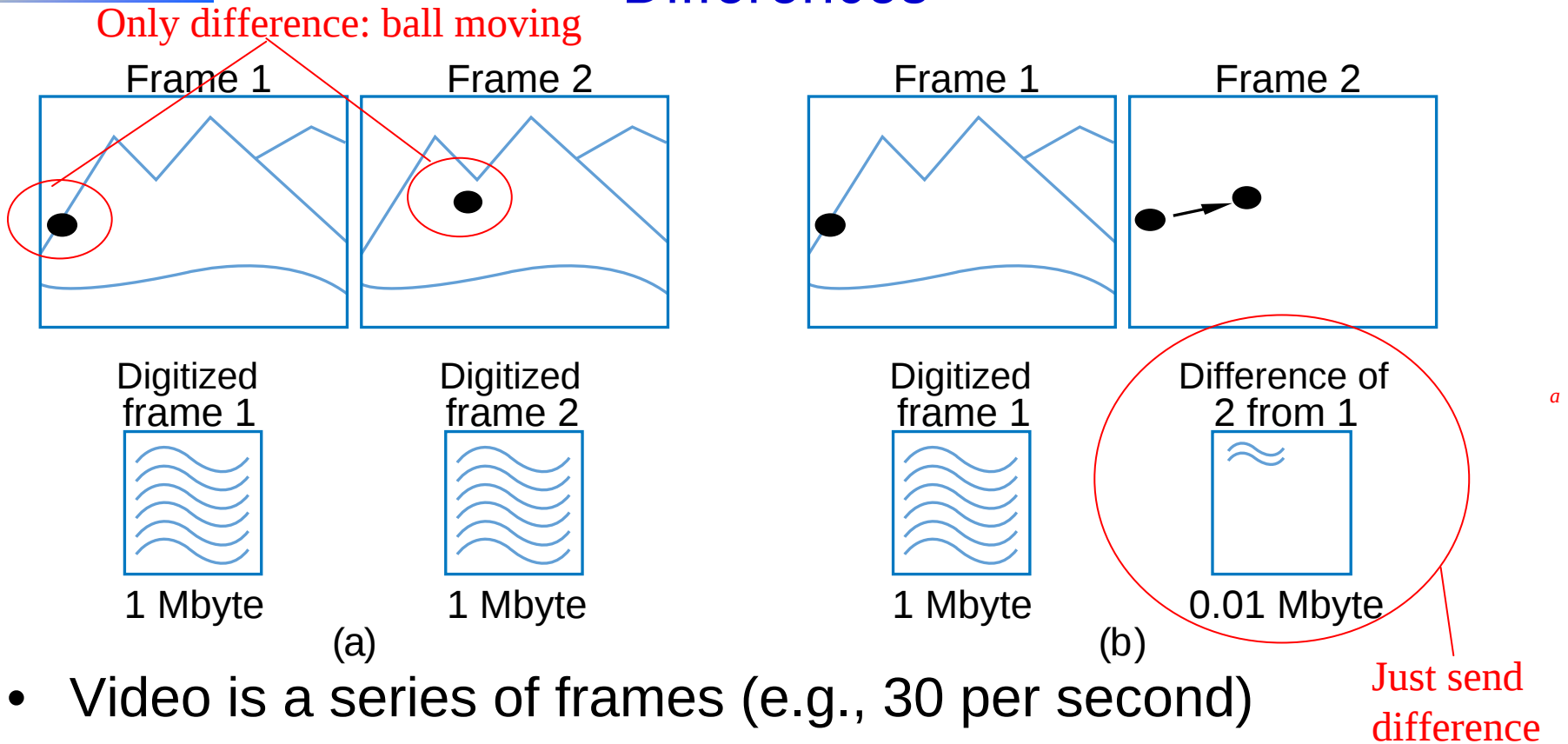
RTL Example: Bus Interface



- Step 3: Connect datapath to controller
- Step 4: Derive controller's FSM



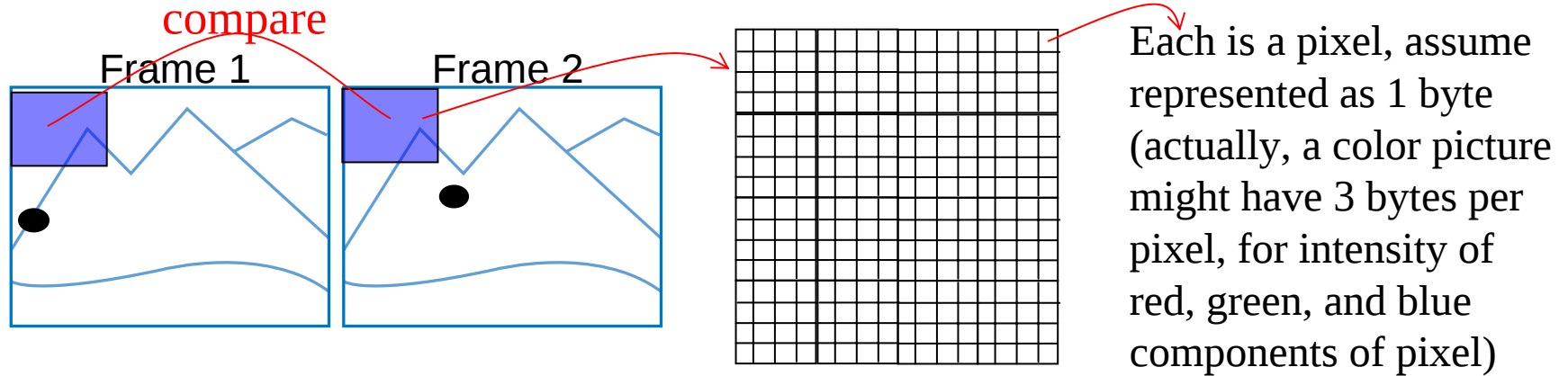
RTL Example: Video Compression – Sum of Absolute Differences



- Video is a series of frames (e.g., 30 per second)
- Most frames similar to previous frame
 - Compression idea: just send difference from previous frame



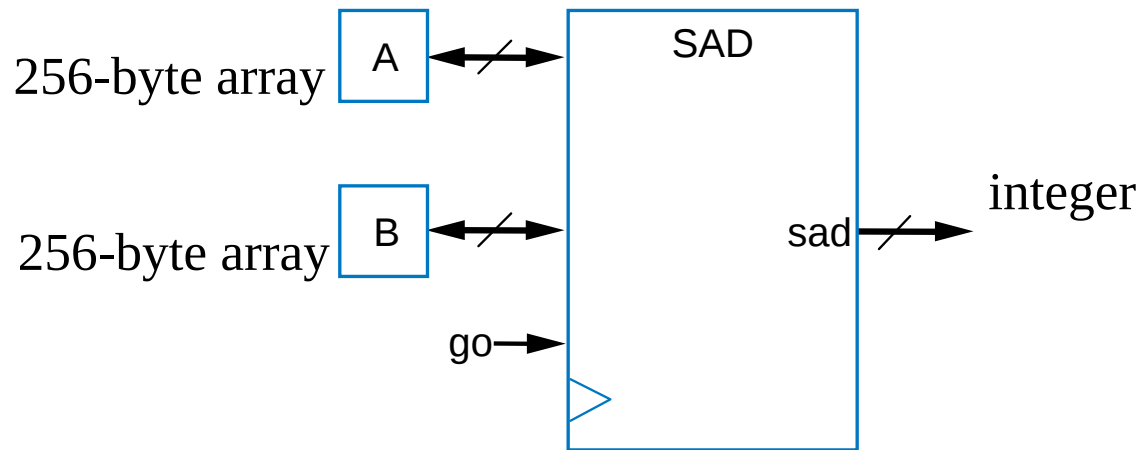
RTL Example: Video Compression – Sum of Absolute Differences



- Need to quickly determine whether two frames are similar enough to just send difference for second frame
 - Compare corresponding 16x16 “blocks”
 - Treat 16x16 block as 256-byte array
 - Compute the absolute value of the difference of each array item
 - Sum those differences – if above a threshold, send complete frame for second frame; if below, can use difference method (using another technique, not described)



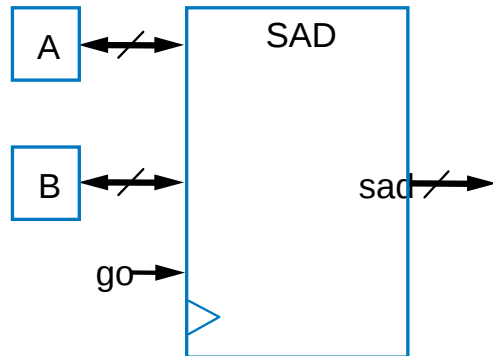
RTL Example: Video Compression – Sum of Absolute Differences



- Want fast sum-of-absolute-differences (SAD) component
 - When $go=1$, sums the differences of element pairs in arrays A and B , outputs that sum

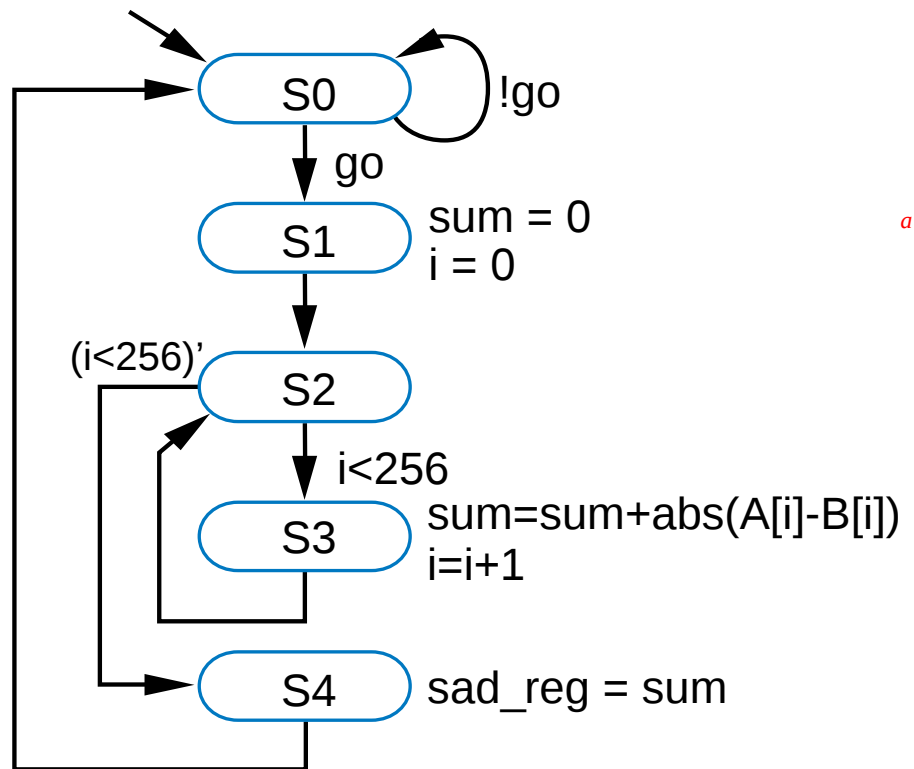


RTL Example: Video Compression – Sum of Absolute Differences



Inputs: A, B (256 byte memory); go (bit)
Outputs: sad (32 bits)
Local registers: sum, sad_reg (32 bits); i (9 bits)

- **S0**: wait for *go*
- **S1**: initialize *sum* and *index*
- **S2**: check if done ($i \geq 256$)
- **S3**: add difference to *sum*, increment index
- **S4**: done, write to output *sad_reg*

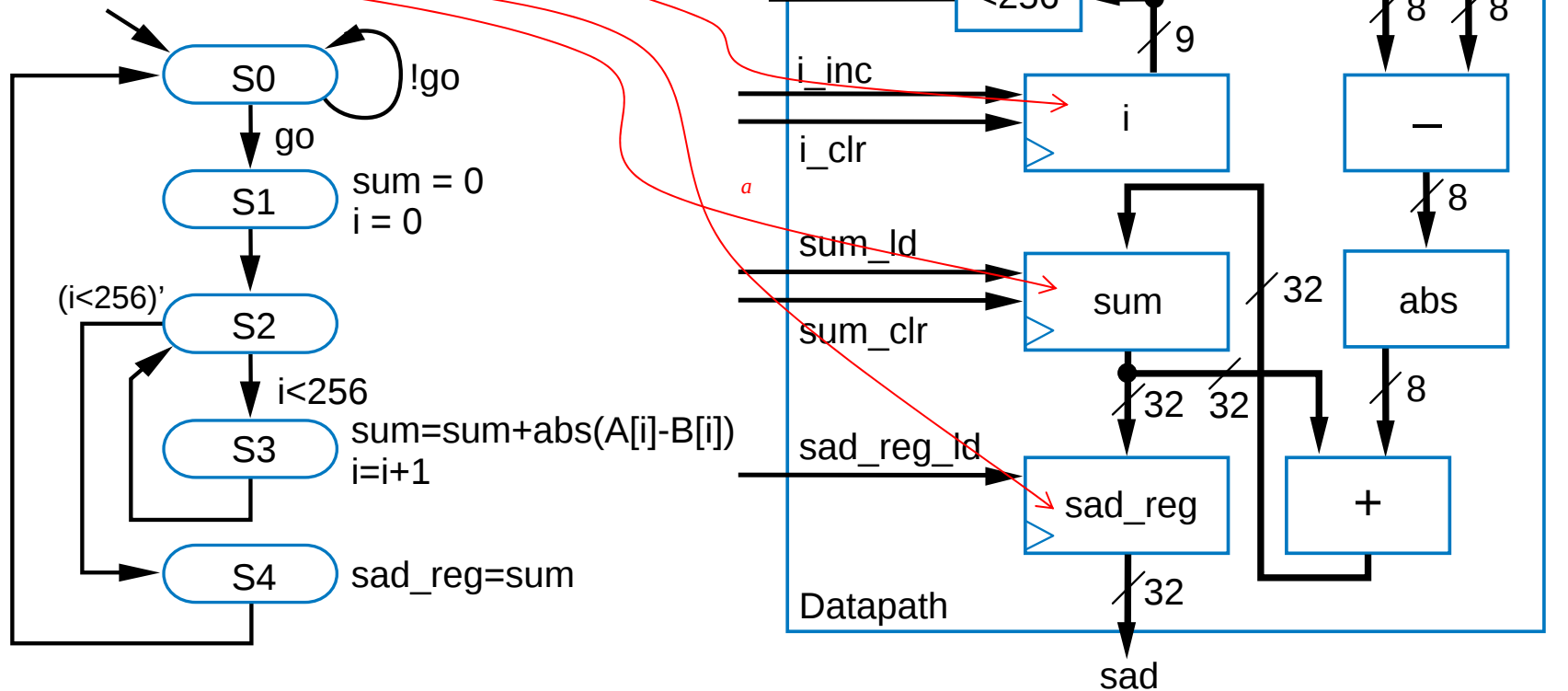


RTL Example: Video Compression – Sum of Absolute Differences

Inputs: A, B (256 byte memory); go (bit)

Outputs: sad (32 bits)

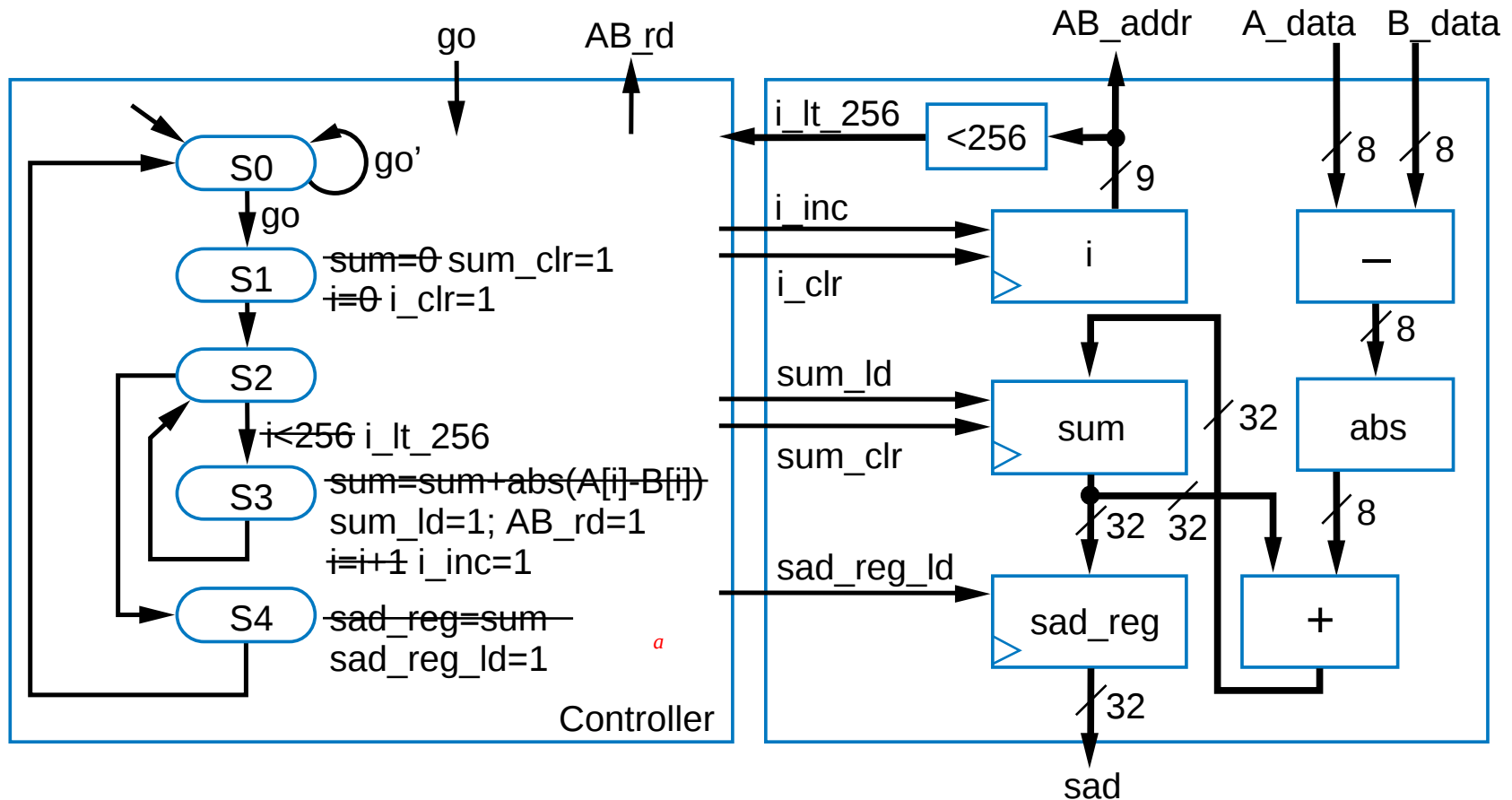
Local registers: sum, sad_reg (32 bits); i (9 bits)



