



Microprocessors and Assembly language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering
Amirkabir University of Technology

provided by: Ali Moradzade 9831058



Microprocessors and Assembly language

Spring 2020

Hamed Farbeh
farbeh@aut.ac.ir

Department of Computer Engineering
Amirkabir University of Technology

Grading

- Final Exam: 40%
- Midterm Exam: 30%
- Homework: 20%
- Attendance: 15%

AND

- Lab

Course Outline

- **Microprocessors and microcontrollers**
 - How they **work**

AND

- How **to work** with them

References

Main

- *Arm Assembly Language Programming and Architecture*, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013
- Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series, Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15
- ARMv7-M Architecture Reference Manual

Complementary material

- *ARM Assembly Language: Fundamentals and Techniques*, 2nd Ed., William Hohl, Christopher Hinds, CRC Press, 2014.
- *Modern Assembly Language Programming with the ARM Processor*, Larry D. Pyeatt, Elsevier, 2016.
- *Embedded Systems:Introduction to ARM® CORTEX-M Microcontrollers*, Vol. 1, 5th Ed., Jonathan W. Valvano, 2014.

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

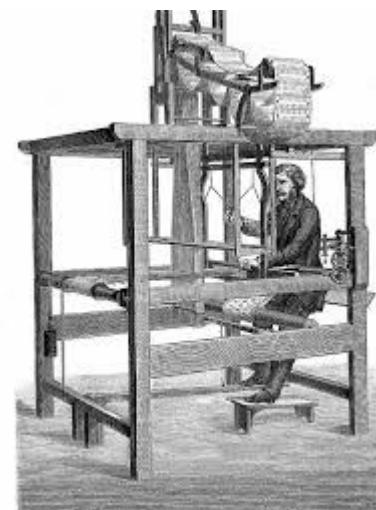
- Computer Organization & Design, The Hardware/Software Interface, 3rd Edition, by D. Patterson and J. Hennessy, Morgan Kaufmann publishing, 2005.
- “Computer Organization & Design” handouts, by Prof. Kumar, UIUC, Fall 2007.
- “Computer Organization I” handouts, FSU, Fall 2007.
- “Machine Structures” handouts, by Prof. D. Garcia, UCB, Spring 2010.
- Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013
- “Computer Organization & Language” handouts, Dept. of CE, SUT, by Prof. H. Asadi, 2016.

Computer History

- **Mechanical Computers**
- **Electro-Mechanical Computers**
- **Electronic Computers**

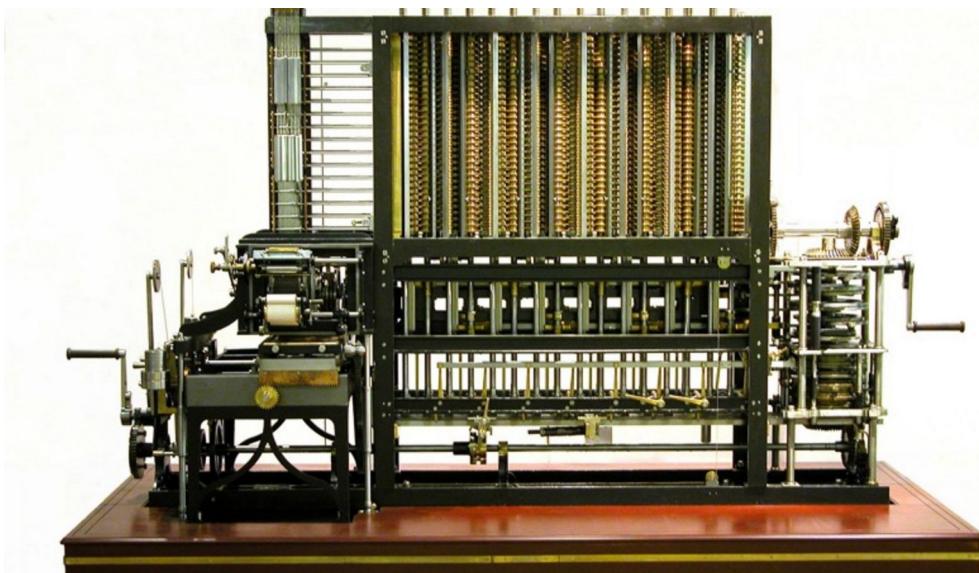
Computer History

- **Mechanical Computers**
 - **Abacus**
 - **Mechanical calculators**
 - **Pascal's calculator (1642)**
 - **Curta (1930s)**
 - **Punched-card data processing (1725, 1800s, 1900s)**



Computer History

- **Mechanical Computers**
 - Difference engine: Charles Babbage (1791-1871)
 - First general-purpose computing device
 - Analytical engine



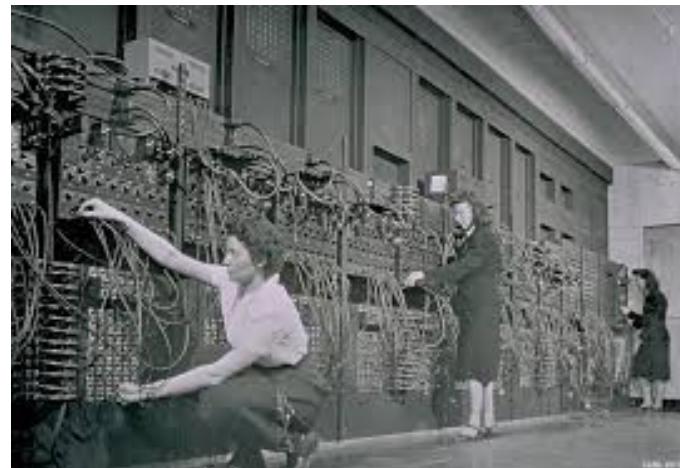
Computer History

- **Electro-Mechanical Computers**
 - Switches, relays
 - Early 1900s
 - Zuse's Z2, Z3, Z4 (world's first commercial digital computer, mechanical memory)



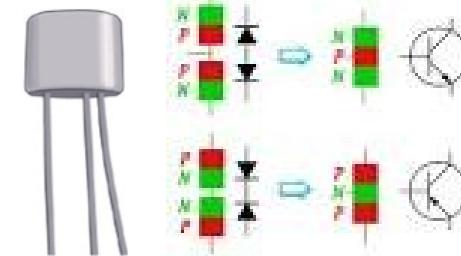
Computer History

- **Electronic Computers**
 - 1st generation
 - vacuum tubes (1945-1955)
 - ENIAC, 1946



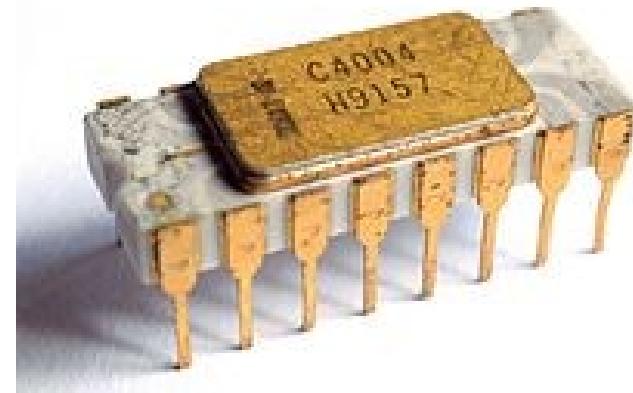
Computer History

- **Electronic Computers**
 - 2nd generation
 - BJT transistors (1955-1965)
 - CDC 6600 (1964-1969)
 - world's fastest computer from 1964 to 1969
 - \$2,370,000
 - 30kW, 6tons
 - 10MHz, 2 MIPS
 - Up to 982 kilobytes



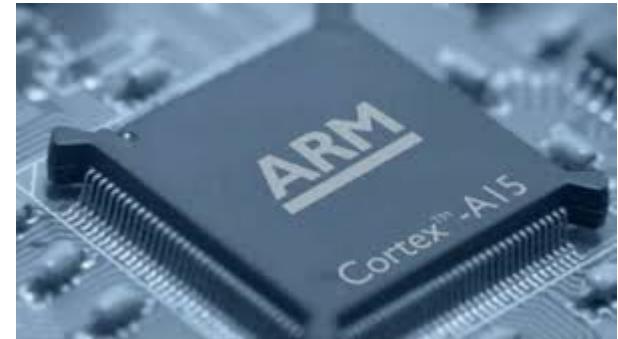
Computer History

- **Electronic Computers**
 - 3rd generation
 - integrated circuits (1965-1974)
 - Intel 4004 (1971)
 - The first commercially available microprocessor
 - 10 μm process
 - 92,000 instructions per second
 - 1 MHz clock rate



Computer History

- **Electronic Computers**
 - 4th generation: VLSI
 - 1974-1989
 - 5th generation: ULSI
 - 1990-present

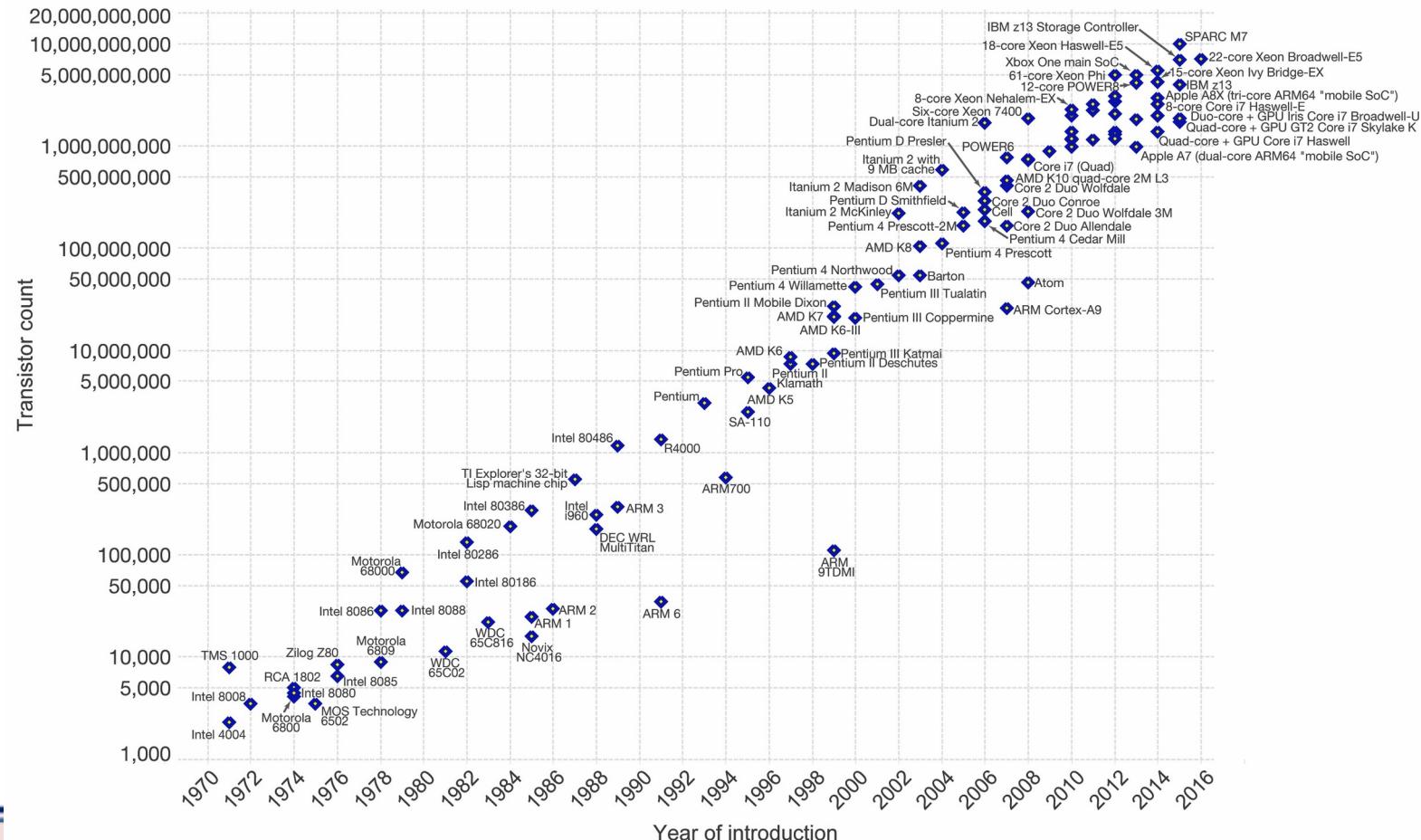


Computer History

| Year | Technology Used in Computers | Relative Performance per unit cost |
|-------|--------------------------------------|------------------------------------|
| ~1950 | Vacuum tube | 1 |
| 1960s | Transistor | 35 |
| 1975 | Integrated circuit | 900 |
| 1995 | Very large scale integrated circuit | 2,400,000 |
| 2005 | Ultra large scale integrated circuit | 6,200,000,000 |

Computer History

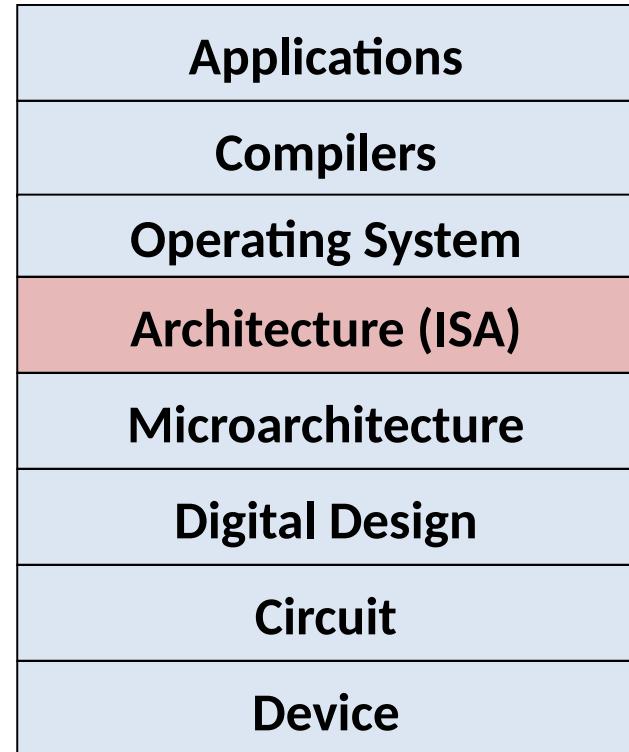
- What happened from 1970?
 - Transistor count doubles every 24-month



Instruction Set Architecture (ISA)

Instruction Set Architecture (ISA)

- A set of instructions used by a machine to run programs
- Interface between hardware & software
- Computer language vocabulary
- Provides an abstraction of hardware implementation
 - Hardware implementation decides what and how instructions are implemented
- ISA specifies
 - Instructions, Registers, Memory access, Input/output



Instruction Set Architecture (ISA)

- How many instructions needed?
- What functionalities required?
 - Load/store
 - Control
 - Arithmetic/logic

Instruction Set Architecture (ISA)

- Key ISA decisions
 - Instruction length
 - Number, length, and types of registers
 - Where operands reside
 - How to access memory
 - Instruction format
 - Operand format
 - How many and how big

Instruction Set Architecture (ISA)

- **Instruction length**

Variable:

x86 – Instructions vary from 1 to 17 Bytes long

VAX – from 1 to 54 Bytes

Fixed:

MIPS, PowerPC, ARM:

all instruction are 4 Bytes long

ISA: Instruction length

- **Variable-length instructions**
 - Require multi-step fetch and decode
 - Allow for a more flexible and compact instruction set
 - CISC processors like x86 & VAX
 - **Complex Instruction Set Computing**
- **Fixed-length instructions**
 - Allow easy fetch and decode
 - Simplify pipelining and parallelism
 - RISC processors like MIPS & PowerPC
 - **Reduced Instruction Set Computing**

ISA: RISC vs. CISC

- **Early Trend**
 - Adding more instructions to next generation CPUs to do more complicated operations
 - VAX arch had an instruction to multiply polynomials!
- **CISC Philosophy**
 - Limited main memory + immature compilers
 - More dense instructions, highly encoded, variable length instructions
 - Data loading as well as calculation

ISA: RISC vs. CISC

- **RISC Philosophy**
 - Keep ISA small and simple
 - Makes it easier to build faster hardware
 - Let SW do complicated operations by composing simpler instructions
- **Note on “Reduced” in RISC Phrase**
 - Amount of work any single instruction accomplishes reduced
 - Single ALU operation, single memory access, ...

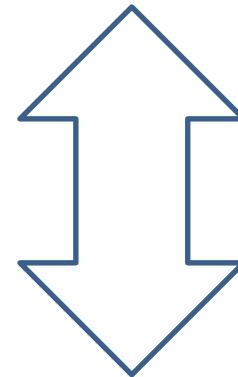
ISA: RISC vs. CISC

- **CISC Problems**
 - Performance tuning challenging
 - Complex/high-level instructions rarely used
 - Slower clock rates
 - Longer time-to-market
 - Due to prolonged design time
- **CISC Features**
 - Ease compiler implementation
 - HW supports all kind of addressing modes

ISA: RISC vs. CISC

- **RISC Features**
 - Low complexity
 - Less error-prone HW implementation
 - Implementation advantages
 - Less transistors
 - Extra space: more registers, cache
 - Marketing
 - Reduced design time

RISC vs. CISC



Hangul vs. Hiragana

ISA: How Many Registers?

- **Advantages of a small number of registers**
 - Requires fewer bits to specify which one
 - Less hardware
 - Faster access
 - Shorter wires
 - Faster index decoding (fewer logic levels)
 - Faster context switch (your OS course!)
 - When all registers need saving

ISA: How Many Registers?

- **Advantages of a larger number of registers**
 - Fewer loads and stores needed
 - Less data transfer between CPU & memory
 - Easier to do several operations at once

ISA: Where Operands Reside?

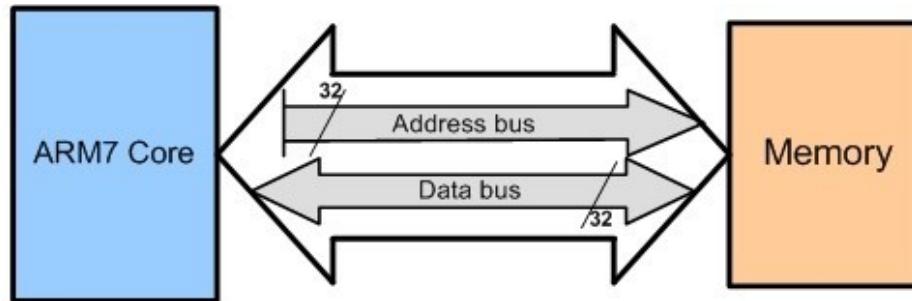
- **Stack Machine**
- **Accumulator Machine**
- **Register-Memory Machine**
- **Load-Store Machine**
 - aka Register-Register Machine

ISA: Where Operands Reside?

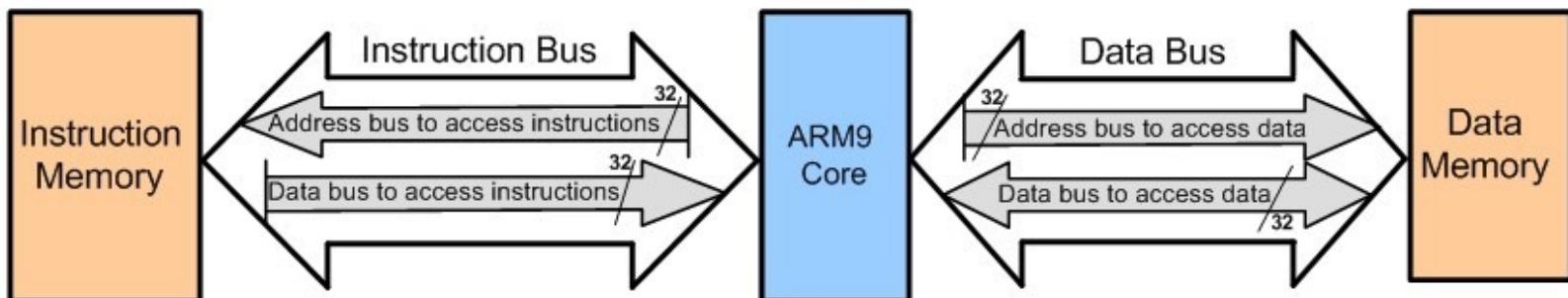
Code sequence for $C = A + B$

| <u>Stack</u> | <u>Accumulator</u> | <u>Register-Memory</u> | <u>Load-Store</u> |
|--------------|--------------------|----------------------------------|-------------------|
| Push A | Load A | Add C, A, B | Load R1,A |
| Push B | Add B | کامپیوتر های ما از این نوع هستند | Load R2,B |
| Add | Store C | | Add R3,R1,R2 |
| Pop C | | | Store C,R3 |

Harvard vs. Von Neumann Arch.



(a) Von Neumann

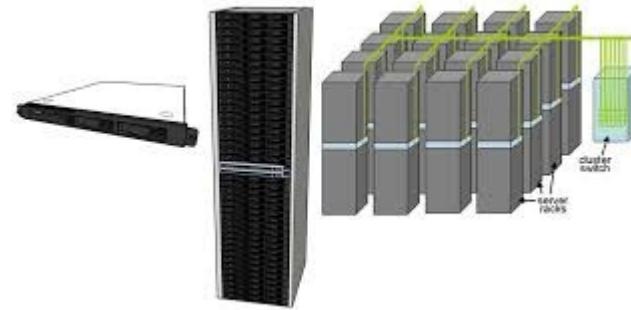


(b) Harvard

Types of Computer Systems

Types of Computer Systems

- **Desktops**
 - General purpose
 - Market dominated by x86 processors
 - Various software
 - For use by individuals
- **Servers and Warehouse-Scale Computers (WSCs)**
 - Thousands of cores and disks
 - Market dominated by x86-compatible server chips
 - Dedicated apps, plus cloud hosting of virtual machines
 - Increasing use of GPUs, FPGAs, and custom hardware to accelerate workloads



Types of Computer Systems

- Mobile (smartphone/tablet)
 - >1 billion sold/year
 - Market dominated by **ARM-ISA-compatible general-purpose processor in system-on-a-chip (SoC)**
 - Plus sea of custom accelerators (radio, image, video, graphics, audio, motion, location, security, etc.)
- Anything Else?



Types of Computer Systems

- **Embedded systems**



Design Criteria

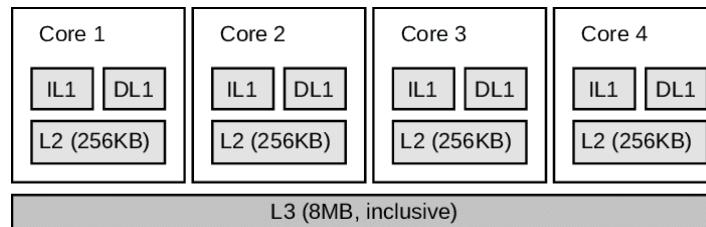
- **Performance**
- **Power consumption**
- **Energy consumption**
- **Area**
- **Weight**
- **Design cost**
- **Implementation cost**
- **Time to market**
- **Maintainability**
- **Debugging**
- **Flexibility**
- **Scalability**
- ...

