EECS 373

Design of Microprocessor-Based Systems

Prabal Dutta
University of Michigan

Lecture 4: Memory-Mapped I/O, Bus Architectures
January 20, 2015

Slides developed in part by
Mark Brehob & Prabal Dutta

Today…

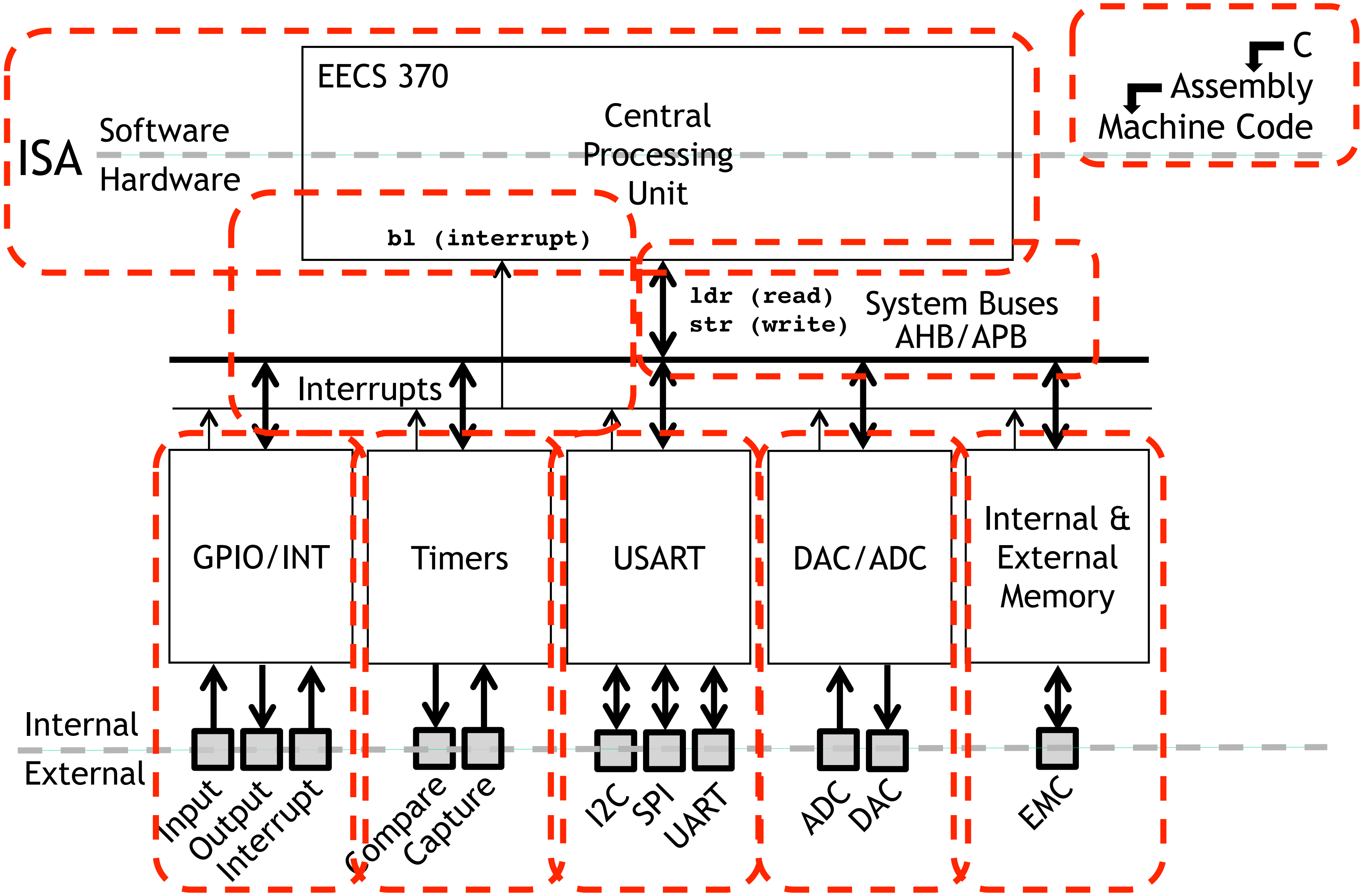
Memory-Mapped I/O

Example Bus with Memory-Mapped I/O

Bus Architectures

AMBA APB

Course Roadmap



Memory-mapped I/O

- Microcontrollers have many interesting peripherals
  - But how do you interact with them?

