RUTGERS

UNIVERSITY | NEW BRUNSWICK

ECE – 332:493 / 579

Hardware/Software Design of Embedded Systems I / III

Spring 2025 – Lecture 11

Prof. Milton Diaz

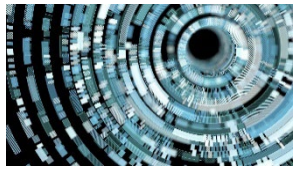
April 17, 2025

Today's Discussion

- Announcements
- Class Schedule
- Vahid – Chapter 5
- Week 13 Module on Canvas
- Lab 4 comments
- Lab 5
- Final Project

Embedded Systems I / III – Special Topics - Elective

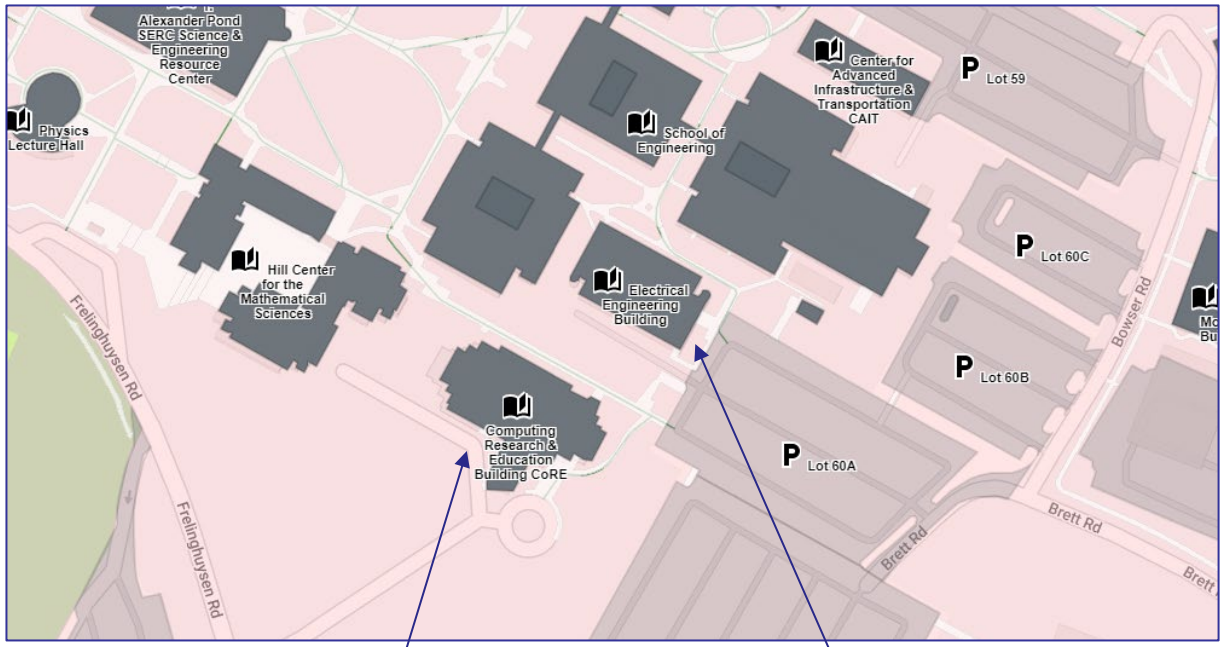
- Class Schedule
  - Thursdays
    - Lectures – CoRE – 538
      - Thursdays 5:40 – 7:00 PM
      - Instructor – Milton Diaz
    - Labs – ECE 103
      - Thursdays 7:30 – 8:50 PM
      - Instructor – Milton Diaz
      - TA/Graders:
        - Atharva Pandhare
        - Keyur Rana



- Virtual Classes only when/if announced
  - Join Webex at lecture designated time

https://rutgers.webex.com/meet/md1632

  - Lab session – Webex room by TAs



Lectures
96 Frelinghuysen Road,
Piscataway, NJ 08854Building
Number: 3883

Labs
94 Brett Road,
Piscataway, NJ 08854-8058
Building Number: 3859

Class Schedule (2025)

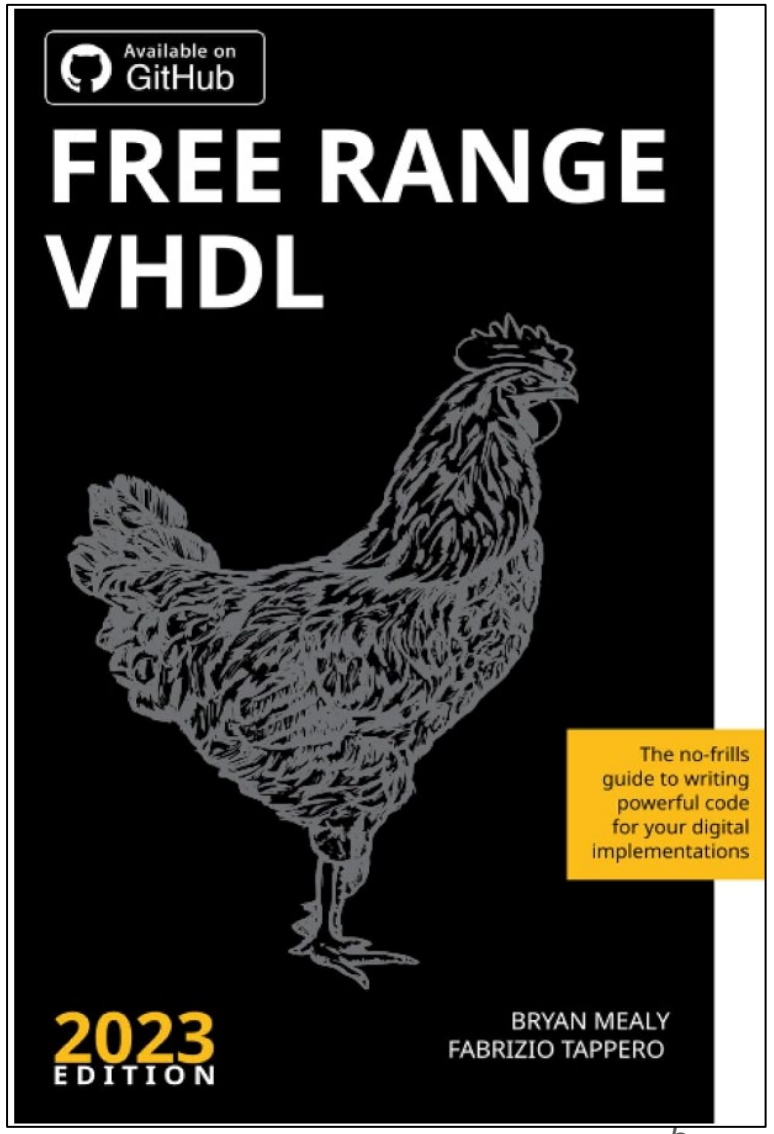
Week	Date	Topics	Quiz	HW	Lab	Readings	Textbook/Ref.
1	Jan. 23rd	Welcome / Introduction			Lab Intro		Diaz
2	Jan. 30th	FPGA Overview / ESD	1	1	Lab 0	Ch. 1 - 5	Taperro (Fried Chicken)
3	Feb. 6th	VHDL – Intro, Invariants, Design Units	2	2	Synth. Labs	Ch. 1 – 5	Taperro (F.C) , Xilinx
4	Feb. 13th	VHDL – Sequential Circuits & RTL	3	3	Lab 1	Ch. 6 – 7	Taperro (Fried Chicken)
5	Feb. 20th	FPGA 7 Series Architecture	4	4	Lab 1	Ch. 6 – 7	Tappero (Fried Chicken)
6	Feb. 27th	VHDL - Models, Operators	5	5	Lab 2	Ch. 8 – 9, 13	Tappero (Fried Chicken)
7	Mar. 6th	VHDL – Finite State Machines *(Start Project Proposal)*	6	6	Lab 3	Ch. 10 - 12	Tappero (Fried Chicken)
8	Mar. 13th	IC Technology	7	7	Lab 3	Ch.1	Vahid
9 ***	Mar. 23th	*** Spring Break ***	***	***	***	***	***
10	Mar. 27th	Single Purpose Processor (Project Proposal Due)	8	8	Lab 4	Ch. 2	Vahid
11	Apr. 3rd	General Purpose Processors	9	9	Lab 4	Ch. 3	Vahid
12	Apr. 10th	Std. Single-Purpose Proc. (Peripherals)	10	10	Lab 5	Ch. 4	Vahid * pre-recorded
13	Apr. 17th	Memory	11	11	Lab 5	Ch. 5	Vahid
14	Apr. 24th	Interfacing I	x	x	Final Project	Ch. 6	Vahid
15	May. 1st	Last lecture – Interfacing II	x	x	Final Project	Ch. 6	Vahid
Finals	Wed. May 14th	Final Project Presentations			ECE 103 Lab		4:00pm – 7:00pm

Reading Assignment - Textbook

- Fabrizio Tappero, Bryan Mealy “Free Range VHDL. The no-frills guide to writing powerful code for your digital implementations” Paperback – 2023

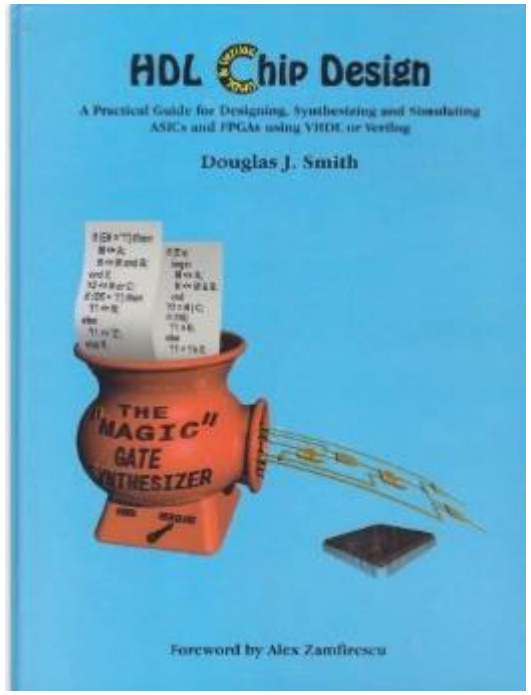
- A re-organized free PDF following sequence of topics is available on CANVAS -- aka “Fried Chicken”

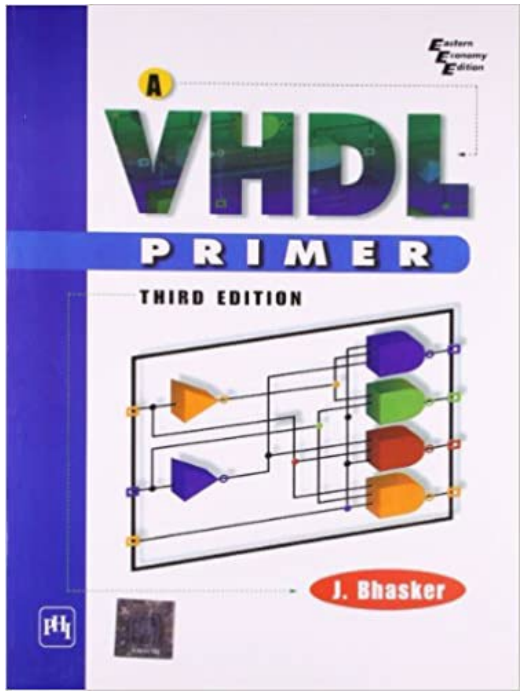
FREE DIGITAL COPY AVAILABLE ON CANVAS COURSE PAGE

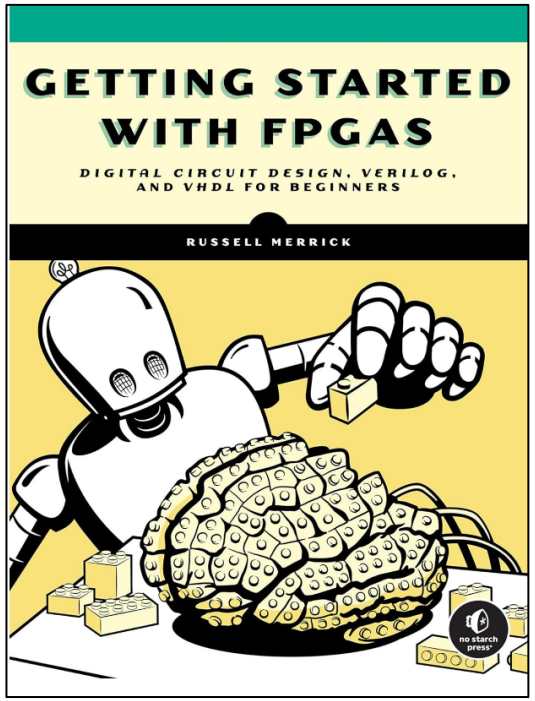


Secondary / Reference Books

- Hdl Chip Design: A Practical Guide for Designing, Synthesizing & Simulating ASICs & FPGAs using VHDL or Verilog Hardcover – March, 1998 -- SOME COPIES AVAILABLE IN THE LAB CABINET
- A VHDL Primer – Third Edition – 1999
- Getting Started with FPGAs – November, 2023 – By author of NANDLAND (VHDL & Embedded topics)

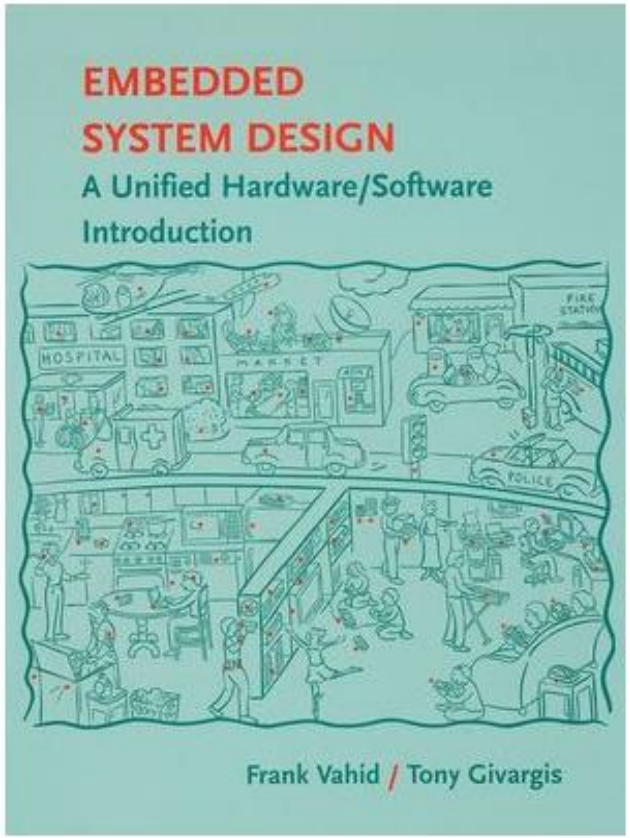






Textbook

- Frank Vahid & Tony Givargis, “Embedded System Design, A Unified Hardware/Software Introduction”
- Modified Slides from this text book will be used
- Where to get it?
  - Some options:
    - Amazon - Embedded System Design
    - Wiley - Print on demand



ISBN-13: 978-0471-38678-0
ISBN-10: 0-471-38678-2

Vahid – Embedded System Design – A Unified Hardware/Software Introduction

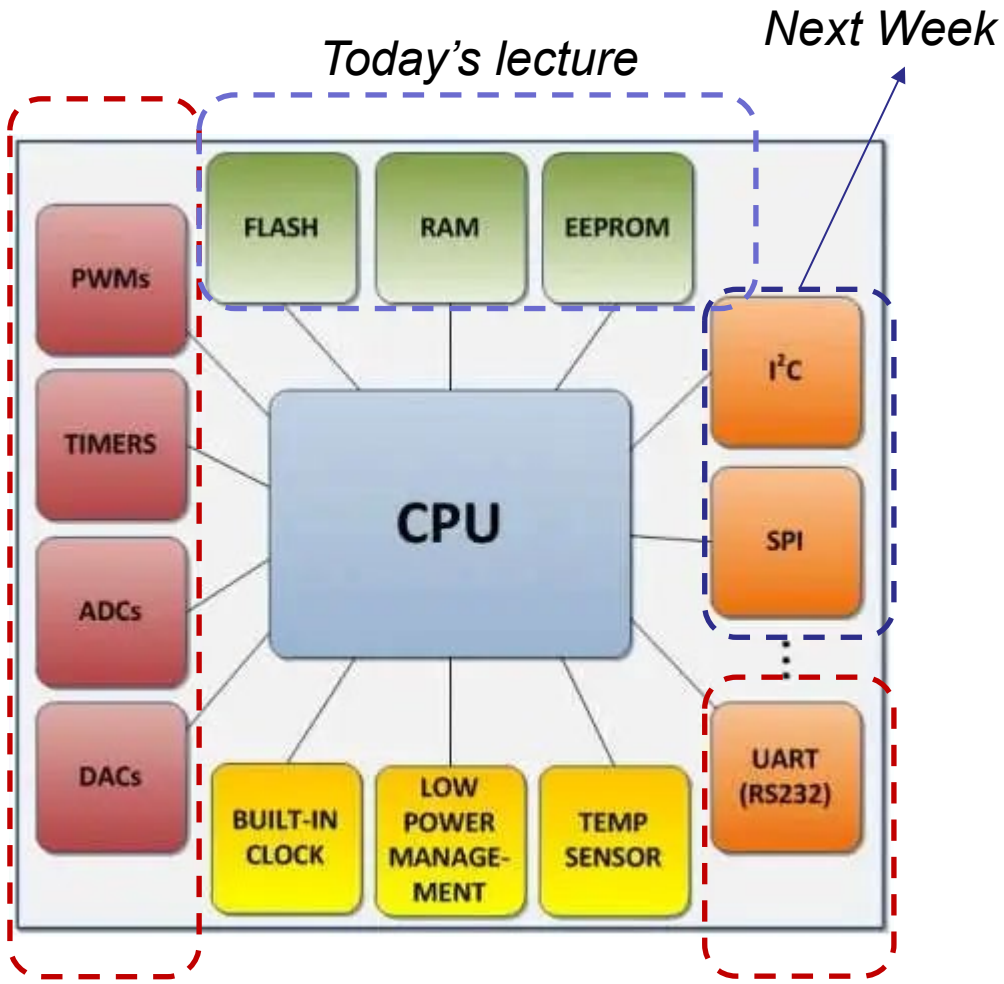
CHAPTER 5 – MEMORY

Outline

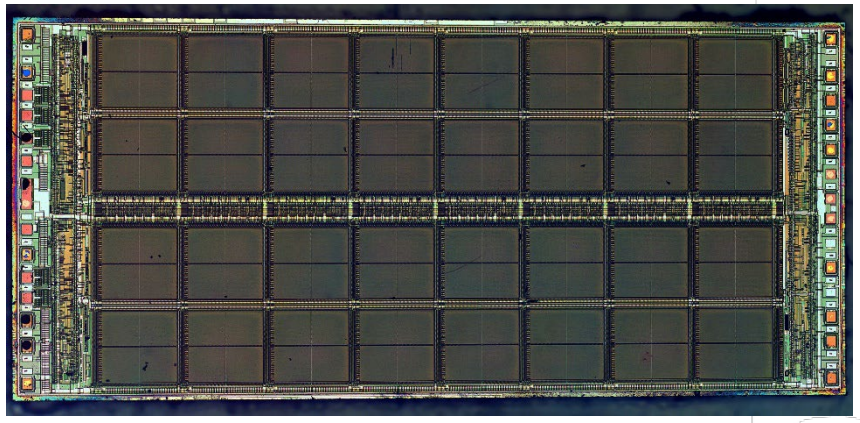
- Memory Write Ability and Storage Permanence
- Common Memory Types
- Composing Memory
- Memory Hierarchy and Cache
- Advanced RAM

Introduction

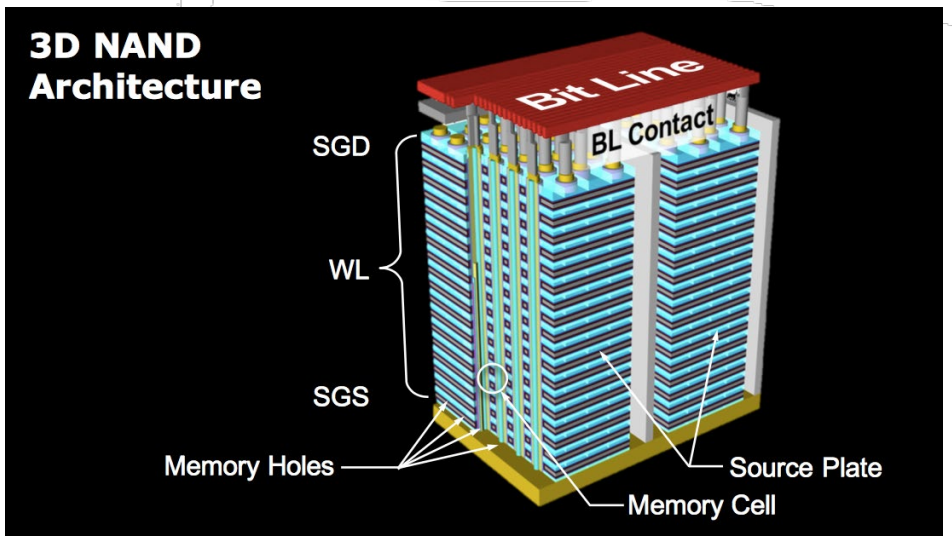
- Embedded system's functionality aspects
  - Processing
    - processors
    - transformation of data
  - Storage
    - memory
    - retention of data
  - Communication
    - buses
    - transfer of data