Options for Building Embedded Systems

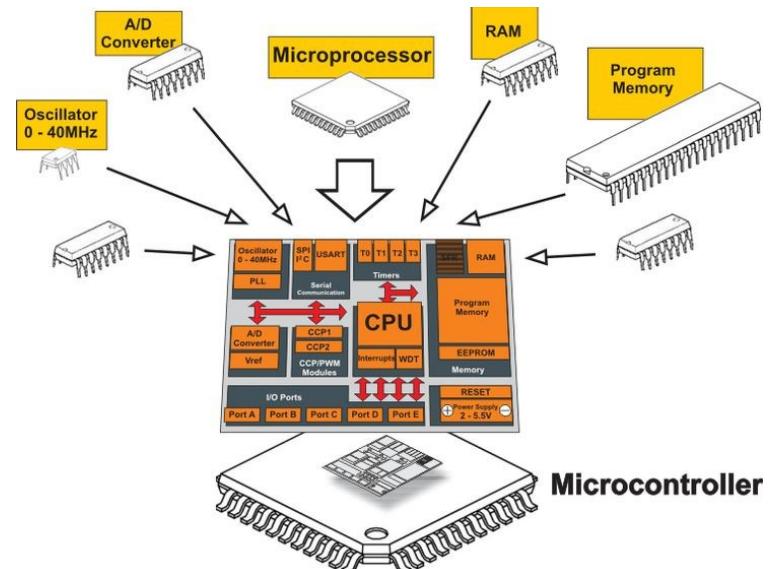
- Dedicated hardware
 - ASIC (Application-specific integrated circuit)
 - Programmable logic (FPGA, PLD)
- Software running on generic hardware
 - Microprocessor + memory + peripherals
 - **Microcontroller** (internal memory & peripherals)
- A combination of all options (Complex systems)

Microprocessor vs. Microcontroller

- **Microprocessor**
 - An IC that has only CPUs inside it
 - No RAM, ROM, or other peripheral



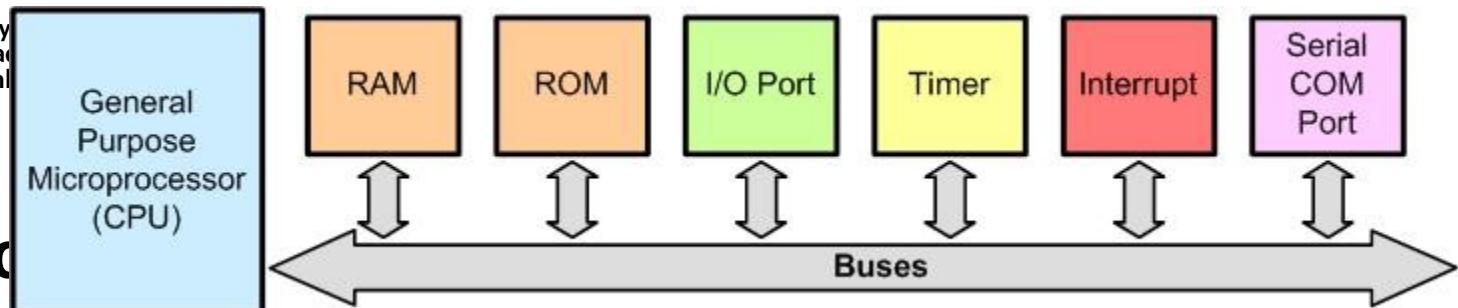
- **Microcontroller (MCU)**



Microprocessor vs. Microcontroller

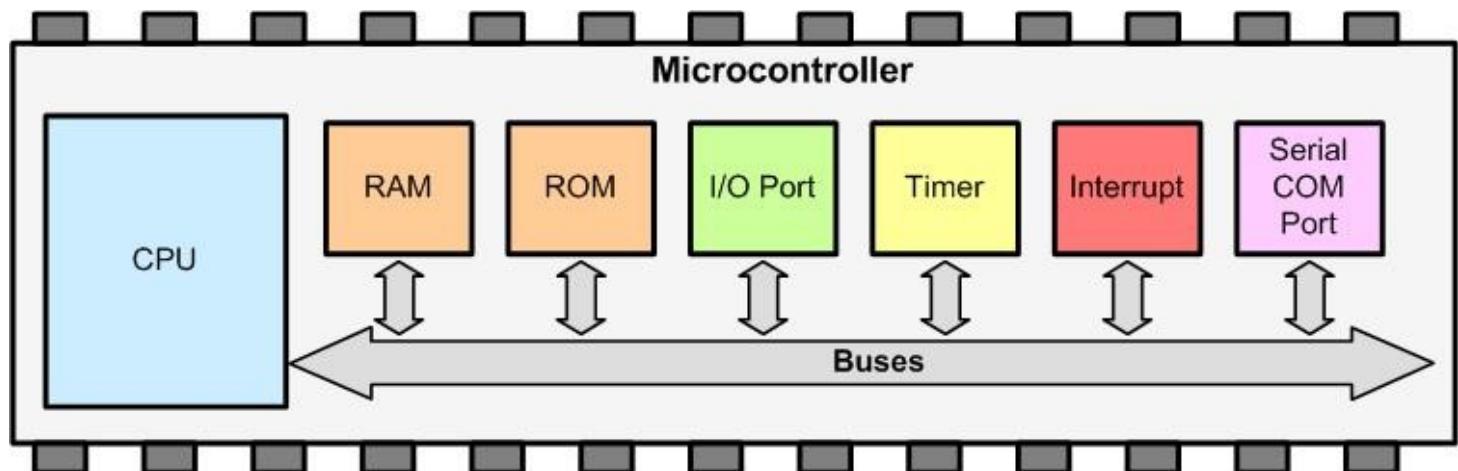
- **Microprocessor**

ROM: read only memory
PROM: programmable read only memory
EROM: erasable read only memory
EEPROM: yadam nist



- **Microcontroller**

Here by rom we mean eeprom



Microprocessor vs. Microcontroller

| Microprocessor | Microcontroller |
|---|---|
| Heart of computer systems | Heart of embedded systems |
| It is just a processor. Memory and I/O components are connected externally | Has processor along with internal memory and I/O components |
| Since memory and I/O have to be connected externally, the circuit becomes larger | Since memory and I/O have to be connected internally, the circuit is small |
| Cannot be used in compact system | It is efficient for compact systems |
| The entire energy consumption is high due to external components | Suitable for battery-operated systems |
| Slow memory access | Fast memory access |
| Is based on Von Neumann architecture (program and data are stored in the same memory module) | Is based on Harvard architecture (program and data memory are separate) |



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 2

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.
- Design of Microprocessor-Based Systems (aka Embedded Systems Design and Implementation), Prabal Dutta, University of Michigan.

Microcontrollers

Bell's Law: A new computer class every decade

“Roughly every decade a new, lower priced computer class forms based on a new programming platform, network, and interface resulting in new usage and the establishment of a new industry.”

- Gordon Bell [1972,2008]

BY GORDON BELL

BELL'S LAW FOR THE BIRTH AND DEATH OF COMPUTER CLASSES

A theory of the computer's evolution.

In the early 1950s, a person could walk inside a computer and by 2010 a single computer (or "cluster") with millions of processors will have expanded to the size of a building. More importantly, computers are beginning to "walk" inside of us. These ends of the computing spectrum illustrate the vast dynamic range in computing power, size, cost, and other factors for early 21st century computer classes.

A computer class is a set of computers in a particular price range with unique or similar programming environments (such as Linux, OS/360, Palm, Symbian, Windows) that support a variety of applications that communicate with people and/or other systems. A new computer class forms and approximately doubles each decade, establishing a new industry. A class may be the consequence and combination of a new platform with a new programming environment, a new network, and new interface with people and/or other information processing systems.

86 January 2008/Vol. 51, No. 1 COMMUNICATIONS OF THE ACM

Microcontrollers: A Brief History

- **1980s and 1990s**
 - Intel: **x86** (8088/86, 80286, 80386, 80486, and Pentium)
 - Motorola (Freescale=>NXP): **68xxx** (68000, 68010, 68020)
 - **High-end embedded systems (Cisco routers)**
 - Intel's 32-bit x86
 - Motorola's 32-bit 68xxx
 - **Low-end embedded systems**
 - Intel's 8-bit 8051
 - Motorola's 8-bit 68HC1

پردازنده با سرعت بالا

پردازنده با سرعت پایین

Microcontrollers: A Brief History

- Major players in the 8-bit market
 - PIC from Microchip: 1976
 - AVR from Atmel (Acquired by Microchip 2016): 1996
 - Leaders in terms of volume (2013): PIC and AVR
 - 2019: Microchip and Renesas
- Late 1990s
 - ARM challenged Intel and Motorola in 32-bit market

Microcontrollers: A Brief History

- RISC features to enhance the performance, **BUT**
 - Intel & Motorola: **compatibility** with legacy software
 - Could not start over
 - Massive amounts of gates to keep up the performance
 - Increased **power consumption** of the x86
 - Unacceptable for **battery-powered** embedded products
 - ARM: A **clean** RISC architecture
 - Leading microcontroller in the 32-bit market

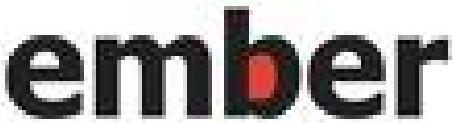
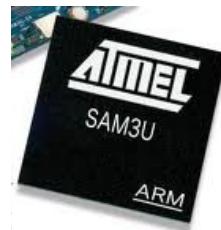
Currently Available Microcontrollers

- **32-bit**
 - ARM, AVR32 (Atmel), ColdFire (Freescale), MIPS32, PIC32 (Microchip), PowerPC, TriCore (Infineon), SuperH
 - **16-bit**
 - MSP430 (TI), HCS12 (Freescale), PIC24 (Microchip), dsPIC (Microchip)
 - **8-bit**
 - 8051, AVR, HCS08 (Freescale), PIC16, PIC18
- فراوانی ۱۶ و ۸ بیتی بسیار بیشتر از ۳۲ بیتی می باشد،
برای مثال شاید در کامپیوتر ما یک ۶۴ یا ۳۲ بیتی باشد ولی ده تا ۸ بیتی باشد (برای مثال برای کیبورد، ماوس، و ...)

Why study the ARM architecture?

The logo consists of the lowercase letters "arm" in a bold, dark blue sans-serif font. The letters are slightly slanted to the right.

Lots of manufacturers ship ARM products



Giant Partners



Google



ARM is the Big Player

- ARM has a huge market share
- 2011: ARM has chips in 90% of the world's mobile handsets
- 2010: ARM has chips in 95% of the smartphone market
 - And, 10% of the notebook market
 - Expected to hit 40% of the notebook market in 2015 (?)
- Heavy use in general embedded systems
 - Cheap to use
 - Flexible

A Brief History of the ARM

- Came out of a company called **Acorn Computers**
 - In United Kingdom in the 1980s
 - Prof. Steve Furber of Manchester Univ. and Sophie Wilson
 - Define the ARM architecture and instructions
 - First ARM chip: 1985
 - Acorn RISC Machine (ARM)
 - Produced by VLSI Technology Corp.
 - World's First commercial RISC processor
 - Unable to compete with x86 (8088, 80286, 80386, ...) PCs from IBM and other personal computer makers

A Brief History of the ARM

- Acorn was forced to push ARM into emb. market
- Apple Corp.
 - Using the ARM chip for the PDA (personal digital assistants) products
- ARM as a new company
 - ARM (Advanced RISC Machine)
 - A big gamble
 - Selling the rights to this new CPU to other silicon manufacturers and design houses



ARM Company Milestones

- 1982: Acorn produced a computer for BBC
- 1983: Acorn began designing its own uP
- 1985: ARMv1 developed (2500 trans., 4MHz)
- 1990: Advanced RISC Machines (ARM) spins out
- 1992: GEC Plessey and Sharp licensed ARM
- 1993: Texas Instrument licensed ARM
- 1995: ARM lunched Software Development Toolkit
- 1996: ARM and MS worked together: Windows SE on ARM
- 1997: Hyundai, Lucent, Philips, Rockwell, and Sony licensed ARM
- 1998: HP, IBM, Matsushita, Seiko Epson, and Qualcomm licensed ARM
- 1999: LSI Logic, STMicroelectronics, and Fujitsu licensed ARM

ARM Company Milestones

- 2000: Agilent, Altera, Micronas, Mitsubishi, Motorola, Sanyo, Triscend and ZTEIC licensed ARM
- 2001: Global UniChip, Samsung and Zeevo licensed ARM
 - Share of the 32-bit embedded RISC microprocessor market: 76.8%
- 2002: Seagate, Broadcom, Philips, Matsushita, Micrel, eSilicon, Chip Express and ITRI licensed ARM
- 2004: The **ARM Cortex** family of processors was announced (Cortex-M3)
- 2005: ARM acquired **Keil Software**
- 2009: ARM launches its smallest, lowest power, most energy efficient processor: **Cortex-M0**
-

Making money from **selling IP (intellectual property)
has made ARM one of the most widely used CPU
architectures in the world**

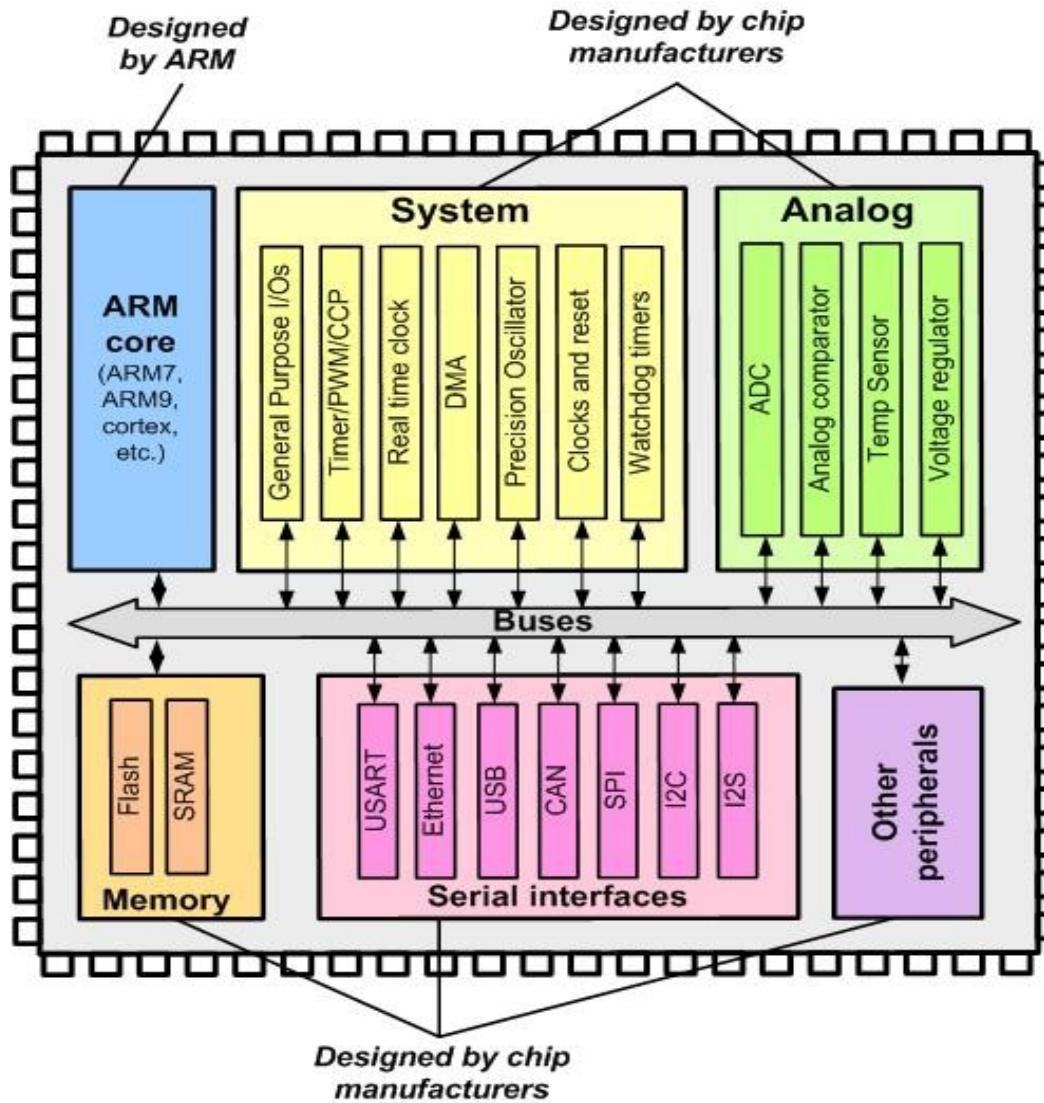
ARM Family Variations

| CORE | Application Cortex Processors | Cortex-A15 |
|------------------------|-------------------------------|------------|
| | Embedded Cortex Processors | Cortex-A9 |
| Classic ARM Processors | | Cortex-A8 |
| ARM11MP | | Cortex-A5 |
| Cortex-M7 | | |
| ARM926 | | SC300 |
| ARM176JZ | | SC00 |
| Cortex-R7 | | |
| ASC100 | | Cortex-M4 |
| ARM968 | | Cortex-M1 |
| ARM1136J | | |
| Cortex-R5 | | |
| ARM7TDMI | | Cortex-M3 |
| ARM946 | | Cortex-M0 |
| ARM1156T2 | | |
| Cortex-R4 | | |
| Cortex-M | | |
| Family | | Cortex-A/R |
| ARM7TDMI | | Cortex-M |
| ARM9E | | Cortex-M |
| ARM11 | | |
| Architecture Version | | |
| ARMv4T | | ARMv8M |
| ARMv5TJ | | |
| ARMv6 | | |
| ARMv7A/R | | |
| ARMv7M/ME | | |

ARM: One CPU, Many Peripherals

- **ARM Defines and holds the copyright to**
 - Details of arch., reg., intr. set, memory map, and timing
 - For the ARM CPU
- **Design houses and semiconductor manufacturers**
 - Add their own peripherals
 - I/O ports, serial port UART, timers, ADC, SPI, DAC, I2C, ...
- **Same instructions and architecture, different peripherals**
 - Incompatibility across vendors
- **Keil IDE:** provides peripheral libraries for chips from various vendors

ARM Simplified Block Diagram





Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 3

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series,
Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15

SAM3X8E Specifications

Configuration Summary

| Feature | SAM3X8E | SAM3X8C | SAM3X4E | SAM3X4C | SAM3A8C | SAM3A4C |
|---------------------------------|---|---|---|---|---|---|
| Flash | 2 x 256 Kbytes | 2 x 256 Kbytes | 2 x 128 Kbytes | 2 x 128 Kbytes | 2 x 256 Kbytes | 2 x 128 Kbytes |
| SRAM | 64 + 32 Kbytes | 64 + 32 Kbytes | 32 + 32 Kbytes | 32 + 32 Kbytes | 64 + 32 Kbytes | 32 + 32 Kbytes |
| NAND Flash Controller (NFC) | Yes | – | Yes | – | – | – |
| NFC SRAM ⁽¹⁾ | 4 Kbytes | – | 4 Kbytes | – | – | – |
| Package | LQFP144 LFBGA144 | LQFP100 TFBGA100 | LQFP144 LFBGA144 | LQFP100 TFBGA100 | LQFP100 TFBGA100 | LQFP100 TFBGA100 |
| Number of PIOs | 103 | 63 | 103 | 63 | 63 | 63 |
| SHDN Pin | Yes | No | Yes | No | No | No |
| EMAC | MII/RMII | RMII | MII/RMII | RMII | – | – |
| External Bus Interface | 16-bit data, 8 chip selects, 23-bit address | – | 16-bit data, 8 chip selects, 23-bit address | – | – | – |
| SDRAM Controller ⁽⁶⁾ | – | – | – | – | – | – |
| Central DMA | 6 | 4 | 6 | 4 | 4 | 4 |
| 12-bit ADC | 16 ch. ⁽²⁾ |
| 12-bit DAC | 2 ch. |
| 32-bit Timer | 9 ch. ⁽³⁾ | 9 ch. ⁽⁴⁾ | 9 ch. ⁽³⁾ | 9 ch. ⁽⁴⁾ | 9 ch. ⁽³⁾ | 9 ch. ⁽³⁾ |
| PDC Channels | 17 | 15 | 17 | 15 | 15 | 15 |
| USART/UART | 3/2 ⁽⁵⁾ | 3/1 | 3/2 ⁽⁵⁾ | 3/1 | 3/1 | 3/1 |
| SPI | 1 SPI controller, 4 chip selects + 3 USART with SPI mode | 1 SPI controller, 4 chip selects + 3 USART with SPI mode | 1 SPI controller, 4 chip selects + 3 USART with SPI mode | 1 SPI controller, 4 chip selects + 3 USART with SPI mode | 1 SPI controller, 4 chip selects + 3 USART with SPI mode | 1 SPI controller, 4 chip selects + 3 USART with SPI mode |
| HSMCI | 1 slot, 8 bits | 1 slot, 4 bits | 1 slot, 8 bits | 1 slot, 4 bits | 1 slot, 4 bits | 1 slot, 4 bits |

SAM3X8E Operating Modes

- **Active Mode**
 - Normal running mode with the core clock running from the fast RC oscillator
- **Low Power Modes**
 - Backup Mode
 - Wait Mode
 - Sleep Mode

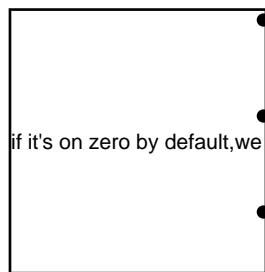
SAM3X8E Operating Modes

- **Low Power Modes**

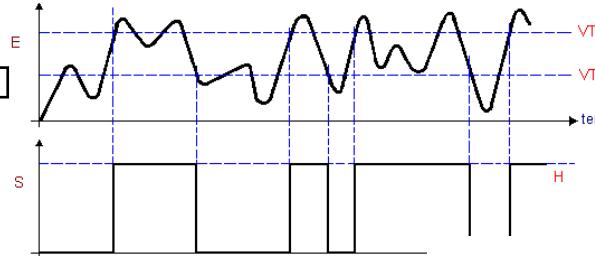
- **Backup Mode** The most saving occurs here
 - Achieve the lowest power consumption possible in a system
 - For tasks not requiring fast startup time (< 0.5 ms)
 - The core is off
- **Wait Mode**
 - Very low power while maintaining the whole device in a powered state for a startup time of less than 10 µs
 - The clocks of the core, peripherals and memories are stopped
 - The core, peripherals, and memories power supplies are still powered
- **Sleep Mode**
 - Only the core clock is stopped

SAM3X8E Input/Output Lines

- General Purpose I/O Lines (GPIO)
 - Managed by Parallel Input/Output (PIO) Controllers
 - I/O Modes



