



Introduction to Digital Design

Week 12: Register Memory
Components and FIFO

Yao Zheng
University of Hawai'i at Mānoa
Department of Electrical and Computer Engineering

1

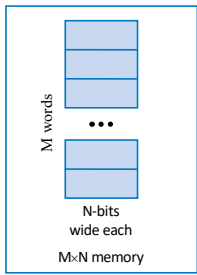
Overview

- Memory components
 - RAM: readable and writable memory
 - SRAM vs DRAM
 - ROM: read-only memory
 - Erasable Programmable ROM vs Flash Memory
- Queues
 - FIFO: A list written to at the back, from read from the front
 - Common Uses of a Queue
- Multiple processors
 - Use multiple processor to design complicated systems
 - Interfacing between processors
- Hierarchy
 - Design philosophy: hierarchy helps us manage complexity
 - Memory hierarchy (optional)

2

Memory Components

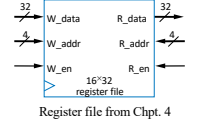
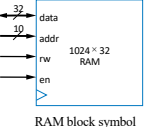
- RTL design instantiates datapath components to create datapath, controlled by a controller
 - Some components are used outside the controller and DP
- **MxN memory**
 - M words, N bits wide each
- Several varieties of memory, which we now introduce



3

Random Access Memory (RAM)

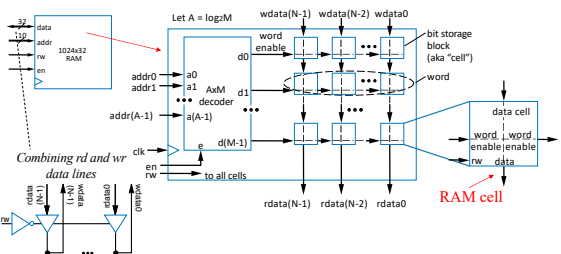
- RAM – Readable and writable memory
 - “Random access memory”
 - Strange name—Created several decades ago to contrast with sequentially-accessed storage like tape drives
 - Logically same as register file—Memory with address inputs, data inputs/outputs, and control
 - RAM usually one port; RF usually 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

4

RAM Internal Structure

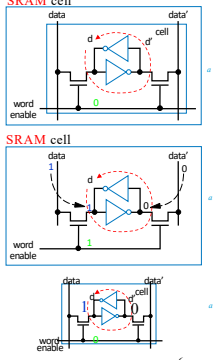
Let $A = \log_2 M$



- 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

5

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

6

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...

7

7

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

8

8

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)

