

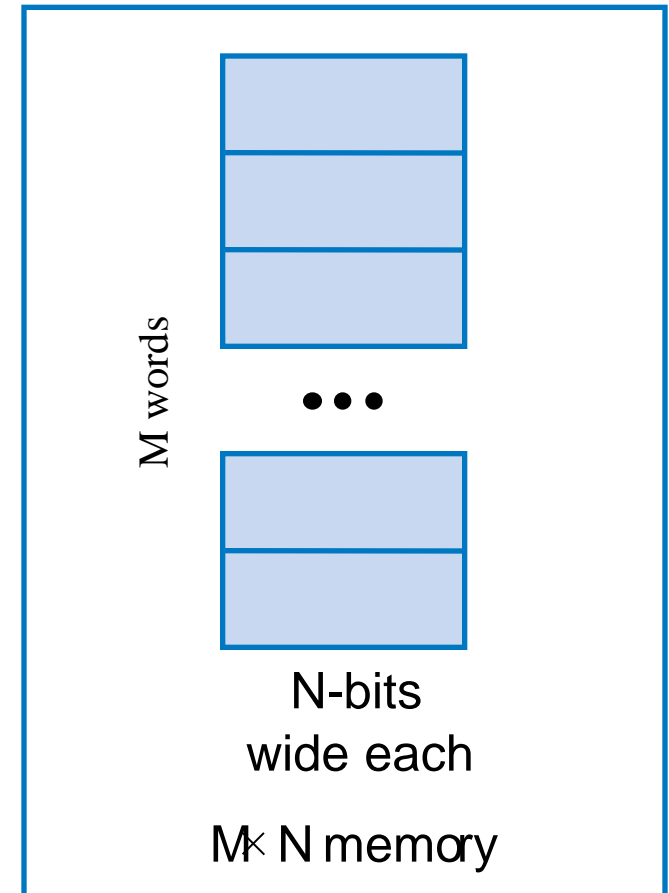


Memory



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



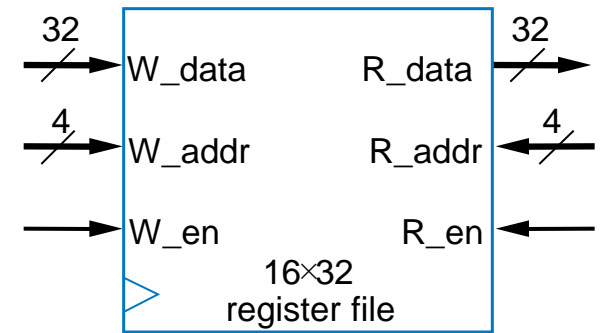
Memory Terms

- Word: a group of bits treated as a single entity.
- Word size: number of bits per word
- Byte: 8 bits
- M x n memory: m = number of words, n = word size
- Memory size: number of words that can be stored in memory
- Memory capacity in bits: $M \times N$
- Address: unique combination of bits identifying the location of a word in memory
- Number of address bits (N) required for a memory of M words
 $N = 1 + \text{integer value of } \log(M)/\log 2$
- Volatile memory: memory values are lost when power is switched off
- Nonvolatile memory: memory values retained on power loss
- Bus contention: More than one output not in the high Z mode

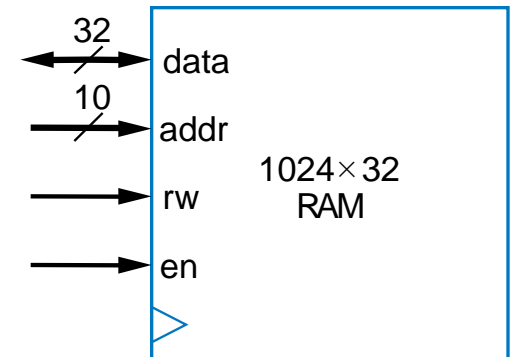


Random Access Memory (RAM)

- 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



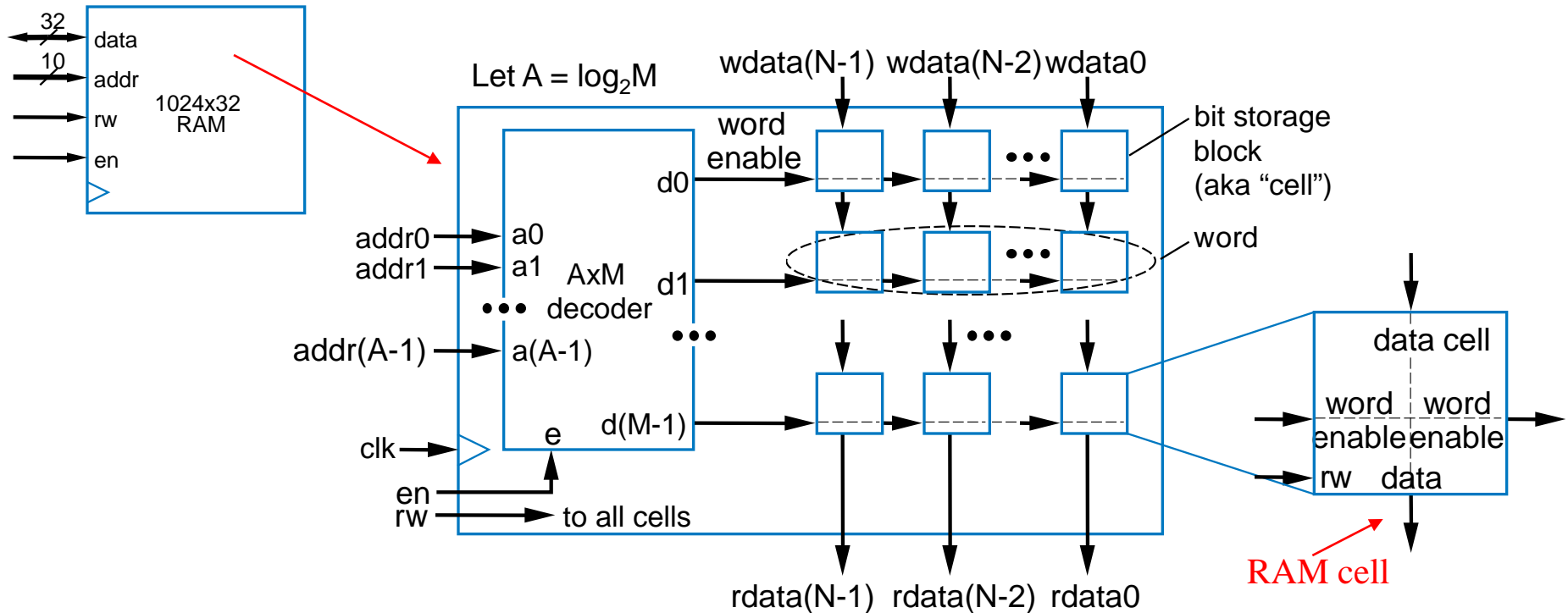
Register file from Chpt. 4



RAM block symbol

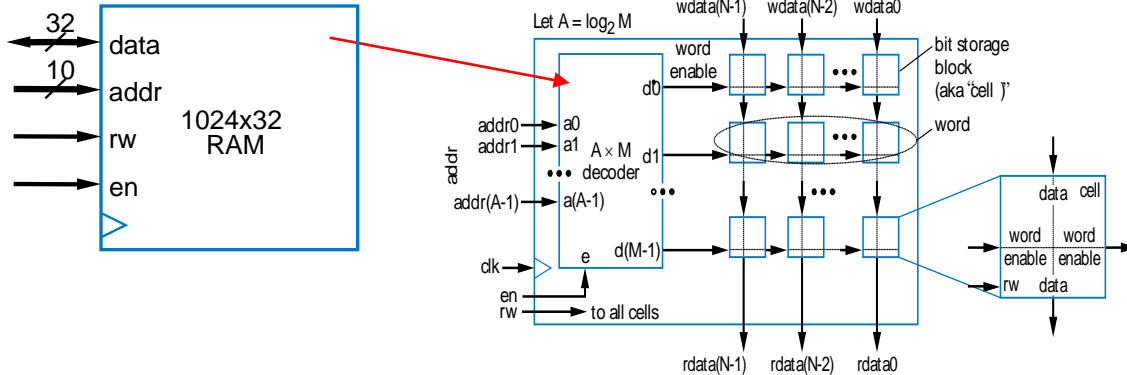


RAM Internal Structure

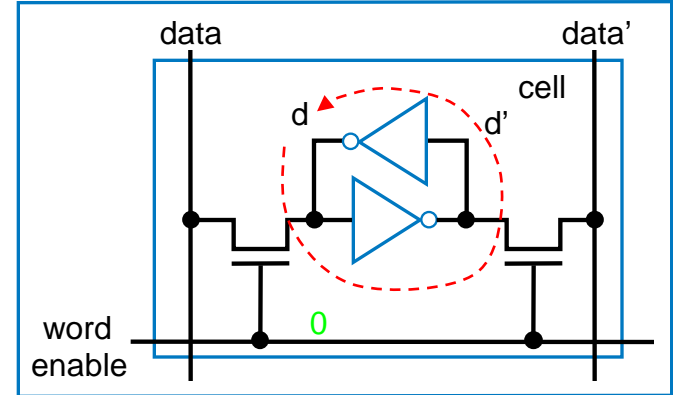


- Similar internal structure as register file
 - Decoder enables appropriate word based on address inputs
 - rw controls whether cell is written or read
 - Let's see what's inside each RAM cell

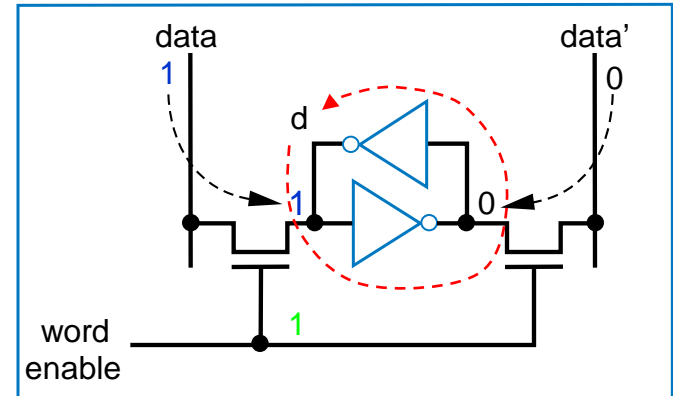
Static RAM (SRAM)



SRAM cell

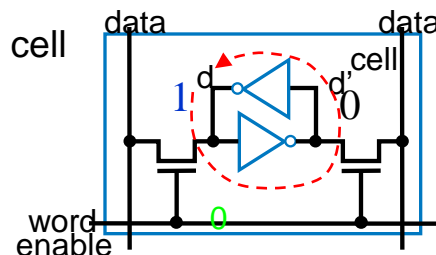


SRAM cell

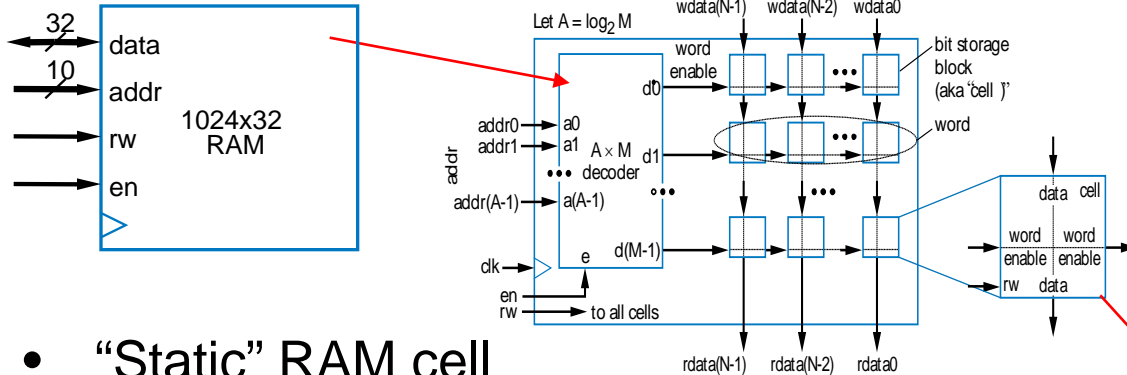


“Static” RAM cell

- 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



Static RAM (SRAM)