Introduction



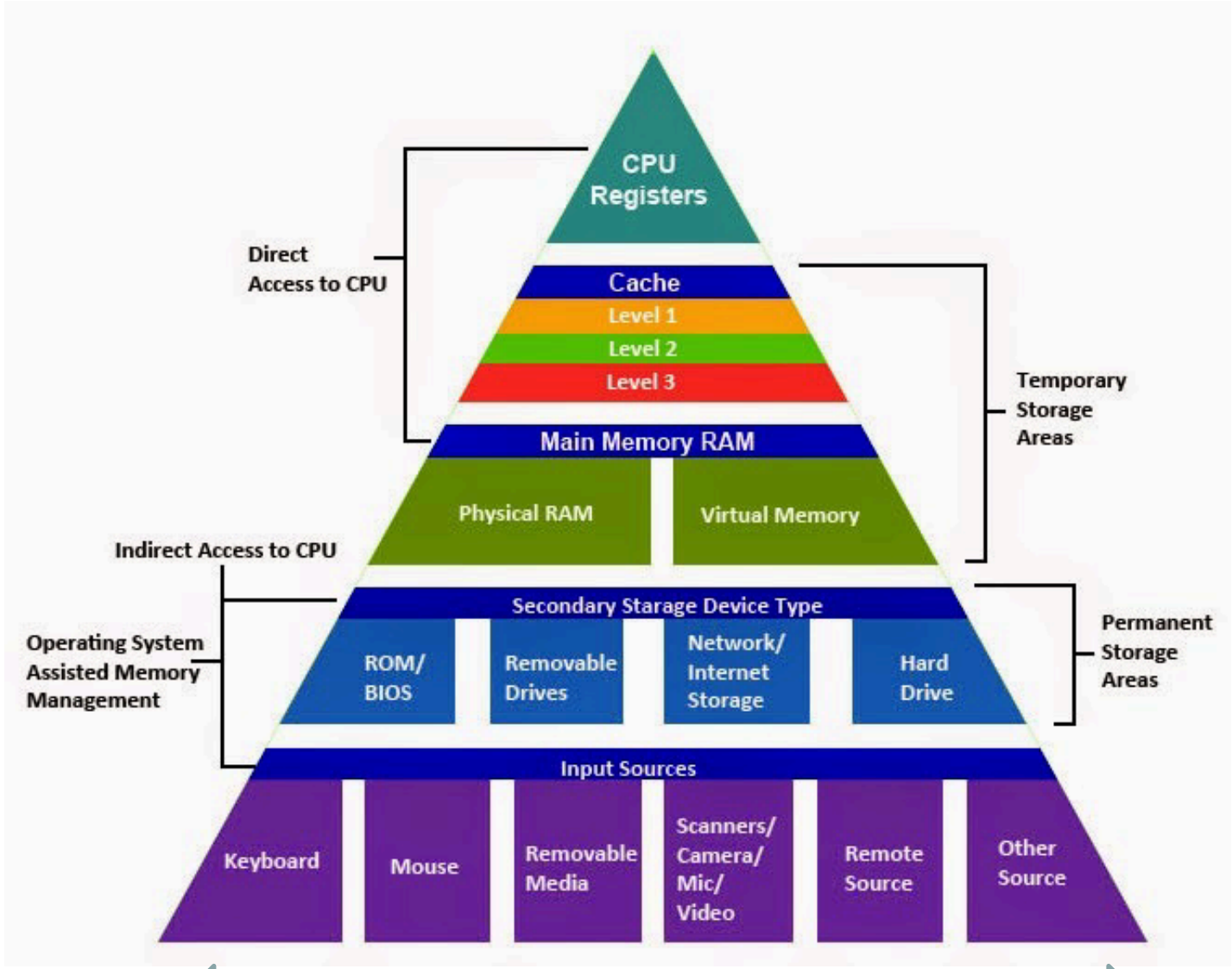
1 Mbit DRAM Micron memory



Memory Taxonomy from 2013 ITRS Emerging Research Devices

Introduction

Latency,
Persistence

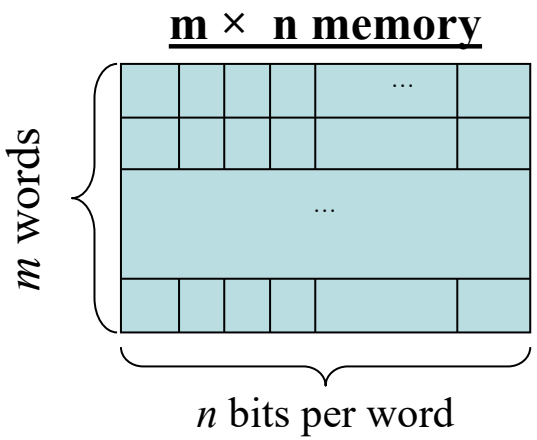


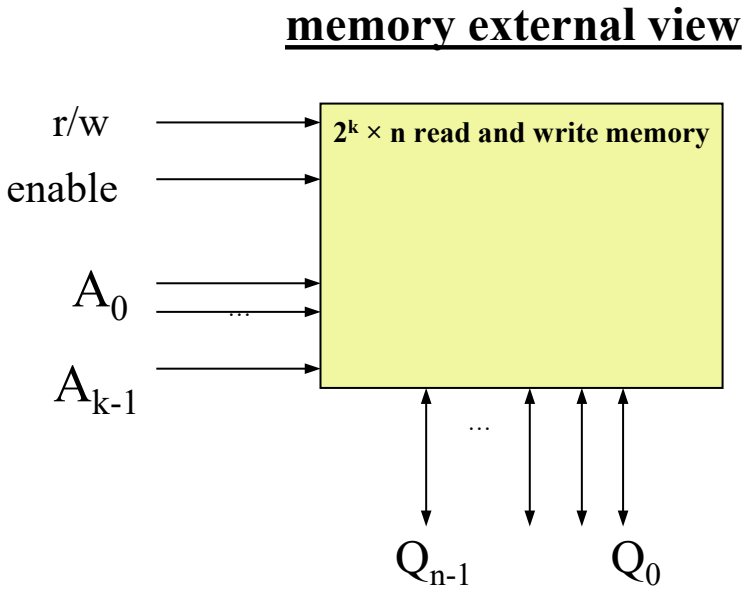
Performance
Endurance
Cost per bit

Volume

Memory: basic concepts

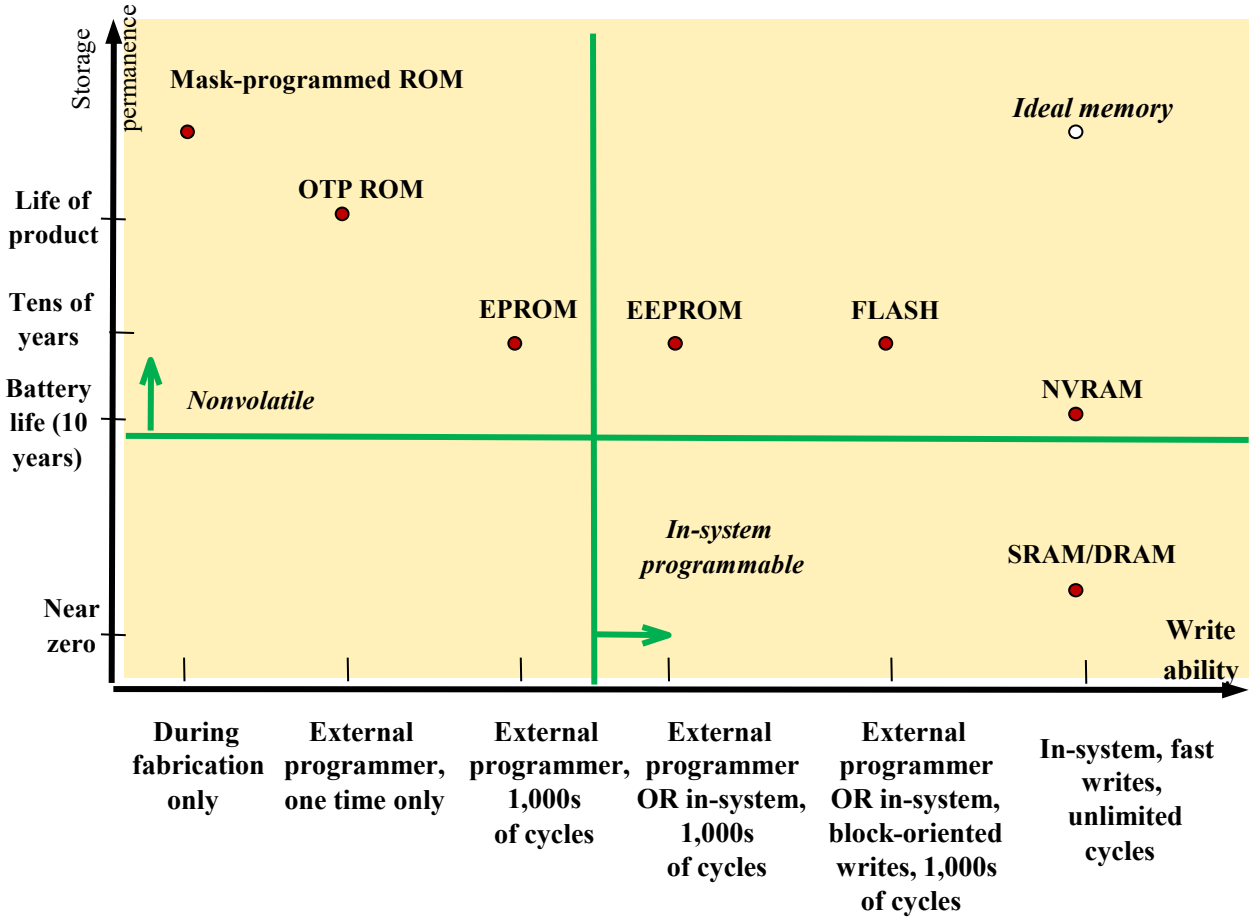
- Stores large number of bits
  - *m* x *n*: *m* words of *n* bits each
  - k = $\text{Log}_2(m)$ address input signals
  - or *m* = 2^k words
  - e.g., 4,096 x 8 memory:
    - 32,768 bits
    - 12 address input signals
    - 8 input/output data signals
- Memory access
  - r/w: selects read or write
  - enable: read or write only when asserted
  - multiport: multiple accesses to different locations simultaneously





Write ability/ storage permanence

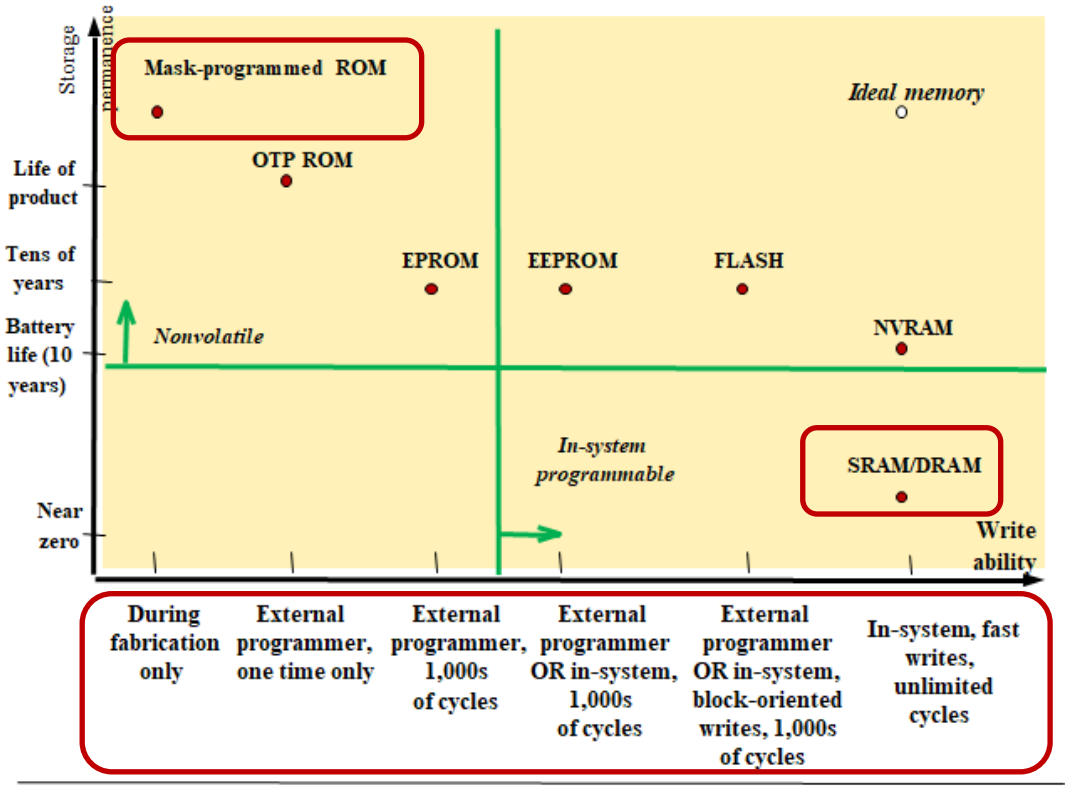
- Traditional ROM/RAM distinctions
  - ROM
    - read only, bits stored without power
  - RAM
    - read and write, lose stored bits without power
- Traditional distinctions blurred
  - Advanced ROMs can be written to
    - e.g., EEPROM
  - Advanced RAMs can hold bits without power
    - e.g., NVRAM
- Write ability
  - Manner and speed a memory can be written
- Storage permanence
  - ability of memory to hold stored bits after they are written



Write ability and storage permanence of memories, showing relative degrees along each axis (not to scale).

Write ability

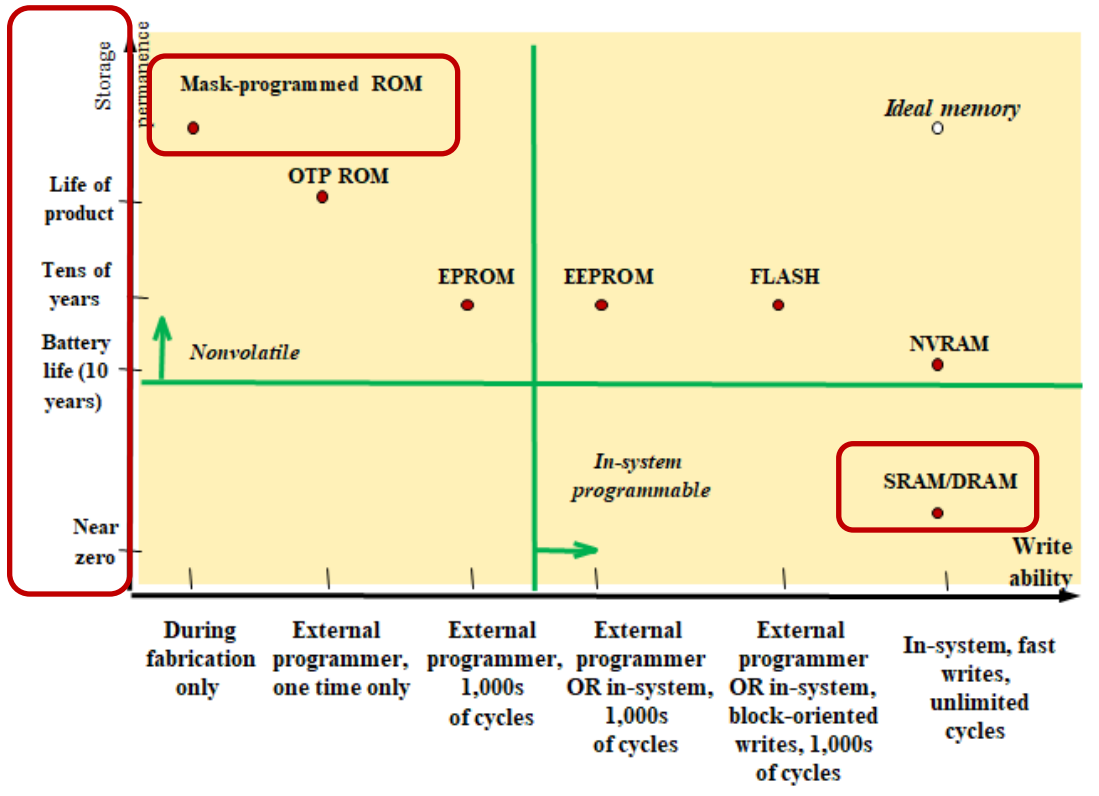
- Ranges of write ability
  - High end
    - processor writes to memory simply and quickly
    - e.g., RAM
  - Middle range
    - processor writes to memory, but slower
    - e.g., FLASH, EEPROM
  - Lower range
    - special equipment, “programmer”, must be used to write to memory
    - e.g., EPROM, OTP ROM
  - Low end
    - bits stored only during fabrication
    - e.g., Mask-programmed ROM
- In-system programmable memory
  - Can be written to by a processor in the embedded system using the memory
  - Memories in high end and middle range of write ability



Write ability and storage permanence of memories, showing relative degrees along each axis (not to scale).

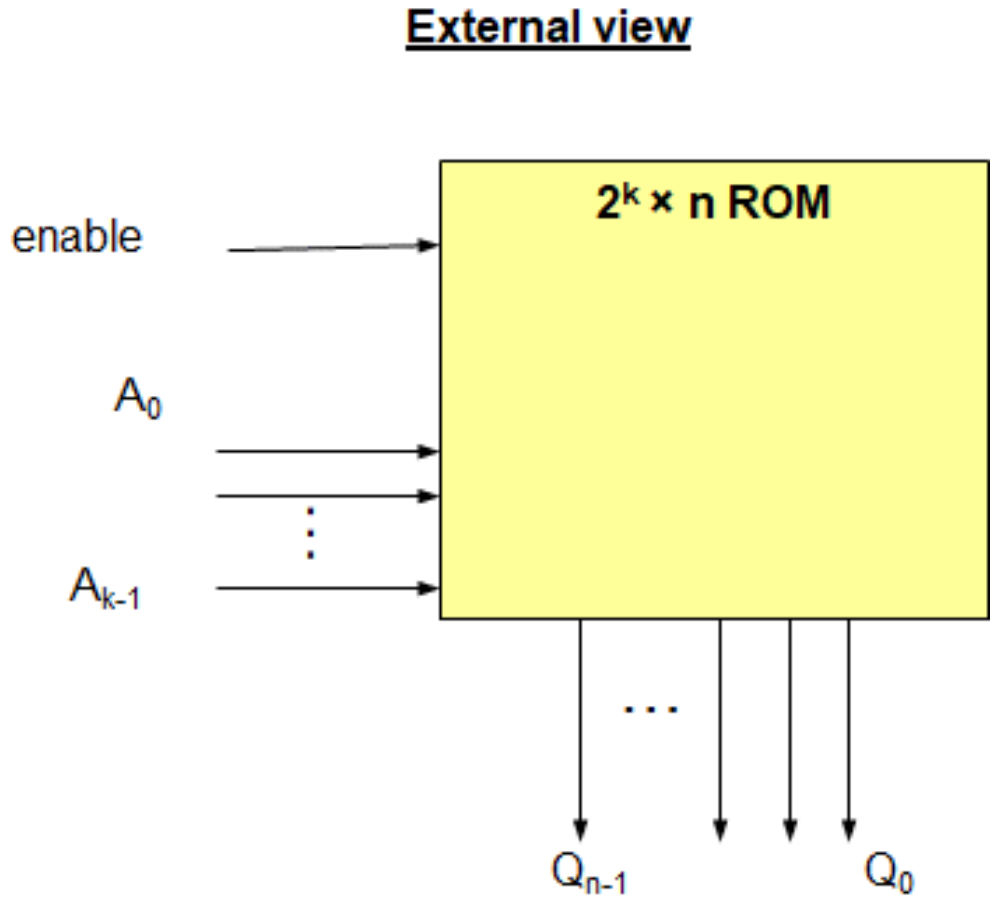
Storage permanence

- Range of storage permanence
  - High end
    - essentially never loses bits
    - e.g., mask-programmed ROM
  - Middle range
    - holds bits days, months, or years after memory's power source turned off
    - e.g., NVRAM
  - Lower range
    - holds bits as long as power supplied to memory
    - e.g., SRAM
  - Low end
    - begins to lose bits almost immediately after written
    - e.g., DRAM
- Nonvolatile memory
  - Holds bits after power is no longer supplied
  - High end and middle range of storage permanence



Write ability and storage permanence of memories, showing relative degrees along each axis (not to scale).