9

9

Naive Register File

10

10

Memory Arrays: Register File

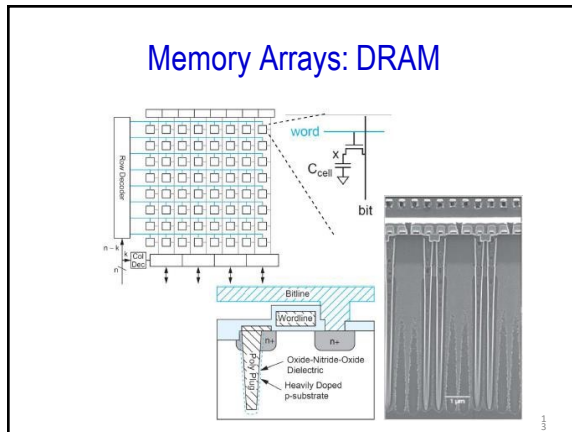
11

11

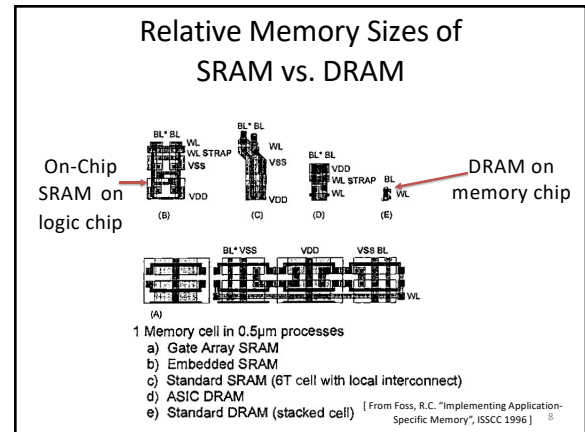
Memory Arrays: SRAM

12

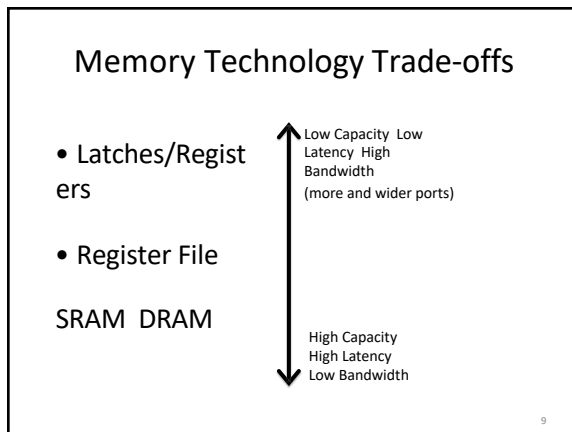
12



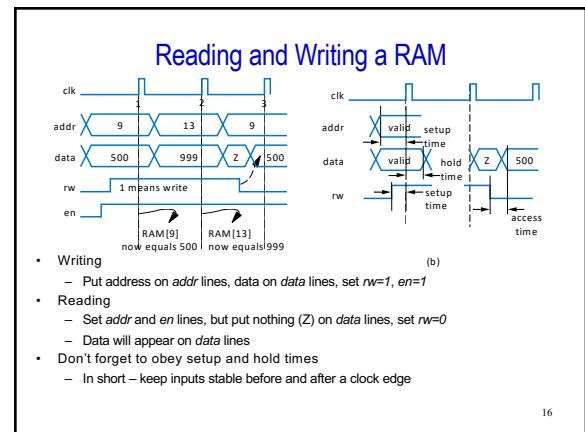
13



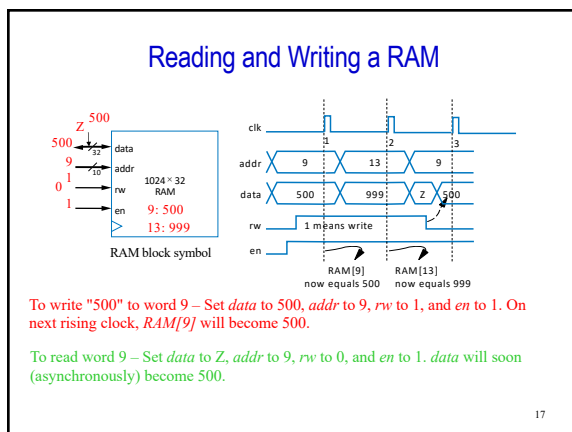
14



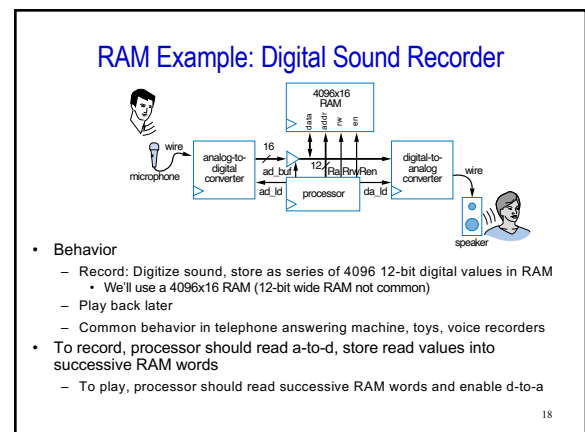
15



16



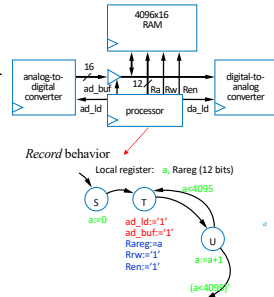
17



18

RAM Example: Digital Sound Recorder

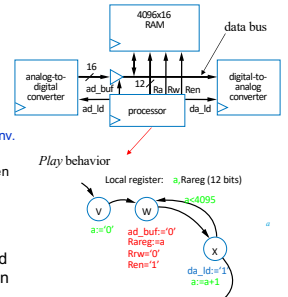
- RTL design of processor
 - Create HLSM
 - Begin with the *record* behavior
 - Create local storage *a*
 - Stores current address, ranges from 0 to 4095 (thus need 12 bits)
 - Create state machine that counts from 0 to 4095 using *a*
 - For each *a*
 - Read analog-to-digital conv.
 - » *ad_id*='1',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



19

RAM Example: Digital Sound Recorder

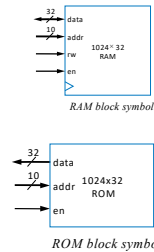
- Now create *play* behavior
- Use local register *a* again, create state machine that counts from 0 to 4095 again
 - For each *a*
 - Read RAM
 - Write to digital-to-analog conv.
 - Note: Must write d-to-a one cycle after reading RAM, when the read data is available on the data bus
- The record and play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



20

Read-Only Memory – ROM

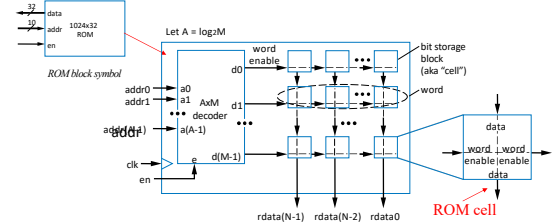
- Memory that can only be read from, not written to
 - Data lines are output only
 - No need for *rw* input