Pull-up example: telephone line



path for going to zero is different

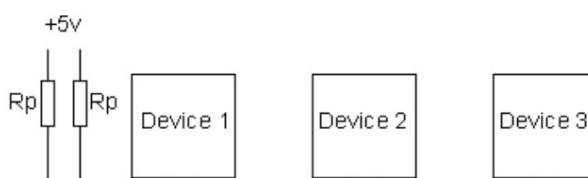
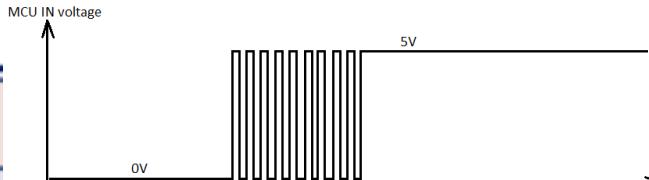
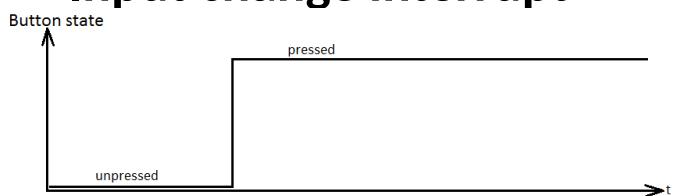
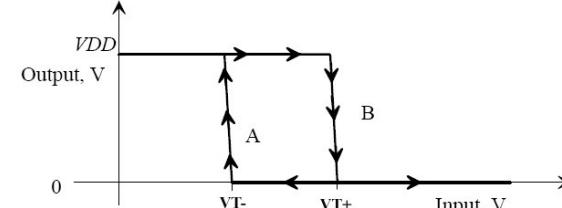
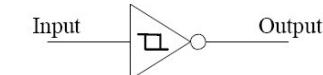
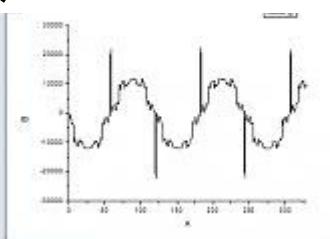
Input Schmitt triggers

Multi-drive (open-drain)

- Glitch filters

- Debouncing

- Input change interrupt



SAM3X8E Input/Output Lines

- System I/O Lines
 - Shared with PIO lines
- NRST Pin
 - Handled by the on-chip reset controller (**bidirectional**)
 - Provides a **reset** signal to the **external** components
 - Is asserted low **externally** to reset the microcontroller
- NRSTB Pin
 - **Asynchronous** reset of the SAM3X/A series when asserted low
- ERASE Pin
 - Reinitialize the Flash content to an **erased** state

SAM3X8E External Memories

- **External Memory Bus**
 - Integrates **Four** External Memory Controllers
 - Static Memory Controller
 - NAND Flash Controller
 - SLC NAND Flash ECC Controller
 - Single Data Rate Synchronous Dynamic Random Access Memory (SDR-SDRAM)
 - Up to **24-bit Address Bus** (up to 16 Mbytes linear per chip select)
 - Up to 8 chip selects, **Configurable Assignment**

Real-time Timer (RTT)

we call it real because it uses the time that we use (i.e seconds),not it's

- A 32-bit counter
- For counting elapsed **seconds**
- Generates a **periodic** interrupt and/or triggers an alarm
- Real-time Timer Value Register
 - Name: RTT_VR
 - Address: 0x400E1A38
 - Access: Read-only

Real-time Clock (RTC)

- RTC peripheral is designed for **very low power** consumption
- Combines a complete time-of-day clock with alarm and a 200-year **Gregorian** calendar
- **Programmable** periodic interrupt

Watchdog Timer (WDT)

- Used to prevent system **lock-up**
 - If the software becomes trapped in a **deadlock**
 - Is loaded with an initial value **greater than the worst case time delay**
 - If not reinitialized before reaching 0
 - An interrupt is generated to reset the processor
- A **12-bit down counter**
 - Watchdog period of up to **16 seconds**

The End!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 4

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013
- Design of Microprocessor-Based Systems (aka Embedded Systems Design and Implementation), Prabal Dutta, University of Michigan
- Cortex™-M3 Revision r2p1 Technical Reference Manual
- ARMv7-M Architecture Reference Manual

Interrupts

Interrupts

- Merriam-Webster
 - “to break the uniformity or continuity of”
- Informs a program of some external events
- Breaks execution flow
- Key questions
 - Where do interrupts come from?
 - How do we save state for later continuation?
 - How can we ignore interrupts?
 - How can we prioritize interrupts?
 - How can we share interrupts?

I/O Data Transfer

- Two key questions to determine how data is transferred to/from a non-trivial I/O device
 - How does the CPU know when data is available?
 - Polling
 - Interrupts
 - How is data transferred into and out of the device?
 - Programmed I/O
 - Direct Memory Access (DMA)

Interrupts

- **Interrupt (a.k.a. exception or trap):**
 - An event that causes the CPU to **stop** executing the current program and **begin** executing a **special piece** of code called an **interrupt handler** or **interrupt service routine** (ISR). Typically, the ISR does some work and then resumes the interrupted program.
- **Interrupts are really **glorified** procedure calls, except that they:**
 - can occur between **any two** instructions
 - are **transparent** to the running program (usually)
 - are **not explicitly** requested by the program (typically)
 - call a procedure at an address determined by the **type of interrupt**, not the program

Two basic types of interrupts

- Those caused by **an instruction** (Trap, exception)
 - TLB miss
 - Illegal/unimplemented instruction
 - div by 0
- Those caused by the **external world** (interrupt, external interrupt)
 - External device
 - Reset button
 - Timer expires
 - Power failure
 - System error

Interrupts: How it works

- Something tells the processor core there is an interrupt
- Core transfers control to code that needs to be executed
- Said code “returns” to old program
- Much harder than it looks
 - Why?

Interrupts: How it works

- How do you figure out where to branch to?
- How do you ensure that you can get back to where you started?
- Don't we have a pipeline? What about partially executed instructions?
- What if we get an interrupt while we are processing our interrupt?
- What if we are in a critical section?

Interrupts: Where

- If you know **what caused** the interrupt then you want to jump to the code that handles that interrupt
 - If you **number** the possible **interrupt** cases, and an interrupt comes in, you can just branch to a location, **using that number** as an offset (this is a branch table)
 - If you don't have the number, you need to **poll all possible** sources of the interrupt to see who caused it
 - Then you branch to the right code

Interrupts: Snazzy architectures

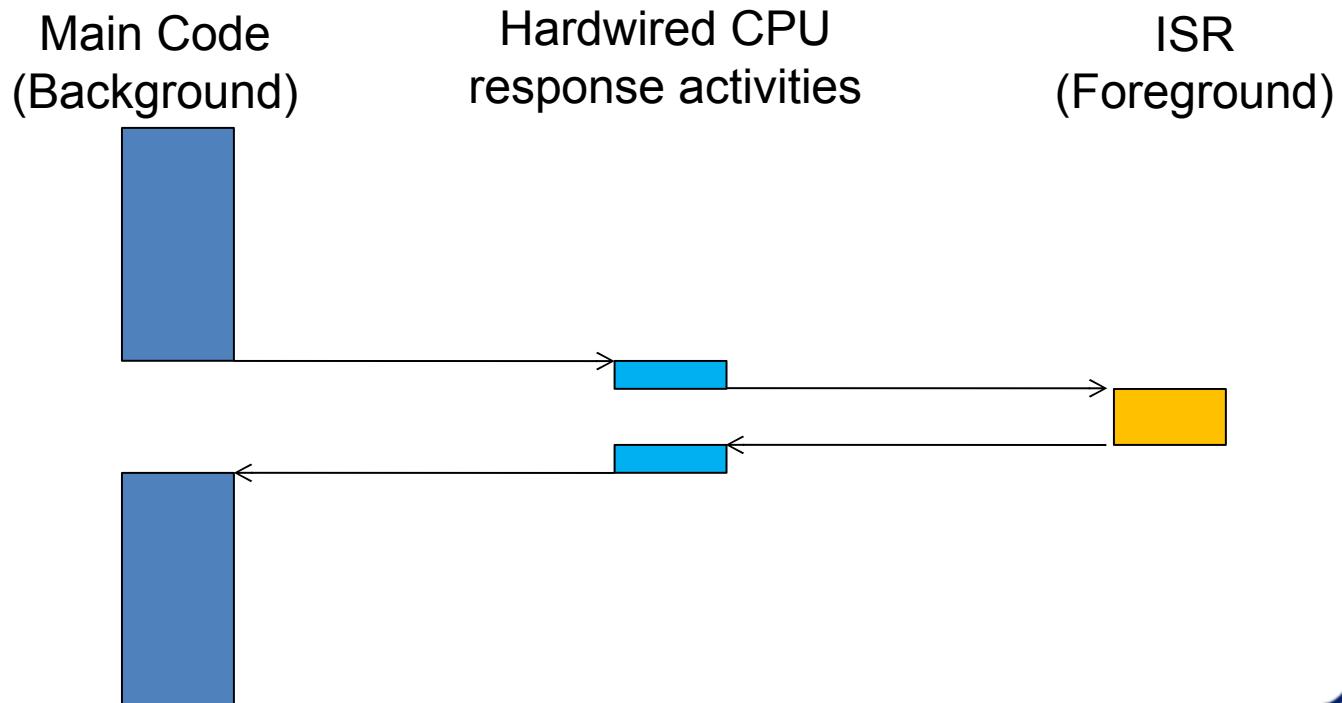
- A modern processor has many (often 50+) instructions **in-flight** at once
 - What do we do with them?
- **Drain the pipeline?**
 - What if one of them causes an exception?
- **Punt all that work**
 - Slows us down
- **What if the instruction that caused the exception was executed before some other instruction?**
 - What if that **other instruction** caused an interrupt?

Nested interrupts

- If we get one interrupt while handling another what to do?
 - Just handle it
 - What if I'm doing something that can't be stopped?
 - Ignore it
 - But what if it is important?
 - Prioritize
 - Take those interrupts you care about and ignore the rest

Interrupt or Exception Processing Sequence

- Other code (background) is running
- Interrupt trigger occurs
- Processor does some hard-wired processing
- Processor executes ISR (foreground), including return-from-interrupt instruction at end
- Processor resumes other code



Interrupt or Exception Processing Sequence

- Finish current instruction (except for lengthy instructions)
- Push context (8 32-bit words) onto current stack (MSP or PSP)
 - xPSR, PC, LR (R14), R12, R3, R2, R1, R0
- Switch to handler/privileged mode, use MSP
- Load PC with address of exception handler
- Load LR with EXC_RETURN code
- Load IPSR with exception number
- Start executing code of exception handler

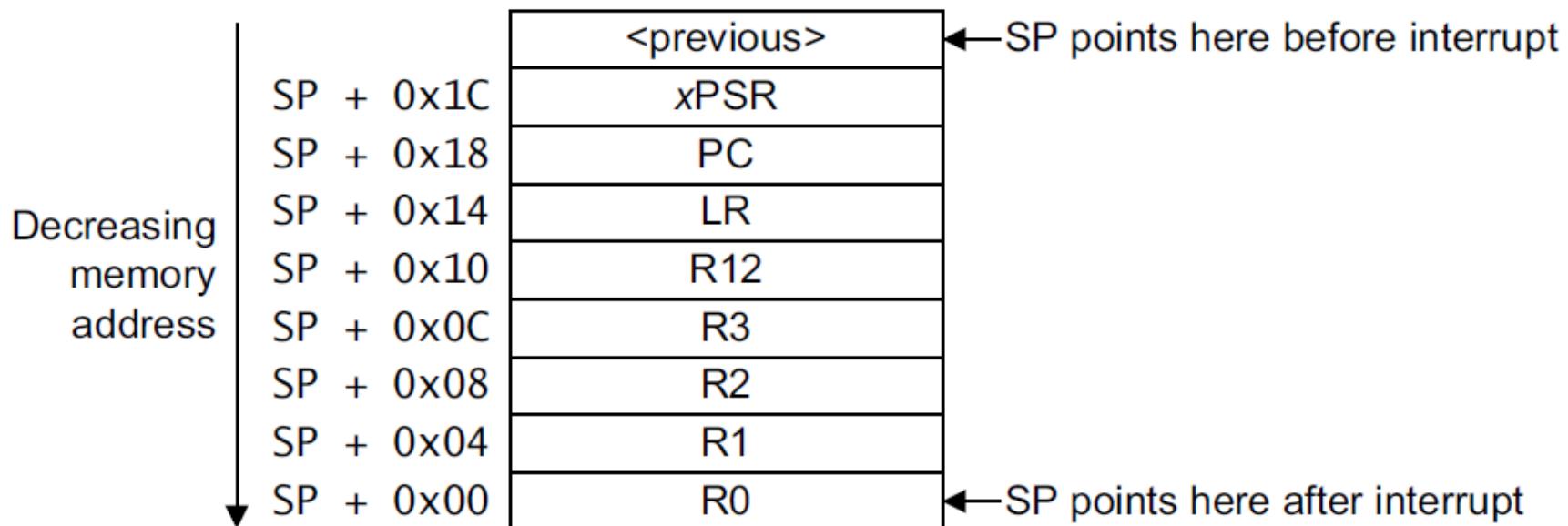
**Usually 16 cycles from exception request to execution of
first instruction in handler**

1. Finish Current Instruction

- Most instructions are short and finish quickly
- Some instructions may take many cycles to execute
 - Load Multiple (LDM), Store Multiple (STM), Push, Pop, MULS (32 cycles for some CPU core implementations)
- This will delay interrupt response significantly
- If one of these is executing when the interrupt is requested, the processor:
 - abandons the instruction
 - responds to the interrupt
 - executes the ISR
 - returns from interrupt
 - restarts the abandoned instruction

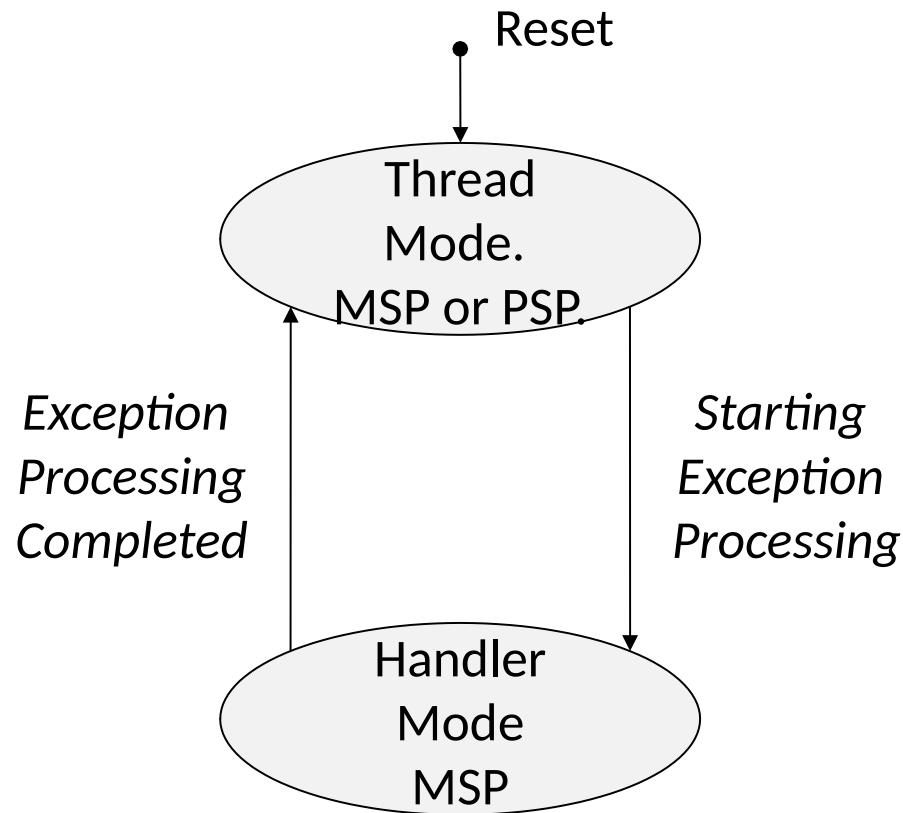
2. Push Context onto Current Stack

- Two SPs: Main (MSP), process (PSP)
- Which is active depends on operating mode, CONTROL register bit 1
- Stack grows toward smaller addresses



3. Switch to Handler/Privileged Mode

- Handler mode always uses Main SP



Exiting an Exception Handler

- Execute instruction triggering exception return processing
- Select return stack, restore context from that stack
- Resume execution of code at restored address

Nested Vectored Interrupt Controller (NVIC)

- NVIC **manages** and **prioritizes** external interrupts
- Exception states: **Inactive**, **Pending**, **Active**, **A&P**
- Processor state is **automatically stored** to the stack on an exception
 - and **automatically restored** from the stack at the end of ISR
- The processor supports **tail-chaining**
 - enables back-to-back interrupts **without** the **overhead** of state saving and restoration
- **Late-arriving** mechanism: to speed up pre-emption
- Dynamic **reprioritization** of interrupts
- **Configurable** number of interrupts: **from 1 to 240**
- Configurable number of interrupt priorities: **from 3 to 8 bits (8 to 256 levels)**
- Priority **masking** to support critical regions
- **Level** and **pulse** detection of interrupt signals

Nested Vectored Interrupt Controller

- Exception types

| Exception type | Position | Priority | Description |
|------------------------|----------|---------------------------|---|
| - | 0 | - | Stack top is loaded from first entry of vector table on reset. |
| Reset | 1 | -3 (highest) | Invoked on power up and warm reset. On first instruction, drops to lowest priority (Thread mode). This is asynchronous. |
| Non-maskable Interrupt | 2 | -2 | Cannot be stopped or pre-empted by any exception but reset. This is asynchronous. |
| Hard Fault | 3 | -1 | All classes of Fault, when the fault cannot activate because of priority or the Configurable Fault handler has been disabled. This is synchronous. |
| Memory Management | 4 | Configurable ^a | <i>Memory Protection Unit</i> (MPU) mismatch, including access violation and no match. This is synchronous. This is used even if the MPU is disabled or not present, to support the <i>Executable Never</i> (XN) regions of the default memory map. |

[Cortex-M3 Technical Reference Manual: Table 5-1]

Nested Vectored Interrupt Controller

- Exception types

| Exception type | Position | Priority | Description |
|--------------------|-------------|---------------------------|---|
| Bus Fault | 5 | Configurable ^a | Pre-fetch fault, memory access fault, and other address/memory related. This is synchronous when precise and asynchronous when imprecise. |
| Usage Fault | 6 | Configurable ^a | Usage fault, such as Undefined instruction executed or illegal state transition attempt. This is synchronous. |
| - | 7-10 | - | Reserved |
| SVCall | 11 | Configurable | System service call with SVC instruction. This is synchronous. |
| Debug Monitor | 12 | Configurable | Debug monitor, when not halting. This is synchronous, but only active when enabled. It does not activate if lower priority than the current activation. |
| - | 13 | - | Reserved |
| PendSV | 14 | Configurable | Pendable request for system service. This is asynchronous and only pended by software. |
| SysTick | 15 | Configurable | System tick timer has fired. This is asynchronous. |
| External Interrupt | 16 and more | Configurable | Asserted from outside the core, INTISR[239:0] , and fed through the NVIC (prioritized). These are all asynchronous. |

Nested Vectored Interrupt Controller

- NVIC register summary

| Address | Name | Type | Required privilege | Reset value | Description |
|-----------------------|-----------------------|------|---------------------------|-------------|--|
| 0xE000E100-0xE000E11C | NVIC_ISER0-NVIC_ISER7 | RW | Privileged | 0x00000000 | <i>Interrupt Set-enable Registers on page 4-4</i> |
| 0xE000E180-0xE000E19C | NVIC_ICER0-NVIC_ICER7 | RW | Privileged | 0x00000000 | <i>Interrupt Clear-enable Registers on page 4-5</i> |
| 0xE000E200-0xE000E21C | NVIC_ISPR0-NVIC_ISPR7 | RW | Privileged | 0x00000000 | <i>Interrupt Set-pending Registers on page 4-5</i> |
| 0xE000E280-0xE000E29C | NVIC_ICPR0-NVIC_ICPR7 | RW | Privileged | 0x00000000 | <i>Interrupt Clear-pending Registers on page 4-6</i> |
| 0xE000E300-0xE000E31C | NVIC_IABR0-NVIC_IABR7 | RW | Privileged | 0x00000000 | <i>Interrupt Active Bit Registers on page 4-7</i> |
| 0xE000E400-0xE000E4EF | NVIC_IPR0-NVIC_IPR59 | RW | Privileged | 0x00000000 | <i>Interrupt Priority Registers on page 4-7</i> |
| 0xE000EF00 | STIR | WO | Configurable ^a | 0x00000000 | <i>Software Trigger Interrupt Register on page 4-8</i> |

The End (for now)!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 5

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Cortex™-M3 Revision r2p1 Technical Reference Manual**
- **ARMv7-M Architecture Reference Manual**
- **Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series,
Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15**

Interrupts (2)

Interrupts Handling

- Functions to access the NVIC

`void NVIC_EnableIRQ (IRQn_Type IRQn)`

enables the specified device specific interrupt IRQ

`uint32_t NVIC_GetEnableIRQ (IRQn_Type IRQn)`

`void NVIC_DisableIRQ (IRQn_Type IRQn)`

`void NVIC_SetPendingIRQ (IRQn_Type IRQn)`

`uint32_t NVIC_GetPendingIRQ (IRQn_Type IRQn)`

`void NVIC_ClearPendingIRQ (IRQn_Type IRQn)`

`uint32_t NVIC_GetActive (IRQn_Type IRQn)`

https://www.keil.com/pack/doc/cmsis/Core/html/group__NVIC__gr.html#details

Interrupts Handling

- Functions to access the NVIC

```
void      NVIC_SetPriority (IRQn_Type IRQn, uint32_t priority)
```

- The number of priority levels is configurable and depends on the implementation of the chip designer. To determine the number of bits implemented for interrupt priority-level registers, write 0xFF to one of the priority-level register, then read back the value. For example, if the minimum number of 3 bits have been implemented, the read-back value is 0xE0
- Writes to unimplemented bits are ignored
- For Cortex-M3, Cortex-M4, and Cortex-M7
 - Dynamic switching of interrupt priority levels is supported
 - Supports 0 to 255 priority levels
 - Priority-level registers have a maximum width of 8 bits and a minimum of 3 bits

https://www.keil.com/pack/doc/cmsis/Core/html/group_NVIC_gr.html#details

Interrupts Handling

- Functions to access the NVIC

`uint32_t NVIC_GetPriority (IRQn_Type IRQn)`

`uint32_t NVIC_GetVector (IRQn_Type IRQn)`

address where ISR starts, example: address of functions is c

This function allows to read the address of an interrupt handler function

`void NVIC_SetVector (IRQn_Type IRQn, uint32_t vector)`

This function allows to change the address of an interrupt handler function

`void NVIC_SystemReset (void)`

This function requests a system reset by setting the SYSRESETREQ flag

https://www.keil.com/pack/doc/cmsis/Core/html/group__NVIC__gr.html#details

Interrupts Handling

- The symbol **__Vectors** is the address of the vector table in the startup code
- Register **SCB->VTOR** holds the start address of the vector table
- SCB: System Control Block, some of its registers:
 - **ICSR**: Interrupt Control and State Register
 - **VTOR**: Vector Table Offset Register
 - **AIRCR**: Application Interrupt and Reset Control Register
 - **SCR**: System Control Register

https://www.keil.com/pack/doc/cmsis/Core/html/group_NVIC_gr.html#details

Vector Table

- At the beginning of the vector table, the initial stack value and the exception vectors of the processor are defined

```
__Vectors    DCD    __initial_sp          ; Top of Stack initialization
              DCD    Reset_Handler        ; Reset Handler
              DCD    NMI_Handler         ; NMI Handler
              DCD    HardFault_Handler   ; Hard Fault Handler
              DCD    MemManage_Handler   ; MPU Fault Handler
              DCD    BusFault_Handler    ; Bus Fault Handler
              DCD    UsageFault_Handler  ; Usage Fault Handler
              DCD    SecureFault_Handler ; Secure Fault Handler
              DCD    0                  ; Reserved
              DCD    0                  ; Reserved
              DCD    0                  ; Reserved
              DCD    SVC_Handler         ; SVCall Handler
              DCD    DebugMon_Handler    ; Debug Monitor Handler
              DCD    0                  ; Reserved
              DCD    PendSV_Handler      ; PendSV Handler
              DCD    SysTick_Handler     ; SysTick Handler
```

https://www.keil.com/pack/doc/cmsis/Core/html/group_NVIC_gr.html#details

Vector Table

- Device Specific Vectors

- Following the processor exception vectors, the vector table contains also the device specific interrupt vectors

```
__Vectors    DCD    __initial_sp      ; Top of Stack initialization  
              DCD    Reset_Handler    ; Reset Handler  
              ...  
              DCD    SysTick_Handler  ; SysTick Handler
```

; device specific interrupts

```
              DCD    WWDG_IRQHandler ; Window Watchdog  
              DCD    PVD_IRQHandler   ; PVD through EXTI Line detect  
              DCD    TAMPER_IRQHandler ; Tamper
```

- All device specific interrupts should have a default interrupt handler function that can be overwritten in user code

- Remapping interrupt vectors by updating VTOR register

- An example: https://www.keil.com/pack/doc/cmsis/Core/html/using_VTOR_pg.html

https://www.keil.com/pack/doc/cmsis/Core/html/group_NVIC_gr.html#details

interrupt handler function

Default_Handler PROC

EXPORT WWDG_IRQHandler [WEAK]

EXPORT PVD_IRQHandler [WEAK]

EXPORT TAMPER_IRQHandler [WEAK]

:

:

WWDG_IRQHandler

PVD_IRQHandler

TAMPER_IRQHandler

:

:

B.