- Step 2: Create datapath



RTL Example: Video Compression – Sum of Absolute Differences



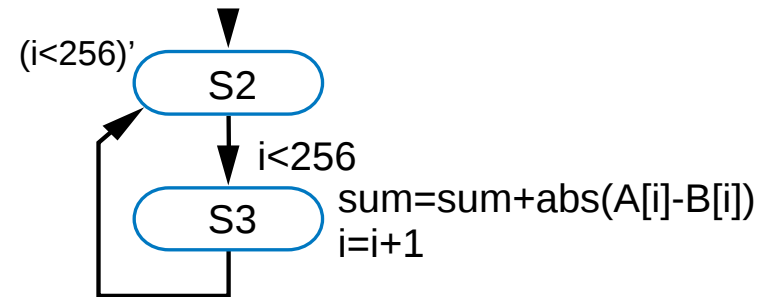
- Step 3: Connect to controller
- Step 4: Replace high-level state machine by FSM



RTL Example: Video Compression – Sum of Absolute Differences

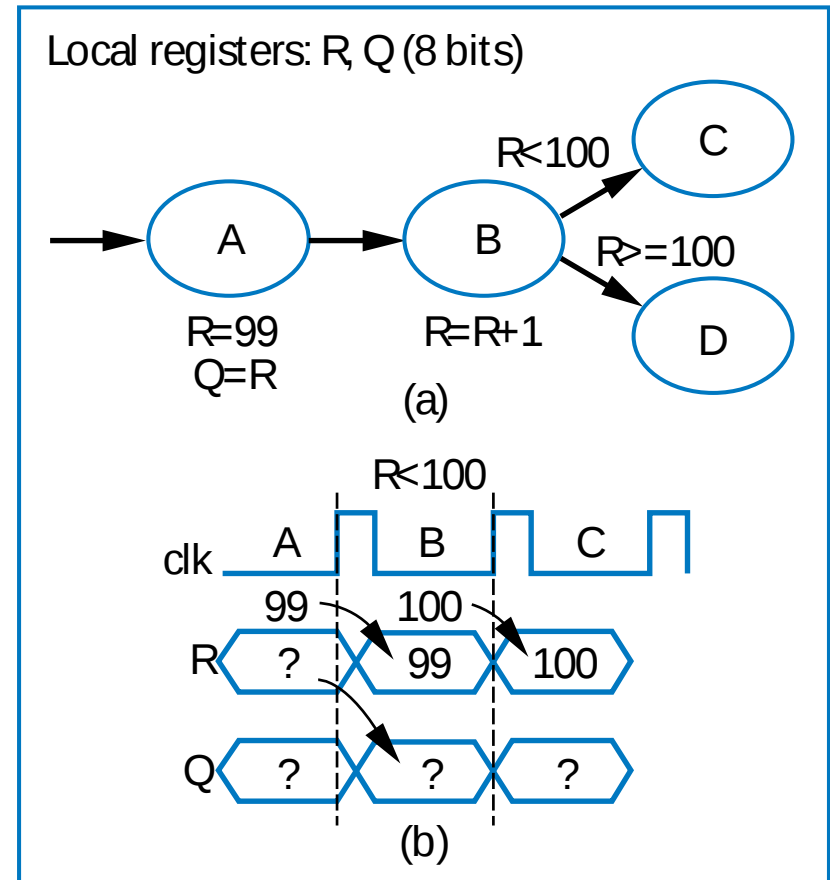
- Comparing software and custom circuit SAD

- Circuit: Two states (**S2** & **S3**) for each i , 256 i 's $\rightarrow$ 512 clock cycles
- Software: Loop (*for* $i = 1$ to 256), but for each i , must move memory to local registers, subtract, compute absolute value, add to sum, increment i – say about 6 cycles per array item $\rightarrow 256 \cdot 6 = 1536$ cycles
- Circuit is about 3 *times* (300%) faster
- Later, we'll see how to build SAD circuit that is even faster



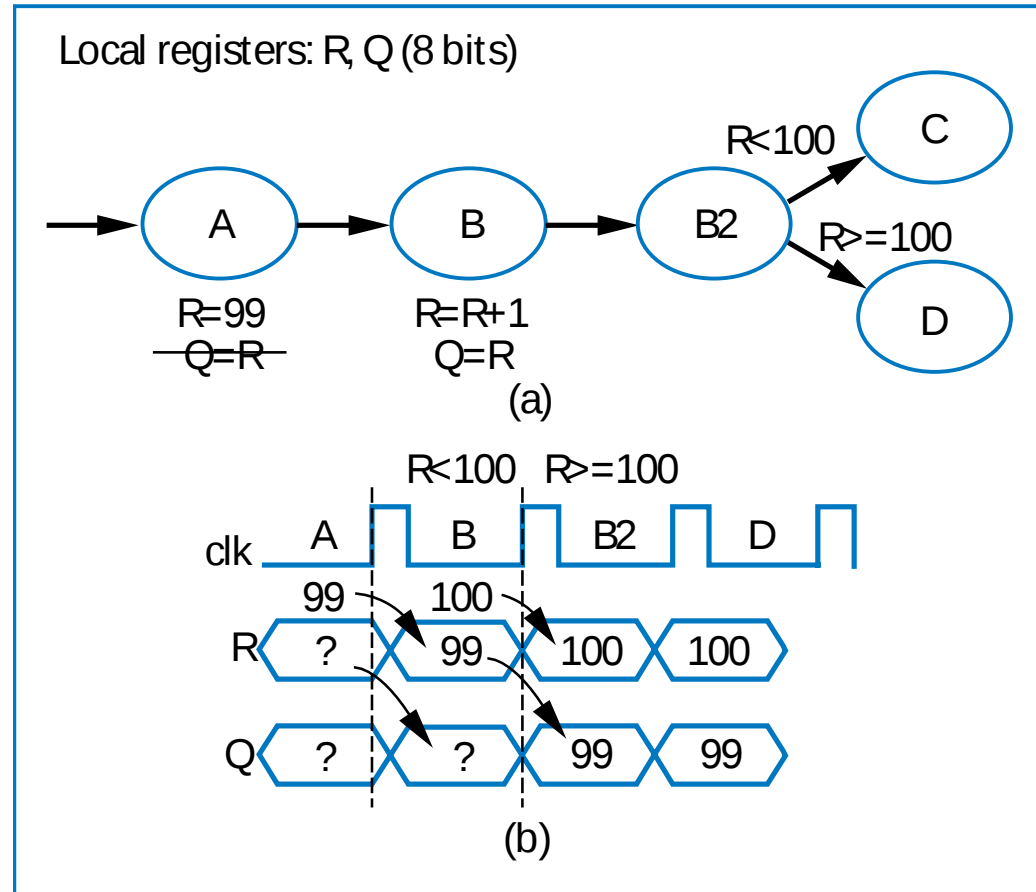
RTL Design Pitfalls and Good Practice

- Common pitfall: Assuming register is update in the state it's written
 - Final value of Q ?
 - Final state?
 - Answers may surprise you
 - Value of Q unknown
 - Final state is **C**, not **D**
 - Why?
 - State **A**: $R=99$ and $Q=R$ happen simultaneously
 - State **B**: R not updated with $R+1$ until next clock cycle, simultaneously with state register being updated



RTL Design Pitfalls and Good Practice

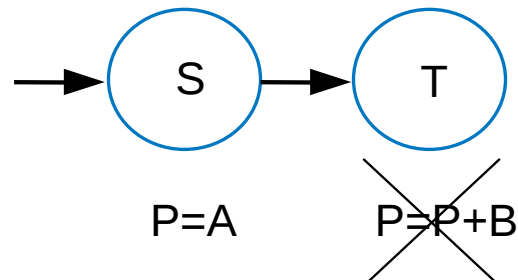
- Solutions
 - Read register in following state ($Q=R$)
 - Insert extra state so that conditions use updated value
 - Other solutions are possible, depends on the example



RTL Design Pitfalls and Good Practice

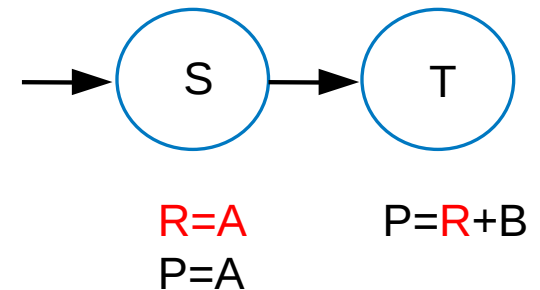
- Common pitfall:
Reading outputs
 - Outputs can only be written
 - Solution: Introduce additional register, which can be written and read

Inputs: A, B (8 bits)
Outputs: P (8 bits)



(a)

Inputs: A, B (8 bits)
Outputs: P (8 bits)
Local register: R (8 bits)

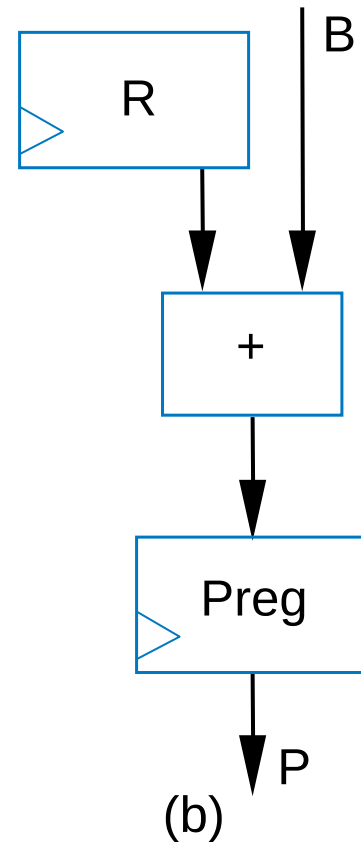
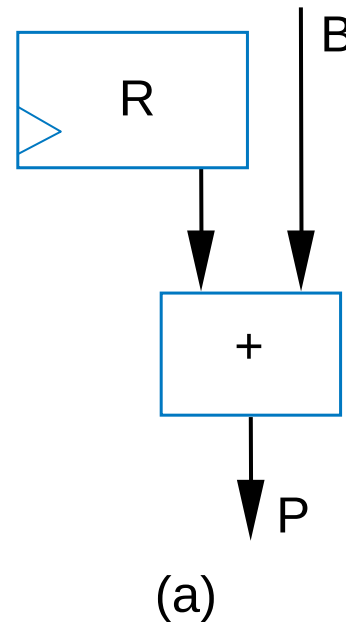


(b)



RTL Design Pitfalls and Good Practice

- Good practice: Register all data outputs
 - In fig (a), output P would show spurious values as addition computes
 - Furthermore, longest register-to-register path, which determines clock period, is not known until that output is connected to another component
 - In fig (b), spurious outputs reduced, and longest register-to-register path is clear



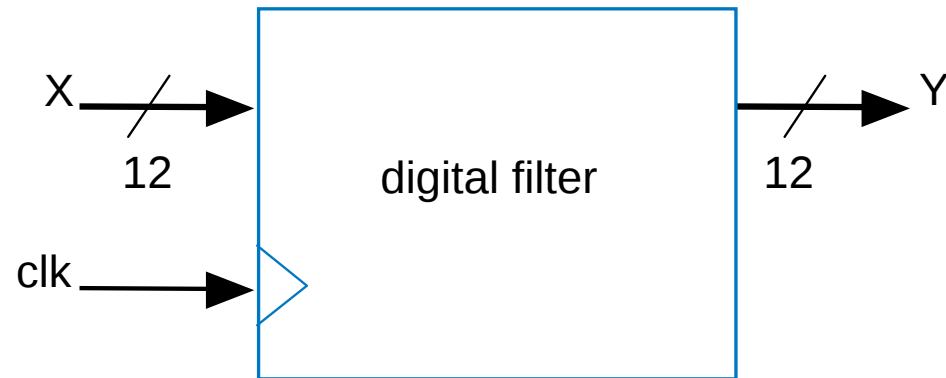
Control vs. Data Dominated RTL Design

- Designs often categorized as control-dominated or data-dominated
 - Control-dominated design – Controller contains most of the complexity
 - Data-dominated design – Datapath contains most of the complexity
 - General, descriptive terms – no hard rule that separates the two types of designs
 - Laser-based distance measurer – control dominated
 - Bus interface, SAD circuit – mix of control and data
 - Now let's do a data dominated design



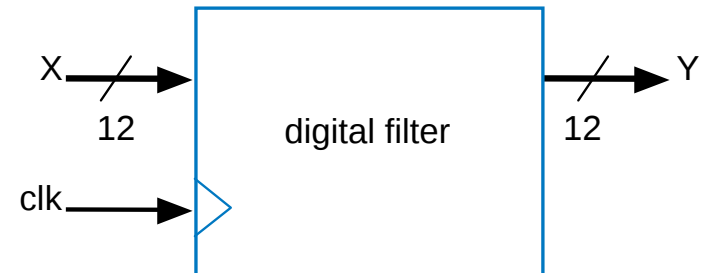
Data Dominated RTL Design Example: FIR Filter

- Filter concept
 - Suppose X is data from a temperature sensor, and particular input sequence is 180, 180, 181, 240, 180, 181 (one per clock cycle)
 - That 240 is probably wrong!
 - Could be electrical noise
 - Filter should remove such noise in its output Y
 - Simple filter: Output average of last N values
 - Small N : less filtering
 - Large N : more filtering, but less sharp output



Data Dominated RTL Design Example: FIR Filter

- FIR filter
 - “Finite Impulse Response”
 - Simply a configurable weighted sum of past input values
 - $y(t) = c0*x(t) + c1*x(t-1) + c2*x(t-2)$
 - Above known as “3 tap”
 - Tens of taps more common
 - Very general filter – User sets the constants ($c0$, $c1$, $c2$) to define specific filter
 - RTL design
 - Step 1: Create high-level state machine
 - But there really is none! Data dominated indeed.
 - Go straight to step 2

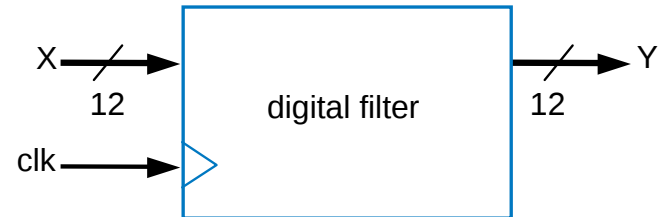


$$y(t) = c0*x(t) + c1*x(t-1) + c2*x(t-2)$$



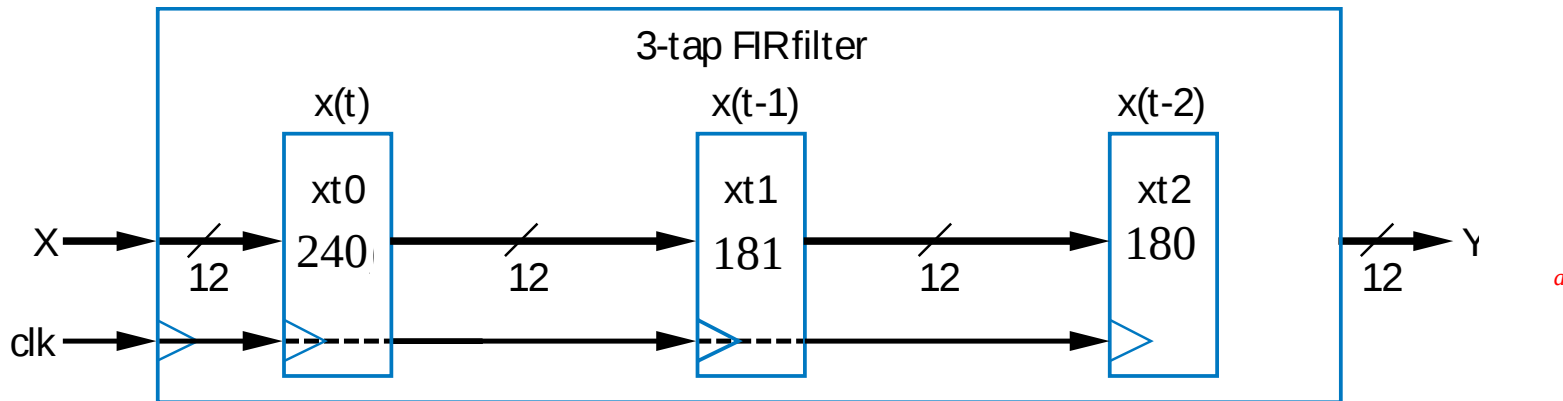
Data Dominated RTL Design Example: FIR Filter

- Step 2: Create datapath
 - Begin by creating chain of xt registers to hold past values of X



$$y(t) = c0*x(t) + c1*x(t-1) + c2*x(t-2)$$

Suppose sequence is: 180, 181, 240



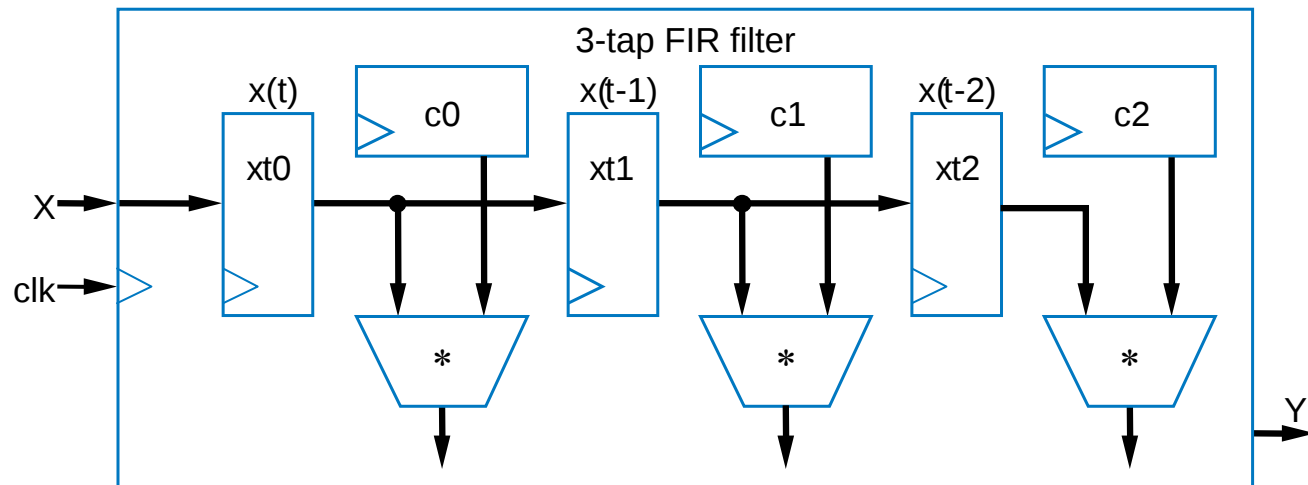
Data Dominated RTL Design Example: FIR Filter

- Step 2: Create datapath (cont.)