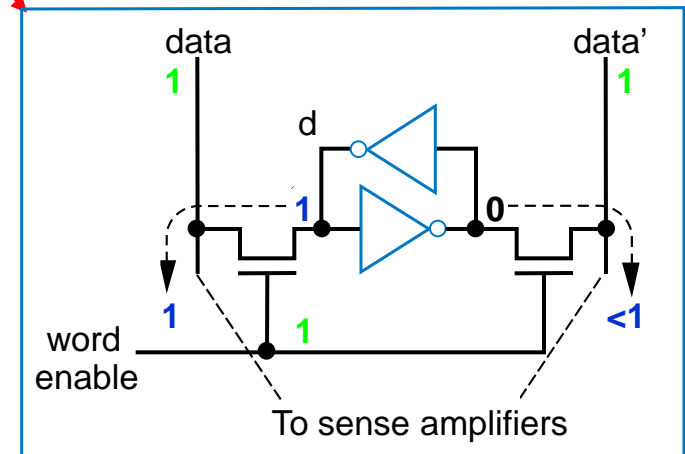
- “Static” RAM cell

- Reading this cell

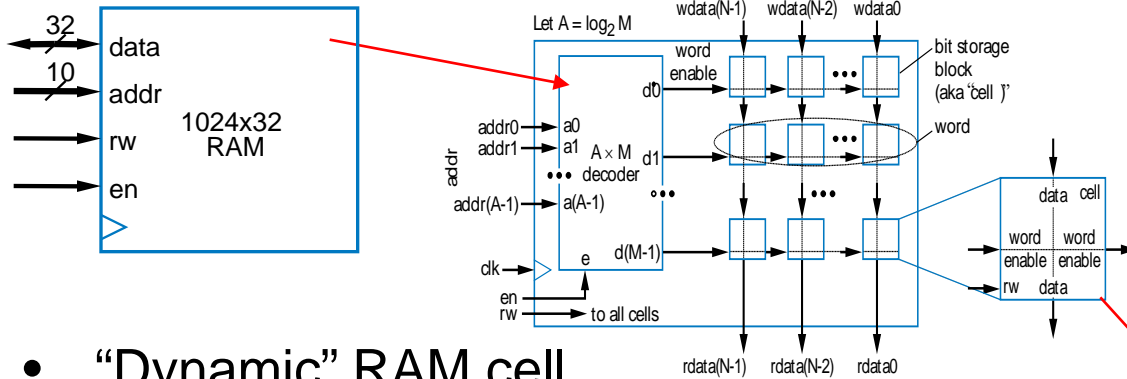
- Somewhat trickier
 - When rw set to read, the RAM logic sets both *data* and *data'* to 1
 - The stored bit *d* will pull either the left line or the right bit down slightly below 1
 - “Sense amplifiers” detect which side is slightly pulled down

- The electrical description of SRAM is really beyond our scope – just general idea here, mainly to contrast with *DRAM*...

SRAM cell

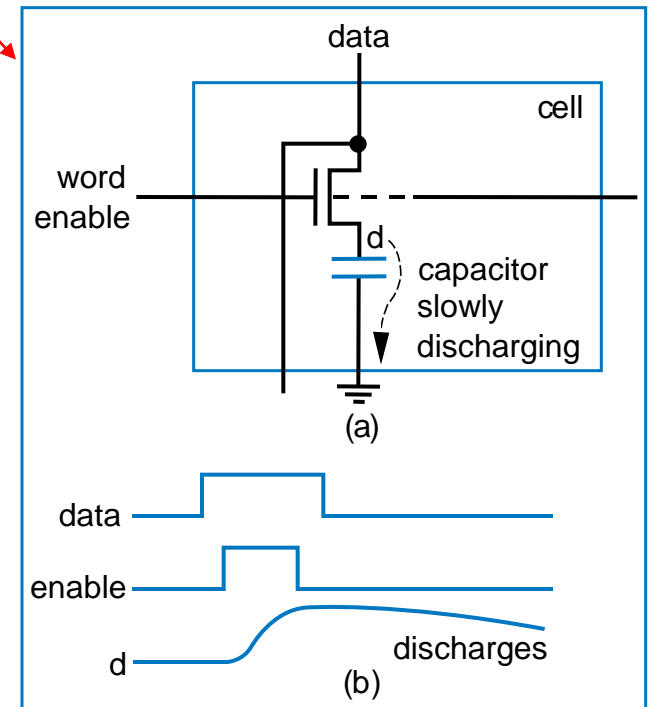


Dynamic RAM (DRAM)



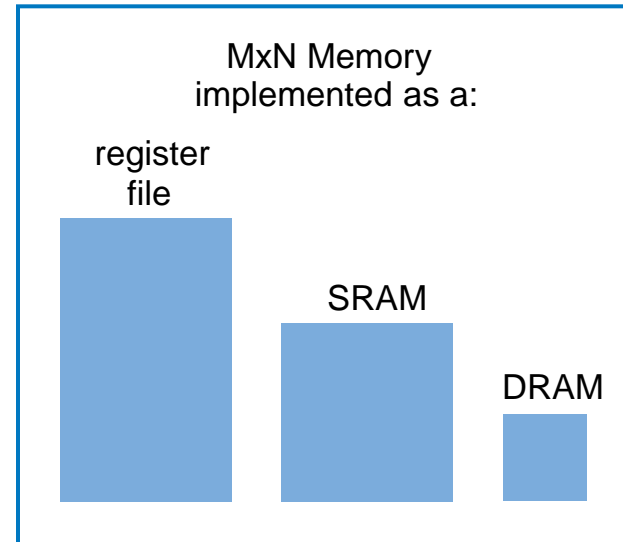
- “Dynamic” RAM cell
 - 1 transistor (rather than 6)
 - Relies on *large* capacitor to store bit
 - Write: Transistor conducts, data voltage level gets stored on top plate of capacitor
 - Read: Just look at value of d
 - Problem: Capacitor discharges over time
 - Must “refresh” regularly, by reading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

DRAM cell



Comparing Memory Types

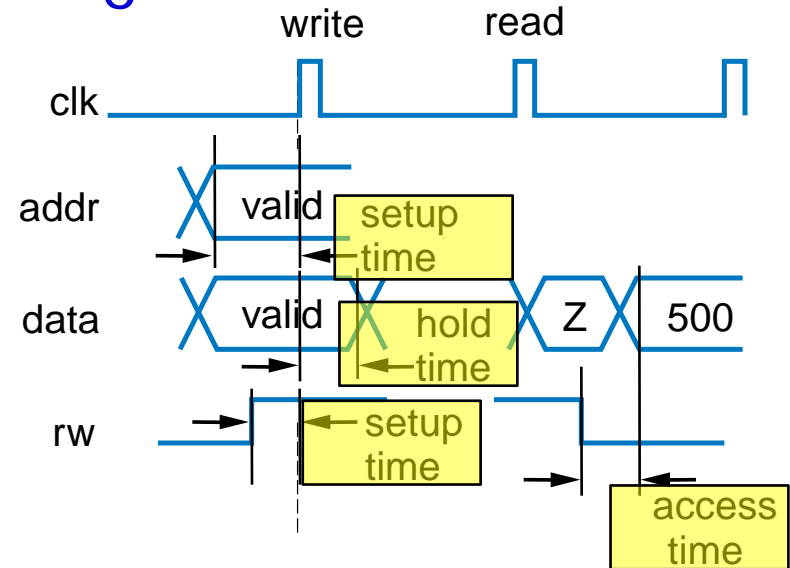
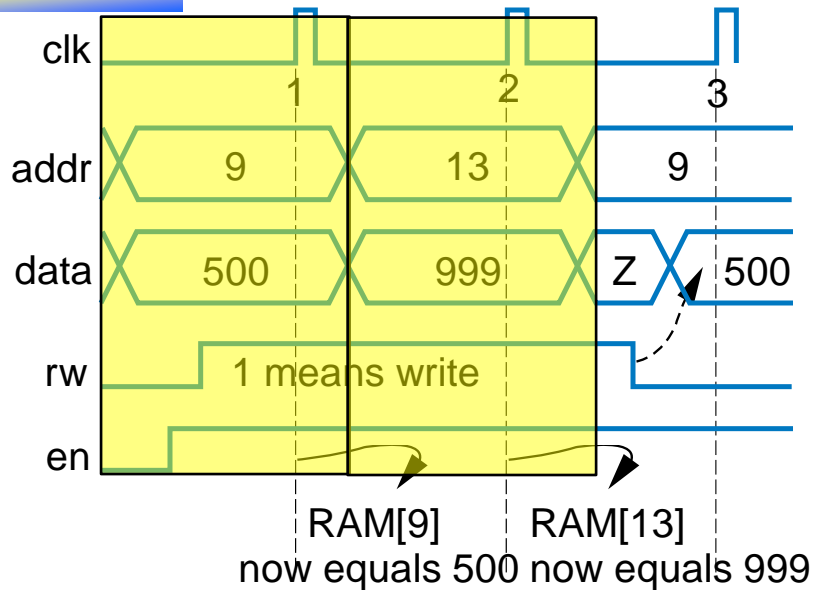
- Register file
 - Fastest
 - But biggest size
- SRAM
 - Fast
 - More compact than register file
- DRAM
 - Slowest
 - And refreshing takes time
 - But very compact
- Use register file for small items, SRAM for large items, and DRAM for huge items
 - Note: DRAM's big capacitor requires a special chip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



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



Reading and Writing a RAM

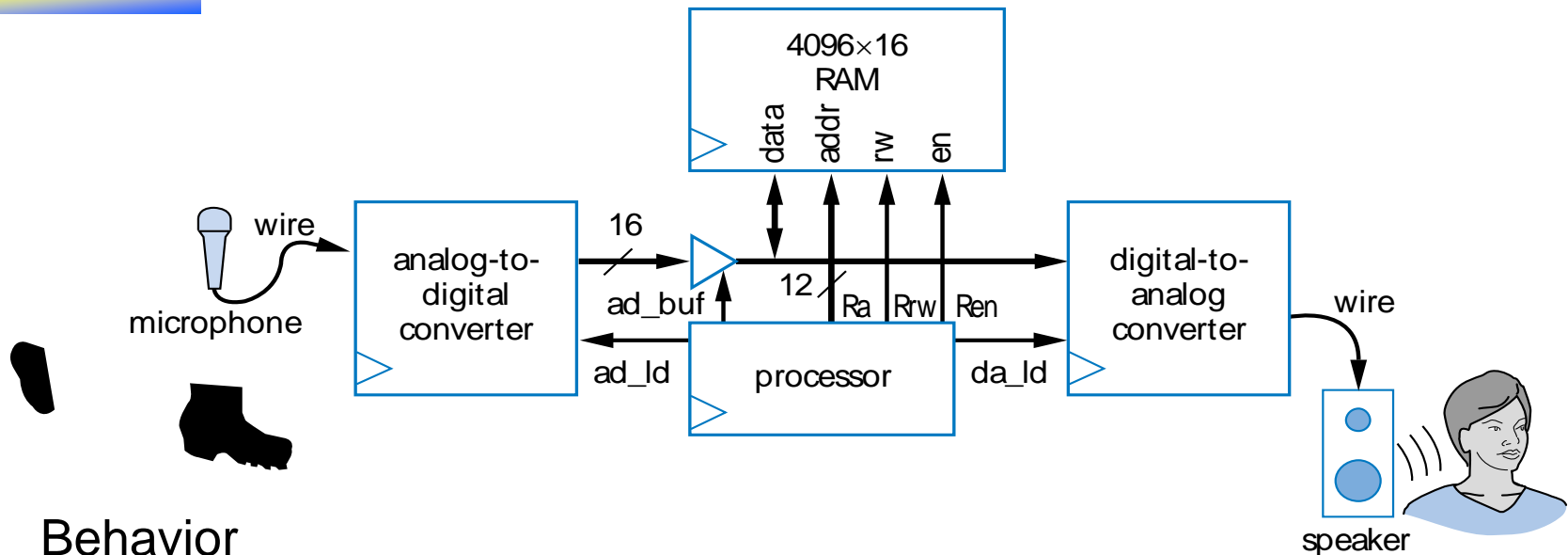


- Writing
 - Put address on *addr* lines, data on *data* lines, set *rw*=1, *en*=1
- Reading
 - Set *addr* and *en* lines, but put nothing (Z) on *data* lines, set *rw*=0
 - Data will appear on *data* lines
- Don't forget to obey setup and hold times
 - In short – keep inputs stable before and after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