ENDP

- The user application may simply define an interrupt handler function by using the handler name

```
void WWDG_IRQHandler(void)
{
  ...
}
```

https://www.keil.com/pack/doc/cmsis/Core/html/group__NVIC__gr.html#details

interrupt handler function

```
#include "LPC17xx.h"
uint32_t active;                                /* Variable to store interrupt active state */
void TIMERO_IRQHandler(void) {                  /* Timer 0 interrupt handler */
    if (LPC_TIM0->IR & (1 << 0)) {          /* Check if interrupt for match channel 0 occured */
        LPC_TIM0->IR |= (1 << 0); /* Acknowledge interrupt for match channel 0 occured */
    }
    active = NVIC_GetActive(TIMERO IRQn);      /* Get interrupt active state of timer 0 */
}
int main (void) {
    /* Set match channel register MR0 to 1 millisecond */
    LPC_TIM0->MR0 = (((SystemCoreClock / 1000) / 4) - 1); /* 1 ms? */
    LPC_TIM0->MCR = (3 << 0); /* Enable interrupt and reset for match channel MR0 */
    NVIC_EnableIRQ(TIMERO IRQn);      /* Enable NVIC interrupt for timer 0 */
    LPC_TIM0->TCR = (1 << 0);      /* Enable timer 0 */
    while(1);
}
```

should be exactly the same as interrupt handler name

https://www.keil.com/pack/doc/cmsis/Core/html/group__NVIC__gr.html#details

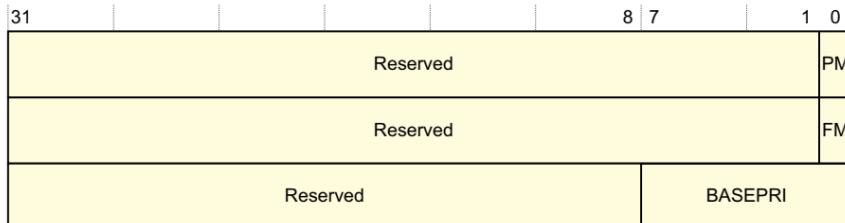
Masking Exception

- **PRIMASK:** Exception mask register (CPU core)
 - Register to mask out configurable exceptions
 - A 1-bit register
 - Setting PRIMASK to 1 raises the execution priority to 0
 - CMSIS-CORE API

```
void __enable_irq()           // clears PM flag  
void __disable_irq()          // sets PM flag  
uint32_t __get_PRIMASK()      // returns value of PRIMASK  
void __set_PRIMASK(uint32_t x) // sets PRIMASK to x
```

for the situations that we have a variable name x, that we don't know

- **BASEPRI:** Mask exception from a defined level
 - An 8-bit register
 - BASEPRI changes the priority level required for exception preemption
 - A value of zero disables masking by BASEPRI
- **FAULTMASK**
 - A 1-bit register
 - Raises the execution priority to -1 (the priority of HardFault)
 - Returning from any exception except NMI clears FAULTMASK to 0



REF: ARMv7-M Architecture Reference Manual

- CMSIS: Cortex Microcontroller Software Interface Standard
 - Is a vendor-independent hardware abstraction layer for microcontrollers that are based on Arm® Cortex® processors
 - Defines generic tool interfaces and enables consistent device support
 - Provides simple software interfaces to the processor and the peripherals
 - Simplifies software re-use
 - Reduces the learning curve for microcontroller developers,
 - Reduces the time to market for new devices
- Has been created to help the industry in standardization
- Enables consistent software layers and device support across a wide range of development tools and microcontrollers
- The CMSIS recommends the following conventions for identifiers:
 - **CAPITAL** names to identify Core Registers, Peripheral Registers, and CPU Instructions.
 - **CamelCase** names to identify function names and interrupt functions.
 - **Namespace_** prefixes avoid clashes with user identifiers and provide functional groups (i.e. for peripherals, RTOS, or DSP Library)

<http://www.keil.com/pack/doc/CMSIS/General/html/index.html>

Maximum Interrupt Rate

- We can only handle so many interrupts per second
 - F_{Max_Int} : maximum interrupt frequency
 - F_{CPU} : CPU clock frequency
 - C_{ISR} : Number of cycles ISR takes to execute
 - $C_{Overhead}$: Number of cycles of overhead for saving state, vectoring, restoring state, etc.
 - $$F_{Max_Int} = F_{CPU} / (C_{ISR} + C_{Overhead})$$
 - Note that model applies only when there is one interrupt in the system
- When processor is responding to interrupts, it isn't executing our other code
 - U_{Int} : Utilization (fraction of processor time) consumed by interrupt processing
 - $$U_{Int} = 100\% * F_{Int} * (C_{ISR} + C_{Overhead}) / F_{CPU}$$
 - CPU looks like it's running the other code with CPU clock speed of $(1 - U_{Int}) * F_{CPU}$

Sharing Data Safely between ISRs and other Threads

- **Volatile data:** can be updated outside of the program's immediate control
 - In computer programming, particularly in the C, C++, C#, and Java programming languages, the volatile keyword indicates that a value may change between different accesses, even if it does not appear to be modified. This keyword prevents an optimizing compiler from optimizing away subsequent reads or writes and thus incorrectly reusing a stale value or omitting writes. Volatile values primarily arise in hardware access (memory-mapped I/O), where reading from or writing to memory is used to communicate with peripheral devices, and in threading, where a different thread may have modified a value. [Wikipedia]
- **Non-atomic shared data:** can be interrupted partway through read or write, is vulnerable to **race conditions**

Non-Atomic Shared Data

- Want to keep track of current time and date
- Use 1 Hz interrupt from timer
- System
 - **TimerVal** structure tracks time and days since some reference event
 - TimerVal's fields are updated by periodic 1 Hz timer ISR

```
void GetDateTime(DateTimeType * DT) {  
    DT->day = TimerVal.day;  
    DT->hour = TimerVal.hour;  
    DT->minute = TimerVal.minute;  
    DT->second = TimerVal.second;  
}
```

```
void DateTimeISR(void) {  
    TimerVal.second++;  
    if (TimerVal.second > 59) {  
        TimerVal.second = 0;  
        TimerVal.minute++;  
        if (TimerVal.minute > 59) {  
            TimerVal.minute = 0;  
            TimerVal.hour++;  
            if (TimerVal.hour > 23) {  
                TimerVal.hour = 0;  
                TimerVal.day++;  
                ... etc.  
            }  
        }  
    }  
}
```

Non-Atomic Shared Data

- **Problem**
 - An interrupt at the wrong time will lead to half-updated data in DT
- **Failure Case**
 - TimerVal is {10, 23, 59, 59} (10th day, 23:59:59)
 - Task code calls GetDateTime(), which starts copying the TimerVal fields to DT:
day = 10, hour = 23
 - A timer interrupt occurs, which updates TimerVal to {11, 0, 0, 0}
 - GetDateTime() resumes executing, copying the remaining TimerVal fields to DT:
minute = 0, second = 0
 - DT now has a time stamp of {10, 23, 0, 0}
 - **The system thinks time just jumped backwards one hour!**
- **Fundamental problem - “race condition”**
 - Preemption enables ISR to interrupt other code and possibly overwrite data
 - Must ensure *atomic (indivisible)* access to the object

Examining the Problem More Closely

- Must protect any data object which both
 - requires multiple instructions to read or write (non-atomic access), and
 - is potentially written by an ISR

```
void GetDateTime(DateTimeType * DT) {  
    uint32_t m;  
    m = __get_PRIMASK();  
    __disable_irq();  
  
    DT->day = TimerVal.day;  
    DT->hour = TimerVal.hour;  
    DT->minute = TimerVal.minute;  
    DT->second = TimerVal.second;  
    __set_PRIMASK(m);  
}
```

The End!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 6

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Design of Microprocessor-Based Systems (aka Embedded Systems Design and Implementation), Prabal Dutta, University of Michigan**
- **Cortex™-M3 Revision r2p1 Technical Reference Manual**
- **ARMv7-M Architecture Reference Manual**
- **Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series, Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15**

Parallel Input/Output Controller (PIO)

PIO Description

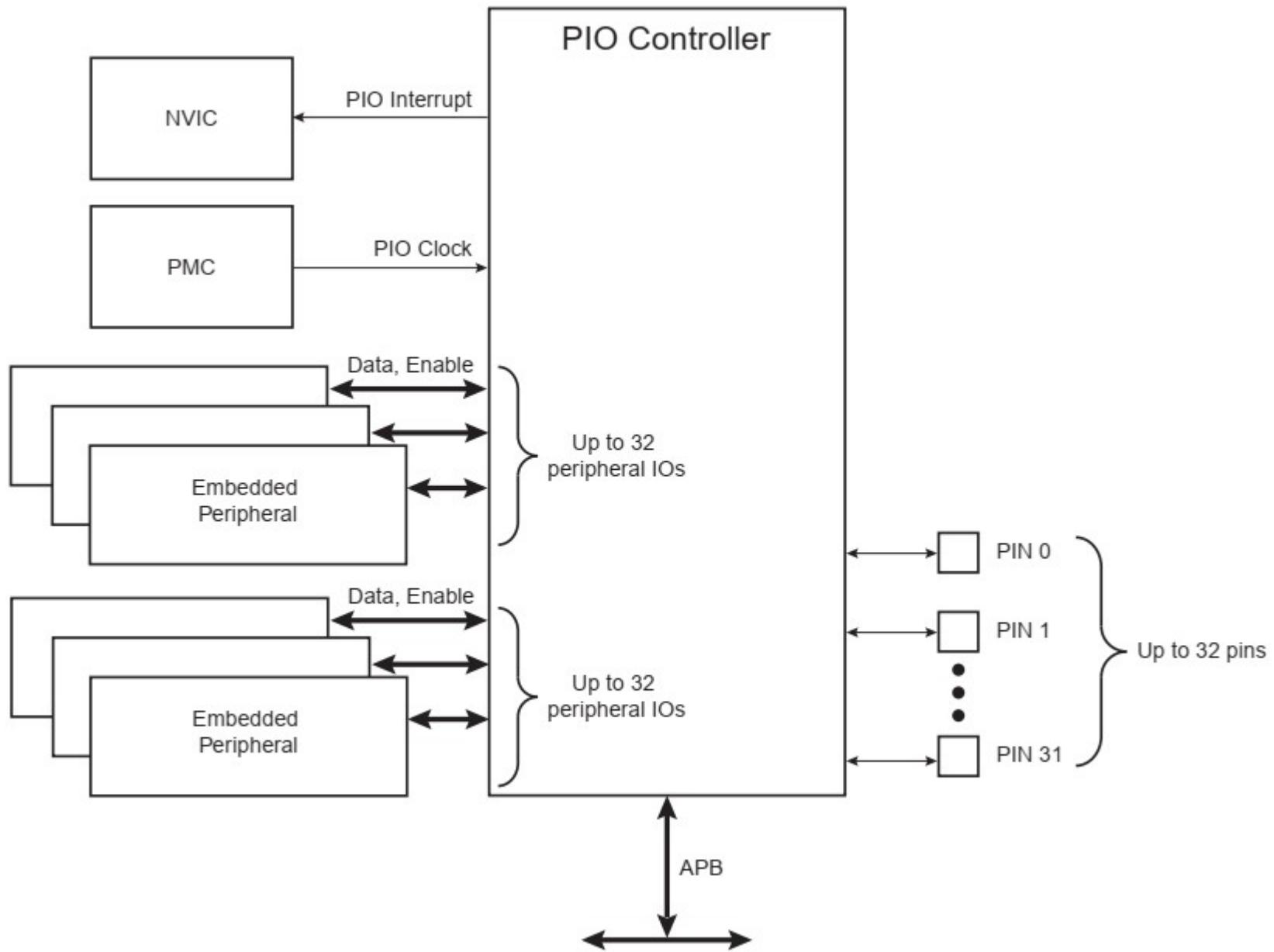
- **Parallel Input/Output Controller (PIO)**
 - manages up to 32 fully programmable input/output lines
 - Each I/O line may be dedicated as a general-purpose I/O or be assigned to a function of an embedded peripheral
- **Each I/O line of the PIO Controller features**
 - An input change interrupt enabling level change detection on any I/O line.
 - Additional Interrupt modes enabling rising edge, falling edge, low level or high level detection on any I/O line.
 - A glitch filter providing rejection of glitches lower than one-half of system clock cycle
 - A debouncing filter providing rejection of unwanted pulses from key or push button operations
 - Multi-drive capability similar to an open drain I/O line
 - Control of the pull-up of the I/O line
 - Input visibility and output control
 - A synchronous output providing up to 32 bits of data output in a single write operation

PIO Description

- **Parallel Input/Output Controller (PIO)**
 - Each PIO Controller controls up to 32 programmable I/O Lines

| Version | 100 pin SAM3X/A | 144 pin SAM3X | 217 pin SAM3X8H ⁽¹⁾ |
|---------|--------------------|------------------|-----------------------------------|
| PIOA | 30 | | 32 |
| PIOB | | 32 | |
| PIOC | 1 | | 31 |
| PIOD | - | 10 | 31 |
| PIOE | - | - | 32 |
| PIOF | - | - | 6 |
| Total | 63 | 103 | 164 |

PIO Block Diagram



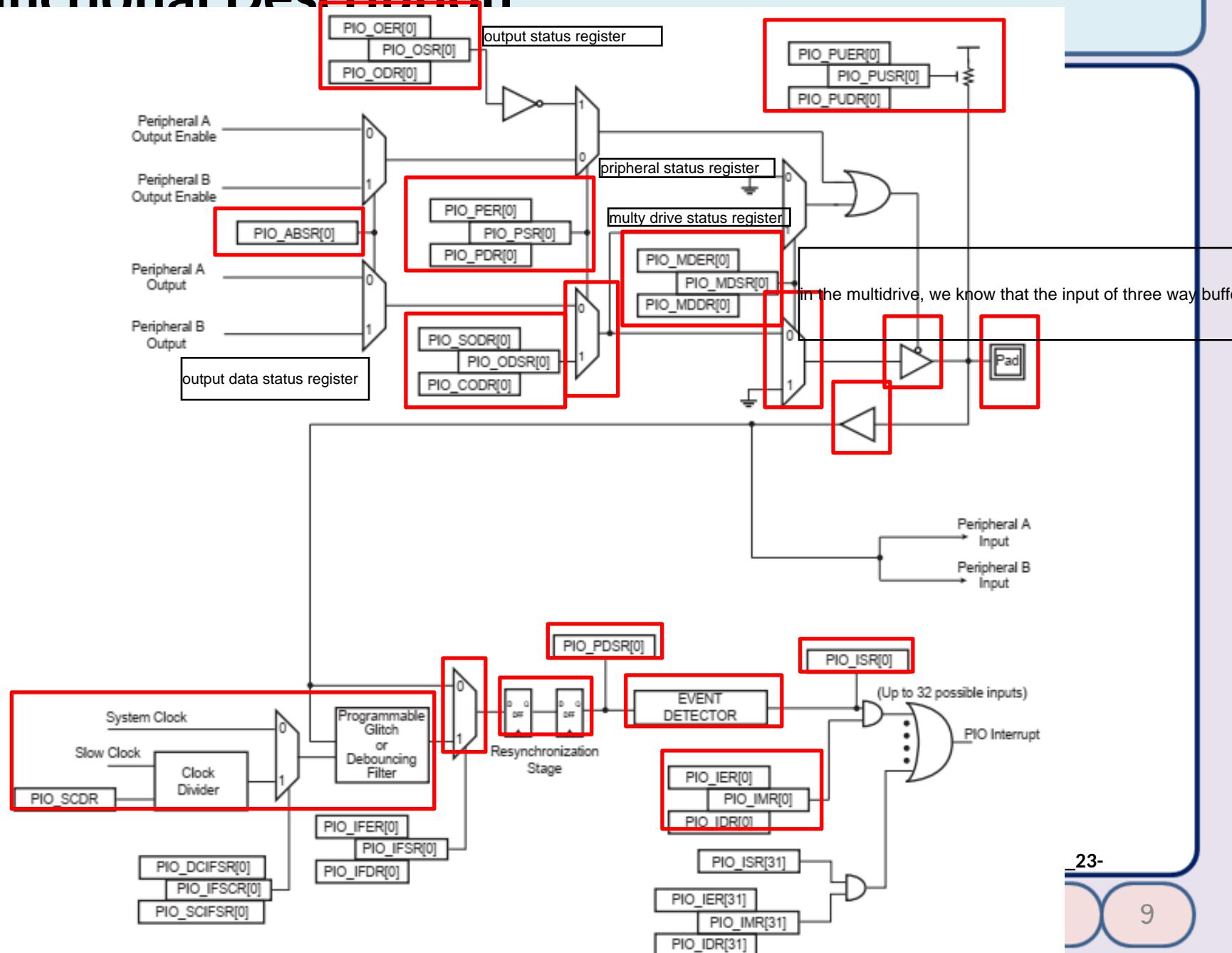
Product Dependencies

- Pin Multiplexing
 - Each pin is configurable, according to product definition
 - A general-purpose I/O line only
 - An I/O line multiplexed with one or two peripheral I/Os
- Power Management
 - Controls the PIO Controller clock in order to save power
 - Writing any of the registers of the user interface does not require the PIO Controller clock to be enabled
 - Not all of the features of the PIO Controller are available when the clock is disabled
 - Input Change Interrupt, Interrupt Modes on a programmable event and the **read of the pin level** require the clock to be validated
 - After a hardware reset, the PIO clock is **disabled by default**
 - The user must configure the Power Management Controller before any **access to the input line information**

Product Dependencies

- **Interrupt Generation**
 - The PIO Controller is connected on one of the sources of the NVIC
 - Using the PIO Controller requires the NVIC to be programmed first
 - The PIO Controller interrupt can be generated only if the PIO Controller clock is enabled

Functional Description



Functional Description

- **Pull-up Resistor Control**
 - **PIO_PUER: Pull-up Resistor Control**
 - **PIO_PUSR: Pull-up Status Register**
 - After reset, all of the pull-ups are enabled, i.e. **PIO_PUSR** resets at the value 0x0
- **I/O Line or Peripheral Function Selection**
 - **PIO_PER: PIO Enable Register**
 - **PIO_PDR: PIO Disable Register**
 - **PIO_PSR: PIO Status Register**
 - indicates whether the pin is controlled by the corresponding peripheral or by the PIO Controller, if 0:
 - pin is controlled by the corresponding on-chip peripheral selected in the **PIO_ABSR** (AB Select Register)

Functional Description

- **Output Control**
 - **PIO_OER: Output Enable Register**
 - **PIO_ODR: Output Disable Register**
 - **PIO_OSRR: Output Status Register**
 - When a bit in this register is at 0, the corresponding I/O line is used as an input only
 - When the bit is at 1, the corresponding I/O line is driven by the PIO controller
 - **PIO_SODR: Set Output Data Register**
 - **PIO_CODR: Clear Output Data Register**
 - **PIO_ODSR: Output Data Status Register**

Functional Description

- **Synchronous Data Output**
 - **Simultaneous write into PIO_SODR and PIO_CODR is not possible**
 - **PIO Controller offers a direct control of PIO outputs by single write access to
PIO_ODSR**

it's the only status register that we can give value, other status registers are initialized by hardware
- **Multi Drive Control (Open Drain)**
 - **Permits several drivers to be connected on the I/O line which is driven low only by each device**
 - **PIO_MDER: Multi-driver Enable Register**
 - **PIO_MDSR: indicates the pins that are configured to support external drivers**
 - **After reset, the Multi Drive feature is disabled on all pins (PIO_MDSR = 0)**

The End!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh
farbeh@aut.ac.ir

Department of Computer Engineering
Amirkabir University of Technology

Lecture 7

Copyright Notice

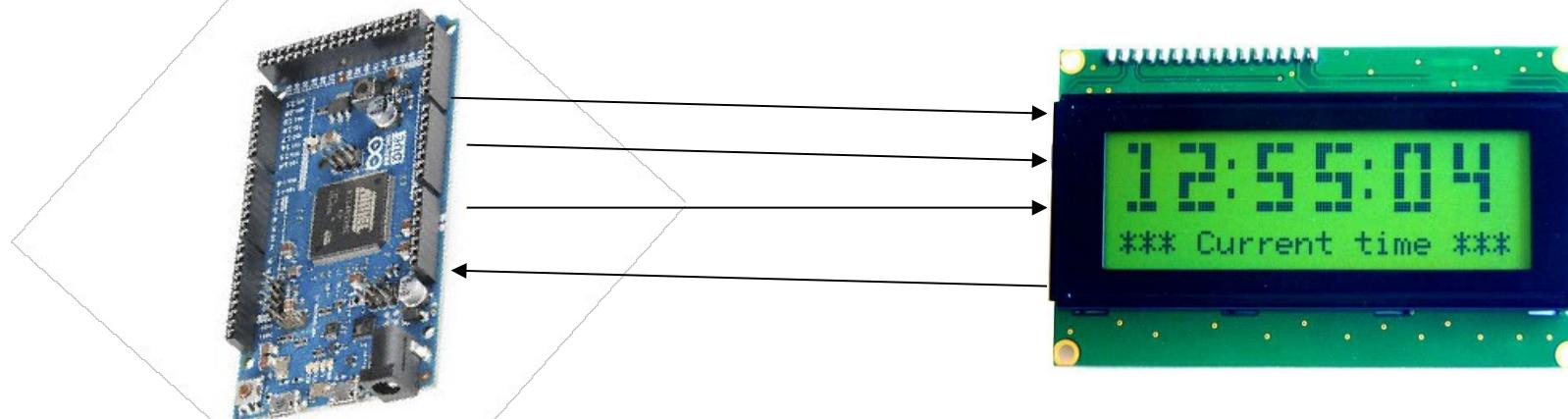
Parts (text & figures) of this lecture are adopted from:

