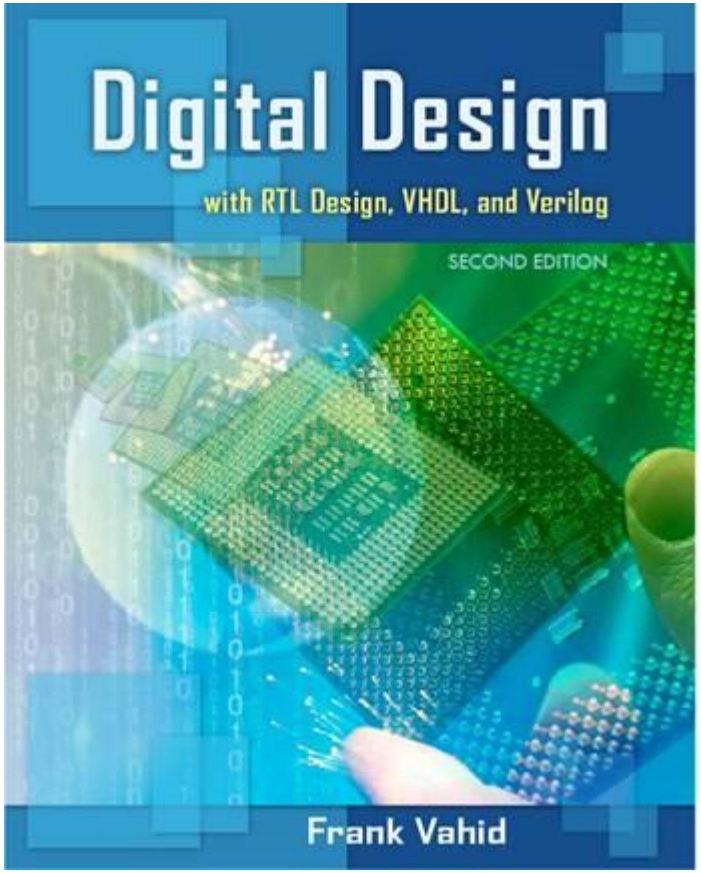


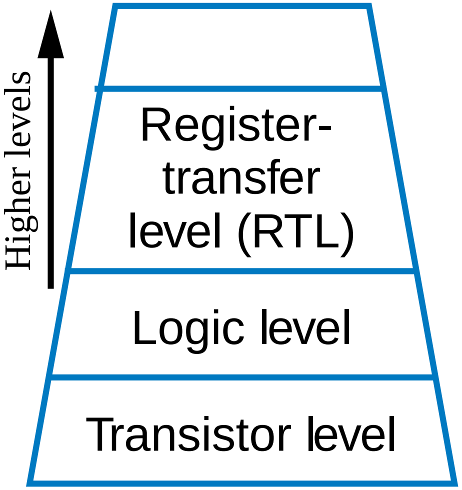
Digital Design

Chapter 5:
Register-Transfer Level
(RTL) Design

Slides to accompany the textbook *Digital Design, with RTL Design, VHDL, and Verilog*, 2nd Edition,
by Frank Vahid, John Wiley and Sons Publishers, 2010.
http://www.ddvahid.com

Introduction

- Chpt 2
  - Capture Comb. behavior: Equations, truth tables
  - Convert to circuit: AND + OR + NOT → Comb. logic
- Chpt 3
  - Capture sequential behavior: FSMs
  - Convert to circuit: Register + Comb. logic → Controller
- Chpt 4
  - Datapath components, simple datapaths
- Chpt 5
  - Capture behavior: High-level state machine
  - Convert to circuit: Controller + Datapath → Processor
  - Known as “RTL” (register-transfer level) design



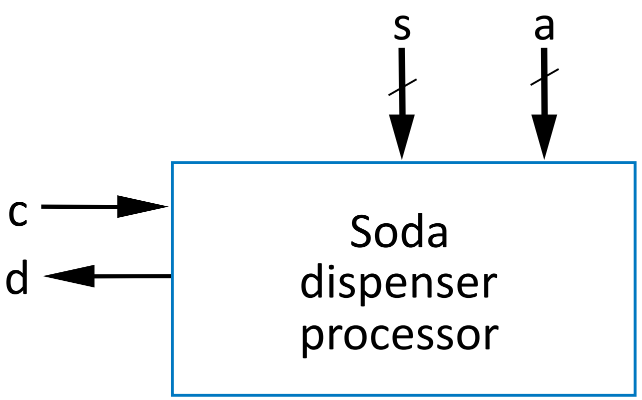
Levels of digital design abstraction

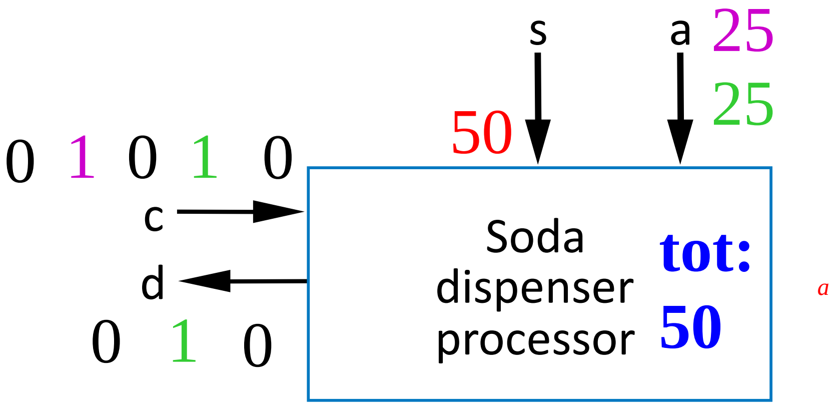
Processors:

- Programmable (microprocessor)
- Custom

High-Level State Machines (HLSMs)

- Some behaviors too complex for equations, truth tables, or FSMs
- Ex: 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
- FSM can't represent…
  - 8-bit input/output
  - Storage of current total
  - Addition (e.g., 25 + 10)

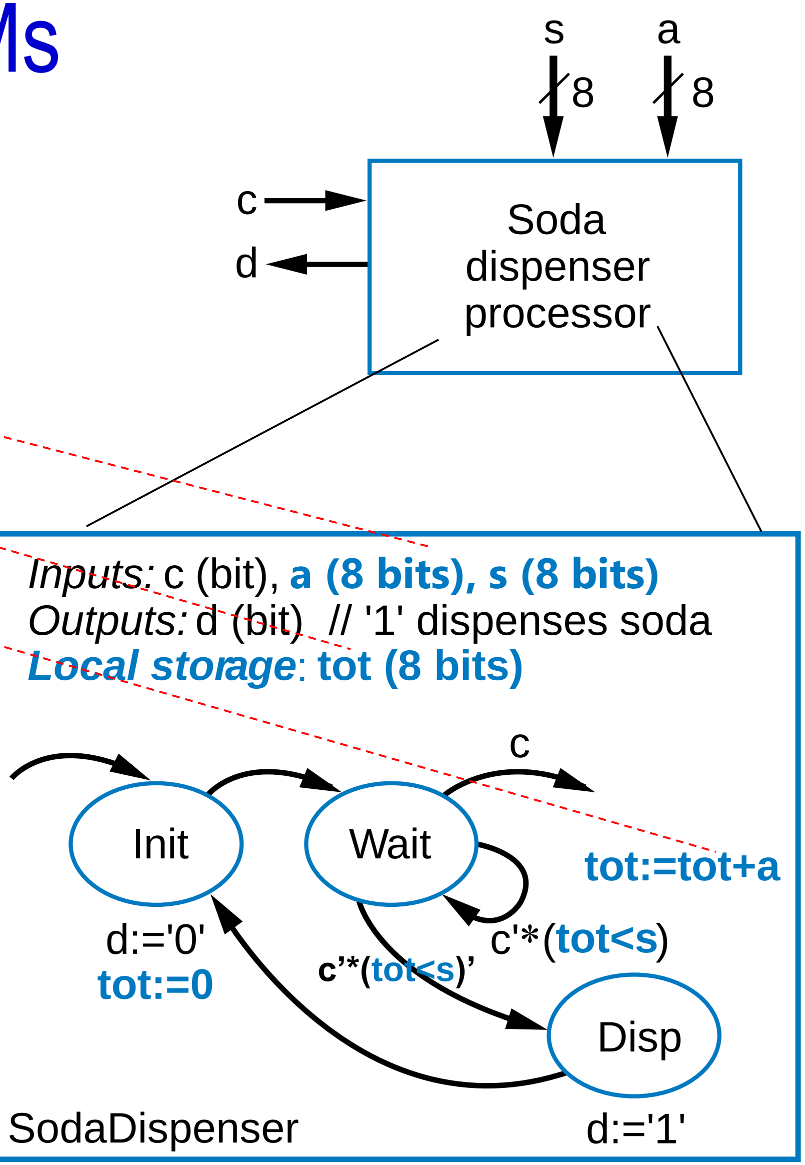




HLSMs

- High-level state machine (HLSM) extends FSM with:
  - Multi-bit input/output
  - Local storage
  - Arithmetic operations

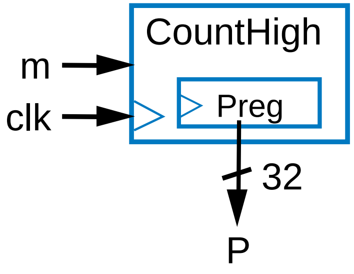
- Conventions
  - Numbers:
    - Single-bit: '0' (single quotes)
    - Integer: 0 (no quotes)
    - Multi-bit: “0000” (double quotes)
  - == for equal, := for assignment
  - Multi-bit outputs *must* be registered via local storage
  - // precedes a comment

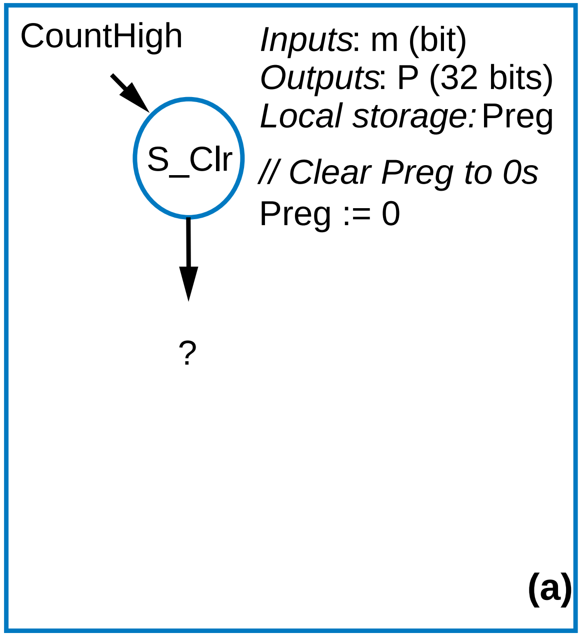


Inputs: c (bit), **a (8 bits), s (8 bits)**
Outputs: d (bit) // '1' dispenses soda
***Local storage*: tot (8 bits)**

Ex: Cycles-High Counter

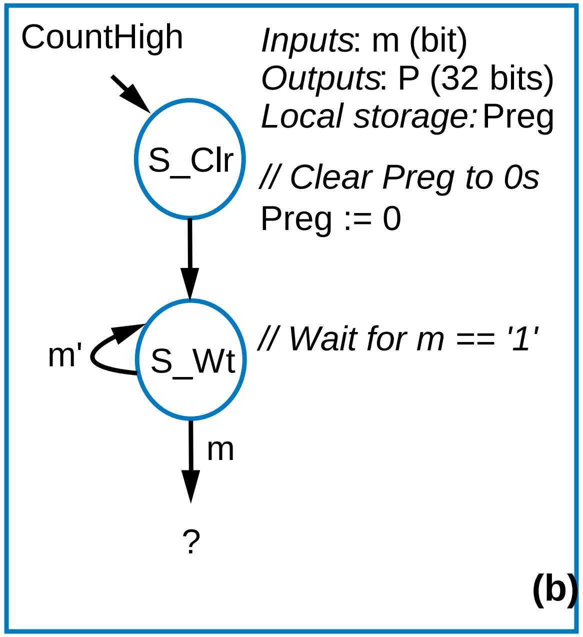
- P = total number (in binary) of cycles that m is 1
- Capture behavior as HLSM
  - Preg required (multibit outputs must be registered)
    - Use to hold count





Inputs: m (bit)
Outputs: P (32 bits)
Local storage: Preg

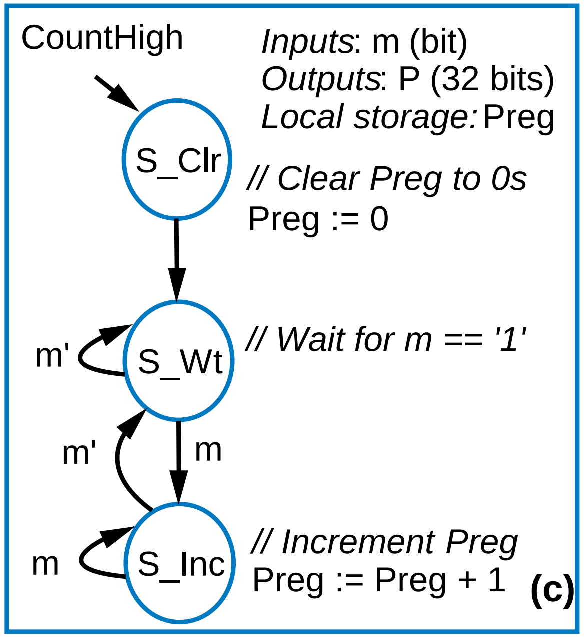
// Clear Preg to 0s
Preg := 0



Inputs: m (bit)
Outputs: P (32 bits)
Local storage: Preg

// Clear Preg to 0s
Preg := 0

// Wait for m == '1'



Inputs: m (bit)
Outputs: P (32 bits)
Local storage: Preg

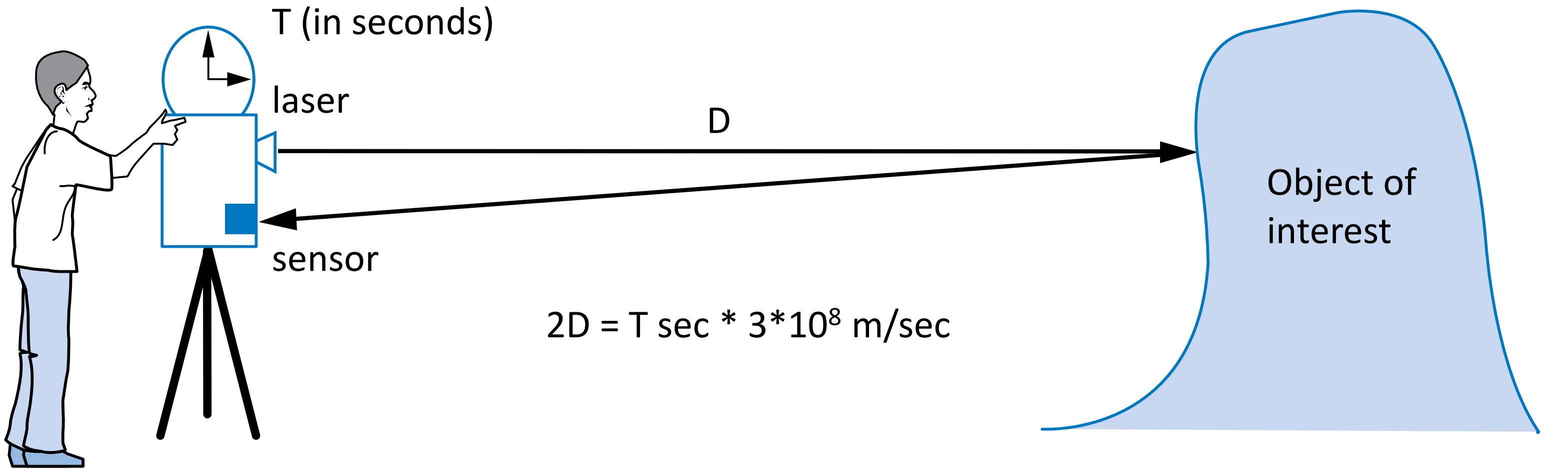
// Clear Preg to 0s
Preg := 0

// Wait for m == '1'

// Increment Preg
Preg := Preg + 1

Note: Could have designed directly using an up-counter. But, that methodology is ad hoc, and won't work for more complex examples, like the next one. *a*

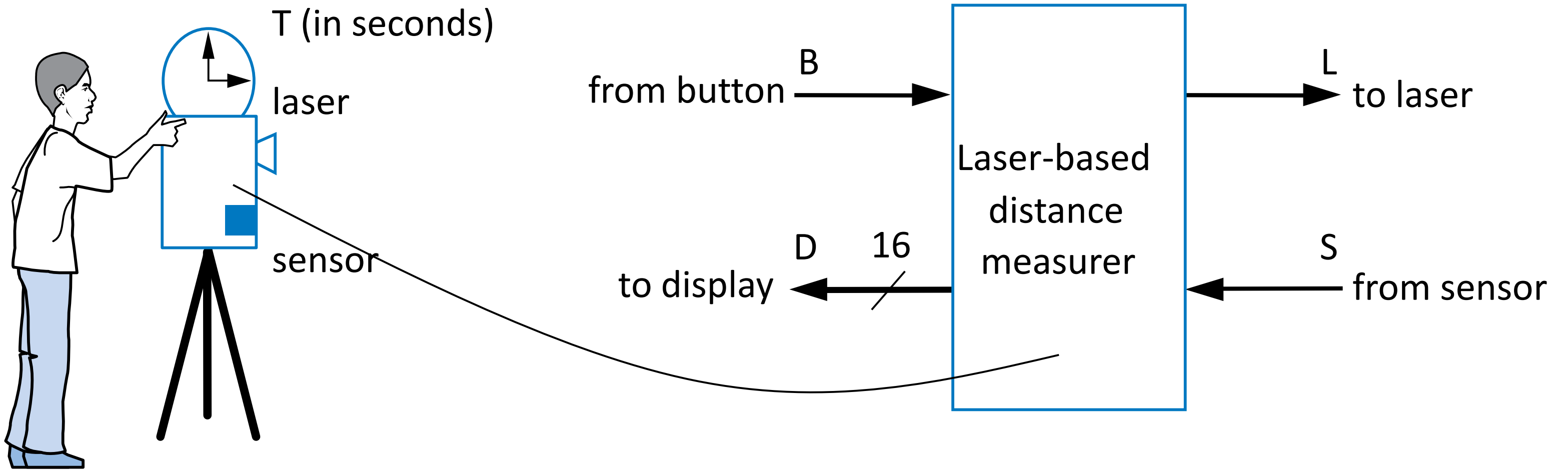
Example: Laser-Based Distance Measurer



a

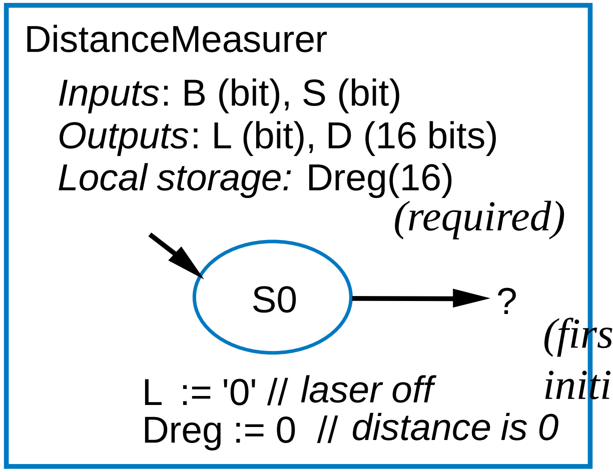
- Laser-based distance measurement – pulse laser, measure time T to sense reflection
  - Laser light travels at speed of light, $3*10^8$ m/sec
  - Distance is thus D = (T sec * $3*10^8$ m/sec) / 2

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, to display computed distance

Example: Laser-Based Distance Measurer



DistanceMeasurer

Inputs: B (bit), S (bit)
Outputs: L (bit), D (16 bits)
Local storage: Dreg(16)

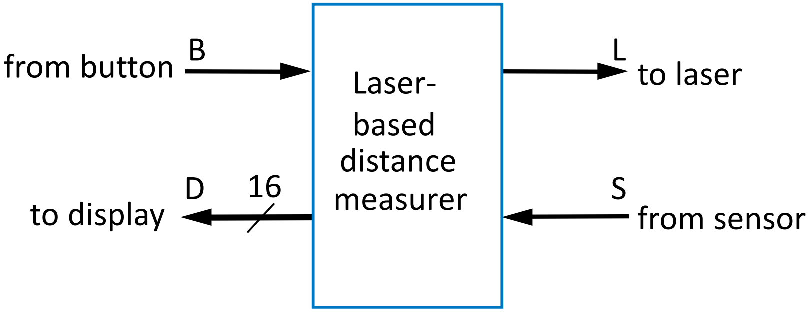
(required)

S0

?

L := '0' // *laser off*
Dreg := 0 // *distance is 0*

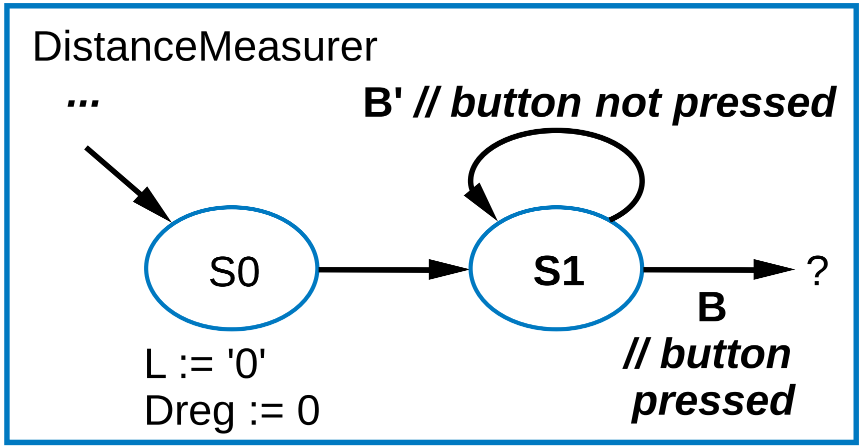
(first state usually initializes the system)

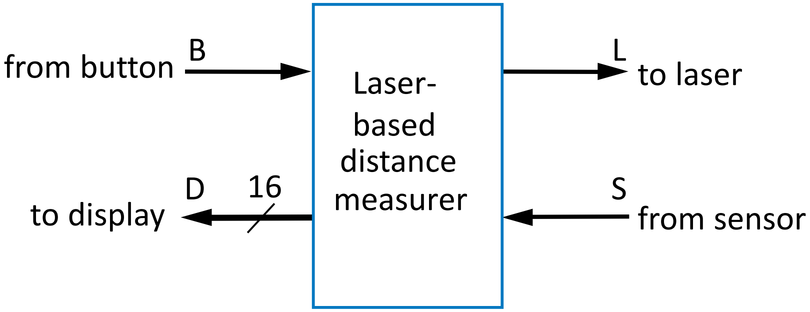


- Declare inputs, outputs, and local storage
  - Dreg required for multi-bit output
- Create initial state, name it **S0**
  - Initialize laser to off (L:='0')
  - Initialize displayed distance to 0 (Dreg:=0)

Recall: '0' means single bit, 0 means integer

Example: Laser-Based Distance Measurer



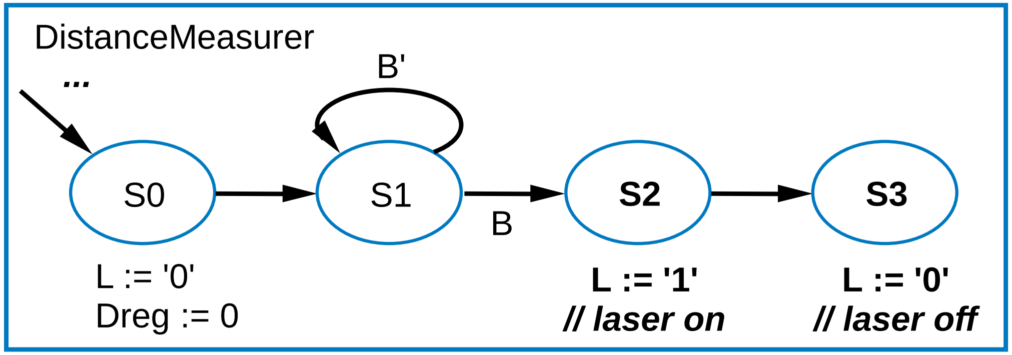


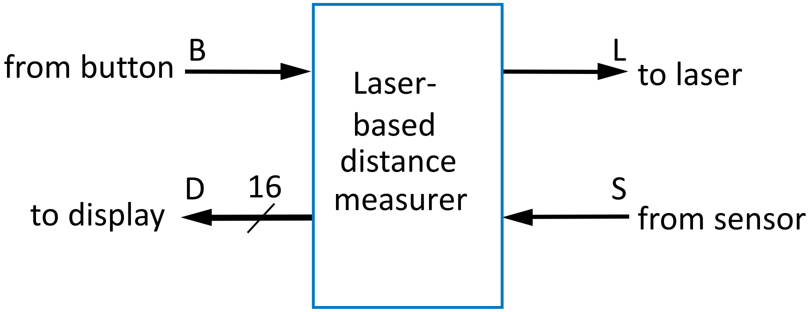
- Add another state, **S1**, that waits for a button press
  - B' – stay in **S1**, keep waiting
  - B – go to a new state **S2**

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

a

Example: Laser-Based Distance Measurer



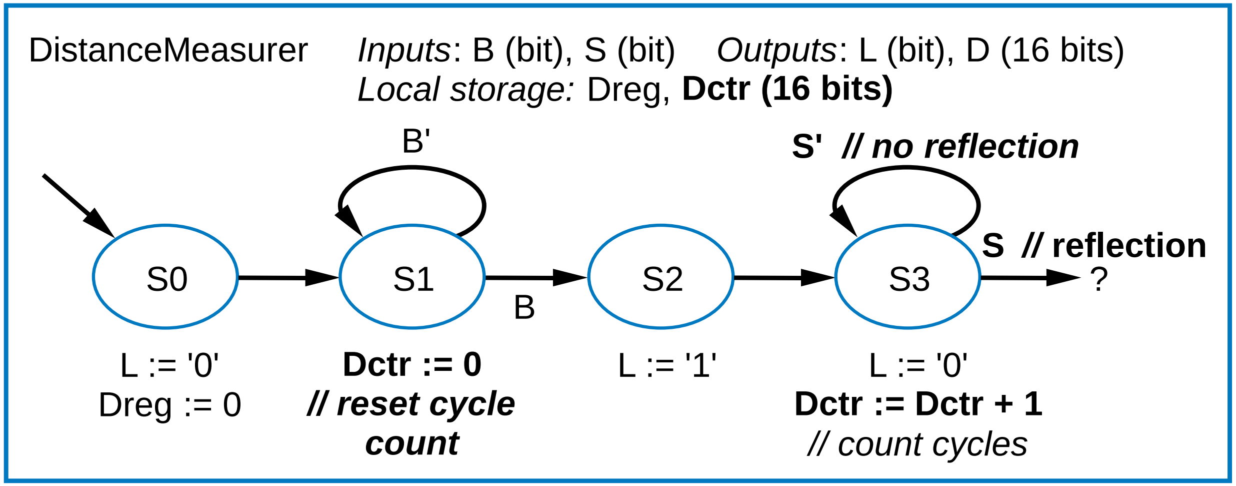


- Add a state **S2** that turns on the laser (L:='1')
- Then turn off laser (L:='0') in a state **S3**