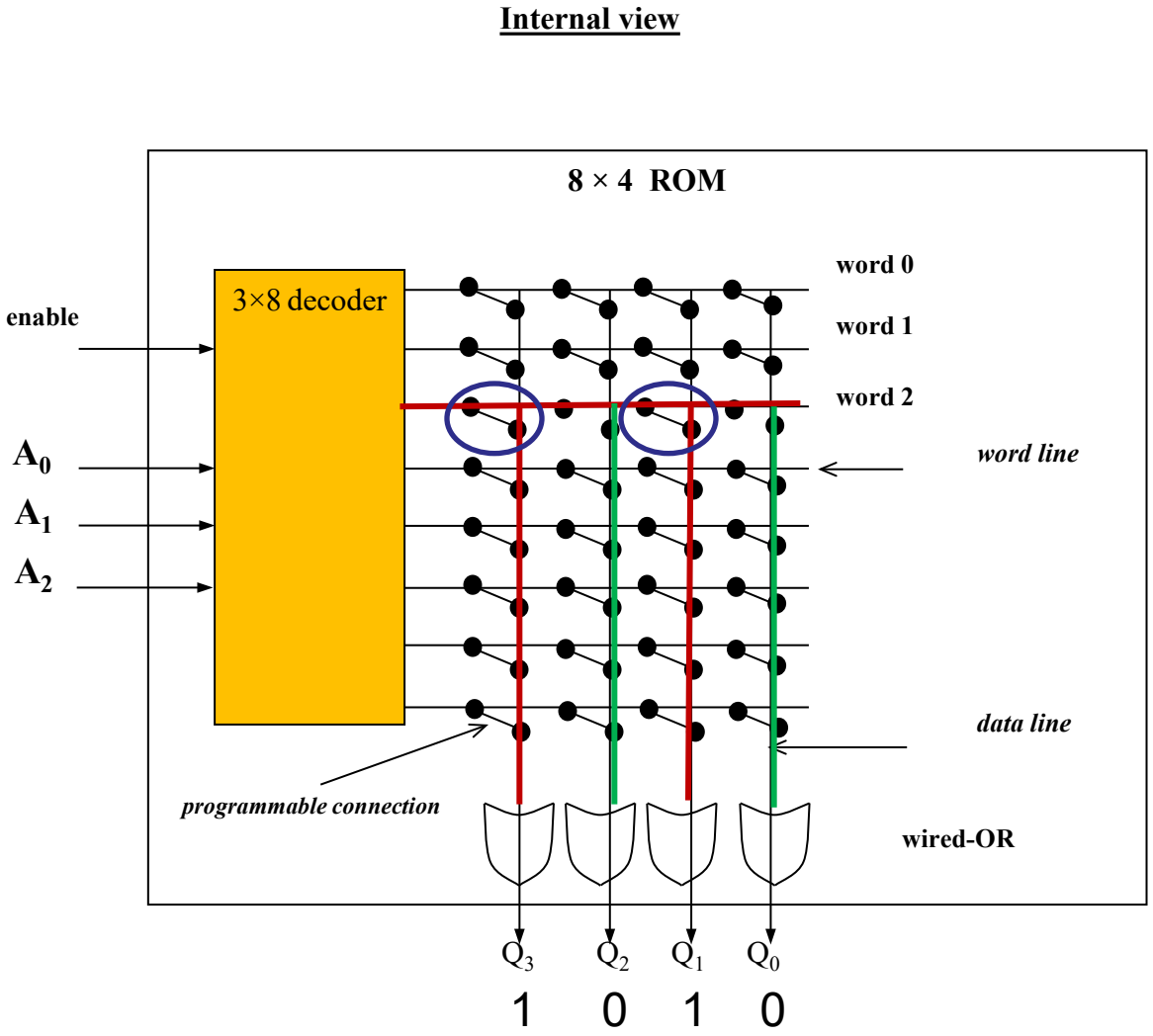
ROM: "Read-Only" Memory



- Nonvolatile memory
- Can be read from but not written to, by a processor in an embedded system
- Traditionally written to, "programmed", before inserting to embedded system
- Uses
  - Store software program for general-purpose processor
    - program instructions can be one or more ROM words
  - Store constant data needed by system
  - Implement combinational circuit

Example: 8 x 4 ROM

- Horizontal lines = words
- Vertical lines = data
- Lines connected only at circles
- Decoder sets word 2's line to 1 if address input is 010
- Data lines Q_3 and Q_1 are set to 1 because there is a "programmed" connection with word 2's line
- Word 2 is not connected with data lines Q_2 and Q_0
- Output is 1010

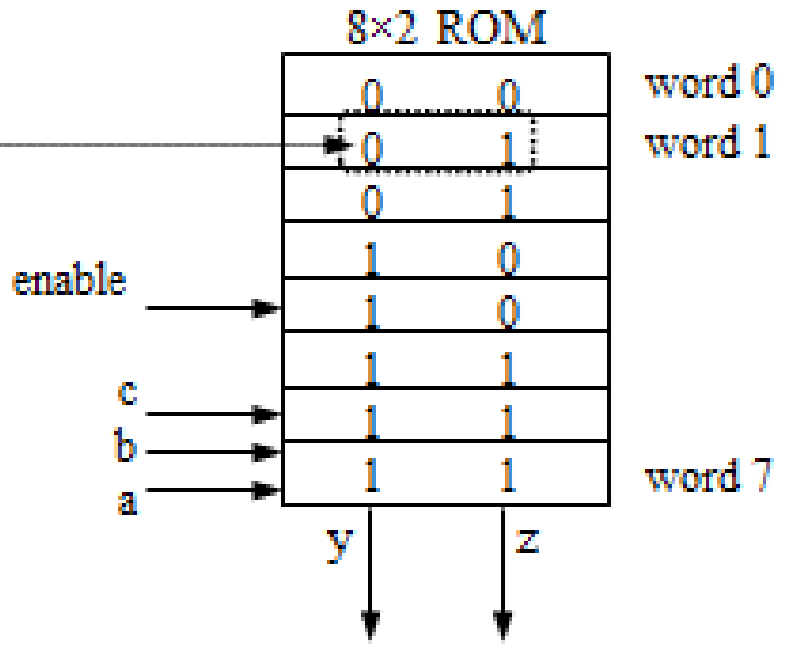


Implementing combinational function

- Any combinational circuit of *n* functions of same *k* variables can be done with 2^*k* x *n* ROM

Truth table

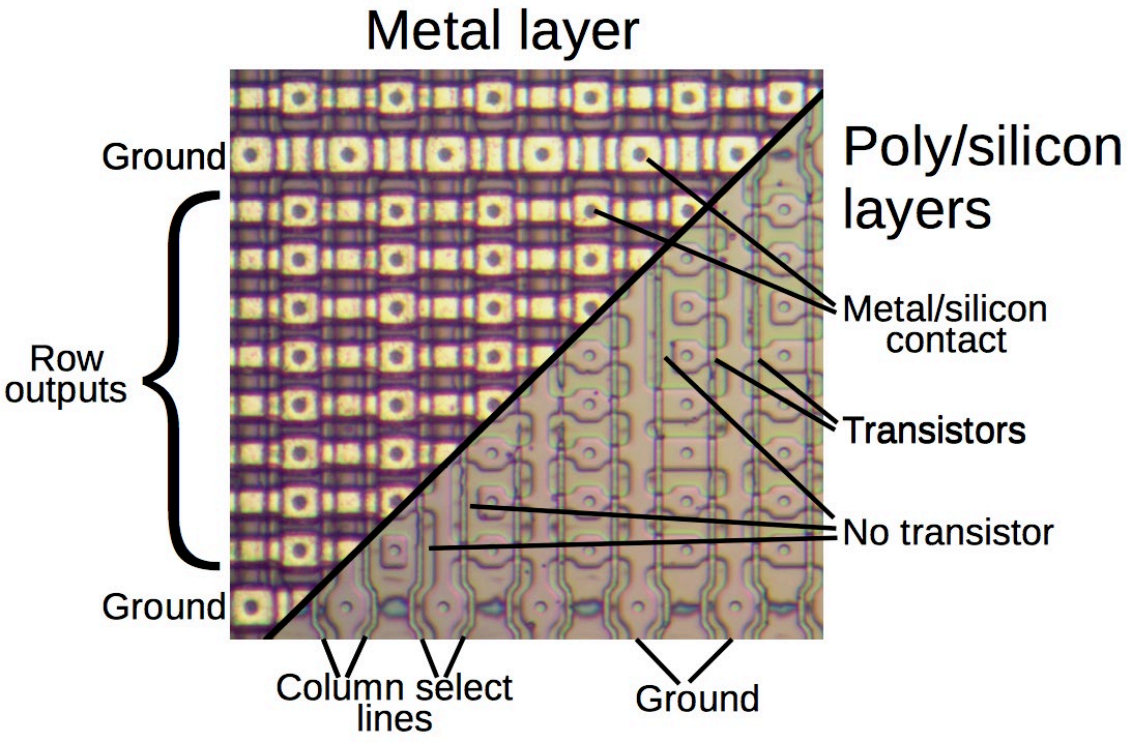
Inputs (address)			Outputs	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1



What does this remind you of?

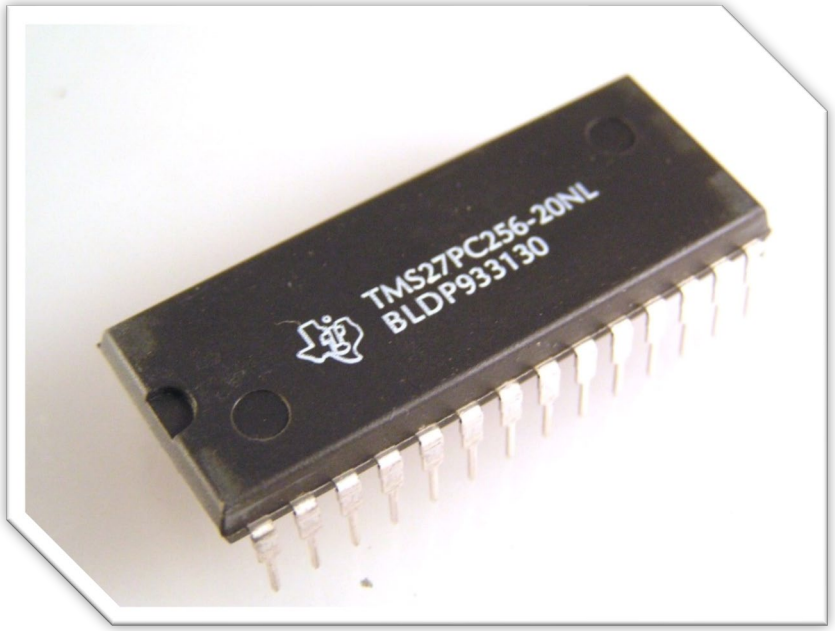
Mask-programmed ROM

Intel 8087 – High density ROM



- Connections “programmed” at fabrication
  - set of masks
- Lowest write ability
  - only once
- Highest storage permanence
  - bits never change unless damaged
- Typically used for final design of high-volume systems
  - spread out NRE cost for a low unit cost

OTP ROM: One-time programmable ROM

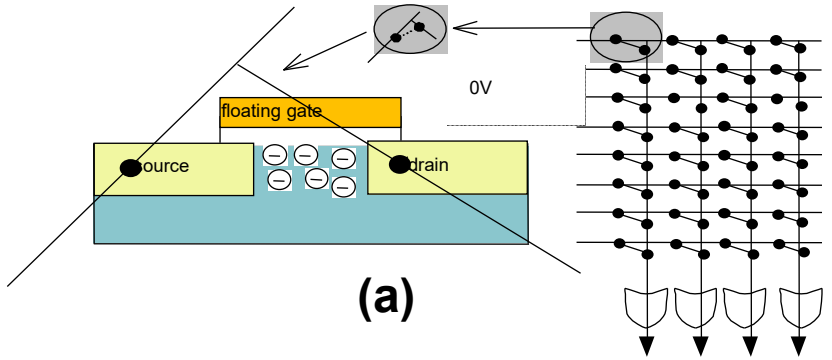


TI OTP ROM 262 kb

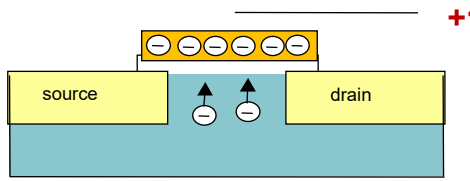
- Connections "programmed" after manufacture by user
  - user provides file of desired contents of ROM
  - file input to machine called ROM programmer
  - each programmable connection is a fuse
  - ROM programmer blows fuses where connections should not exist
- Very low write ability
  - typically written only once and requires ROM programmer device
- Very high storage permanence
  - bits don't change unless reconnected to programmer and more fuses blown
- Commonly used in final products
  - cheaper, harder to inadvertently modify

EPROM: Erasable programmable ROM

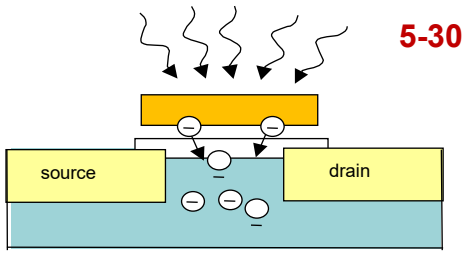
- **Programmable component is a MOS transistor**
  - Transistor has “floating” gate surrounded by an insulator
  - **(a)** Negative charges form a channel between source and drain storing a logic 1
  - **(b)** Large positive voltage at gate causes negative charges to move out of channel and get trapped in floating gate storing a logic 0
  - **(c)** (Erase) Shining UV rays on surface of floating-gate causes negative charges to return to channel from floating gate restoring the logic 1
  - **(d)** An EPROM package showing quartz window through which UV light can pass
- **Better write ability**
  - can be erased and reprogrammed thousands of times
- **Reduced storage permanence**
  - program lasts about 10 years but is susceptible to radiation and electric noise
- **Typically used during design development**



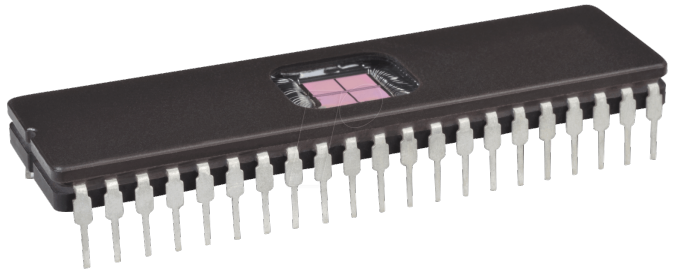
(a)



(b)



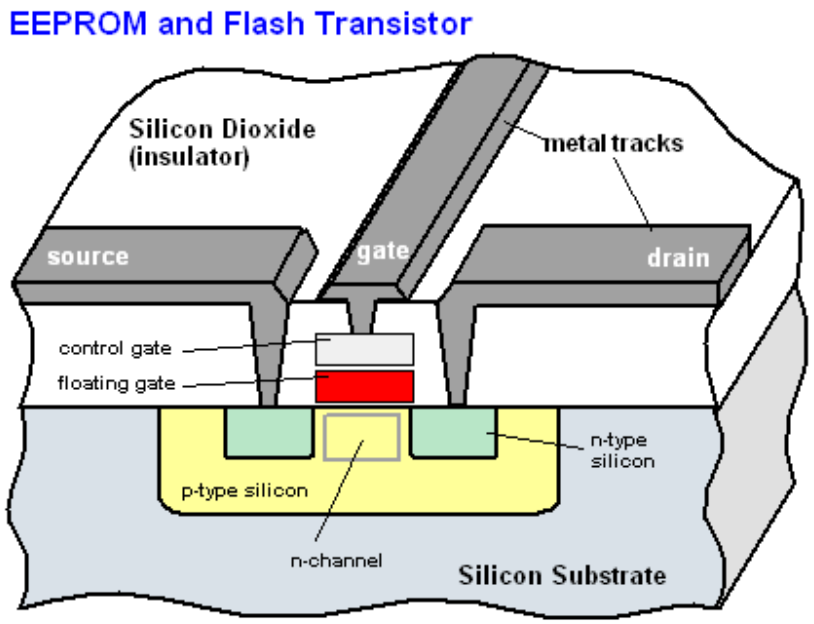
(c)



(d)

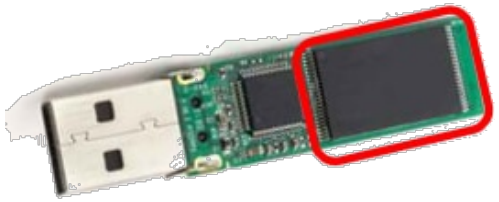
EEPROM: Electrically erasable programmable ROM

- **Programmed and erased electronically**
  - typically by using higher than normal voltage
  - can program and erase individual words
- **Better write ability**
  - can be in-system programmable with built-in circuit to provide higher than normal voltage
    - built-in memory controller commonly used to hide details from memory user
  - writes very slow due to erasing and programming
    - "busy" pin indicates to processor EEPROM still writing
  - can be erased and programmed tens of thousands of times
- **Similar storage permanence to EPROM (about 10 years)**
- **Far more convenient than EPROMs, but more expensive**



Flash Memory

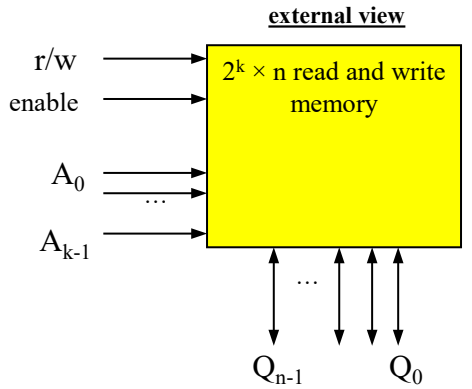
- Extension of EEPROM
  - Same floating gate principle
  - Same write ability and storage permanence
- Fast erase
  - Large blocks of memory erased at once, rather than one word at a time
  - Blocks typically several thousand bytes large

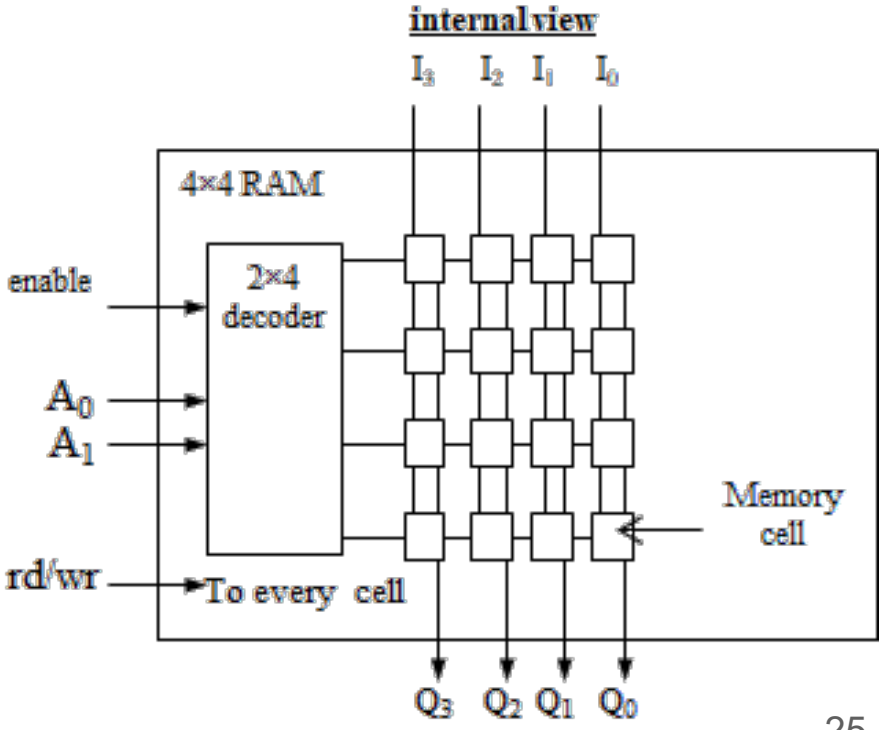


- Writes to single words may be slower
  - Entire block must be read, word updated, then entire block written back
- Used with embedded systems storing large data items in nonvolatile memory
  - e.g., digital cameras, TV set-top boxes, cell phones

RAM: "Random-access" memory

- **Typically volatile memory**
  - bits are not held without power supply
- **Read and written to easily by embedded system during execution**
- **Internal structure more complex than ROM**
  - a word consists of several memory cells, each storing 1 bit
  - each input and output data line connects to each cell in its column
  - rd/wr connected to every cell
  - when row is enabled by decoder, each cell has logic that stores input data bit when rd/wr indicates write or outputs stored bit when rd/wr indicates read



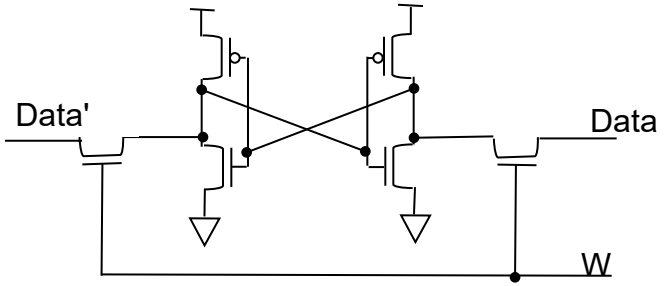


Basic types of RAM

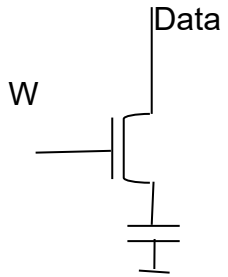
- **SRAM: Static RAM**
  - Memory cell uses flip-flop to store bit
  - Requires 6 transistors
  - Holds data as long as power supplied
- **DRAM: Dynamic RAM**
  - Memory cell uses MOS transistor and capacitor to store bit
  - More compact than SRAM
  - "Refresh" required due to capacitor leak
    - word's cells refreshed when read
  - Typical refresh rate 15.625 microsec.
  - Slower to access than SRAM

memory cell internals

SRAM



DRAM

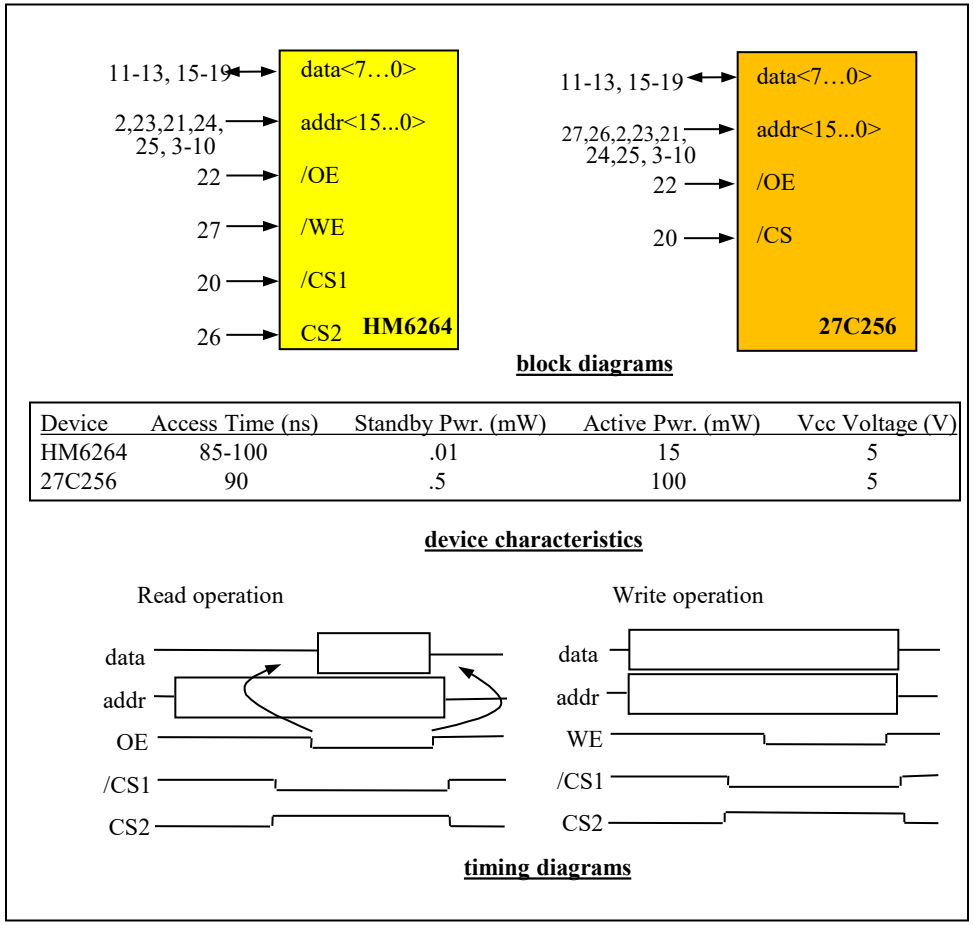


RAM variations

- **PSRAM: Pseudo-static RAM**
  - DRAM with built-in memory refresh controller
  - Popular low-cost high-density alternative to SRAM
- **NVRAM: Nonvolatile RAM**
  - Holds data after external power removed
  - Battery-backed RAM
    - SRAM with own permanently connected battery
    - writes as fast as reads
    - no limit on number of writes unlike nonvolatile ROM-based memory
  - SRAM with EEPROM or flash
    - stores complete RAM contents on EEPROM or flash before power turned off