- Advantages over RAM
 - Compact: May be smaller
 - **Nonvolatile**: Saves bits even if power supply is turned off
 - Speed: May be faster (especially than DRAM)
 - Low power: Doesn't need power 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



21

Read-Only Memory – ROM

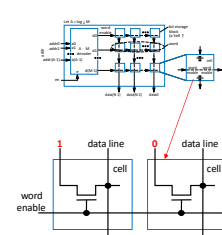
- Internal logical structure similar to RAM, without the data input lines



22

ROM Types

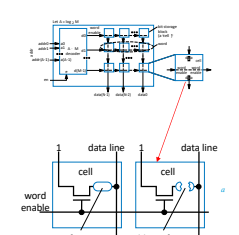
- If a ROM can only be read, how are the stored bits stored in the first place?
 - Storing bits in a ROM known as *programming*
 - Several methods
- **Mask-programmed ROM**
 - Bits are hardwired as 0s or 1s during chip manufacturing
 - 2-bit word on right stores "10"
 - word enable (from decoder) simply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



23

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**

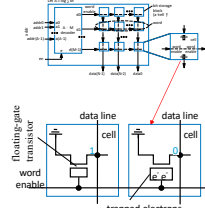


24

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



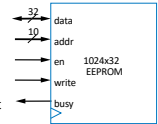
25

25

ROM Types

• Electronically-Erasable Programmable ROM (EEPROM)

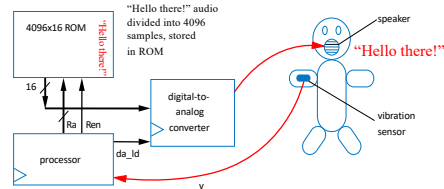
- Similar to EPROM
 - Uses floating-gate transistor, electronic programming to trap electrons in certain cells
- But erasing done *electronically*, not using UV light
- Erasing done one word at a time
- **Flash memory**
 - Like EEPROM, but all words (or large blocks of words) can be erased *simultaneously*
 - 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