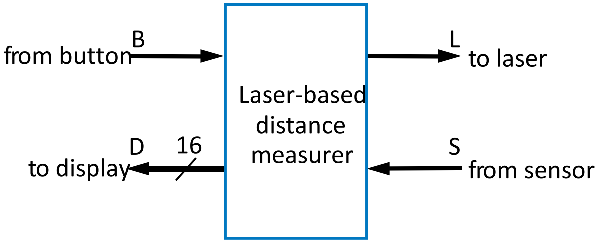
Q: What do next? A: Start timer, wait to sense reflection

a

Example: Laser-Based Distance Measurer



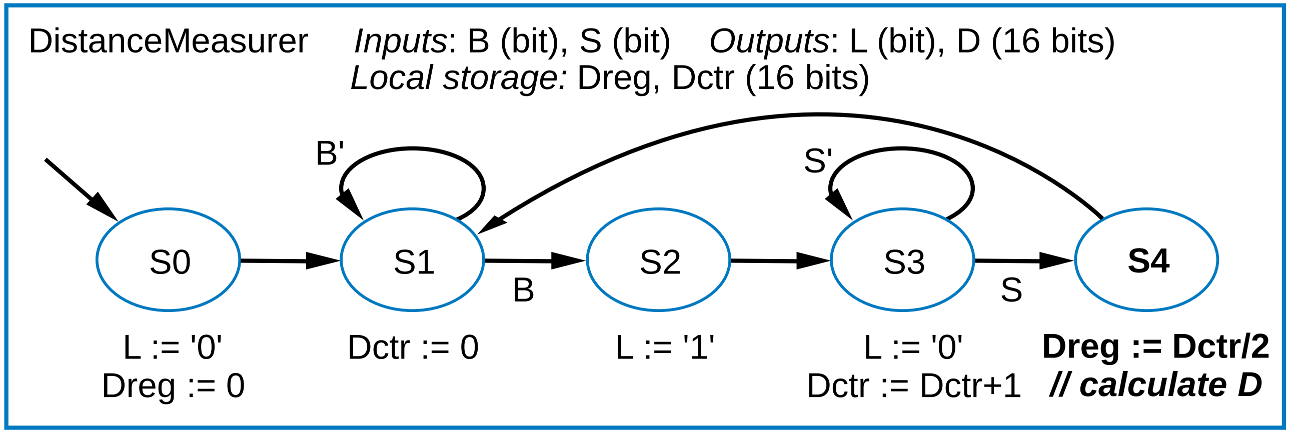
DistanceMeasurer *Inputs*: B (bit), S (bit) *Outputs*: L (bit), D (16 bits)
Local storage: Dreg, **Dctr (16 bits)**

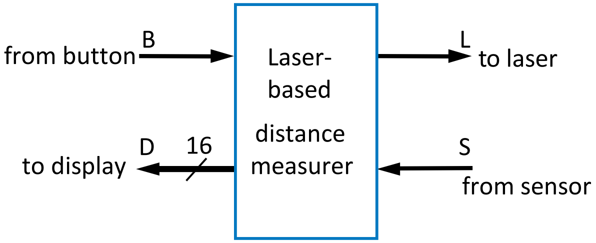


a

- Stay in **S3** until sense reflection (S)
- To measure time, count cycles while in **S3**
  - To count, declare local storage *Dctr*
  - Initialize *Dctr* to 0 in **S1**. In **S2** would have been O.K. too.
    - Don't forget to initialize local storage—common mistake
  - Increment *Dctr* each cycle in **S3**

Example: Laser-Based Distance Measurer



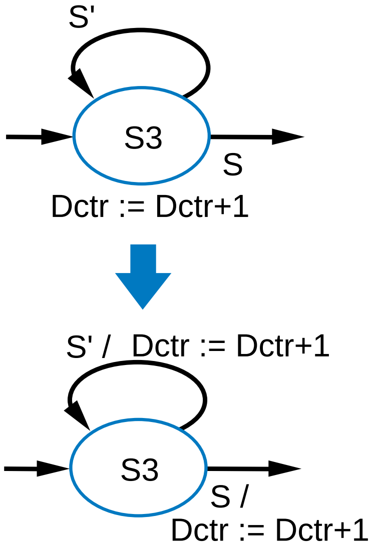


a

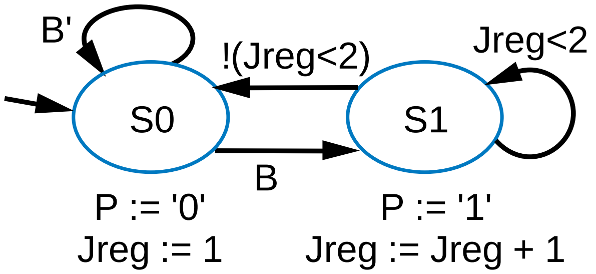
- Once reflection detected (S), go to new state **S4**
  - Calculate distance
  - Assuming clock frequency is 3x10^{8}, *Dctr* holds number of meters, so Dreg:=Dctr/2
- After **S4**, go back to **S1** to wait for button again

HLSM Actions: Updates Occur Next Clock Cycle

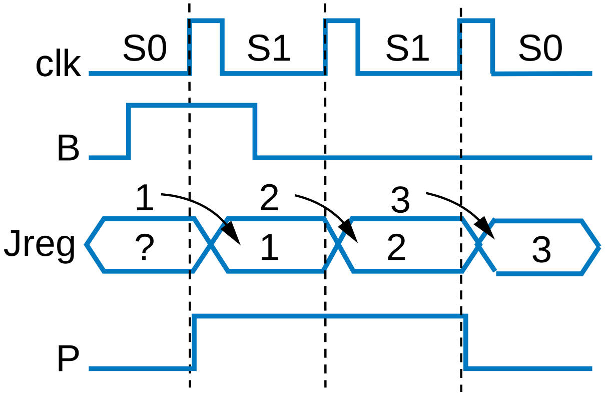
- Local storage updated on clock edges only
  - Enter state on clock edge
  - Storage writes in that state occur on *next* clock edge
  - Can think of as occurring on outgoing transitions
- ***Thus***, transition conditions use the OLD value, not the newly-written value
  - Example:



Inputs: B (bit)
Outputs: P (bit) // if B, 2 cycles high
Local storage: Jreg (8 bits)



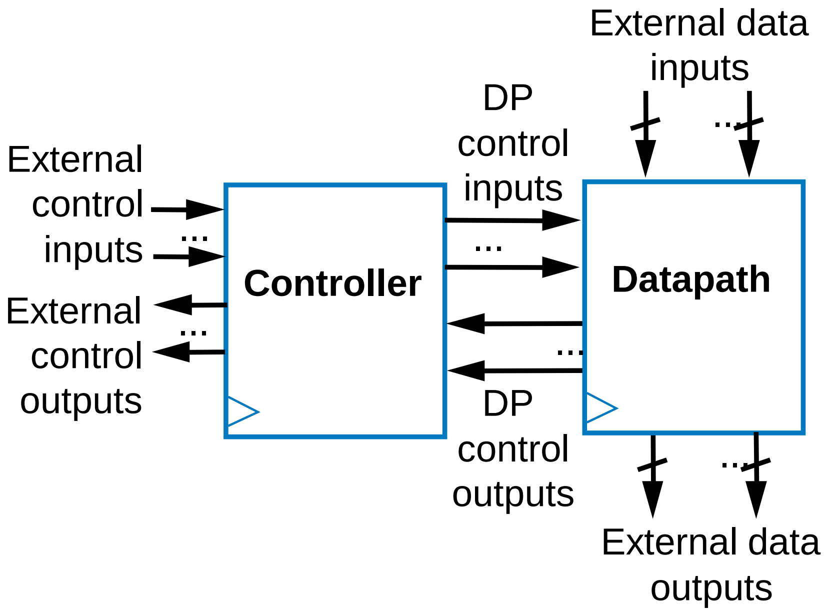
(**a**)



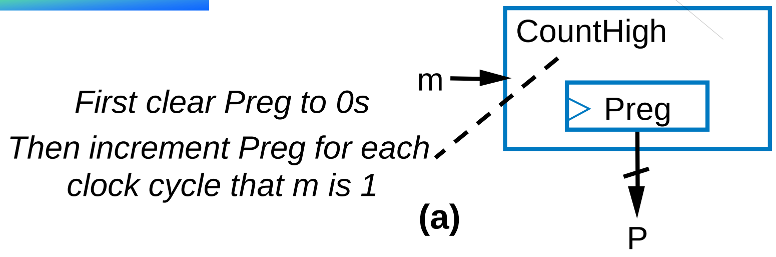
(**b**)

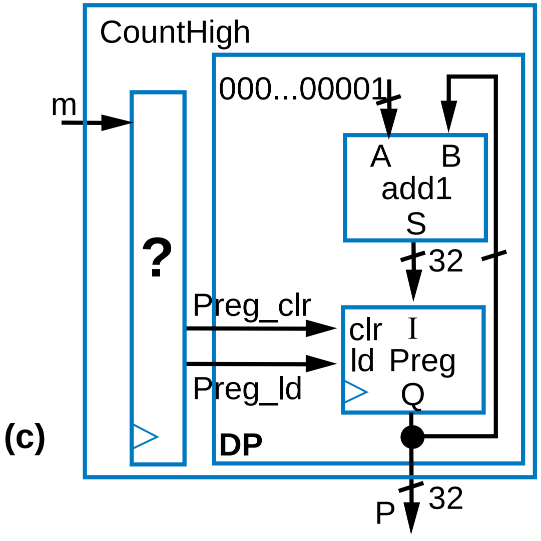
RTL Design Process

- Capture behavior
- Convert to circuit
  - Need target architecture
  - Datapath capable of HLSM's data operations
  - Controller to control datapath



Ctrl/DP Example for Earlier Cycles-High Counter



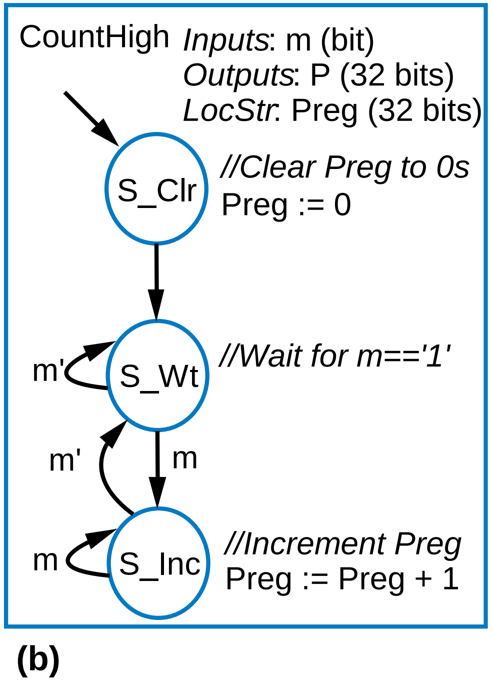


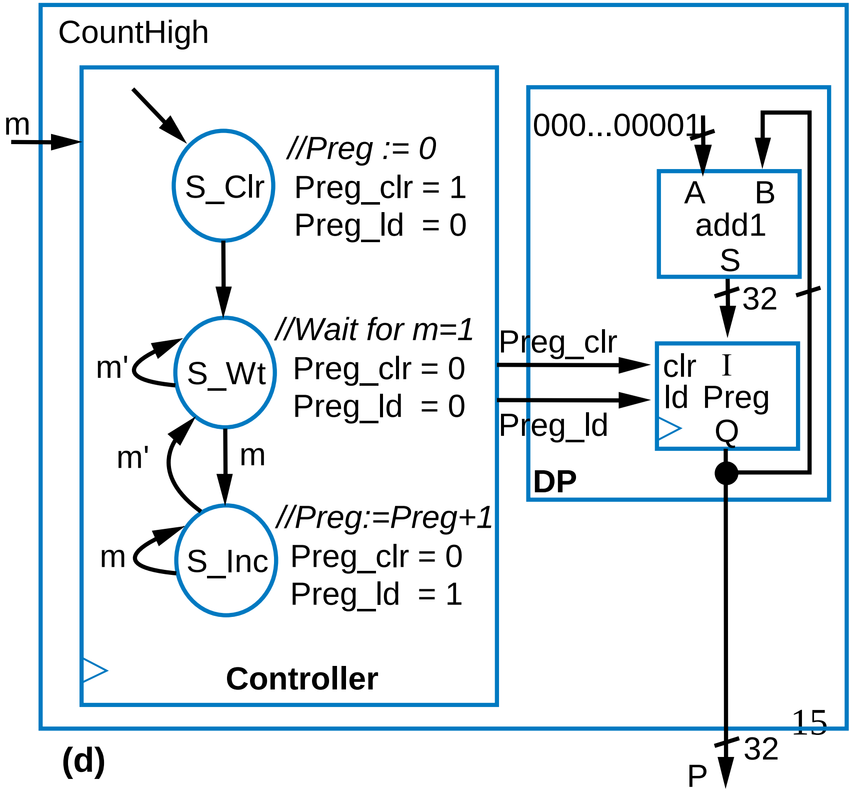
Create DP

Connect with controller

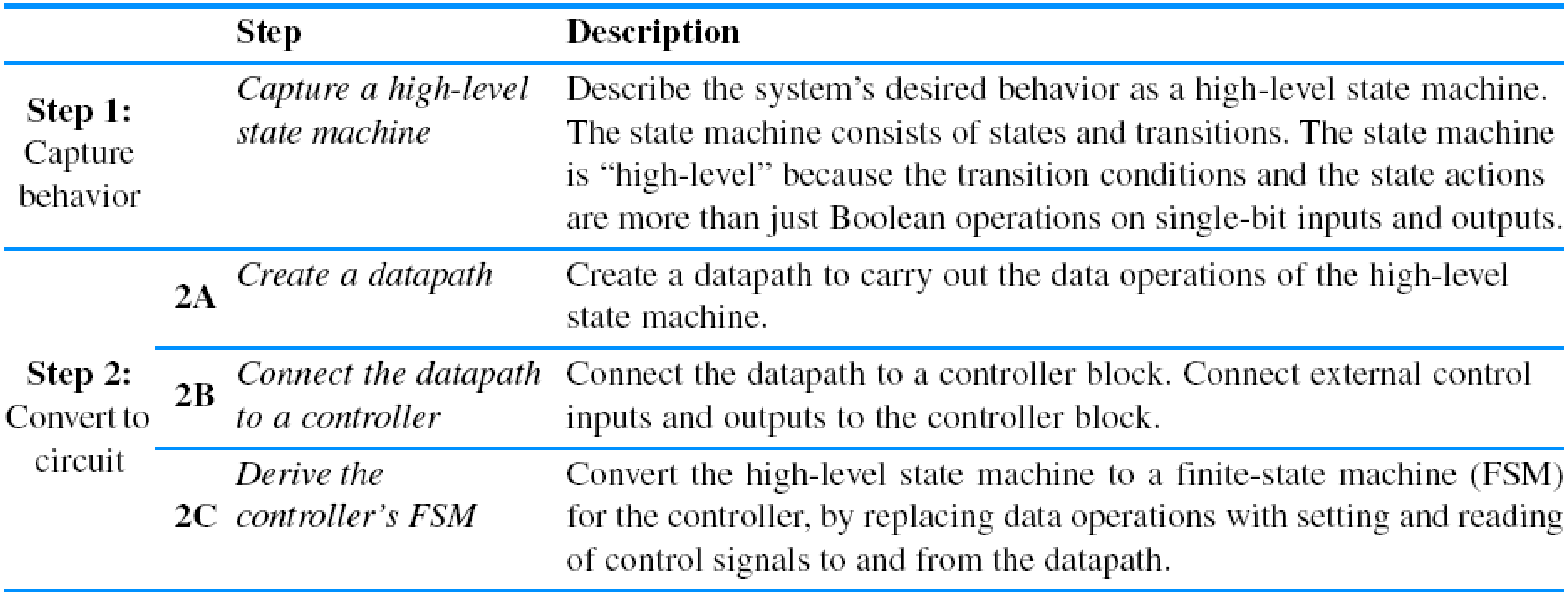
Derive controller

We created this HLSM earlier





RTL Design Process

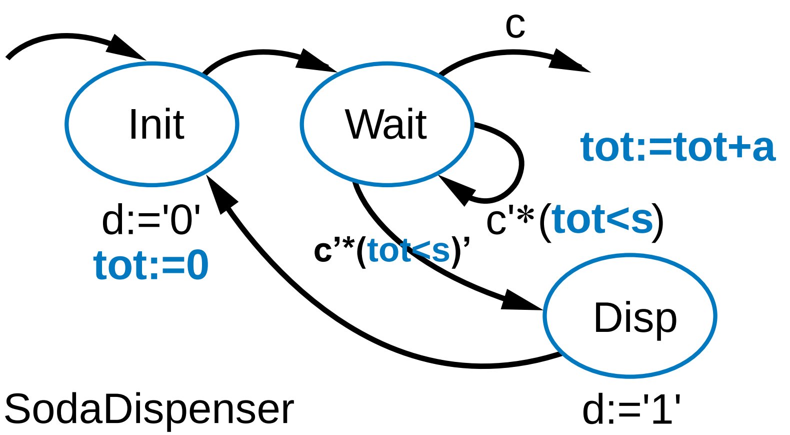


		Step	Description
Step 1: Capture behavior		*Capture a high-level state machine*	Describe the system's desired behavior as a high-level state machine. The state machine consists of states and transitions. The state machine is "high-level" because the transition conditions and the state actions are more than just Boolean operations on single-bit inputs and outputs.
Step 2: Convert to circuit	**2A**	*Create a datapath*	Create a datapath to carry out the data operations of the high-level state machine.
	2B	*Connect the datapath to a controller*	Connect the datapath to a controller block. Connect external control inputs and outputs to the controller block.
	2C	*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.

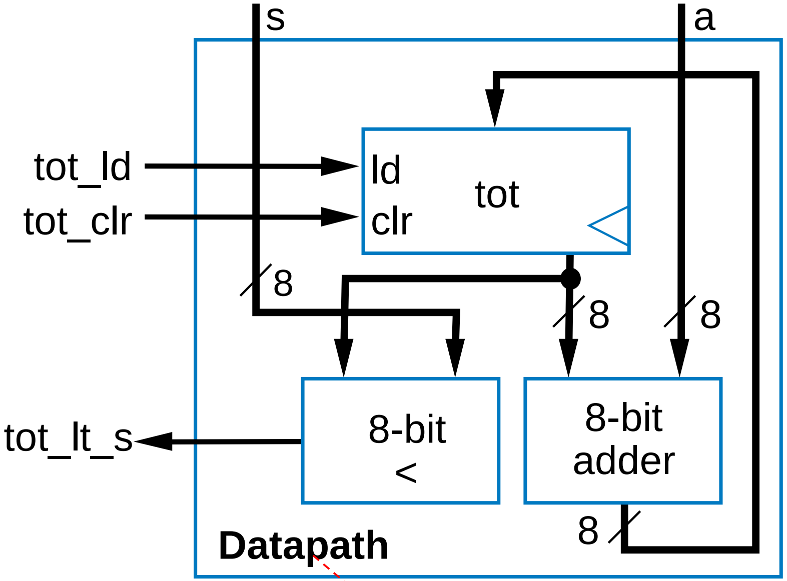
Example: Soda Dispenser from Earlier

- Quick overview example.
  More details of each step to come.

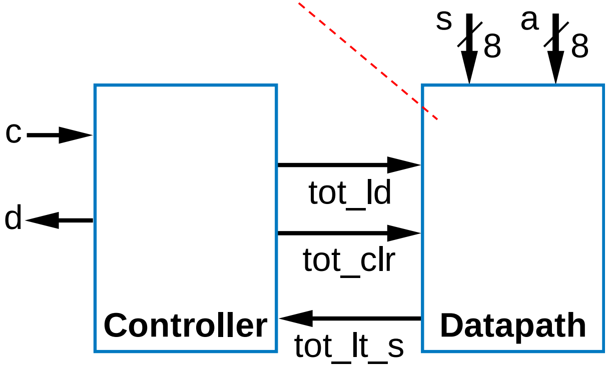
Inputs: c (bit), **a (8 bits), s (8 bits)**
Outputs: d (bit) // '1' dispenses soda
***Local storage*: tot (8 bits)**



Step 1



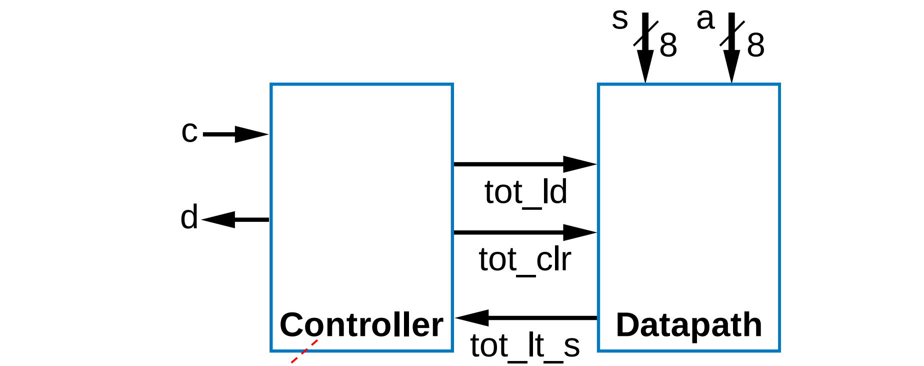
Step 2A



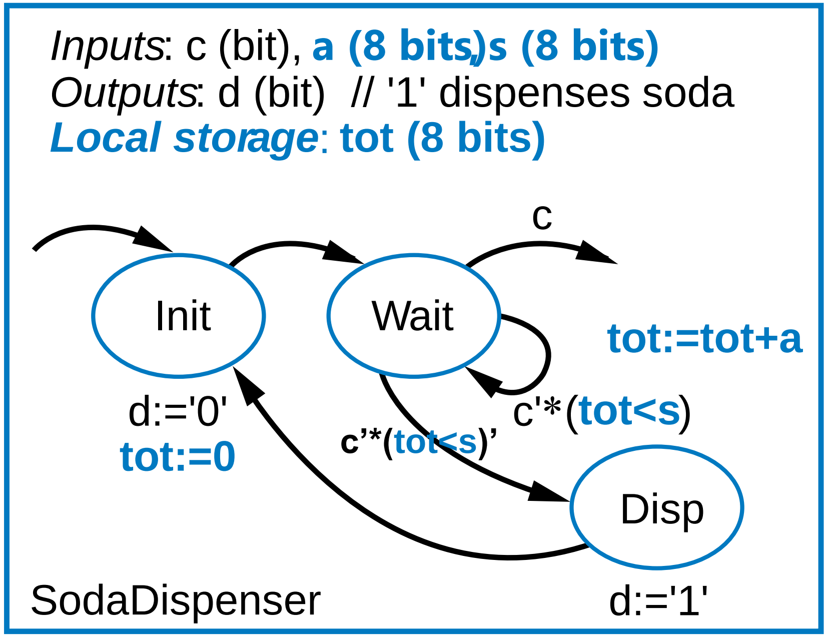
Step 2B

Example: Soda Dispenser

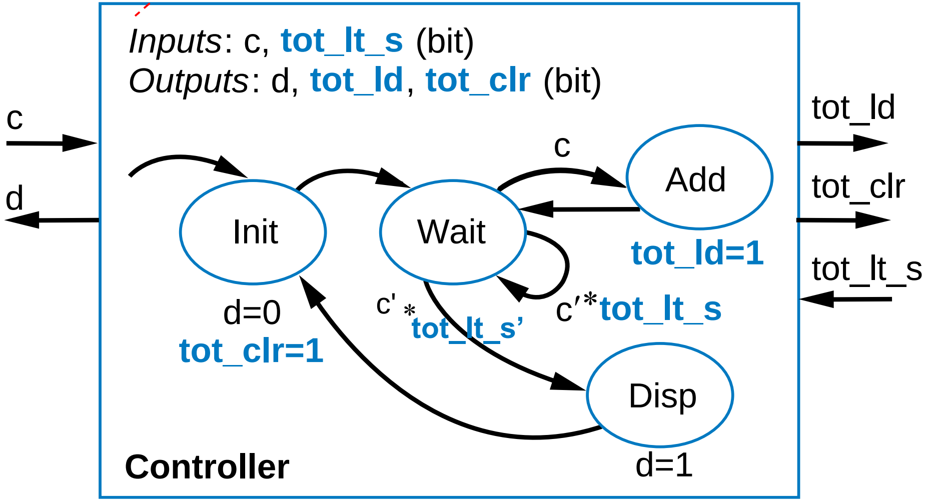
- Quick overview example.
  More details of each step to come.



Step 2B


Inputs: c (bit), **a (8 bits)**, **s (8 bits)**
Outputs: d (bit) // '1' dispenses soda
***Local storage*: tot (8 bits)**

Step 1

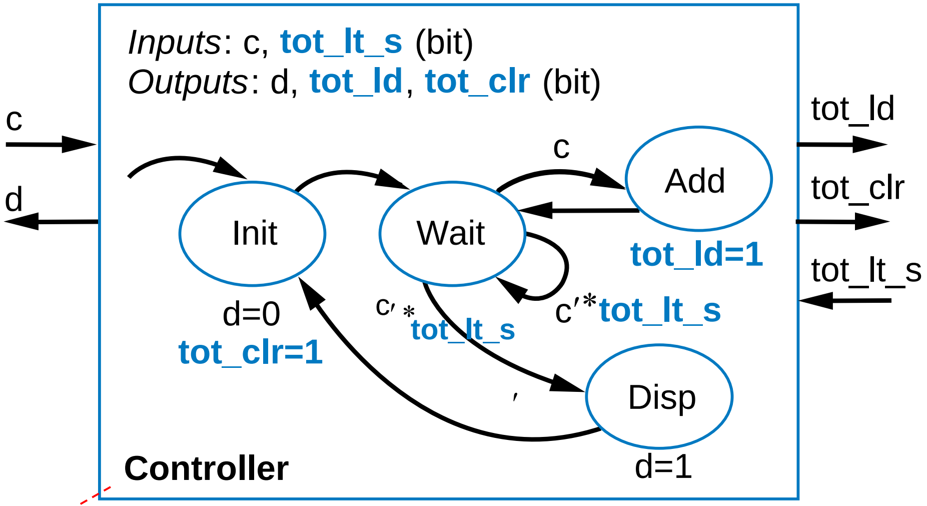

Inputs: c, **tot_lt_s** (bit)
Outputs: d, **tot_ld**, **tot_clr** (bit)

Step 2C

Example: Soda Dispenser

- Quick overview example.
  More details of each step to come.