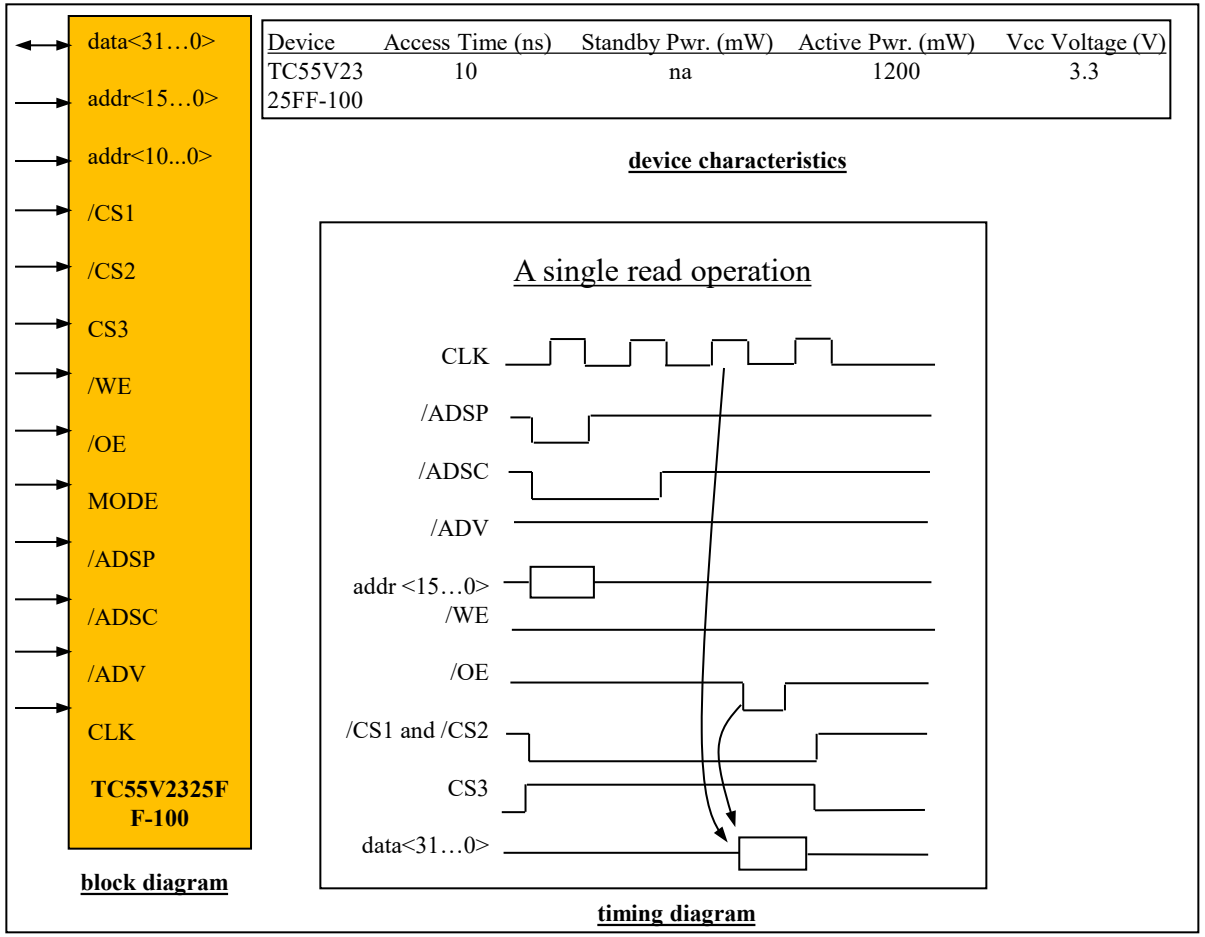
Example: HM6264 & 27C256 RAM/ROM devices

- Low-cost low-capacity memory devices
- Commonly used in 8-bit microcontroller-based embedded systems
- First two numeric digits indicate device type
  - RAM: 62
  - ROM: 27
- Subsequent digits indicate capacity in kilobits



Example: TC55V2325FF-100 memory device

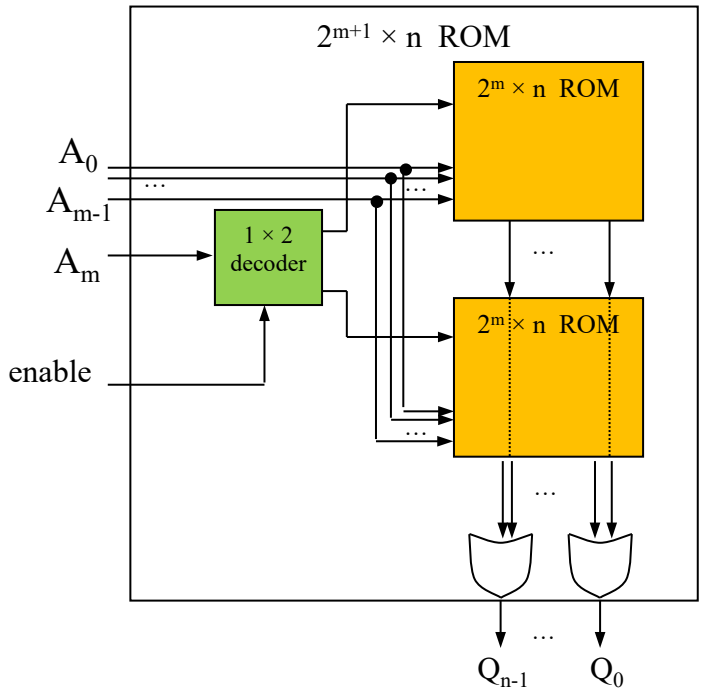
- 2-megabit synchronous pipelined burst SRAM memory device
- Designed to be interfaced with 32-bit processors
- Capable of fast sequential reads and writes as well as single byte I/O



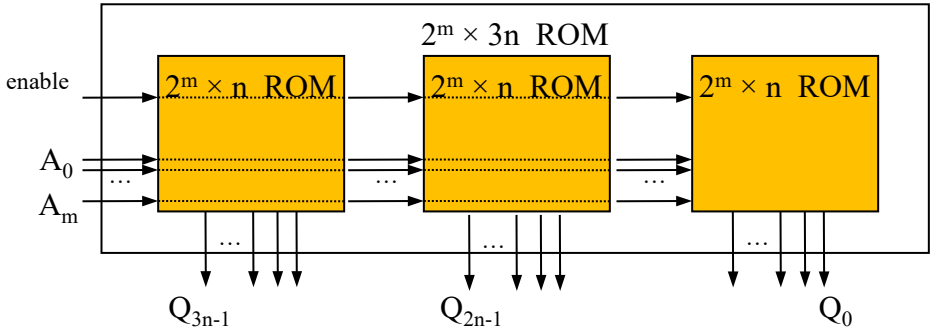
Composing memory

- Memory size needed often differs from size of readily available memories
- When available memory is larger, simply ignore unneeded high-order address bits and higher data lines
- When available memory is smaller, compose several smaller memories into one larger memory
  - Connect side-by-side to increase width of words
  - Connect top to bottom to increase number of words
    - added high-order address line selects smaller memory containing desired word using a decoder
  - Combine techniques to increase number and width of words

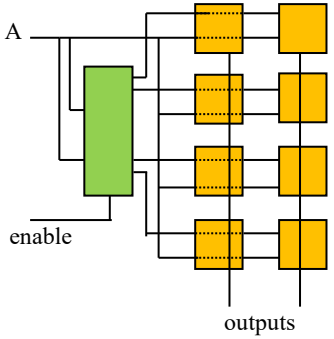
Increase number of words



Increase width of words

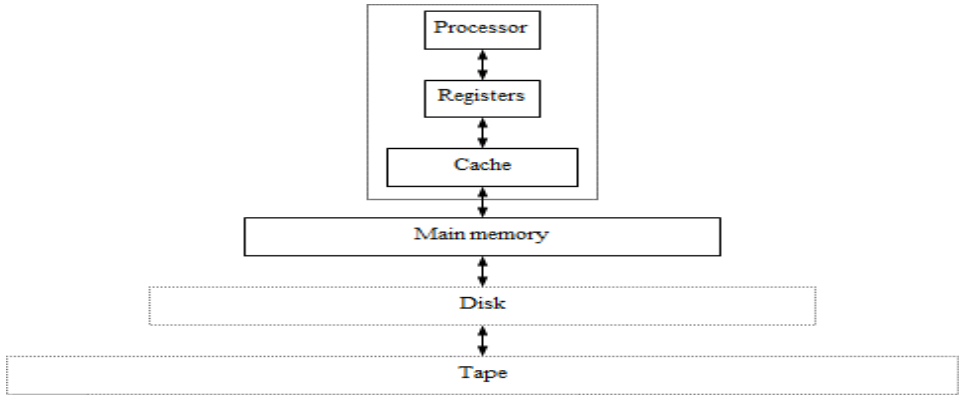


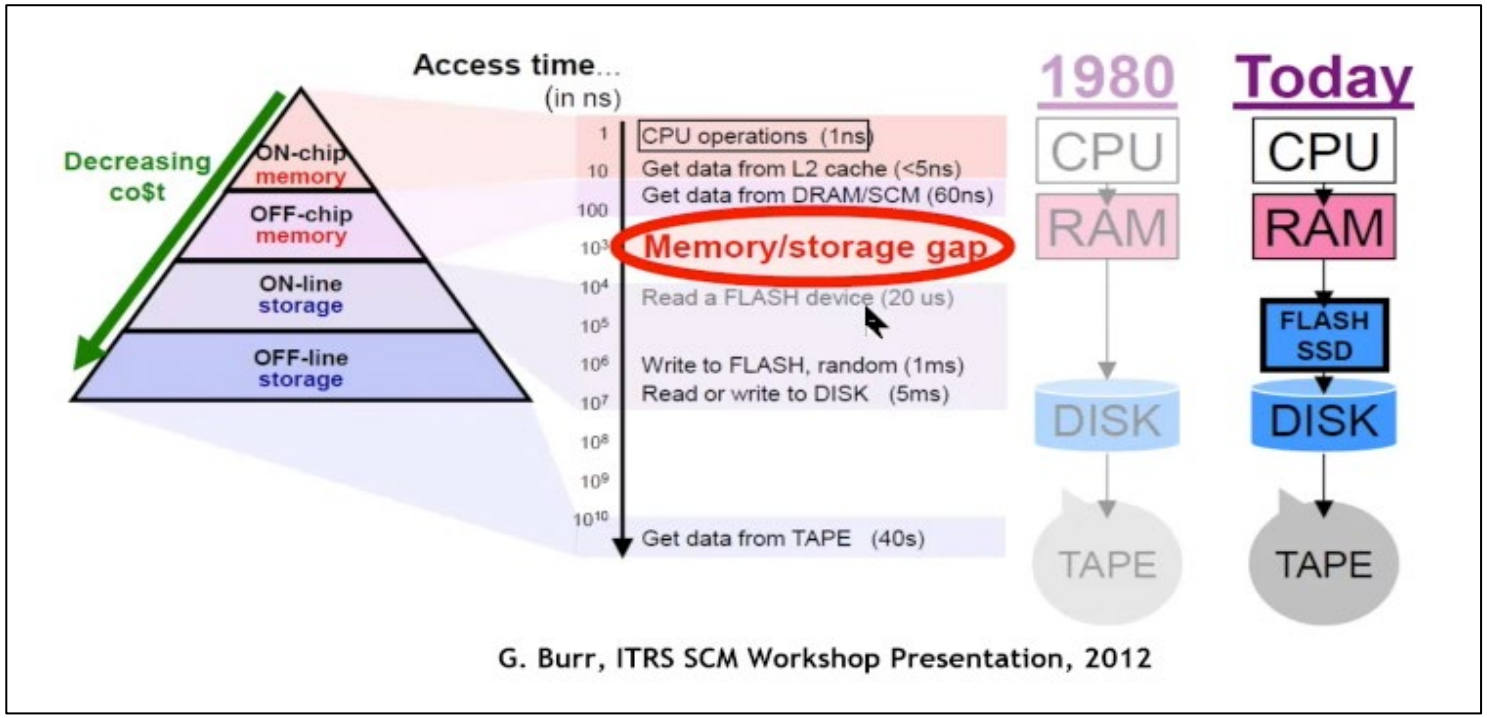
Increase number and width of words



Memory hierarchy

- Want inexpensive, fast memory
- Main memory
  - Large, inexpensive, slow memory stores entire program and data
- Cache
  - Small, expensive, fast memory stores copy of likely accessed parts of larger memory
  - Can be multiple levels of cache





Cache

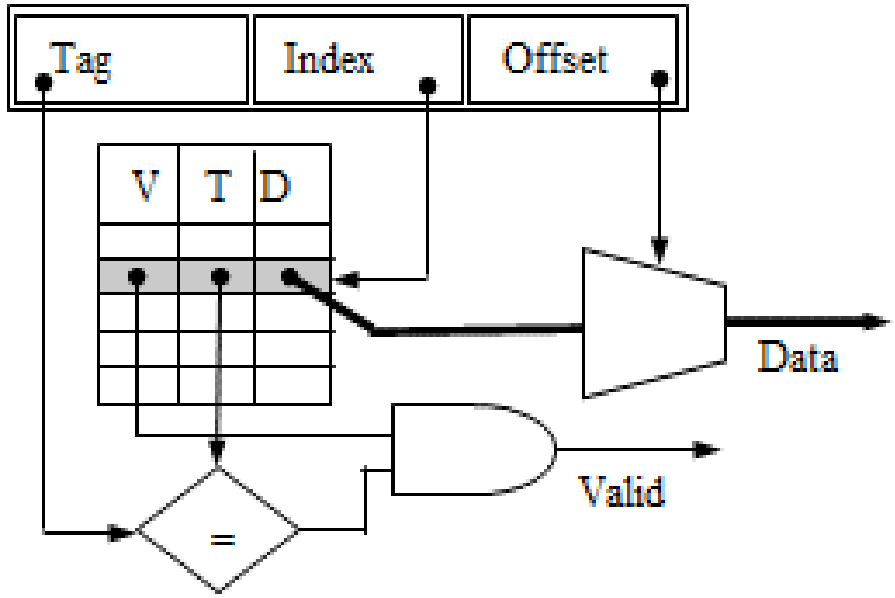
- **Usually designed with SRAM**
  - faster but more expensive than DRAM
- **Usually on same chip as processor**
  - space limited, so much smaller than off-chip main memory
  - faster access (1 cycle vs. several cycles for main memory)
- **Cache operation:**
  - Request for main memory access (read or write)
  - First, check cache for copy
    - cache hit
      - copy is in cache, quick access
    - cache miss
      - copy not in cache, read address and possibly its neighbors into cache
- **Several cache design choices**
  - cache mapping, replacement policies, and write techniques

Cache mapping

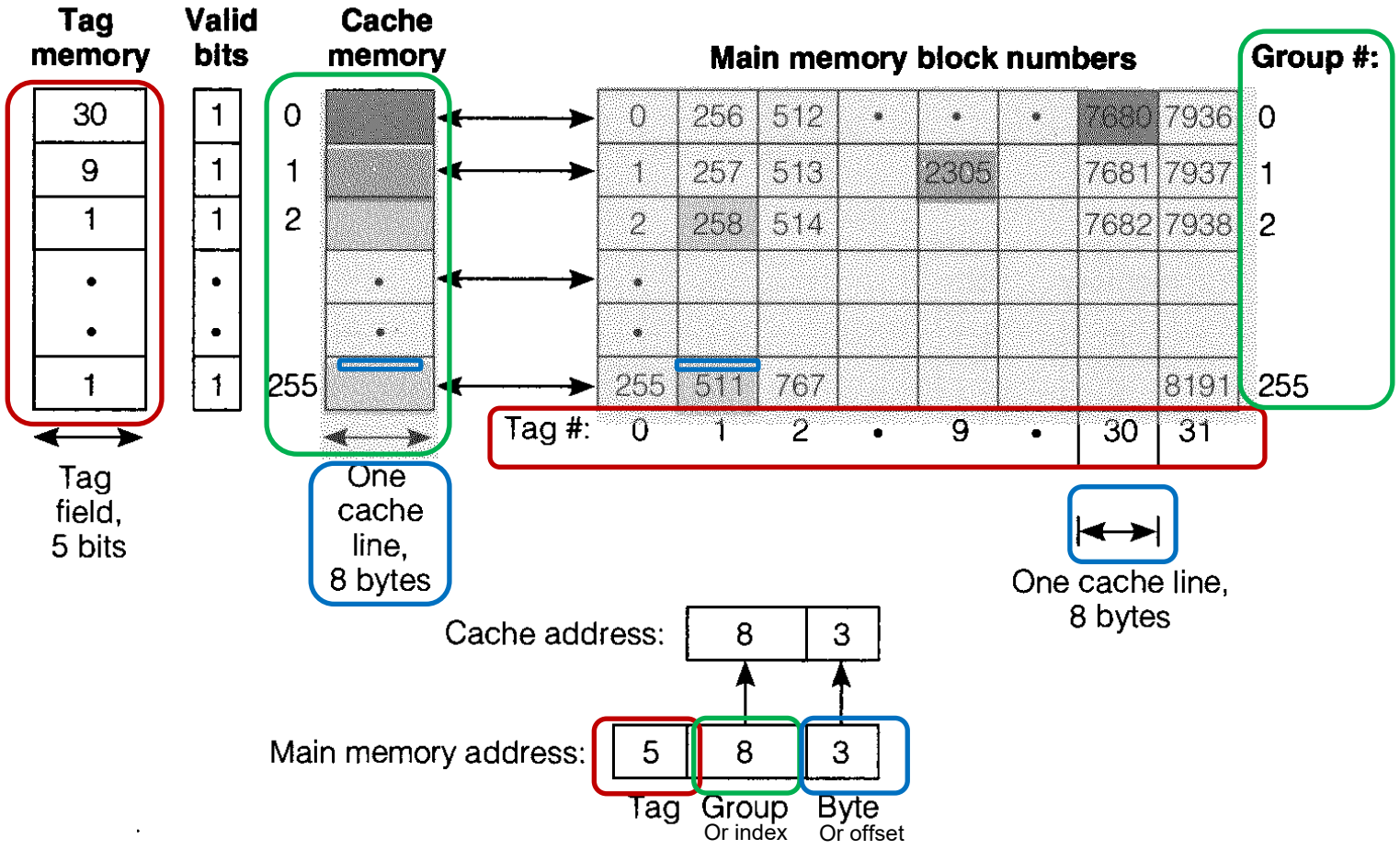
- Far fewer number of available cache addresses
- Are address' contents in cache?
- Cache mapping used to assign main memory address to cache address and determine hit or miss
- Three basic techniques:
  - Direct mapping
  - Fully associative mapping
  - Set-associative mapping
- Caches partitioned into indivisible blocks or lines of adjacent memory addresses
  - usually 4 or 8 addresses per line

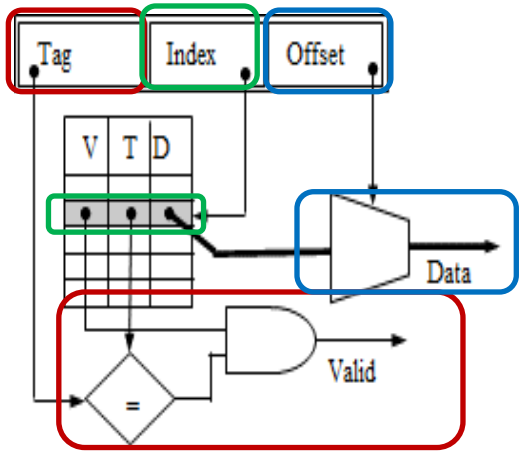
Direct mapping

- Main memory address divided into 2 fields
  - Index
    - cache address
    - number of bits determined by cache size
  - Tag
    - compared with tag stored in cache at address indicated by index
    - if tags match, check valid bit
- Valid bit
  - indicates whether data in slot has been loaded from memory
- Offset
  - used to find particular word in cache line