- Need to:
  - Send commands
  - Configure device
  - Receive data

- But we don't want new processor instructions for everything
  - Actually, it would be great if the processor didn't know anything weird was going on at all

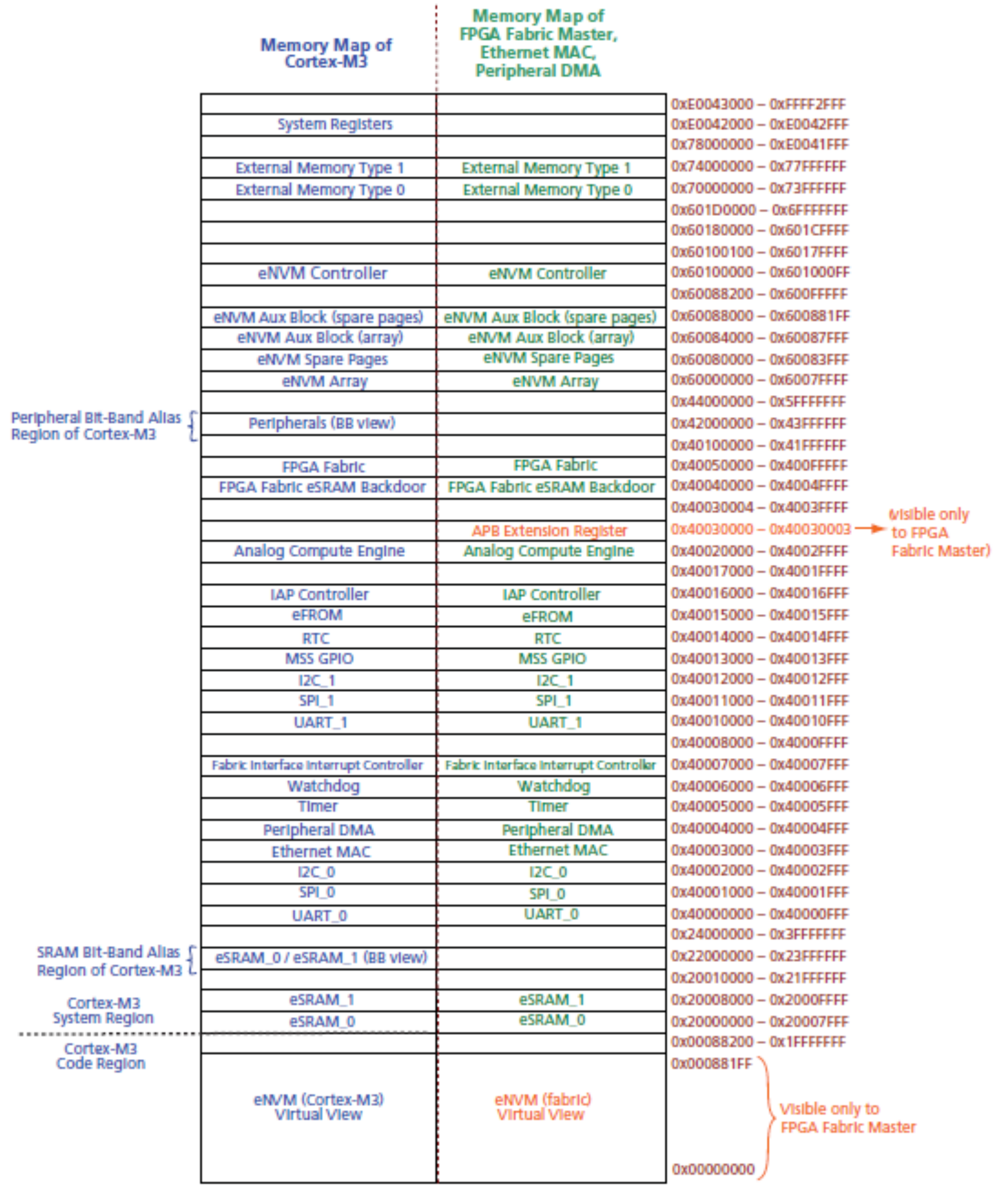
Memory-mapped I/O

- Instead of real memory, some addresses map to I/O devices instead

Example:

- Address 0x80000004 is a General Purpose I/O (GPIO) Pin
  - Writing a 1 to that address would turn it on
  - Writing a 0 to that address would turn it off
  - Reading at that address would return the value (1 or 0)