	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
		• • •				• • •			



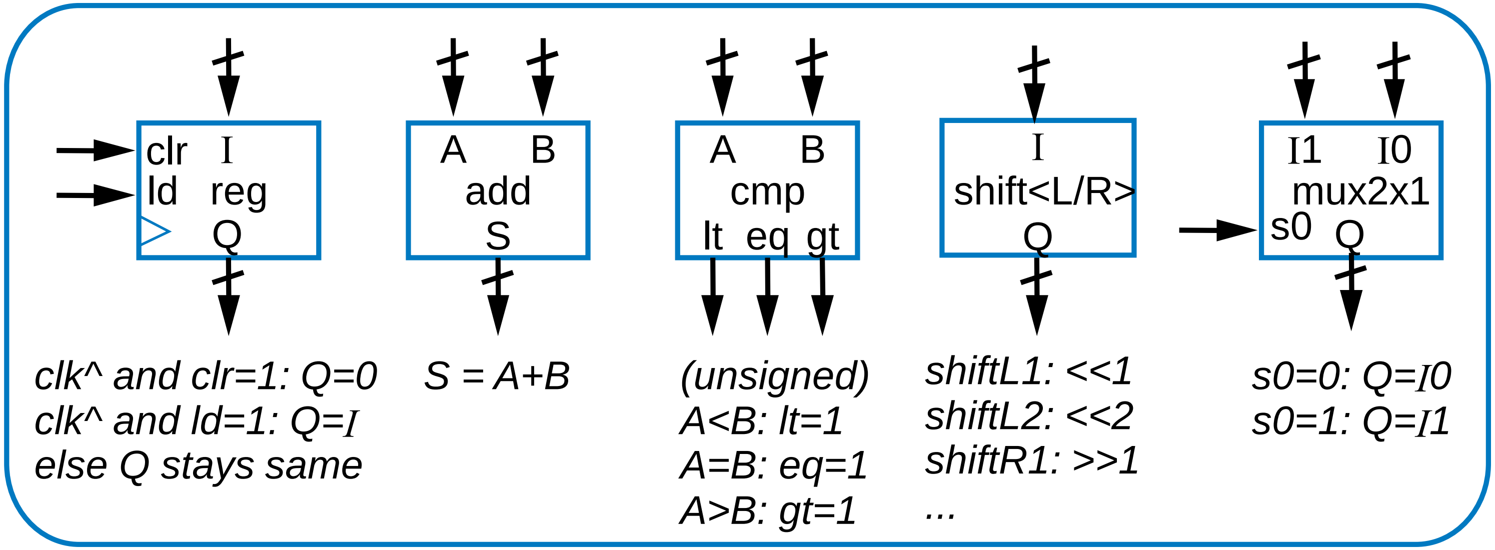
Step 2C

Use controller design process (Ch3) to complete the design

a

RTL Design Process—Step 2A: Create a datapath

- Sub-steps
  - HLSM data inputs/outputs → Datapath inputs/outputs.
  - HLSM local storage item → Instantiated register
    - "Instantiate": Add new component ("instance") to design
  - Each HLSM state action and transition condition data computation → Datapath components and connections
    - Also instantiate multiplexors as needed
- Need component library from which to choose



clk^ and clr=1: Q=0
clk^ and ld=1: Q=I
else Q stays same

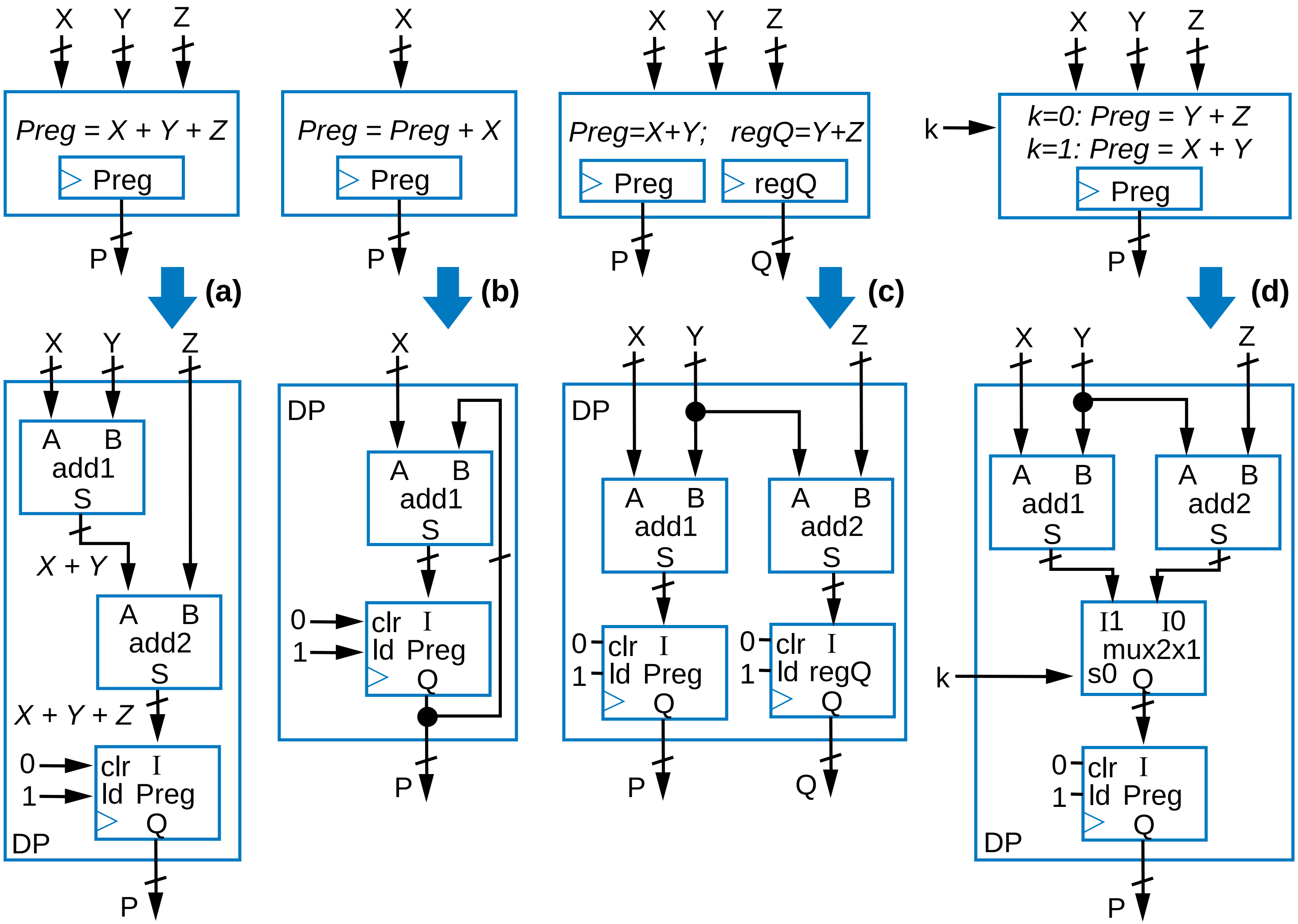
S = A+B

(unsigned)
A<B: lt=1
A=B: eq=1
A>B: gt=1

shiftL1: <<1
shiftL2: <<2
shiftR1: >>1
...

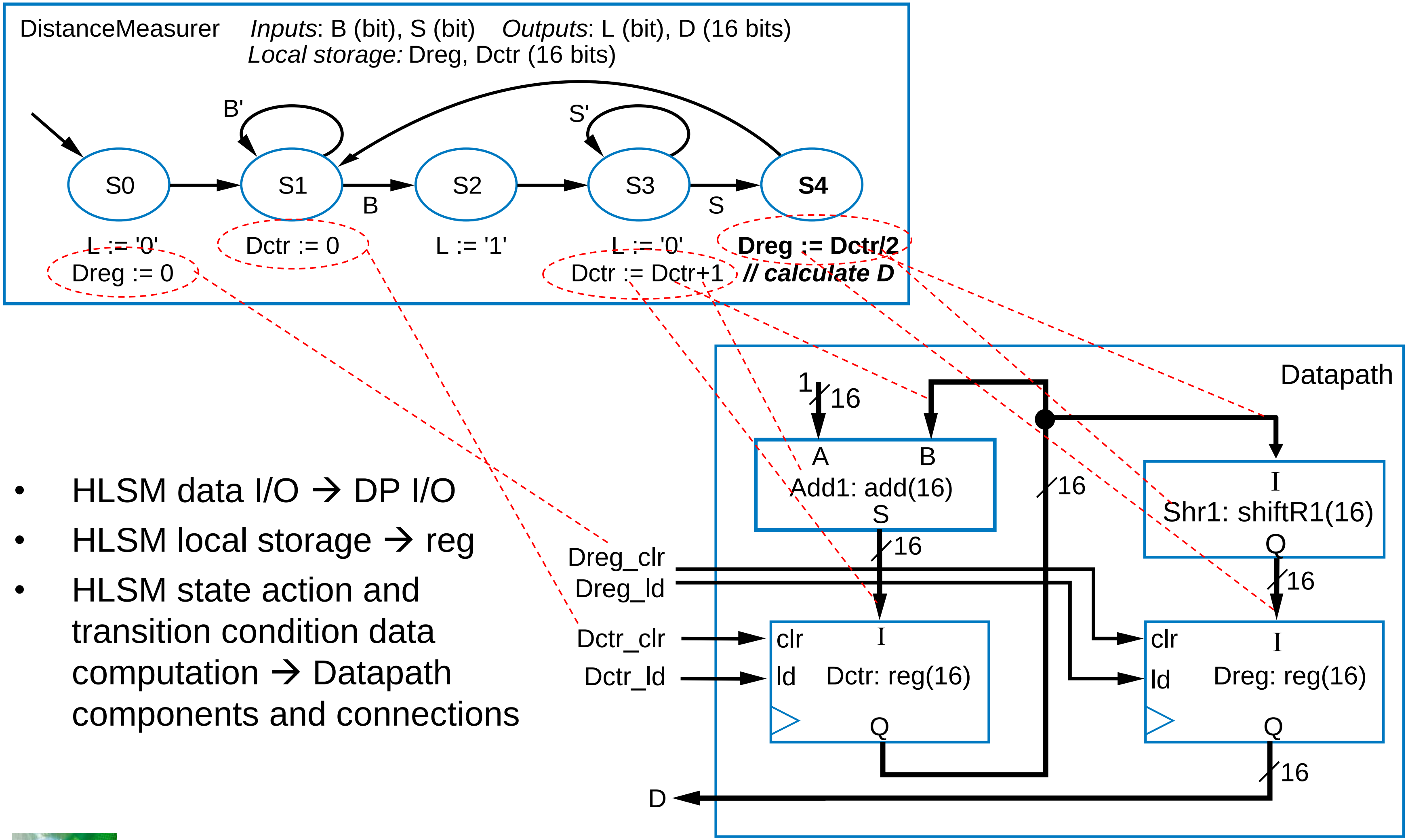
s0=0: Q=I0
s0=1: Q=I1

Step 2A: Create a Datapath—Simple Examples



a

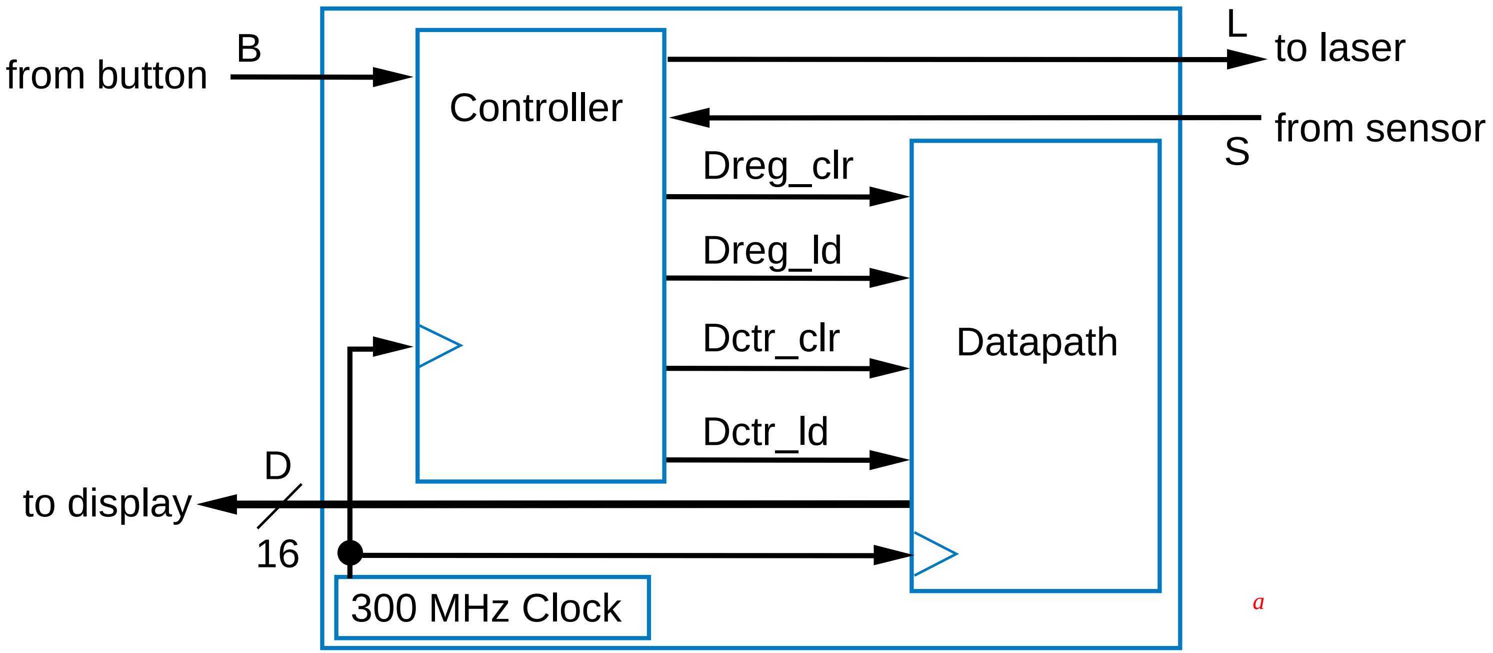
Laser-Based Distance Measurer—Step 2A: Create a Datapath



a

- HLSM data I/O → DP I/O
- HLSM local storage → reg
- HLSM state action and transition condition data computation → Datapath components and connections

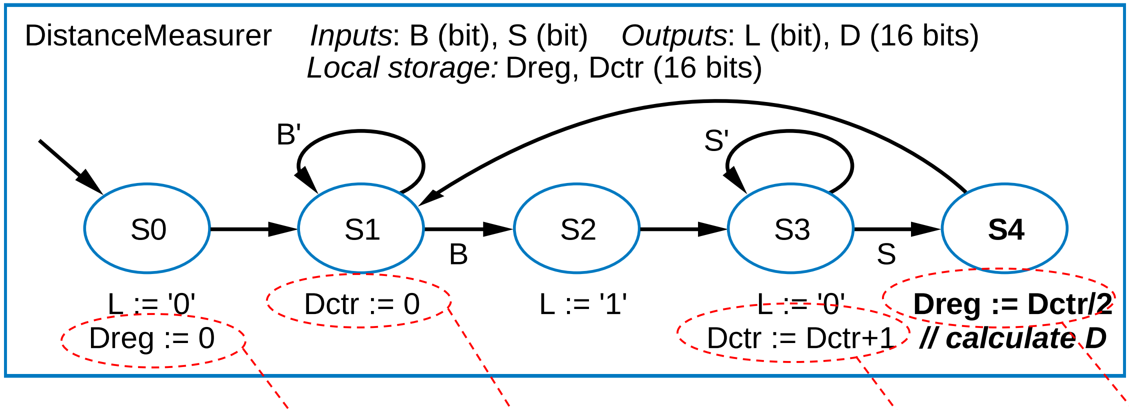
Laser-Based Distance Measurer—Step 2B: Connecting the Datapath to a Controller

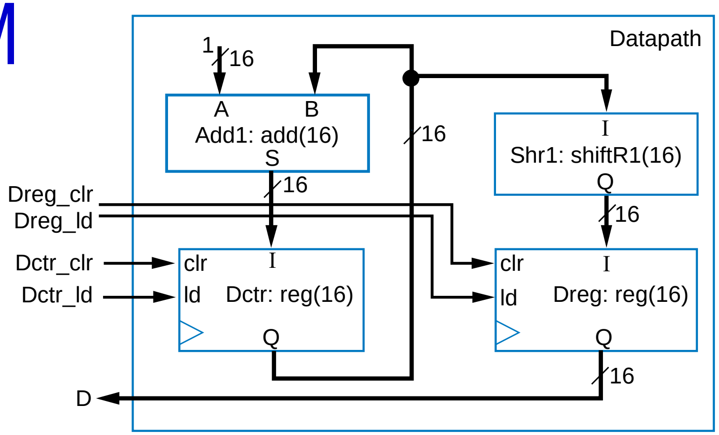


a

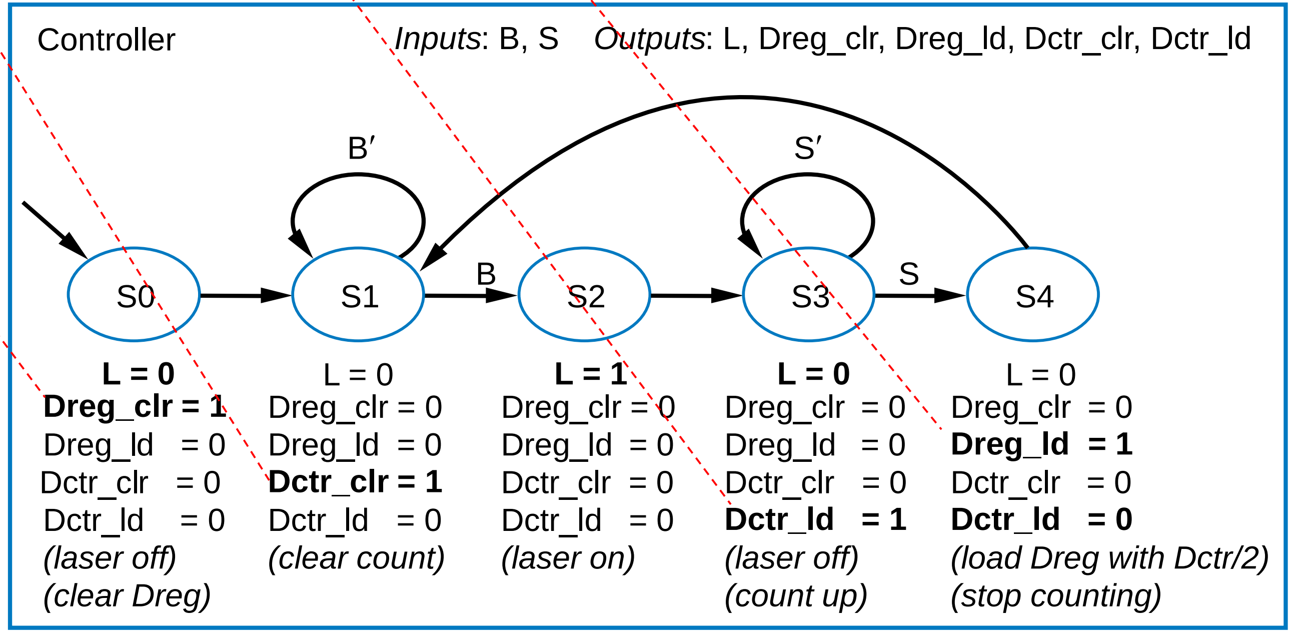
Laser-Based Distance Measurer—Step 2C: Derive the Controller FSM

HLSM


DistanceMeasurer *Inputs*: B (bit), S (bit) *Outputs*: L (bit), D (16 bits)
Local storage: Dreg, Dctr (16 bits)



- FSM has same states, transitions, and control I/O
- Achieve each HLSM data operation using datapath control signals in FSM


Controller *Inputs*: B, S *Outputs*: L, Dreg_clr, Dreg_ld, Dctr_clr, Dctr_ld

S0:
L = 0
Dreg_clr = 1
Dreg_ld = 0
Dctr_clr = 0
Dctr_ld = 0
(laser off)
(clear Dreg)

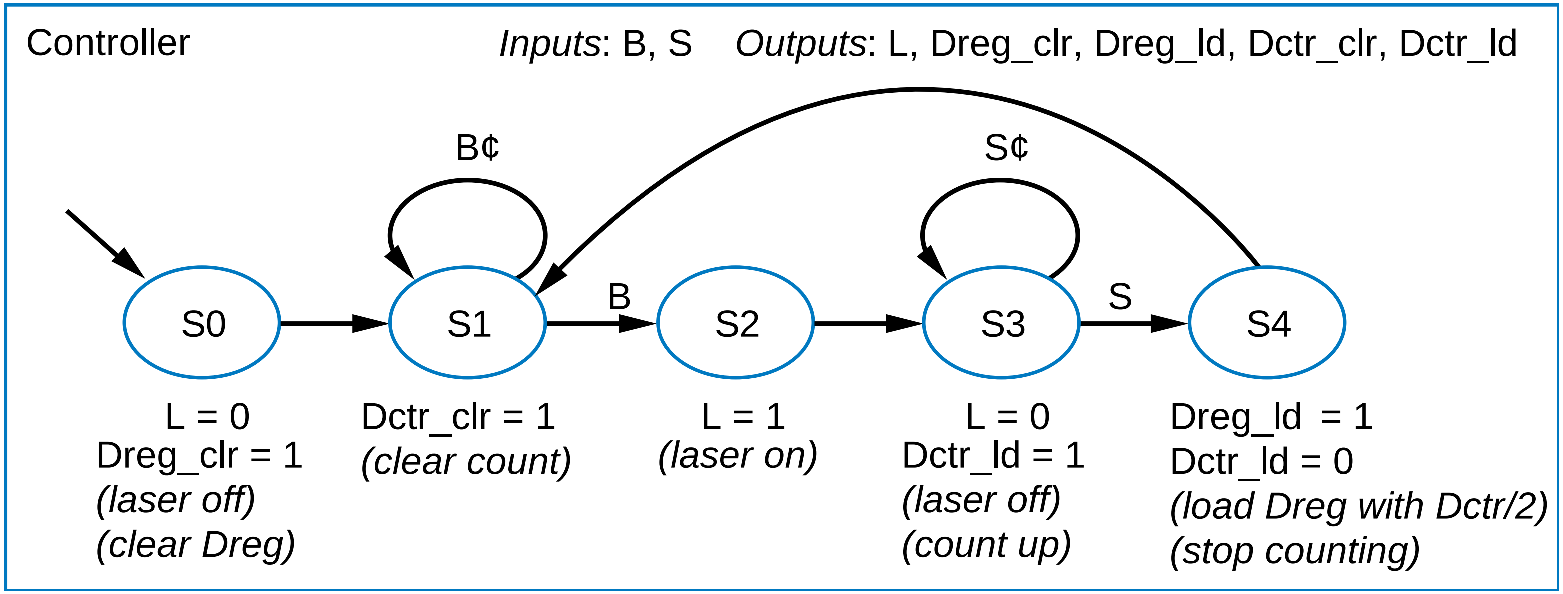
S1:
L = 0
Dreg_clr = 0
Dreg_ld = 0
Dctr_clr = 1
Dctr_ld = 0
(clear count)

S2:
L = 1
Dreg_clr = 0
Dreg_ld = 0
Dctr_clr = 0
Dctr_ld = 0
(laser on)

S3:
L = 0
Dreg_clr = 0
Dreg_ld = 0
Dctr_clr = 0
Dctr_ld = 1
(laser off)
(count up)

S4:
L = 0
Dreg_clr = 0
Dreg_ld = 1
Dctr_clr = 0
Dctr_ld = 0
(load Dreg with Dctr/2)
(stop counting)

Laser-Based Distance Measurer—Step 2C: Derive the Controller FSM

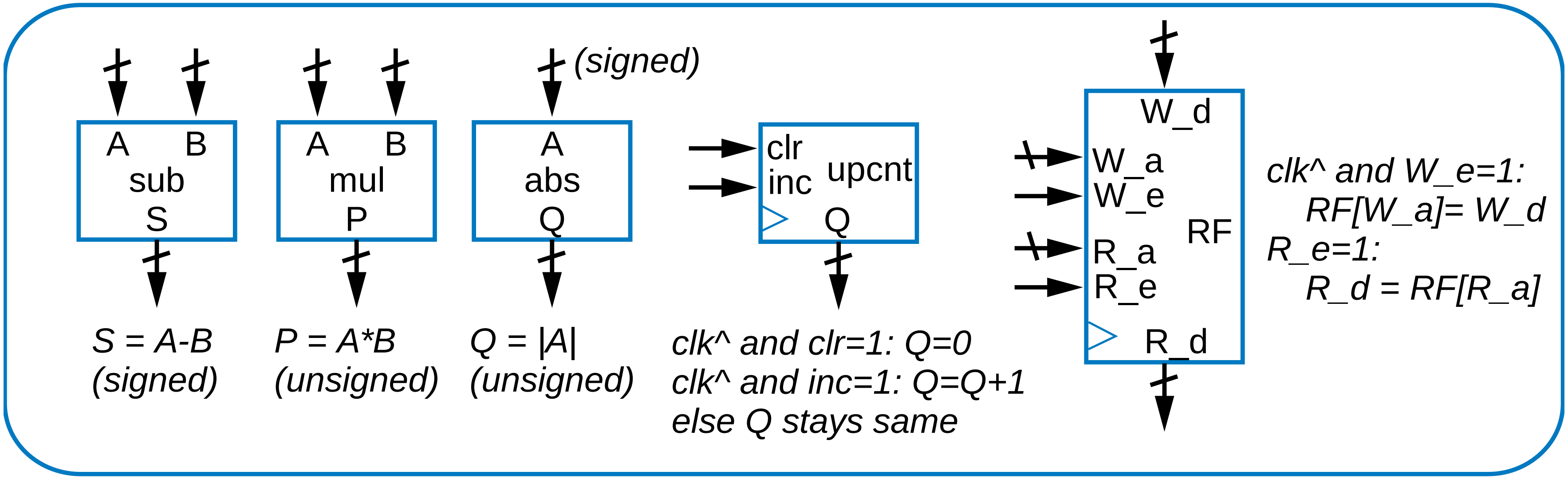


- Same FSM, using convention of unassigned outputs implicitly assigned 0

Some assignments to 0 still shown, due to their importance in understanding desired controller behavior

More RTL Design

- Additional datapath components



S = A-B (signed)

P = A*B (unsigned)

Q = |A| (unsigned)

clk^ and clr=1: Q=0
clk^ and inc=1: Q=Q+1
else Q stays same

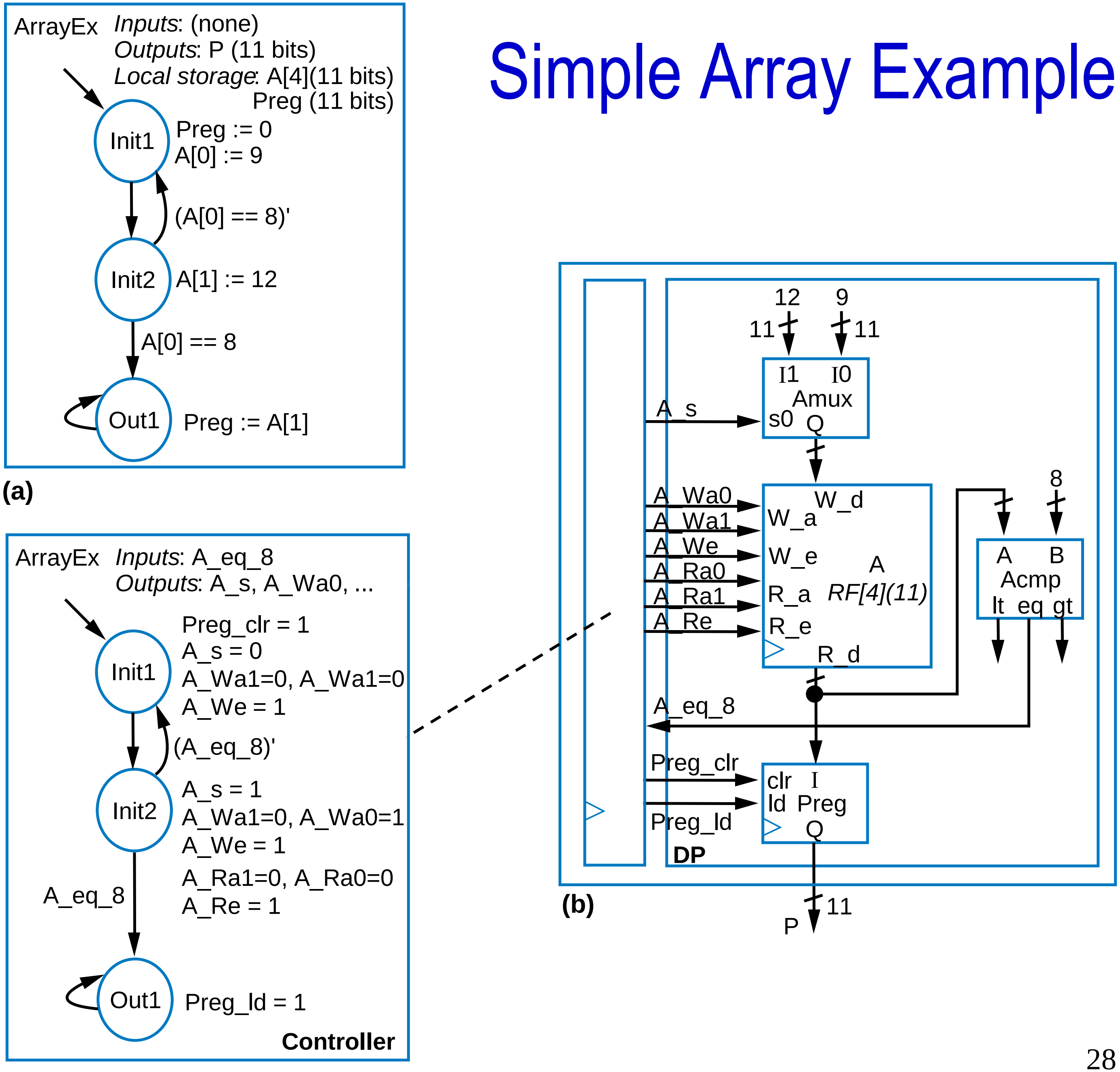
clk^ and W_e=1:
RF[W_a]= W_d
R_e=1:
R_d = RF[R_a]