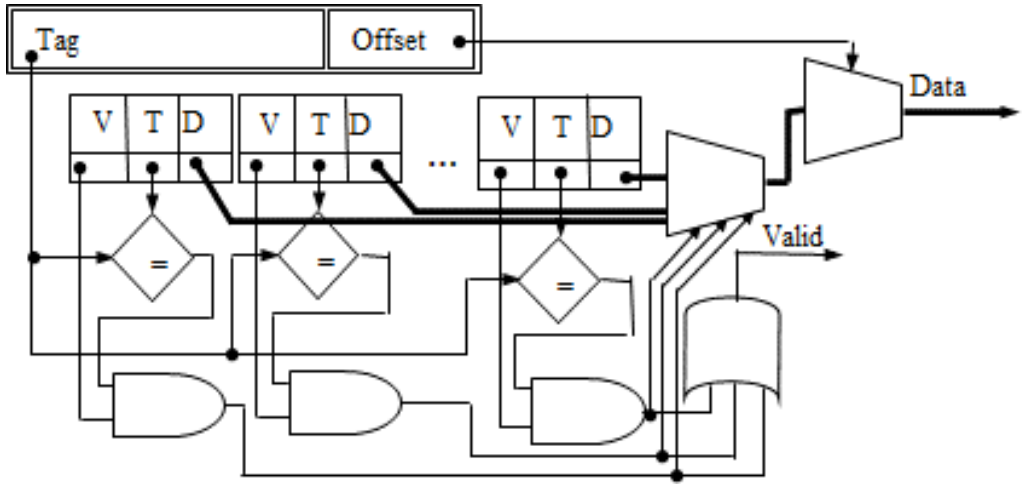
Direct mapping



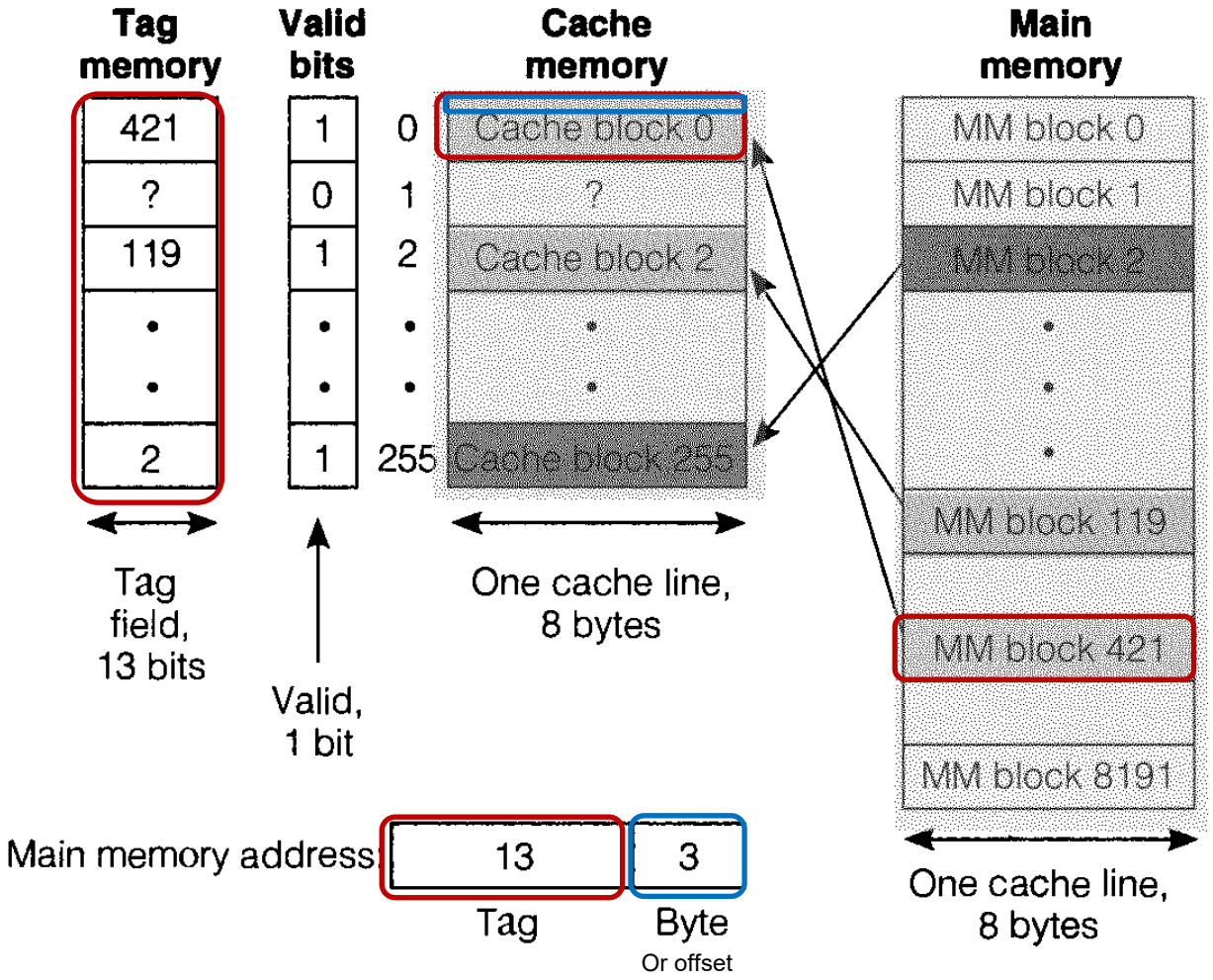


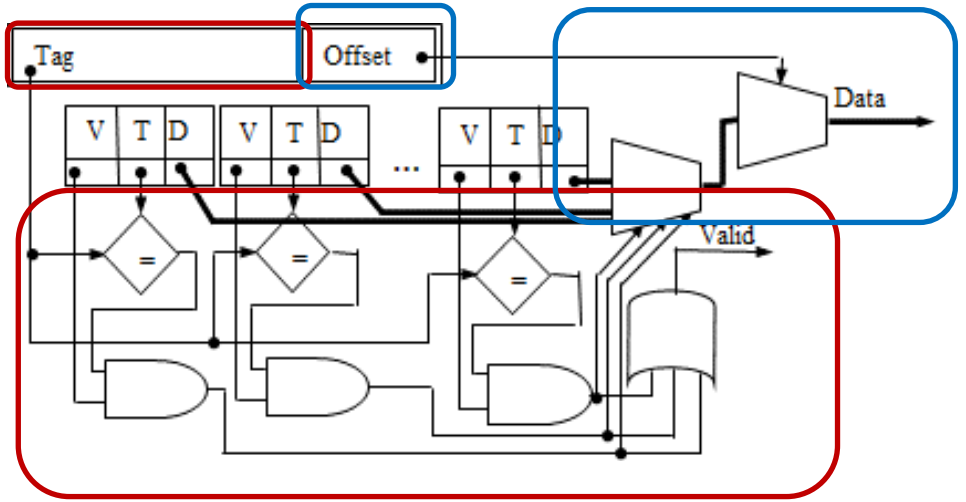
Fully associative mapping

- Complete main memory address stored in each cache address
- All addresses stored in cache simultaneously compared with desired address
- Valid bit and offset same as direct mapping



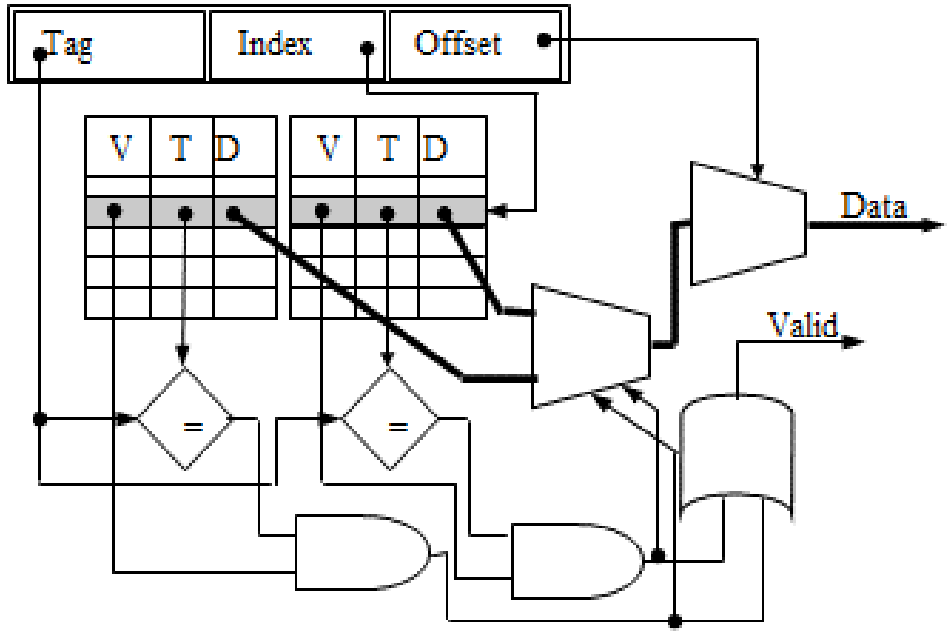
Fully associative mapping



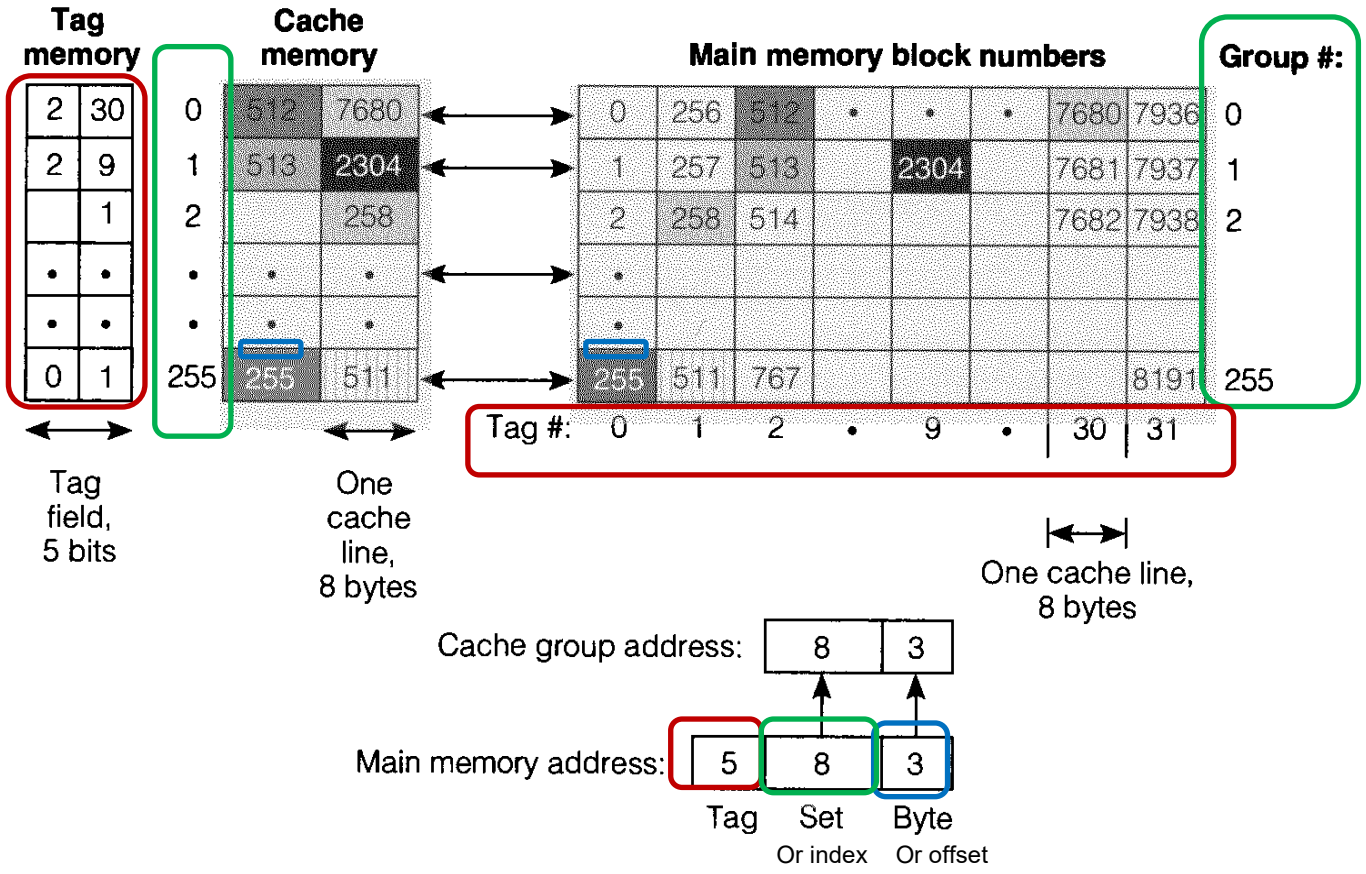


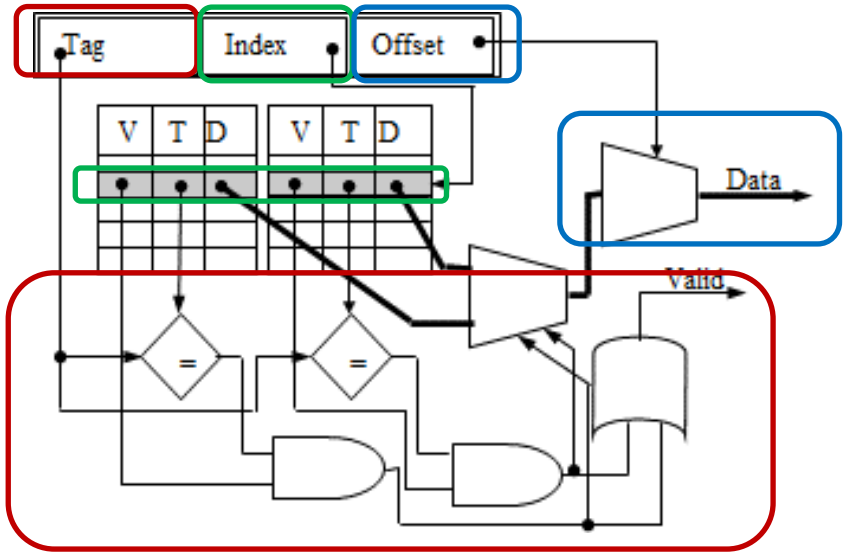
Set-associative mapping

- Compromise between direct mapping and fully associative mapping
- Index same as in direct mapping
- But, each cache address contains content and tags of 2 or more memory address locations
- Tags of that **set** simultaneously compared as in fully associative mapping
- Cache with set size N called N-way set-associative
  - 2-way, 4-way, 8-way are common



Set-associative mapping





Cache-replacement policy

- Technique for choosing which block to replace
  - when fully associative cache is full
  - when set-associative cache's line is full
- Direct mapped cache has no choice
- Random
  - replace block chosen at random
- LRU: least-recently used
  - replace block not accessed for longest time
- FIFO: first-in-first-out
  - push block onto queue when accessed
  - choose block to replace by popping queue

Cache write techniques

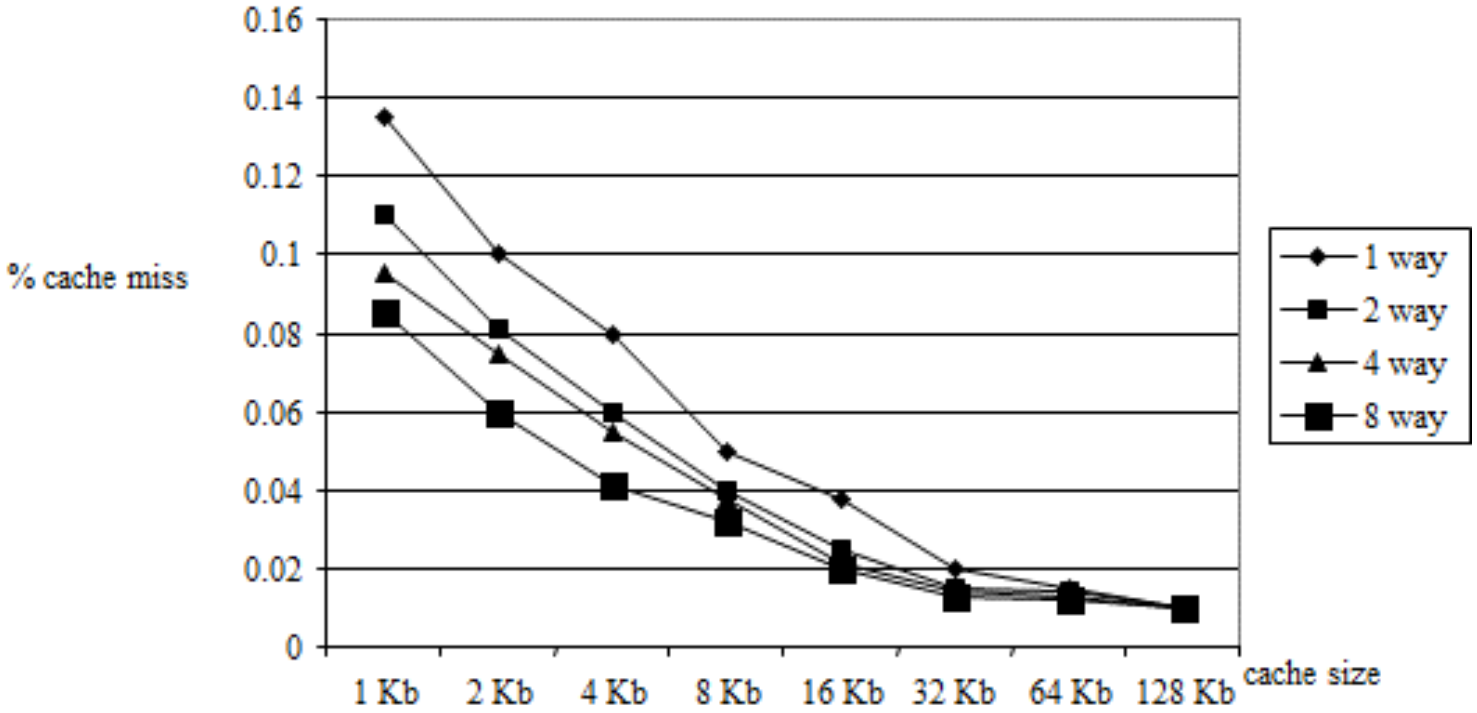
- When written, data cache must update main memory
- Write-through
  - write to main memory whenever cache is written to
  - easiest to implement
  - processor must wait for slower main memory write
  - potential for unnecessary writes
- Write-back
  - main memory only written when “dirty” block replaced
  - extra dirty bit for each block set when cache block written to
  - reduces number of slow main memory writes

Cache impact on system performance

- Most important parameters in terms of performance:
  - Total size of cache
    - total number of data bytes cache can hold
    - tag, valid and other house keeping bits not included in total
  - Degree of associativity
  - Data block size
- Larger caches achieve lower miss rates but higher access cost
  - e.g.,
    - 2 Kbyte cache: miss rate = 15%, hit cost = 2 cycles, miss cost = 20 cycles
      - avg. cost of memory access = (0.85 * 2) + (0.15 * 20) = 4.7 cycles
    - 4 Kbyte cache: miss rate = 6.5%, hit cost = 3 cycles, miss cost will not change
      - avg. cost of memory access = (0.935 * 3) + (0.065 * 20) = 4.105 cycles **(improvement)**
    - 8 Kbyte cache: miss rate = 5.565%, hit cost = 4 cycles, miss cost will not change
      - avg. cost of memory access = (0.94435 * 4) + (0.05565 * 20) = 4.8904 cycles **(worse)**

Cache performance trade-offs

- Improving cache hit rate without increasing size
  - Increase line size
  - Change set-associativity

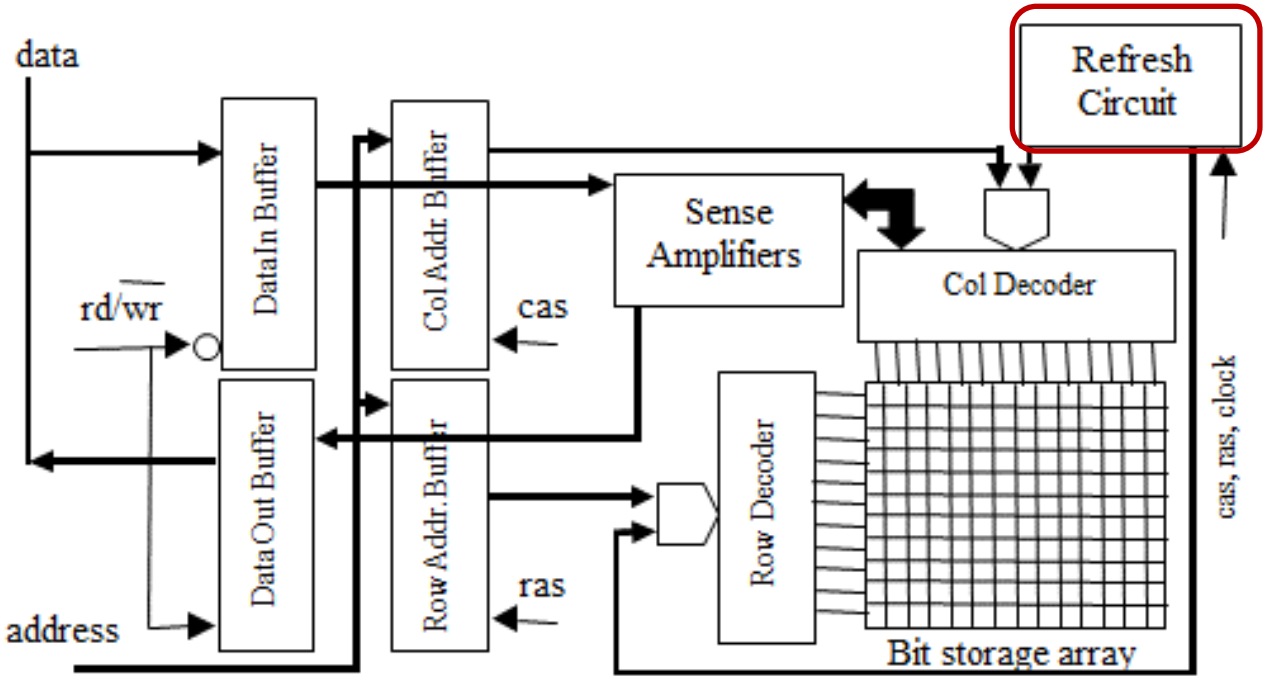


Advanced RAM

- DRAMs commonly used as main memory in processor based embedded systems
  - high capacity, low cost
- Many variations of DRAMs proposed
  - need to keep pace with processor speeds
  - FPM DRAM: fast page mode DRAM
  - EDO DRAM: extended data out DRAM
  - SDRAM/ESDRAM: synchronous and enhanced synchronous DRAM
  - RDRAM: rambus DRAM