- Instantiate registers for c_0 , c_1 , c_2
- Instantiate multipliers to compute $c*x$ values



$$y(t) = c_0 * x(t) + c_1 * x(t-1) + c_2 * x(t-2)$$

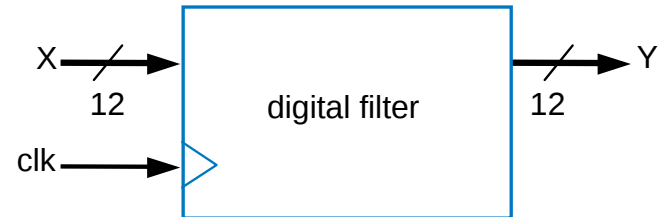


a

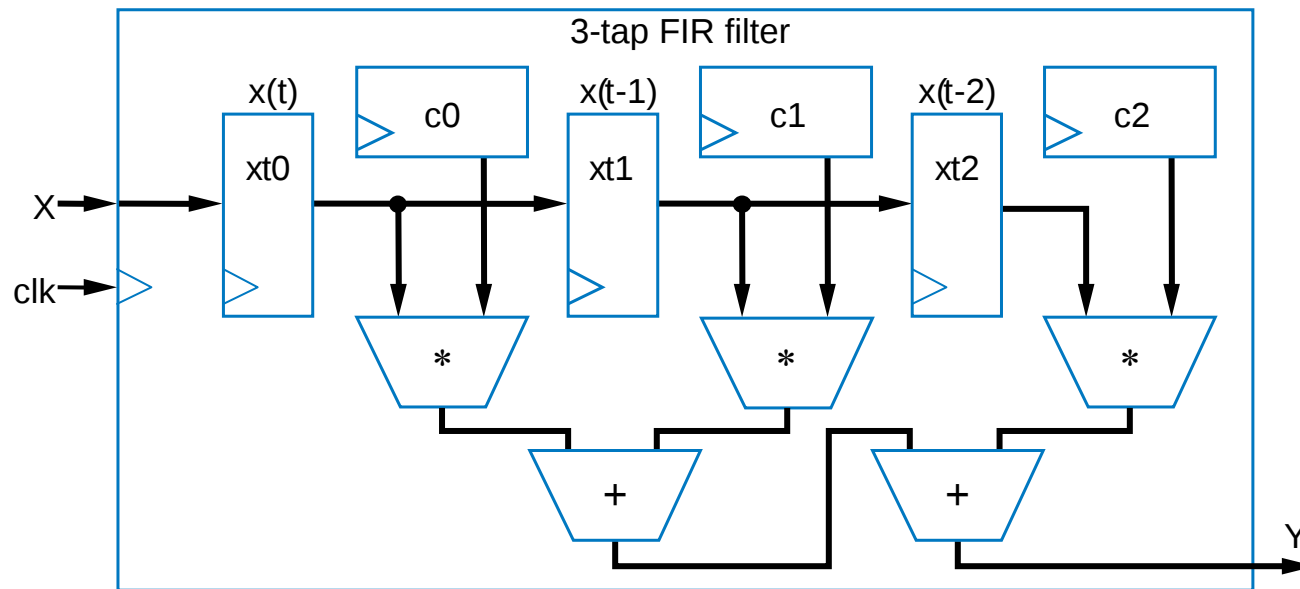


Data Dominated RTL Design Example: FIR Filter

- Step 2: Create datapath (cont.)
 - Instantiate adders

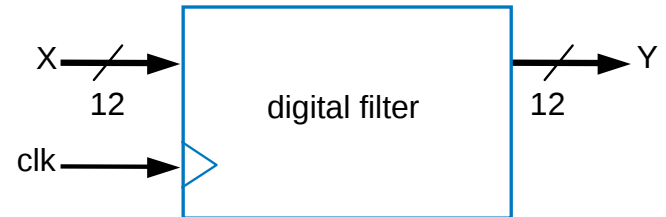


$$y(t) = c_0 * x(t) + c_1 * x(t-1) + c_2 * x(t-2)$$

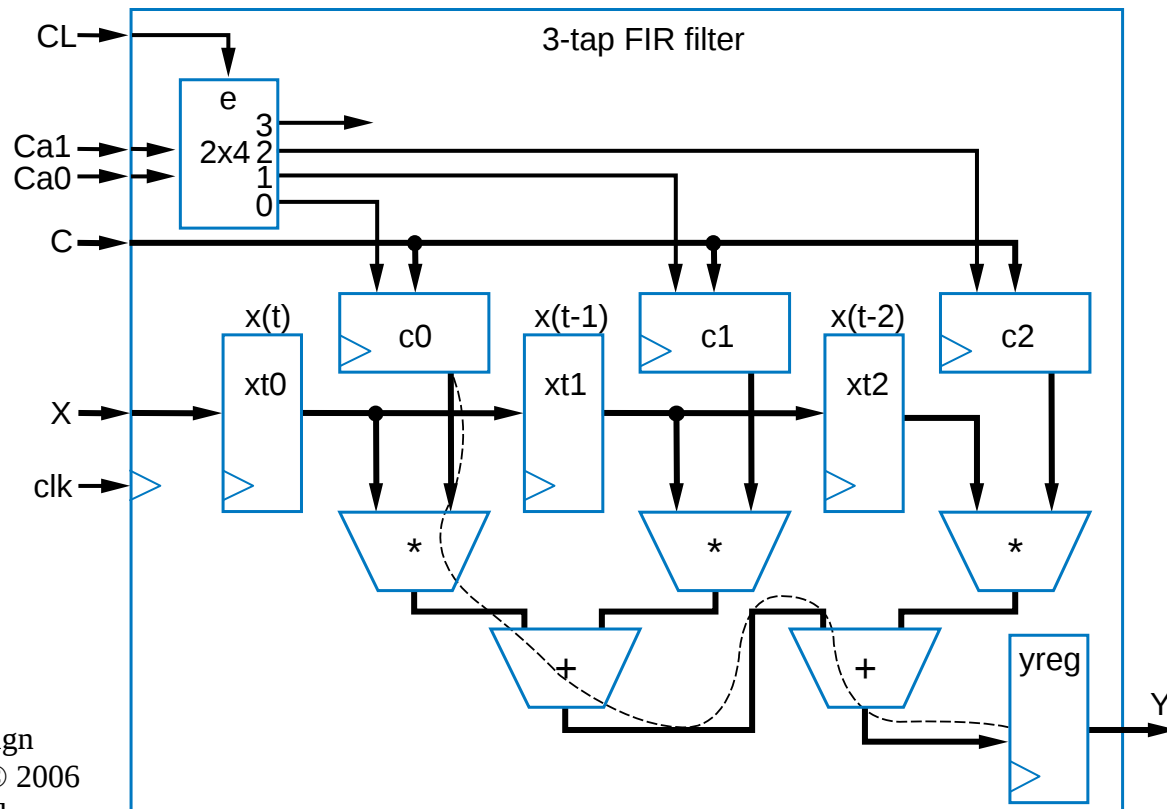


Data Dominated RTL Design Example: FIR Filter

- Step 2: Create datapath (cont.)
 - Add circuitry to allow loading of particular c register



$$y(t) = c0*x(t) + c1*x(t-1) + c2*x(t-2)$$



Data Dominated RTL Design Example: FIR Filter

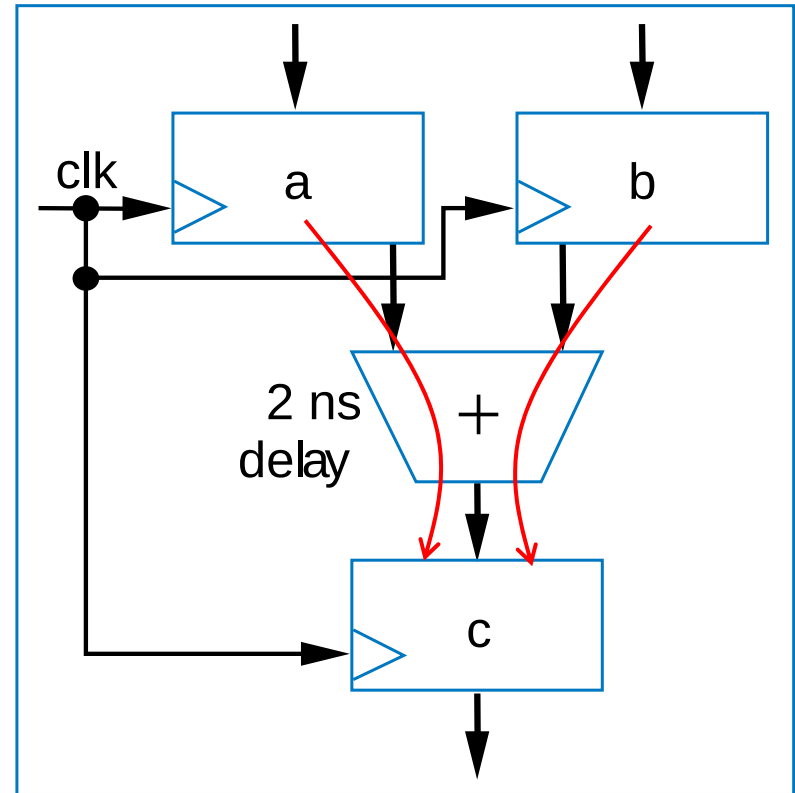
$$y(t) = c0*x(t) + c1*x(t-1) + c2*x(t-2)$$

- Step 3 & 4: Connect to controller, Create FSM
 - No controller needed
 - Extreme data-dominated example
 - (Example of an extreme control-dominated design – an FSM, with no datapath)
- Comparing the FIR circuit to a software implementation
 - Circuit
 - Assume adder has 2-gate delay, multiplier has 20-gate delay
 - Longest path goes through one multiplier and two adders
 - $20 + 2 + 2 = 24$ -gate delay
 - 100-tap filter, following design on previous slide, would have about a 34-gate delay: 1 multiplier and 7 adders on longest path
 - Software
 - 100-tap filter: 100 multiplications, 100 additions. Say 2 instructions per multiplication, 2 per addition. Say 10-gate delay per instruction.
 - $(100*2 + 100*2)*10 = 4000$ gate delays
 - Circuit is more than 100 times faster (10,000% faster). Wow.



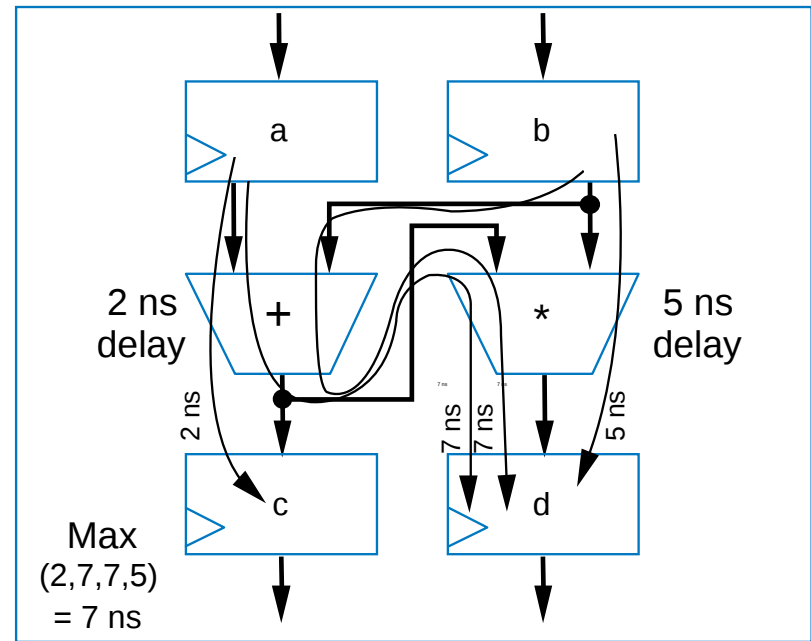
Determining Clock Frequency

- Designers of digital circuits often want fastest performance
 - Means want high clock frequency
- Frequency limited by **longest register-to-register delay**
 - Known as **critical path**
 - If clock is any faster, incorrect data may be stored into register
 - Longest path on right is 2 ns
 - Ignoring wire delays, and register setup and hold times, for simplicity



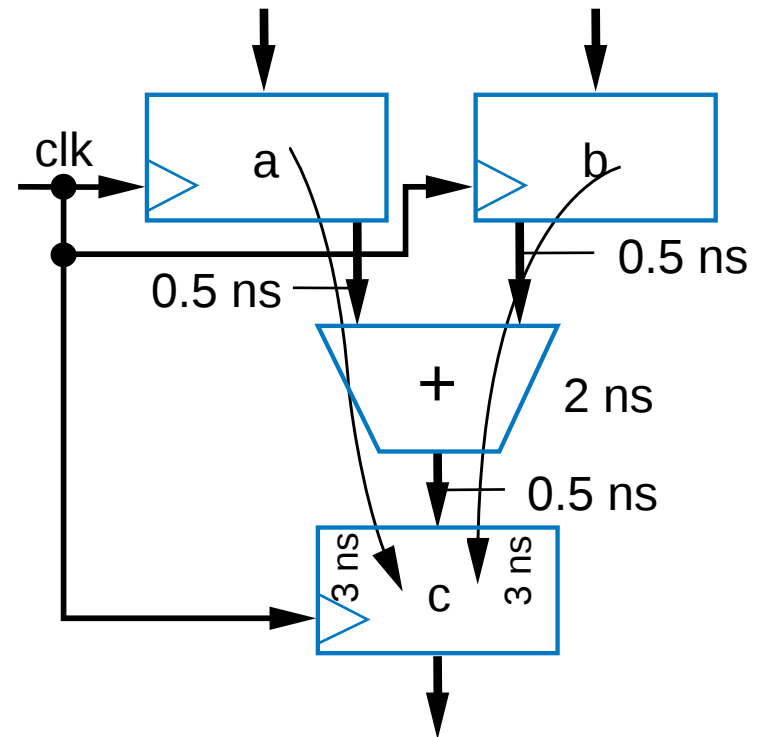
Critical Path

- Example shows four paths
 - a to c through +: 2 ns
 - a to d through + and *: 7 ns
 - b to d through + and *: 7 ns
 - b to d through *: 5 ns
- Longest path is thus 7 ns
- Fastest frequency
 - $1 / 7 \text{ ns} = 142 \text{ MHz}$



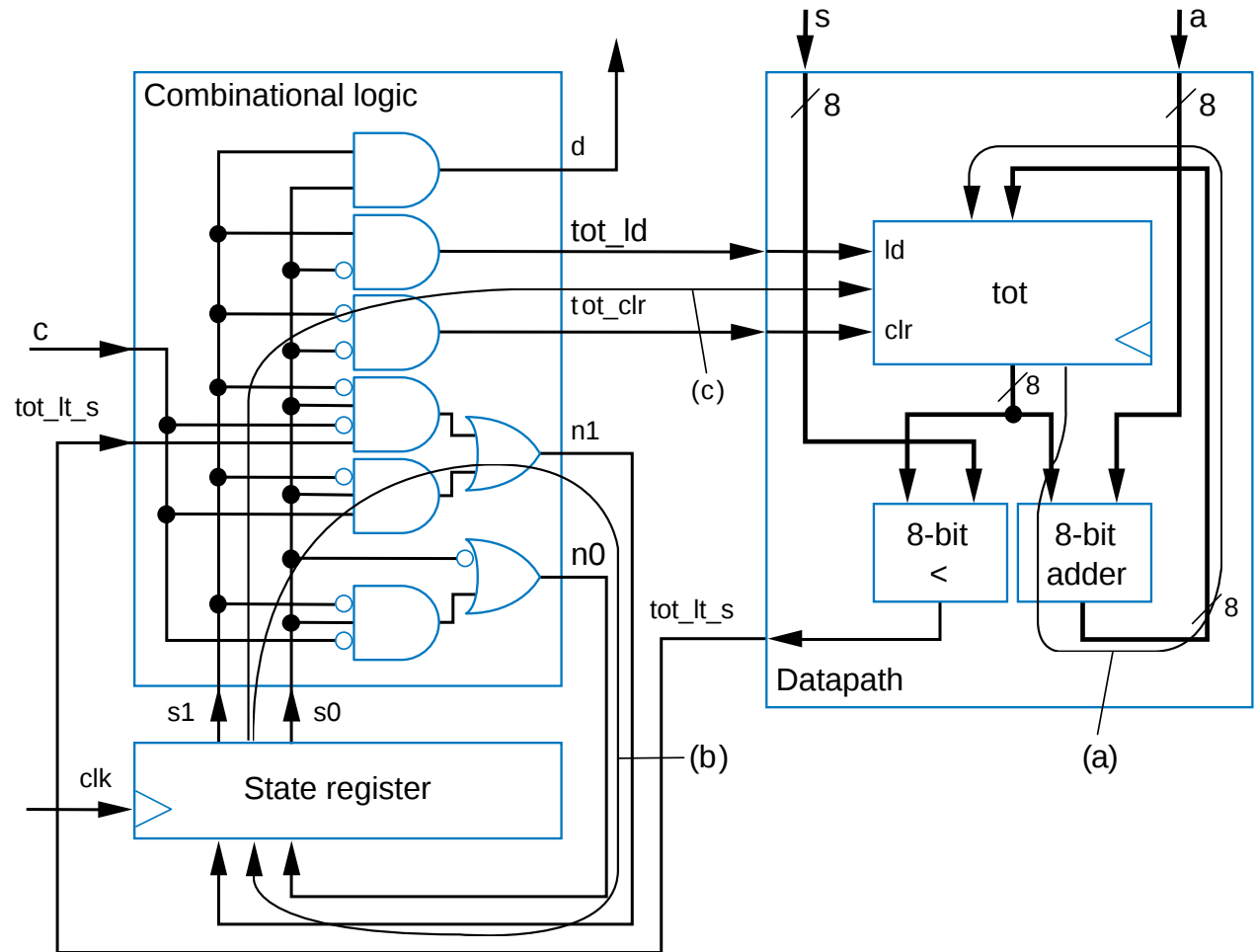
Critical Path Considering Wire Delays

- Real wires have delay too
 - Must include in critical path
- Example shows two paths
 - Each is $0.5 + 2 + 0.5 = 3$ ns
- Trend
 - 1980s/1990s: Wire delays were tiny compared to logic delays
 - But wire delays not shrinking as fast as logic delays
 - Wire delays may even be greater than logic delays!
- Must also consider register setup and hold times, also add to path
- Then add some time to the computed path, just to be safe
 - e.g., if path is 3 ns, say 4 ns instead

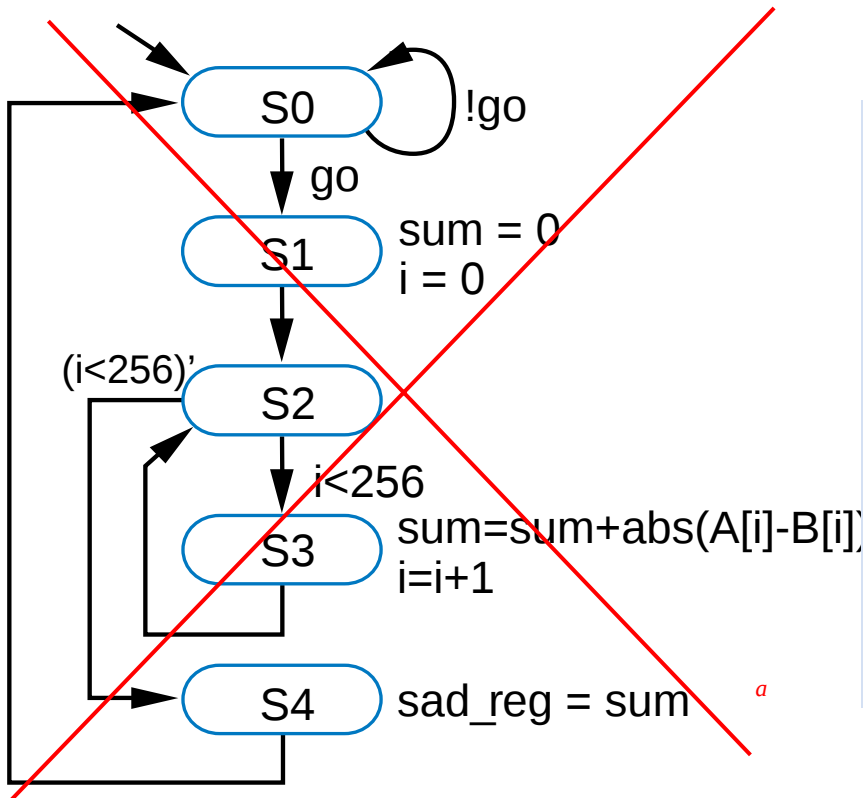


A Circuit May Have Numerous Paths

- Paths can exist
 - In the datapath
 - In the controller
 - Between the controller and datapath
 - May be hundreds or thousands of paths
- Timing analysis tools that evaluate all possible paths automatically very helpful



Behavioral Level Design: C to Gates



C code

```
int SAD (byte A[256], byte B[256]) // not quite C syntax
{
    uint sum; short uint I;
    sum = 0;
    i = 0;
    while (i < 256) {
        sum = sum + abs(A[i] - B[i]);
        i = i + 1;
    }
    return sum;
}
```

- Earlier sum-of-absolute-differences example
 - Started with high-level state machine
 - C code is an even better starting point -- easier to understand



Behavioral-Level Design: Start with C (or Similar Language)

- Replace first step of RTL design method by two steps
 - Capture in C, then convert C to high-level state machine
 - How convert from C to high-level state machine?