26

26

ROM Example: Talking Doll

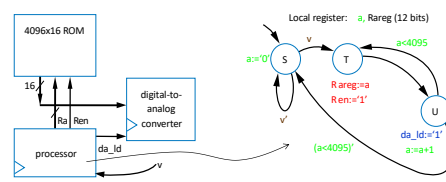


- Doll plays prerecorded message, triggered by vibration
 - Message must be stored without power supply → Use a ROM, not a RAM, because ROM is nonvolatile
 - And because message will never change, may use a mask-programmed ROM or OTP ROM
 - Processor should wait for vibration ($v=1$), then read words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

27

27

ROM Example: Talking Doll



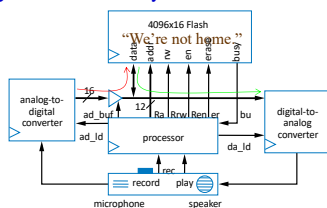
- HLSM
 - Create state machine that **waits** for $v=1$, and then **counts** from 0 to 4095 using a local storage a
 - For each a , **read** ROM, **write** to digital-to-analog converter

28

28

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

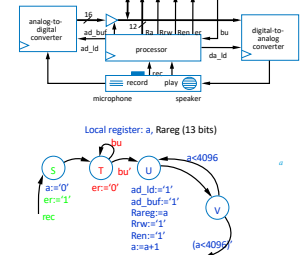


29

29

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

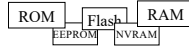


30

30

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

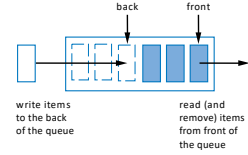


31

31

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)



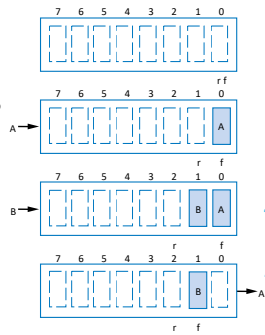
5.8

32

32

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)

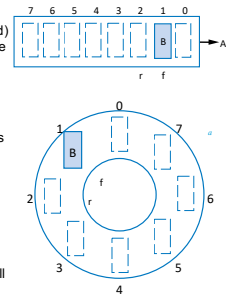


33

33

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)

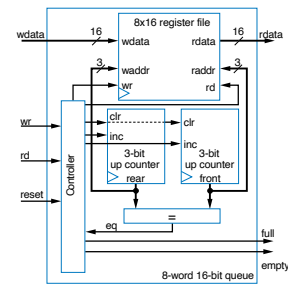


34

34

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



35

35

Common Uses of a Queue

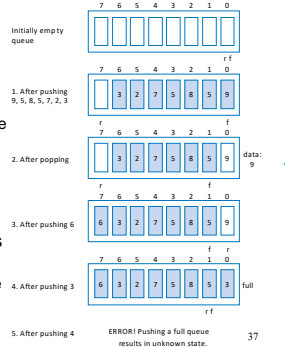
- Computer keyboard
 - Pushes pressed keys onto queue, meanwhile pops and sends to computer
- Digital video recorder
 - Pushes captured frames, meanwhile pops frames, compresses them, and stores them
- Computer network routers
 - Pushes incoming packets onto queue, meanwhile pops packets, processes destination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

36

36

Queue Usage Example

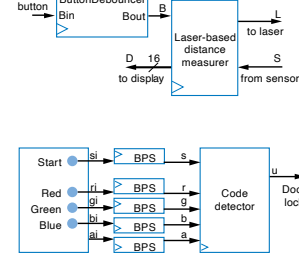
- Example series of pushes and pops
 - Note how rear and front pointers move
 - Note that popping doesn't really remove the data from the queue, but that data is no longer accessible
 - Note how rear (and front) wraps around from address 7 to 0
- Note: pushing a full queue is an error
 - So is popping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