Smartfusion Memory Map



	Memory Map of Cortex-M3	Memory Map of FPGA Fabric Master, Ethernet MAC, Peripheral DMA		
			0xE0043000 – 0xFFFF2FFF	
	System Registers		0xE0042000 – 0xE0042FFF	
			0x78000000 – 0xE0041FFF	
	External Memory Type 1	External Memory Type 1	0x74000000 – 0x77FFFFFF	
	External Memory Type 0	External Memory Type 0	0x70000000 – 0x73FFFFFF	
			0x601D0000 – 0x6FFFFFFF	
			0x60180000 – 0x601CFFFF	
			0x60100100 – 0x6017FFFF	
	eNVM Controller	eNVM Controller	0x60100000 – 0x601000FF	
			0x60088200 – 0x600FFFFF	
	eNVM Aux Block (spare pages)	eNVM Aux Block (spare pages)	0x60088000 – 0x600881FF	
	eNVM Aux Block (array)	eNVM Aux Block (array)	0x60084000 – 0x60087FFF	
	eNVM Spare Pages	eNVM Spare Pages	0x60080000 – 0x60083FFF	
	eNVM Array	eNVM Array	0x60000000 – 0x6007FFFF	
			0x44000000 – 0x5FFFFFFF	
Peripheral Bit-Band Alias Region of Cortex-M3	Peripherals (BB view)		0x42000000 – 0x43FFFFFF	
			0x40100000 – 0x41FFFFFF	
	FPGA Fabric	FPGA Fabric	0x40050000 – 0x400FFFFF	
	FPGA Fabric eSRAM Backdoor	FPGA Fabric eSRAM Backdoor	0x40040000 – 0x4004FFFF	
			0x40030004 – 0x4003FFFF	
		APB Extension Register	0x40030000 – 0x40030003	(Visible only to FPGA Fabric Master)
	Analog Compute Engine	Analog Compute Engine	0x40020000 – 0x4002FFFF	
			0x40017000 – 0x4001FFFF	
	IAP Controller	IAP Controller	0x40016000 – 0x40016FFF	
	eFROM	eFROM	0x40015000 – 0x40015FFF	
	RTC	RTC	0x40014000 – 0x40014FFF	
	MSS GPIO	MSS GPIO	0x40013000 – 0x40013FFF	
	I2C_1	I2C_1	0x40012000 – 0x40012FFF	
	SPI_1	SPI_1	0x40011000 – 0x40011FFF	
	UART_1	UART_1	0x40010000 – 0x40010FFF	
			0x40008000 – 0x4000FFFF	
	Fabric Interface Interrupt Controller	Fabric Interface Interrupt Controller	0x40007000 – 0x40007FFF	
	Watchdog	Watchdog	0x40006000 – 0x40006FFF	
	Timer	Timer	0x40005000 – 0x40005FFF	
	Peripheral DMA	Peripheral DMA	0x40004000 – 0x40004FFF	
	Ethernet MAC	Ethernet MAC	0x40003000 – 0x40003FFF	
	I2C_0	I2C_0	0x40002000 – 0x40002FFF	
	SPI_0	SPI_0	0x40001000 – 0x40001FFF	
	UART_0	UART_0	0x40000000 – 0x40000FFF	
			0x24000000 – 0x3FFFFFFF	
SRAM Bit-Band Alias Region of Cortex-M3	eSRAM_0 / eSRAM_1 (BB view)		0x22000000 – 0x23FFFFFF	
			0x20010000 – 0x21FFFFFF	
Cortex-M3 System Region	eSRAM_1	eSRAM_1	0x20008000 – 0x2000FFFF	
	eSRAM_0	eSRAM_0	0x20000000 – 0x20007FFF	
Cortex-M3 Code Region			0x00088200 – 0x1FFFFFFF	
	eNVM (Cortex-M3) Virtual View	eNVM (fabric) Virtual View	0x000881FF … 0x00000000	Visible only to FPGA Fabric Master

Figure 2-4 • System Memory Map with 64 Kbytes of SRAM

Memory-mapped I/O

- Instead of real memory, some addresses map to I/O devices instead

- But how do you make this happen?
  - *MAGIC* isn't a bad guess, but not very helpful

Let's start by looking at how a memory bus works

Today…

Memory-Mapped I/O

Example Bus with Memory-Mapped I/O

Bus Architectures

AMBA APB

Bus terminology

- Any given transaction have an "*initiator*" and "*target*"

- Any device capable of being an initiator is said to be a "*bus master*"
  - In many cases there is only one bus master (*single master* vs. *multi-master*).

- A device that can only be a target is said to be a slave device.