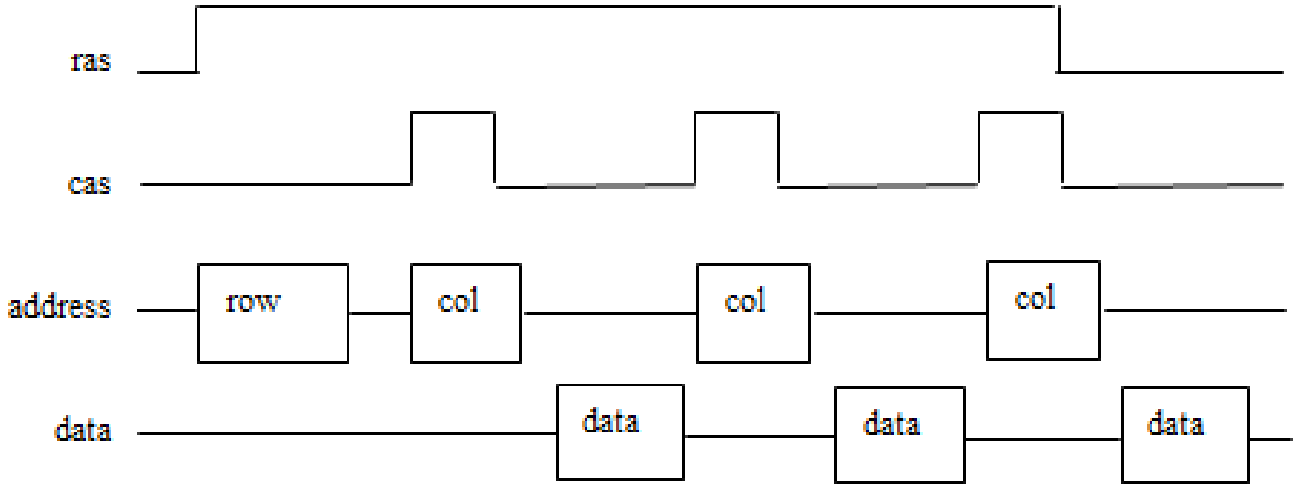
Basic DRAM

- Address bus multiplexed between row and column components
- Row and column addresses are latched in, sequentially, by strobing *ras* and *cas* signals, respectively
- Refresh circuitry can be external or internal to DRAM device
  - strobes consecutive memory address periodically causing memory content to be refreshed
  - Refresh circuitry disabled during read or write operation



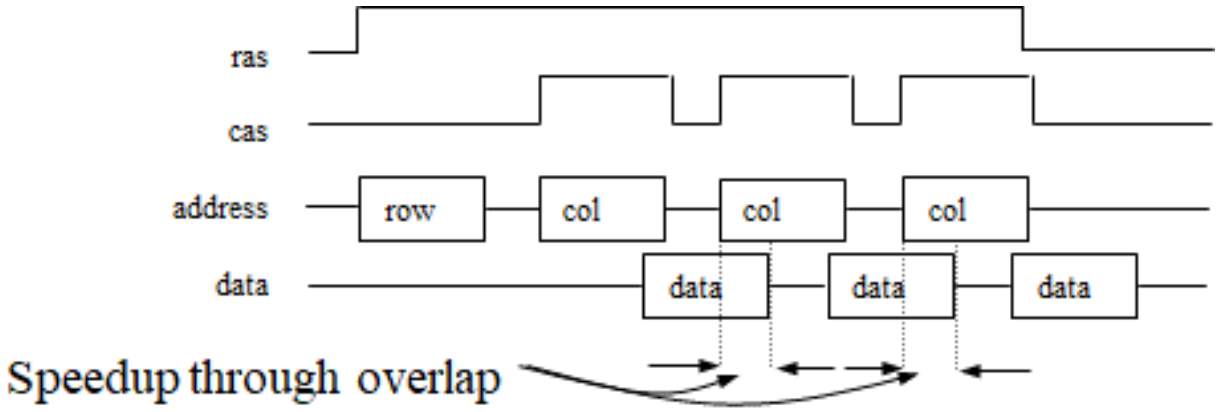
Fast Page Mode DRAM (FPM DRAM)

- Each row of memory bit array is viewed as a page
- Page contains multiple words
- Individual words addressed by column address
- Timing diagram:
  - row (page) address sent
  - 3 words read consecutively by sending column address for each
- Extra cycle eliminated on each read/write of words from same page



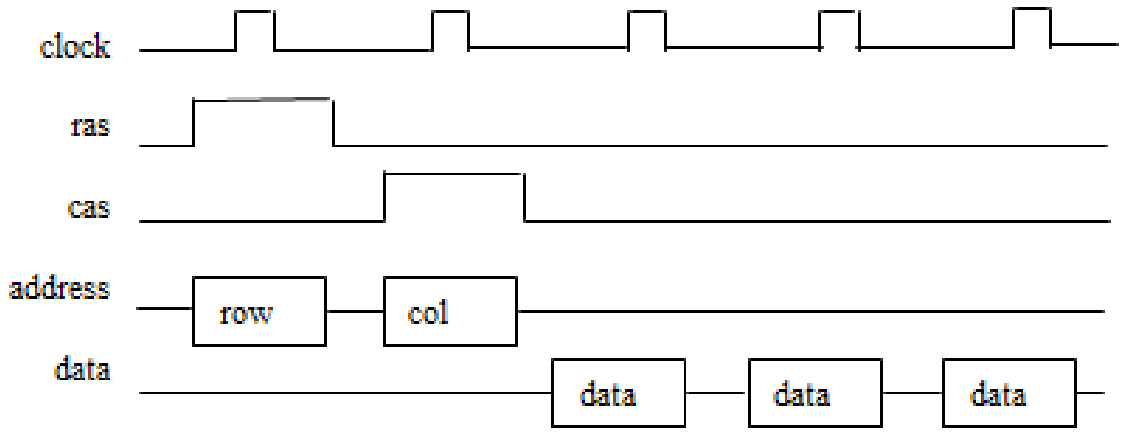
Extended data out DRAM (EDO DRAM)

- Improvement of FPM DRAM
- Extra latch before output buffer
  - allows strobing of *cas* before data read operation completed
- Reduces read/write latency by additional cycle



(S)ynchronous and Enhanced Synchronous (ES) DRAM

- SDRAM latches data on active edge of clock
- Eliminates time to detect *ras/cas* and *rd/wr* signals
- A counter is initialized to column address then incremented on active edge of clock to access consecutive memory locations
- ESDRAM improves SDRAM
  - added buffers enable overlapping of column addressing
  - faster clocking and lower read/write latency possible



Rambus DRAM (RDRAM)

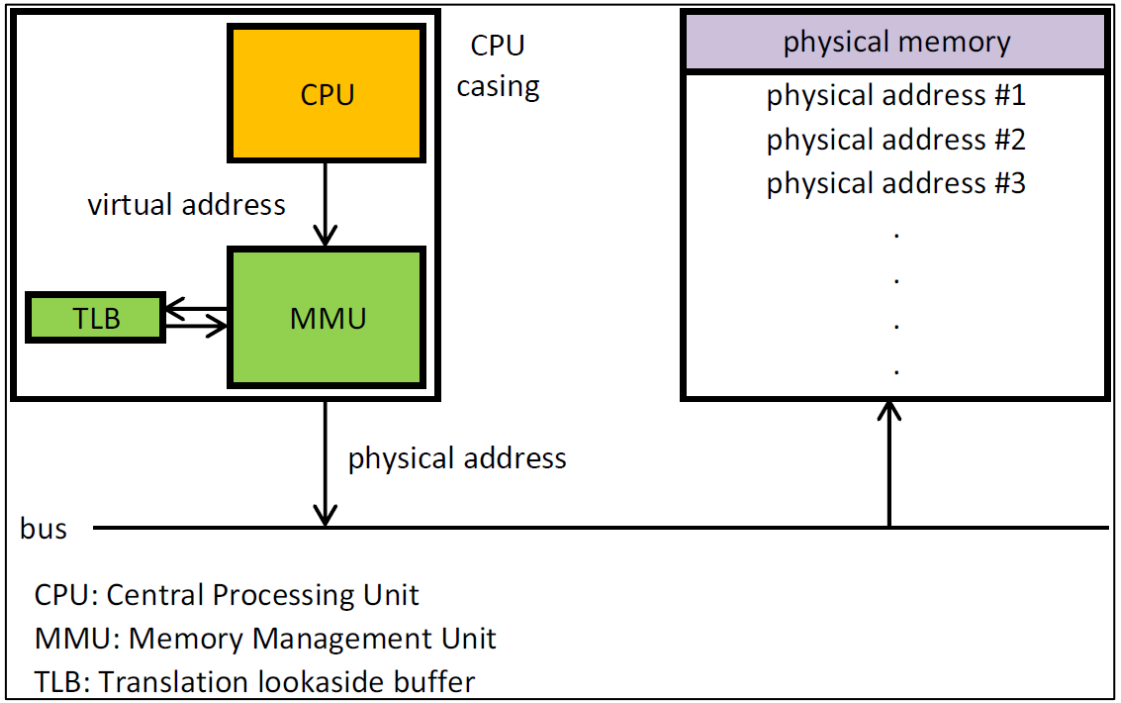
- More of a bus interface architecture than DRAM architecture
- Data is latched on both rising and falling edge of clock
- Broken into 4 banks each with own row decoder
  - can have 4 pages open at a time
- Capable of very high throughput

DRAM integration problem

- SRAM easily integrated on same chip as processor
- DRAM more difficult
  - Different chip making process between DRAM and conventional logic
  - Goal of conventional logic (IC) designers:
    - minimize parasitic capacitance to reduce signal propagation delays and power consumption
  - Goal of DRAM designers:
    - create capacitor cells to retain stored information
  - Integration processes beginning to appear

Memory Management Unit (MMU)

- Duties of MMU
  - Handles DRAM refresh, bus interface and arbitration
  - Takes care of memory sharing among multiple processors
  - Translates logic memory addresses from processor to physical memory addresses of DRAM
- Modern CPUs often come with MMU built-in
- Single-purpose processors can be used



Week 13 Module

- Reading from Vahid Book – Chapter 5
- Quiz 11 – 10 Questions from today's lecture on memories
- HW11 – VHDL (Need to create a FSM)

Continue:

- Lab 5 - Two week lab - Processor
- Final Project due May 3 – see details on Canvas

Preview of communication protocols

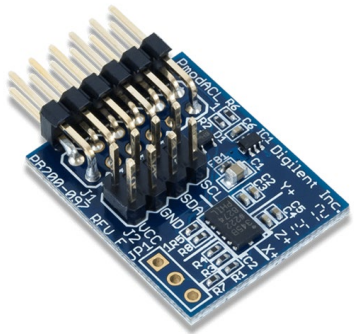
- PMODS typically use one or more of these:

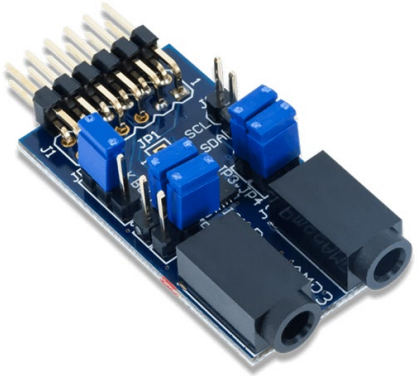
- I2C – Inter IC
- SPI – Serial Peripheral Interface
- UART – Universal Asynchronous Receiver/Transmitter
- GPIO – General Purpose input/output

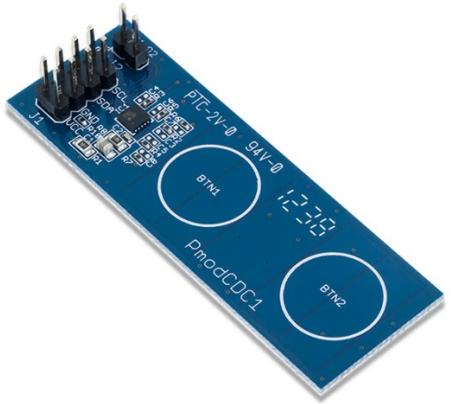
- Need to read PMOD manual for more information. Following slides show some links for sample code.

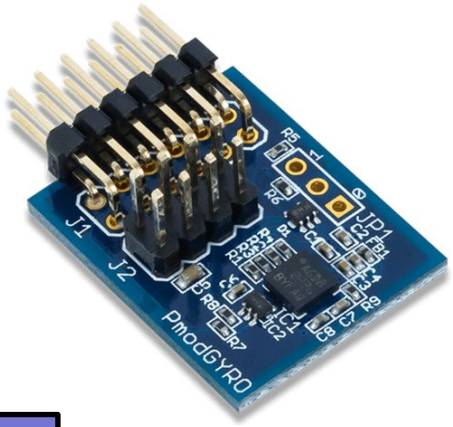
Example PMODs using I2C

PMOD	Description
ACL	3-axis accelerometer
AMP3	Stereo Power Amplifier
CDC1	Capacitive Input Buttons
GYRO	3-axis Digital Gyroscope
HYGRO	Digital Humidity and Temperature Sensor
IOXP	I/O Expansion Module
RTCC	Real-Time Clock/Calendar
TMP3	Digital Temperature Sensor







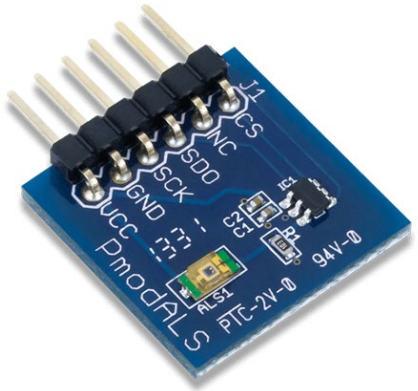


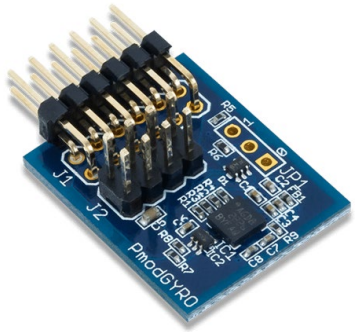
I2C Master (VHDL)
I2C Transceiver (VHDL)

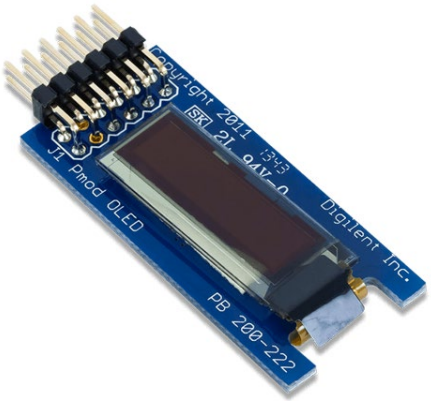
Read support pages for respective PMOD for more information & implementation

Example PMODs using SPI

PMOD	Description
ACL	3-axis accelerometer
AD1	Two 12-bit A/D inputs
ALS	Ambient Ligth Sensor
DA1	Four 8-bit D/A Outputs
DPG1	Differential Pressure Gauge Sensor
GYRO	3-axis Digital Gyroscope
JSTK2	Two-axis Joystick
OLED	128 x 32 Pixel Monochromatic OLED Display





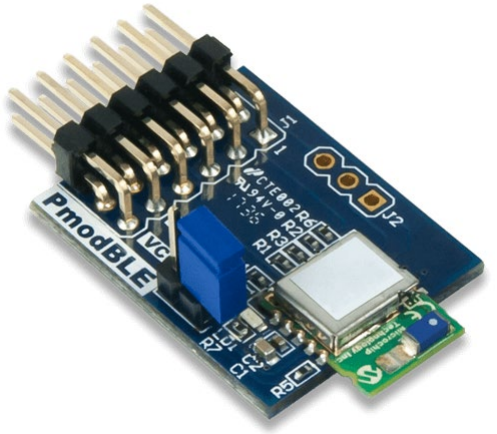


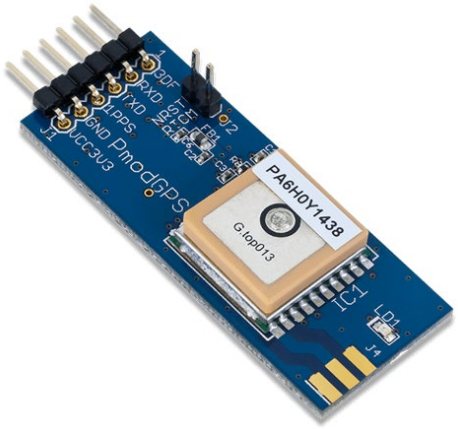
[SPI Master (VHDL)]
[SPI Slave (VHDL)]

Some SPI are senders or receivers only
Read support pages for respective PMOD for more information & implementation

Example PMODs using UART

PMOD	Description
BLE	Bluetooth Low Energy Inferface
BT2	Bluetooth Inferface
GPS	GPS Receiver
MAXSONAR	Maxbotic Ultrasonic Range Finder
USBUART	USB to UART Interface

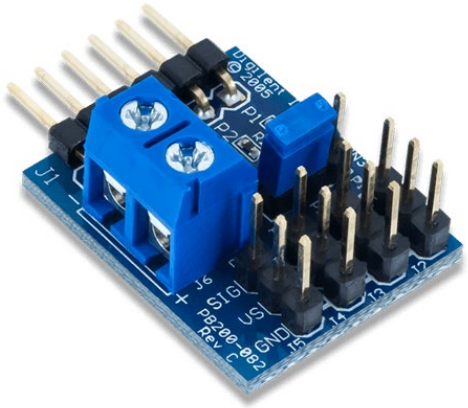


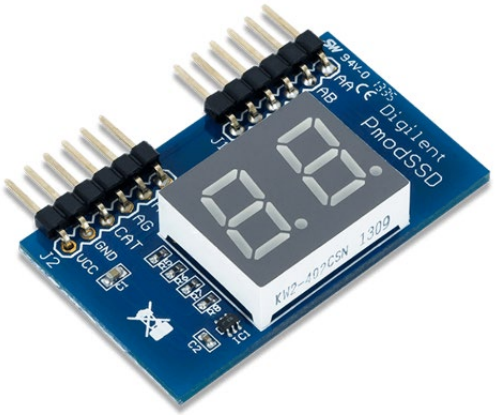


Read support pages for respective PMOD for more information & implementation

PMODs using GPIO (Custom)

- General Purpose I/O
- **Not a protocol**
- PMOD responds to a set of prescribed conditions or signals as defined for its functionality





PMOD	Description
CON3	R/C Servo Connectors
KYPD	16-button Keypad
SSD	Seven-Segment Display
STEP	Stepper Motor Driver

Read support pages for respective PMOD for more information & implementation

Lab Summary

	Lab	Name	Description
☑	Lab Intro	Vivado Design Flow	Understand Vivado tool basics / Interface with Zybo
☑	Lab 0	Blinker	Code for basic counter block, analyze counter timing
☑	Lab Synth	Synthesis	Define, understand and use synthesis concepts (no lab report required)
☑	Lab 1	Clocks, counters & buttons	Clock divider, button de-bouncer, counter design
☑	Lab 2	The only time you have to do math	Full adder and opcode implementation
☑	Lab 3	Where no clock has gone before	State machines, UART interface
☑	Lab 4	Now you see it, now you don't	Video timing basics, VGA & HDMI
	Lab 5	It's all about the processors	ASIP implementation - Assembly Code - UART Interface - Memory - FSM - Video

Lab complexity advances as new topics are introduced

LAB 4 - COMMENTS

Lab 4 – Now you see it, now you don't

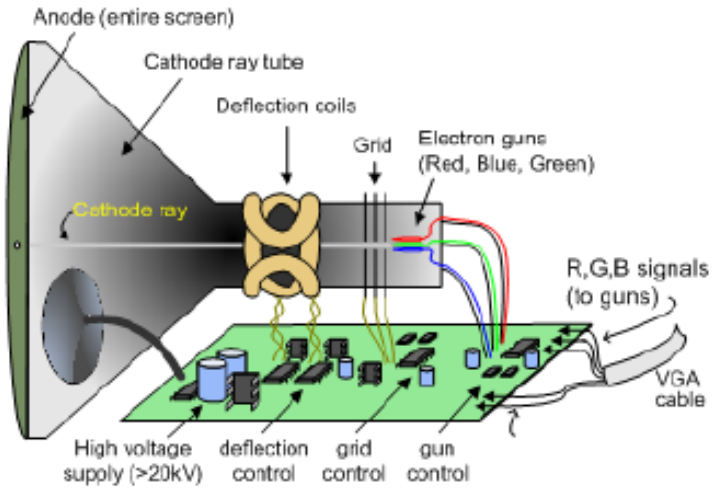
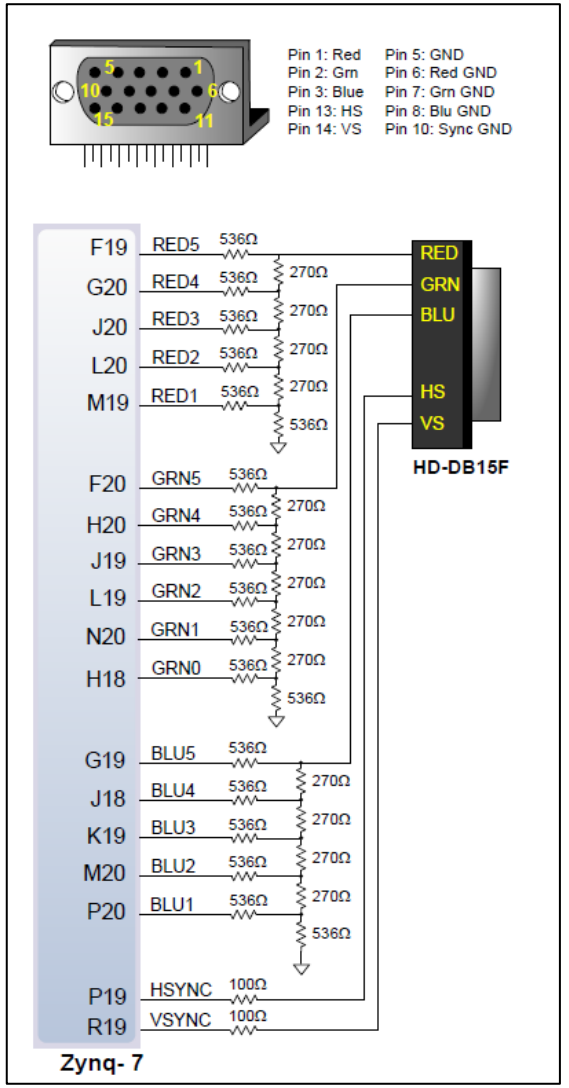


Figure 4.1: CRT Monitor Hardware [3]

VGA allows 16-bit color depth => 65,536 colors
R(5 bits), G(6 bits), B(5 bits)

The R-2R resistor ladder has a 0.7V maximum voltage – max brightness. Our bits control voltage applied to each channel.