- **Design of Microprocessor-Based Systems (aka Embedded Systems Design and Implementation), Prabal Dutta, University of Michigan**
- **Cortex™-M3 Revision r2p1 Technical Reference Manual**
- **ARMv7-M Architecture Reference Manual**
- **Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series, Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15**

Serial Peripheral Interface (SPI)

SPI Description

- **Synchronous serial data link**
 - Provides communication with external devices in Master or Slave Mode
 - Is essentially a shift register that serially transmits data bits to other SPIs
 - Fast, Easy to use, Simple
- A communication protocol using 4 wires
 - Also known as a 4 wire bus
- Used to communicate across small distances



SPI Capabilities

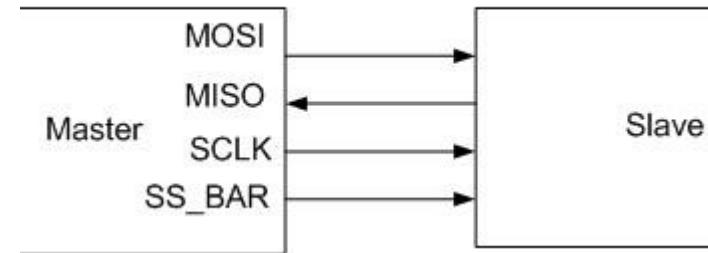
- **Always Full Duplex**
 - Communicating in two directions at the same time
- **Multiple Mbps transmission speed**
- **Transfers data in 8 to 16 bit characters**
- **Multiple slaves**
 - Daisy-chaining possible
- **Master controls the data flow**
 - Other devices act as slaves which have data shifted into and out by the master

SPI Protocol

- **Wires**

- **Two data lines and two control lines**
- **Master Out Slave In (MOSI)**
 - Supplies the output data from the master shifted into the input(s) of the slave(s)
- **Master In Slave Out (MISO)**
 - Supplies the output data from a slave to the input of the master
- **Serial Clock (SPCK)**
 - Is driven by the master and regulates the flow of the data bits. The master may transmit data at a variety of baud rates; the SPCK line cycles once for each bit that is transmitted
- **Slave Select (NSS)**
 - Allows slaves to be turned on and off by hardware

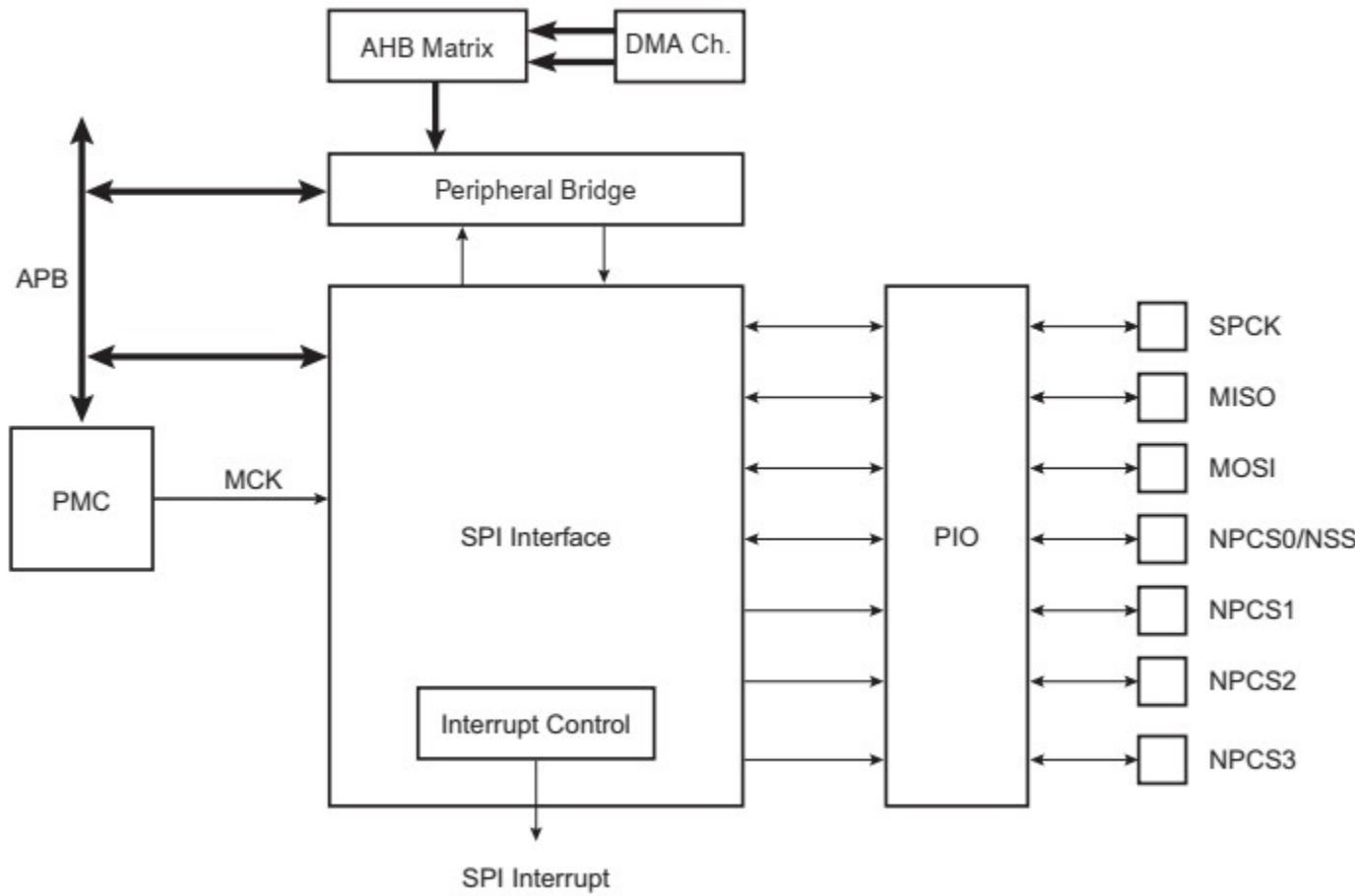
when NSS becomes 0 the connections starts ..



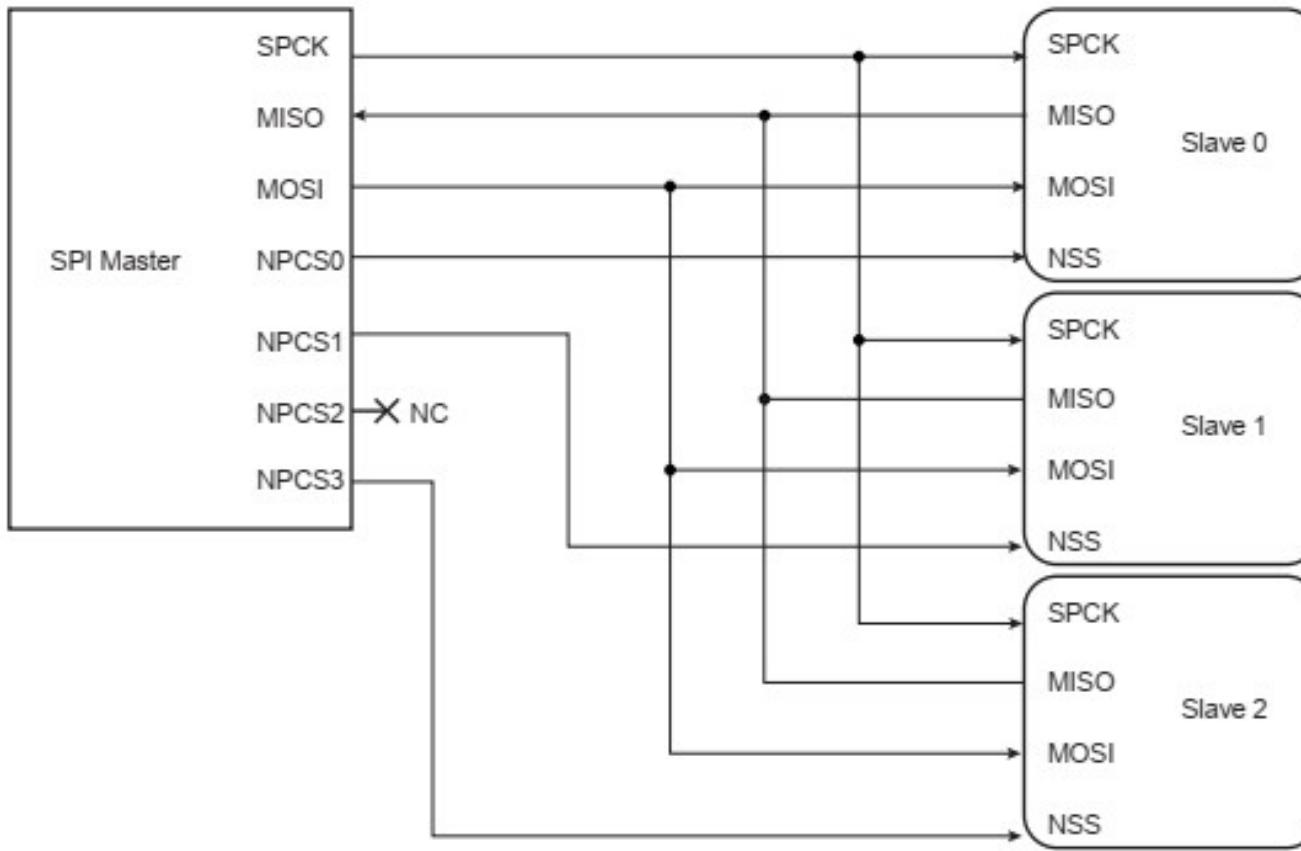
Embedded Characteristics

- **Supports Communication with Serial External Devices**
 - **Four Chip Selects with External Decoder Support Allow Communication with Up to 15 Peripherals** one state is for not starting a connection so we have 15 instead of 16
 - **Serial Memories, such as DataFlash and 3-wire EEPROMs**
 - **Serial Peripherals, such as ADCs, DACs, LCD Controllers, CAN Controllers and Sensors**
 - **External Co-processors**

Block Diagram



Block Diagram



Product Dependencies

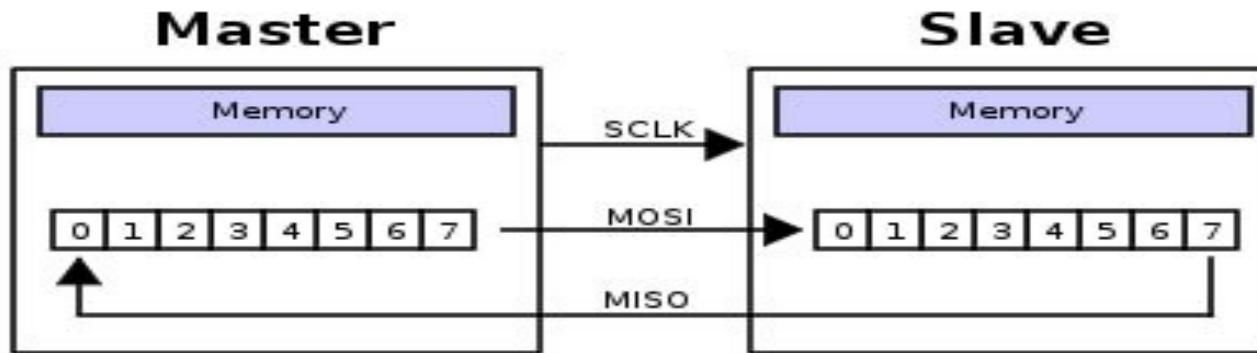
- **NOTE**
 - The pins used for interfacing the compliant external devices may be multiplexed with PIO lines. The programmer must first program the PIO controllers to assign the SPI pins to their peripheral functions

Functional Description

- **Modes of Operation: Master and Slave**
 - **Master Mode: MSTR bit = 1 in the Mode Register (SPI_MR)**
 - The pins NPCS0 to NPCS3 are all configured as outputs
 - SPCK pin is driven
 - The MISO line is wired on the receiver input
 - The MOSI line driven as an output by the transmitter
 - **Slave Mode: MSTR bit = 0 in the Mode Register (SPI_MR)**
 - The MISO line is driven by the transmitter output
 - The MOSI line is wired on the receiver input
 - The SPCK pin is driven by the transmitter to synchronize the receiver
 - The NPCS0 pin becomes an input, and is used as a Slave Select signal (NSS)
 - The pins NPCS1 to NPCS3 are not driven and can be used for other purposes

Data Transfer

- Two phases and two polarities of clock
 - program clock polarity: CPOL bit in the Chip Select Register
 - program clock phase: NCPHA bit
 - These two parameters determine the edges of the clock signal on which data is driven and sampled



| SPI Mode | CPOL | NCPHA | Shift SPCK Edge | Capture SPCK Edge | SPCK Inactive Level |
|----------|------|-------|-----------------|-------------------|---------------------|
| 0 | 0 | 1 | Falling | Rising | Low |
| 1 | 0 | 0 | Rising | Falling | Low |
| 2 | 1 | 1 | Rising | Falling | High |
| 3 | 1 | 0 | Falling | Rising | High |

SPI User Interface

| Offset | Register | Name | Access | Reset |
|-----------------|-----------------------------------|----------|------------|------------|
| 0x00 | Control Register | SPI_CR | Write-only | --- |
| 0x04 | Mode Register | SPI_MR | Read-write | 0x0 |
| 0x08 | Receive Data Register | SPI_RDR | Read-only | 0x0 |
| 0x0C | Transmit Data Register | SPI_TDR | Write-only | --- |
| 0x10 | Status Register | SPI_SR | Read-only | 0x000000F0 |
| 0x14 | Interrupt Enable Register | SPI_IER | Write-only | --- |
| 0x18 | Interrupt Disable Register | SPI_IDR | Write-only | --- |
| 0x1C | Interrupt Mask Register | SPI_IMR | Read-only | 0x0 |
| 0x20 - 0x2C | Reserved | | | |
| 0x30 | Chip Select Register 0 | SPI_CSR0 | Read-write | 0x0 |
| 0x34 | Chip Select Register 1 | SPI_CSR1 | Read-write | 0x0 |
| 0x38 | Chip Select Register 2 | SPI_CSR2 | Read-write | 0x0 |
| 0x3C | Chip Select Register 3 | SPI_CSR3 | Read-write | 0x0 |
| 0x4C - 0xE0 | Reserved | - | - | - |
| 0xE4 | Write Protection Control Register | SPI_WPMR | Read-write | 0x0 |
| 0xE8 | Write Protection Status Register | SPI_WPSR | Read-only | 0x0 |
| 0x00E8 - 0x00F8 | Reserved | - | - | - |
| 0x00FC | Reserved | - | - | - |

SPI Control Register

- **Write-only Register**
 - SPIEN: SPI Enable
 - SPIDIS: SPI Disable
 - If both SPIEN and SPIDIS are equal to one when the control register is written, the SPI is disabled
 - SWRST: SPI Software Reset
 - Reset the SPI. A software-triggered hardware reset of the SPI interface is performed
 - The SPI is in slave mode after software reset

When spi starts, it is on slave by default, and we should set the corresponding bit manually.

| | | | | | | | |
|-------|----|----|----|----|----|--------|----------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| - | - | - | - | - | - | - | LASTXFER |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| - | - | - | - | - | - | - | - |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| - | - | - | - | - | - | - | - |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| SWRST | - | - | - | - | - | SPIDIS | SPIEN |

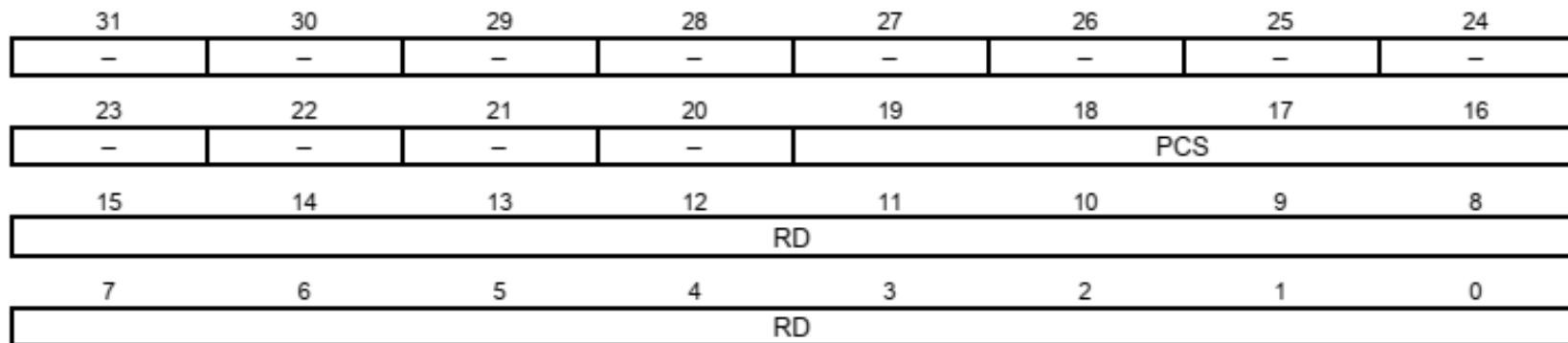
SPI Mode Register

- **Read-Write Register**
 - **MSTR:** Master/Slave Mode
 - **PS:** Peripheral Select
 - 0 = Fixed Peripheral Select.
 - 1 = Variable Peripheral Select
 - **PCSDEC:** Chip Select Decode
 - 0 = The chip selects are directly connected to a peripheral device
 - 1 = The four chip select lines are connected to a 4- to 16-bit decoder

| | | | | | | | |
|--------|----|-------|---------|----|--------|----|------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| DLYBCS | | | | | | | |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| - | - | - | - | | PCS | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| - | - | - | - | - | - | - | - |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| LLB | - | WDRBT | MODFDIS | - | PCSDEC | PS | MSTR |

SPI Receive Data Register

- **Read-Only Register**
 - **RD: Receive Data**
 - Data received by the SPI Interface is stored in this register right-justified
 - **PCS: Peripheral Chip Select**
 - In Master Mode only, these bits indicate the value on the NPCS pins at the end of a transfer

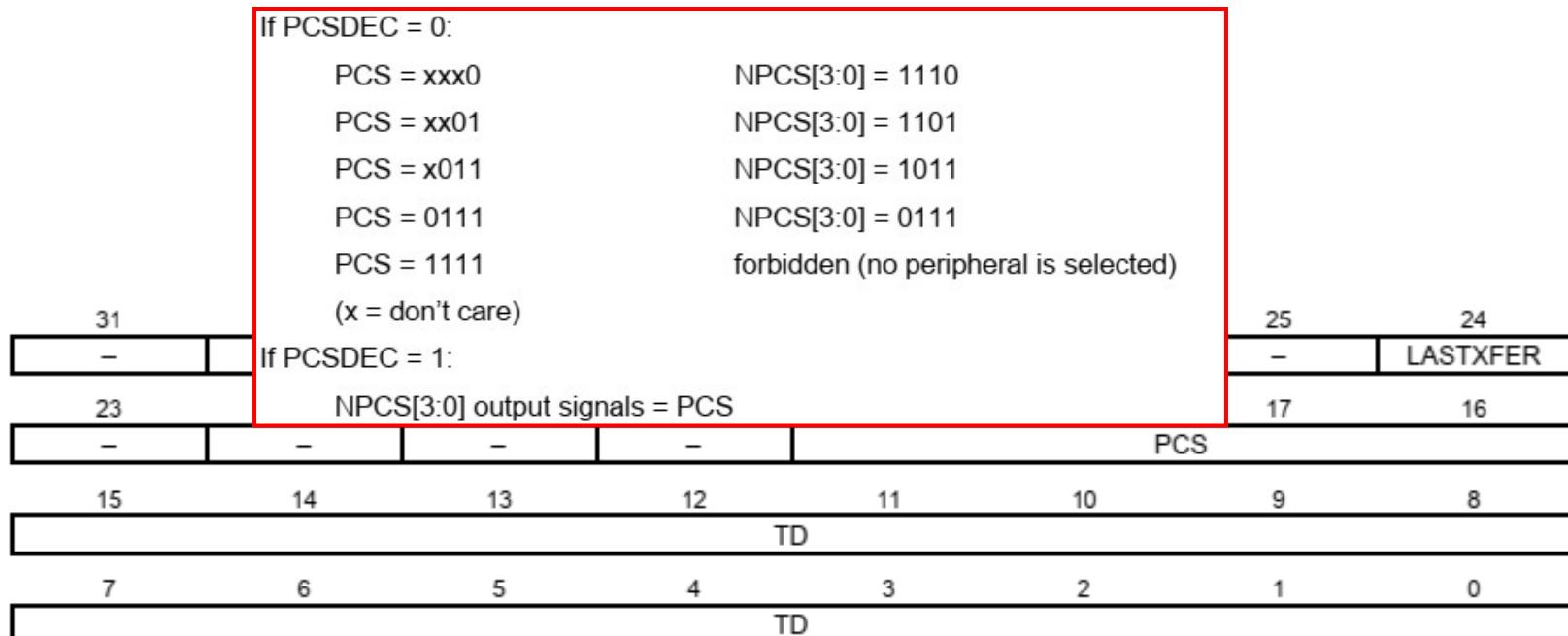


SPI Transmit Data Register

by default wires of slaved select are 1, and when we select them we make the corresponding bit 0

- **Write-Only Register**

- **TD: Transmit Data**
 - Data to be transmitted by the SPI Interface is stored in this register
- **PCS: Peripheral Chip Select**
 - This field is only used if Variable Peripheral Select is active



The End (for now)!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh
farbeh@aut.ac.ir

Department of Computer Engineering
Amirkabir University of Technology

Lecture 8

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Design of Microprocessor-Based Systems (aka Embedded Systems Design and Implementation), Prabal Dutta, University of Michigan**
- **UM10204: I2C-bus specification and user manual, Rev. 6 — 4 April 2014, User manual**
- **ARMv7-M Architecture Reference Manual**
- **Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series, Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15**

I²C-bus specification

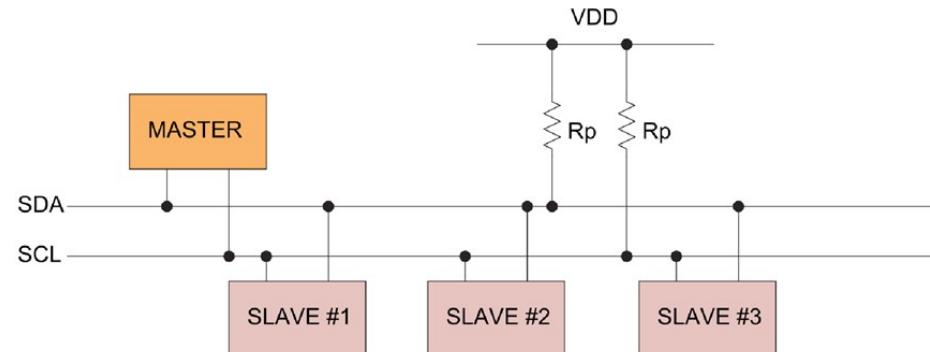
I²C Description

- **Synchronous serial data link**
 - Multi-master, multi-slave
 - Invented in 1982 by Philips Semiconductor (now NXP Semiconductors)
 - Widely used for attaching lower-speed peripheral ICs to processors and microcontrollers in short-distance
 - Appropriate for peripherals where simplicity and low manufacturing cost are more important than speed

<https://en.wikipedia.org/wiki/I%C2%BCC>

I²C Design

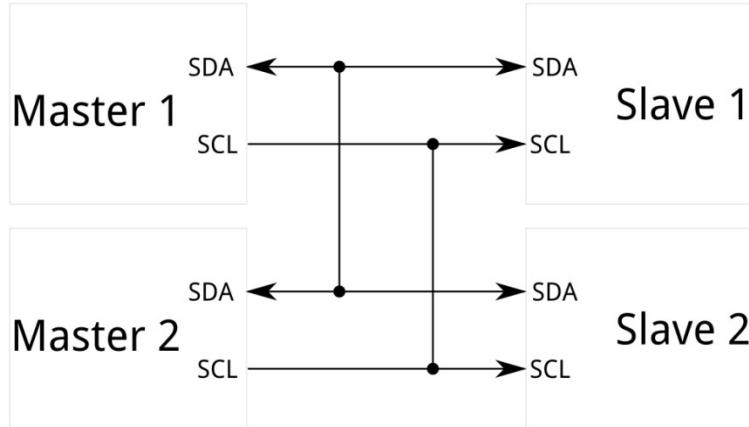
- Two bidirectional open collector or open drain lines
 - open collector (or open drain): behaves like a switch that is either connected to ground or disconnected
 - Serial Data Line (SDA) and Serial Clock Line (SCL)
 - pulled up with resistors
- 7-bit address space (rarely-used 10-bit extension)
- Bus speeds
 - 100 kbit/s: standard mode
 - 400 kbit/s: Fast mode
 - 10 kbit/s: low-speed mode
 - 1 Mbit/s: Fast mode plus
 - 3.4 Mbit/s: High Speed mode
 - 5 Mbit/s Ultra Fast-mode



the clock we refer to here, is not our MCU clock(in Mhz), this has been slower for external use.