Basic example

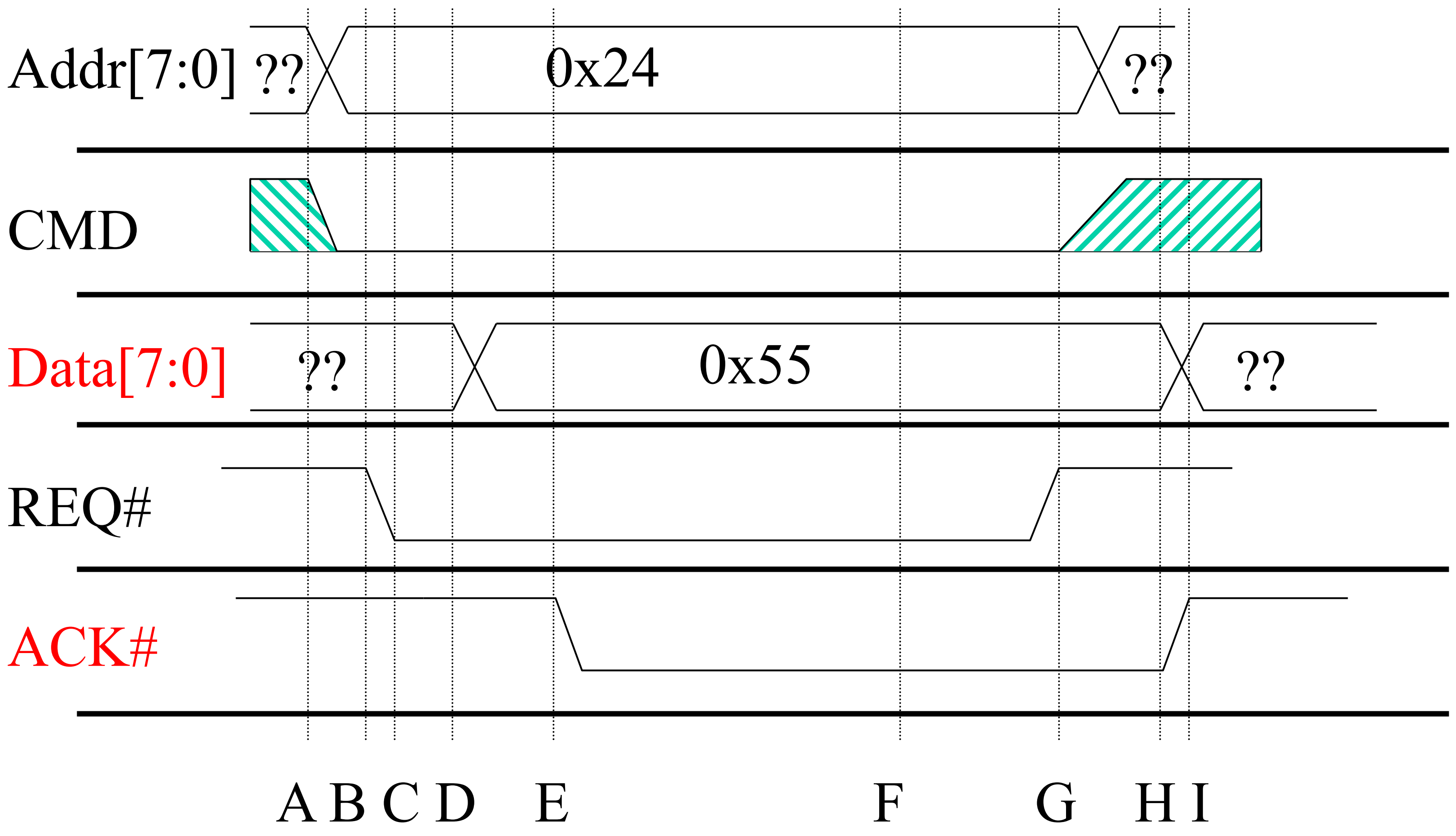
Let's demonstrate a hypothetical example bus

- Characteristics
  - Asynchronous (no clock)
  - One Initiator and One Target

- Signals
  - Addr[7:0], Data[7:0], CMD, REQ#, ACK#
    - CMD=0 is read, CMD=1 is write.
    - REQ# low means initiator is requesting something.
    - ACK# low means target has done its job.

Read transaction

Initiator wants to read location 0x24



A read transaction

- Say initiator wants to read location 0x24
  A. Initiator sets Addr=0x24, CMD=0
  B. Initiator *then* sets REQ# to low
  C. Target sees read request
  D. Target drives data onto data bus
  E. Target *then* sets ACK# to low
  F. Initiator grabs the data from the data bus
  G. Initiator sets REQ# to high, stops driving Addr and CMD
  H. Target stops driving data, sets ACK# to high terminating the transaction
  I. Bus is seen to be idle

A write transaction

- Say initiator wants to write 0xF4 location 0x31
  A. Initiator sets Addr=0x24, CMD=1, Data=0xF4
  B. Initiator *then* sets REQ# to low
  C. Target sees write request
  D. Target reads data from data bus
     (only needs to store in register, not write all the way to memory)
  E. Target *then* sets ACK# to low.
  F. Initiator sets REQ# to high, stops driving other lines
  G. Target sets ACK# to high, terminating the transaction
  H. Bus is seen to be idle.

Returning to memory-mapped I/O

Now that we have an example bus, how would memory-mapped I/O work on it?

Example peripherals

- 0x00000004: Push Button - Read-Only
  - Pushed -> 1
  - Not Pushed -> 0
- 0x00000005: LED Driver - Write-Only
  - On -> 1
  - Off -> 0

The push-button
(if Addr=0x04 write 0 or 1 depending on button)