(b)



RAM Example: Digital Sound Recorder

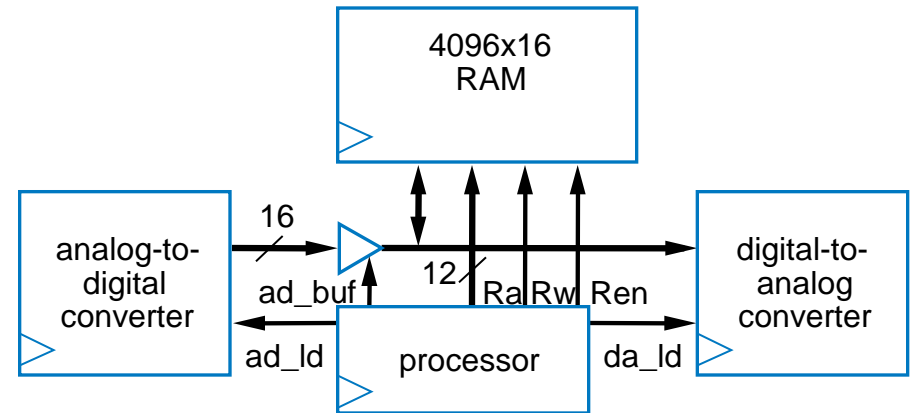


- Behavior
 - Record: Digitize sound, store as series of 4096 12-bit digital values in RAM
 - We'll use a 4096x16 RAM (12-bit wide RAM not common)
 - Play back later
 - Common behavior in telephone answering machine, toys, voice recorders
- To record, processor should read a-to-d, store read values into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



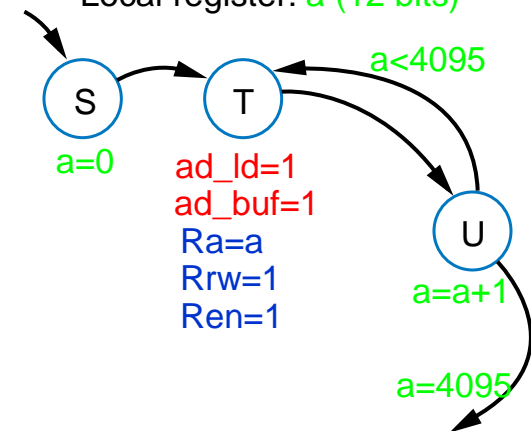
RAM Example: Digital Sound Recorder

- 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_id*=1, *ad_buf*=1
 - Write to RAM at address *a*
 - » *Ra*=*a*, *Rrw*=1, *Ren*=1



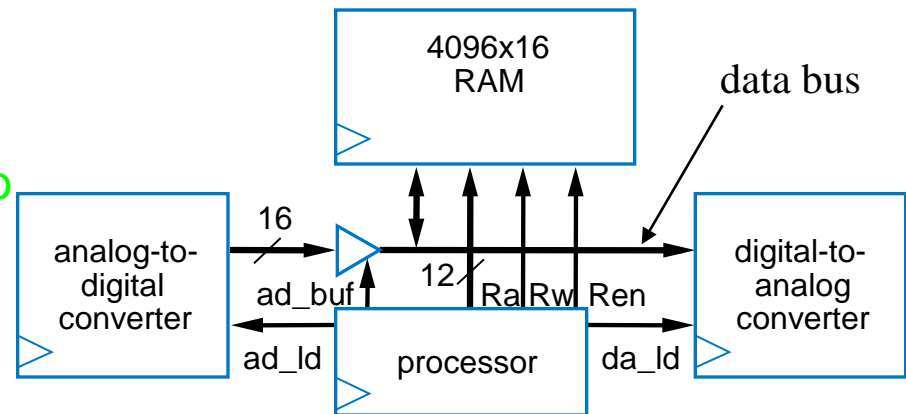
Record behavior

Local register: *a* (12 bits)

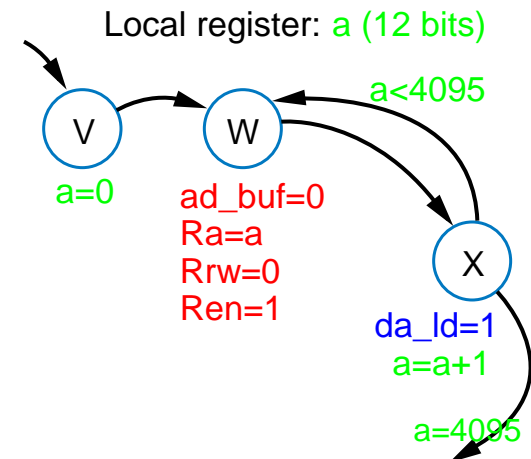


RAM Example: Digital Sound Recorder

- 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

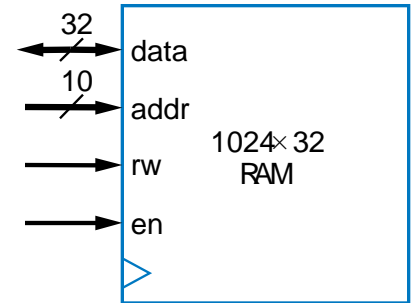


Play behavior

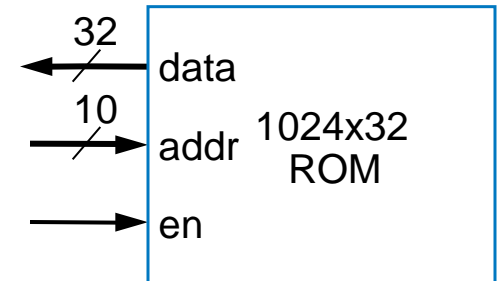


Read-Only Memory – ROM

- 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



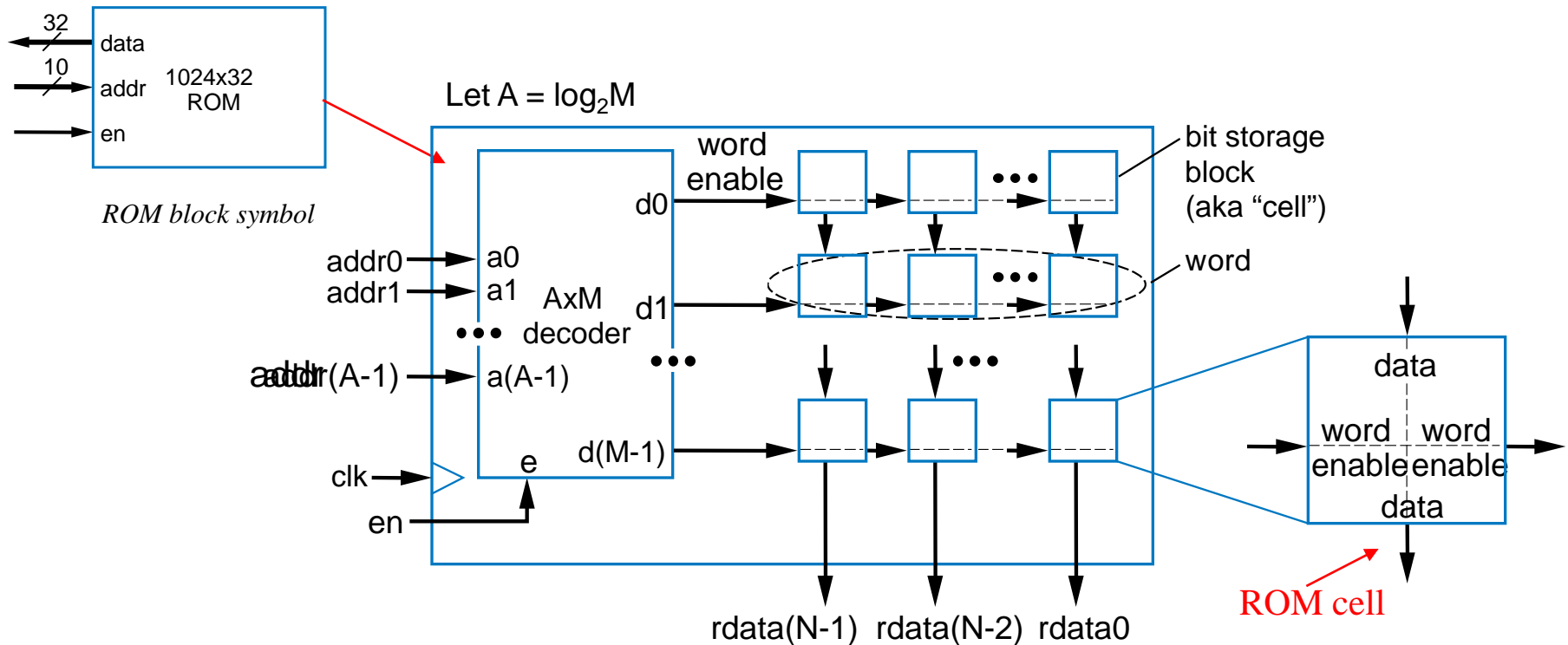
RAM block symbol



ROM block symbol



Read-Only Memory – ROM

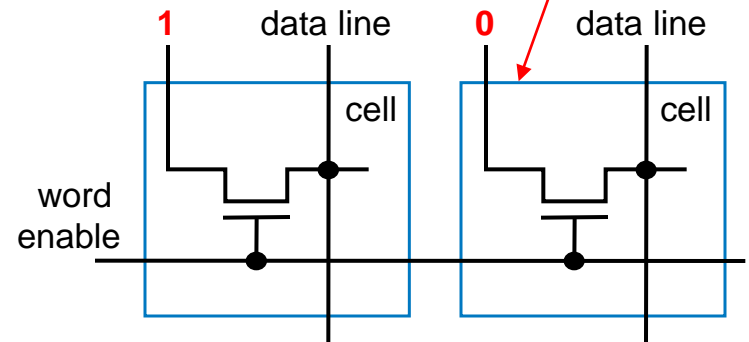
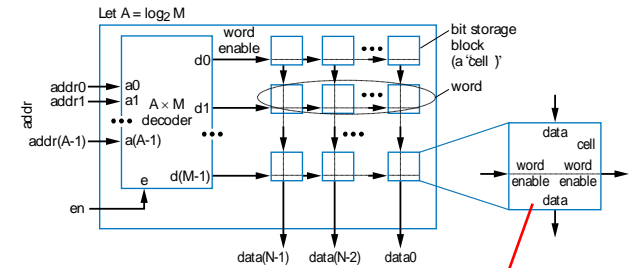