Step 1A: Capture in C

Step 1B: Convert to high-level state machine

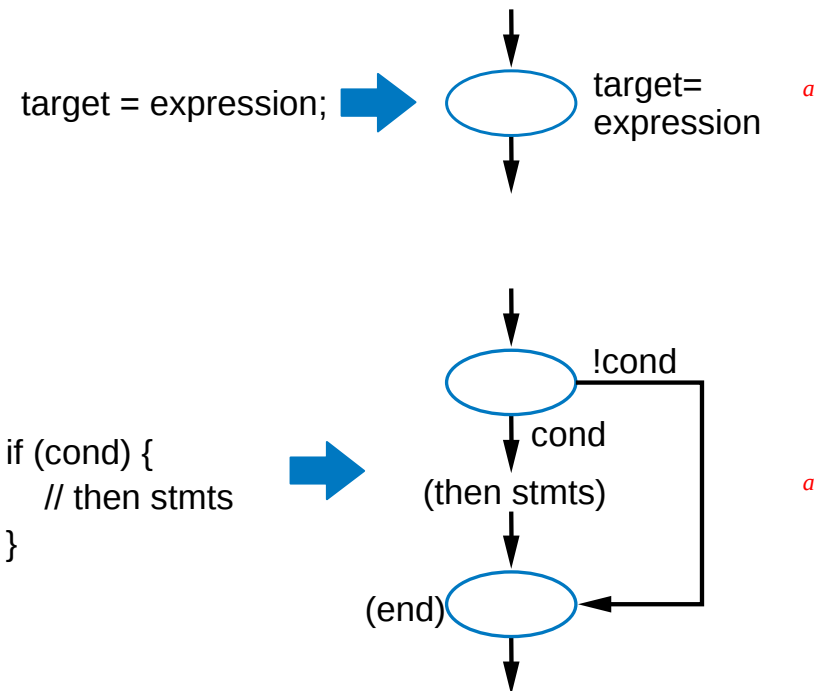
a

Step	Description
Step 1	Capture a high-level state machine Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is "high-level" because the transition conditions and the state actions are more than just Boolean operations on bit inputs and outputs.
Step 2	Create a datapath Create a datapath to carry out the data operations of the high-level state machine.
Step 3	Connect the datapath to a controller Connect the datapath to a controller block. Connect external Boolean inputs and outputs to the controller block.
Step 4	Derive the controller's FSM Convert the high-level state machine to a finite-state machine (FSM) for the controller, by replacing data operations with setting and reading of control signals to and from the datapath.



Converting from C to High-Level State Machine

- Convert each C construct to equivalent states and transitions
- *Assignment* statement
 - Becomes one state with assignment
- *If-then* statement
 - Becomes state with condition check, transitioning to “then” statements if condition true, otherwise to ending state
 - “then” statements would also be converted to states

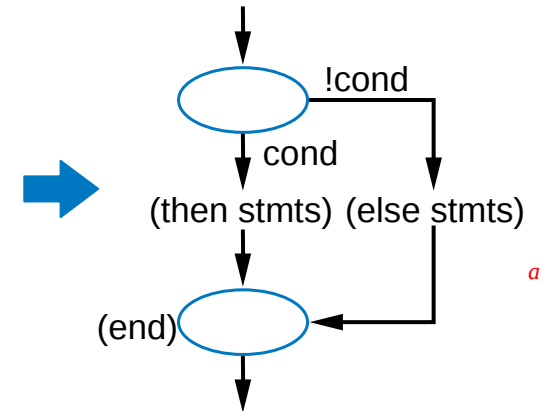


Converting from C to High-Level State Machine

- *If-then-else*

- Becomes state with condition check, transitioning to “then” statements if condition true, or to “else” statements if condition false

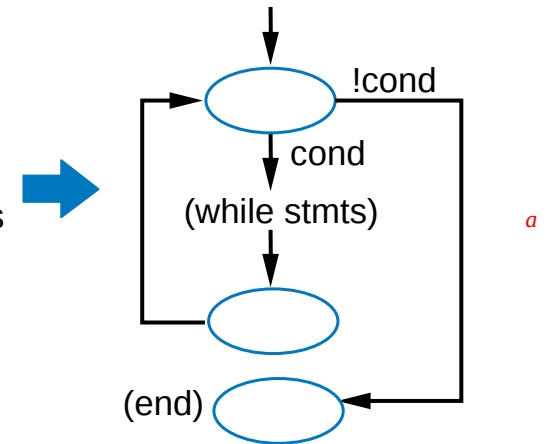
```
if (cond) {  
    // then stmts  
}  
else {  
    // else stmts  
}
```



- *While loop* statement

- Becomes state with condition check, transitioning to while loop's statements if true, then transitioning back to condition check

```
while (cond) {  
    // while stmts  
}
```



Simple Example of Converting from C to High-Level State Machine

Inputs: uint X, Y
Outputs: uint Max

```
if (X > Y) {
```

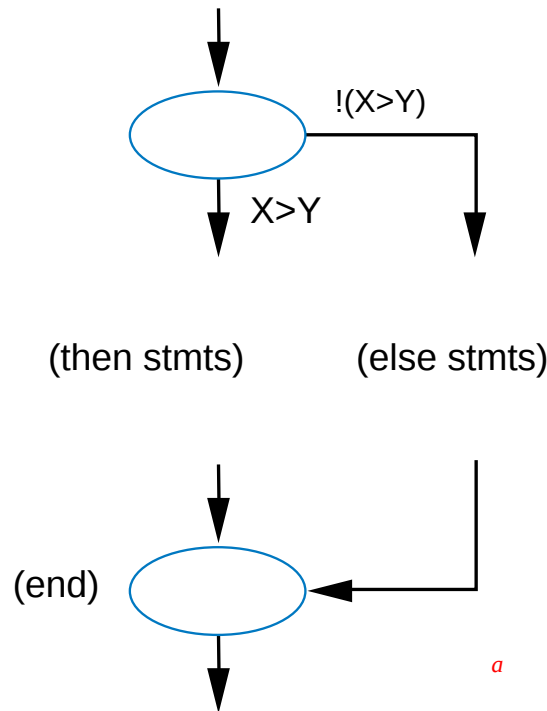
```
    Max = X;
```

```
}
```

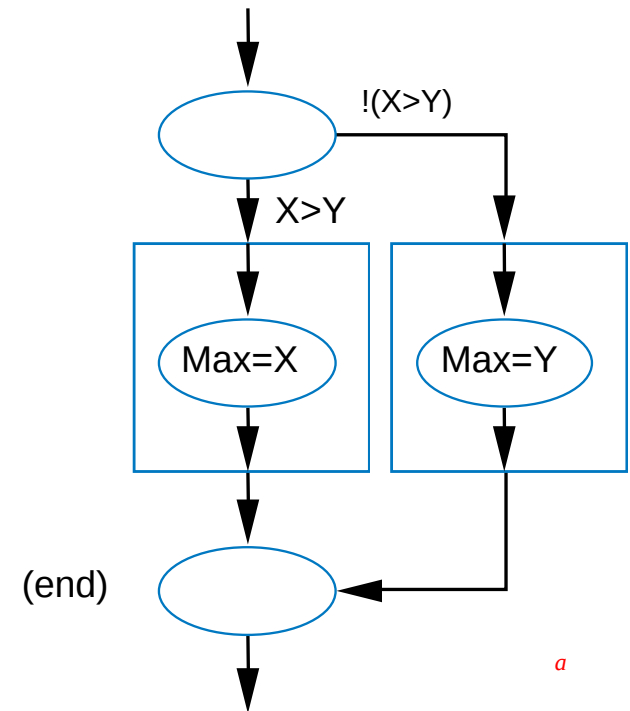
```
else {
```

```
    Max = Y;
```

```
}
```



(a)



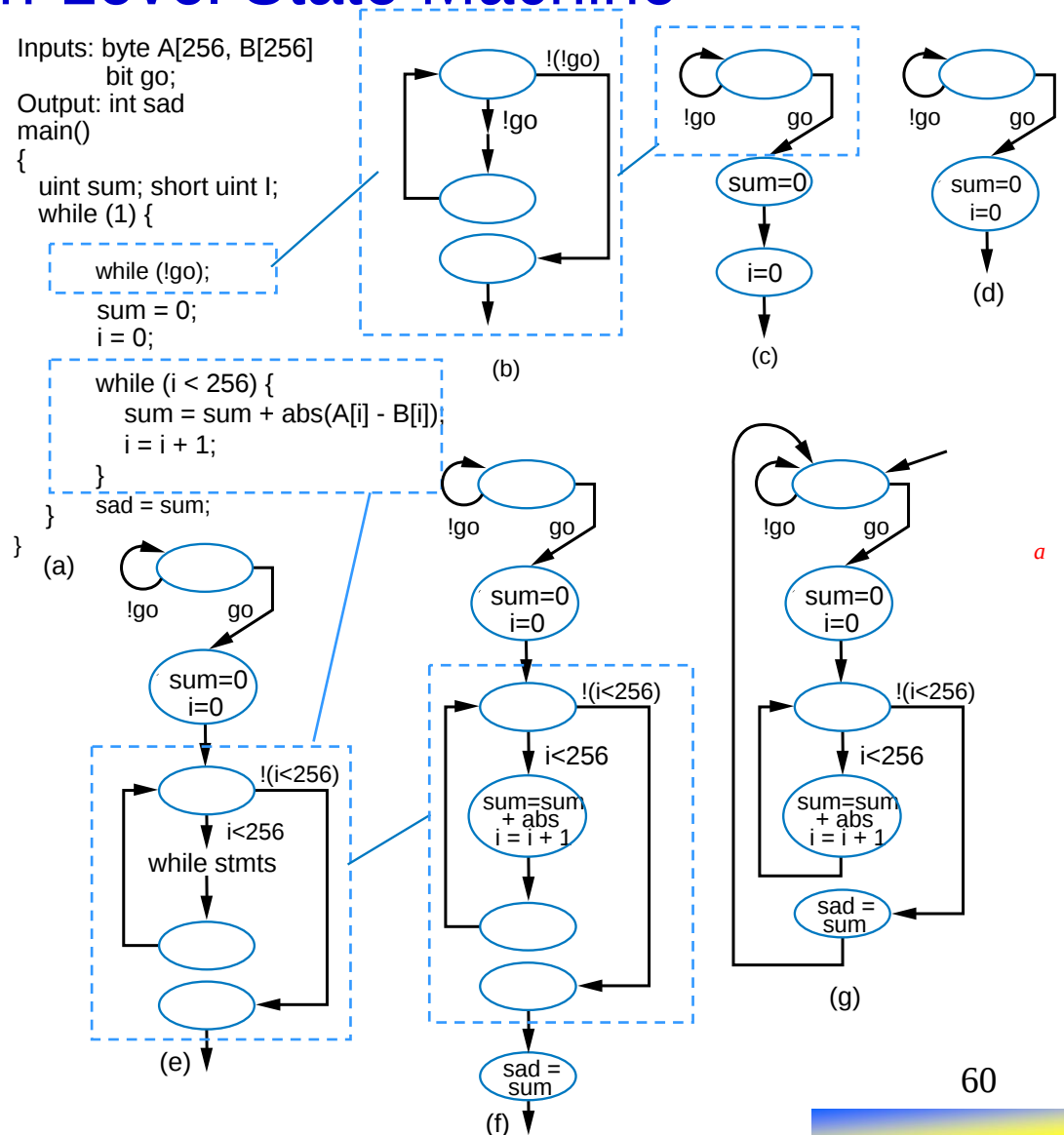
(c)

- Simple example: Computing the maximum of two numbers
 - Convert if-then-else statement to states (b)
 - Then convert assignment statements to states (c)



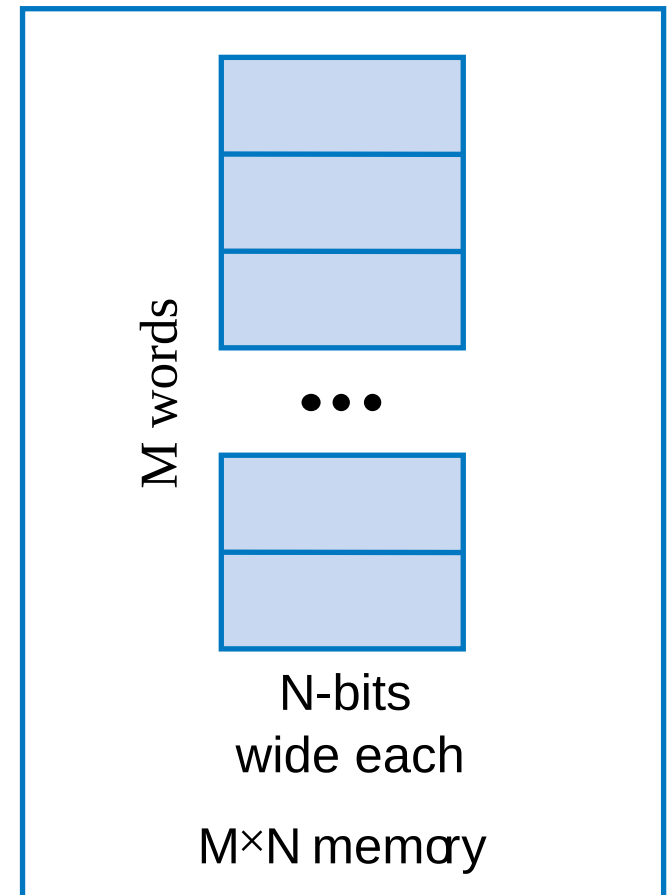
Example: Converting Sum-of-Absolute-Differences C code to High-Level State Machine

- Convert each construct to states
 - Simplify when possible, e.g., merge states
- From high-level state machine, follow RTL design method to create circuit
- Thus, can convert C to gates using straightforward automatable process
 - Not all C constructs can be efficiently converted
 - Use C subset if intended for circuit
 - Can use languages other than C, of course



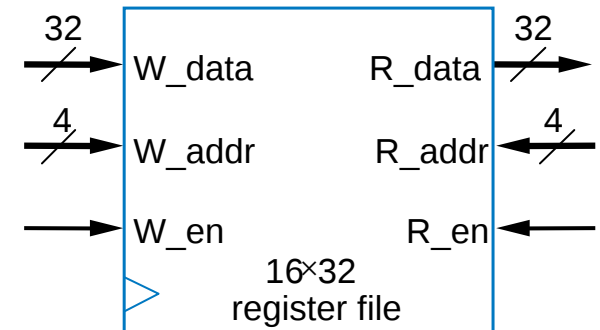
Memory Components

- Register-transfer level design instantiates datapath components to create datapath, controlled by a controller
 - A few more components are often used outside the controller and datapath
- ***MxN memory***
 - M words, N bits wide each
- Several varieties of memory, which we now introduce

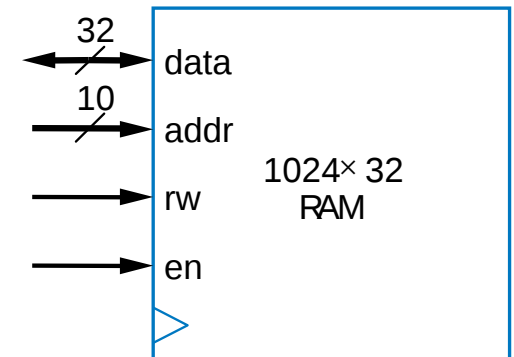


Random Access Memory (RAM)

- RAM – Readable and writable memory
 - “Random access memory”
 - Strange name – Created several decades ago to contrast with sequentially-accessed storage like tape drives
 - Logically same as register file – Memory with address inputs, data inputs/outputs, and control
 - RAM usually just one port; register file usually two or more
 - RAM vs. register file
 - RAM typically larger than *roughly* 512 or 1024 words
 - RAM typically stores bits using a bit storage approach that is more efficient than a flip flop
 - RAM typically implemented on a chip in a square rather than rectangular shape – keeps longest wires (hence delay) short