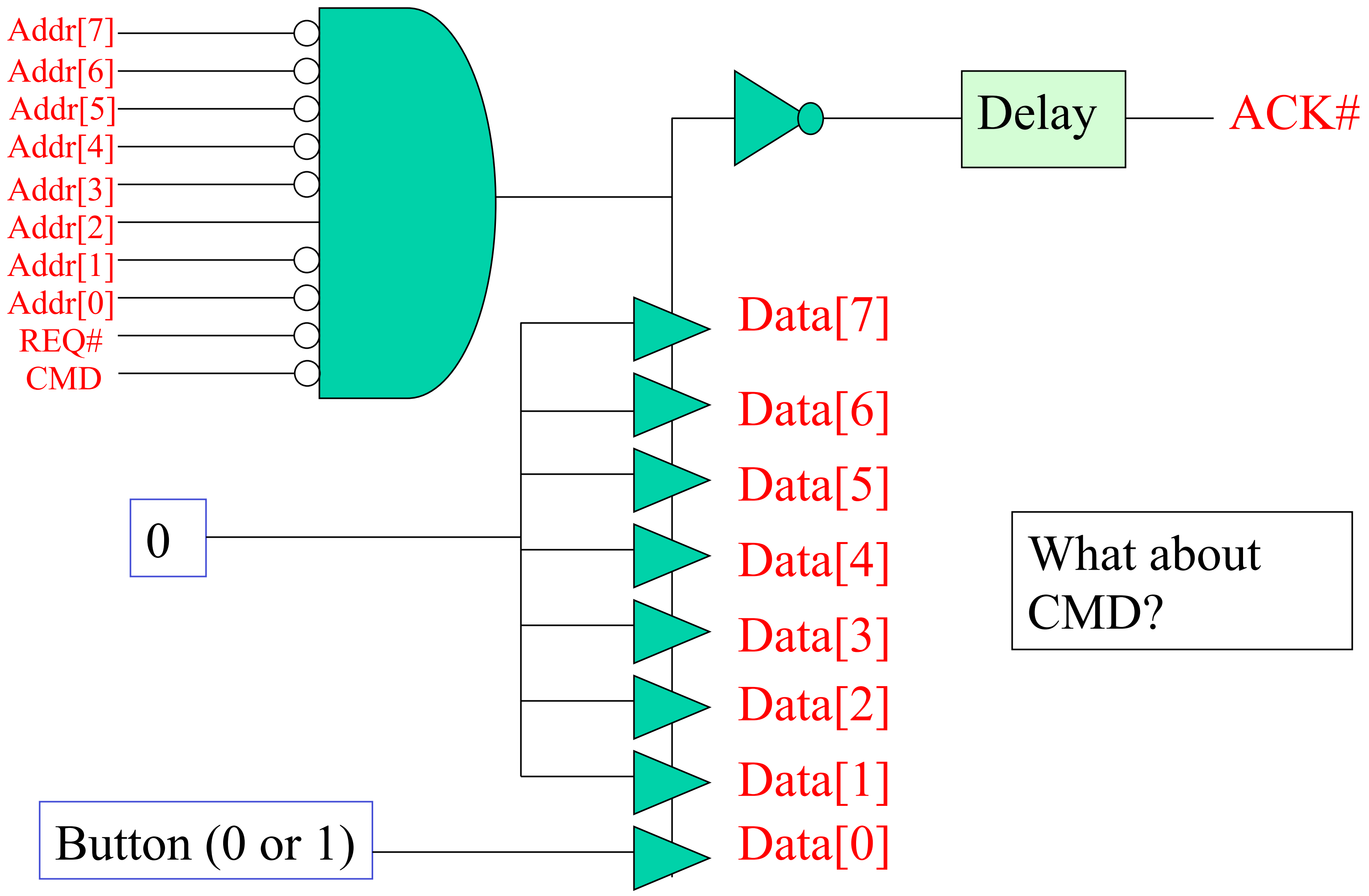
Addr[7]
Addr[6]
Addr[5]
Addr[4]
Addr[3]
Addr[2]
Addr[1]
Addr[0]
REQ#
CMD

ACK#

Data[7]
Data[6]
Data[5]
Data[4]
Data[3]
Data[2]
Data[1]
Data[0]

Button (0 or 1)

The push-button (if Addr=0x04 write 0 or 1 depending on button)