- Internal logical structure similar to RAM, without the data input lines



ROM Types

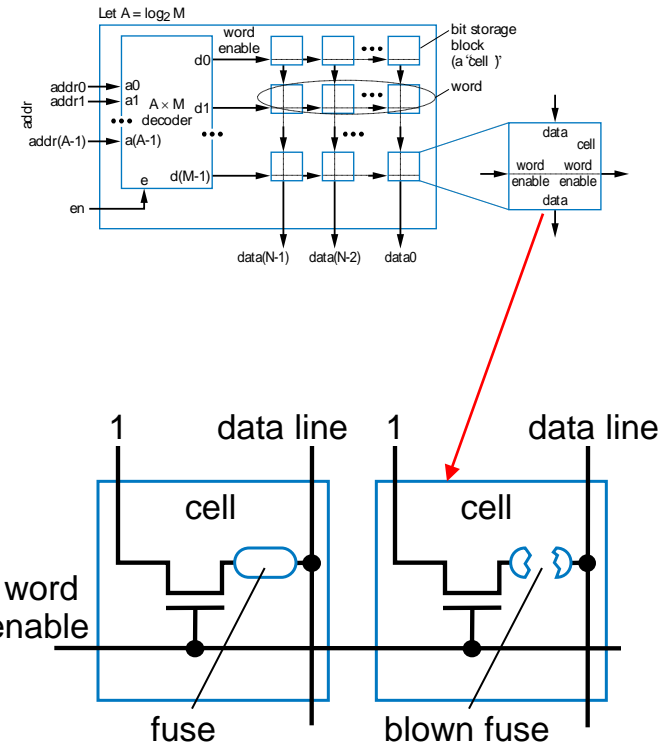
- If a ROM can only be read, how are the stored bits stored in the first place?
 - Storing bits in a ROM known as *programming*
 - Several methods
- **Mask-programmed ROM**
 - Bits are hardwired as 0s or 1s during chip manufacturing
 - 2-bit word on right stores “10”
 - word enable (from decoder) simply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



ROM Types

- **Fuse-Based Programmable ROM**

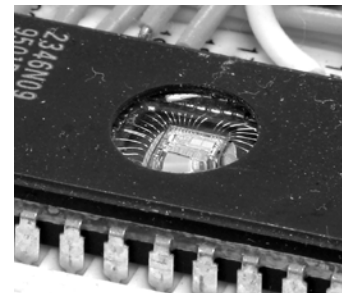
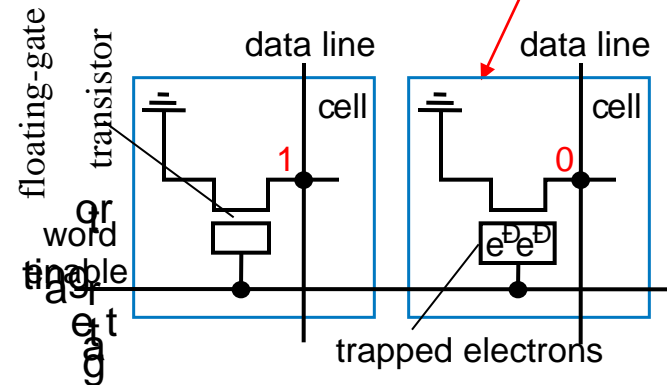
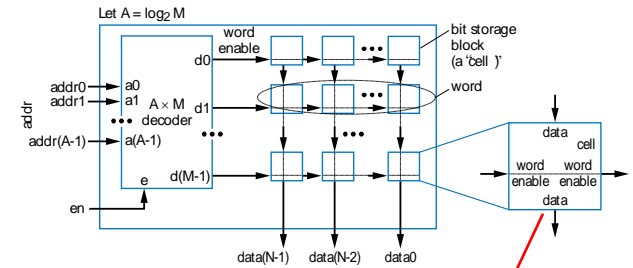
- Each cell has a fuse
- A special device, known as a programmer, blows certain fuses (using higher-than-normal voltage)
 - Those cells will be read as 0s (involving some special electronics)
 - Cells with unblown fuses will be read as 1s
 - 2-bit word on right stores “10”
- Also known as **One-Time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



ROM Types

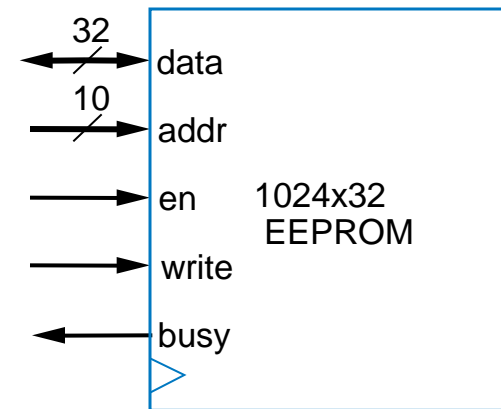
- **Erasable Programmable ROM (EPROM)**

- 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



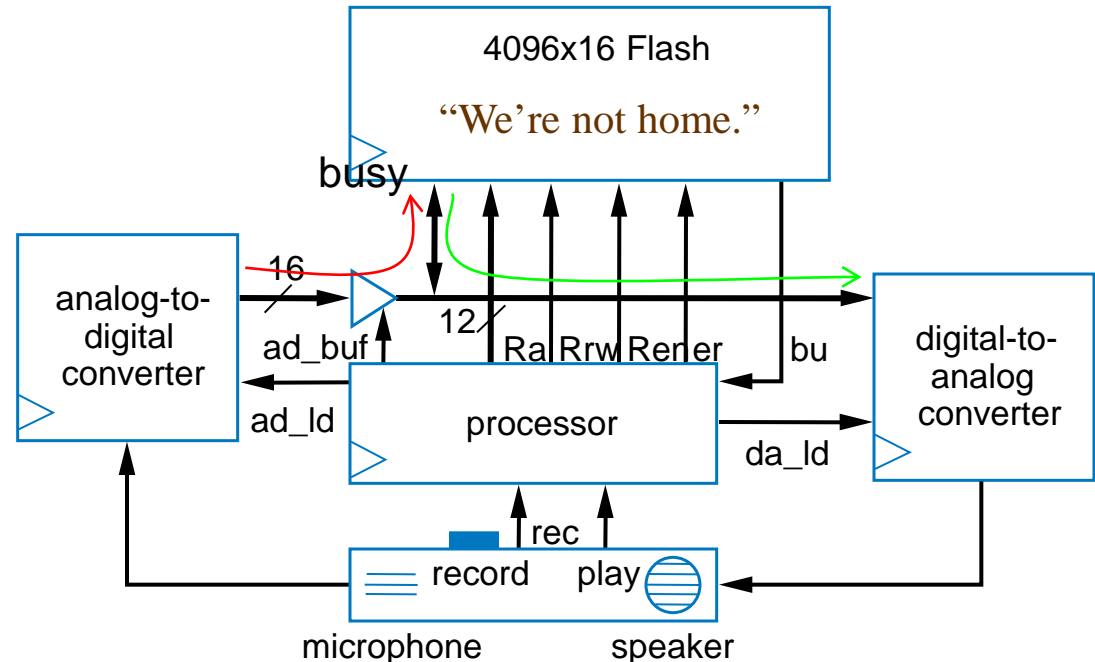
ROM Types

- **Electrically-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



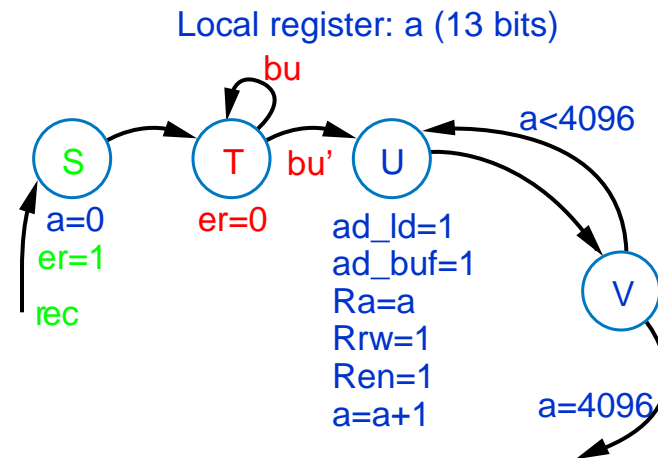
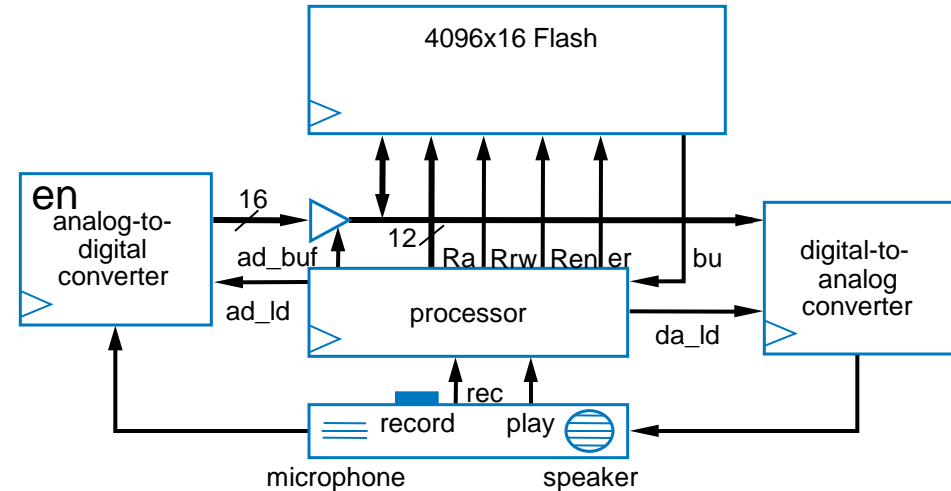
ROM Example: Digital Telephone Answering Machine Using a Flash Memory

- 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



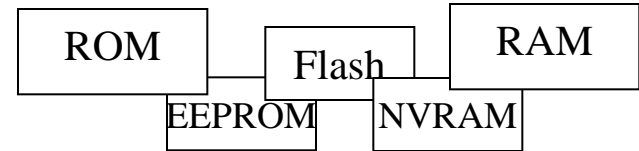
ROM Example: Digital Telephone Answering Machine Using a Flash Memory

- 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



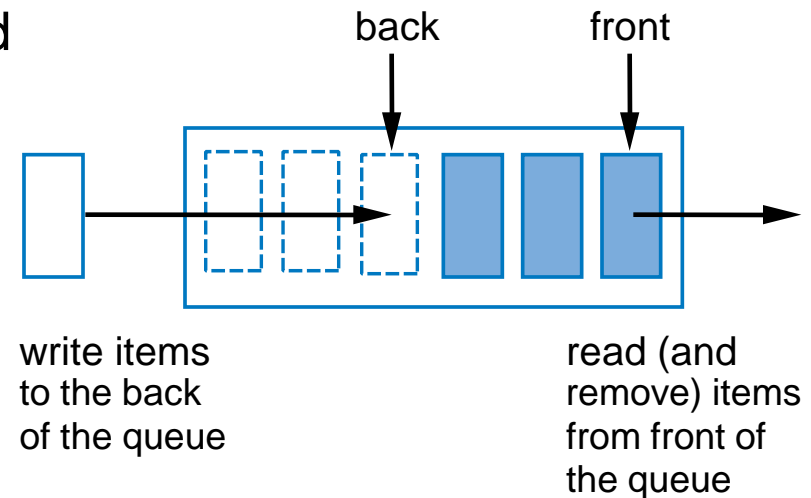
Blurring of Distinction Between ROM and RAM

- 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, and 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)