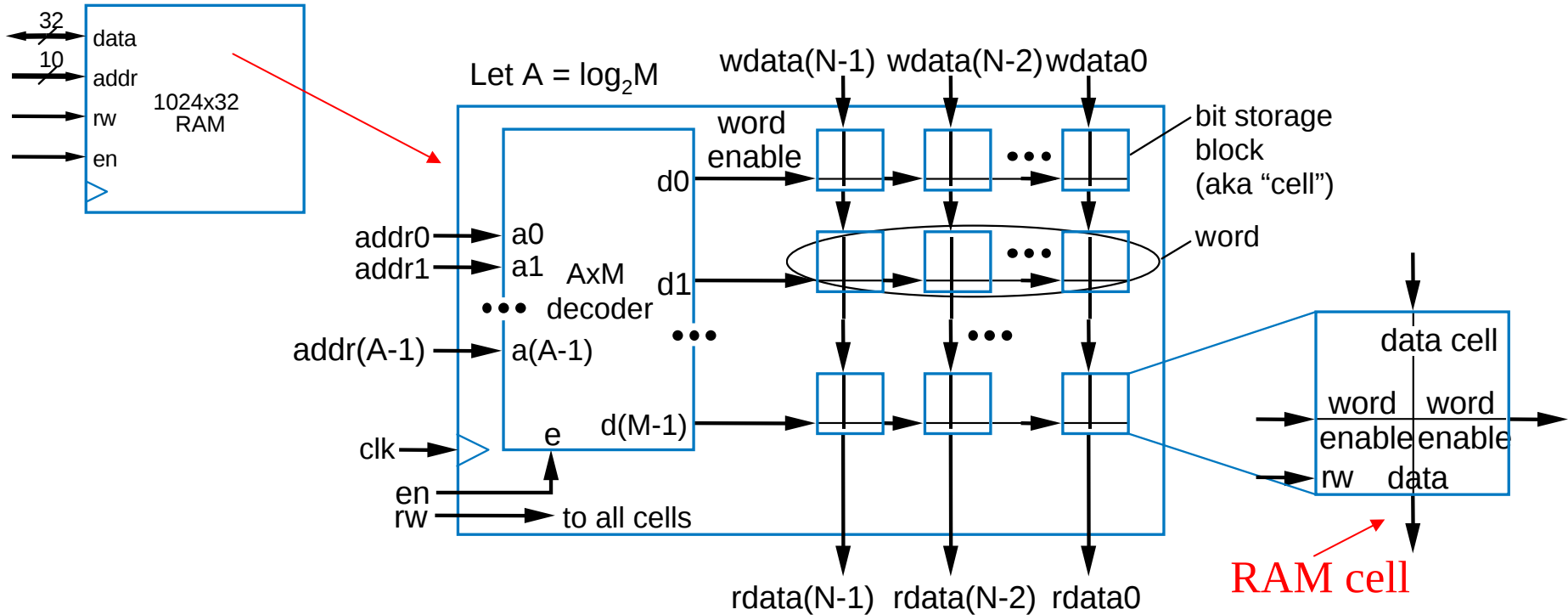
Register file from Chpt. 4



RAM block symbol



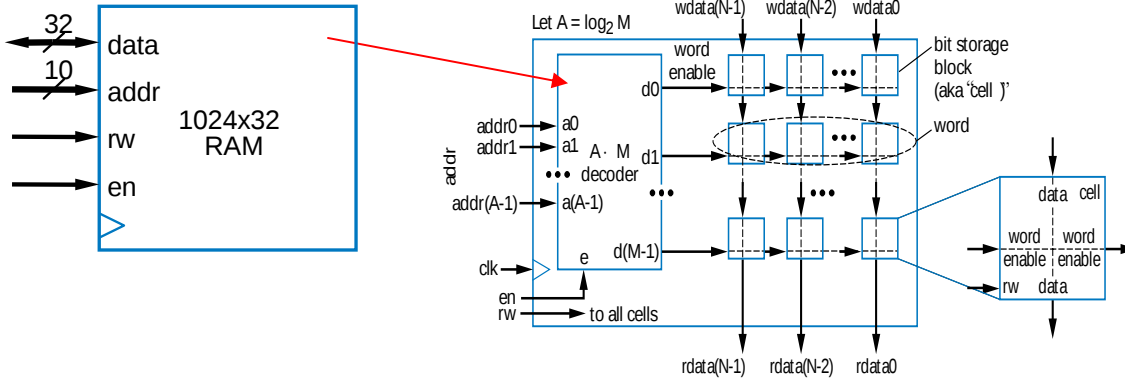
RAM Internal Structure



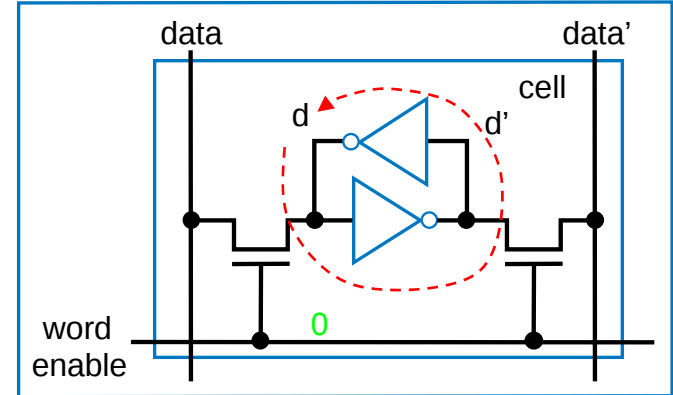
- Similar internal structure as register file
 - Decoder enables appropriate word based on address inputs
 - rw controls whether cell is written or read
 - Let's see what's inside each RAM cell



Static RAM (SRAM)

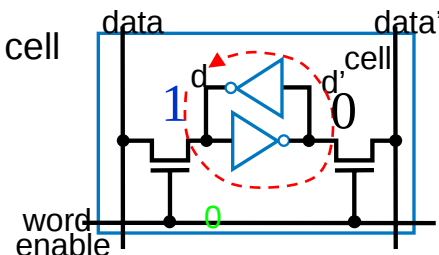
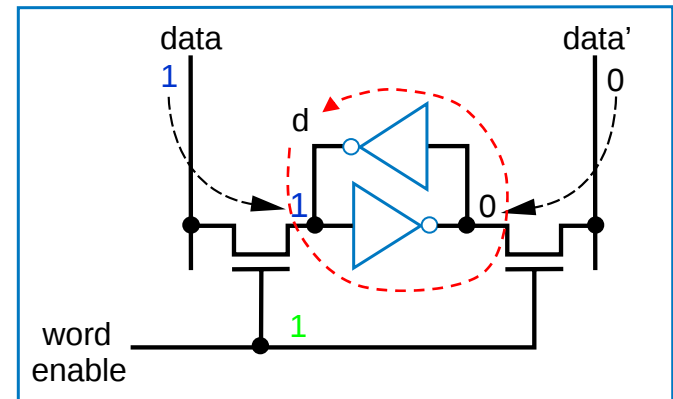


SRAM cell

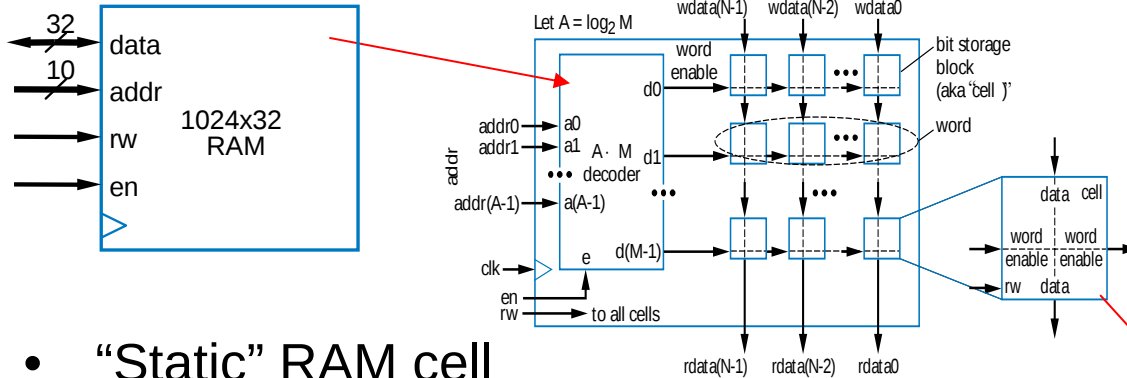


- “Static” RAM cell
 - 6 transistors (recall inverter is 2 transistors)
 - Writing this cell
 - *word enable* input comes from decoder
 - When 0, value *d* loops around inverters
 - That loop is where a bit stays stored
 - When 1, the *data* bit value enters the loop
 - *data* is the bit to be stored in this cell
 - *data'* enters on other side
 - Example shows a “1” being written into cell

SRAM cell

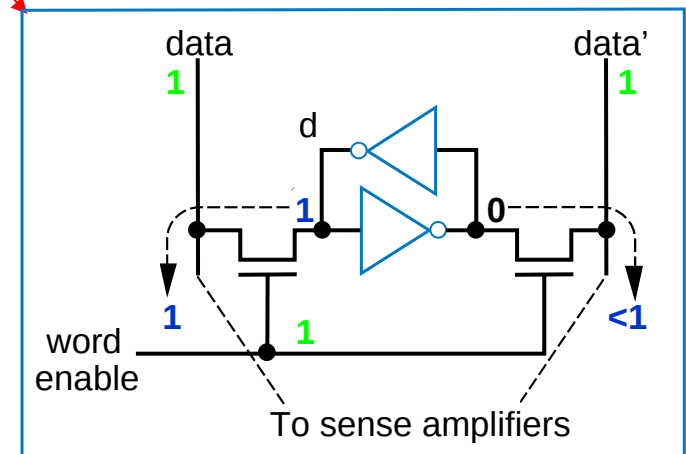


Static RAM (SRAM)

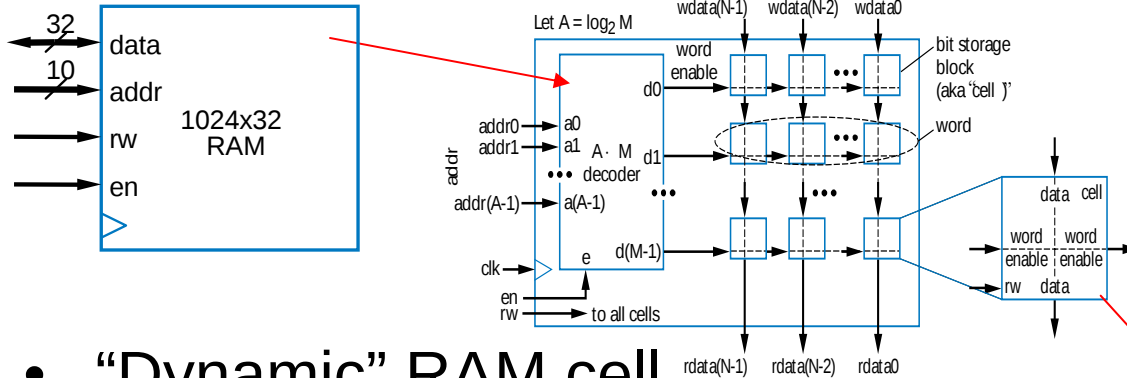


- “Static” RAM cell
 - Reading this cell
 - Somewhat trickier
 - When *rw* set to read, the RAM logic sets both *data* and *data'* to 1
 - The stored bit *d* will pull either the left line or the right bit down slightly below 1
 - “Sense amplifiers” detect which side is slightly pulled down
 - The electrical description of SRAM is really beyond our scope – just general idea here, mainly to contrast with *DRAM*...

SRAM cell

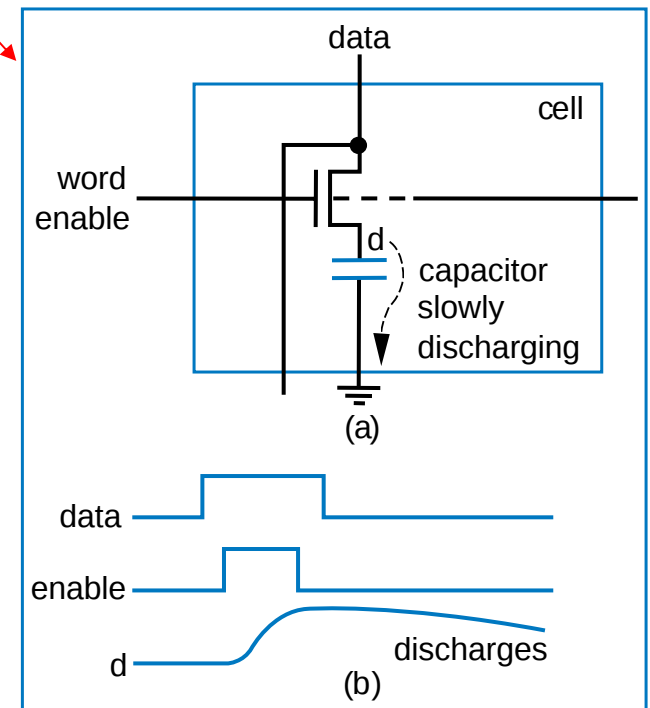


Dynamic RAM (DRAM)



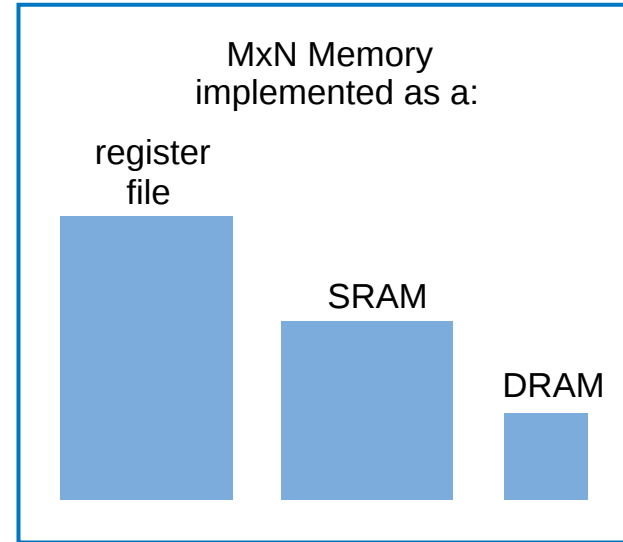
- “Dynamic” RAM cell
 - 1 transistor (rather than 6)
 - Relies on *large* capacitor to store bit
 - Write: Transistor conducts, data voltage level gets stored on top plate of capacitor
 - Read: Just look at value of d
 - Problem: Capacitor discharges over time
 - Must “refresh” regularly, by reading d and then writing it right back

DRAM cell



Comparing Memory Types

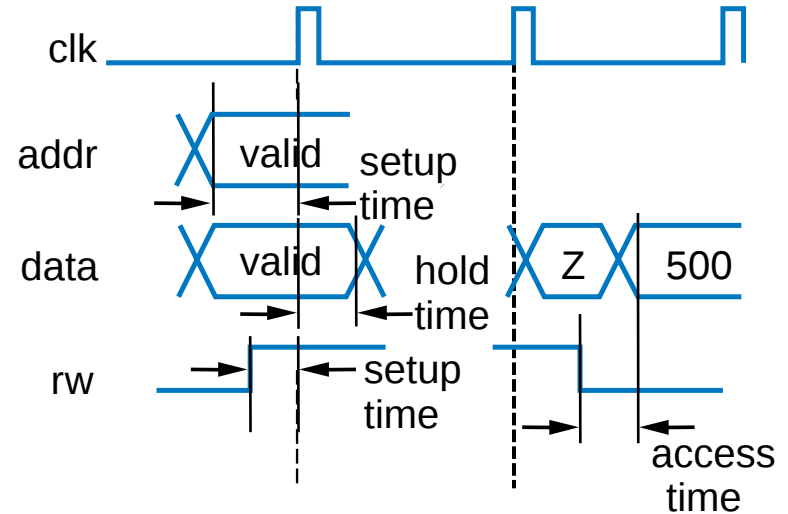
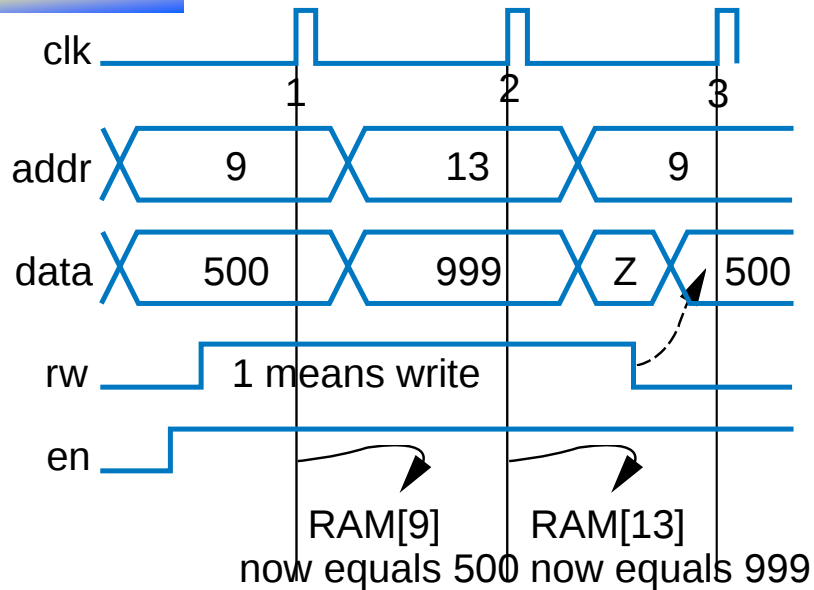
- Register file
 - Fastest
 - But biggest size
- SRAM
 - Fast
 - More compact than register file
- DRAM
 - Slowest
 - And refreshing takes time
 - But very compact
- Use register file for small items, SRAM for large items, and DRAM for huge items
 - Note: DRAM's big capacitor requires a special chip design process, so DRAM is often a separate chip



Size comparison for same number of bits (not to scale)