RTL Design Involving Register File or Memory

- HLSM *array*: Ordered list of items
  - Ex: Local storage: A[4](8-bit) – 4 8-bit items
  - Accessed using notation "A[i]", i is *index*
  - A[0] := 9;   A[1] := 8;   A[2] := 7;   A[3] := 22
    - Array contents now: <9, 8, 7, 22>
    - X := A[1] will set X to 8
    - Note: First element's index is 0
- Array can be mapped to instantiated register file or memory

Simple Array Example

a



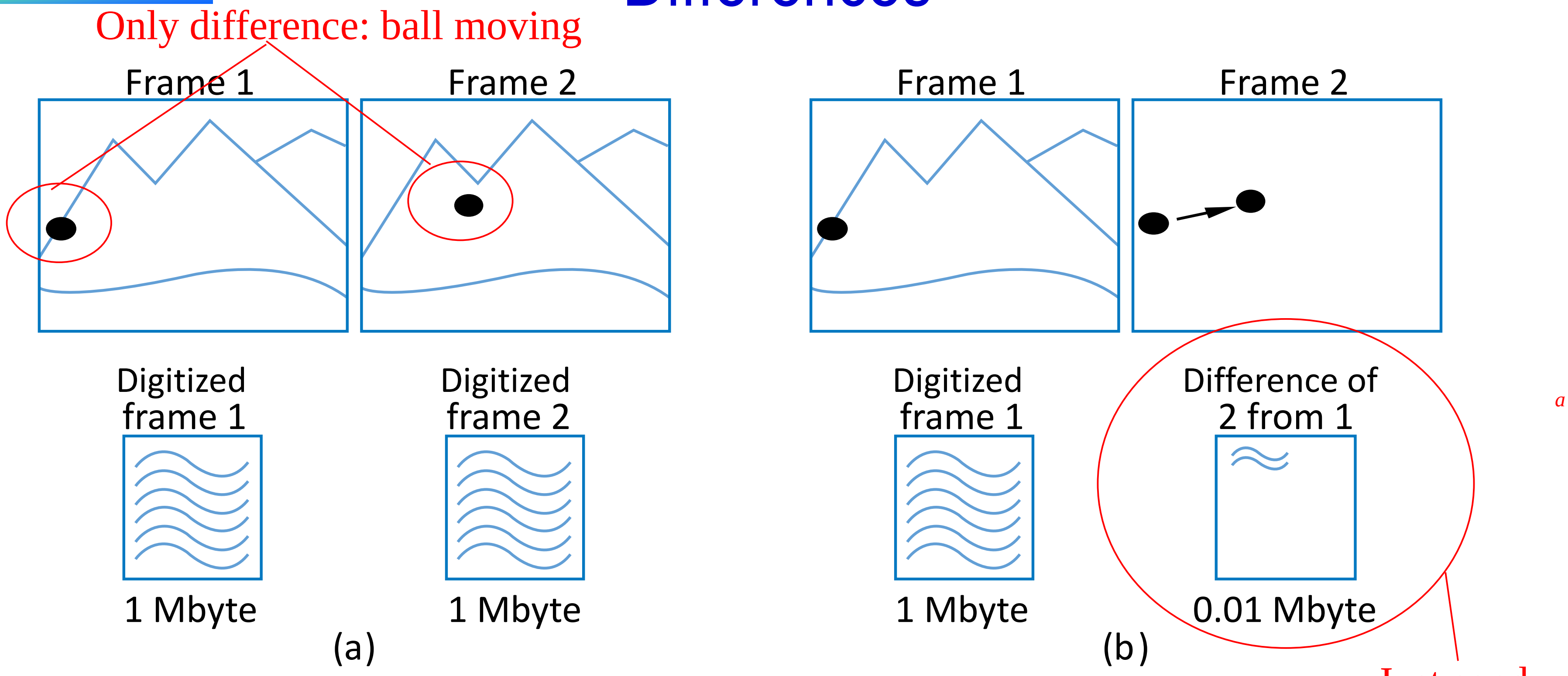
(a)

(b)

(c)

a

RTL Example: Video Compression – Sum of Absolute Differences

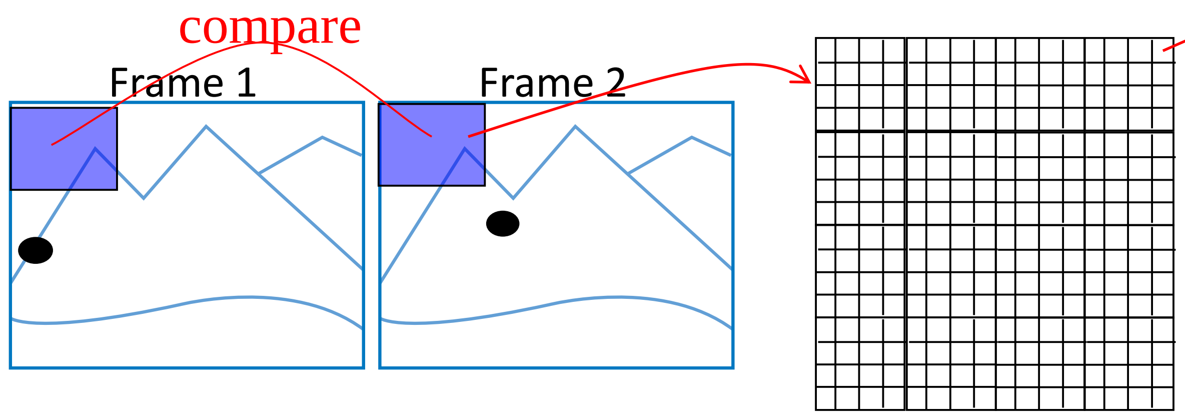


Only difference: ball moving

Just send difference

- 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

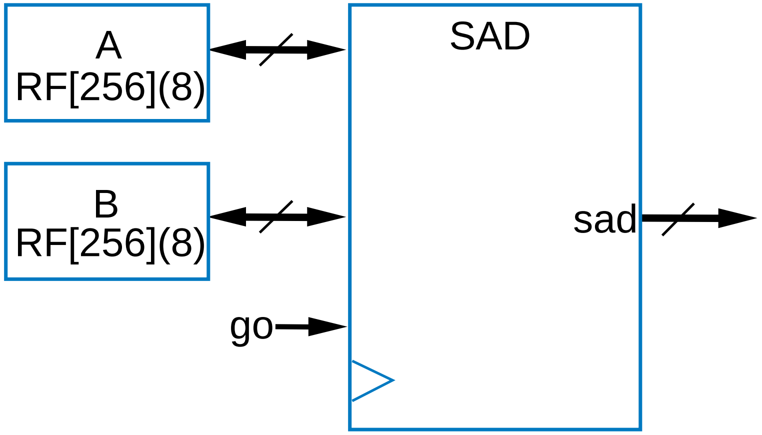
RTL Example: Video Compression – Sum of Absolute Differences



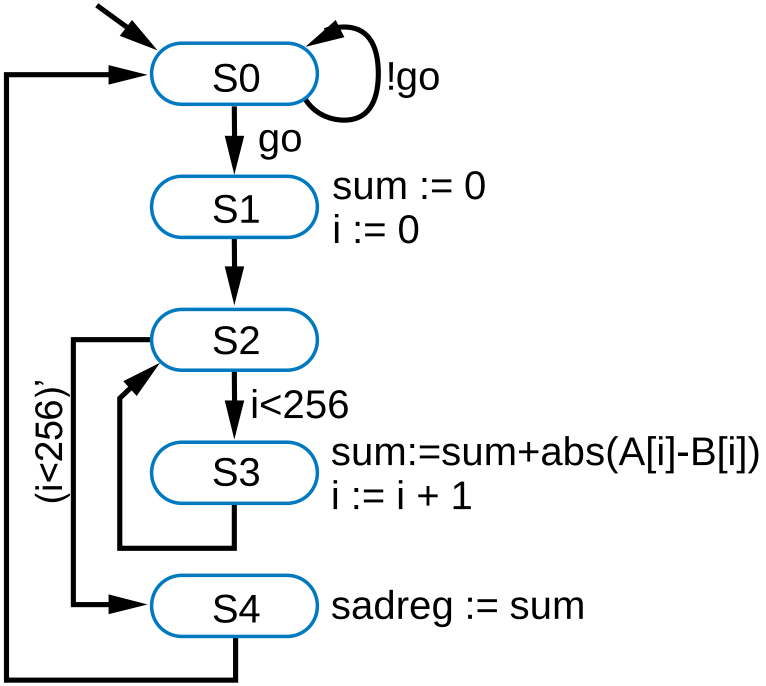
Each is a pixel, assume represented as 1 byte (actually, a color picture might have 3 bytes per pixel, for intensity of red, green, and blue components of pixel)

- 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)

Array Example: Video Compression—Sum-of-Absolute Differences



Inputs: A, B [256](8 bits); go (bit)
Outputs: sad (32 bits)
Local storage: sum, sadreg (32 bits); i (9 bits)

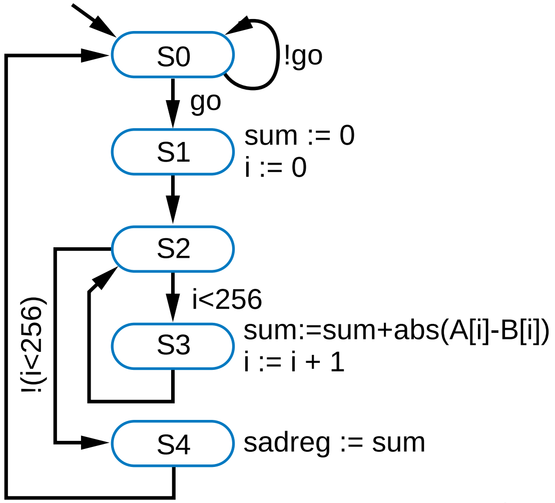


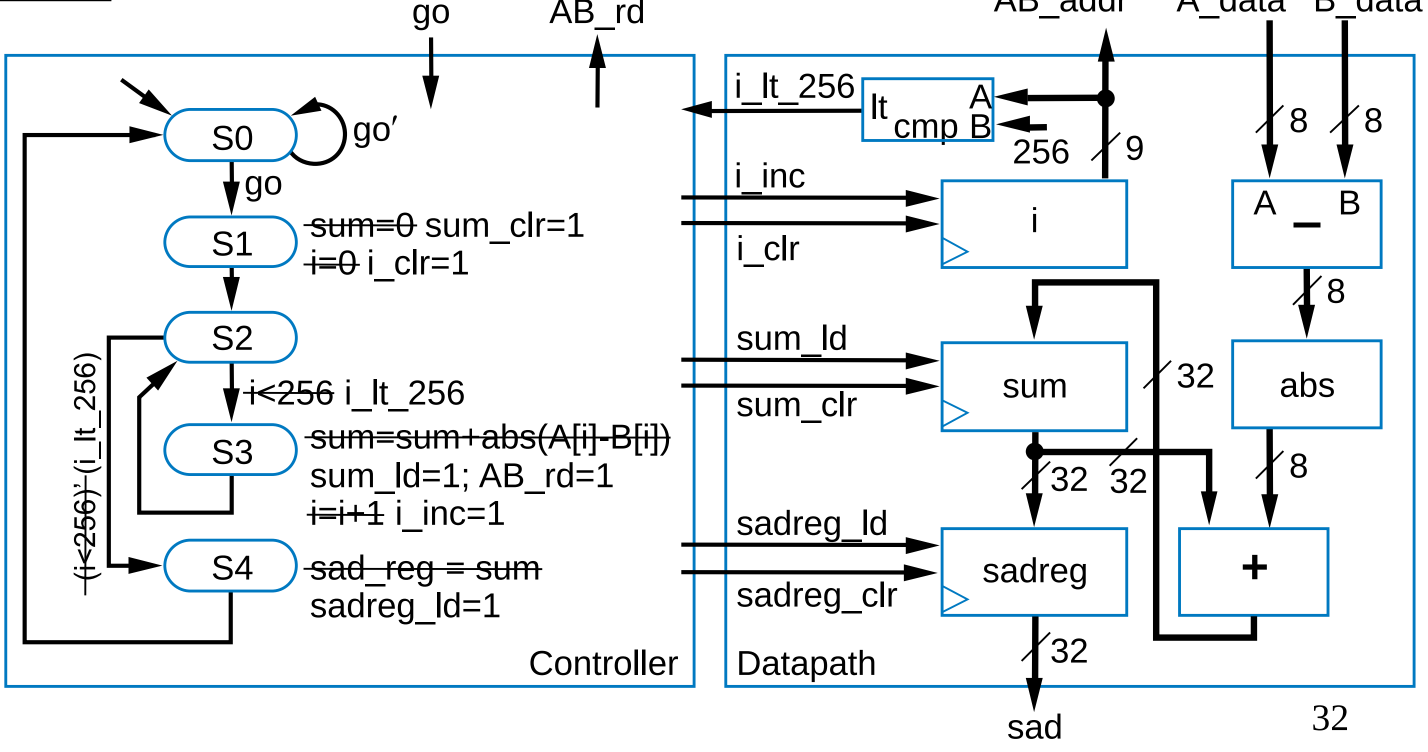
(**b**)

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

Array Example: Video Compression—Sum-of-Absolute Differences

Inputs: A, B [256](8 bits); go (bit)
Outputs: sad (32 bits)
Local storage: sum, sadreg (32 bits); i (9 bits)

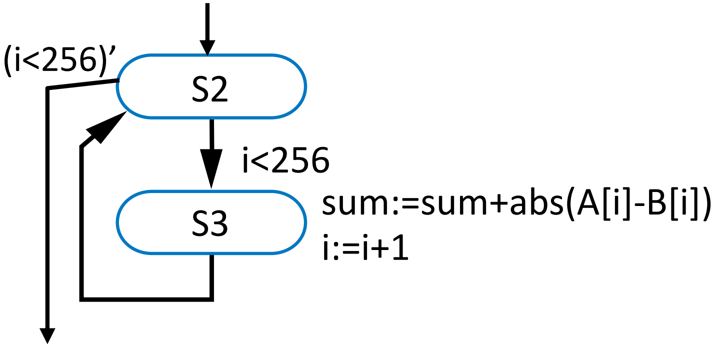




a

Circuit vs. Microprocessor

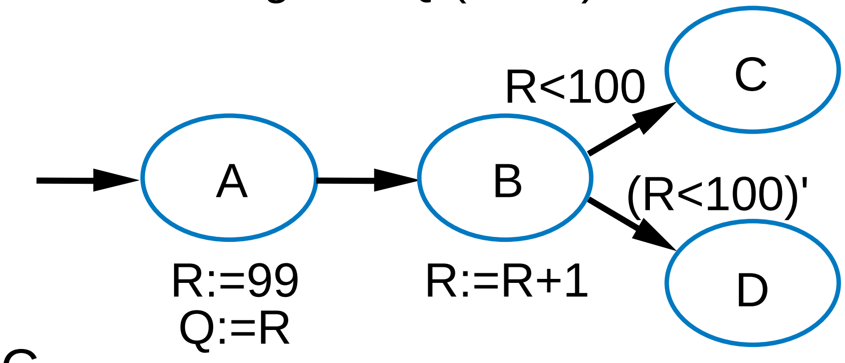
- Circuit: Two states (**S2** & **S3**) for each *i*, 256 *i*'s→ 512 clock cycles
- Microprocessor: Loop (*for i = 1 to 256*), but for each *i*, must move memory to local registers, subtract, compute absolute value, add to sum, increment *i* – say 6 cycles per array item → 256*6 = 1536 cycles
- Circuit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

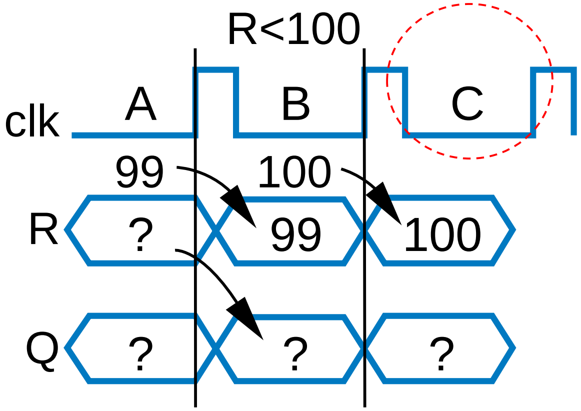


Common RTL Design Pitfall Involving Storage Updates

- Questions
  - Value of Q after state A?
  - Final state is C or D?
- Answers
  - Q is NOT 99 after state A
  - Q is 99 in state B, so final state is C
  - Storage update actions in state occur *simultaneously* on *next* clock edge
    - Thus, order actions are written is irrelevant
    - A's actions same if:
      - Q:=R   R:=99   or
      - R:=99   Q:=R

Local storage: R, Q (8 bits)

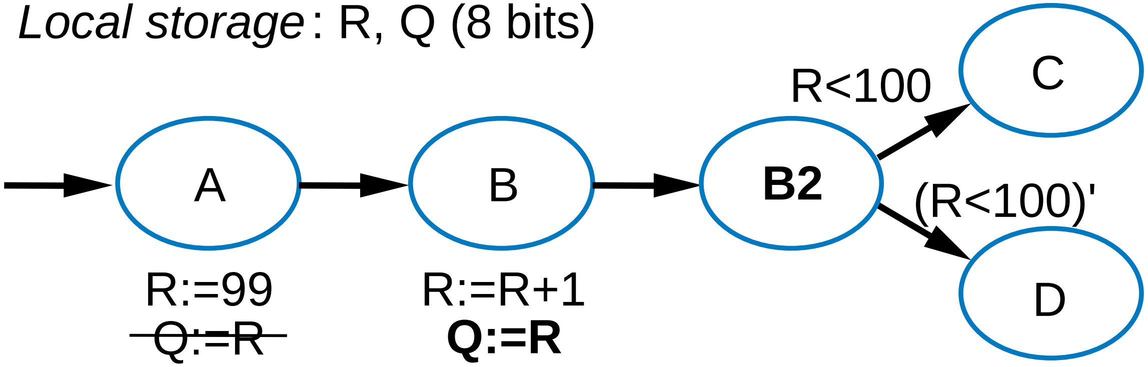


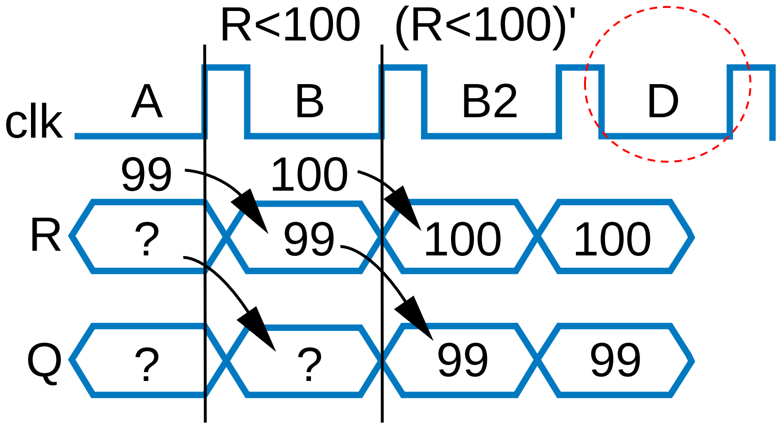


Common RTL Design Pitfall Involving Storage Updates

- New HLSM using extra state so read of R occurs after write of R

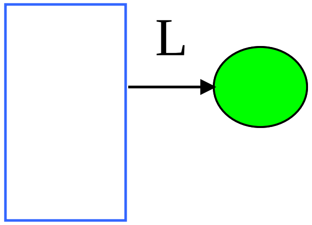
Local storage: R, Q (8 bits)

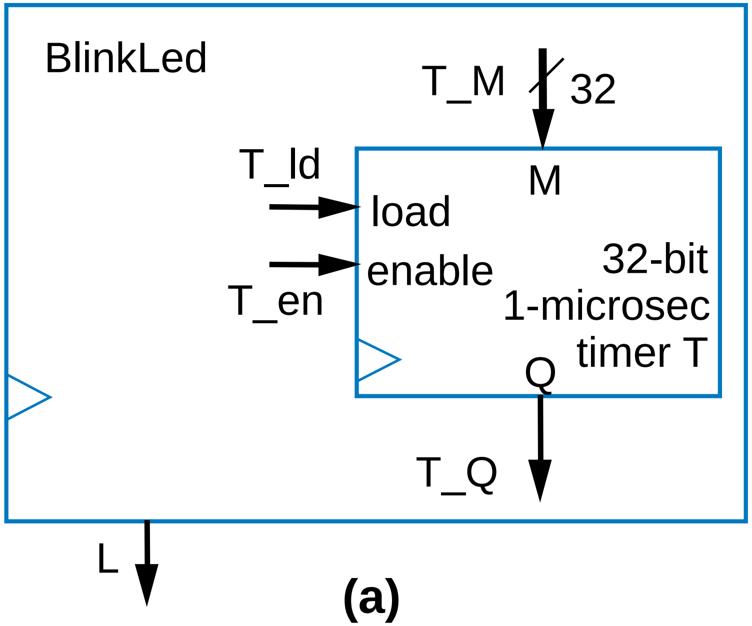




RTL Design Involving a Timer

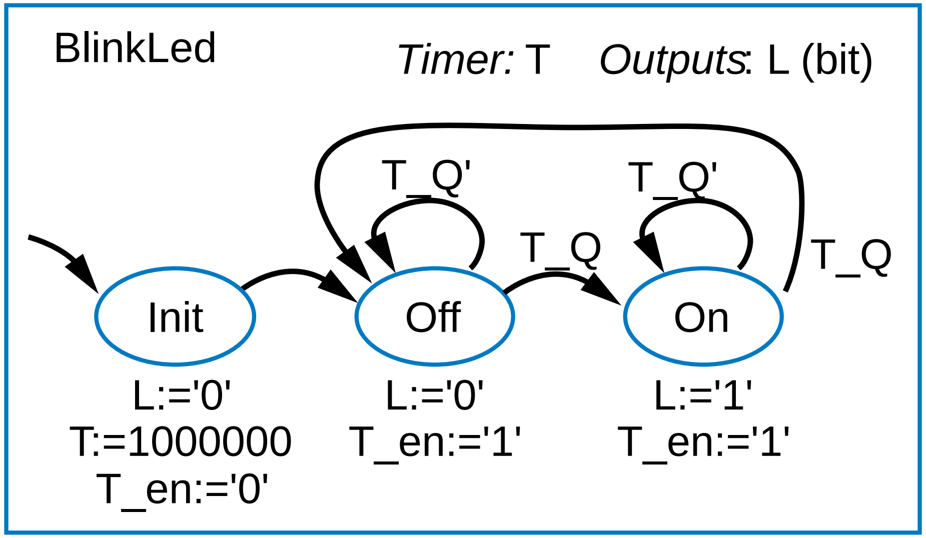
- Commonly need explicit time intervals
  - Ex: Repeatedly blink LED on 1 second, off 1 second
- Pre-instantiate timer that HLSM can then use





(a)

Pre-instantiated timer

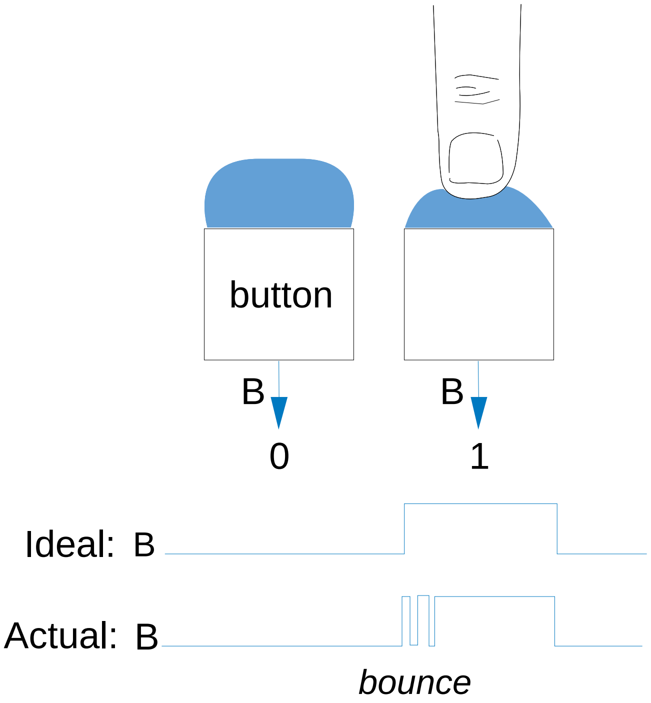


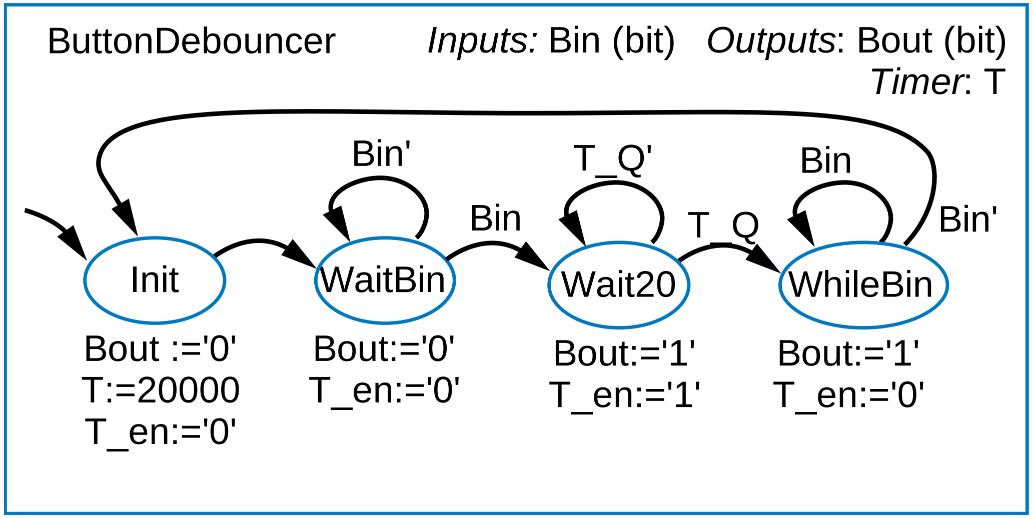
(b)

HLSM making use of timer

Button Debouncing

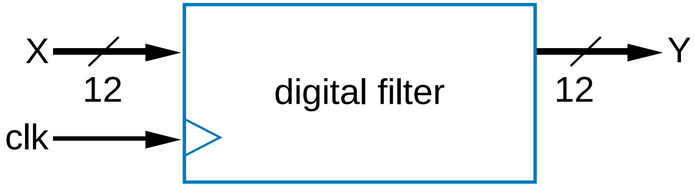
- Press button
  - Ideally, output changes to 1
  - Actually, output bounces
    - Due to mechanical reasons
    - Like ball bouncing when dropped to floor
- Digital circuit can convert actual signal closer to ideal signal





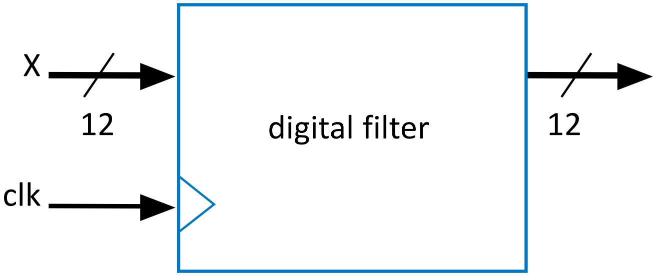
Data Dominated RTL Design Example

- Data dominated design: Extensive DP, simple controller
- Control dominated design: Complex controller, simple DP
- Example: Filter
  - Converts digital input stream to new digital output stream
  - Ex: Remove noise
    - 180, 180, 181, 180, 240, 180, 181
    - 240 is probably noise, filter might replace by 181
  - Simple filter: Output average of last *N* values
    - Small *N*: less filtering
    - Large *N*: more filtering, but less sharp output