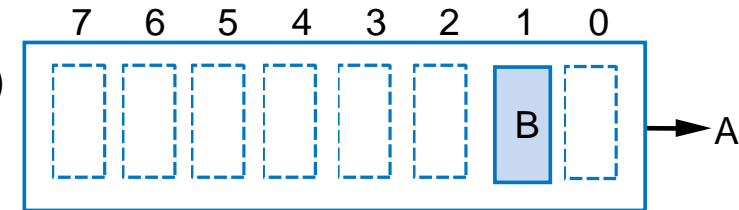


- 



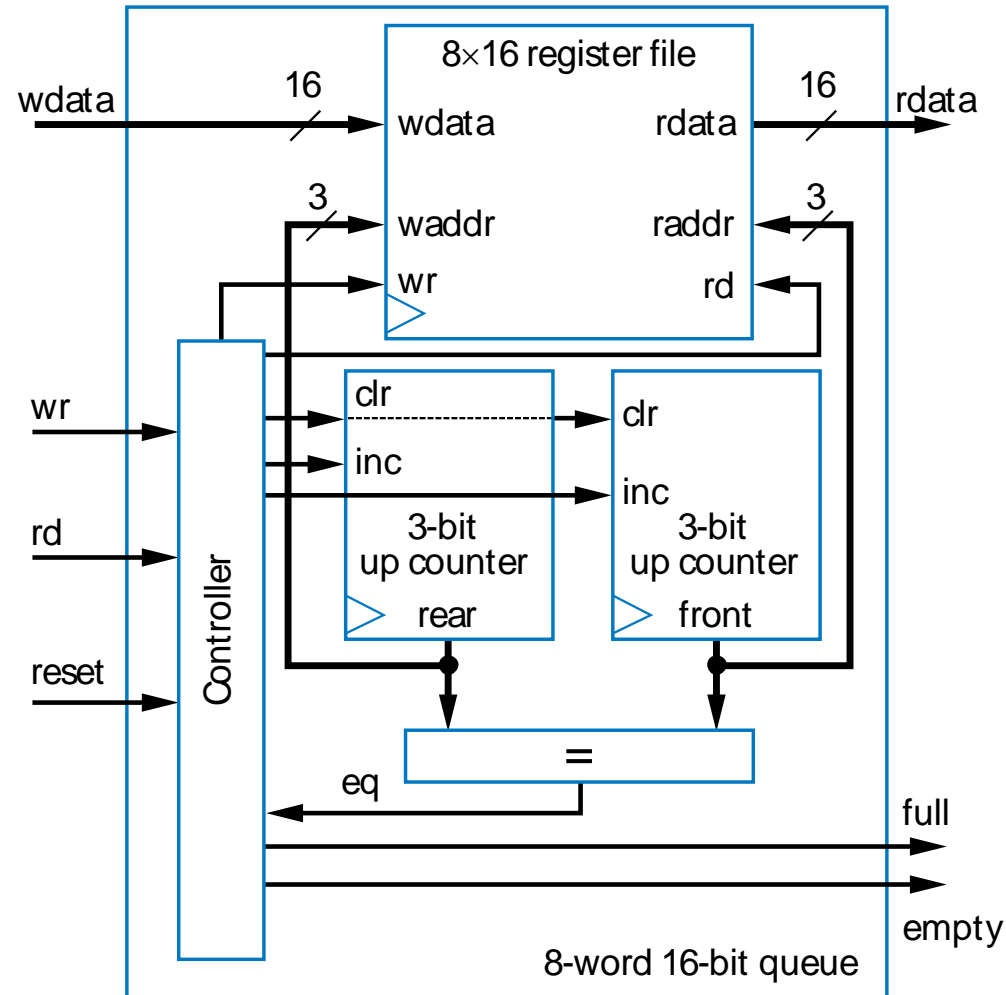
Queues

- 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



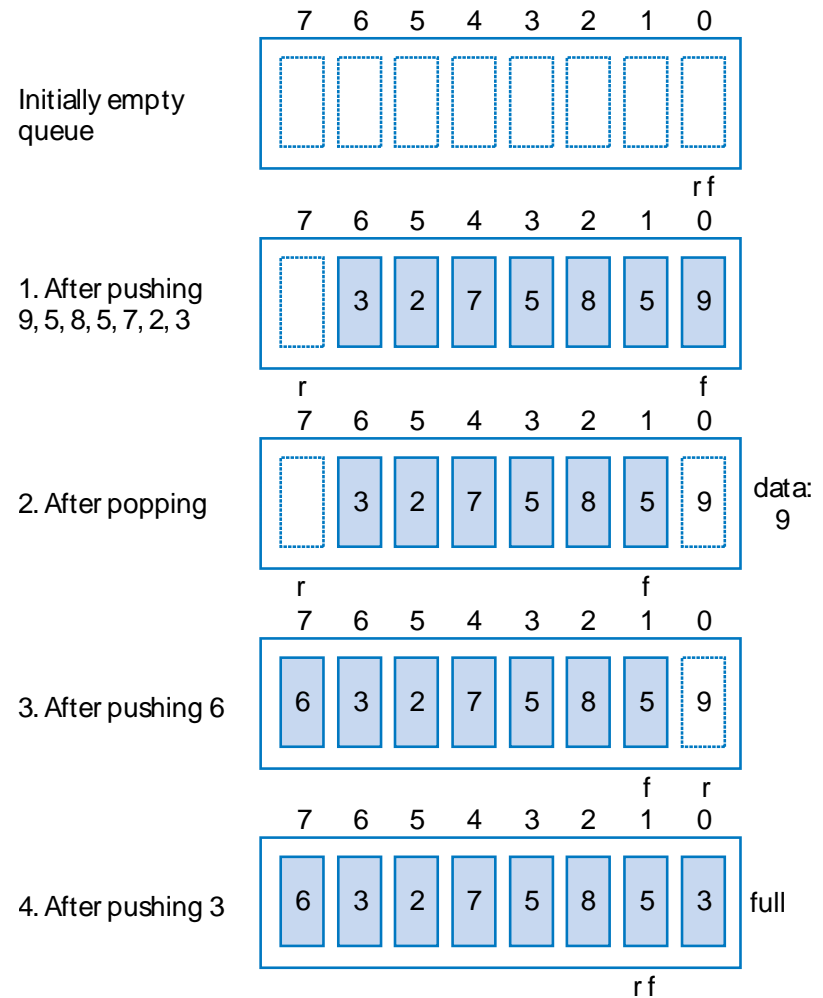
Common Uses of a Queue

- 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



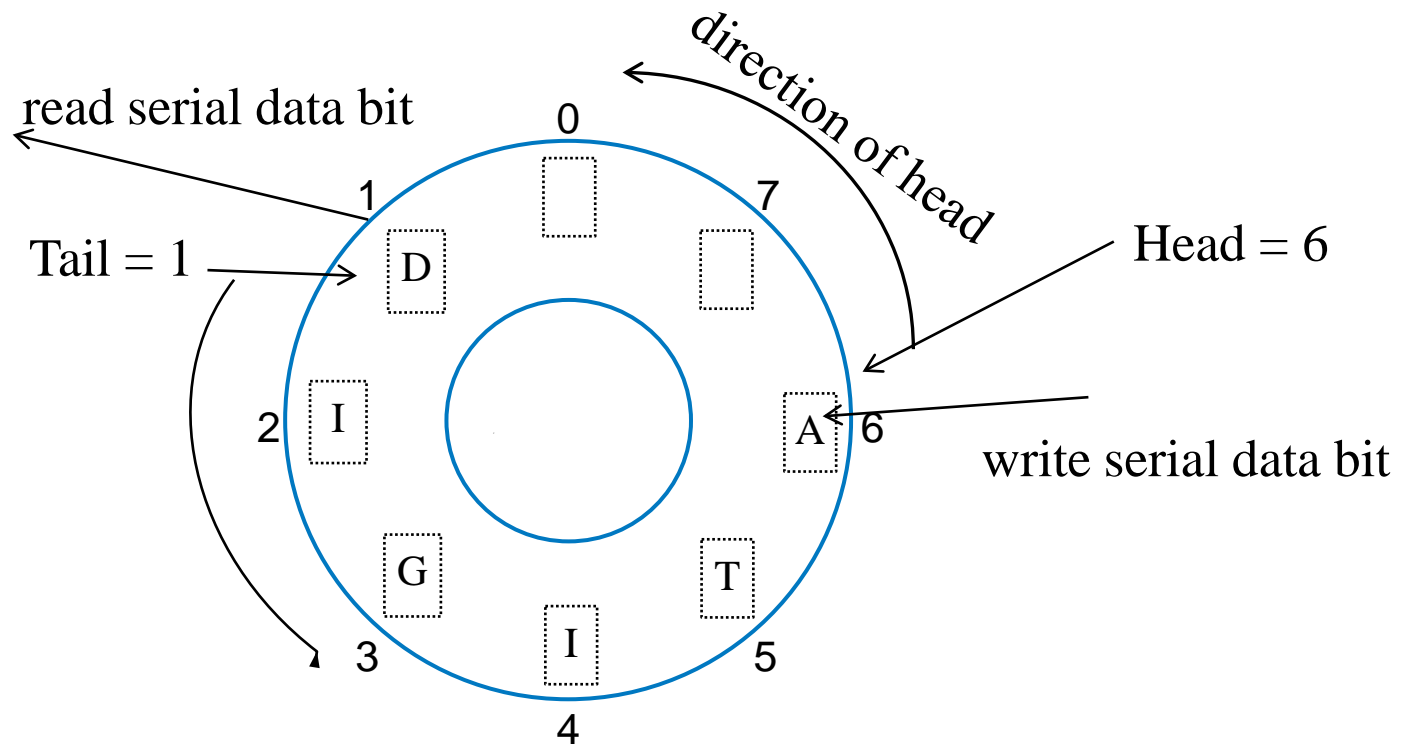
Queue Usage Example

- 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



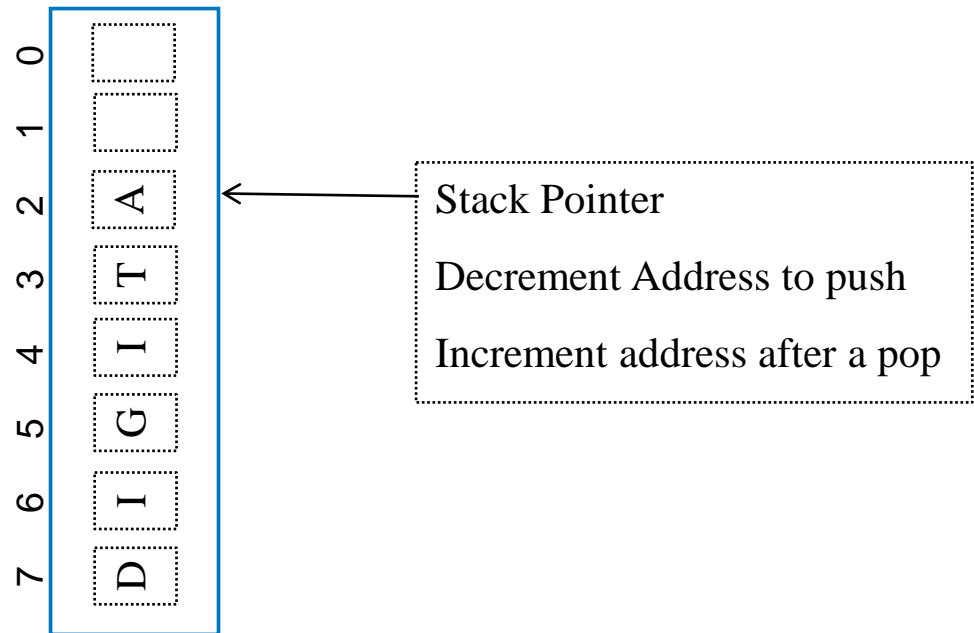
Circular Que (Buffer)

- Commonly used as temporary storage for serial data transfer
- Head pointer (address) must not go past tail
- Reading comes in bursts and is faster than write