Reading and Writing a RAM

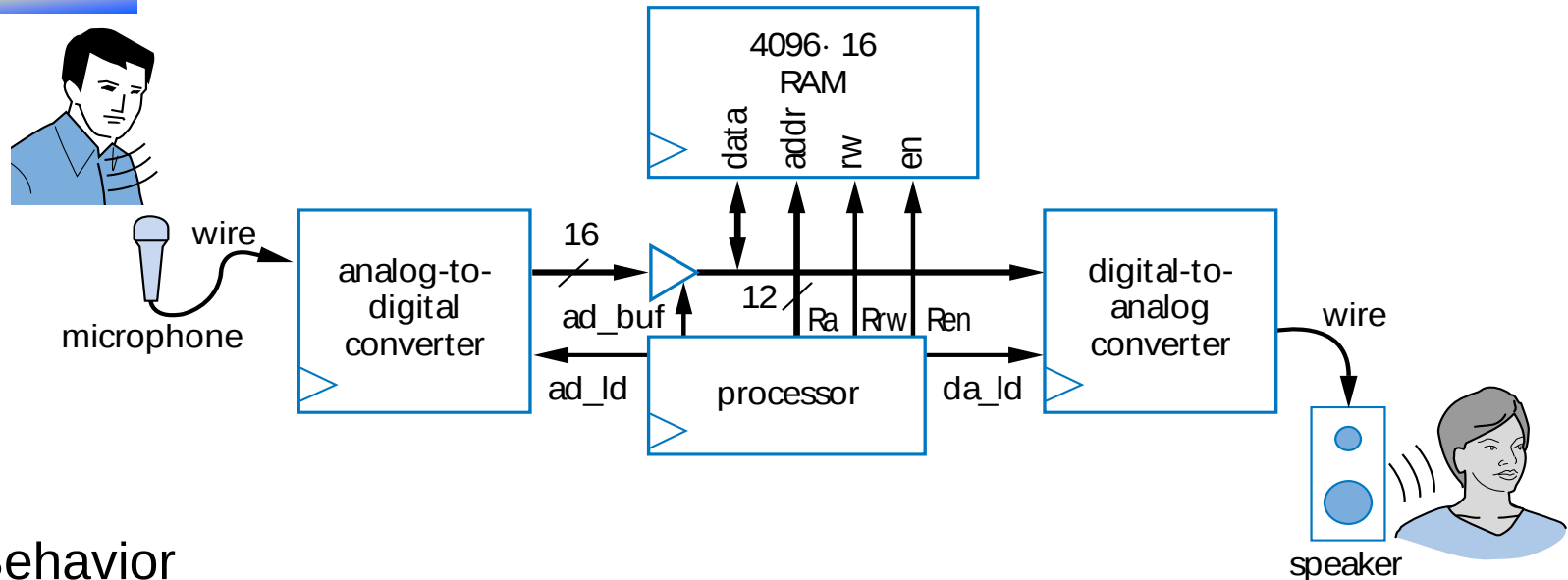


- Writing
 - Put address on *addr* lines, data on *data* lines, set *rw*=1, *en*=1
- Reading
 - Set *addr* and *en* lines, but put nothing (Z) on *data* lines, set *rw*=0
 - Data will appear on *data* lines
- Don't forget to obey setup and hold times
 - In short – keep inputs stable before and after a clock edge

(b)



RAM Example: Digital Sound Recorder

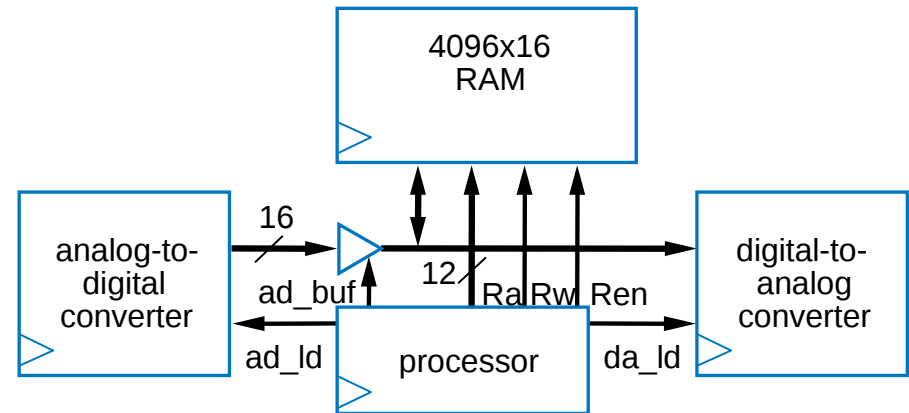


- Behavior
 - Record: Digitize sound, store as series of 4096 12-bit digital values in RAM
 - We'll use a 4096x16 RAM (12-bit wide RAM not common)
 - Play back later
 - Common behavior in telephone answering machine, toys, voice recorders
- To record, processor should read a-to-d, store read values into successive RAM words
 - To play, processor should read successive RAM words and enable d-to-a

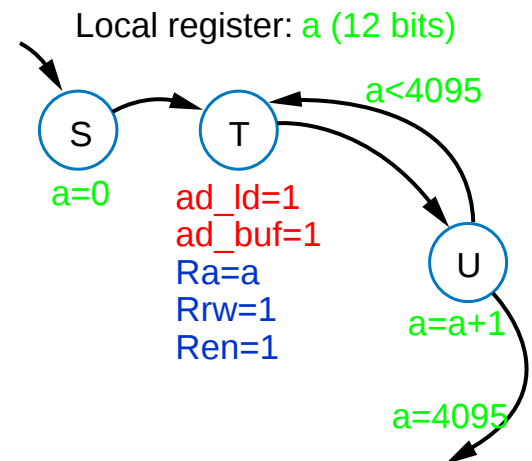


RAM Example: Digital Sound Recorder

- RTL design of processor
 - Create high-level state machine
 - Begin with the *record* behavior
 - Keep local register *a*
 - Stores current address, ranges from 0 to 4095 (thus need 12 bits)
 - Create state machine that counts from 0 to 4095 using *a*
 - For each *a*
 - Read analog-to-digital conv.
 - » $ad_ld=1, ad_buf=1$
 - Write to RAM at address *a*
 - » $Ra=a, Rrw=1, Ren=1$

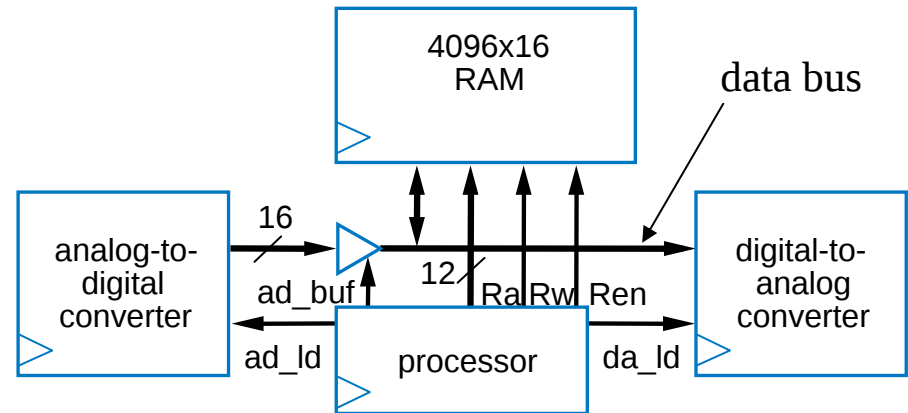


Record behavior

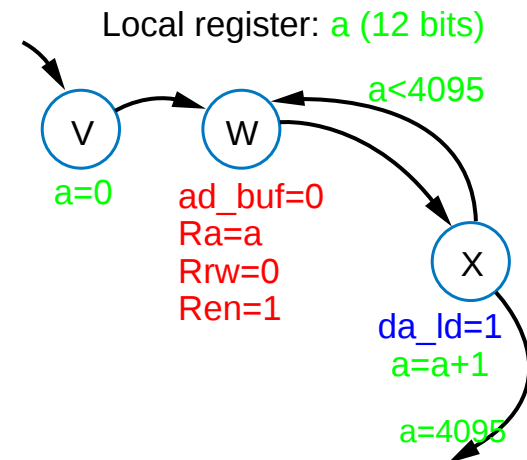


RAM Example: Digital Sound Recorder

- Now create *play* behavior
- Use local register *a* again, create **state machine that counts from 0 to 4095** again
 - For each *a*
 - **Read RAM**
 - **Write to digital-to-analog conv.**
 - Note: Must write d-to-a one cycle *after* reading RAM, when the read data is available on the data bus
- The record and play state machines would be parts of a larger state machine controlled by signals that determine when to record or play

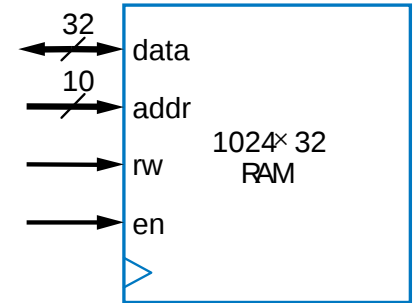


Play behavior

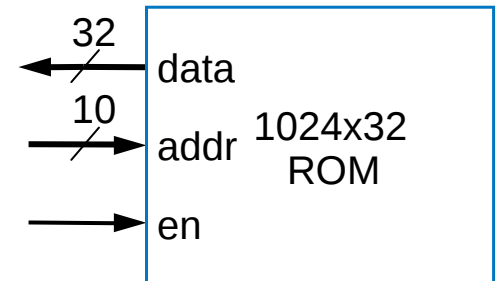


Read-Only Memory – ROM

- Memory that can only be read from, not written to
 - Data lines are output only
 - No need for *rw* input
- Advantages over RAM
 - Compact: May be smaller
 - **Nonvolatile**: Saves bits even if power supply is turned off
 - Speed: May be faster (especially than DRAM)
 - Low power: Doesn't need power supply to save bits, so can extend battery life
- Choose ROM over RAM if stored data won't change (or won't change often)
 - For example, a table of Celsius to Fahrenheit conversions in a digital thermometer



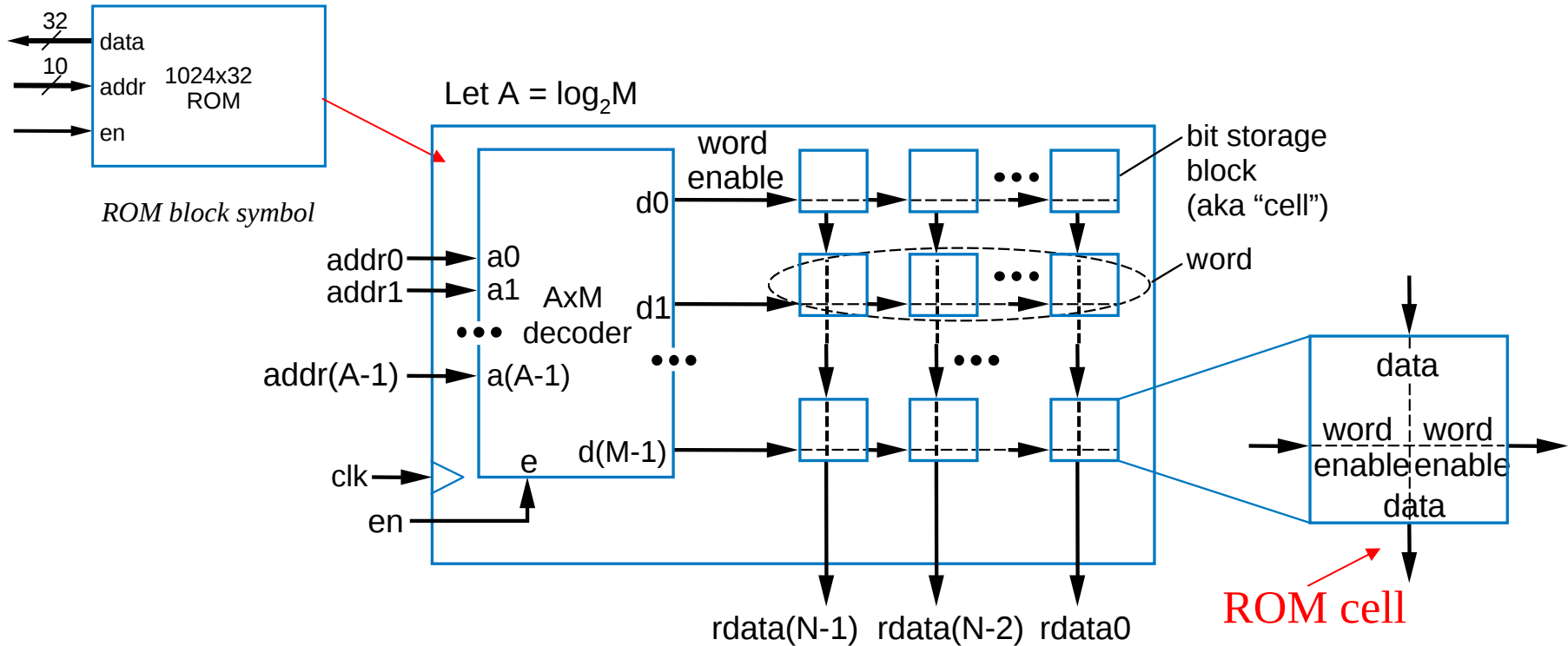
RAM block symbol



ROM block symbol



Read-Only Memory – ROM

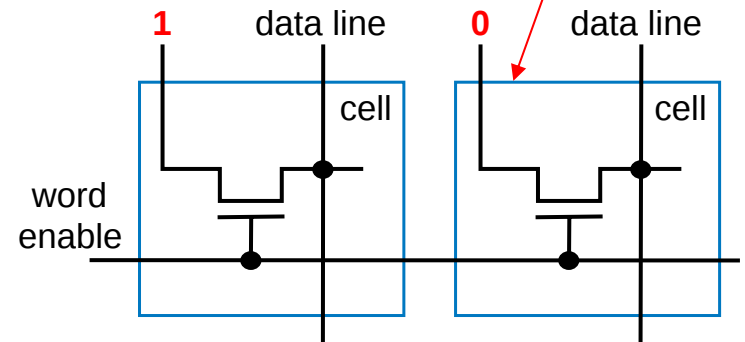
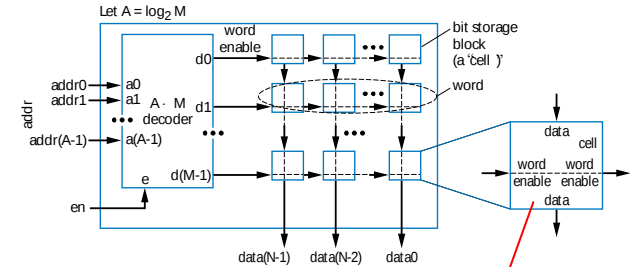


- Internal logical structure similar to RAM, without the data input lines



ROM Types

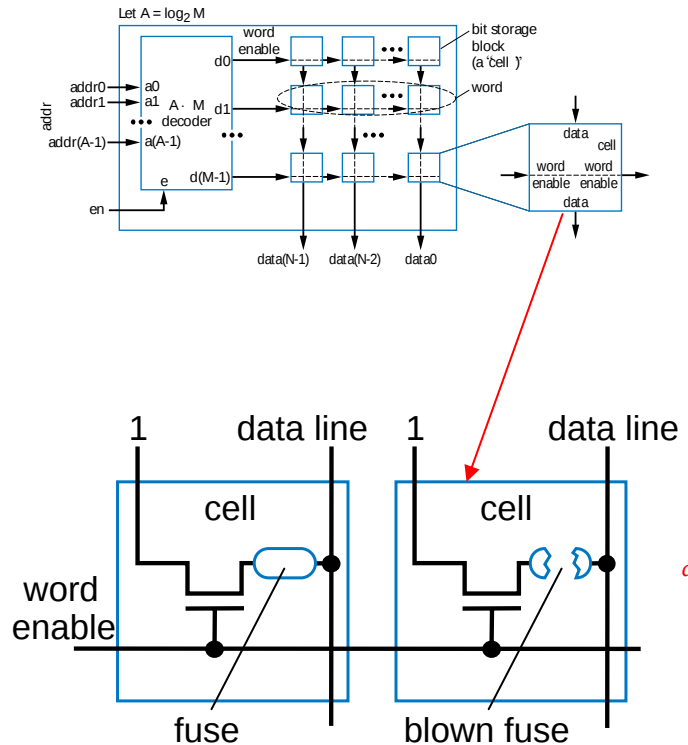
- If a ROM can only be read, how are the stored bits stored in the first place?
 - Storing bits in a ROM known as *programming*
 - Several methods
- **Mask-programmed ROM**
 - Bits are hardwired as 0s or 1s during chip manufacturing
 - 2-bit word on right stores “10”
 - word enable (from decoder) simply passes the hardwired value through transistor
 - Notice how compact, and fast, this memory would be



ROM Types

- **Fuse-Based Programmable ROM**

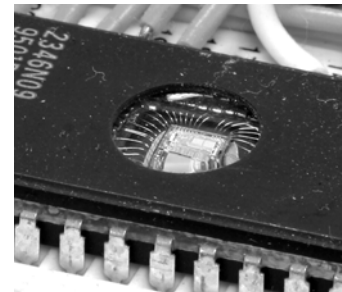
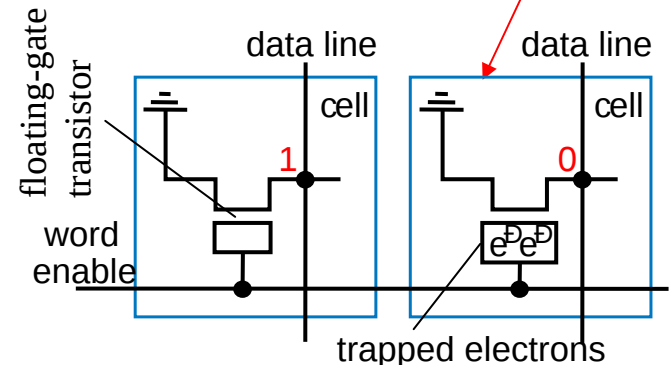
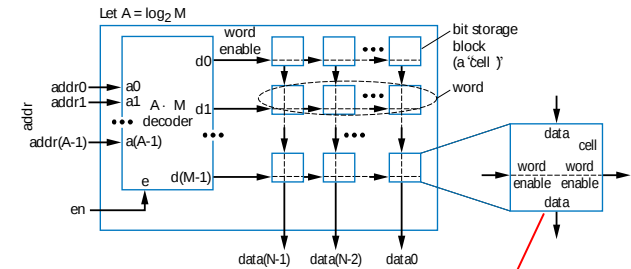
- Each cell has a fuse
- A special device, known as a programmer, blows certain fuses (using higher-than-normal voltage)
 - Those cells will be read as 0s (involving some special electronics)
 - Cells with unblown fuses will be read as 1s
 - 2-bit word on right stores “10”
- Also known as **One-Time Programmable (OTP) ROM**



ROM Types

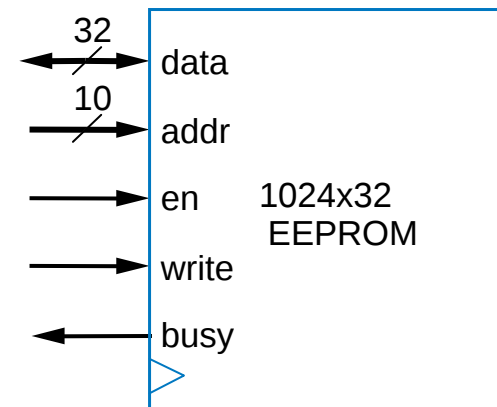
- **Erasable Programmable ROM (EPROM)**

- Uses “floating-gate transistor” in each cell
- Special programmer device uses higher-than-normal voltage to cause electrons to *tunnel* into the gate
 - Electrons become trapped in the gate
 - Only done for cells that should store 0
 - Other cells (without electrons trapped in gate) will be 1
 - 2-bit word on right stores “10”
 - Details beyond our scope – just general idea is necessary here
- To erase, shine ultraviolet light onto chip
 - Gives trapped electrons energy to escape
 - Requires chip package to have window