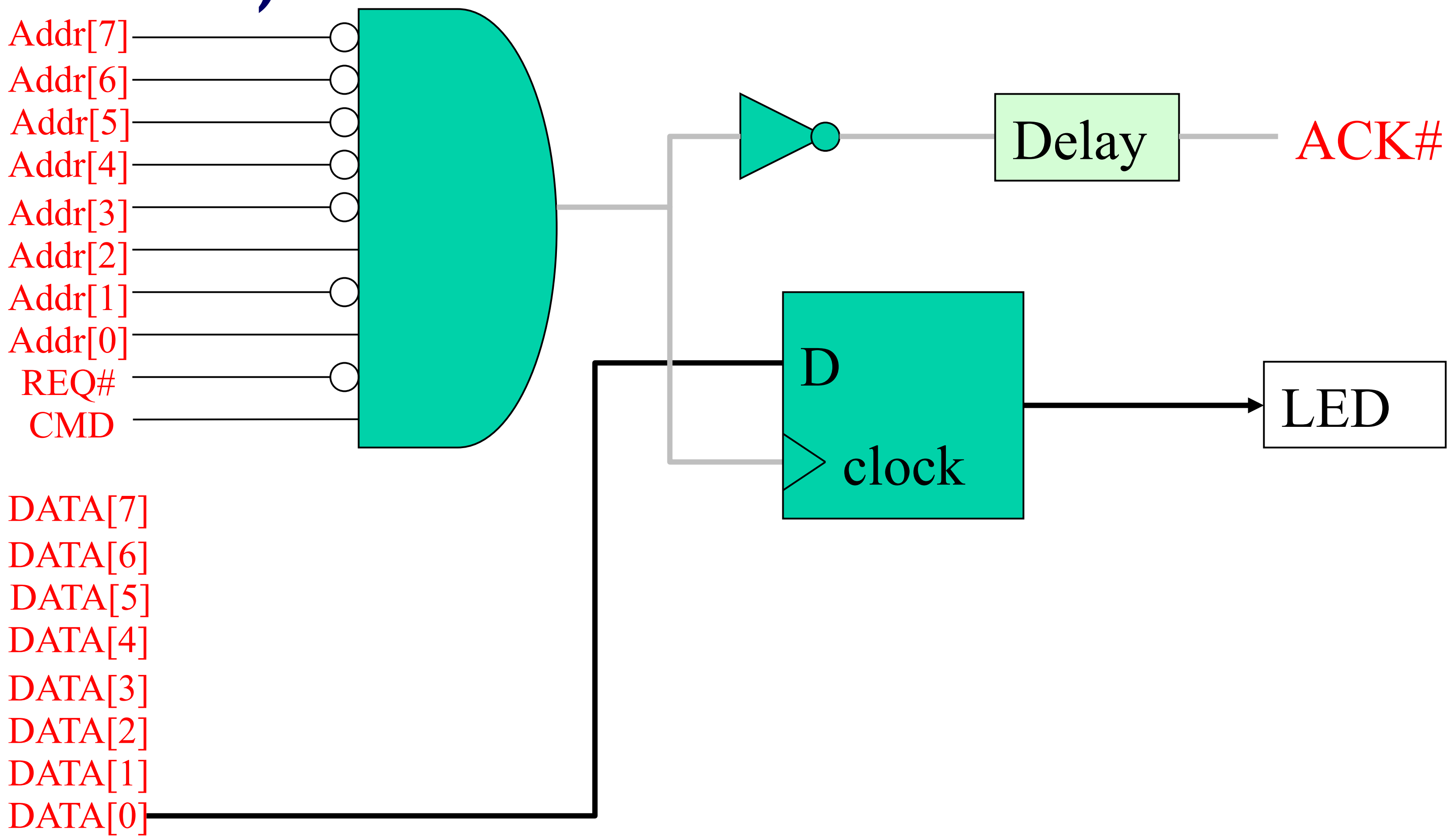


What about CMD?

The LED
(1 bit reg written by LSB of address 0x05)

Addr[7]
Addr[6]
Addr[5]
Addr[4]
Addr[3]
Addr[2]
Addr[1]
Addr[0]
REQ#
CMD
DATA[7]
DATA[6]
DATA[5]
DATA[4]
DATA[3]
DATA[2]
DATA[1]
DATA[0]
ACK#
LED

The LED (1 bit reg written by LSB of address 0x05)



Let’s write a simple assembly program
Light on if button is pressed.

Peripheral Details

- 0x00000004: Push Button - Read-Only
  - Pushed -> 1
  - Not Pushed -> 0
- 0x00000005: LED Driver - Write-Only
  - On -> 1
  - Off -> 0

Today…

Memory-Mapped I/O

Example Bus with Memory-Mapped I/O

Bus Architectures

AMBA APB

Driving shared wires

- It is commonly the case that some shared wires might have more than one potential device that needs to drive them.
  - For example there might be a shared data bus that is used by the targets and the initiator. We saw this in the simple bus.
  - In that case, we need a way to allow one device to control the wires while the others "stay out of the way"
    - Most common solutions are:
      - using tri-state drivers (so only one device is driving the bus at a time)
      - using open-collector connections (so if any device drives a 0 there is a 0 on the bus otherwise there is a 1)

Or just say no to shared wires.

- Another option is to not share wires that could be driven by more than one device...
  - This can be really expensive.
    - Each target device would need its own data bus.
    - That’s a LOT of wires!
  - Not doable when connecting chips on a PCB as you are paying for each pin.
  - Quite doable (though not pretty) inside of a chip.

Wire count

- Say you have a single-master bus with 5 other devices connected and a 32-bit data bus.
  - If we share the data bus using tri-state connections, each device has "only" 32-pins.
  - If each device that could drive data has it's own bus…
    - Each slave would need _____ pins for data
    - The master would need ______ pins for data
- Again, recall pins==$$$$$$.

What happens when this “instruction” executes?