37

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



38

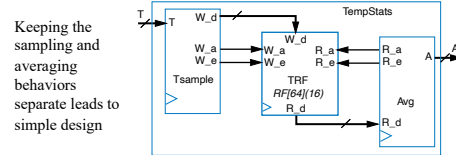
Interfacing Multiple Processors

- 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*

39

Ex: Temperature Statistics with Multiple Processors

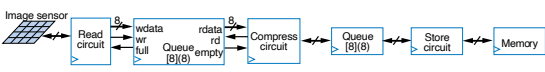
- 16-bit unsigned input T from temperature sensor, 16-bit output A. Sample T every 1 second. Compute output A every minute, should equal average of most recent 64 samples.
- Single HLMS: Complicated
- Instead, two HLMSs (and hence two processors) and shared register file
 - Tsample HLMS: Store T into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



40

Ex: Digital Camera with Mult. Processors and Queue

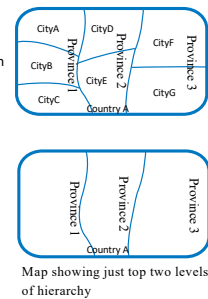
- Read and Compress processors (Ch 1)
 - Compress may take longer, depends on picture
 - Use queue, read can push additional pics (up to 8)
 - Likewise, use queue between Compress and Store



41

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

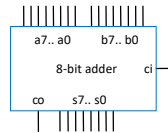


42

Hierarchy and Abstraction

- Abstraction

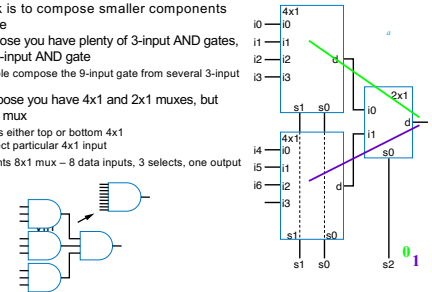
- Hierarchy often involves not just grouping items into a new item, but also associating higher-level behavior with the new item, known as abstraction.
 - Ex: 8-bit adder has understandable high-level behavior—adds two 8-bit binary numbers
- Frees designer from having to remember, or even understand,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



43

Hierarchy and Composing Larger Components from Smaller Versions

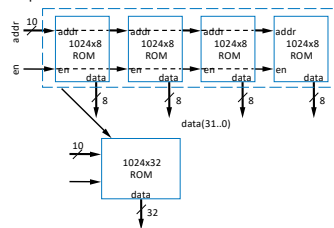
- A common task is to compose smaller components into a larger one
 - Gates: Suppose you have plenty of 3-input AND gates, but need a 9-input AND gate
 - Can simply compose the 9-input gate from several 3-input gates
 - Muxes: Suppose you have 4x1 and 2x1 muxes, but need an 8x1 mux
 - s2 selects either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



44

Hierarchy and Composing Larger Components from Smaller Versions

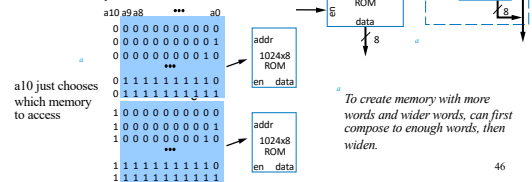
- Composing memory very common
- Making memory words wider
 - Easy – just place memories side-by-side until desired width obtained
 - Share address/control lines, concatenate data lines
 - Example: Compose 1024x8 ROMs into 1024x32 ROM



45

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



46

Summary

- Memory components
 - RAM: readable and writable memory
 - SRAM vs DRAM
 - ROM: read-only memory
 - Erasable Programmable ROM vs Flash Memory
- Queues
 - FIFO: A list written to at the back, from read from the front
 - Common Uses of a Queue
- Multiple processors
 - Use multiple processor to design complicated systems
 - Interfacing between processors
- Hierarchy
 - Design philosophy: hierarchy helps us manage complexity
 - Memory hierarchy (optional)

47

47