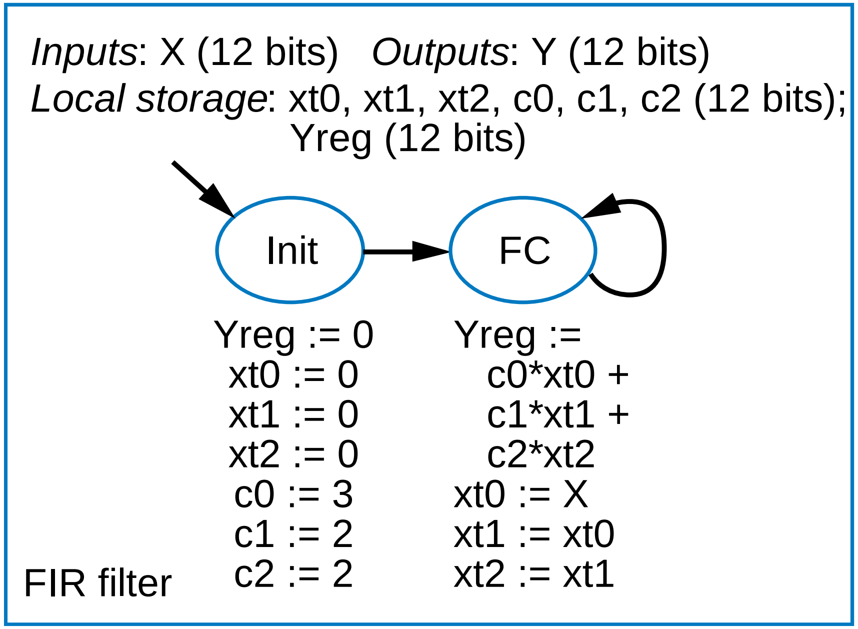


Data Dominated RTL Design Example: FIR Filter

- 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 HLSM
    - Very simple states/transitions



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



Inputs: X (12 bits)   *Outputs*: Y (12 bits)
Local storage: xt0, xt1, xt2, c0, c1, c2 (12 bits);
Yreg (12 bits)

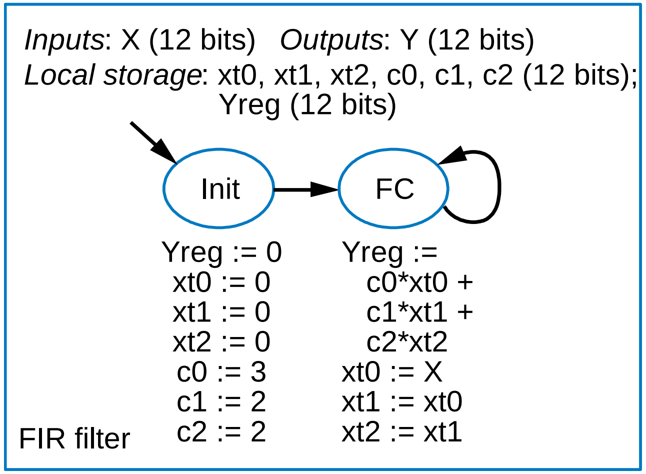
Init:
Yreg := 0
xt0 := 0
xt1 := 0
xt2 := 0
c0 := 3
c1 := 2
c2 := 2

FC:
Yreg :=
c0*xt0 +
c1*xt1 +
c2*xt2
xt0 := X
xt1 := xt0
xt2 := xt1

FIR filter

Assume constants set to 3, 2, and 2

FIR Filter



Inputs: X (12 bits)   *Outputs*: Y (12 bits)

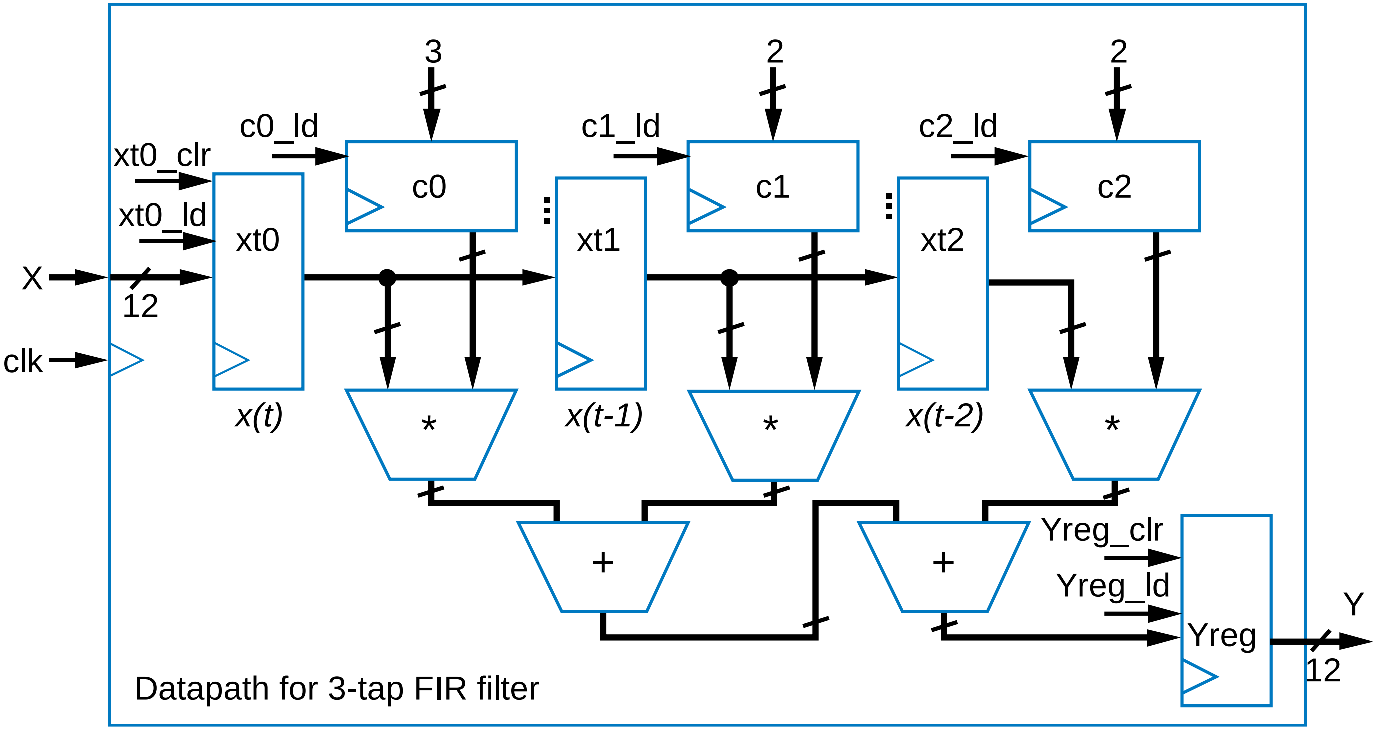
Local storage: xt0, xt1, xt2, c0, c1, c2 (12 bits);
Yreg (12 bits)

Init:
Yreg := 0
xt0 := 0
xt1 := 0
xt2 := 0
c0 := 3
c1 := 2
c2 := 2

FC:
Yreg :=
c0*xt0 +
c1*xt1 +
c2*xt2
xt0 := X
xt1 := xt0
xt2 := xt1

FIR filter

- Step 2A: Create datapath
- Step 2B: Connect Ctrlr/DP (as earlier examples)
- Step 2C: Derive FSM
  - Set clr and ld lines appropriately



Datapath for 3-tap FIR filter

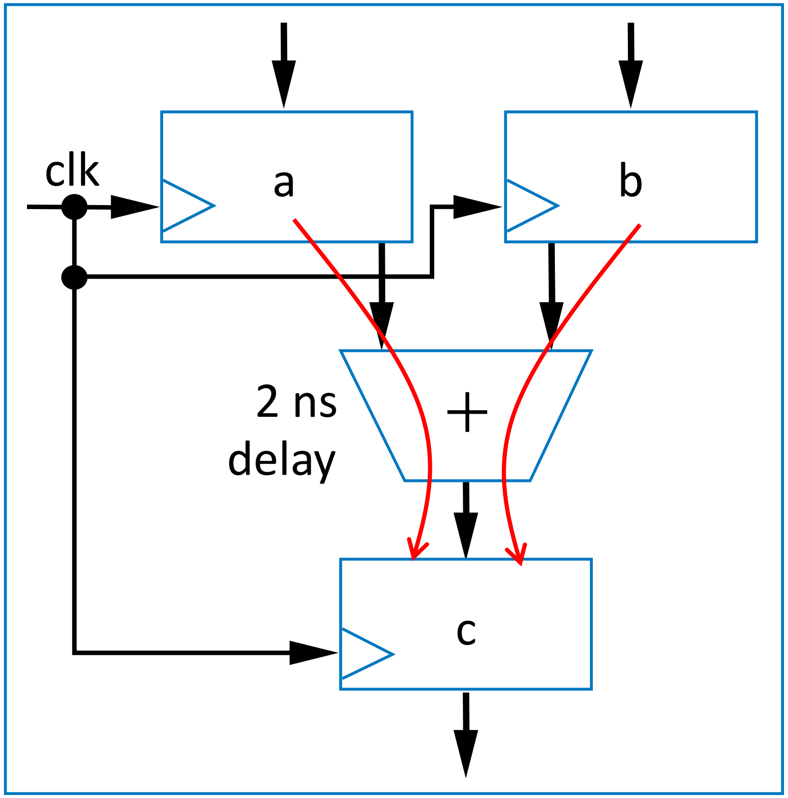
Circuit vs. Microprocessor

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

- Comparing the FIR circuit to microprocessor instructions
  - Microprocessor
    - 100-tap filter: 100 multiplications, 100 additions. Say 2 instructions per multiplication, 2 per addition. Say 10 ns per instruction.
    - (100*2 + 100*2)*10 = 4000 ns
  - Circuit
    - Assume adder has 2 ns delay, multiplier has 20 ns delay
    - Longest path goes through one multiplier and two adders
      - 20 + 2 + 2 = 24 ns delay
    - 100-tap filter, following design on previous slide, would have about a 34 ns delay: 1 multiplier and 7 adders on longest path
  - Circuit is more than 100 times faster (4000/34). Wow.

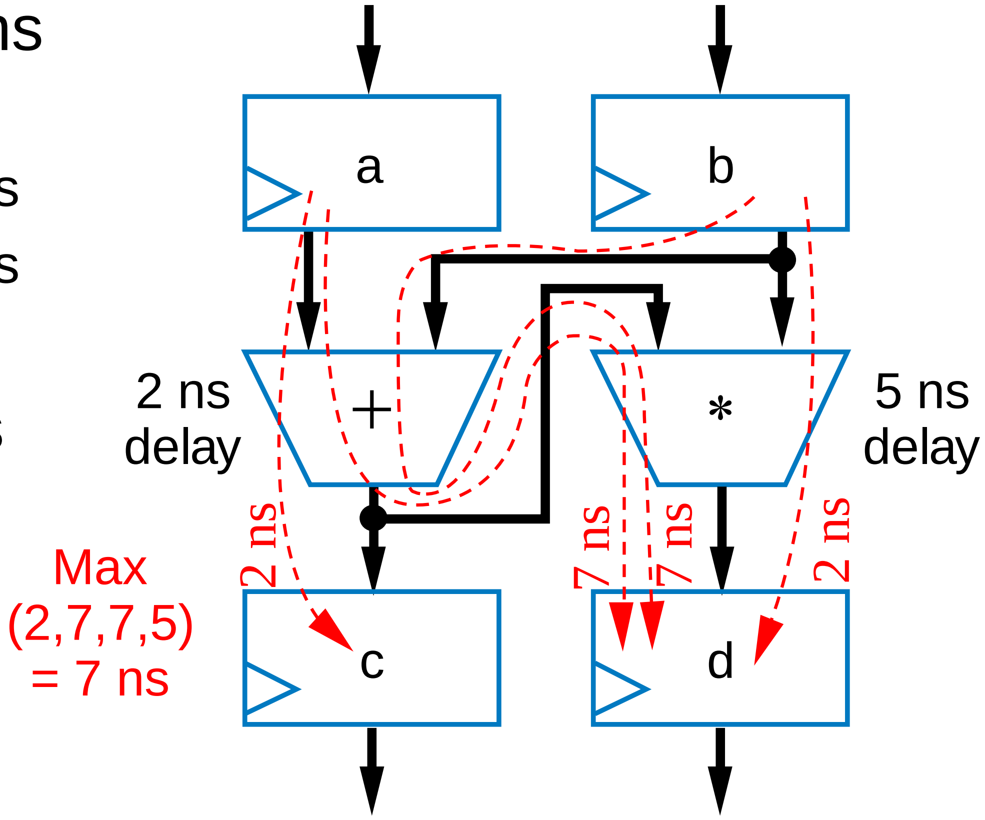
Determining Clock Frequency

- 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

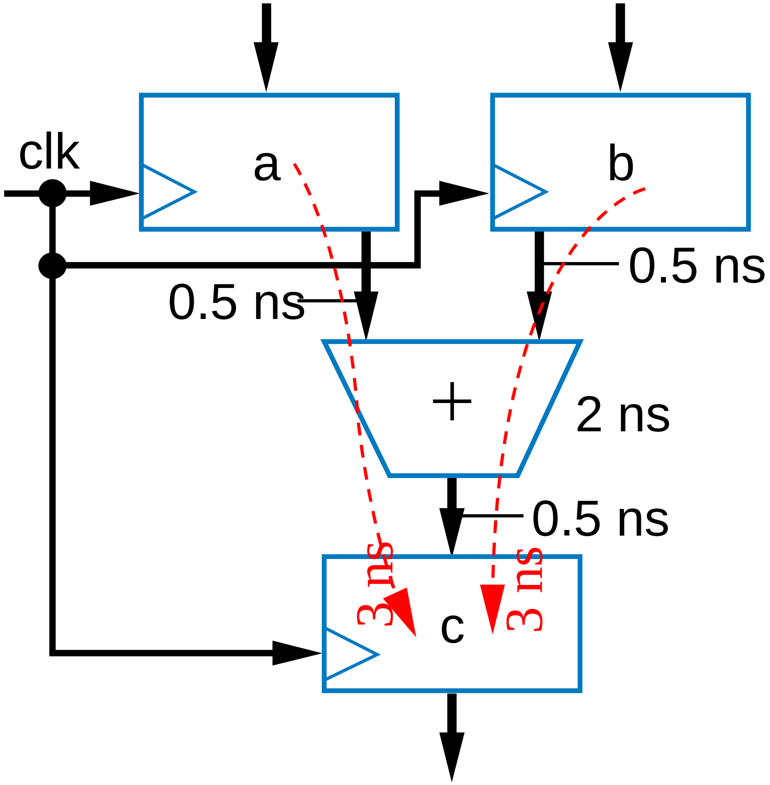
- Example shows four paths
  - a to c through +: 2 ns
  - a to d through + and *: 7 ns
  - b to d through + and *: 7 ns
  - b to d through *: 5 ns
- Longest path is thus 7 ns
- Fastest frequency
  - 1 / 7 ns = 142 MHz



Max (2,7,7,5) = 7 ns

Critical Path Considering Wire Delays

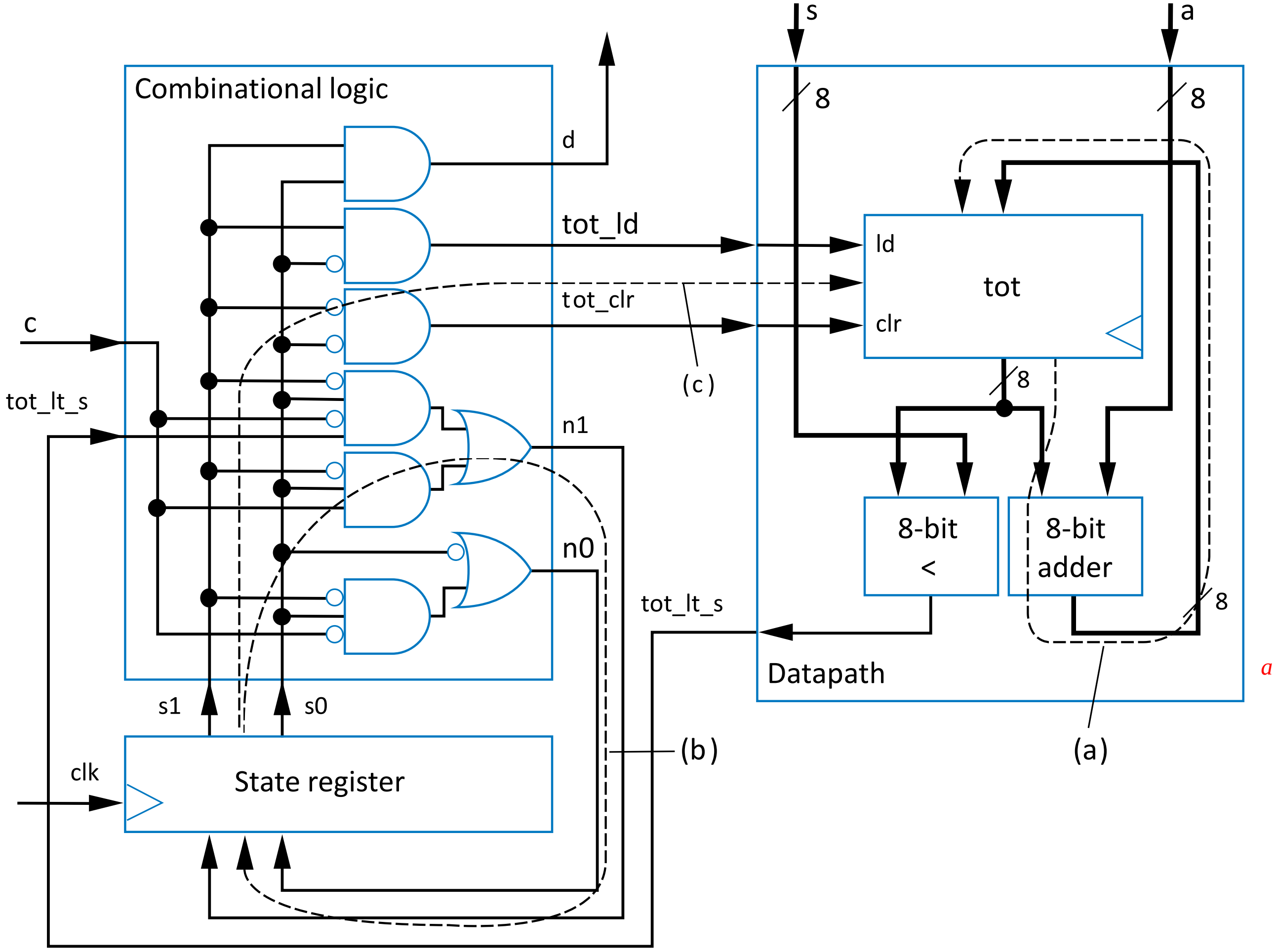
- Real wires have delay too
  - Must include in critical path
- Example shows two paths
  - Each is 0.5 + 2 + 0.5 = 3 ns
- Trend
  - 1980s/1990s: Wire delays were tiny compared to logic delays
  - But wire delays not shrinking as fast as logic delays
    - Wire delays may even be greater than logic delays!
- 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

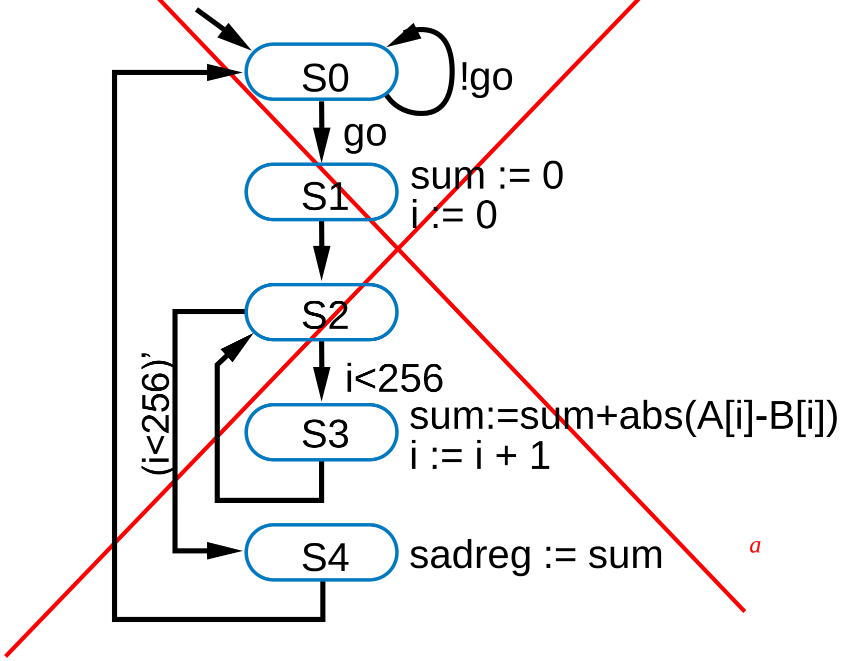
A Circuit May Have Numerous Paths

- 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

Inputs : A, B [256](8 bits); go (bit)
Outputs : sad (32 bits)
Local storage : sum, sadreg (32 bits); i (9 bits)



C code

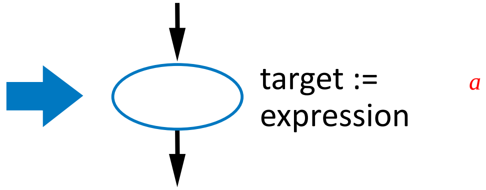
```
int SAD (byte A[256], byte B[256]) // not quite C syntax
{
    uint sum; short uint I;
    sum = 0;
    i = 0;
    while (i < 256) {
        sum = sum + abs(A[i] – B[i]);
        i = i + 1;
    }
    return sum;
}
```

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

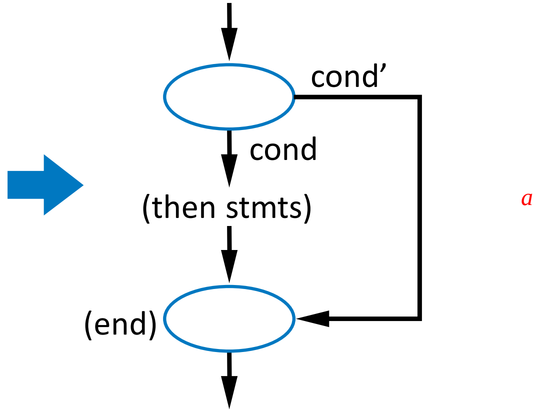
Converting from C to High-Level State Machine

- 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

```
target = expression;
```



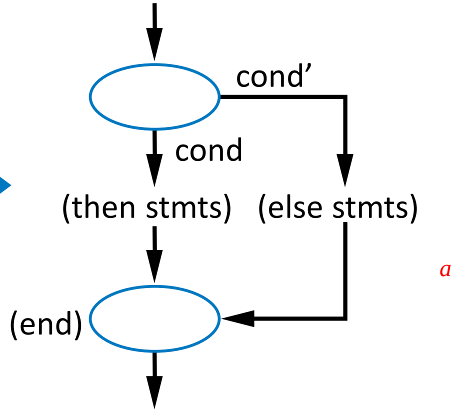
```
if (cond) {
   // then stmts
}
```



Converting from C to High-Level State Machine

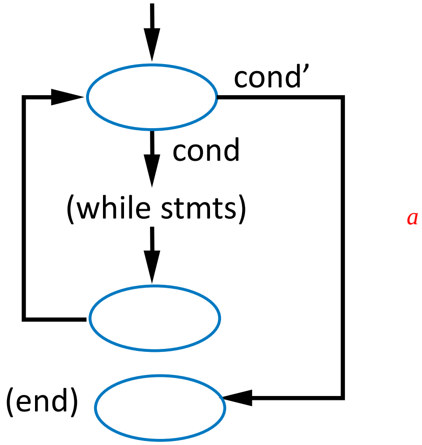
- *If-then-else*
  - Becomes state with condition check, transitioning to “then” statements if condition true, or to “else” statements if condition false

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

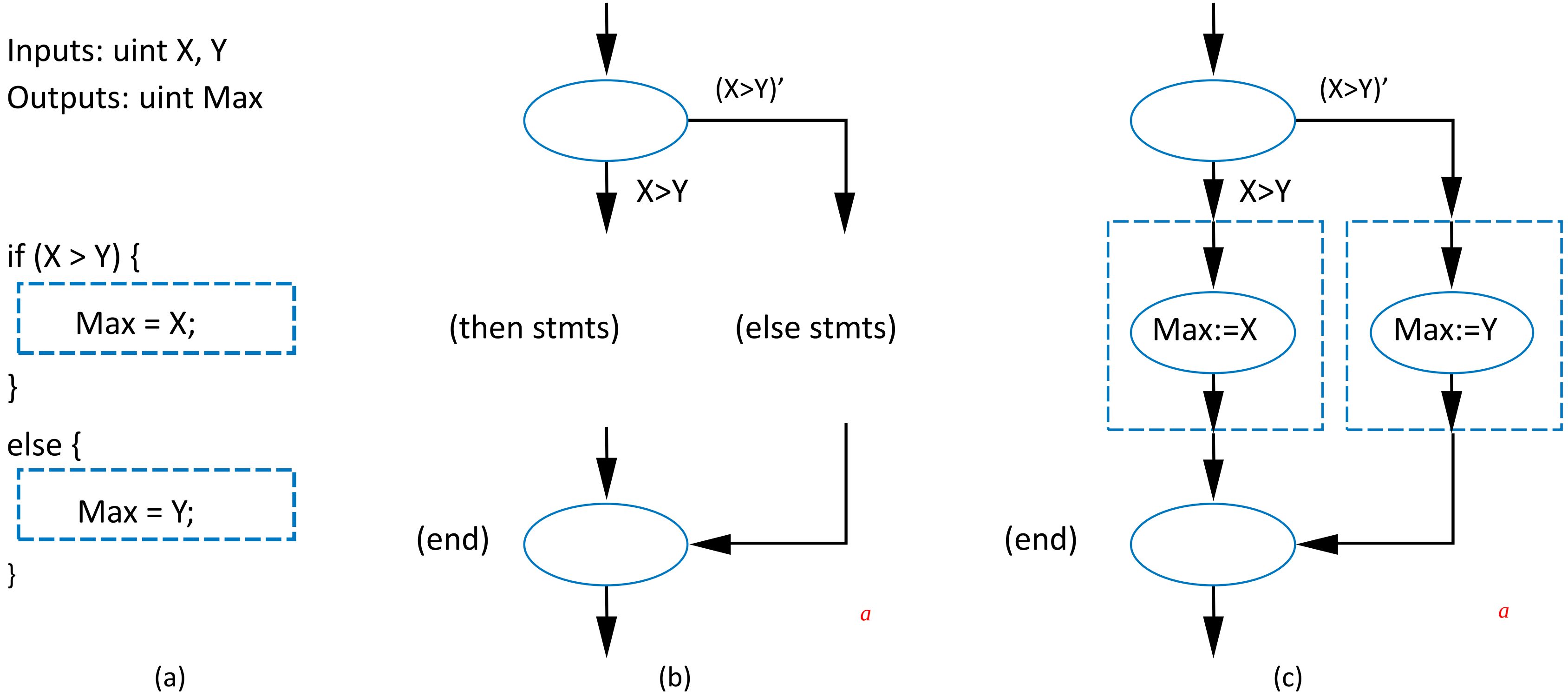


- *While loop* statement
  - Becomes state with condition check, transitioning to while loop’s statements if true, then transitioning back to condition check