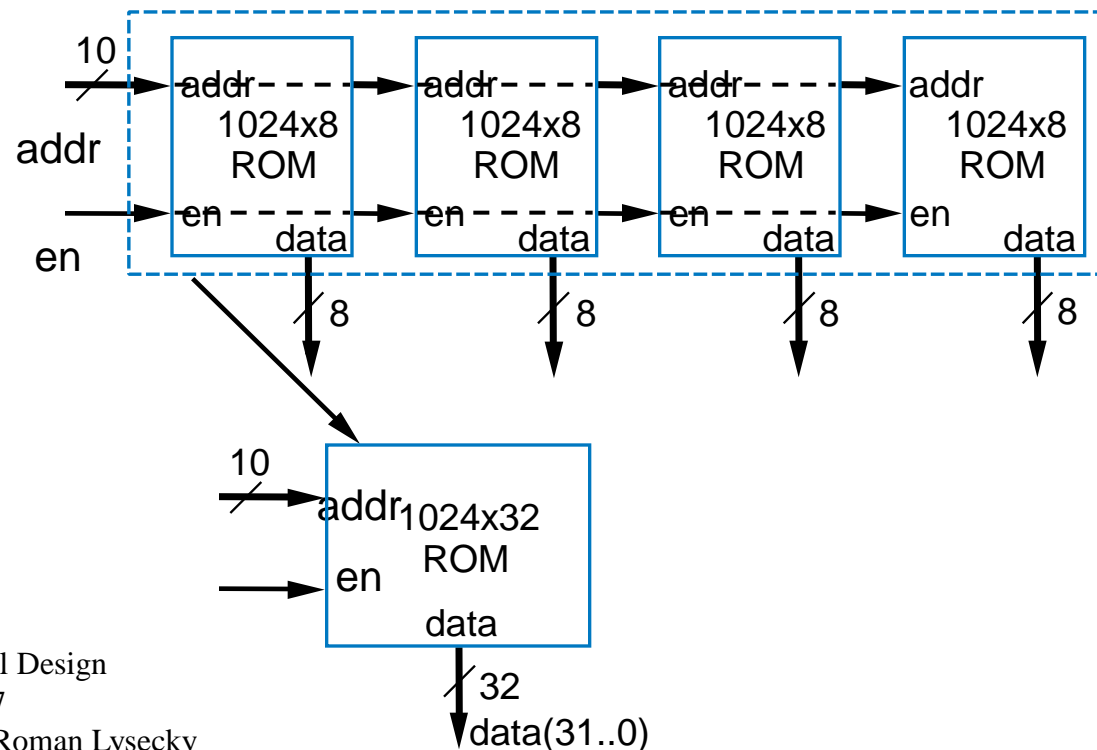
Stack

- Used in computers to temporarily hold data that will be needed again shortly
- Data is pushed (written) on to the stack and popped (read) off the stack
- First in Last out (FILO)
- May push from low to high address or vice versa



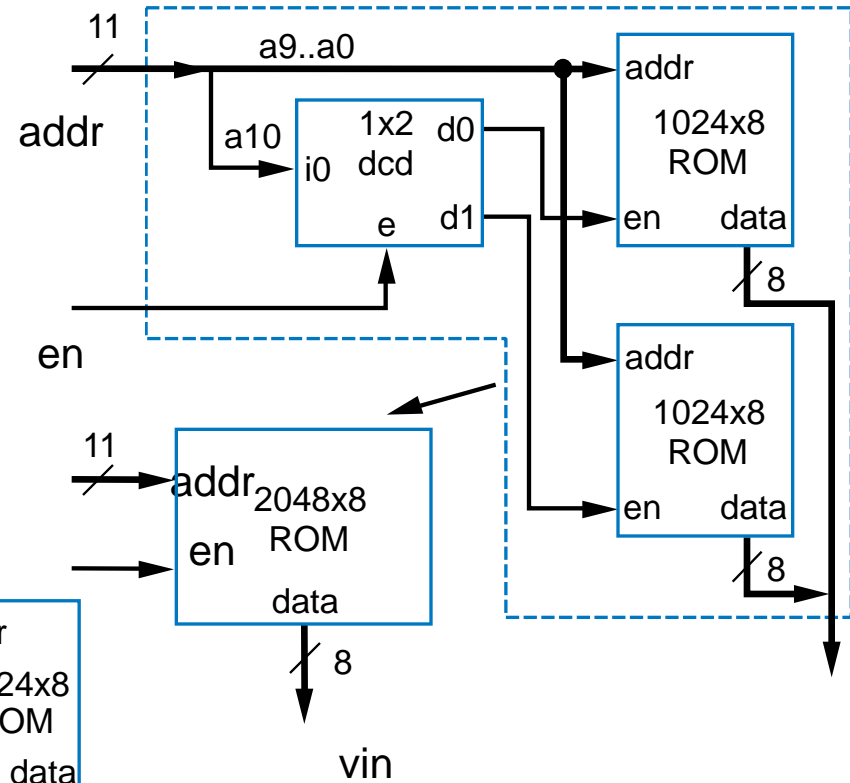
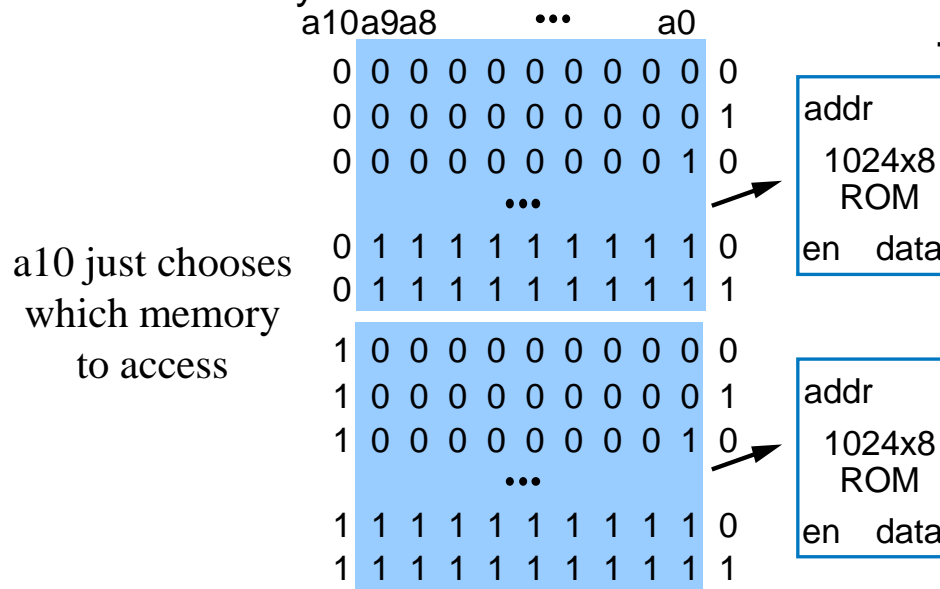
Hierarchy and Composing Larger Components from Smaller Versions

- 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



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



Memory

Verilog



RTL Example of ROM Using Verilog

onRTL Description of a ROM Verilog Example

```
/*  
* ROM_RTL.V  
* Behavioral Example of 16x4 ROM  
*/
```

```
module rom_rtl(ADDR, DATA) ;  
input [3:0] ADDR ;  
output [3:0] DATA ;  
  
reg [3:0] DATA ;  
  
// A memory is implemented  
// using a case statement
```

```
always @(ADDR)  
begin  
case (ADDR)  
4'b0000 : DATA = 4'b0000 ;  
4'b0001 : DATA = 4'b0001 ;  
4'b0010 : DATA = 4'b0010 ;  
4'b0011 : DATA = 4'b0100 ;  
4'b0100 : DATA = 4'b1000 ;  
4'b0101 : DATA = 4'b1000 ;  
4'b0110 : DATA = 4'b1100 ;  
4'b0111 : DATA = 4'b1010 ;  
4'b1000 : DATA = 4'b1001 ;  
4'b1001 : DATA = 4'b1001 ;  
4'b1010 : DATA = 4'b1010 ;  
4'b1011 : DATA = 4'b1100 ;  
4'b1100 : DATA = 4'b1001 ;  
4'b1101 : DATA = 4'b1001 ;  
4'b1110 : DATA = 4'b1101 ;  
4'b1111 : DATA = 4'b1111 ;  
endcase  
end
```

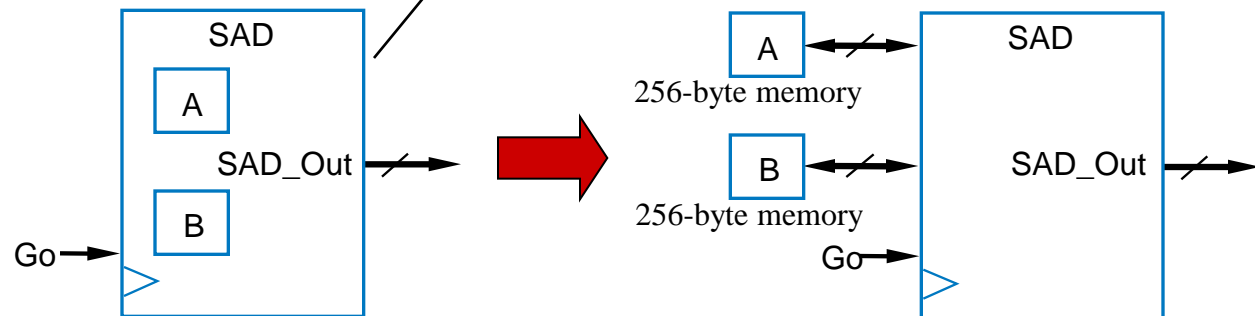
```
endmodule
```



Accessing Memory

- Memory may be initially accessed as array for simplicity
 - SAD example
 - Declared two 256-item arrays
- Eventually, may want to describe memory as external component
 - Closer to real implementation

```
module SAD(Go, SAD_Out);  
  
    input Go;  
    output reg [31:0] SAD_Out;  
  
    reg [7:0] A [0:255];  
    reg [7:0] B [0:255];
```



Simple Memory Entity

- Simple read-only memory
 - Addr and data ports only
 - Declares array named Memory for storage
 - Procedure uses continuous assignment statement to always output Memory data element corresponding to current address input value

```
`timescale 1 ns/1 ns

module SADMemo(Addr, Data);

    input [7:0] Addr;
    output [7:0] Data;

    reg [7:0] Memory [0:255];

    assign Data = Memory[Addr];
endmodule
```

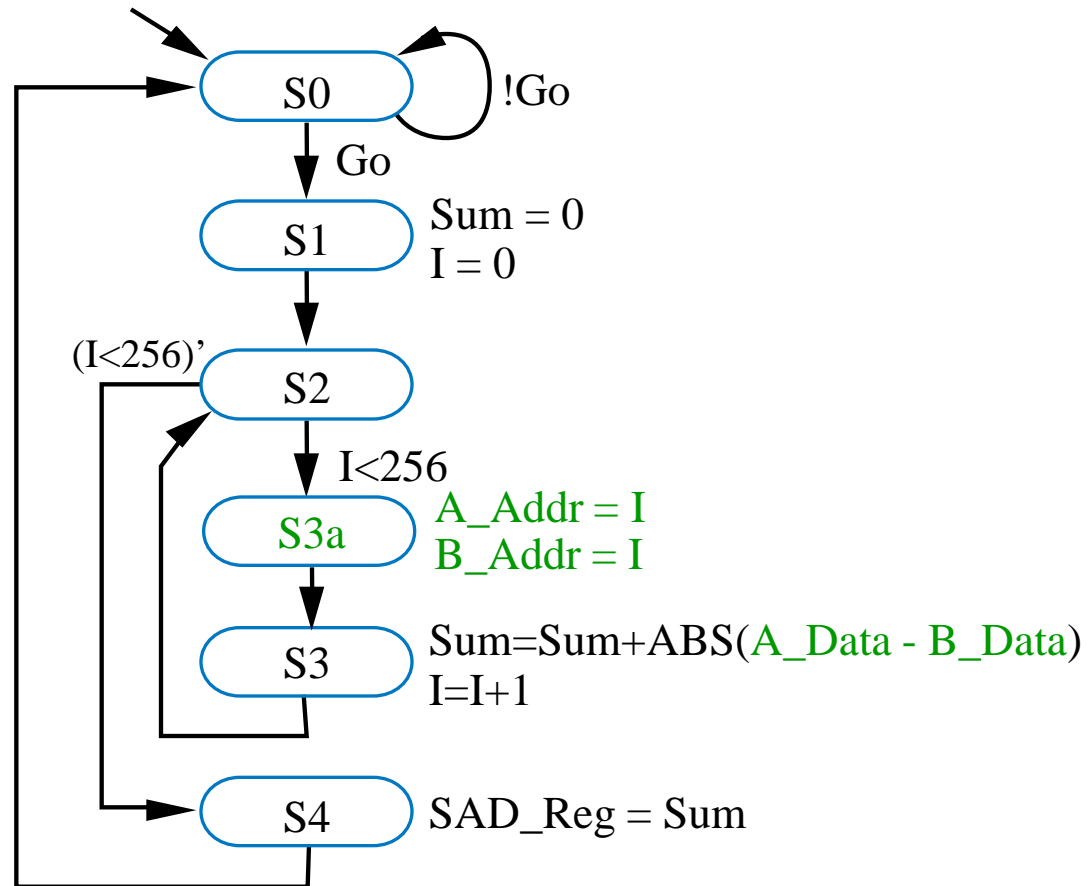


Accessing Memory

- Modify SAD's (some of absolute differences) HLASM with extra state (S3a) for reading the memories