Our memory is implemented for a 480x480 image display => 230400 depth with 8 bits per word (to minimize memory) with R(3), G(3) and B(2) – good enough for our JPG conversion.



R,G,B: Red, Green, Blue (8-bit color image for our VGA implementation)

Lab 4 – Now you see it, now you don't

Our HDMI Colors can be represented in hex or decimal form (24-bit color image)

Blue:
RGB Decimal 0,0,255
HEX = 0000FF

Decimal = 240+15=255
= (15 x 16) + (15 x 1)

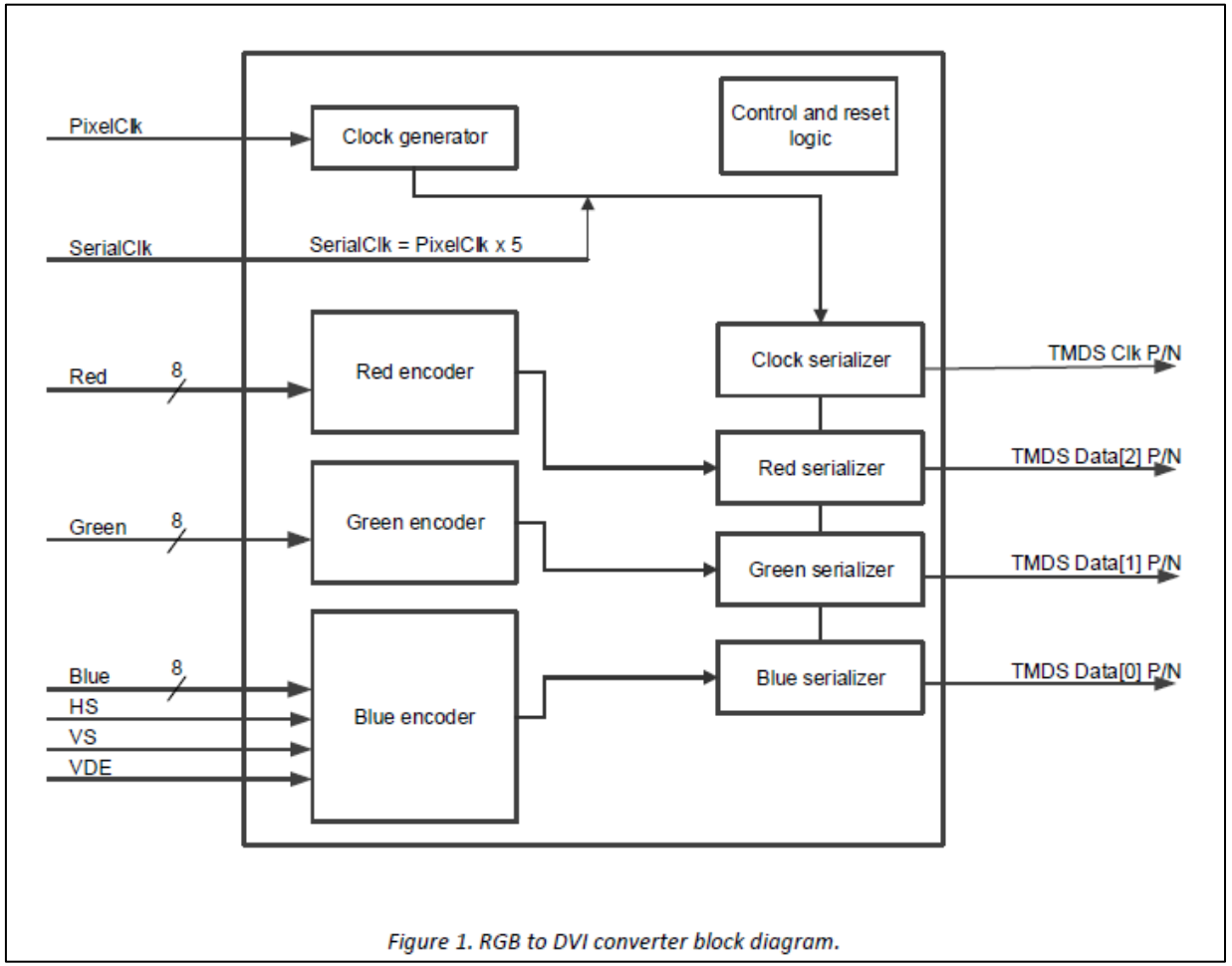
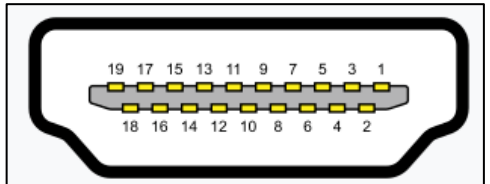


Figure 1. RGB to DVI converter block diagram.

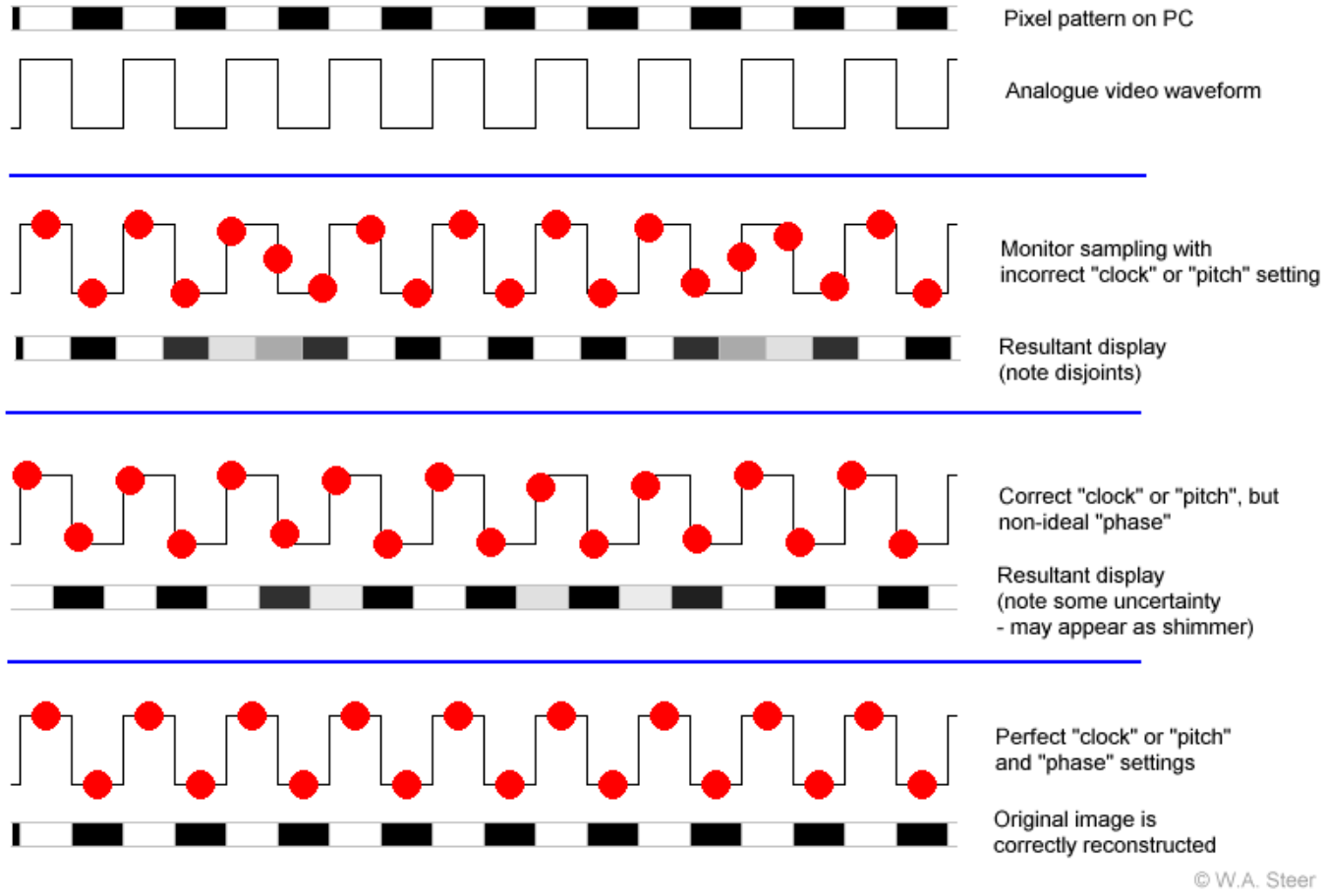


HDMI type A receptacle

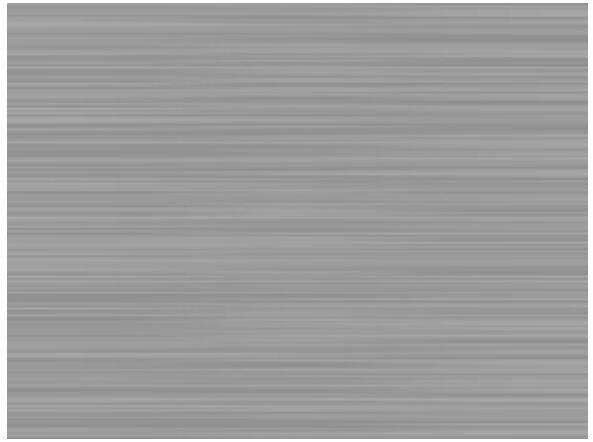
Pin 1	TMDS Data2+
Pin 2	TMDS Data2 Shield
Pin 3	TMDS Data2−
Pin 4	TMDS Data1+
Pin 5	TMDS Data1 Shield
Pin 6	TMDS Data1−
Pin 7	TMDS Data0+
Pin 8	TMDS Data0 Shield
Pin 9	TMDS Data0−
Pin 10	TMDS Clock+
Pin 11	TMDS Clock Shield
Pin 12	TMDS Clock−
Pin 13	Consumer Electronics Control (CEC)
Pin 14	Reserved (HDMI 1.0–1.3a) Utility/HEAC+ (HDMI 1.4+, optional, HDMI Ethernet Channel (HEC) and Audio Return Channel (ARC))
Pin 15	SCL (I^2C serial clock for DDC)
Pin 16	SDA (I^2C serial data for DDC)
Pin 17	Ground (for DDC, CEC, ARC, and HEC)
Pin 18	+5 V (up to 50 mA)
Pin 19	Hot Plug Detect (all versions) HEAC− (HDMI 1.4+, optional, HDMI Ethernet Channel and Audio Return Channel)

Conversion from RGB to HDMI on Zybo requires bit weight adjustment.
e.g. Multiply by 255/7 (RGB range/bit range) and transform to 8 bits.

Pixel Clock & Phase



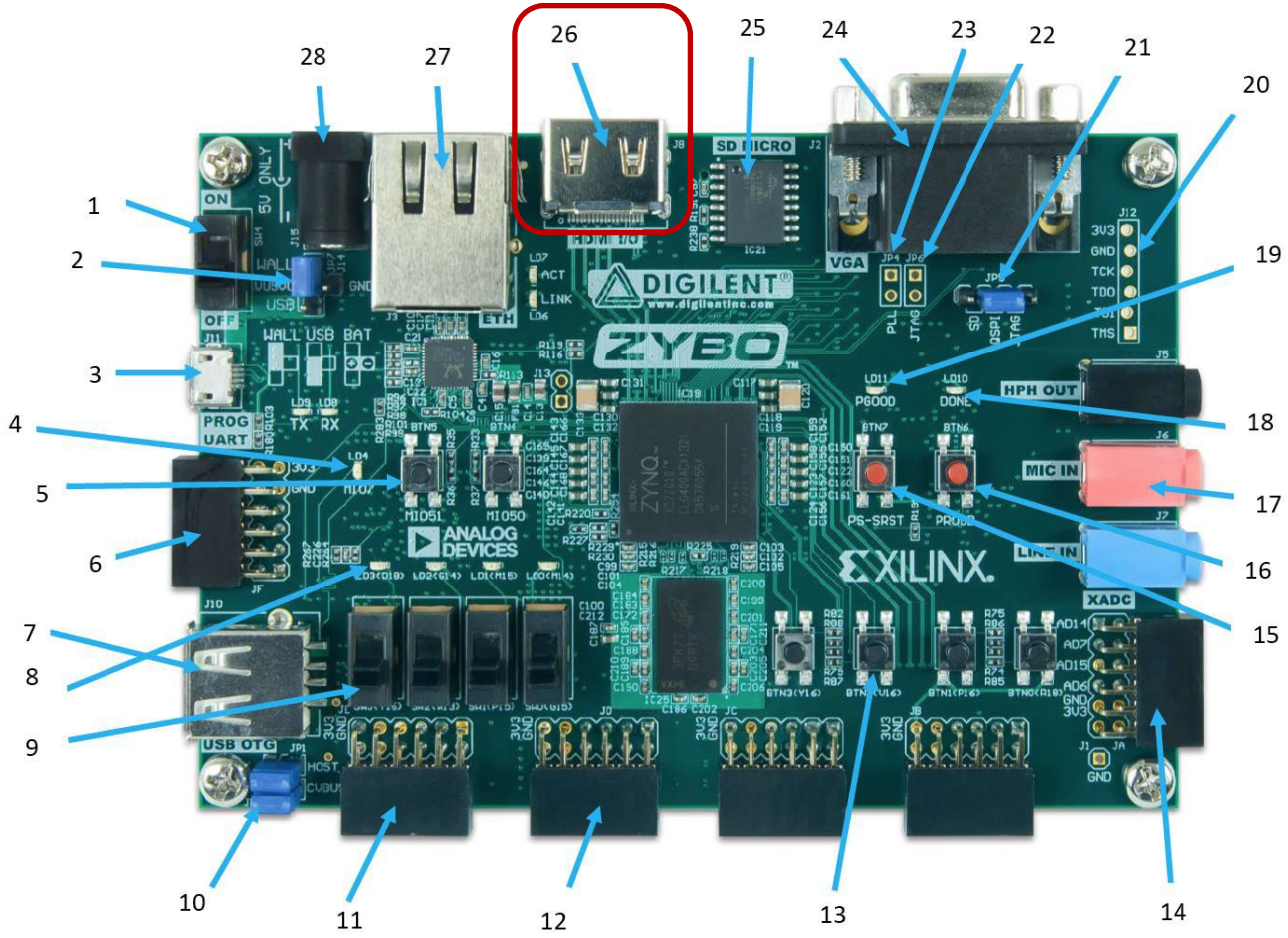
Incorrect Pixel Clock



Incorrect Phase – Adjust settings on monitor

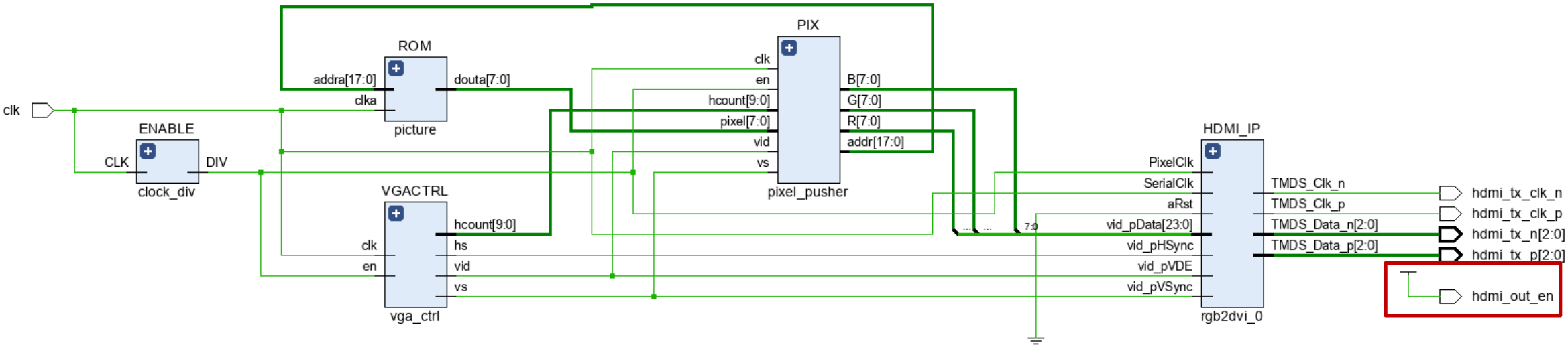
Source: http://www.techmind.org/lcd/phasexplan.html

DVI/HDMI PORT



- Instead of using VGA, you can use HDMI/DVI port
- PMOD IP that you can interface with this port

DVI/HDMI PORT



- DVI uses 8 bits for RGB, total of 24 bits per pixel
- Can fill ‘0’s from VGA but not the right method
- Better to scale VGA pixel info to DVI pixel info

LAB 5

Lab 5 – It's all about the Processors

- 2 weeks for lab – More information provided than prior semesters
- Everything you have done in past labs will be implemented in this one
  - ALU from Lab 2
  - UART from Lab 3
  - Pixel Pusher, vga_ctrl from Lab 4
  - Other blocks and implementations to be created

- Task 1: Assembly Code, Assembler, simulator provided
- Task 2: Create "regs" memory, and "framebuffer"
- Task 3: The Controller
  - Create FSM – States for each instruction (support assembly program instructions as minimum)
- Task 4: IP integrator, simulation

Assembly Code (provided)

```
// Lab 5 Assembly Program
.data
val1: str "hello_world"

.text
// send text over UART
ori $r3 $zero 0 // counter for string
ori $r4 $zero 0 // counter for video memory
ori $r5 $zero 1 // constant 1
ori $r6 $zero 4096 // constant for vid mem addr max

// read char, if not zero print it else end
beg:
lw $r7 $r3 val1
beq $r7 $zero loop2
send $r7
add $r3 $r3 $r5 // increment char pointer
j beg

loop2:
recv $r8
add $r4 $zero $zero

loop3:
beq $r4 $r6 loop2
wpix $r4 $r8
add $r4 $r4 $r5
j loop3

//end:
```

Data section

R3 = 0

R4 = 0

R5 = 1

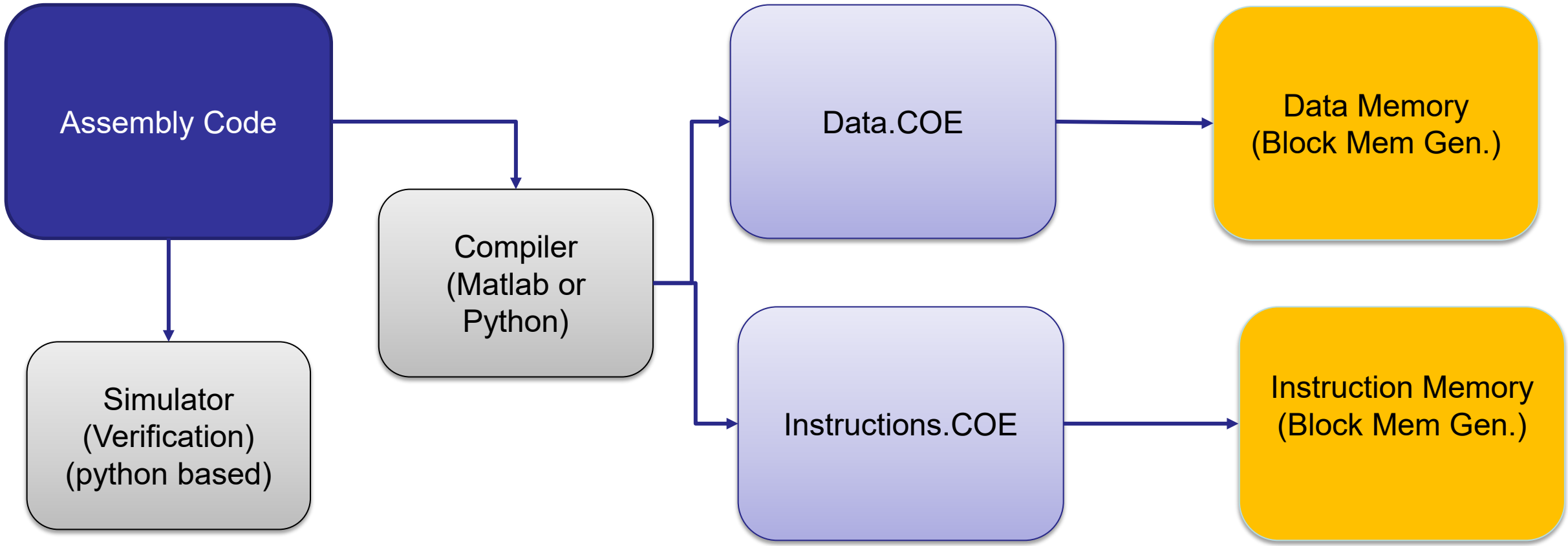
R6 = 4096

UART characters

Receive UART characters for video

Framebuffer fill loop

Lab 5 – It's all about the Processors



Lab 5 – Simulator

```
Parsed instructions:
[0]: ori $r3 $zero 0
[1]: ori $r4 $zero 0
[2]: ori $r5 $zero 1
[3]: ori $r6 $zero 4096
beg:
[4]: lw $r7 $r3 val1
[5]: beq $r7 $zero loop2
[6]: send $r7
[7]: add $r3 $r3 $r5
[8]: j beg
loop2:
[9]: recv $r8
[10]: add $r4 $zero $zero
loop3:
[11]: beq $r4 $r6 loop2
[12]: wpix $r4 $r8
[13]: add $r4 $r4 $r5
[14]: j loop3

Please enter desired breakpoints:
(one at a time as integers, negative number to finish)
Breakpoint:
```