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



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



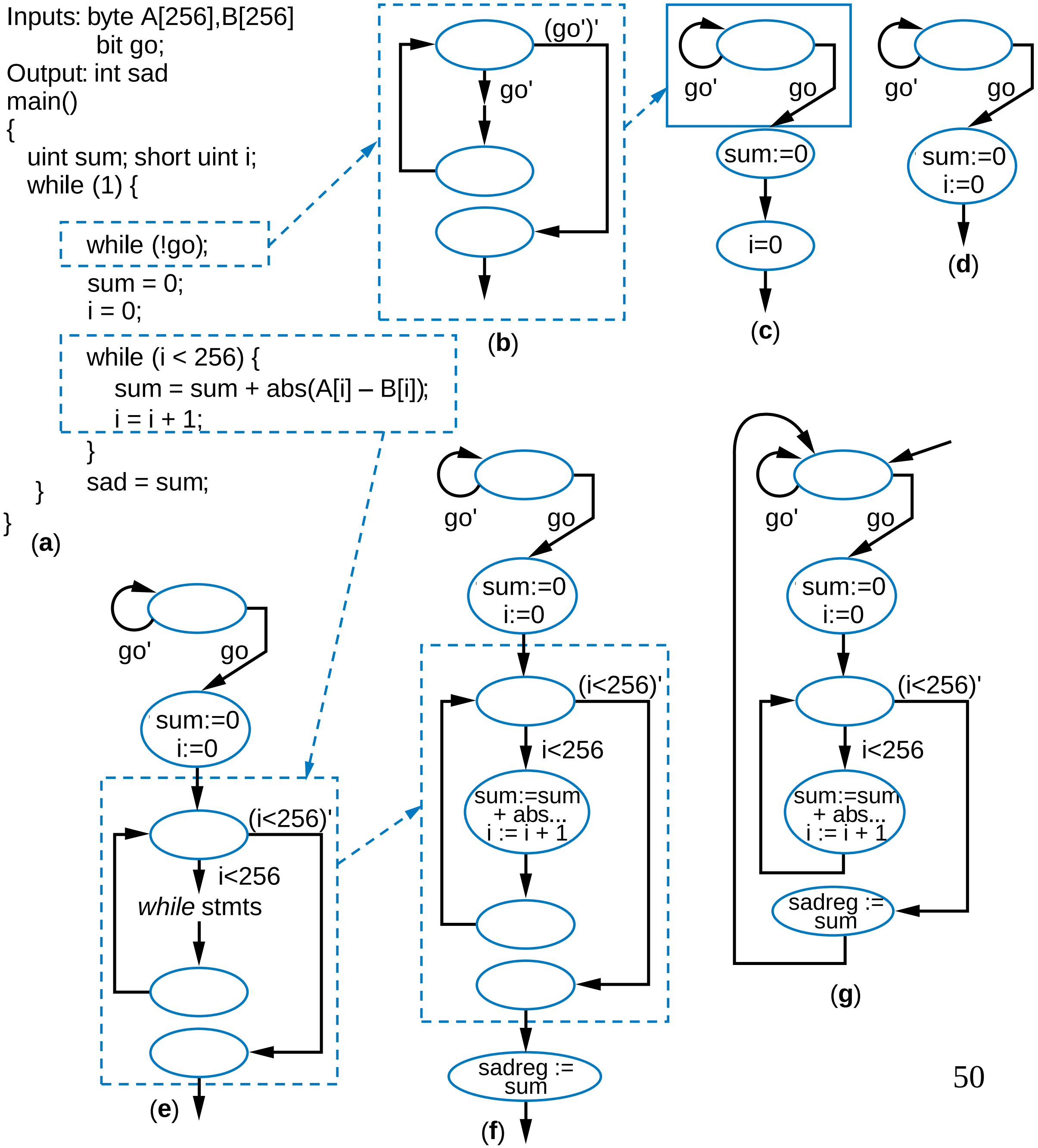
Inputs: uint X, Y
Outputs: uint Max

```
if (X > Y) {
    Max = X;
}
else {
    Max = Y;
}
```

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

Example: SAD C code to HLSM

- Convert each construct to states
  - Simplify, e.g., merge states
- RTL design process to convert to circuit
- Can thus convert C to circuit using straightforward process
  - Actually, subset of C (not all C constructs easily convertible)
  - Can use language other than C



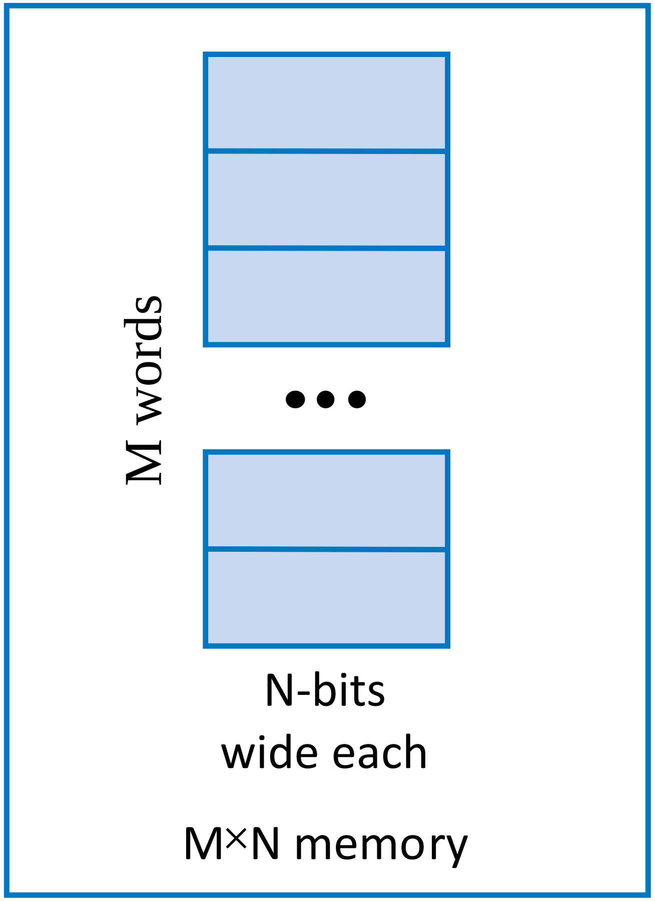
```
Inputs: byte A[256],B[256]
          bit go;
Output: int sad
main()
{
   uint sum; short uint i;
   while (1) {

      while (!go);
      sum = 0;
      i = 0;
      while (i < 256) {
         sum = sum + abs(A[i] – B[i]);
         i = i + 1;
      }
      sad = sum;
   }
}
```

(a)

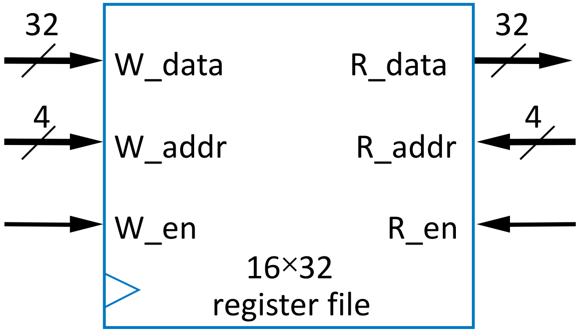
Memory Components

- RTL design instantiates datapath components to create datapath, controlled by a controller
  - Some components are used outside the controller and DP
- ***MxN memory***
  - M words, N bits wide each
- Several varieties of memory, which we now introduce

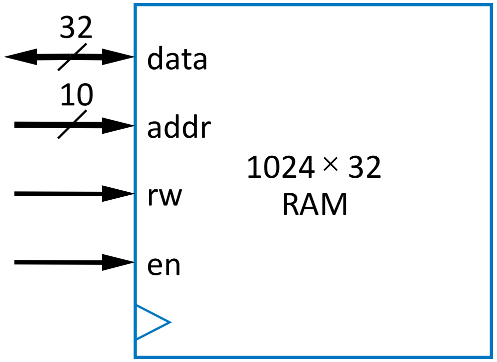


Random Access Memory (RAM)

- 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 one port; RF usually two or more
  - RAM vs. RF
    - RAM typically larger than *about* 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

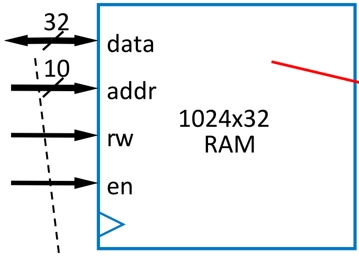


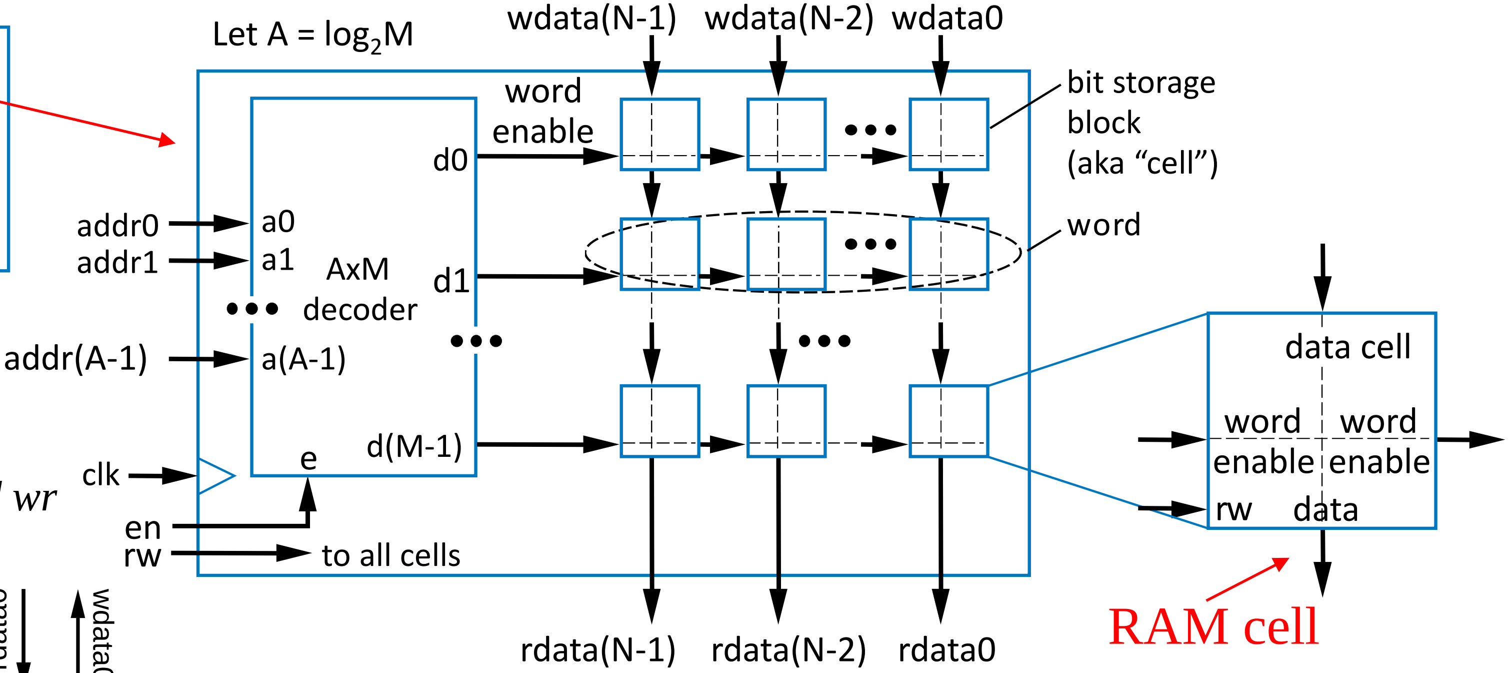
Register file from Chpt. 4



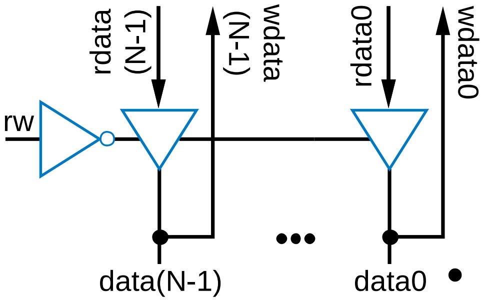
RAM block symbol

RAM Internal Structure



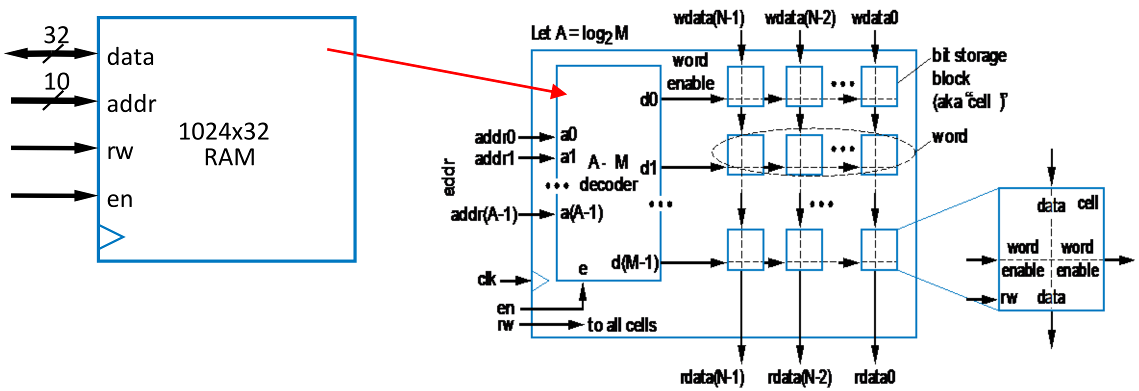


Combining rd and wr data lines



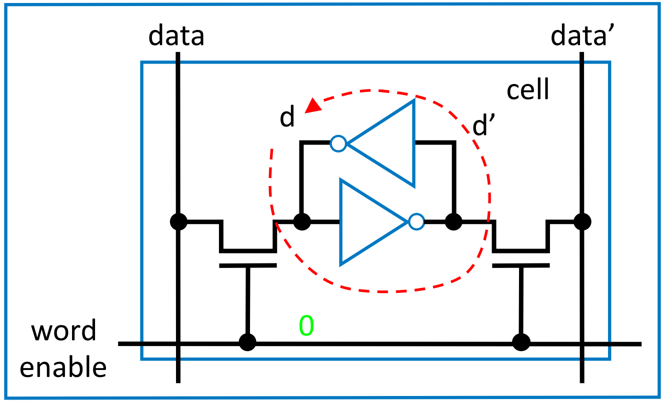
- Similar internal structure as register file
  - Decoder enables appropriate word based on address inputs
  - rw controls whether cell is written or read
  - rd and wr data lines typically combined
  - Let’s see what’s inside each RAM cell

Static RAM (SRAM)

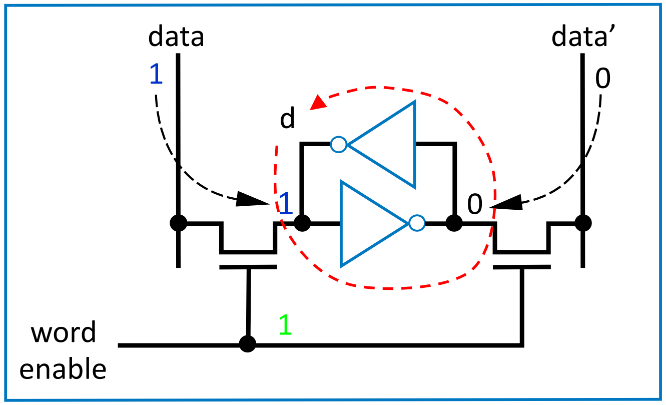


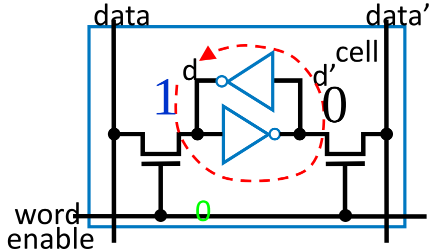
- “Static” RAM cell
  - 6 transistors (recall inverter is 2 transistors)
  - Writing this cell
    - *word enable* input comes from decoder
    - When 0, value *d* loops around inverters
      - That loop is where a bit stays stored
    - When 1, the *data* bit value enters the loop
      - *data* is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

SRAM cell

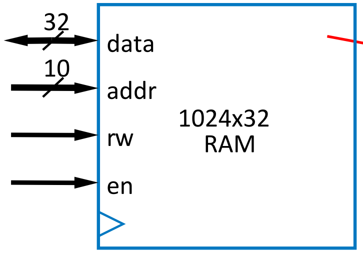


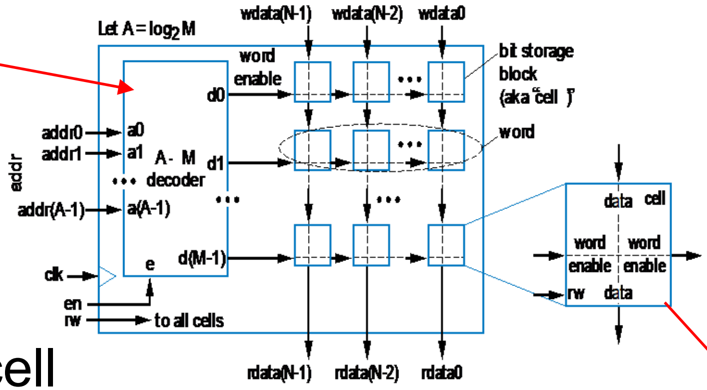
SRAM cell





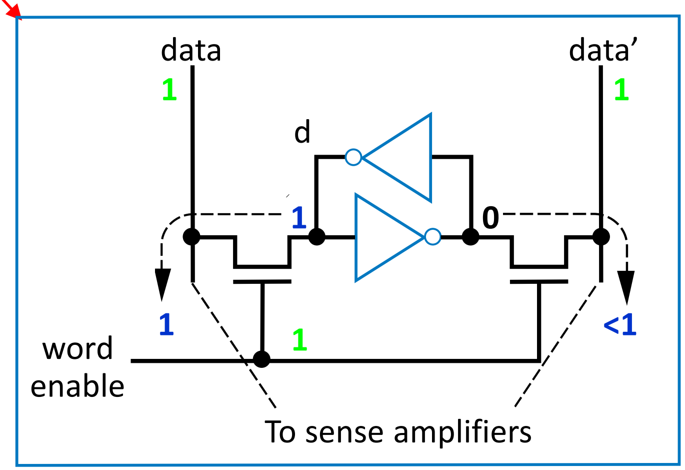
Static RAM (SRAM)



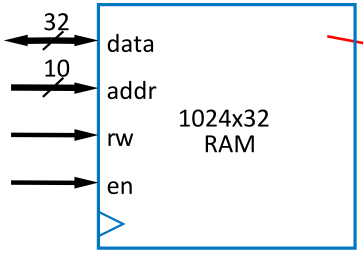


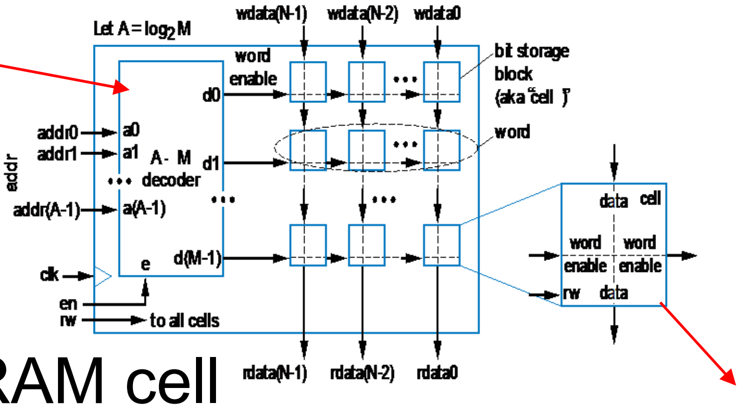
- "Static" RAM cell
  - Reading this cell
    - Somewhat trickier
    - When rw set to read, the RAM logic sets both *data* and *data'* to 1
    - The stored bit d will pull either the left line or the right bit down slightly below 1
    - "Sense amplifiers" detect which side is slightly pulled down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

SRAM cell



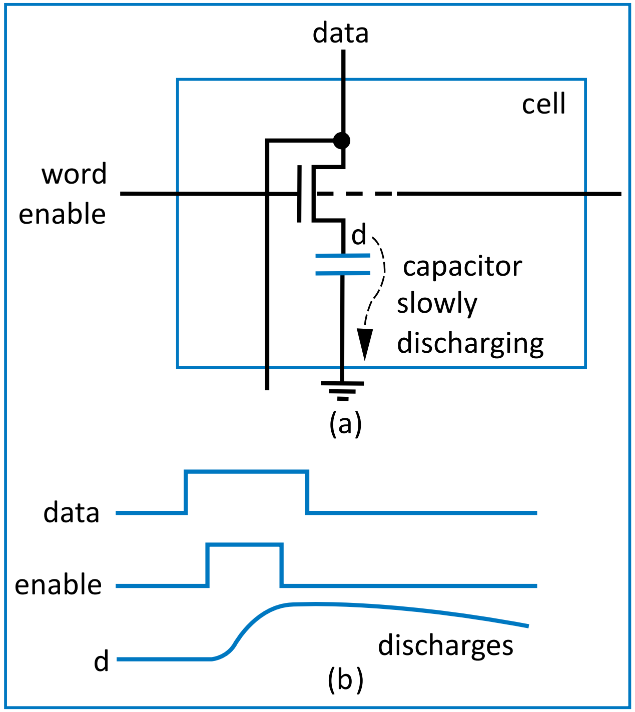
Dynamic RAM (DRAM)





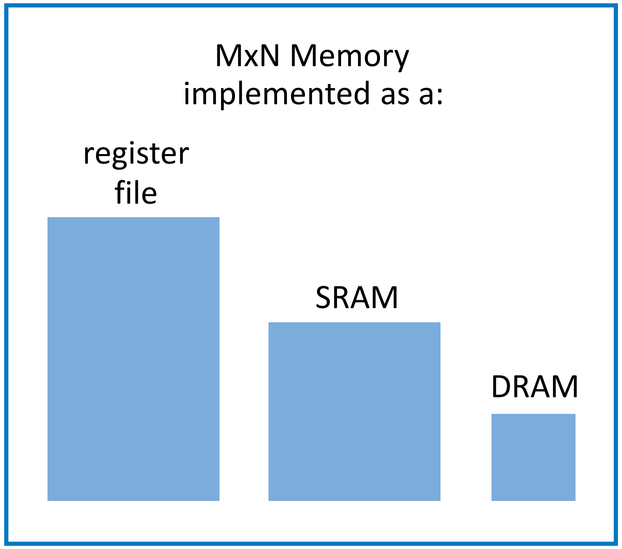
- "Dynamic" RAM cell
  - 1 transistor (rather than 6)
  - Relies on *large* capacitor to store bit
    - Write: Transistor conducts, data voltage level gets stored on top plate of capacitor
    - Read: Just look at value of *d*
    - Problem: Capacitor discharges over time
      - Must "refresh" regularly,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

DRAM cell



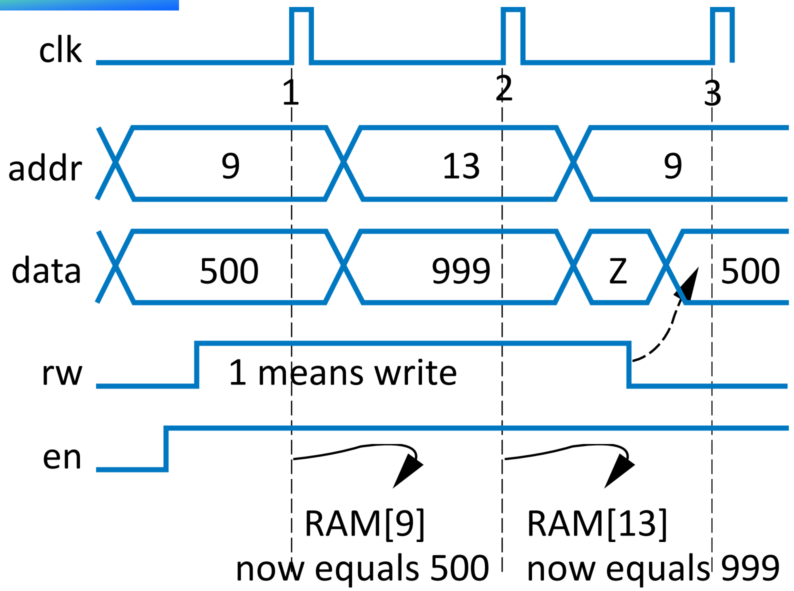
Comparing Memory Types

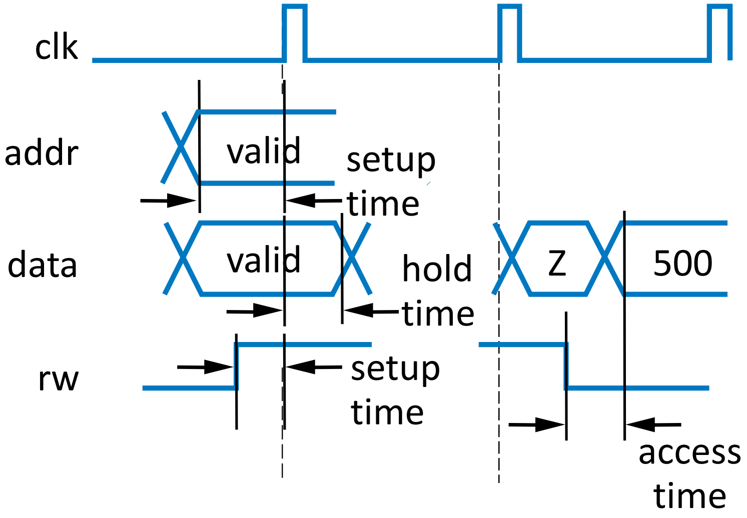
- Register file
  - Fastest
  - But biggest size
- SRAM
  - Fast
  - More compact than register file
- DRAM
  - Slowest
    - And refreshing takes time
  - But very compact
- Use register file for small items, SRAM for large items, and DRAM for huge items
  - Note: DRAM's big capacitor requires a special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

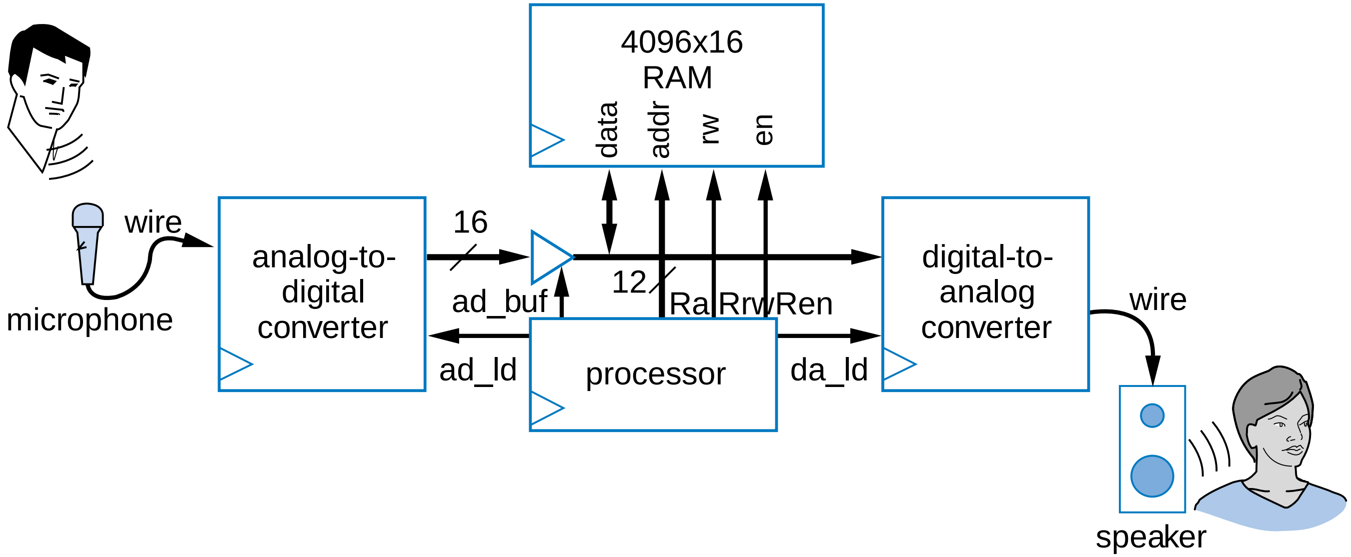




(b)