Local registers: Sum, SAD_Reg (32 bits); I (integer)

- Extra state necessary due to use of fully-synchronous HLASM (modeled as one procedure sensitive only to clock)
- A_addr & B_addr are output ports connected to memory address inputs
- A_data and B_data are input ports connected to memory data outputs



```

        always @(posedge Clk) begin
            if (Rst==1) begin
                A_Addr <= 0;
                B_Addr <= 0;
                State <= S0;
                Sum <= 0;
                SAD_Reg <= 0;
                I <= 0;
            end
            else begin
                case (State)
                    S0: begin
                        if (Go==1)
                            State <= S1;
                        else
                            State <= S0;
                    end
                    S1: begin
                        Sum <= 0;
                        I <= 0;
                        State <= S2;
                    end
                    S2: begin
                        if (!(I==255))
                            State <= S3a;
                        else
                            State <= S4;
                    end
                endcase
            end
        end

        S3a: begin
            A_Addr <= I;
            B_Addr <= I;
            State <= S3;
        end
        S3: begin
            Sum <= Sum +
                ABSDiff(A_Data, B_Data)
            I <= I + 1;
            State <= S2;
        end
        S4: begin
            SAD_Reg <= Sum;
            State <= S0;
        end
    endcase
end
endmodule

assign SAD_Out = SAD_Reg;

function [7:0] ABSDiff;
    input [7:0] A;
    input [7:0] B;
    begin
        if (A>B) ABSDiff = A - B;
        else ABSDiff = B - A;
    end
endfunction

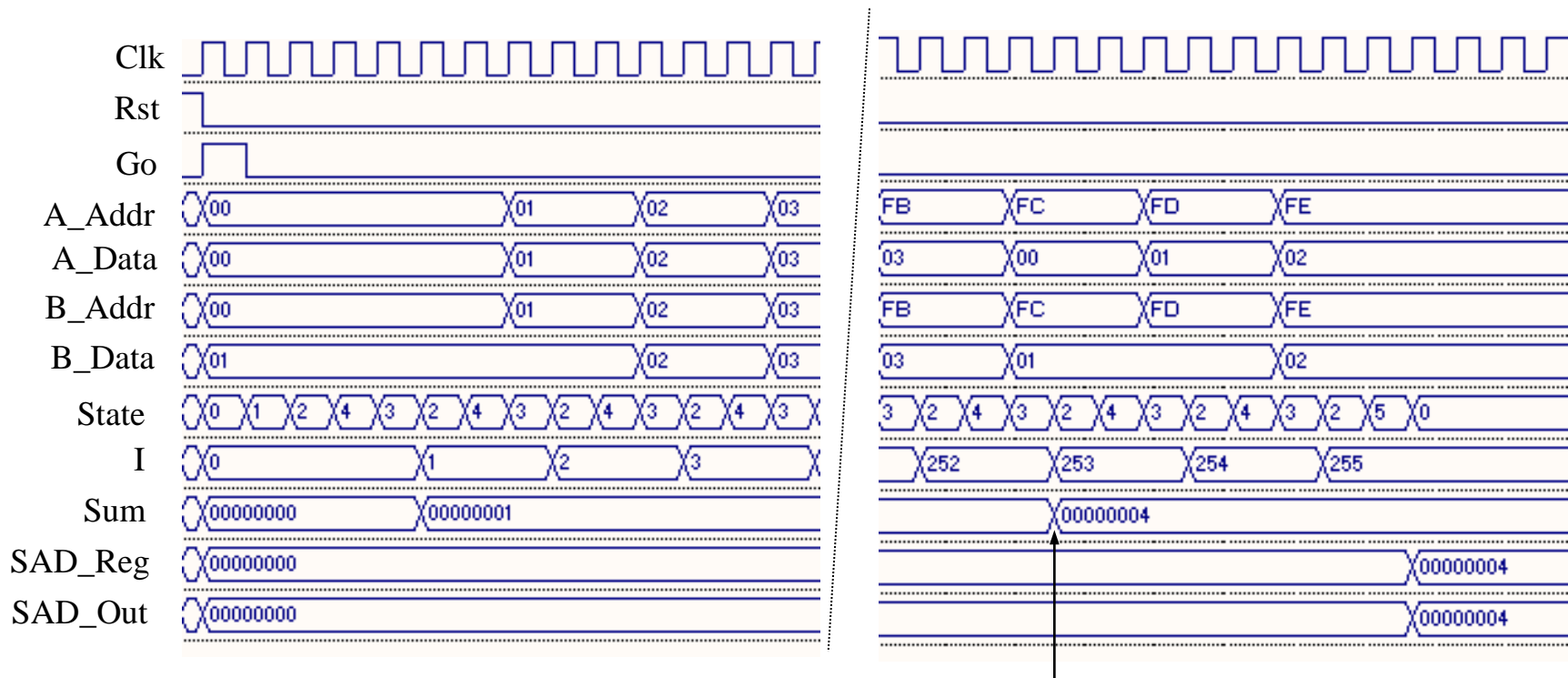
```

Created AbsDiff function for improved readability

Note: Violated our own guideline of using begin-end block with if-else statements, to save space in this figure

Waveforms

- Waveforms now include address and data lines with memories
 - We also added some internal signals, State, I, Sum, and SAD_Reg
- SAD_Out result appears later than previously, due to extra state



15390 ns

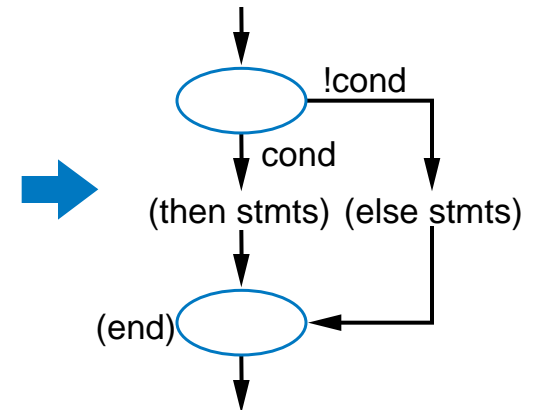


Converting from C to High-Level State Machine

- *If-then-else*

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

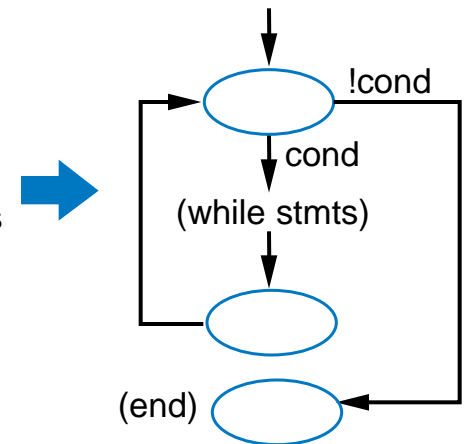
```
if (cond) {  
    // then stmts  
}  
else {  
    // else stmts  
}
```



- *While loop* statement

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

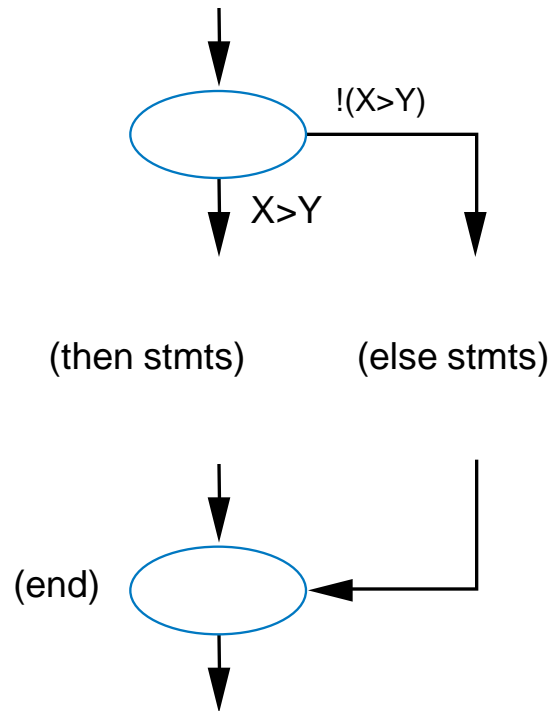
```
while (cond) {  
    // while stmts  
}
```



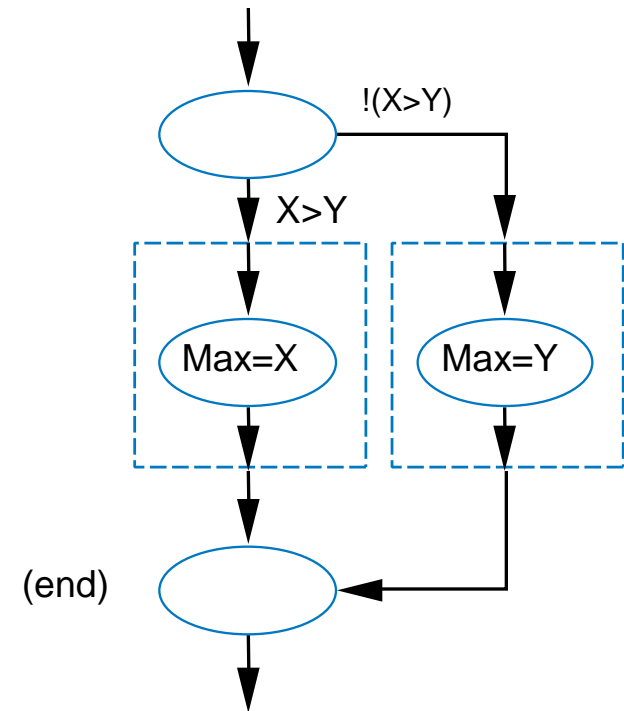
Simple Example of Converting from C to High-Level State Machine

Inputs: uint X, Y
Outputs: uint Max

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



(a)