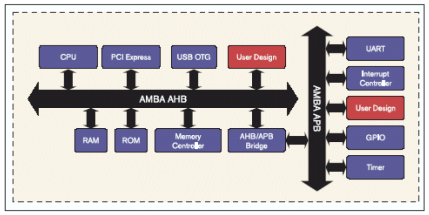
```
#include <stdio.h>
#include <inttypes.h>

#define REG_FOO 0x40000140

main () {
  uint32_t *reg = (uint32_t *)(REG_FOO);
  *reg += 3;

  printf(“0x%x\n”, *reg); // Prints out new value
}
```

“*reg += 3” is turned into a ld, add, str sequence

- Load instruction
  - A bus read operation commences
  - The CPU drives the address “reg” onto the address bus
  - The CPU indicated a read operation is in process (e.g. R/W#)
  - Some “handshaking” occurs
  - The target drives the contents of “reg” onto the data lines
  - The contents of “reg” is loaded into a CPU register (e.g. r0)
- Add instruction
  - An immediate add (e.g. add r0, #3) adds three to this value
- Store instruction
  - A bus write operation commences
  - The CPU drives the address “reg” onto the address bus
  - The CPU indicated a write operation is in process (e.g. R/W#)
  - Some “handshaking” occurs
  - The CPU drives the contents of “r0” onto the data lines
  - The target stores the data value into address “reg”

Details of the bus "handshaking" depend on the particular memory/peripherals involved

- SoC memory/peripherals
  - AMBA AHB/APB



- NAND Flash
  - Open NAND Flash Interface (ONFI)

- DDR SDRAM
  - JEDEC JESD79, JESD79-2F, etc.

Why use a standardized bus?

- Downsides
  - Have to follow the specification
  - Probably has actions that are unnecessary

- Upside
  - Generic systems
  - Allows modules to be reused on different systems