- 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

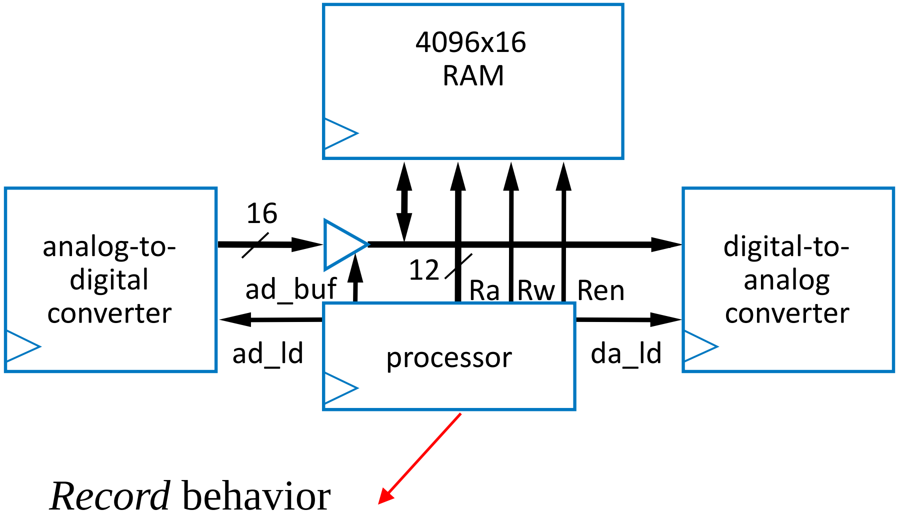
RAM Example: Digital Sound Recorder



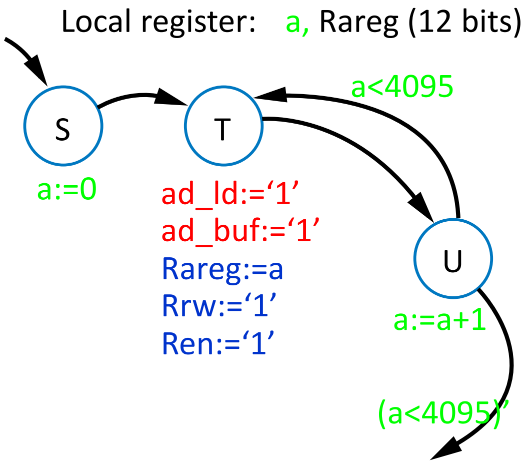
- Behavior
  - Record: Digitize sound, store as series of 4096 12-bit digital values in RAM
    - We'll use a 4096x16 RAM (12-bit wide RAM not common)
  - Play back later
  - Common behavior in telephone answering machine, toys, voice recorders
- To record, processor should read a-to-d, store read values 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 HLSM
  - Begin with the *record* behavior
  - Create local storage 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*
        - » Rareg:=a, Rrw:=‘1’, Ren:=‘1’



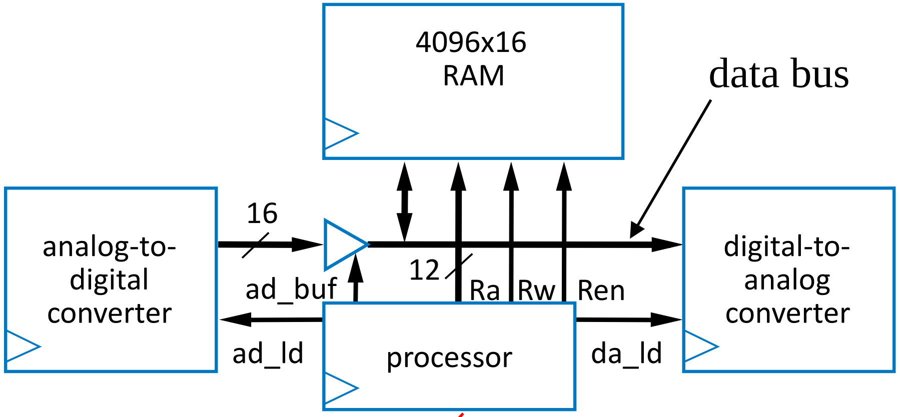
Record behavior



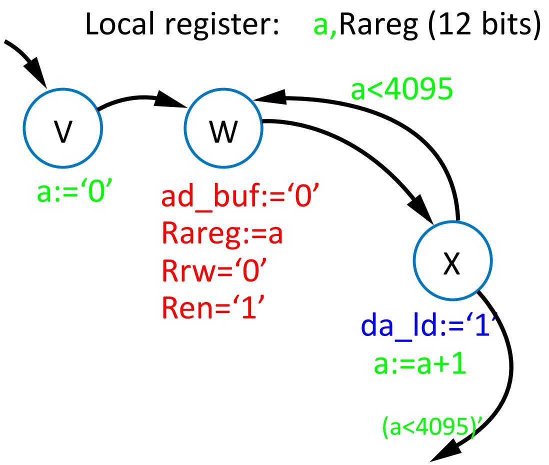
a

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



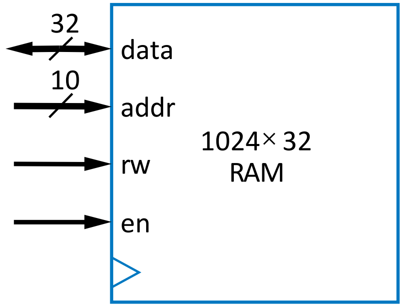
Play behavior



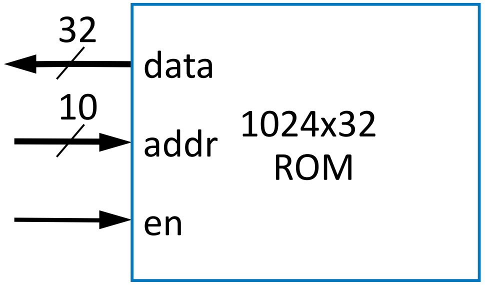
a

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

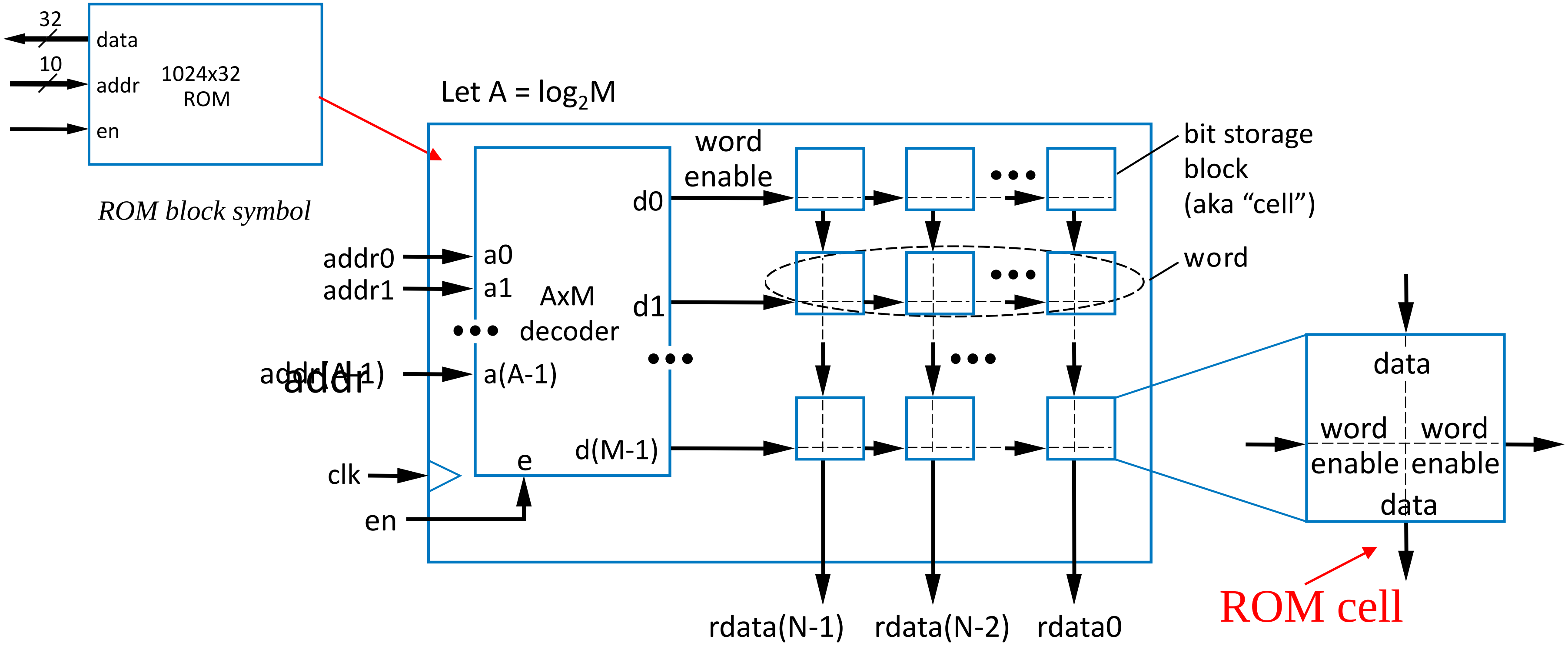


RAM block symbol



ROM block symbol

Read-Only Memory – ROM

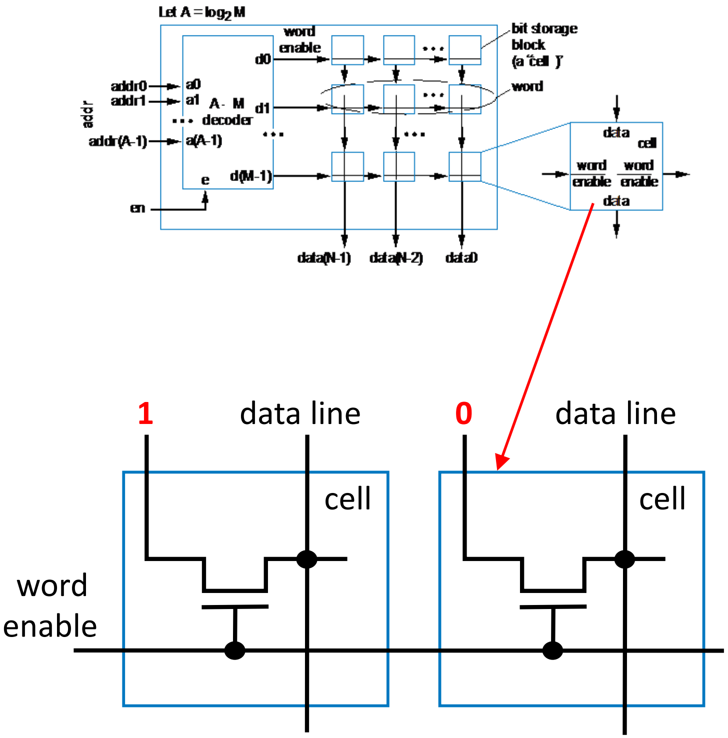


ROM block symbol

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

ROM Types

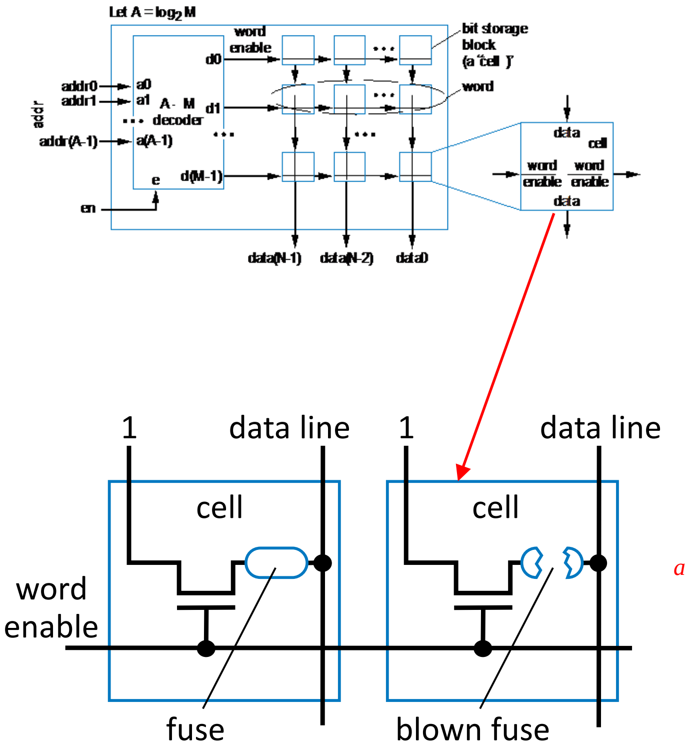
- If a ROM can only be read, how are the stored bits stored in the first place?
  - Storing bits in a ROM known as *programming*
  - Several methods
- ***Mask-programmed ROM***
  - Bits are hardwired as 0s or 1s during chip manufacturing
    - 2-bit word on right stores “10”
    - word enable (from decoder)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



Digital Design 2e
Copyright © 2010
Frank Vahid

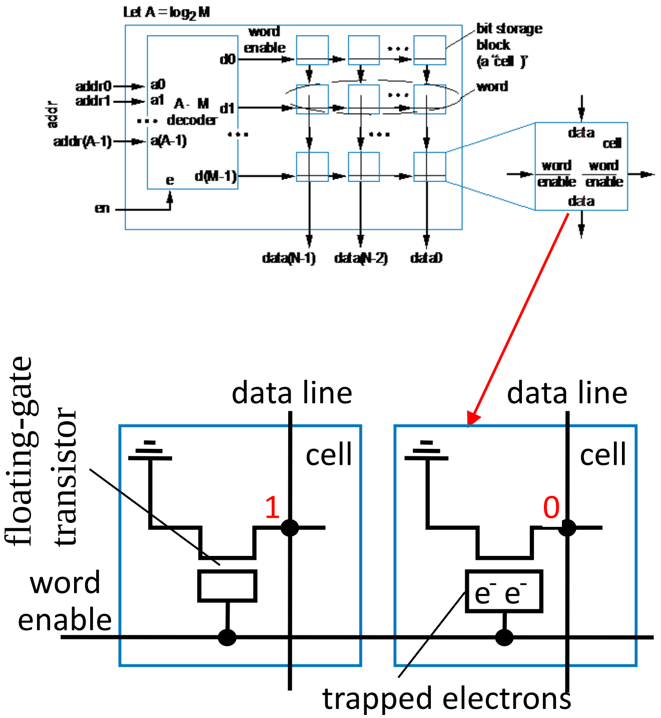
ROM Types

- ***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***

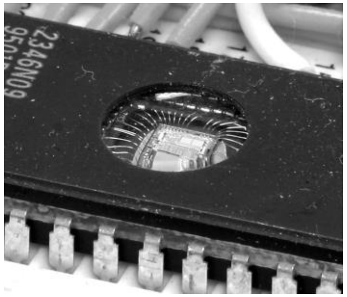


ROM Types

- ***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

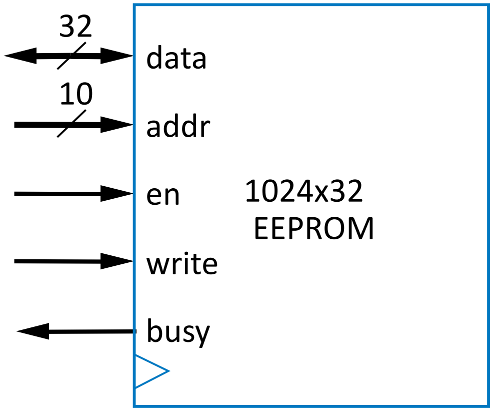


a

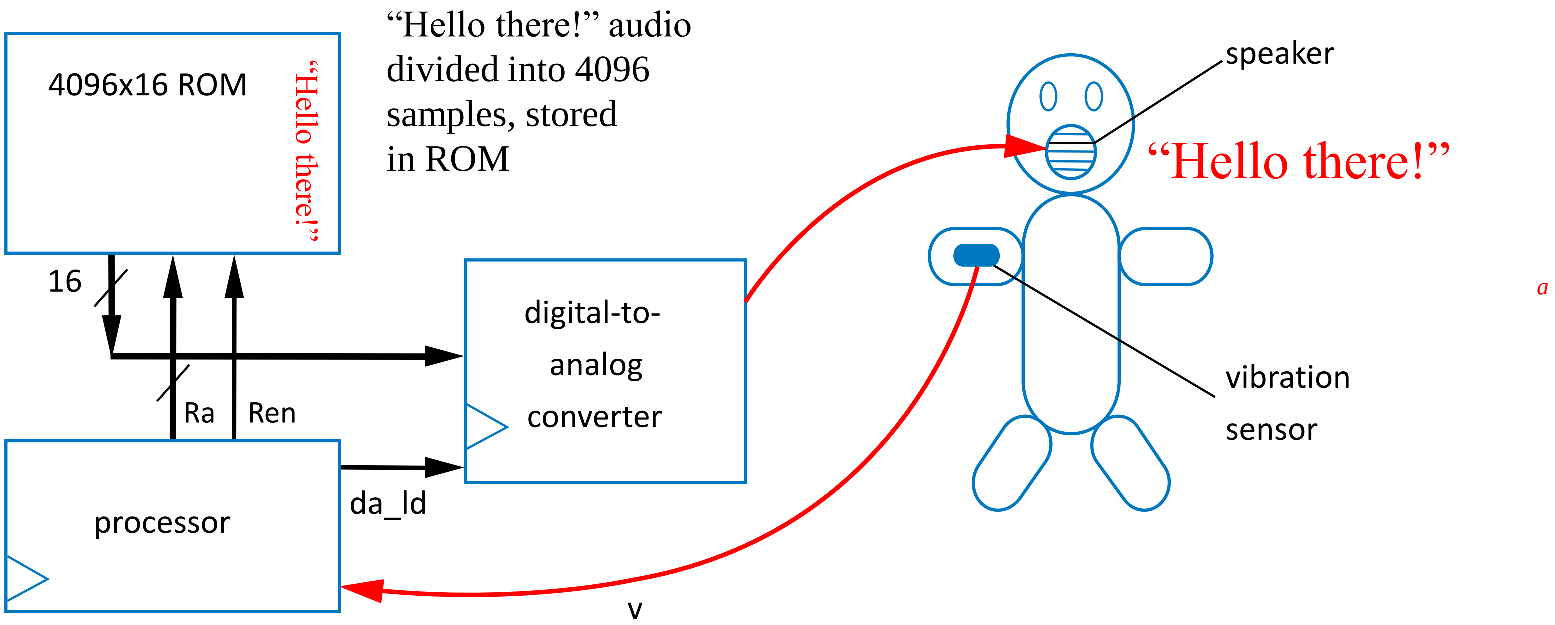


ROM Types

- ***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*
  - Became very common starting in 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



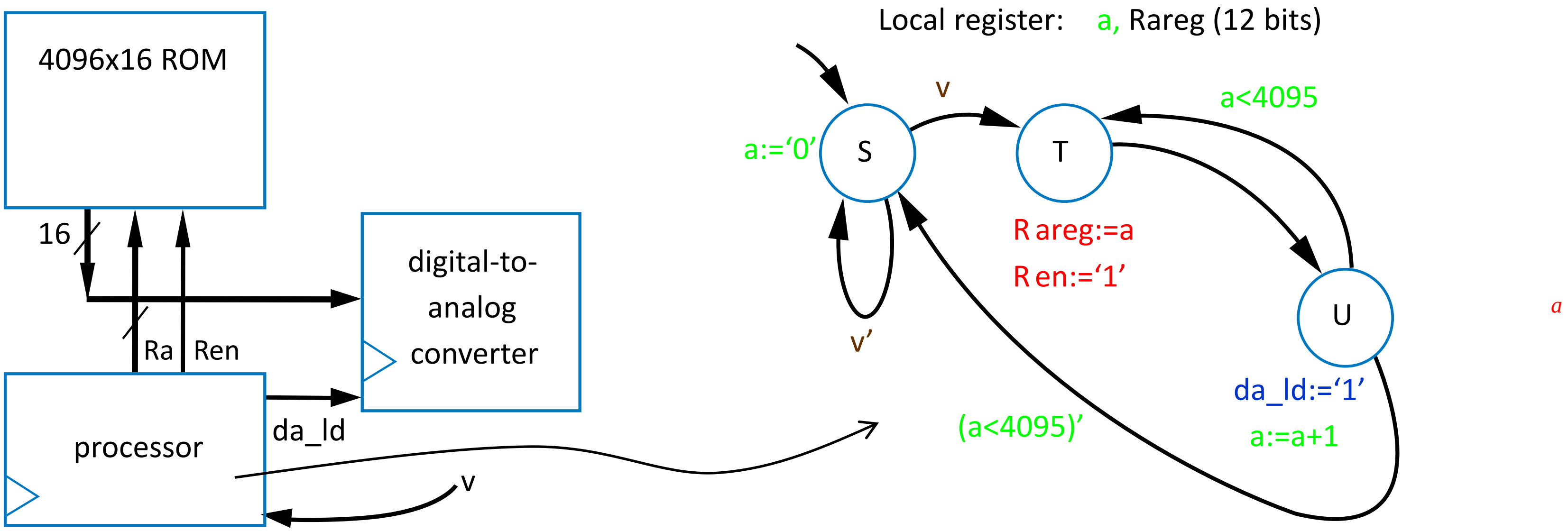
ROM Example: Talking Doll



“Hello there!” audio divided into 4096 samples, stored in ROM

- Doll plays prerecorded message, triggered 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, may 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

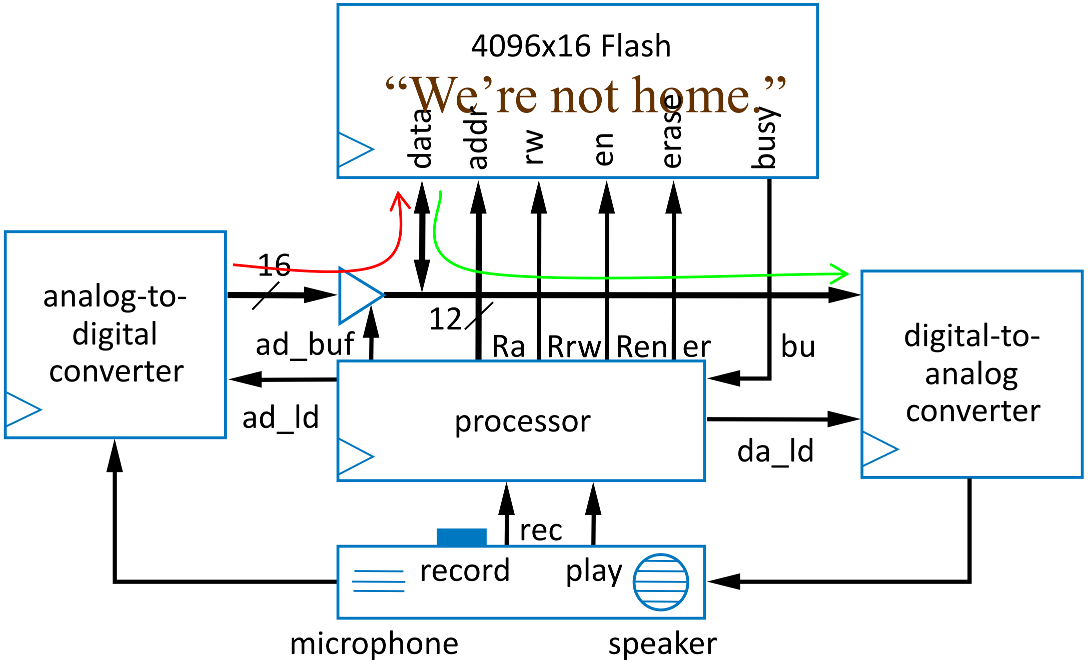
ROM Example: Talking Doll



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

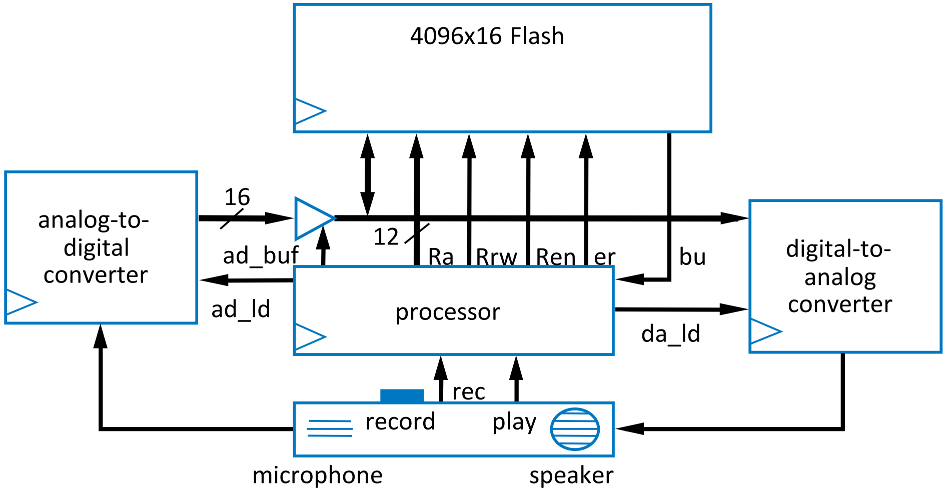
ROM Example: Digital Telephone Answering Machine Using a Flash Memory

- 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

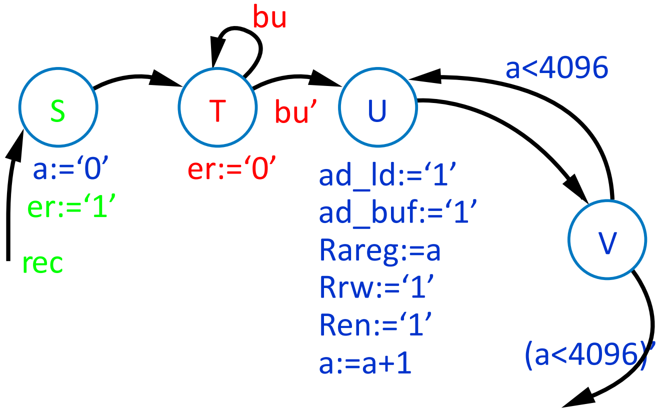


ROM Example: Digital Telephone Answering Machine Using a Flash Memory

- HLSM
  - 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*

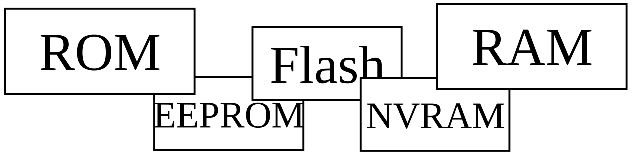


Local register: a, Rareg (13 bits)



a

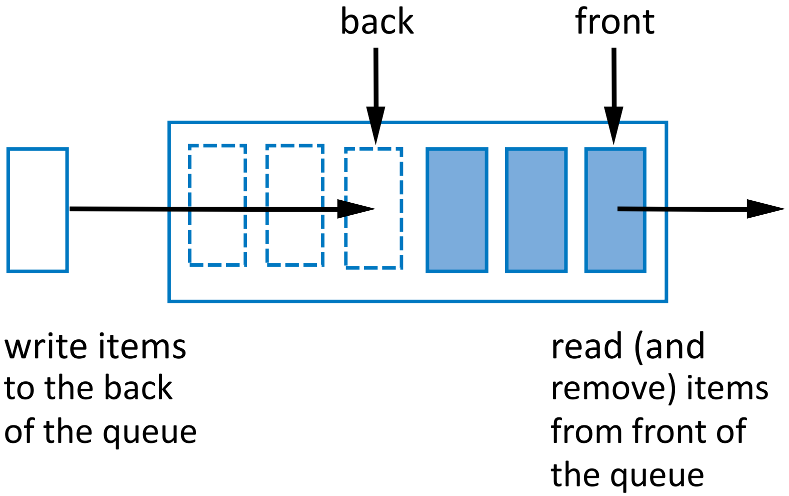
Blurring of Distinction Between ROM and RAM