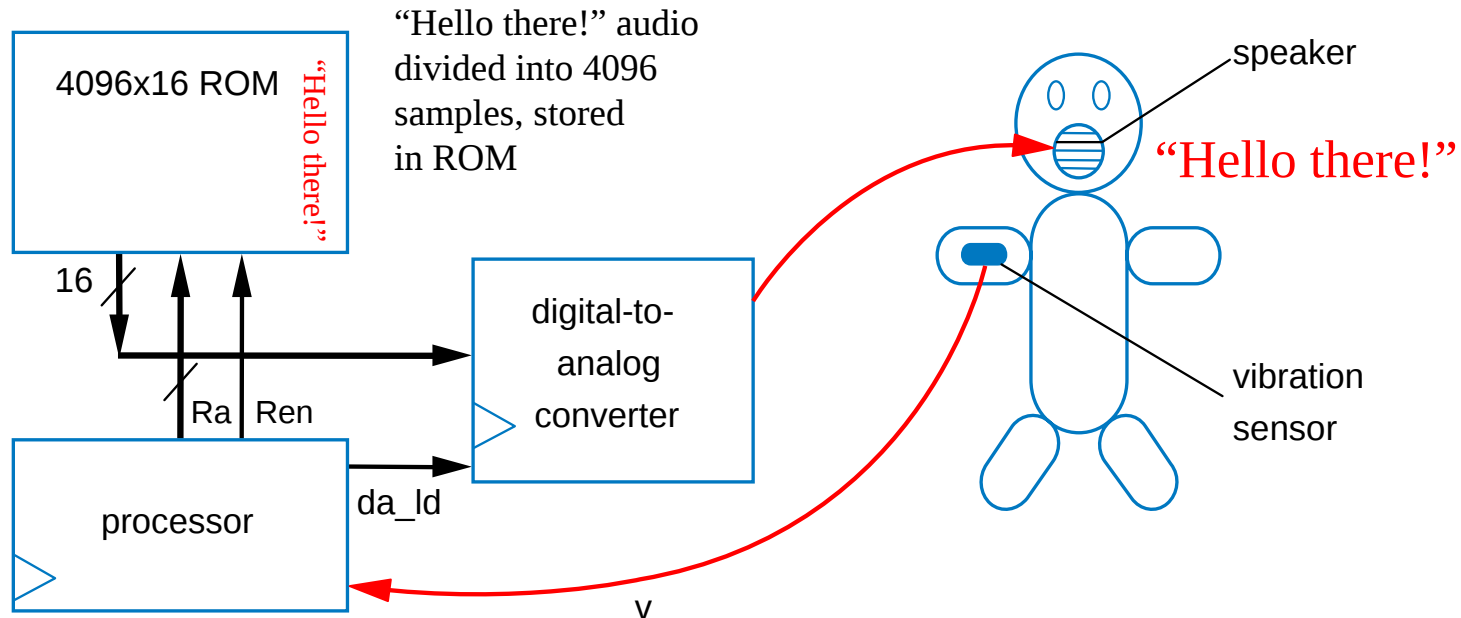


ROM Types

- **Electronically-Erasable Programmable ROM (EEPROM)**
 - Similar to EPROM
 - Uses floating-gate transistor, electronic programming to trap electrons in certain cells
 - But erasing done *electronically*, not using UV light
 - Erasing done one word at a time
- **Flash memory**
 - Like EEPROM, but all words (or large blocks of words) can be erased *simultaneously*
 - Become common relatively recently (late 1990s)
- Both types are in-system programmable
 - Can be programmed with new stored bits while in the system in which the ROM operates
 - Requires bi-directional data lines, and write control input
 - Also need busy output to indicate that erasing is in progress – erasing takes some time

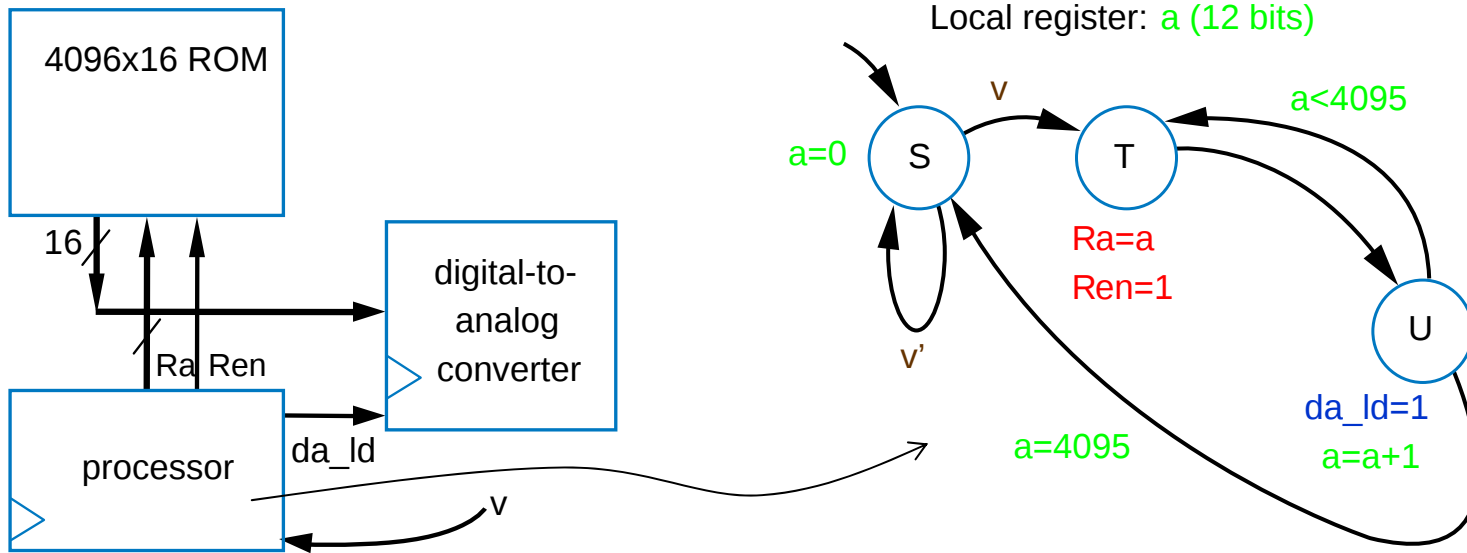


ROM Example: Talking Doll



- Doll plays prerecorded message, trigger by vibration
 - Message must be stored without power supply → Use a ROM, not a RAM, because ROM is nonvolatile
 - And because message will never change, use a mask-programmed ROM or OTP ROM
 - Processor should wait for vibration ($v=1$), then read words 0 to 4095 from the ROM, writing each to the d-to-a

ROM Example: Talking Doll



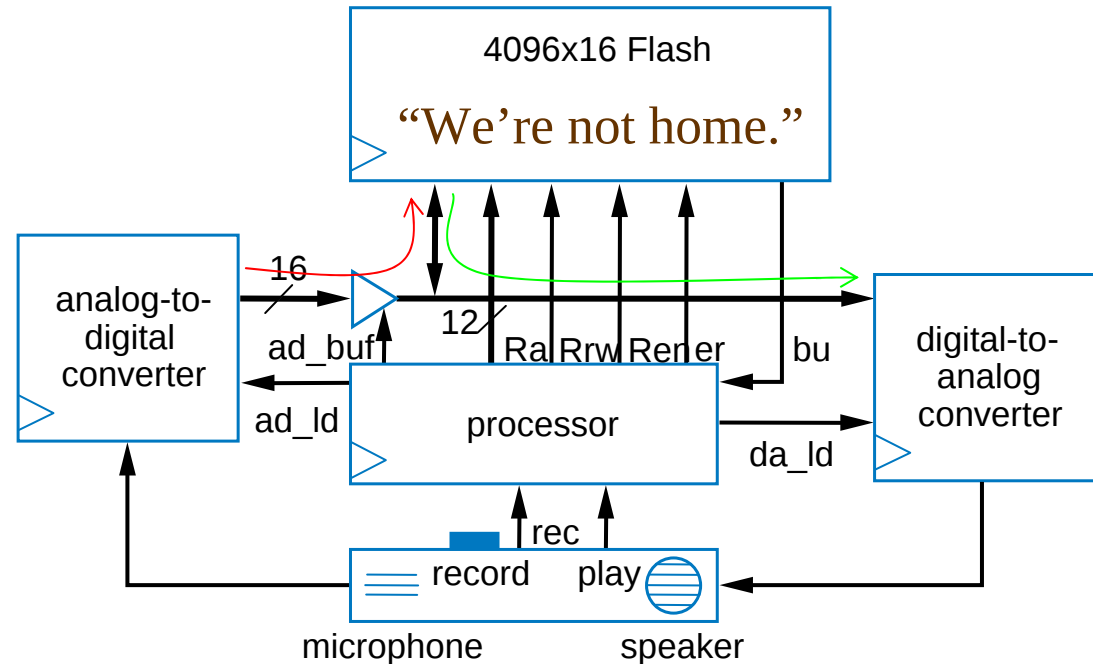
- High-level state machine

- Create state machine that waits for $v=1$, and then counts from 0 to 4095 using a local register a
- For each a , read ROM, write to digital-to-analog converter



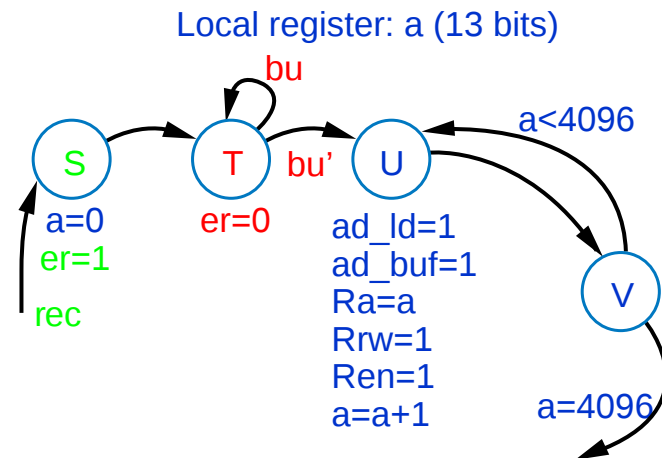
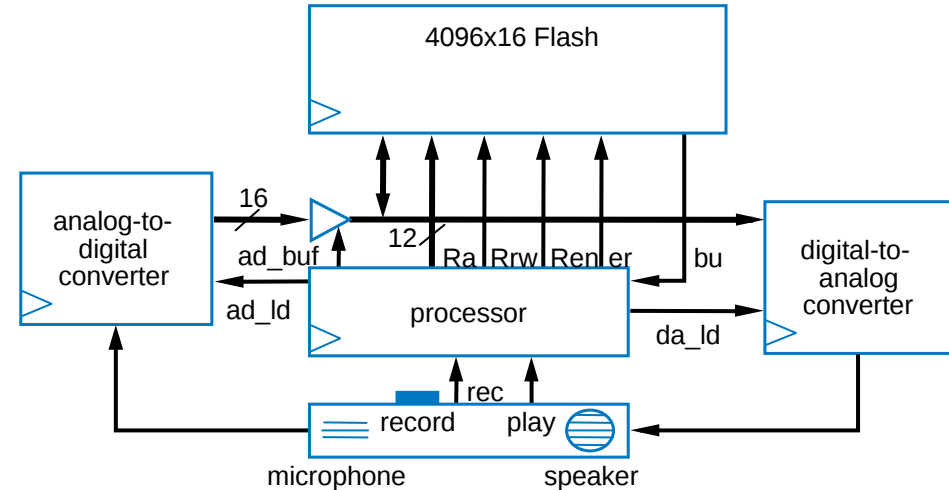
ROM Example: Digital Telephone Answering Machine Using a Flash Memory

- Want to record the **outgoing announcement**
 - When $rec=1$, **record** digitized sound in locations 0 to 4095
 - When $play=1$, **play** those stored sounds to digital-to-analog converter
- What type of memory?
 - Should store without power supply – ROM, not RAM
 - Should be in-system programmable – EEPROM or Flash, not EPROM, OTP ROM, or mask-programmed ROM
 - Will always erase entire memory when reprogramming – Flash better than EEPROM



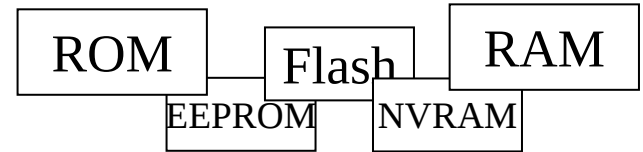
ROM Example: Digital Telephone Answering Machine Using a Flash Memory

- High-level state machine
 - Once $rec=1$, begin erasing flash by setting $er=1$
 - Wait for flash to finish erasing by waiting for $bu=0$
 - Execute loop that sets local register a from 0 to 4095, reading analog-to-digital converter and writing to flash for each a



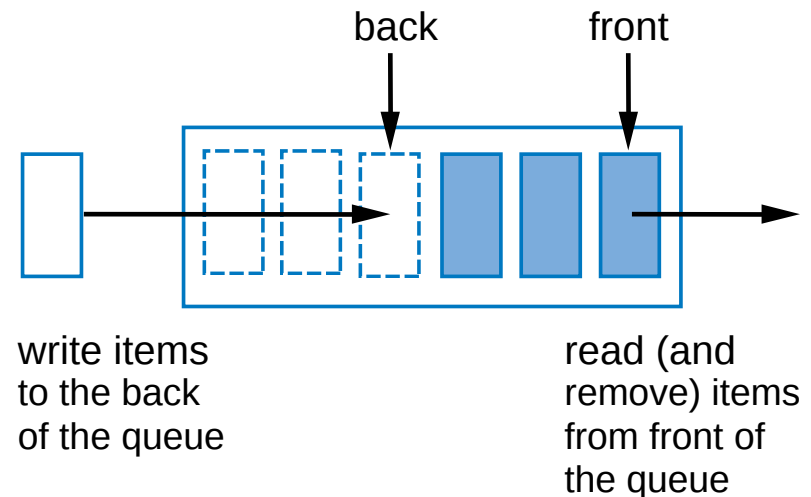
Blurring of Distinction Between ROM and RAM

- We said that
 - RAM is readable and writable
 - ROM is read-only
- But some ROMs act almost like RAMs
 - EEPROM and Flash are in-system programmable
 - Essentially means that writes are slow
 - Also, number of writes may be limited (perhaps a few million times)
- And, some RAMs act almost like ROMs
 - Non-volatile RAMs: Can save their data without the power supply
 - One type: Built-in battery, may work for up to 10 years
 - Another type: Includes ROM backup for RAM – controller writes RAM contents to ROM before turning off
- New memory technologies evolving that merge RAM and ROM benefits
 - e.g., MRAM
- Bottom line
 - Lot of choices available to designer, must find best fit with design goals



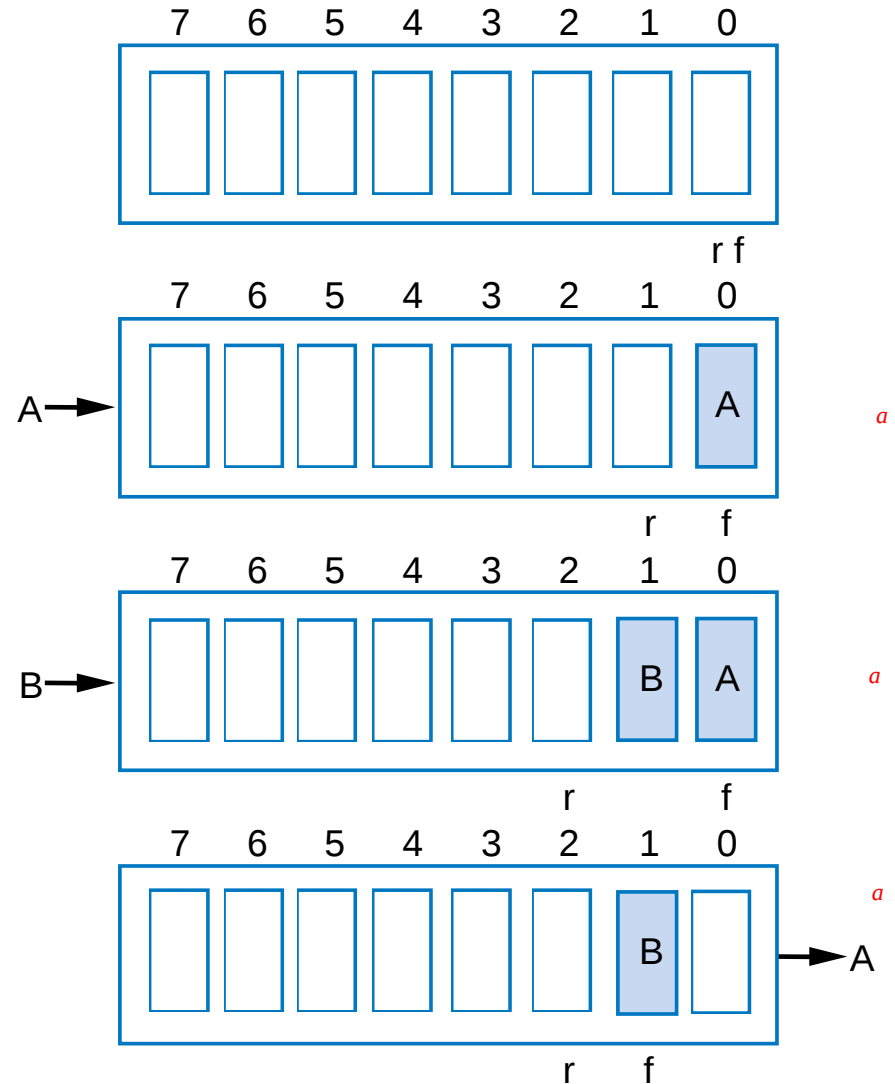
Queues

- A queue is another component sometimes used during RTL design
- **Queue**: A list written to at the back, from read from the front
 - Like a list of waiting restaurant customers
- Writing called a **push**, reading called a **pop**
- Because first item written into a queue will be the first item read out, also called a **FIFO** (first-in-first-out)