Today…

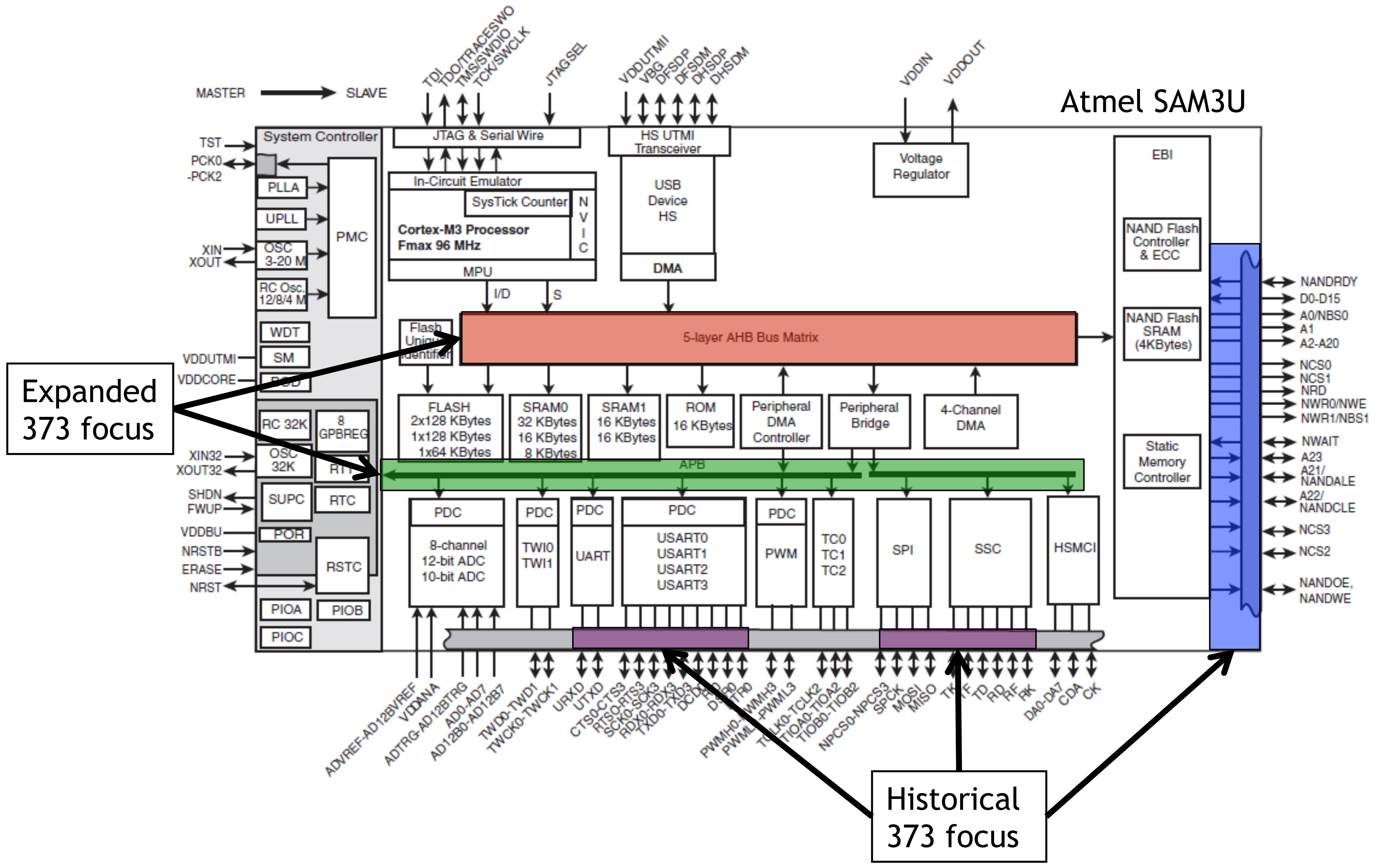
Memory-Mapped I/O

Example Bus with Memory-Mapped I/O

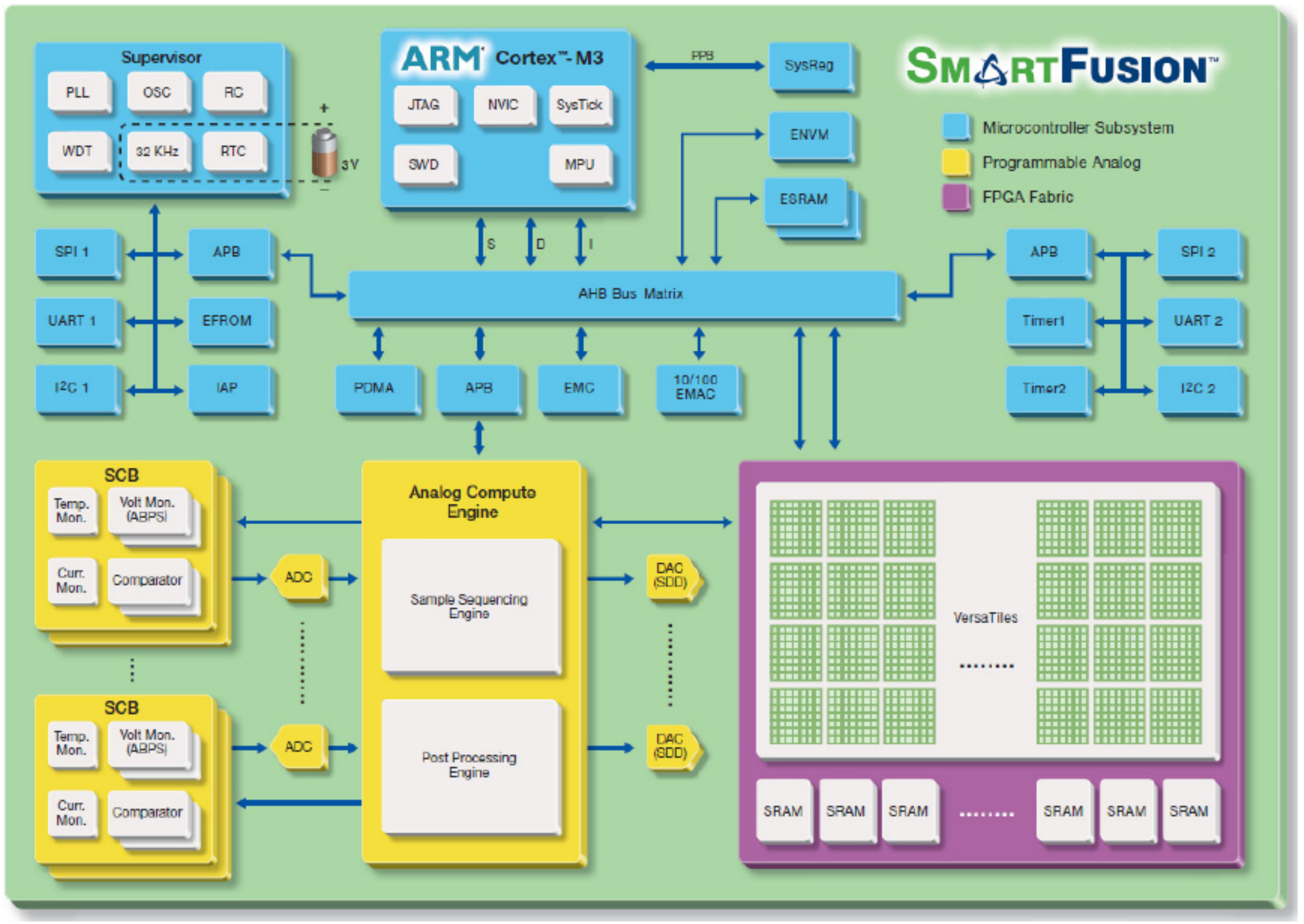
Bus Architectures

AMBA APB

Modern embedded systems have multiple busses



Actel SmartFusion system/bus architecture



Questions?

Comments?

Discussion?