- 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

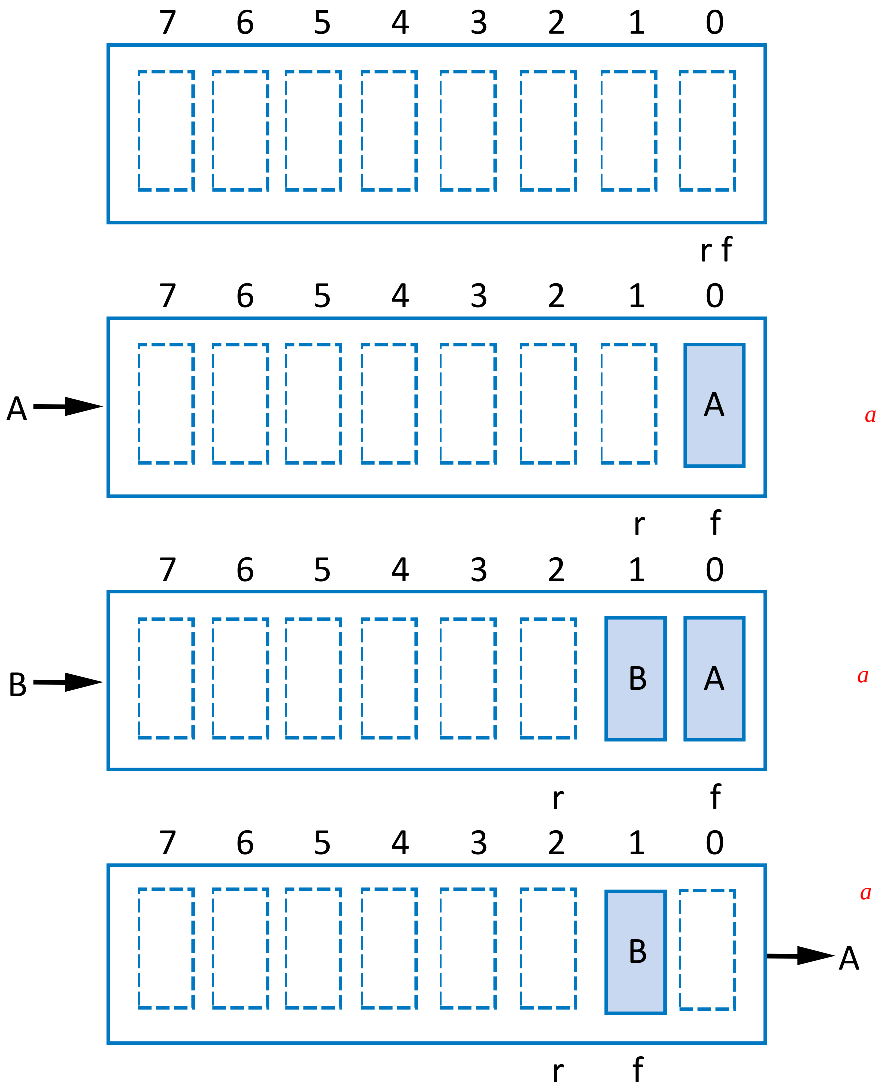
Queues (FIFOs)

- 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)


back
front
write items to the back of the queue
read (and remove) items from front of the queue

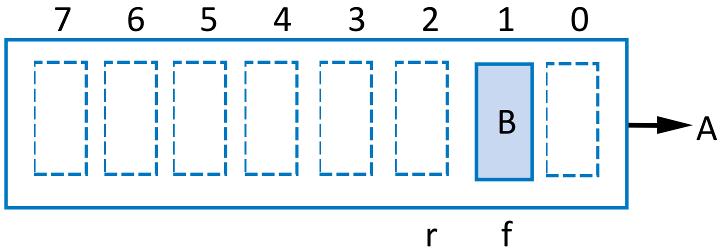
Queues

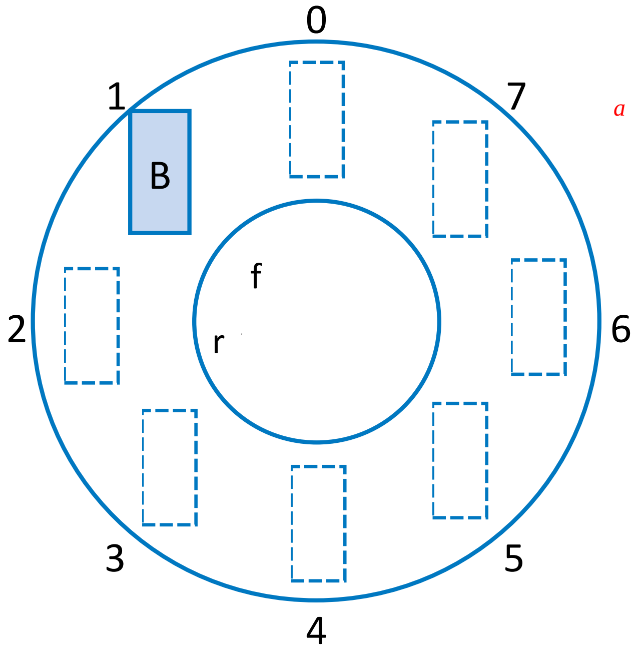
- Queue has addresses, and two pointers: *rear* and *front*
  - Initially both point to 0
- Push (write)
  - Item written to address pointed to by *rear*
  - *rear* incremented
- Pop (read)
  - Item read from address pointed to by *front*
  - *front* incremented
- If front or rear reaches 7, next (incremented)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



Queues

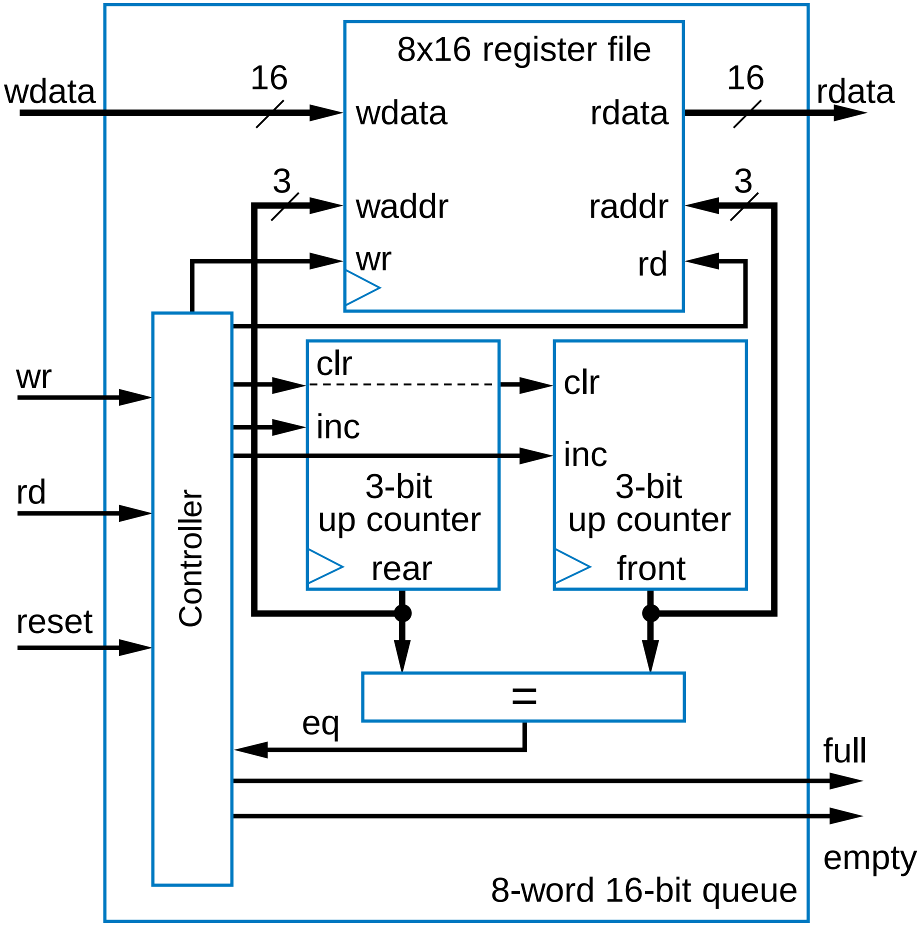
- Treat memory as a circle
  - If front or rear reaches 7, next (incremented) value should be 0 rather than 8 (for a queue with addresses 0 to 7)
- Two conditions of interest
  - Full queue – no room for more items
    - In 8-entry queue, means 8 items present
    - No further pushes allowed until a 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
  - So is popping an empty queue

Initially empty queue

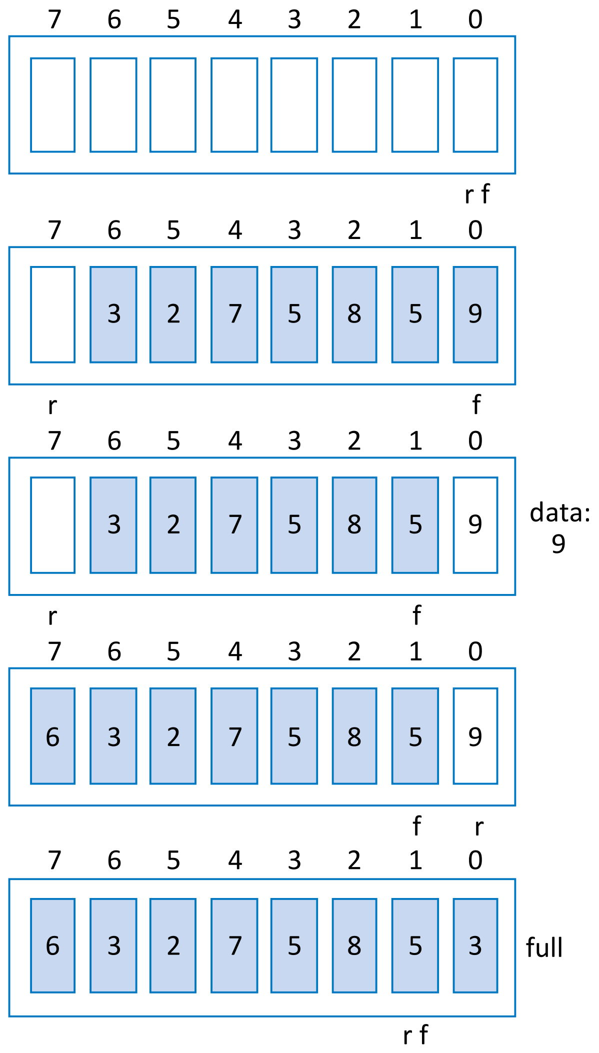
1. After pushing 9, 5, 8, 5, 7, 2, 3

2. After popping

3. After pushing 6

4. After pushing 3

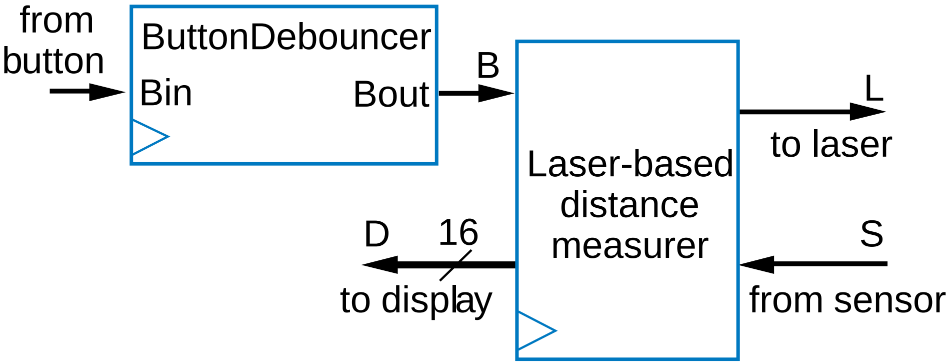
5. After pushing 4

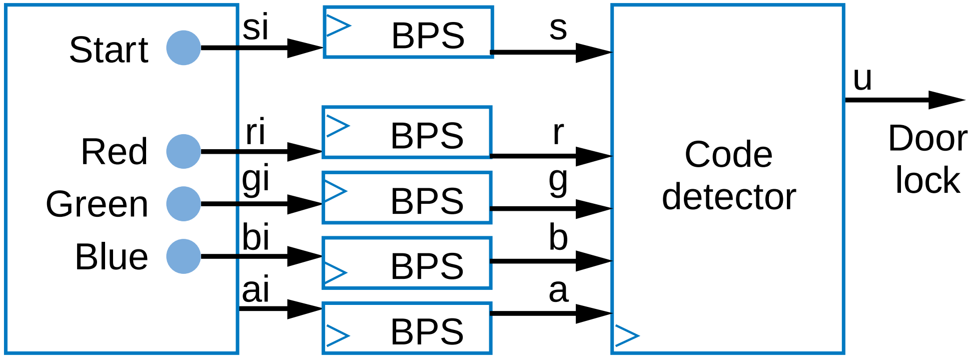


ERROR! Pushing a full queue results in unknown state.

Multiple Processors

- Using multiple processors can ease design
  - Keeps distinct behaviors separate
  - Ex: Laser-based distance measurer with button debounce
    - Use two processors
  - Ex: Code detector with button press synchronizers (BPS)
    - BPS processor for each input, plus CodeDetector processor





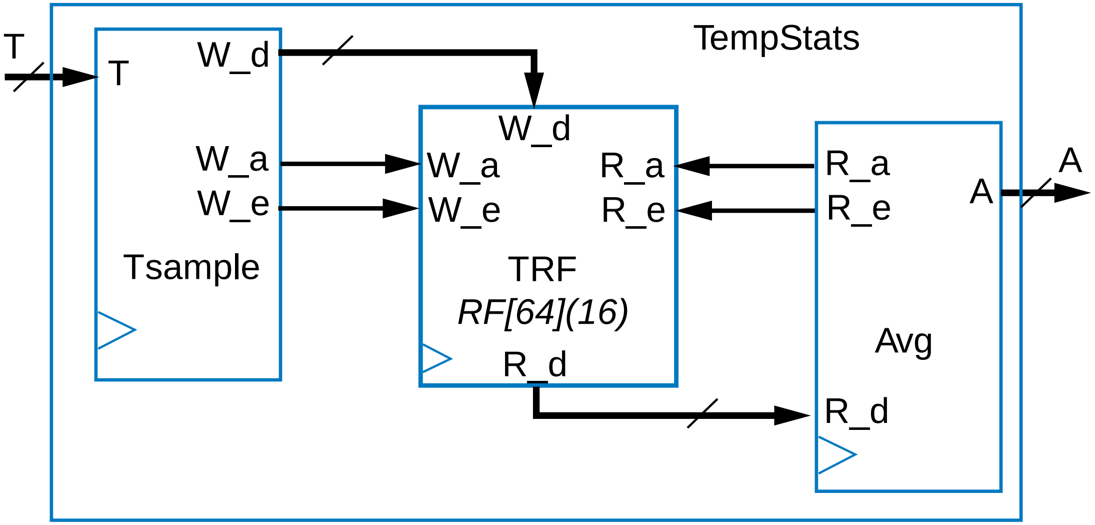
Interfacing Multiple Processors

- Use signal, register, or other component outside processors
  - Known as ***global***
- Common methods use global...
  - control signal, data signal, register, register file, queue
- Typically all multiple processors and clocked globals use same clock
  - *Synchronized*

Ex: Temperature Statistics with Multiple Processors

- 16-bit unsigned input T from temperature sensor, 16-bit output A. Sample T every 1 second. Compute output A every minute, should equal average of most recent 64 samples.
- Single HLSM: Complicated
- Instead, two HLSMs (and hence two processors) and shared register file
  - Tsample HLSM: Store T into successive RF address, once per sec.
  - Avg HLSM: Compute and output average of all 64 RF words, once per min.
  - Note that each uses distinct timer

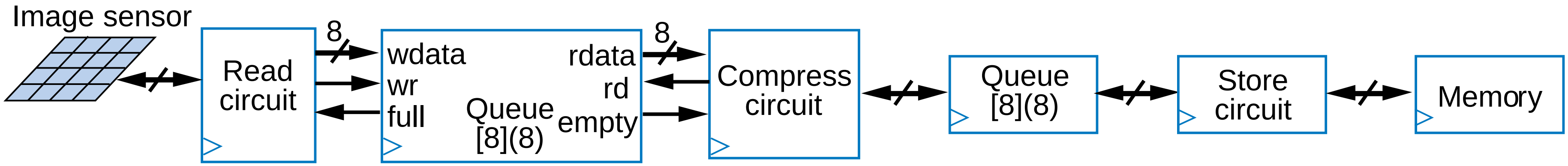
Keeping the sampling and averaging behaviors separate leads to simple design



a

Ex: Digital Camera with Mult. Processors and Queue

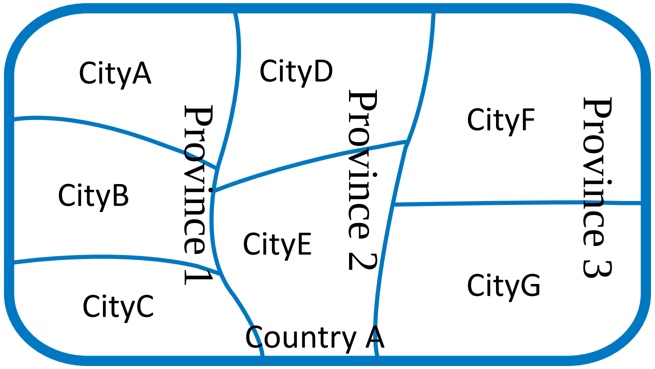
- Read and Compress processors (Ch 1)
  - Compress may take longer, depends on picture
  - Use queue, read can push additional pics (up to 8)
  - Likewise, use queue between Compress and Store

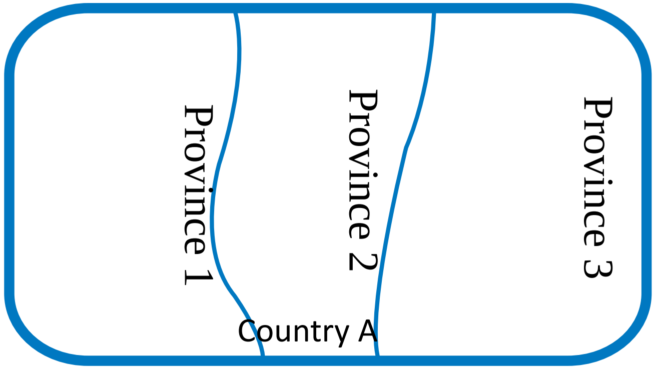


a

Hierarchy – A Key Design Concept

- Hierarchy
  - Organization with few items at the top, with each item decomposed into other items
  - Common example: Country
    - 1 item at 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

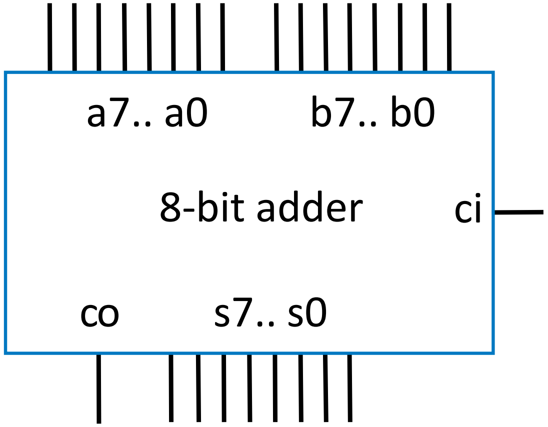




Map showing just top two levels of hierarchy

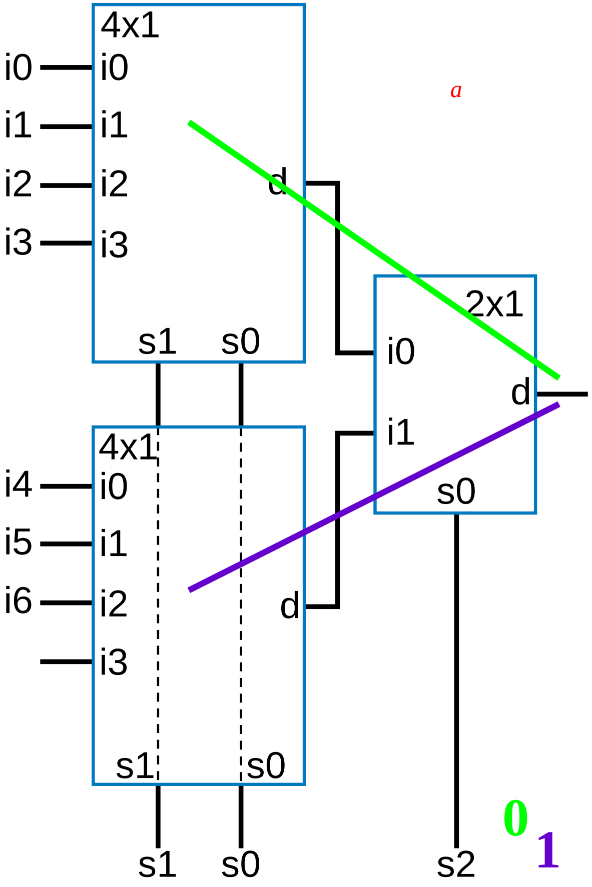
Hierarchy and Abstraction

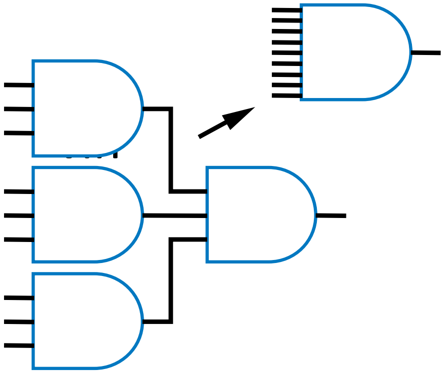
- Abstraction
  - Hierarchy often involves not just grouping items into a new item, but also associating higher-level behavior with the new item, known as *abstraction*
    - Ex: 8-bit adder has understandable high-level behavior—adds two 8-bit binary numbers
  - Frees designer from having to remember,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



Hierarchy and Composing Larger Components from Smaller Versions

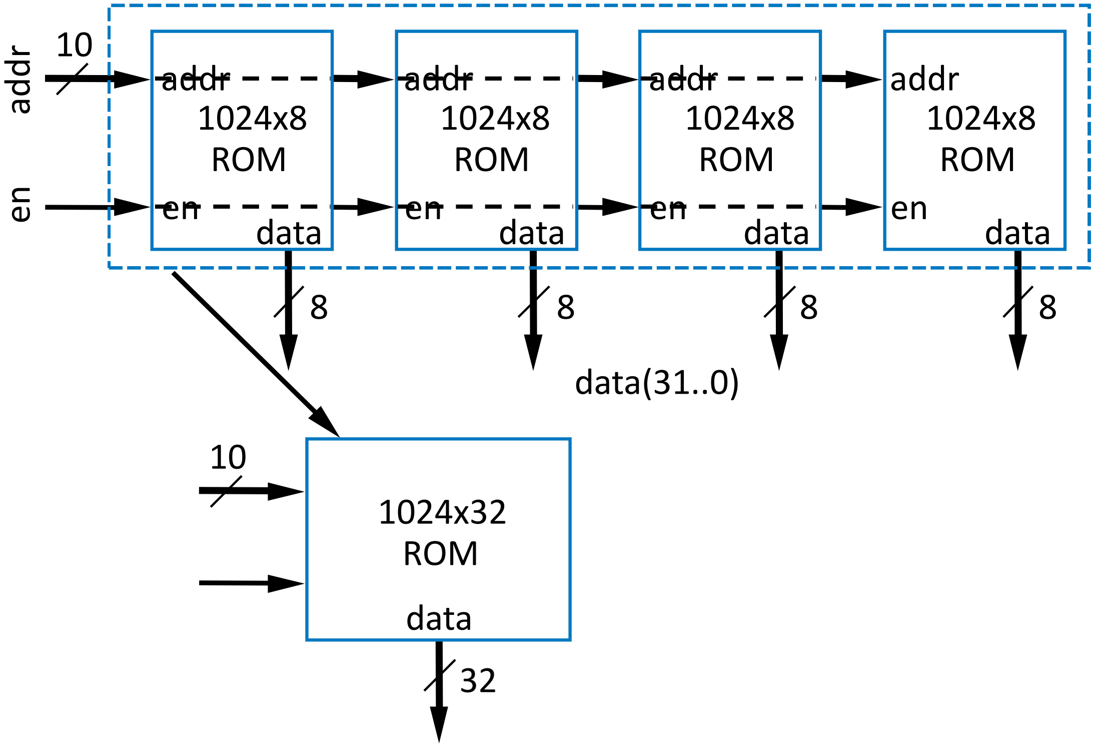
- A common task is to compose smaller components into a larger one
  - Gates: Suppose you have plenty of 3-input AND gates, but need a 9-input AND gate
    - Can simple compose the 9-input gate from several 3-input gates
  - Muxes: Suppose you have 4x1 and 2x1 muxes, but need an 8x1 mux
    - s2 selects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





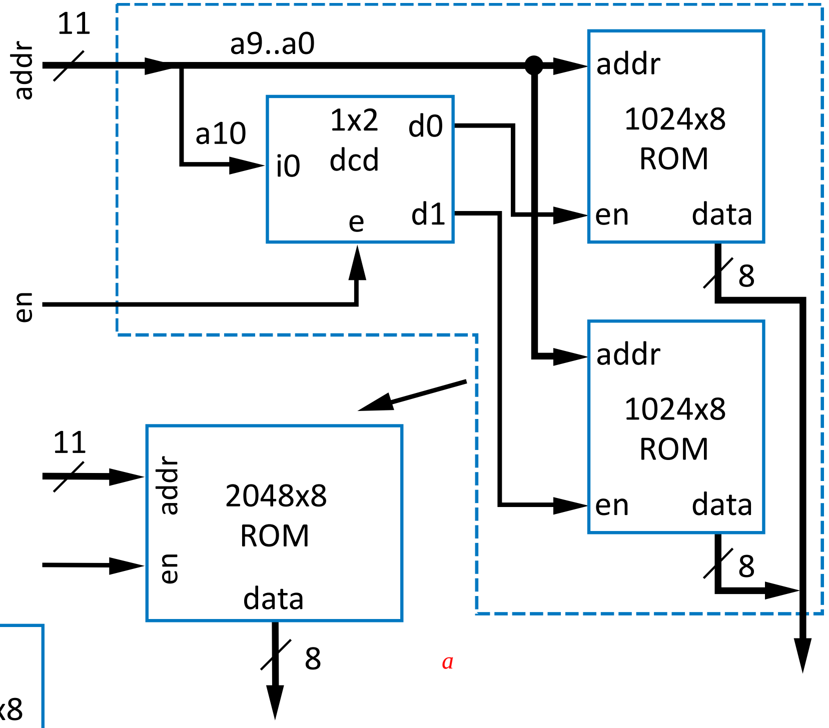
Hierarchy and Composing Larger Components from Smaller Versions

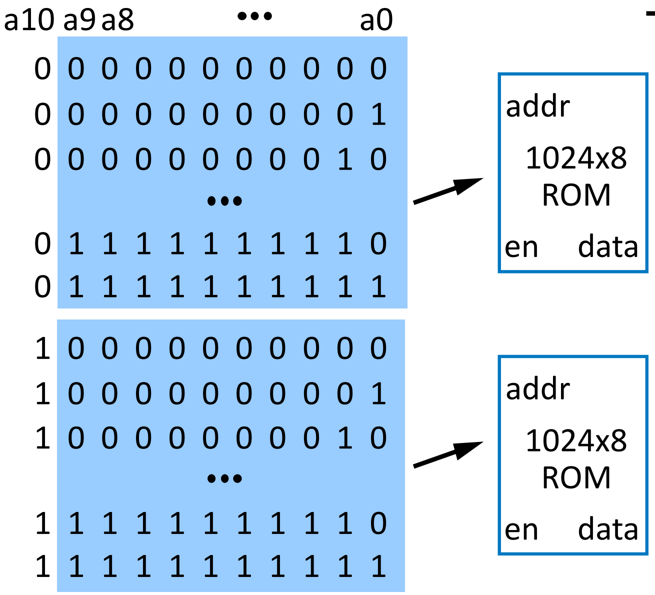
- Composing memory very common
- Making memory words wider
  - Easy – just place memories side-by-side until desired width obtained
  - Share address/control lines, concatenate data lines
  - Example: Compose 1024x8 ROMs into 1024x32 ROM



Hierarchy and Composing Larger Components from Smaller Versions

- Creating memory with more words
  - Put memories on top of one another until the number of desired words is achieved
  - Use decoder to select among the memories
    - Can use highest order address input(s) as decoder input
    - Although actually, any address line could be used
  - Example: Compose 1024x8 memories into 2048x8 memory





a10	a9	a8				•••				a0
0	0	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	0	0	0	1	0
					•••					
0	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	1
1	0	0	0	0	0	0	0	0	1	0
					•••					
1	1	1	1	1	1	1	1	1	1	0
1	1	1	1	1	1	1	1	1	1	1

a10 just chooses which memory to access

To create memory with more words and wider words, can first compose to enough words, then widen.

Chapter Summary

- Modern digital design involves creating processor-level components
- High-level state machines
- RTL design process
  - 1. Capture behavior: Use HLSM
  - 2. Convert to circuit
    - A. Create datapath   B. Connect DP to controller   C. Derive controller FSM
- More RTL design
  - More components, arrays, timers, control vs. data dominated
- 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
- Memory components (RAM, ROM)
- Queues
- Multiple processors
- Hierarchy: A key concept used throughout Chapters 2-5