I²C Design

- **Operating roles**
 - **Master:** generates the clock and initiates communication with slaves
 - **Slave:** receives the clock and responds when addressed by the master
- **Multi-master bus:** any number of master nodes can be present
- **Master and slave roles may be changed between messages**



at each instance of time we can just have one master.

I²C Design

- **Potential modes of operation for a given bus device**
 - Master – master node is sending data to a slave
 - Master receive – master node is receiving data from a slave
 - Slave transmit – slave node is sending data to the master
 - Slave receive – slave node is receiving data from the master

we can just be transmitter or receiver, so our connection is HALF DUPLEX!

I²C Data Transfer

- Message delimiters
 - START and STOP signals
 - Are distinct from the data bits (0 and 1)
- The master is initially in master transmit mode by sending a START followed by the 7-bit address of the slave
 - followed by a single bit representing whether it wishes to write (0) to or read (1) from the slave
- The slave will respond with an ACK bit (active low for acknowledged)
 - If slave exists
- The address and the data bytes are sent most significant bit first

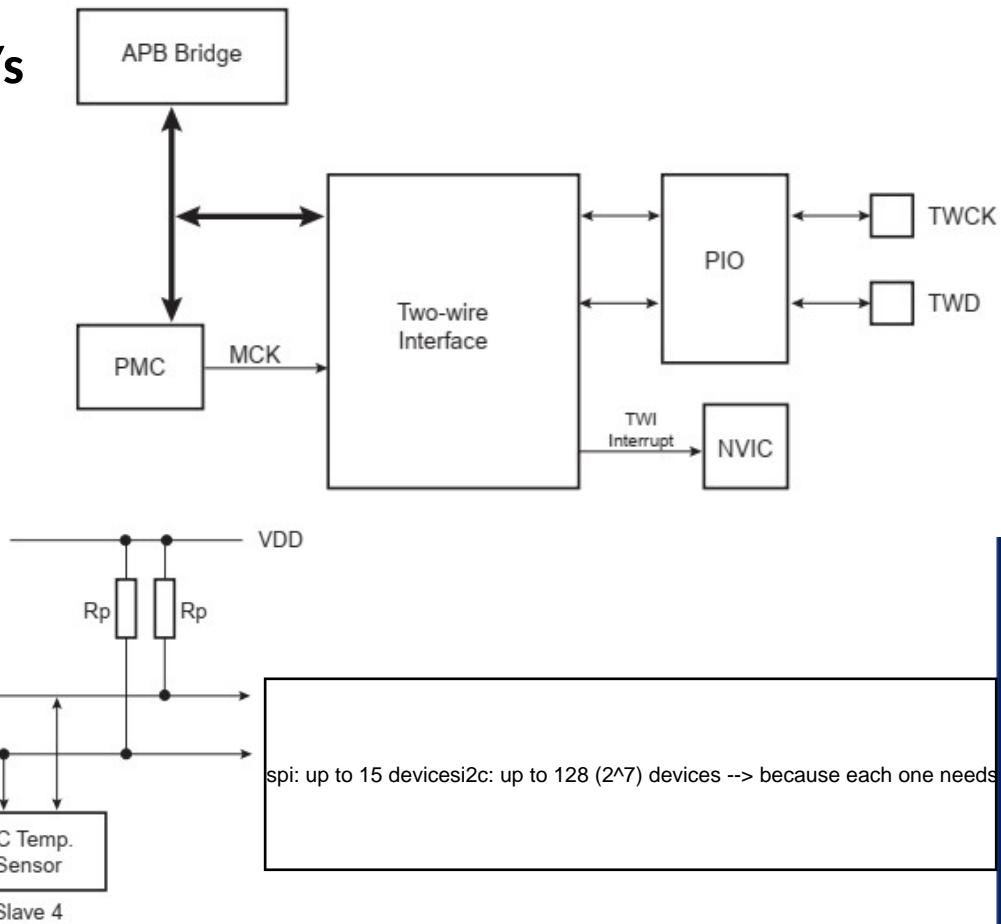
I²C Data Transfer

- The start bit is indicated by a high-to-low transition of SDA
 - with SCL high
- The stop bit is indicated by a low-to-high transition of SDA
 - with SCL high
- All other transitions of SDA take place with SCL low
- To write to the slave
 - The Master repeatedly sends a byte with the slave sending an ACK bit
- To read from slave
 - Master repeatedly receives a byte from the slave, and send an ACK bit after every byte except the last one

<https://en.wikipedia.org/wiki/I%C2%BCC>

Two-wire Interface (TWI)

- I2C-Compatible interface of ATMEL microcontroller
 - speeds of up to 400 Kbits/s



The End (for now)!

<https://i2c.info/i2c-bus-specification>



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 9

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

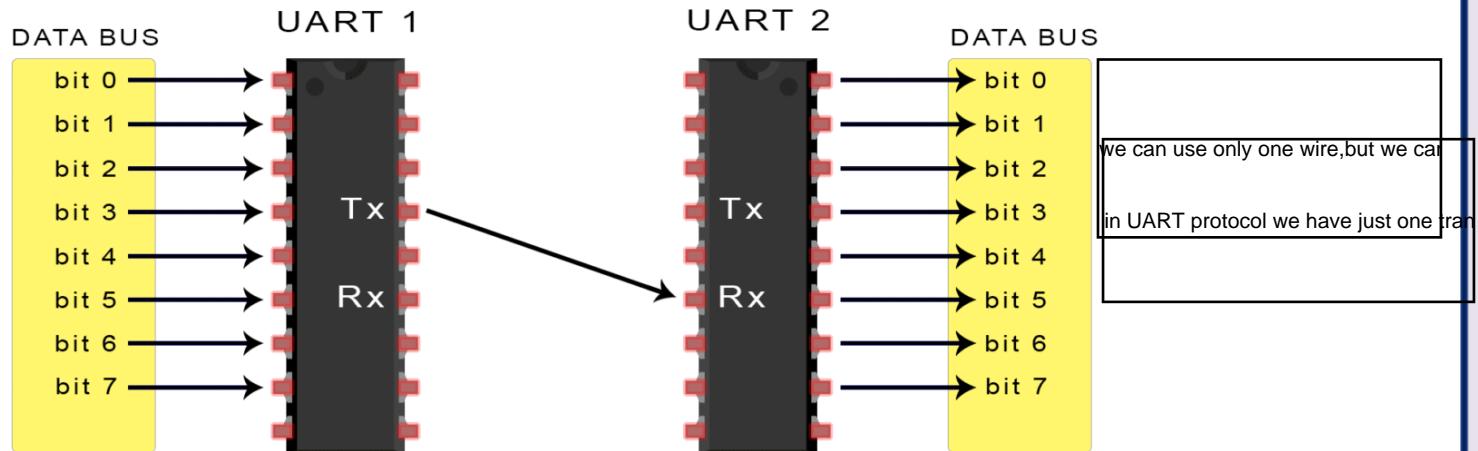
- Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series,
[Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15](#)
- <http://www.circuitbasics.com/basics-uart-communication/>

Universal Asynchronous Receiver/Transmitter (UART)

Basics of UART Communication

- Asynchronous serial communication
 - Uses two wires to transmit data between devices
 - Data flows from the Tx pin of the transmitting UART
 - to the Rx pin of the receiving UART
 - No clock signal to synchronize the output of bits from the transmitting UART
 - to the sampling of bits by the receiving UART

we can send and receive using Tx and Rx simultaneously



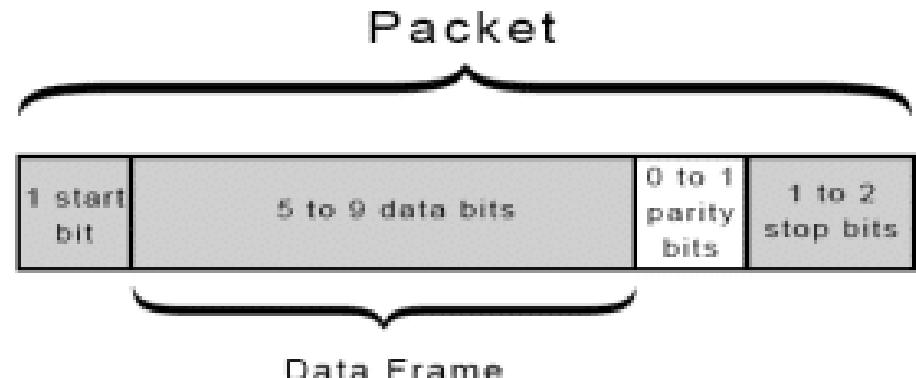
we can use only one wire, but we can send and receive simultaneously
in UART protocol we have just one frame

UART Communication Details

- The transmitting UART adds start and stop bits
 - to the data packet being transferred
- These bits define the beginning and end of the data packet
 - so the receiving UART knows when to start reading the bits
- The receiving UART detects a start bit
 - Then, starts to read the incoming bits
 - at a specific frequency known as the baud rate
- Baud rate: a measure of the speed of data transfer
 - expressed in bits per second (bps)
- The baud rate between the transmitting and receiving UARTs can only differ by about 10%

UART Communication Details

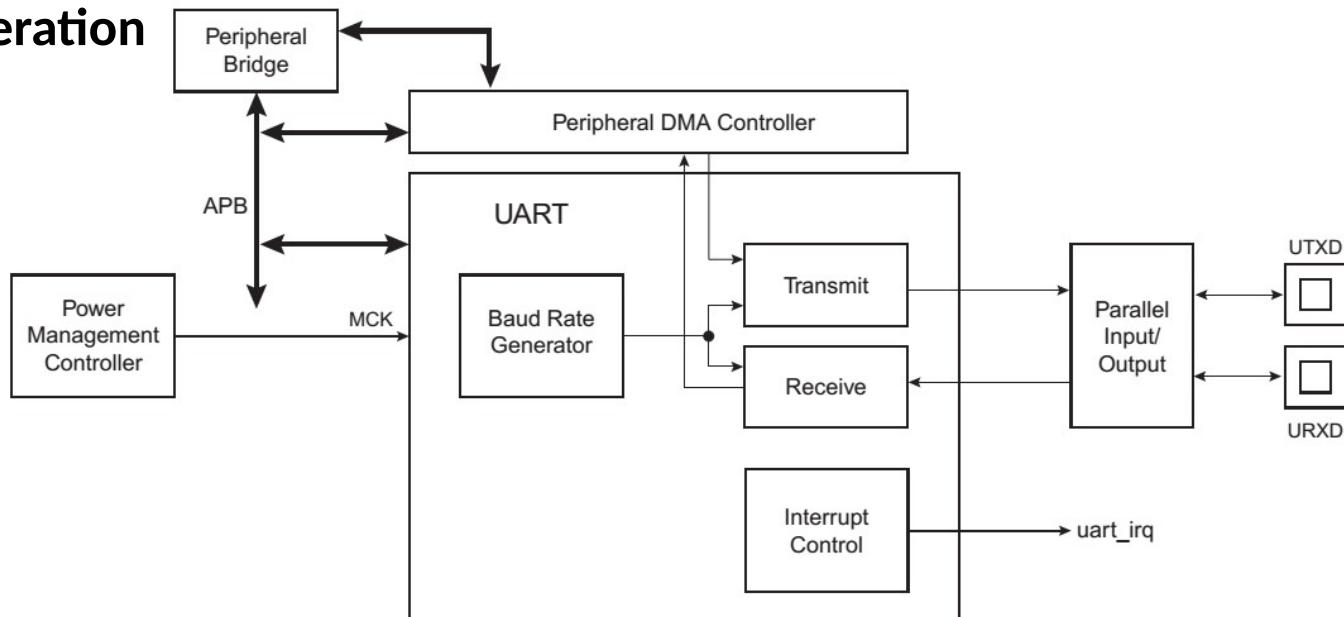
- When not transmitting data
 - The UART transmission line is held at a high voltage level
- START BIT
 - The transmitting UART pulls the line from high to low for one clock cycleclock cycle and baud rate are same.
- STOP BITS
 - The sending UART drives the data transmission line from a low voltage to a high voltage for at least two bit durations



UART in SAM3X8E

- Independent Receiver and Transmitter
 - with a Common Programmable Baud Rate Generator
- Even, Odd, Mark or Space Parity Generation
- Parity, Framing and Overrun Error Detection
- Interrupt Generation

space --> always 0 mark --> always 1



UART in SAM3X8E

- The UART pins are multiplexed with PIO lines
- The programmer must first configure the PMC to enable the UART clock
 - **PMC: Power Management Controller**
without clock UART cannot work.
- Only 8-bit character is supported (with parity)

| Instance | Signal | I/O Line | Peripheral |
|----------|--------|----------|------------|
| UART | URXD | PA8 | A |
| UART | UTXD | PA9 | A |

Baud Rate Generator in SAM3X8E

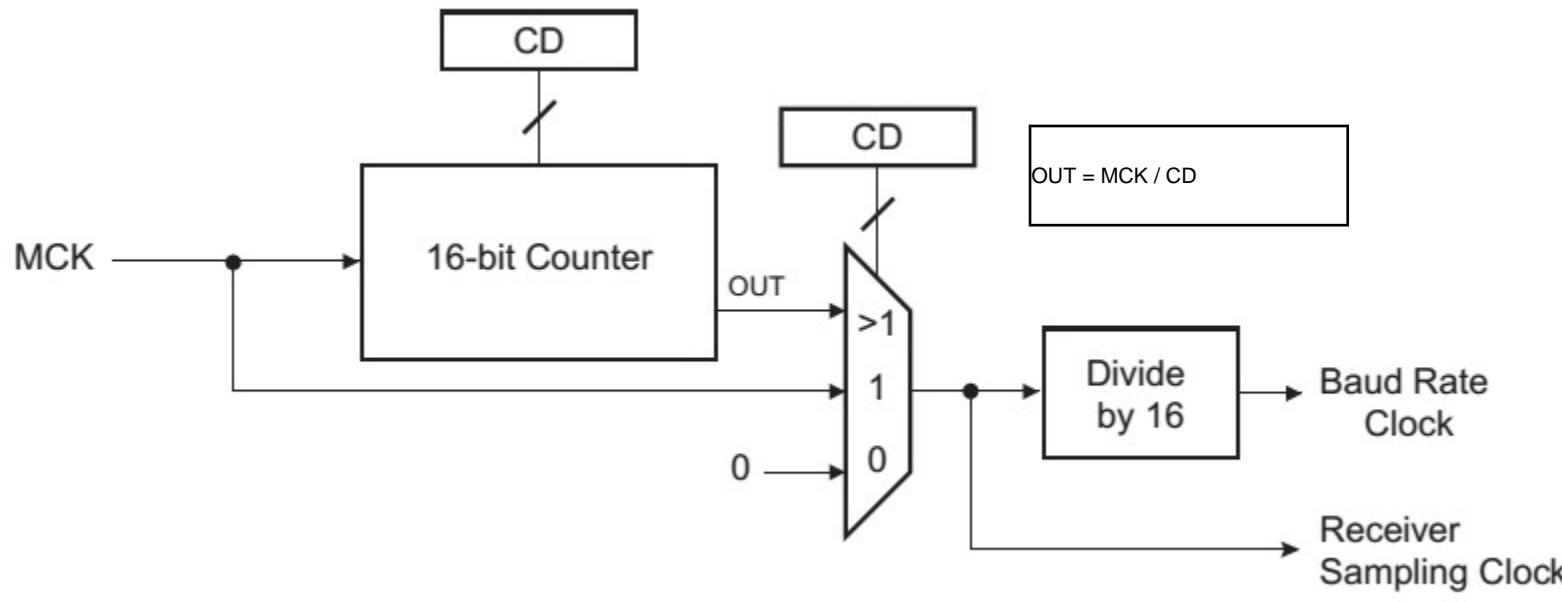
- The baud rate clock
 - Is the master clock divided by 16 times the value (CD) written in **UART_BRGR (Baud Rate Generator Register)**

$$\text{Baud Rate} = \frac{\text{MCK}}{16 \times \text{CD}}$$

value of CD stored in a 32 bit register, that we use only 16 bit of it so

- If **UART_BRGR** is set to 0, the baud rate clock is disabled and the UART remains inactive
- The maximum allowable baud rate is Master Clock divided by 16
 - The minimum allowable baud rate is Master Clock divided by (16 x 65536)

Baud Rate Generator in SAM3X8E



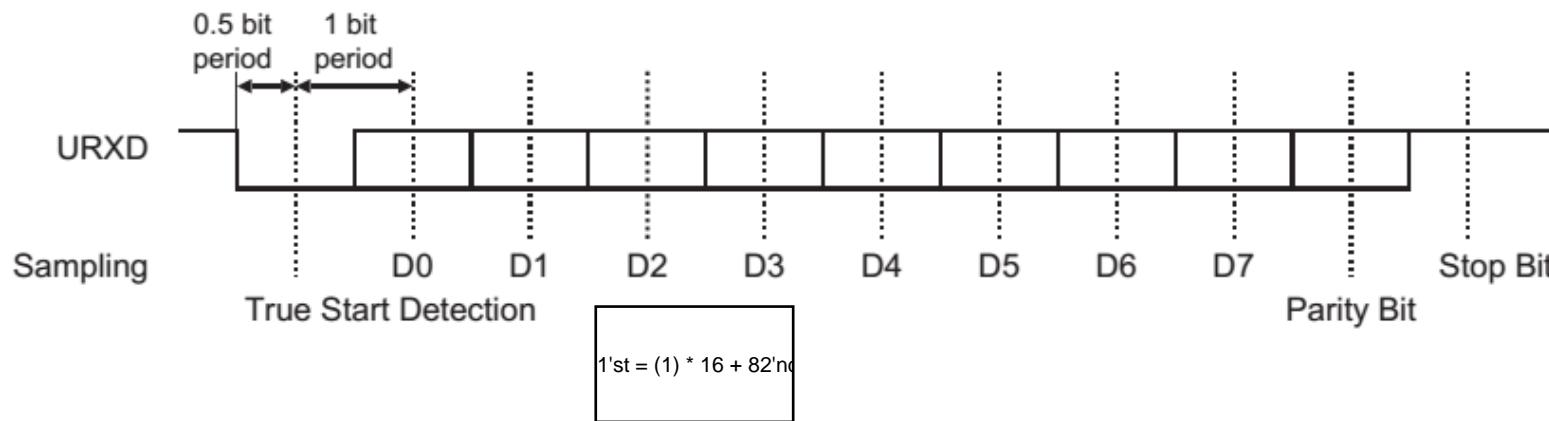
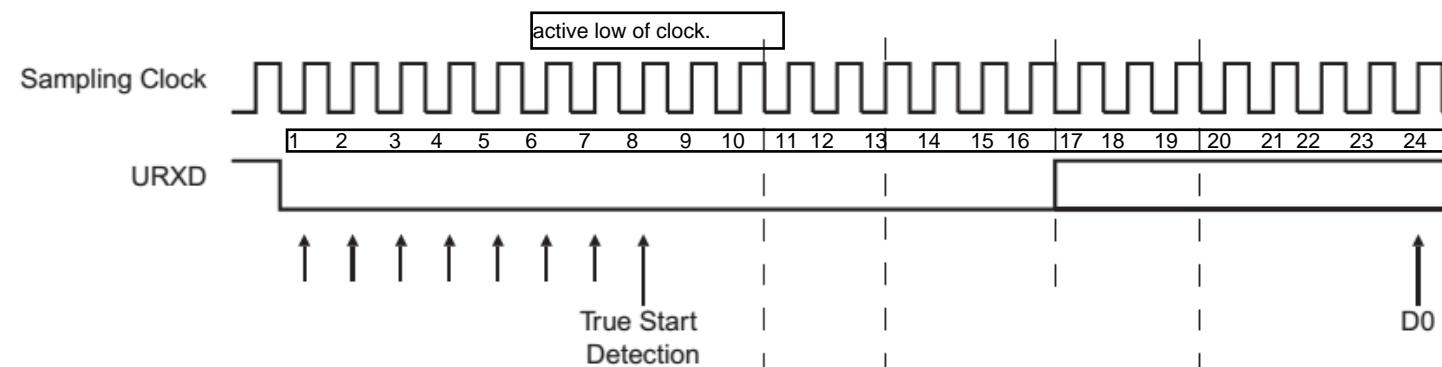
UART Receiver in SAM3X8E

- Receiver Reset, Enable and Disable
 - After device reset, the UART receiver is disabled
 - To enable: RXEN = 1 in the control register (UART_CR)
 - If enabled: the receiver starts looking for a start bit
 - To disable: RXDIS = 1 in the control register (UART_CR)
 - If the receiver is waiting for a start bit, it is immediately stopped
 - If the receiver has already detected a start bit and is receiving the data, it waits for the stop bit before actually stopping its operation
 - To reset: RSTRX =1 in the control register (UART_CR)
 - The receiver immediately stops its current operations and is disabled, whatever its current state!