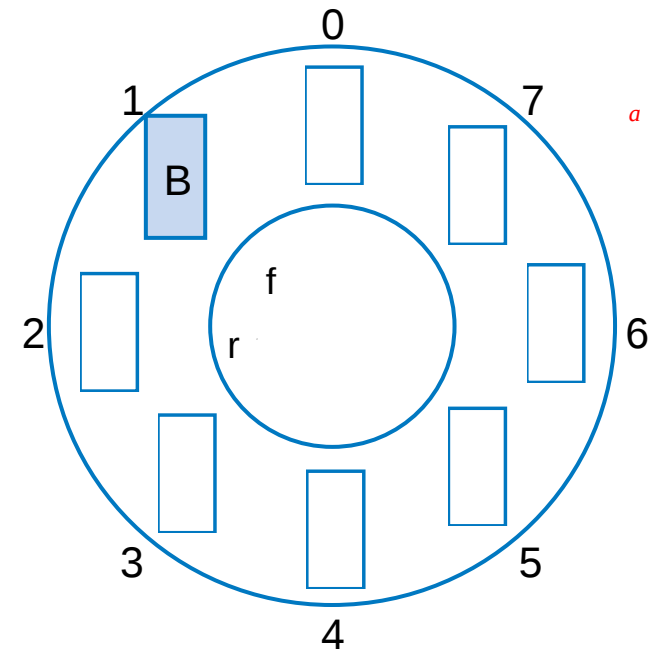
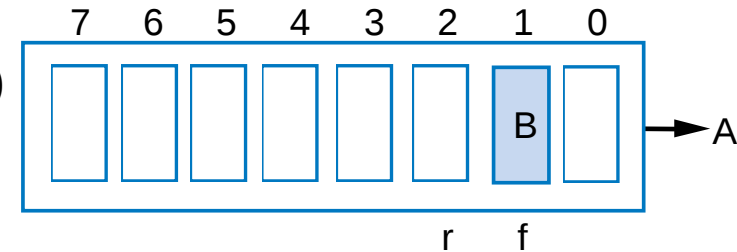
Queues

- Queue has addresses, and two pointers: *rear* and *front*
 - Initially both point to 0
- Push (write)
 - Item written to address pointed to by *rear*
 - rear* incremented
- Pop (read)
 - Item read from address pointed to by *front*
 - front* incremented
- If front or rear reaches 7, next (incremented) value should be 0 (for a queue with addresses 0 to 7)



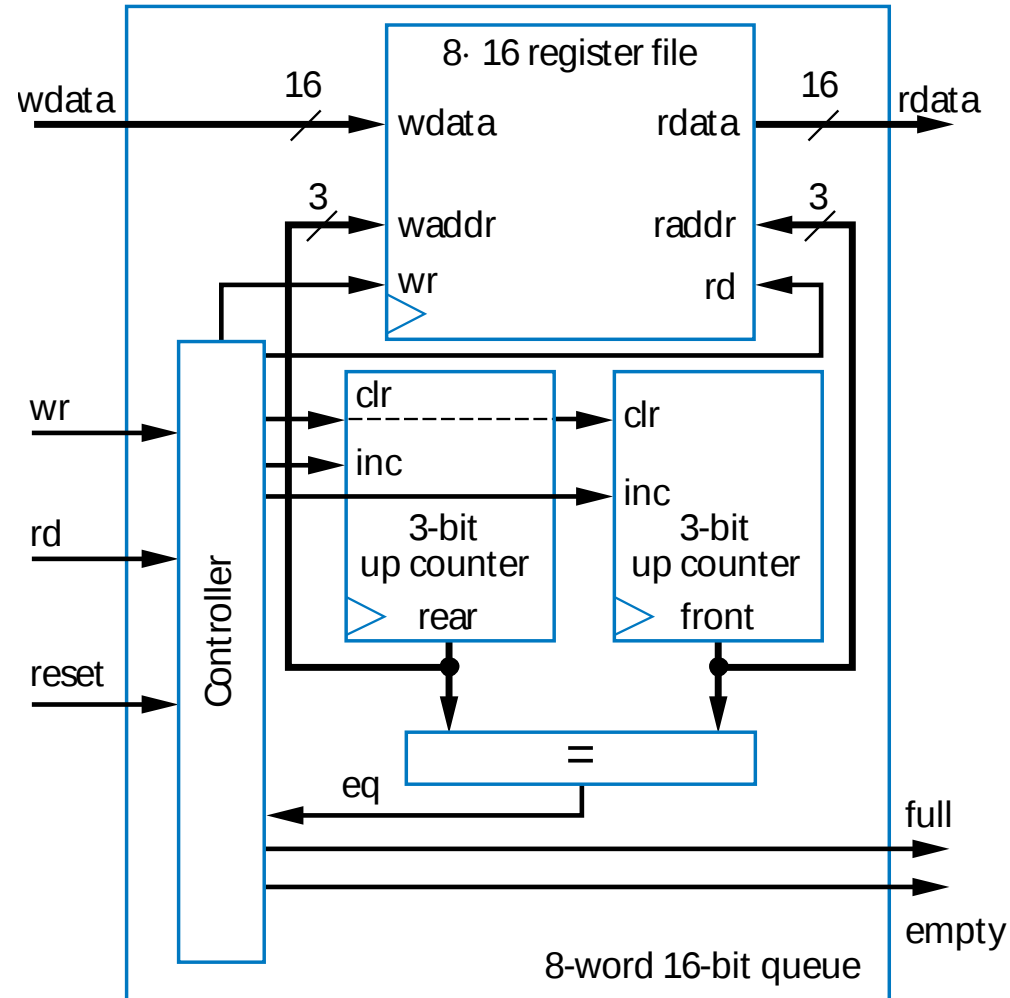
Queues

- Treat memory as a circle
 - If front or rear reaches 7, next (incremented) value should be 0 rather than 8 (for a queue with addresses 0 to 7)
- Two conditions of interest
 - Full queue – no room for more items
 - In 8-entry queue, means 8 items present
 - No further pushes allowed until a pop occurs
 - Causes front=rear
 - Empty queue – no items
 - No pops allowed until a push occurs
 - Causes front=rear
 - Both conditions have front=rear
 - To detect whether front=rear means full or empty, need state machine that detects if previous operation was push or pop, sets full or empty output signal (respectively)



Queue Implementation

- Can use register file for item storage
- Implement *rear* and *front* using up counters
 - rear used as register file's write address, front as read address
- Simple controller would set control lines for pushes and pops, and also detect full and empty situations
 - FSM for controller not shown



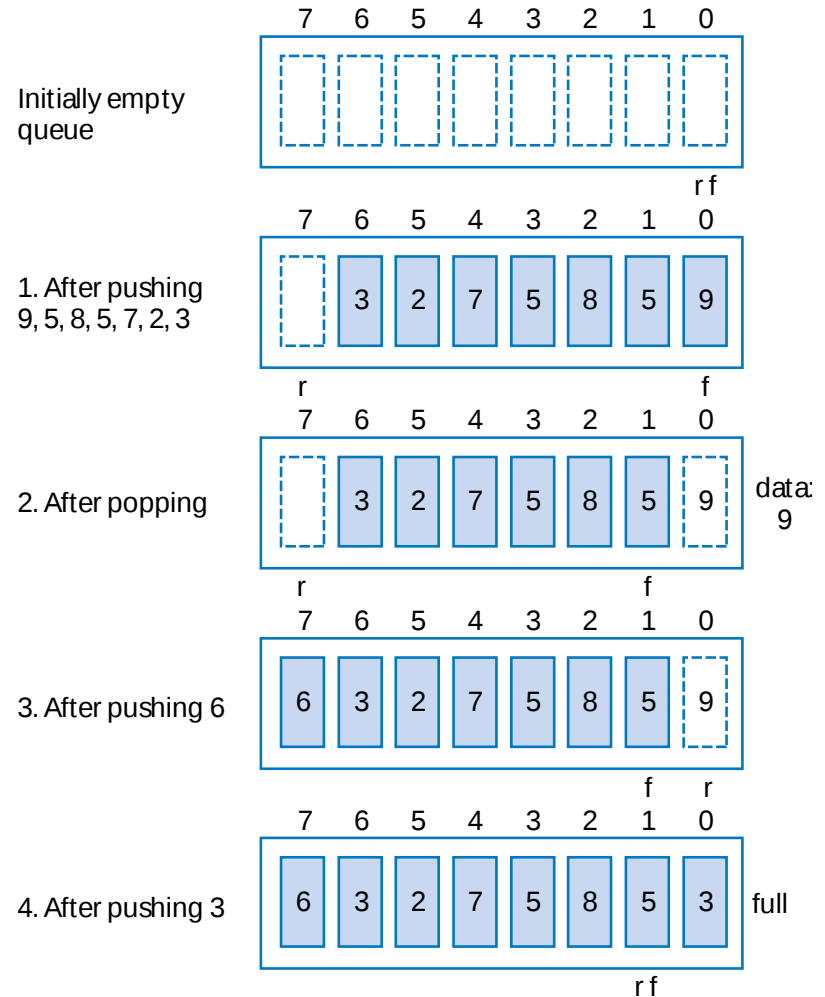
Common Uses of a Queue

- Computer keyboard
 - Pushes pressed keys onto queue, meanwhile pops and sends to computer
- Digital video recorder
 - Pushes captured frames, meanwhile pops frames, compresses them, and stores them
- Computer network routers
 - Pushes incoming packets onto queue, meanwhile pops packets, processes destination information, and forwards each packet out over appropriate port



Queue Usage Example

- Example series of pushes and pops
 - Note how rear and front pointers move
 - Note that popping doesn't really remove the data from the queue, but that data is no longer accessible
 - Note how rear (and front) wraps around from address 7 to 0
- Note: pushing a full queue is an error
 - As is popping an empty queue

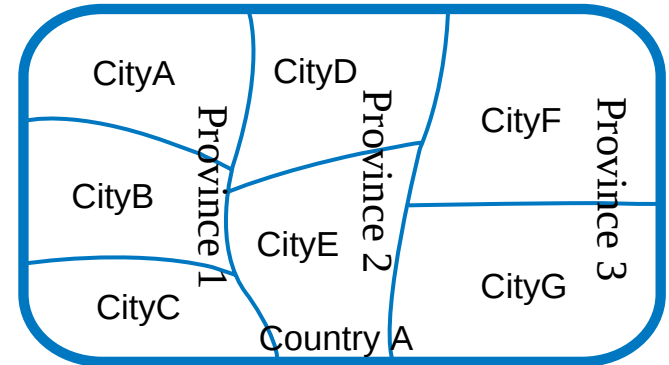


ERROR! Pushing a full queue results in unknown state

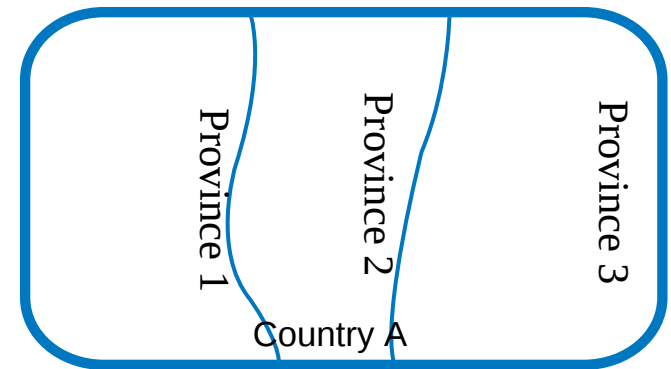


Hierarchy – A Key Design Concept

- Hierarchy
 - An organization with a few items at the top, with each item decomposed into other items
 - Common example: A country
 - 1 item at the top (the country)
 - Country item decomposed into state/province items
 - Each state/province item decomposed into city items
- Hierarchy helps us ***manage complexity***
 - To go from transistors to gates, muxes, decoders, registers, ALUs, controllers, datapaths, memories, queues, etc.
 - Imagine trying to comprehend a controller and datapath at the level of gates



Map showing all levels of hierarchy

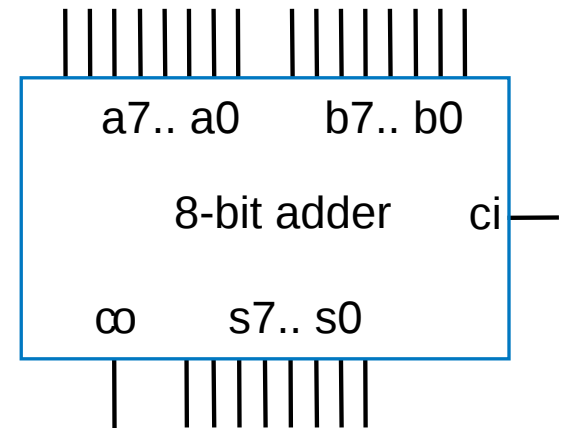


Map showing just top two levels of hierarchy



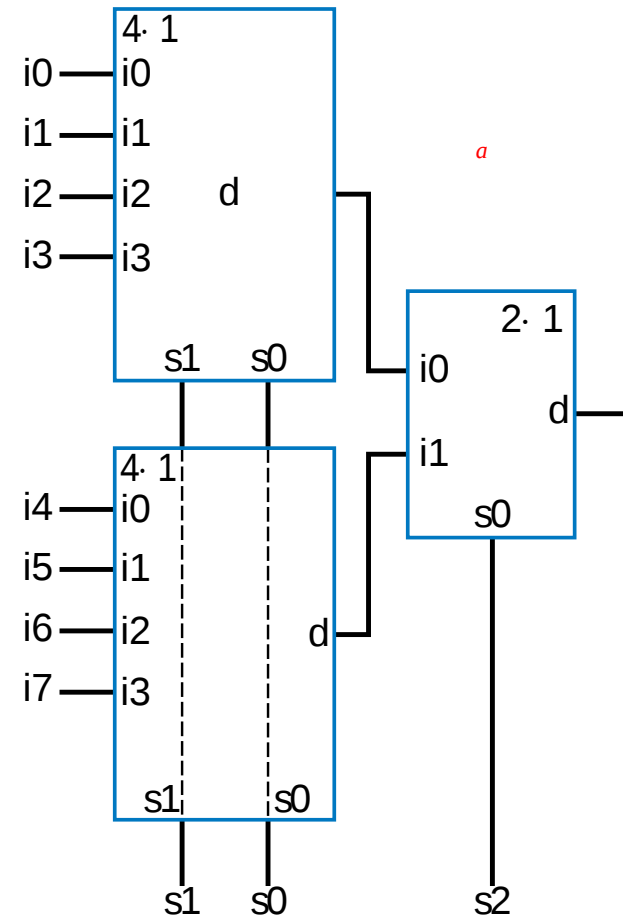
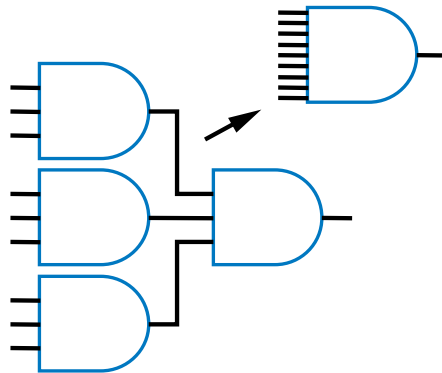
Hierarchy and Abstraction

- Abstraction
 - Hierarchy often involves not just grouping items into a new item, but also associating higher-level behavior with the new item, known as abstraction
 - e.g., an 8-bit adder has an understandable high-level behavior – it adds two 8-bit binary numbers
 - Frees designer from having to remember, or even from having to understand, the lower-level details



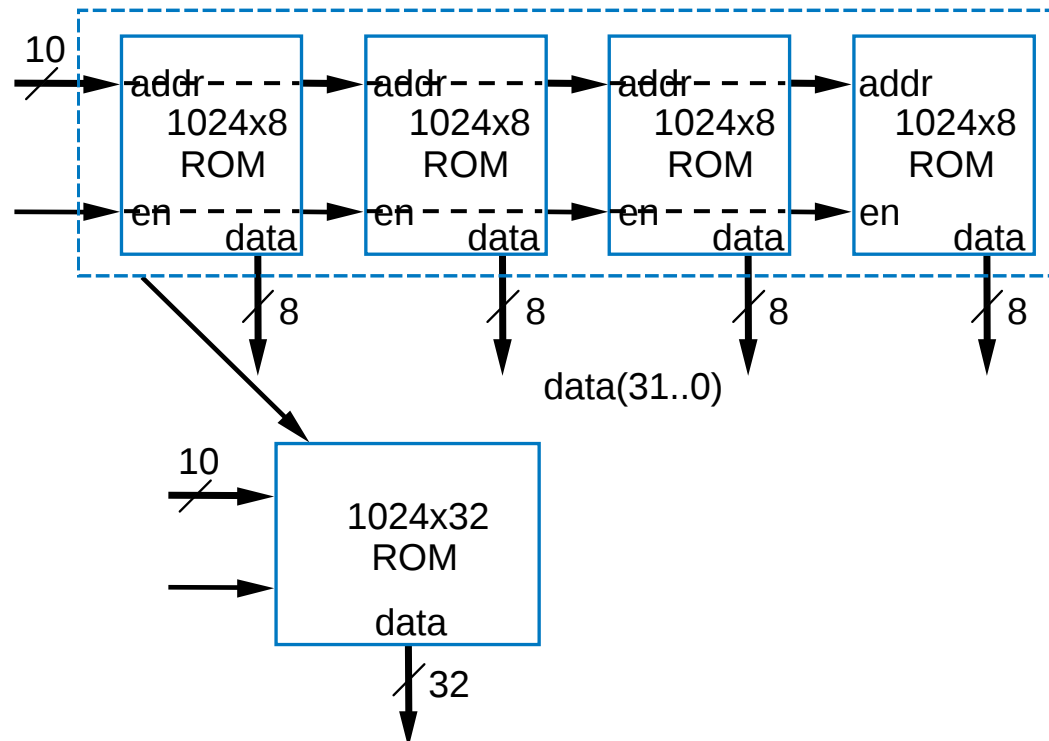
Hierarchy and Composing Larger Components from Smaller Versions

- A common task is to compose smaller components into a larger one
 - Gates: Suppose you have plenty of 3-input AND gates, but need a 9-input AND gate
 - Can simple compose the 9-input gate from several 3-input gates
 - Muxes: Suppose you have 4x1 and 2x1 muxes, but need an 8x1 mux
 - s2 selects either top or bottom 4x1
 - s1s0 select particular 4x1 input
 - Implements 8x1 mux – 8 data inputs, 3 selects, one output



Hierarchy and Composing Larger Components from Smaller Versions

- Composing memory very common
- Making memory words wider
 - Easy – just place memories side-by-side until desired width obtained
 - Share address/control lines, concatenate data lines
 - Example: Compose 1024x8 ROMs into 1024x32 ROM



Chapter Summary

- Modern digital design involves creating processor-level components
- Four-step RTL method can be used
 - 1. High-level state machine 2. Create datapath 3. Connect datapath to controller 4. Derive controller FSM
- Several example
 - Control dominated, data dominated, and mix
- Determining fastest clock frequency
 - By finding critical path
- Behavioral-level design – C to gates
 - By using method to convert C (subset) to high-level state machine
- Additional RTL components
 - Memory: RAM, ROM
 - Queues
- Hierarchy: A key concept used throughout Chapters 2-5