Step 1: Choose a breakpoint
Step 2: Enter a negative number to see options
Step 3: Select different options from menu.
e..g. “regs”

```
Options:
        Show program ('prg')
        Show data labels ('dlabels')
        Show text labels ('tlabels')
        Show instruction ('instr')
        Show registers ('regs')
        Show breakpoints ('bps')
        Add breakpoint ('addbp')
        Remove breakpoint ('rmbp')
        Step a single instruction ('step')
        Peek at video memory ('vPeek')
        Peek at data memory ('dPeek')
        Poke a register value ('rPoke')
        Poke a video memory value ('vPoke')
        Poke a data memory value ('dPoke')
        Repeat this command list ('help')
        Exit debug shell ('exit')


Enter your choice:
```

Lab 5 – It's all about the Processors

Data Memory
(Block Mem Gen.)
32K x 16
SRAM

Framebuffer
4K x 16
Dual Port RAM
(single write port)

Instruction Memory
(Block Mem Gen.)
16K x 32
ROM

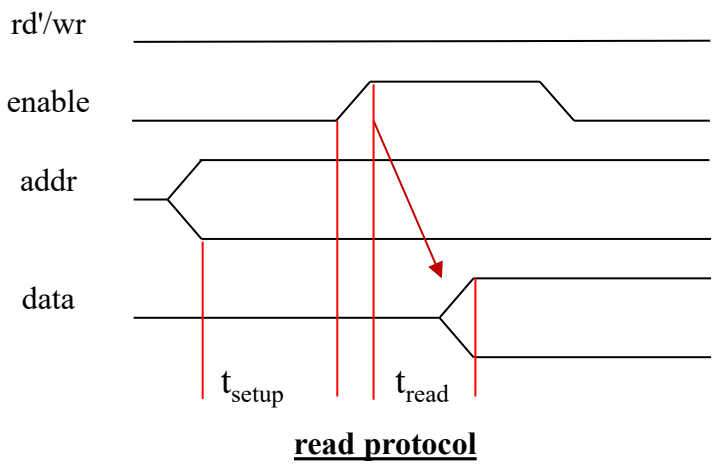
Registers
32 x 16
Dual Port RAM
(read & write)

- Define memory / regs blocks
- Bring in other blocks into the IP integrator GUI:
  - UART
  - Pixel Pusher
  - VGA_ctrl
  - ALU
  - Clk_Div
  - Clk_Div (25 MHz)
  - Debounce

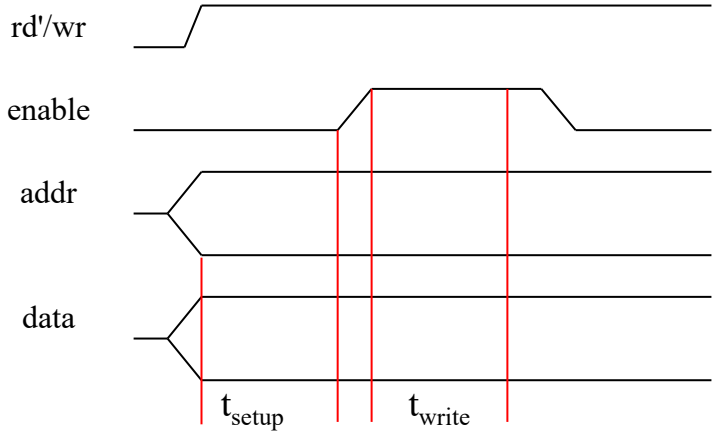
Table 5.4: GRISC Register File Organization

Register Index	Alias	Size (bits)	Special Behavior
0	$zero	16	Resets to 0 every clock tick
1	$pc	16	Program Counter
2	$ra	16	Return Address for JAL
3-31	$r3-$r31	16	

Lab 5 – It's all about the Processors



read protocol



write protocol

Control the memory by setting read / write signals inside state machine. Example: (Cycle by Cycle)

1. Set address & data bus
2. Assert write enable
3. De-assert write enable

Lab 5 – Instruction types

Instruction Type	Opcode (4 downto 3)	Format
R	0b00 or 0b01	[op][reg1][reg2][reg3]
I	0b10	[op][reg1][reg2][imm]
J	0b11	[op][imm]

Lab 5 – It's all about the Processors

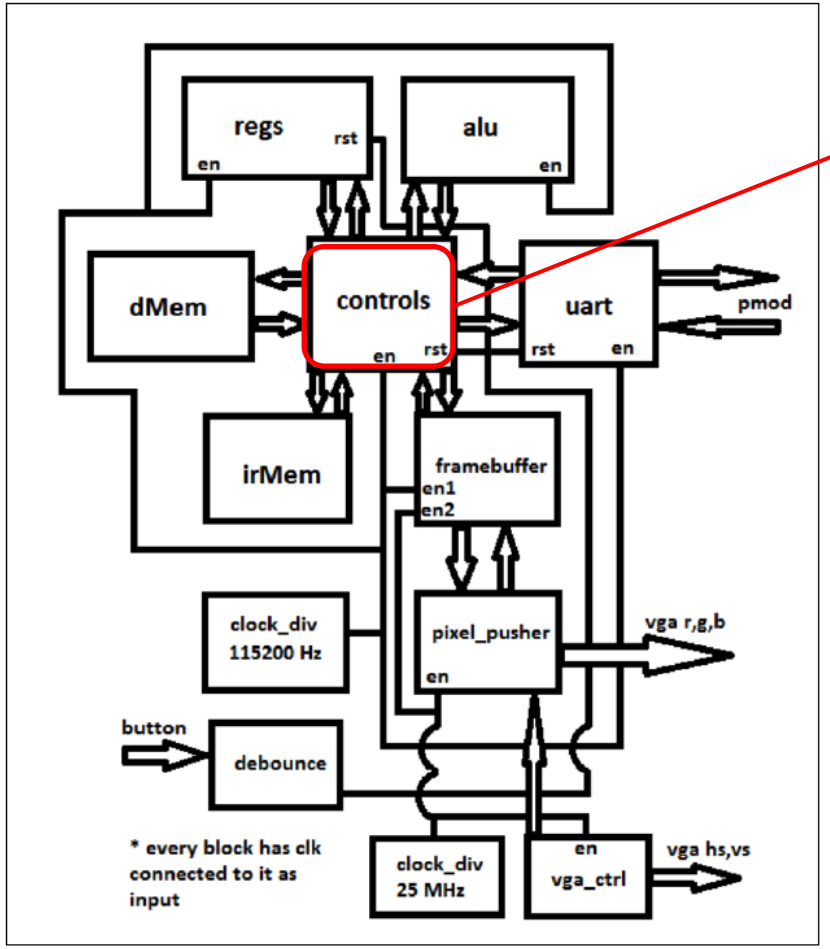
Table 5.3: GRISC ISA

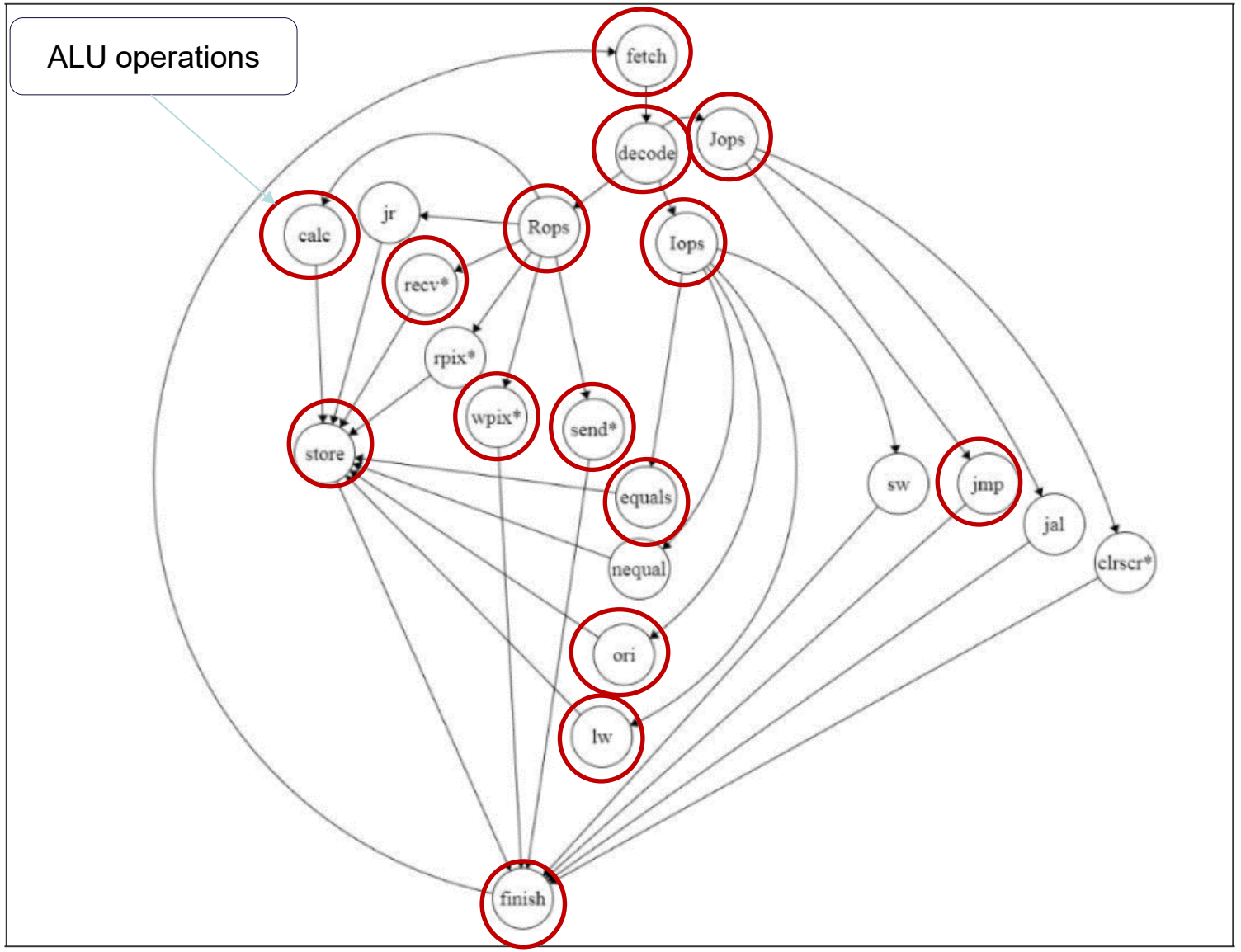
Instruction	Opcode	Format	Meaning
add	0b00000	[op][reg1][reg2][reg3]	reg1 = reg2 + reg3
sub	0b00001	[op][reg1][reg2][reg3]	reg1 = reg2 - reg3
sll	0b00010	[op][reg1][reg2][*reg3]	reg1 = reg2 << 1
srl	0b00011	[op][reg1][reg2][*reg3]	reg1 = reg2 << 1
sra	0b00100	[op][reg1][reg2][*reg3]	reg1 = reg2 >>> 1
and	0b00101	[op][reg1][reg2][reg3]	reg1 = reg2 and reg3
or	0b00110	[op][reg1][reg2][reg3]	reg1 = reg2 or reg3
xor	0b00111	[op][reg1][reg2][reg3]	reg1 = reg2 xor reg3
slt	0b01000	[op][reg1][reg2][reg3]	reg1 = (reg2 < reg3) ? 1 : 0
sgt	0b01001	[op][reg1][reg2][reg3]	reg1 = (reg2 > reg3) ? 1 : 0
seq	0b01010	[op][reg1][reg2][reg3]	reg1 = (reg2 == reg3) ? 1 : 0
send (asip)	0b01011	[op][reg1][*reg2][*reg3]	sendUART(reg1[7:0])
recv (asip)	0b01100	[op][reg1][*reg2][*reg3]	reg1 = 0x00 & recvUART()
jr	0b01101	[op][reg1][*reg2][*reg3]	pc = reg1
wpix (asip)	0b01110	[op][reg1][reg2][*reg3]	framebuffer[reg1[11:0]] = reg2[15:0]
rpix (asip)	0b01111	[op][reg1][reg2][*reg3]	reg1 = framebuffer[reg2[11:0]]
beq	0b10000	[op][reg1][reg2][imm]	if(reg1 == reg2) pc = imm
bne	0b10001	[op][reg1][reg2][imm]	if(reg1 != reg2) pc = imm
ori	0b10010	[op][reg1][reg2][imm]	reg1 = reg2 or imm
lw	0b10011	[op][reg1][reg2][imm]	reg1 = dmem[reg2 + imm]
sw	0b10100	[op][reg1][reg2][imm]	dmem[reg2 + imm] = reg1
j	0b11000	[op][imm]	pc = imm
jal	0b11001	[op][imm]	ra = pc, pc = imm
clrscr (asip)	0b11010	[op][imm]	framebuffer[all] = 0

*Note: * means operand ignored*

8 instructions needed to run assembly program

Lab 5 FSM


regs
rst
en
alu
en
dMem
controls
en
rst
uart
rst
en
pmod
irMem
framebuffer
en1
en2
clock_div
115200 Hz
pixel_pusher
en
vga r,g,b
button
debounce
* every block has clk connected to it as input
clock_div
25 MHz
en
vga_ctrl
vga hs,vs


ALU operations
fetch
decode
Jops
calc
jr
Rops
Iops
recv*
rpix*
wpix*
send*
store
equals
nequal
sw
jmp
jal
clrscr*
ori
lw
finish

Lab 5 – It's all about the Processors

Finite State Machine

Table 5.2: "Controls" State Transition and Behavior Table

Current State	Actions	Next State
fetch	get current pc from reg into signal	decode
decode	store irMem[pc_signal] into a signal store pc_signal+1 into register 1	Rops if opcode top bits are 00 or 01 else Iops if 10 else Jops
Rops	Break up instruction into arguments. Use arguments to fetch register contents for reg2 and reg3 into signals	jr if opcode is 01101 else recv if 01100 else rpix if 01111 else wpix if 01110 else send if 01011 else calc
Iops	Break up instruction into arguments. Use arguments to fetch register contents for reg2 into signal	equals if opcode bottom 3 bits are 000 else nequal if 001 else ori if 010 else lw if 011 else sw
Jops	Break up instruction into arguments	jmp if opcode is 11000 else jal if 11001 else clrscr
calc	Apply the register operands and the correct opcode to the ALU and store the result into an alu result signal	store

```
case ps is
    when fetch =>
        rID1 <= "00001"; -- register 1 - PC
        ps <= fetch1;

    when fetch1 =>
        pc <= unsigned(regrD1(13 downto 0));
        ps <= decode;
```

Table 5.5: GRISC Instruction Decoding Table

Instruction Type	Opcode (4 downto 3)	Format
R	0b00 or 0b01	[op][reg1][reg2][reg3]
I	0b10	[op][reg1][reg2][imm]
J	0b11	[op][imm]

Lab 5 – It's all about the Processors

Finite State Machine

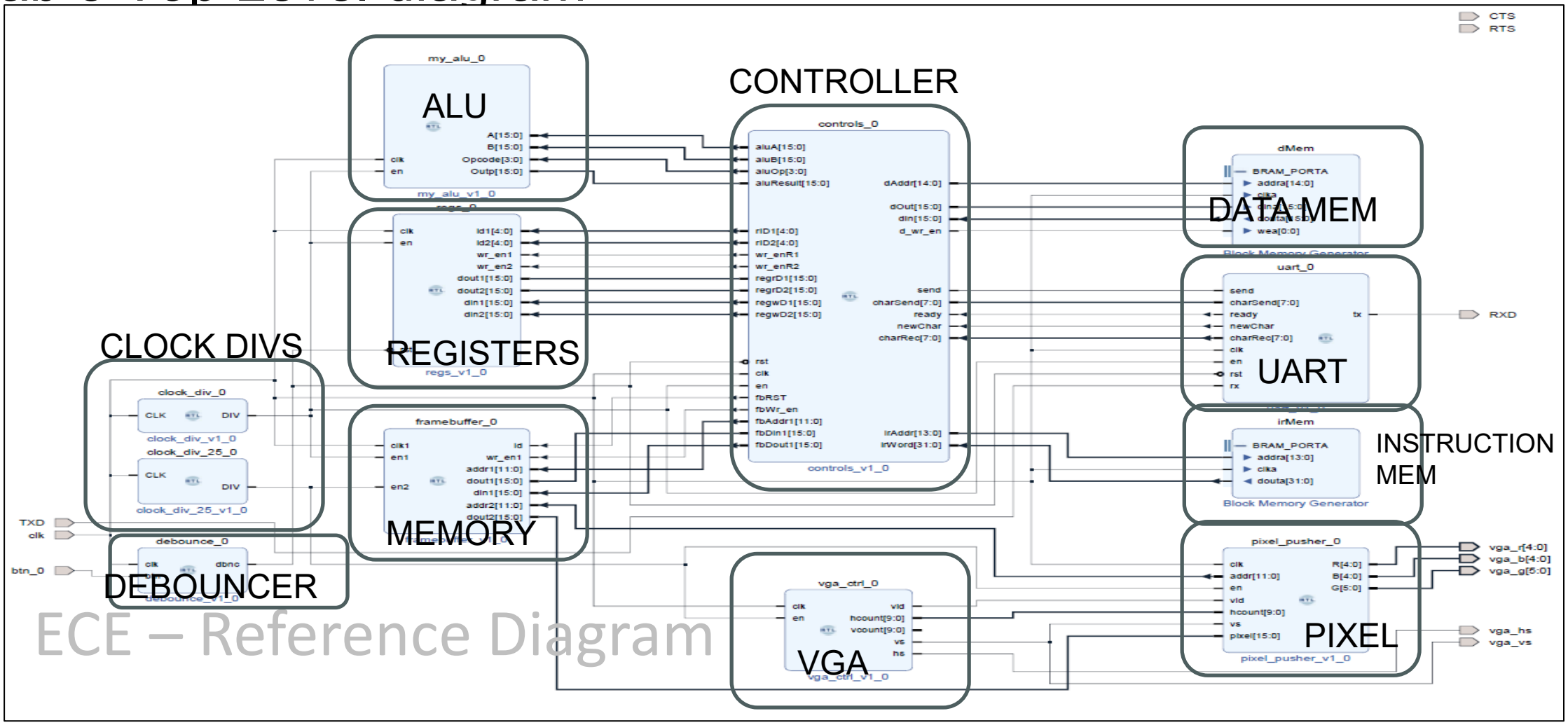
Table 5.3: GRISC ISA

Instruction	Opcode	Format	Meaning
add	0b00000	[op][reg1][reg2][reg3]	reg1 = reg2 + reg3
sub	0b00001	[op][reg1][reg2][reg3]	reg1 = reg2 - reg3
sll	0b00010	[op][reg1][reg2][*reg3]	reg1 = reg2 << 1
srl	0b00011	[op][reg1][reg2][*reg3]	reg1 = reg2 << 1
sra	0b00100	[op][reg1][reg2][*reg3]	reg1 = reg2 >>> 1
and	0b00101	[op][reg1][reg2][reg3]	reg1 = reg2 and reg3
or	0b00110	[op][reg1][reg2][reg3]	reg1 = reg2 or reg3
xor	0b00111	[op][reg1][reg2][reg3]	reg1 = reg2 xor reg3
slt	0b01000	[op][reg1][reg2][reg3]	reg1 = (reg2 < reg3) ? 1 : 0
sgt	0b01001	[op][reg1][reg2][reg3]	reg1 = (reg2 > reg3) ? 1 : 0
seq	0b01010	[op][reg1][reg2][reg3]	reg1 = (reg2 == reg3) ? 1 : 0

```
when calc =>
    if opcode = "00000" then -- add
        aluA <= reg2data;
        aluB <= reg3data;
        aluOp <= x"0";
        ps <= calc2;
    end if;

when calc2 =>
        regIdata <= aluResult;
        ps <= store;
```

Lab 5 Top Level diagram



Final project

For guidance on using PMODs and other useful information:

- Check all information posted on the Extra Credit module on Canvas to help on your final project

- **LAB 6 – More fun with Zynq:**
  - Uses ARM processor to read current time using RTCC pmod and display on the OLED PMOD
  - Useful Links provided

- **Lab 7ex – PWM**
  - Creates custom IP core for a Pulse Width Modulator controller

To the Lab