Start Detection and Data Sampling

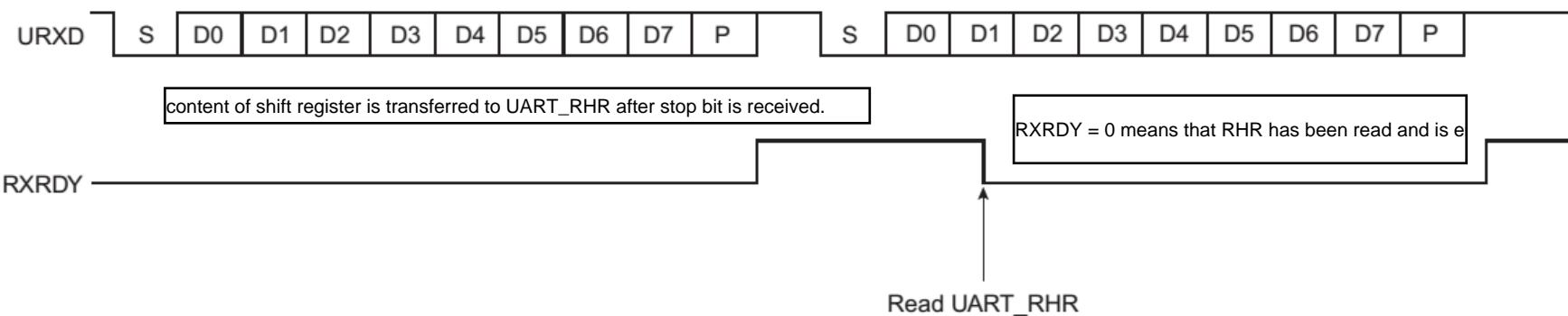
- A low level (space) on URXD is interpreted as a valid start bit
 - if it is detected for more than 7 cycles of the sampling clock
 - Sampling clock = $16 \times$ baud rate
- The receiver samples the URXD at the theoretical midpoint of each bit
 - When a valid start bit has been detected
 - It is assumed that each bit lasts 16 cycles of the sampling clock (1-bit period):
the bit sampling point is eight cycles (0.5-bit period) after the start of the bit
 - The first sampling point is therefore 24 cycles (1.5-bit periods) after the falling edge of the start bit was detected
16 bit takes to completely send the start bit, in sender.
 - Each subsequent bit is sampled 16 cycles (1-bit period) after the previous one

Start Detection and Data Sampling



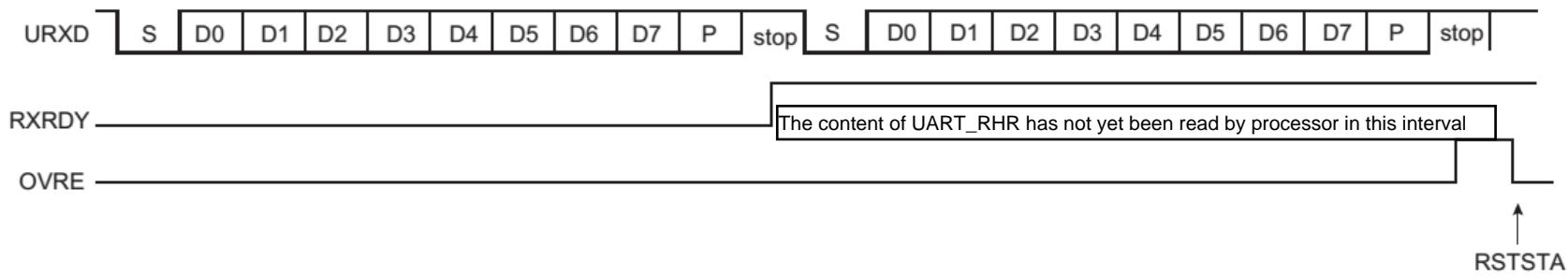
Receiver Ready

- When a complete character is received
 - Is transferred to the receive holding register (UART_RHR)
 - The RXRDY status bit in UART_SR (Status Register) is set
 - The bit RXRDY is automatically cleared when the UART_RHR is read



Receiver Overrun

- If **UART_RHR** has not been read by the software (or the Peripheral Data Controller or DMA Controller) since the last transfer
 - and a new character is received
 - the **OVRE** status bit in **UART_SR** is set
- **OVRE** is cleared when the software writes the control register **UART_CR** with the bit **RSTSTA** (Reset Status) at 1



Transmitter

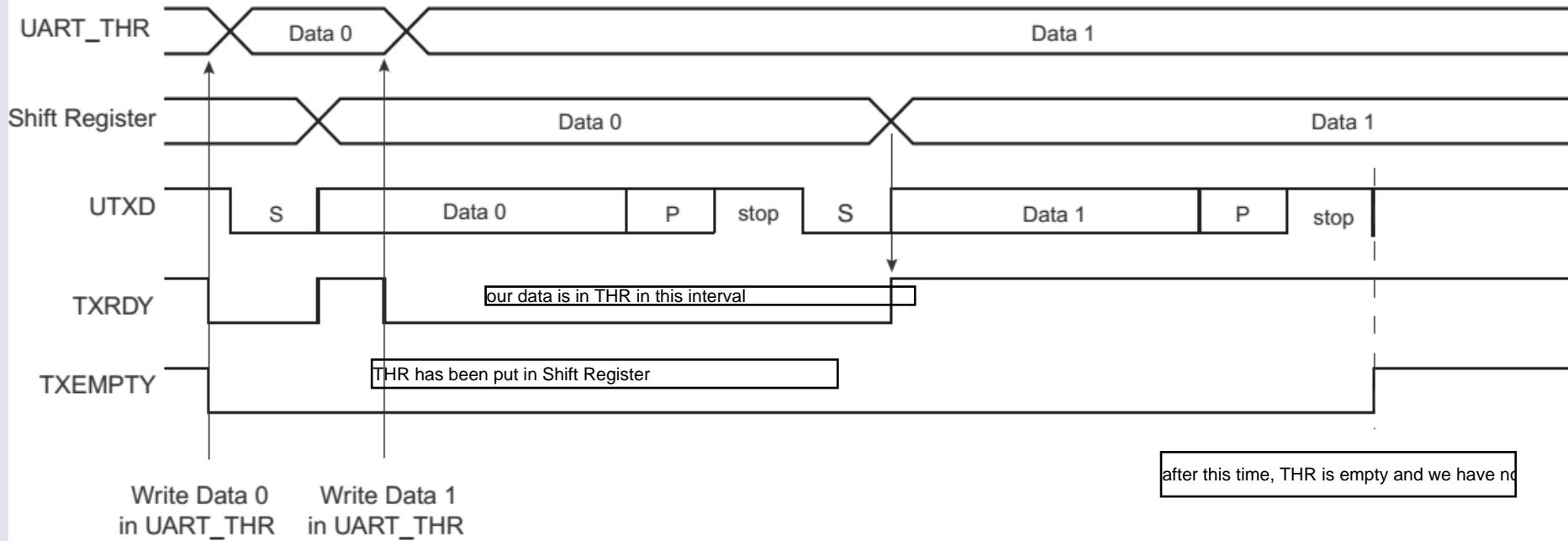
- The UART transmitter is disabled after device reset
 - To enable: TXEN = 1 in the control register **UART_CR**
 - The transmitter waits for a character to be written in the **Transmit Holding Register (UART_THR)** before actually starting the transmission
 - To disable: TXDIS = 1 in the control register **UART_CR**
 - If the transmitter is not operating, it is immediately stopped
 - If a character is being processed into the Shift Register and/or a character has been written in the **Transmit Holding Register**, the characters are completed before the transmitter is stopped
 - To reset: RSTTX = 1 in the **UART_CR**

Transmitter Control

- When enabled, the bit TXRDY is set in the UART_SR
- The transmission starts when
 - The programmer writes in the UART_THR and
 - The written character is transferred from UART_THR to the Shift Register
 - The TXRDY bit remains high until a second character is written in UART_THR
- As soon as the first character is completed
 - The last character written in UART_THR is transferred into the shift register and TXRDY rises again
 - Showing that the holding register is empty
- When both the Shift Register and UART_THR are empty
 - All the characters written in UART_THR have been processed
 - The TXEMPTY bit rises after the last stop bit has been completed

TXRDY = 1 --> UART_THR is empty

Transmitter Control



when content of THR is empty and shift register's content are sent, the TXEMPTY becomes 1

UART Registers

- **UART Control Register: UART_CR**

- **RSTRX: Reset Receiver**
- **RSTTX: Reset Transmitter**
- **RXEN: Receiver Enable**
- **RXDIS: Receiver Disable**
- **TXEN: Transmitter Enable**
- **TXDIS: Transmitter Disable**
- **RSTSTA: Reset Status Bits** (Resets the status bits PARE, FRAME and OVRE in the UART_SR)

| | | | | | | | |
|-------|------|-------|------|-------|-------|----|--------|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| - | - | - | - | - | - | - | - |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| - | - | - | - | - | - | - | - |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| - | - | - | - | - | - | - | RSTSTA |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| TXDIS | TXEN | RXDIS | RXEN | RSTTX | RSTRX | - | - |

Ch. 34: Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series, Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15

UART Registers

- **UART Mode Register: UART_MR**

| | | | | | | | |
|--------|----|----|----|-----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| - | - | - | - | - | - | - | - |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| - | - | - | - | - | - | - | - |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| CHMODE | - | - | - | PAR | - | - | - |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| - | - | - | - | - | - | - | - |

| Value | Name | Description |
|-------|-------|---------------------------|
| 0 | EVEN | Even parity |
| 1 | ODD | Odd parity |
| 2 | SPACE | Space: parity forced to 0 |
| 3 | MARK | Mark: parity forced to 1 |
| 4 | NO | No parity |

UART Registers

- **UART Interrupt Enable Register: UART_IER**

- RXRDY: Enable RXRDY Interrupt
- TXRDY: Enable TXRDY Interrupt
- ENDRX: Enable End of Receive Transfer Interrupt
- ENDTX: Enable End of Transmit Interrupt
- OVRE: Enable Overrun Error Interrupt
- FRAME: Enable Framing Error Interrupt
- PARE: Enable Parity Error Interrupt
- TXEMPTY: Enable TXEMPTY Interrupt
- TXBUFE: Enable Buffer Empty Interrupt
- RXBUFF: Enable Buffer Full Interrupt

UART Interrupt Disable Register

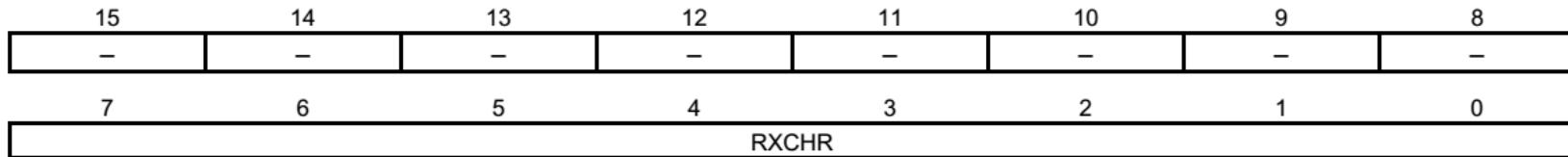
UART Interrupt Mask Register

UART Status Register

| | | | | | | | |
|------|-------|------|--------|--------|----|---------|-------|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| — | — | — | RXBUFF | TXBUFE | — | TXEMPTY | — |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| PARE | FRAME | OVRE | ENDTX | ENDRX | — | TXRDY | RXRDY |

UART Registers

- **UART Receiver Holding Register: UART_RHR**
 - Read-only
 - RXCHR: Received Character
 - Last received character if RXRDY is set
- **UART Transmit Holding Register: UART_THR**
 - Write-only
 - TXCHR: Character to be Transmitted
 - Next character to be transmitted after the current character if TXRDY is not set



UART Registers

- **UART Baud Rate Generator Register: UART_BRGR**

- CD: Clock Divisor
- 0 = Baud Rate Clock is disabled
- **1 to 65,535 = MCK / (CD x 16)**

value of CD can be 1 to 65,536

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 |
| - | - | - | - | - | - | - | - |
| 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
| - | - | - | - | - | - | - | - |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 |
| CD | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| CD | | | | | | | |

Ch. 34: Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series, Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15

The End (for now)!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 10

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series,
[Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15](#)

Universal Synchronous Asynchronous Receiver/Transmitter (USART)

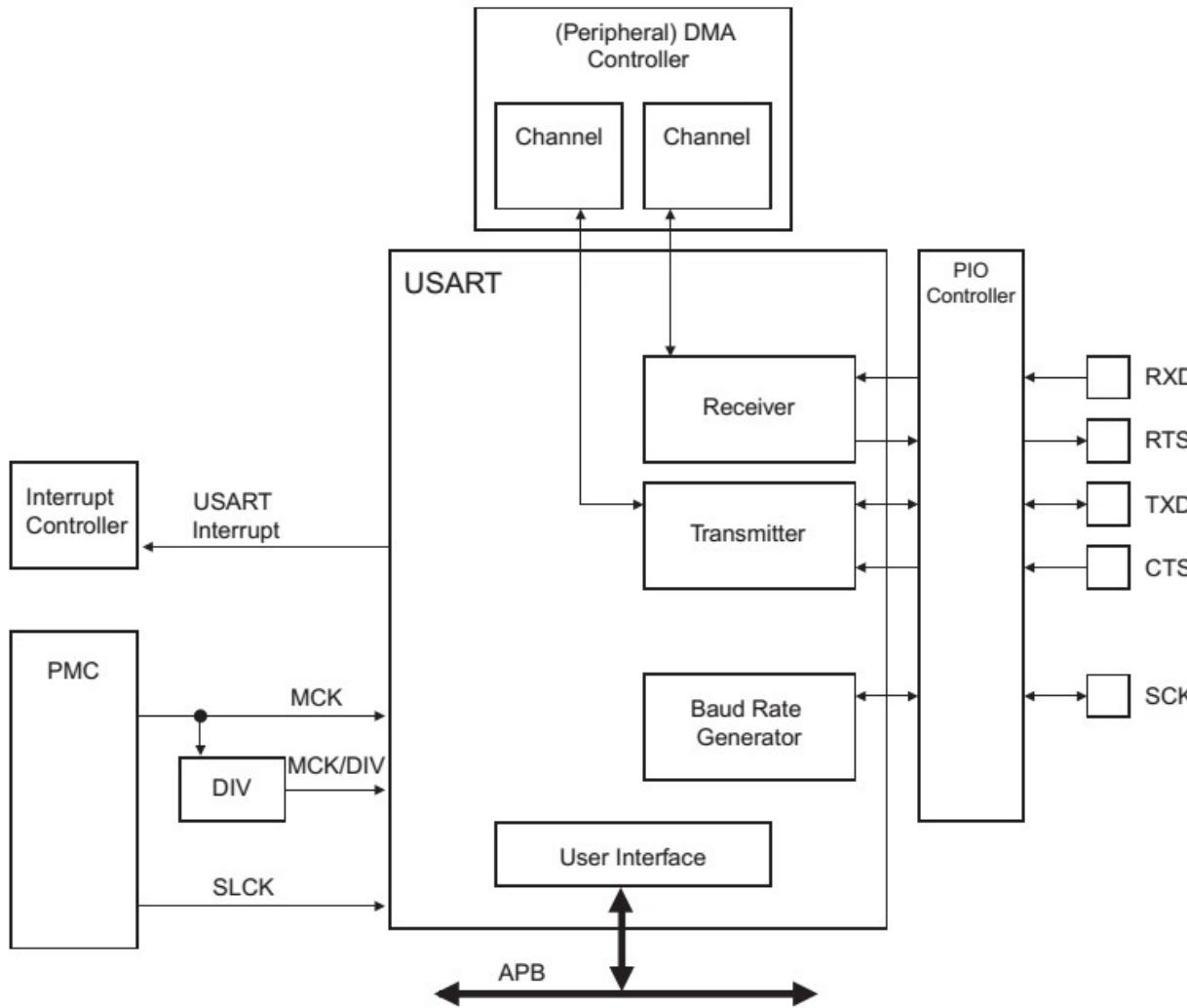
cannot be synchronous and asynchronous simultaneously.

USART Characteristics

UART is also full-duplex

- **5- to 9-bit Full-duplex Synchronous or Asynchronous Communications**
- **1, 1.5 or 2 Stop Bits in Asynchronous Mode** in UART we don't have 1.5
- **1 or 2 Stop Bits in Synchronous Mode**
- **Configurable MSB- or LSB-first** in UART LSB is first.
- **By 8 or by 16 Over-sampling Receiver Frequency**
- **Optional Hardware Handshaking RTS-CTS**
- **RS485 with Driver Control Signal**
- **SPI Mode**
- **LIN Mode (USART0 only)**

USART Block Diagram



I/O Lines Description

| Name | Description | Type | Active Level |
|------|---|--------|--------------|
| SCK | Serial Clock | I/O | |
| TXD | Transmit Serial Data or Master Out Slave In (MOSI) in SPI Master Mode or Master In Slave Out (MISO) in SPI Slave Mode | I/O | |
| RXD | Receive Serial Data or Master In Slave Out (MISO) in SPI Master Mode or Master Out Slave In (MOSI) in SPI Slave Mode | Input | |
| CTS | Clear to Send or Slave Select (NSS) in SPI Slave Mode | Input | Low |
| RTS | Request to Send or Slave Select (NSS) in SPI Master Mode | Output | Low |

transmitter RTS connects to receiver CTS and vice versa.

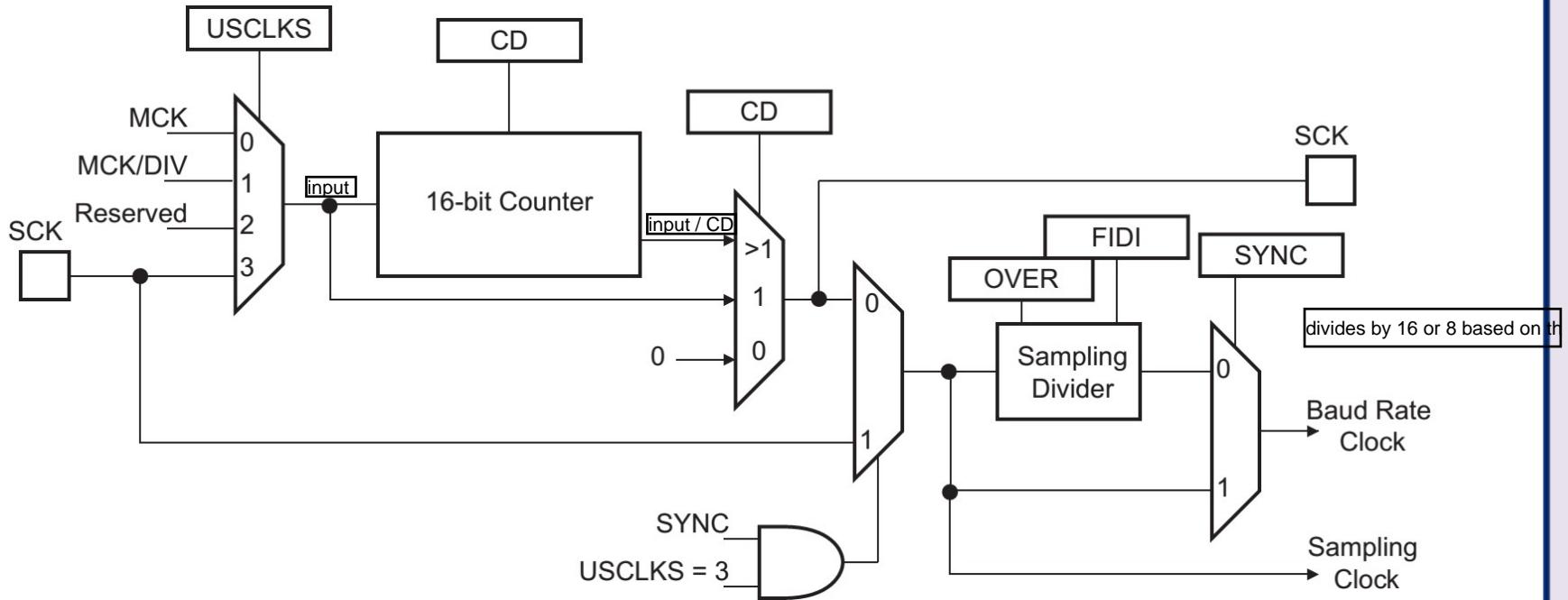
I/O Lines Description

| Instance | Signal | I/O Line | Peripheral |
|----------|--------|----------|------------|
| USART0 | CTS0 | PB26 | A |
| USART0 | RTS0 | PB25 | A |
| USART0 | RXD0 | PA10 | A |
| USART0 | SCK0 | PA17 | B |
| USART0 | TXD0 | PA11 | A |
| USART1 | CTS1 | PA15 | A |
| USART1 | RTS1 | PA14 | A |
| USART1 | RXD1 | PA12 | A |
| USART1 | SCK1 | PA16 | A |
| USART1 | TXD1 | PA13 | A |
| USART2 | CTS2 | PB23 | A |
| USART2 | RTS2 | PB22 | A |
| USART2 | RXD2 | PB21 | A |
| USART2 | SCK2 | PB24 | A |
| USART2 | TXD2 | PB20 | A |
| USART3 | CTS3 | PF4 | A |
| USART3 | RTS3 | PF5 | A |
| USART3 | RXD3 | PD5 | B |
| USART3 | SCK3 | PE16 | B |
| USART3 | TXD3 | PD4 | B |

Interrupt Peripheral IDs

| Instance | ID |
|----------|----|
| USART0 | 17 |
| USART1 | 18 |
| USART2 | 19 |
| USART3 | 20 |

Baud Rate Generator



If selected, the frequency of the signal provided on SCK must be at least 3 times lower than MCK in USART mode, or 6 in SPI mode