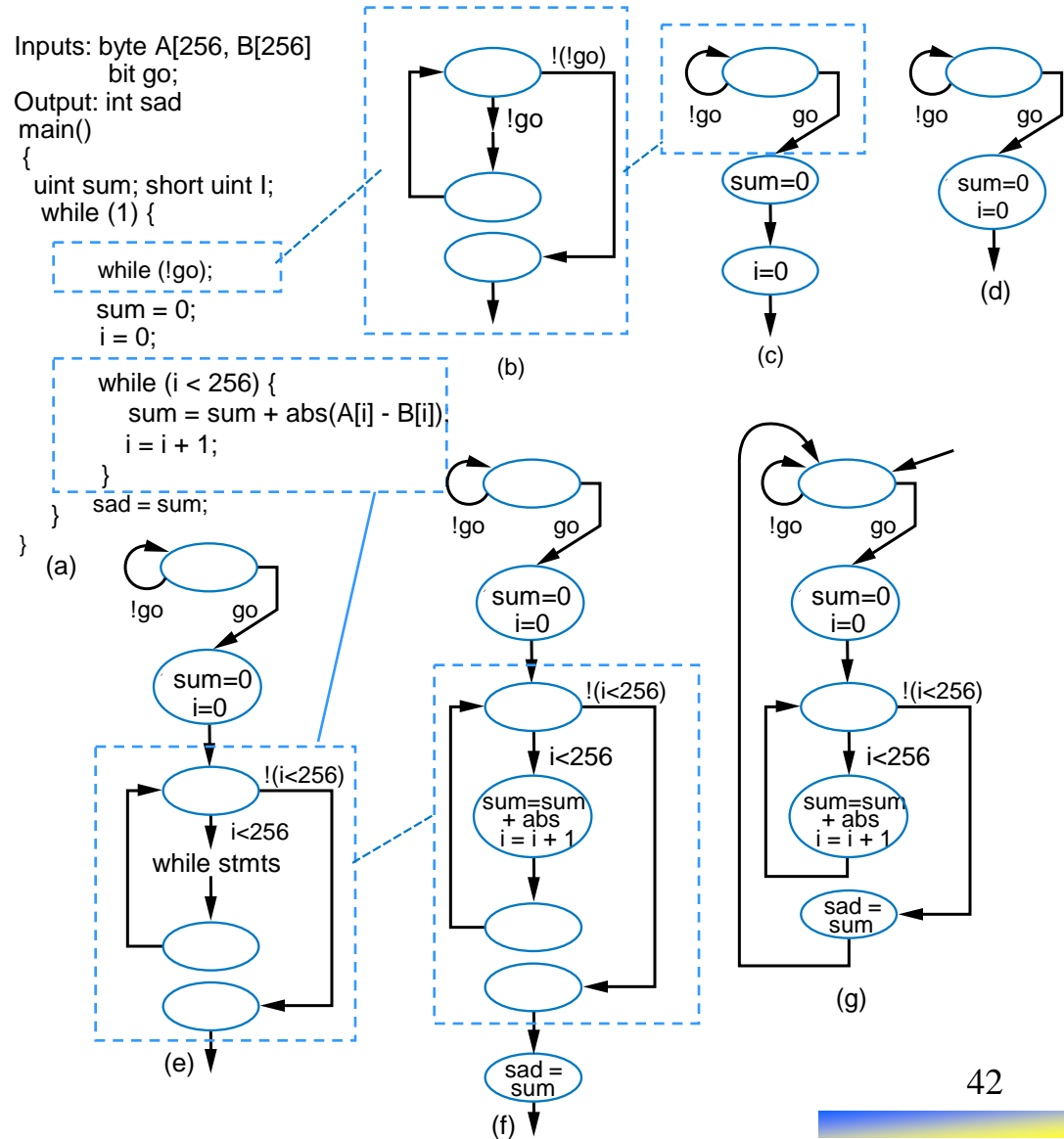
(c)

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



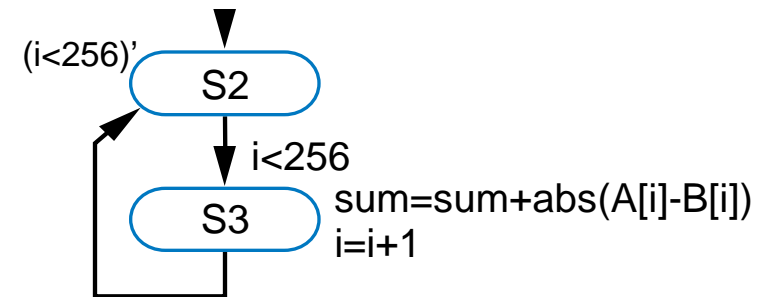
Example: Converting Sum-of-Absolute-Differences C code to High-Level State Machine

- 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



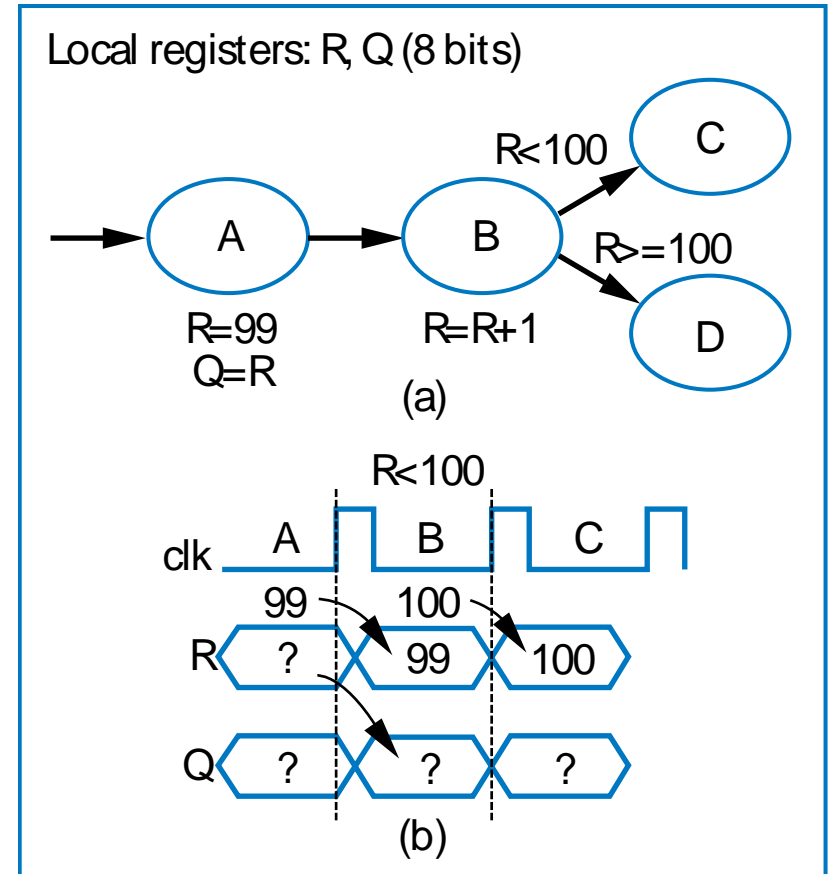
RTL Example: Video Compression – Sum of Absolute Differences

- 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

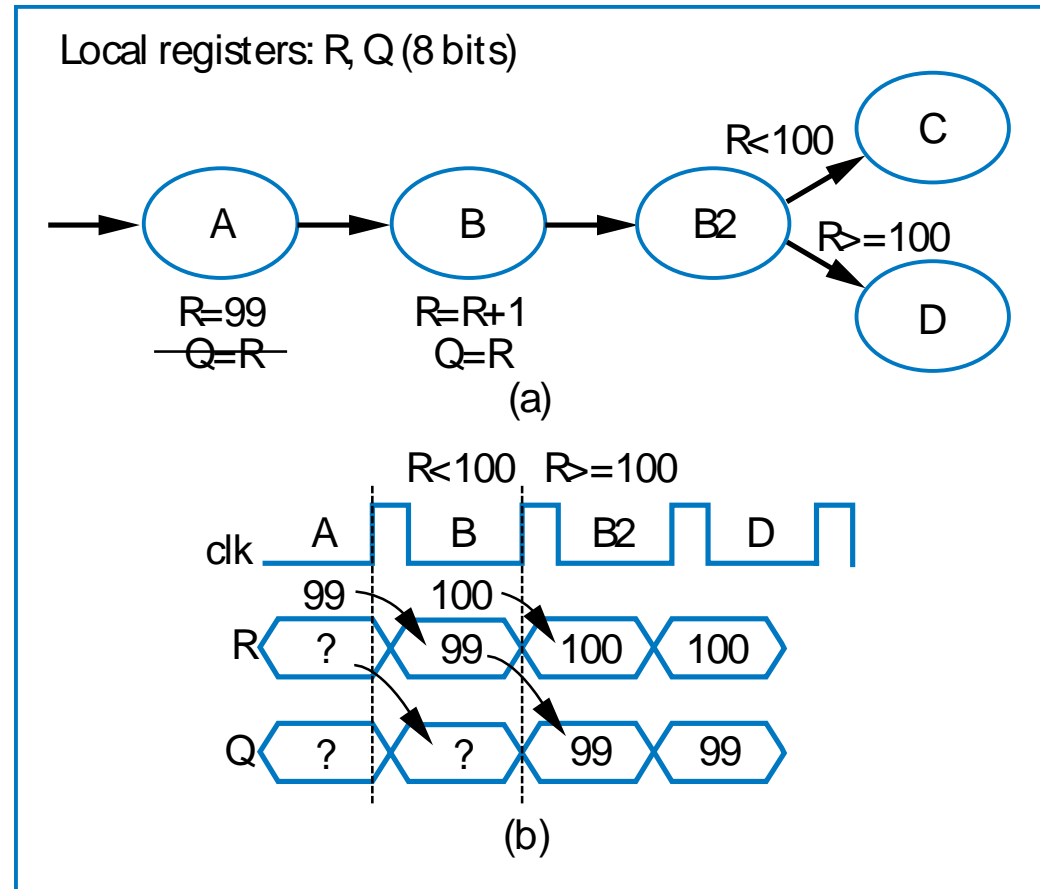
- Common pitfall: Assuming register is update in the state it's written
 - Final value of Q ?
 - Final state?
 - Answers may surprise you
 - Value of Q unknown
 - Final state is **C**, not **D**
 - Why?
 - State **A**: $R=99$ and $Q=R$ happen simultaneously
 - State **B**: R not updated with $R+1$ until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



RTL Design Pitfalls and Good Practice

- Solutions

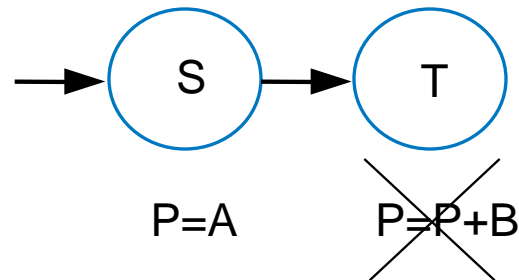
- Read register in following state ($Q=R$)
- Insert extra state so that conditions use updated value
- Other solutions are possible, depends on the example



RTL Design Pitfalls and Good Practice

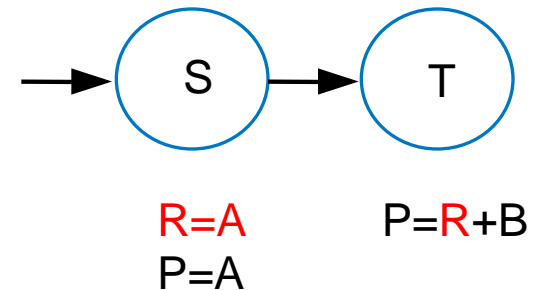
- Common pitfall: Reading outputs
 - Outputs can only be written
 - Solution: Introduce additional register, which can be written and read

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



(a)

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

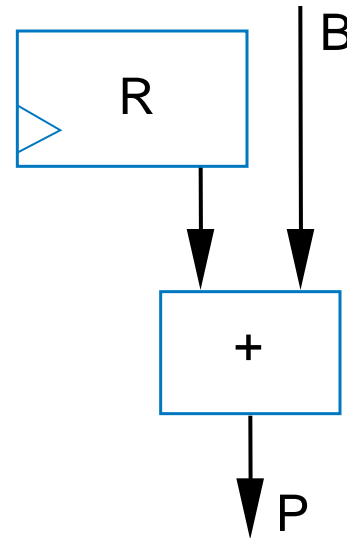


(b)

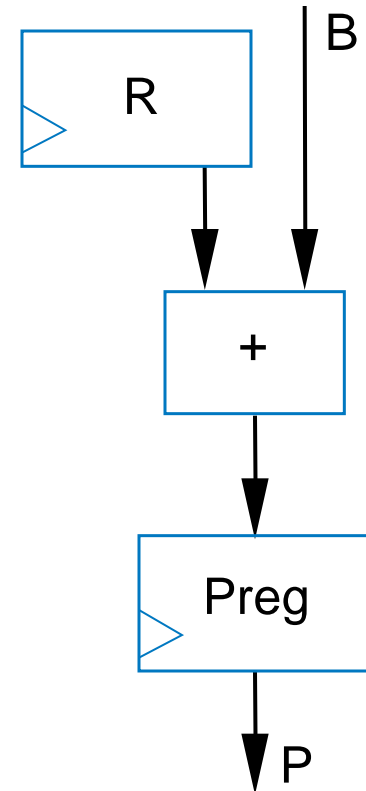


RTL Design Pitfalls and Good Practice

- 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



(a)



(b)