Baud Rate

Asynchronous Mode

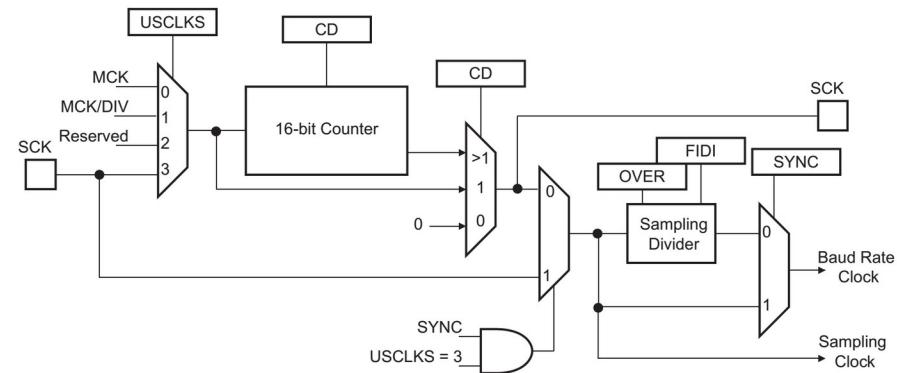
$$\text{Baudrate} = \frac{\text{SelectedClock}}{(8(2 - \text{Over})\text{CD})}$$

Fractional Baud Rate: Program the FP field in the US_BRGR

$$\text{Baudrate} = \frac{\text{SelectedClock}}{\left(8(2 - \text{Over})\left(\text{CD} + \frac{\text{FP}}{8}\right)\right)}$$

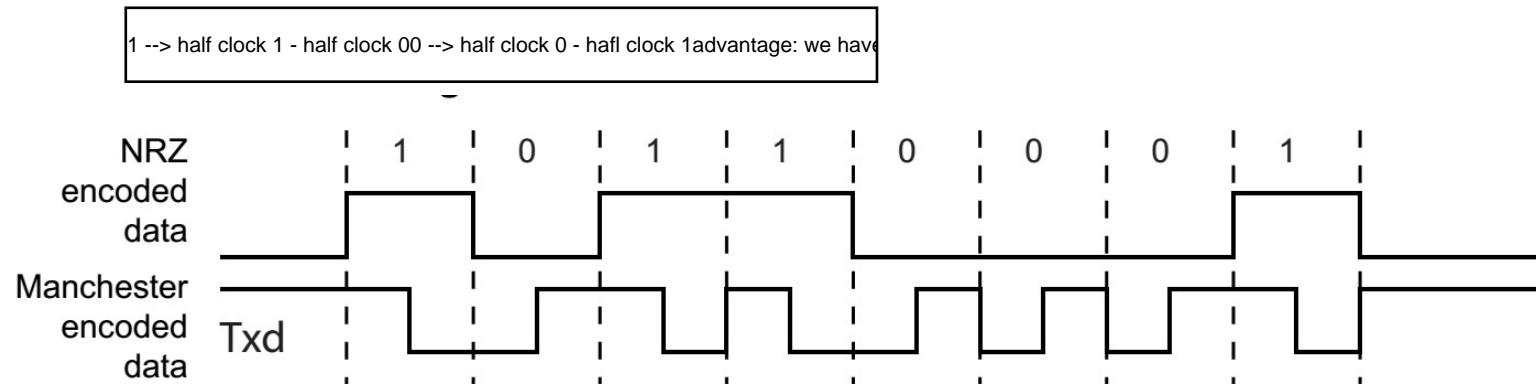
Synchronous Mode

$$\text{BaudRate} = \frac{\text{SelectedClock}}{\text{CD}}$$



Data Transmission

- The transmitter performs the same in synchronous and asynchronous modes
 - One start bit, up to 9 data bits, one optional parity bit and up to two stop bits are successively shifted out on the TXD pin at each falling edge of the programmed serial clock
- Manchester mode
 - To enable this mode, MAN=1 field in the US_MR



Data Transmission

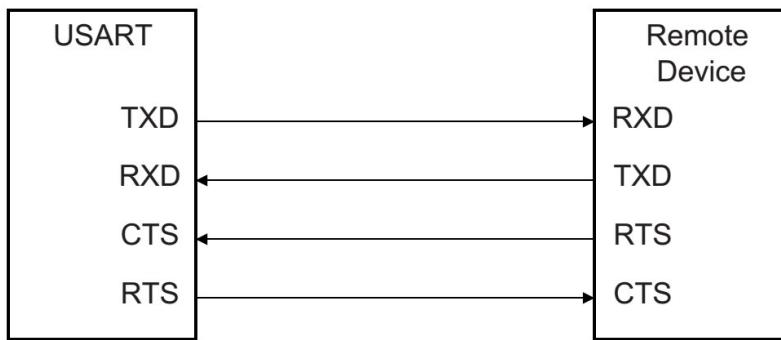
- **Asynchronous Receiver**
 - **Oversampling either 16 or 8 times the Baud Rate clock**
 - Depending on the OVER bit in the US_MR
- **Synchronous Receiver**
 - **The receiver samples on each rising edge of the Baud Rate Clock**
 - **If a low level is detected, it is considered as a start**
 - **Synchronous mode operations provide a high speed transfer capability**

synchronous mode speed is 8 or 16 times faster than asynchronous.

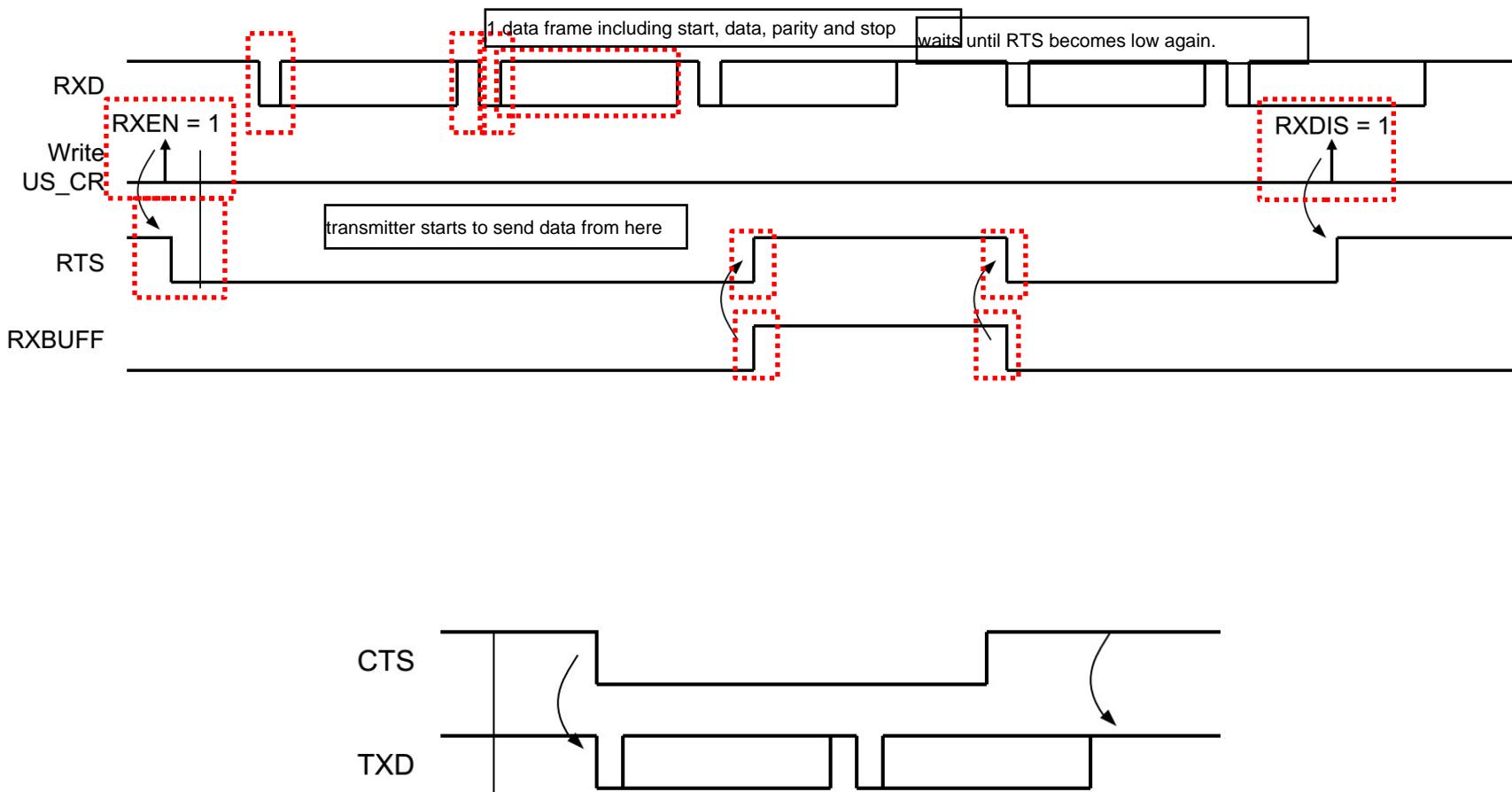
Hardware Handshaking

- Receiver operation
 - The RTS (Request To Send) pin is driven high if
 - The receiver is disabled
 - The status RXBUFF (Receive Buffer Full) is high
 - The remote device does not start transmitting while its CTS pin (driven by RTS) is high
 - When the receiver is enabled, the RTS falls
 - Indicating to the remote device that it can start transmitting

RTS is active low



Hardware Handshaking



The End (for now)!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 11

Copyright Notice

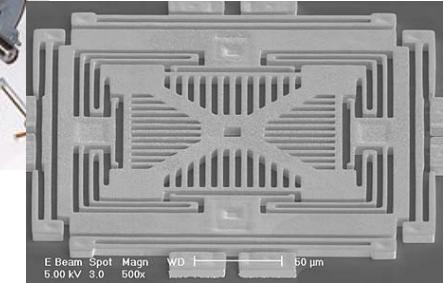
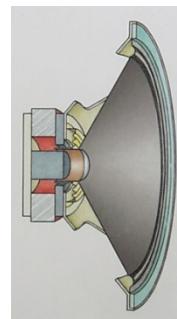
Parts (text & figures) of this lecture are adopted from:

- Prabal Dutta, “EECS 373 Design of Microprocessor-Based Systems,”
University of Michigan
- Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series,
[Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15](#)

Sampling, ADCs, and DACs

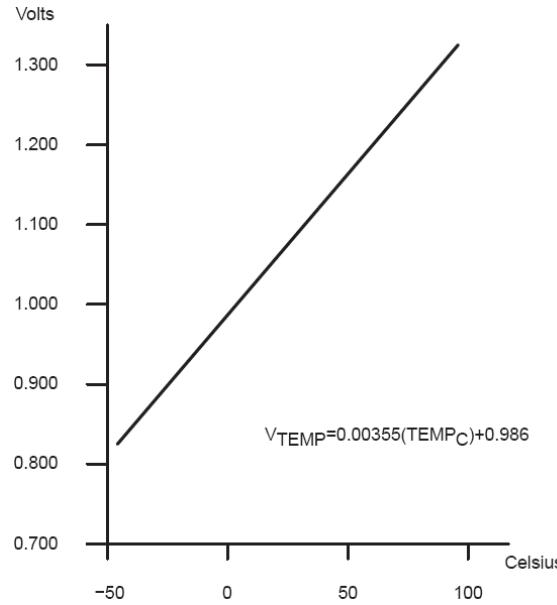
We live in an analog world

- Everything in the physical world is an analog signal
 - Sound, light, temperature, pressure
- Need to convert into electrical signals
 - Transducers: converts one type of energy to another
 - Electro-mechanical, Photonic, Electrical, ...
 - Examples
 - Microphone/speaker
 - Thermocouples
 - Accelerometers



Transducers

- Transducers convert one form of energy into another
 - Allow us to convert physical phenomena to a voltage potential in a well-defined way



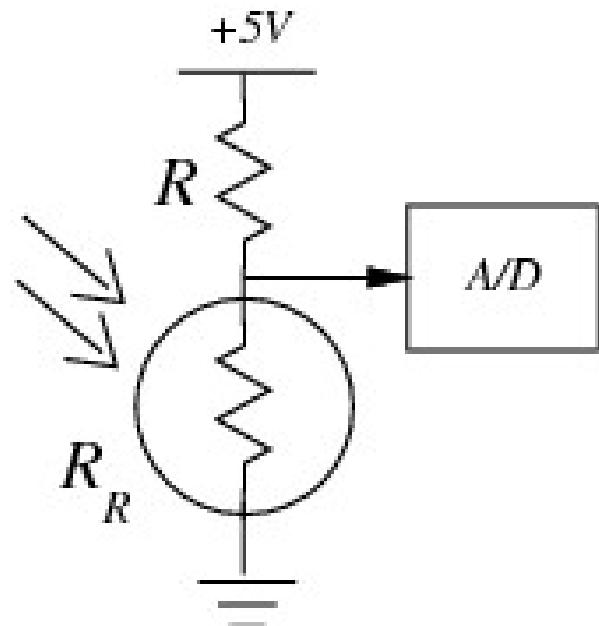
A transducer is a device that converts one type of energy to another. The conversion can be to/from electrical, electro-mechanical, electromagnetic, photonic, photovoltaic, or any other form of energy. While the term transducer commonly implies use as a sensor/detector, any device which converts energy can be considered a transducer. [\[Wikipedia\]](#)

An Example: Photocell

- Convert light to voltage with a CdS Photocell

$$V_{\text{signal}} = (+5V) R_R / (R + R_R)$$

- Choose $R=R_R$ at median of intended range
- Cadmium Sulfide (CdS)
- $t_{RC} = (R+R_R) * C_I$
 - Typically $R \sim 50-200k\Omega$
 - $C \sim 20pF$
 - So, $t_{RC} \sim 20-80\mu s$
 - $f_{RC} \sim 10-50kHz$

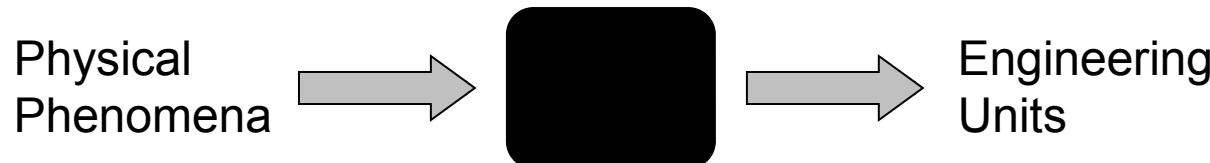


Many other common sensors

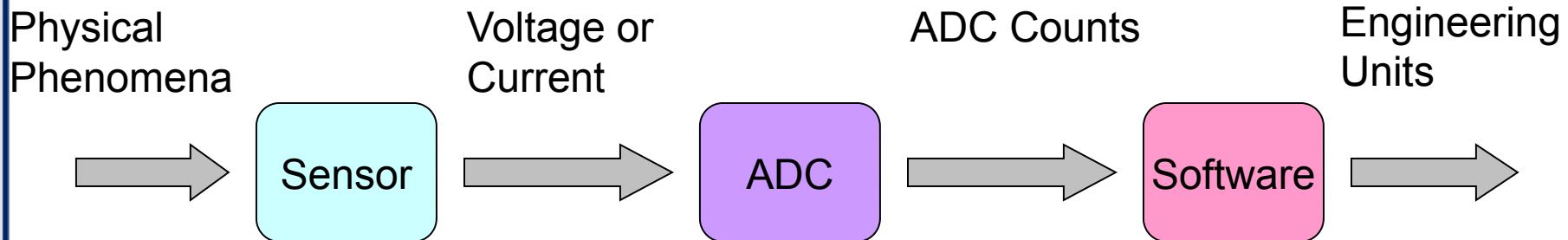
- Force
- Sound
 - Microphones
- Position
 - Gyros
- Acceleration
- Field
- ...

Going from Analog to Digital

- What we want

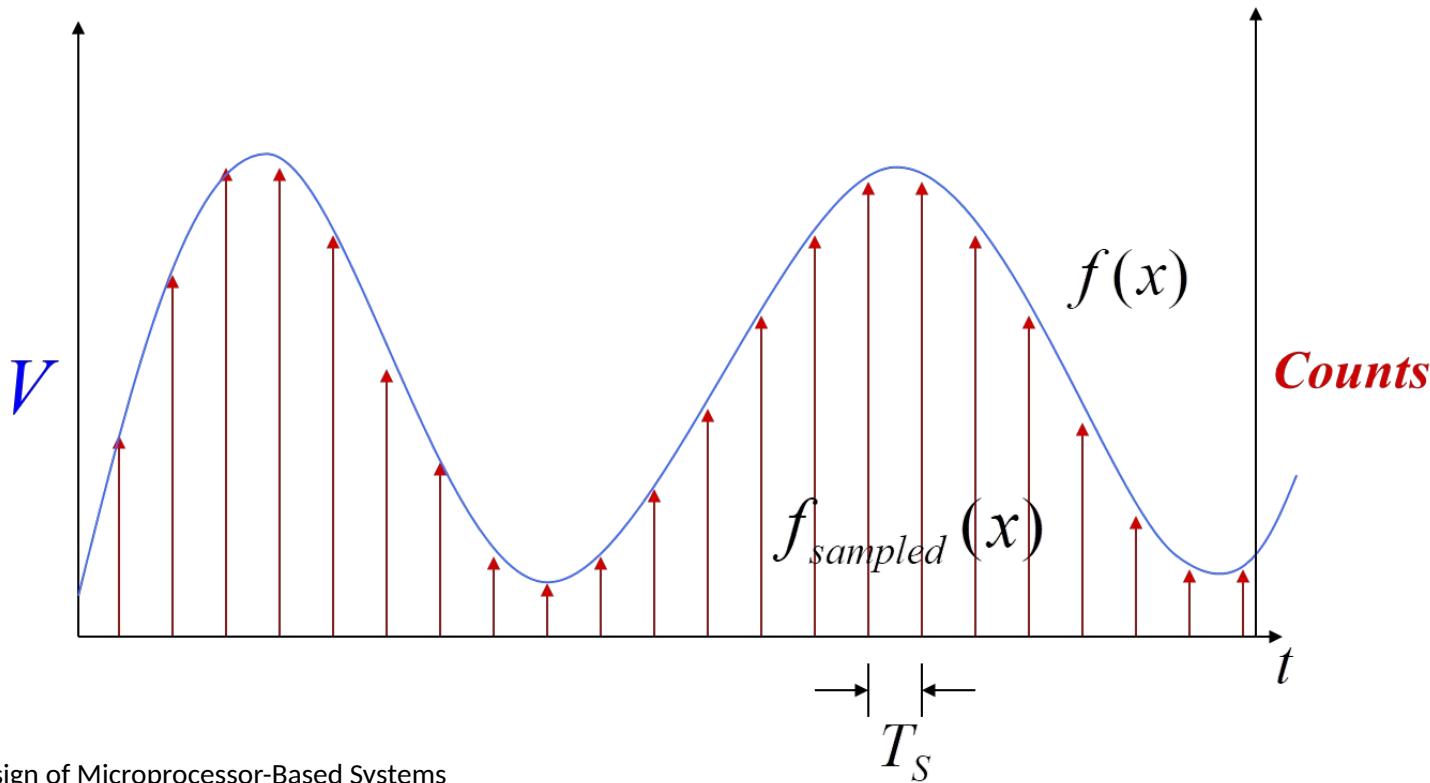


- How we have to get there



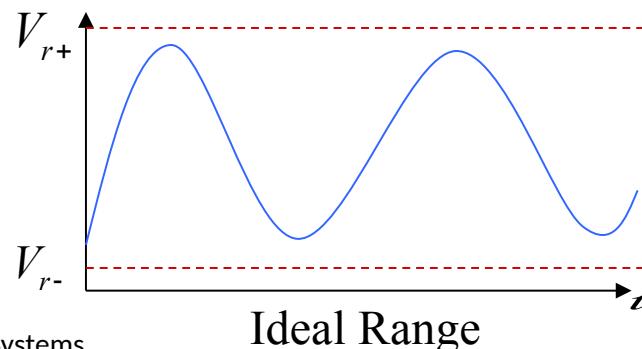
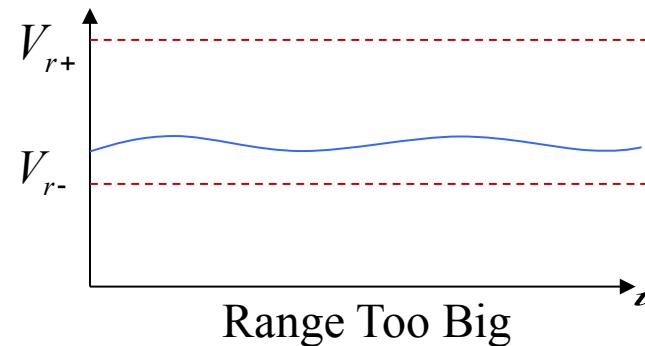
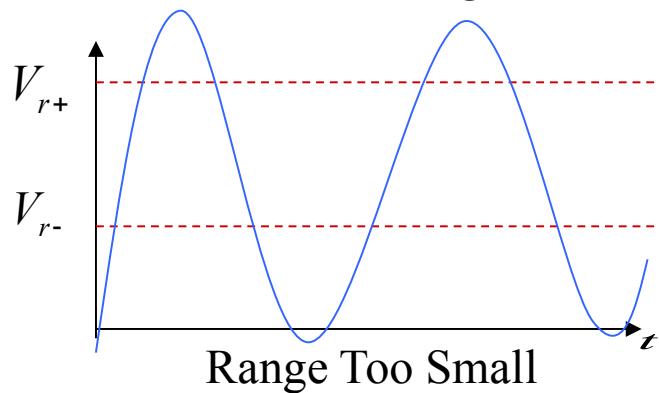
Representing an analog signal digitally

- How do we represent an analog signal?
 - As a time series of discrete values
 - On MCU: read the ADC data register periodically



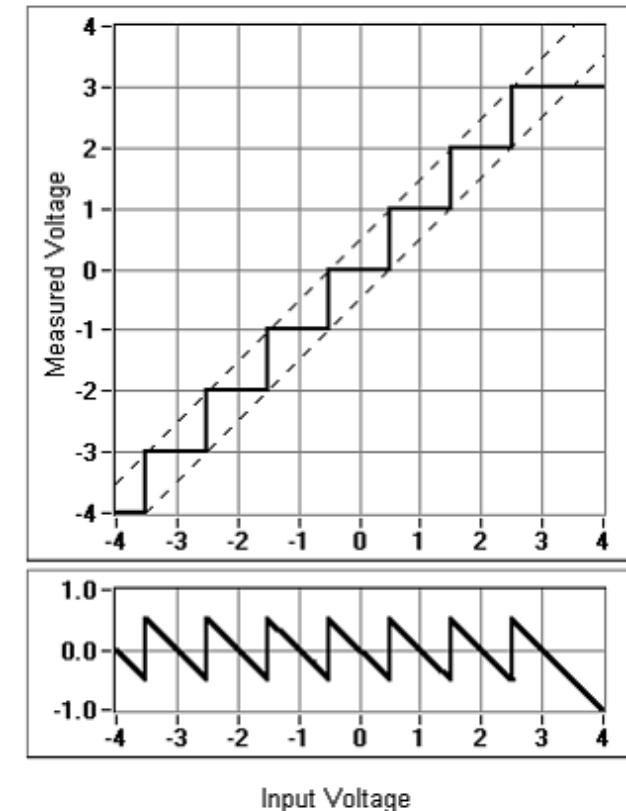
Choosing the horizontal range

- What do the sample values represent?
 - Some fraction within the range of values
 - What range to use?



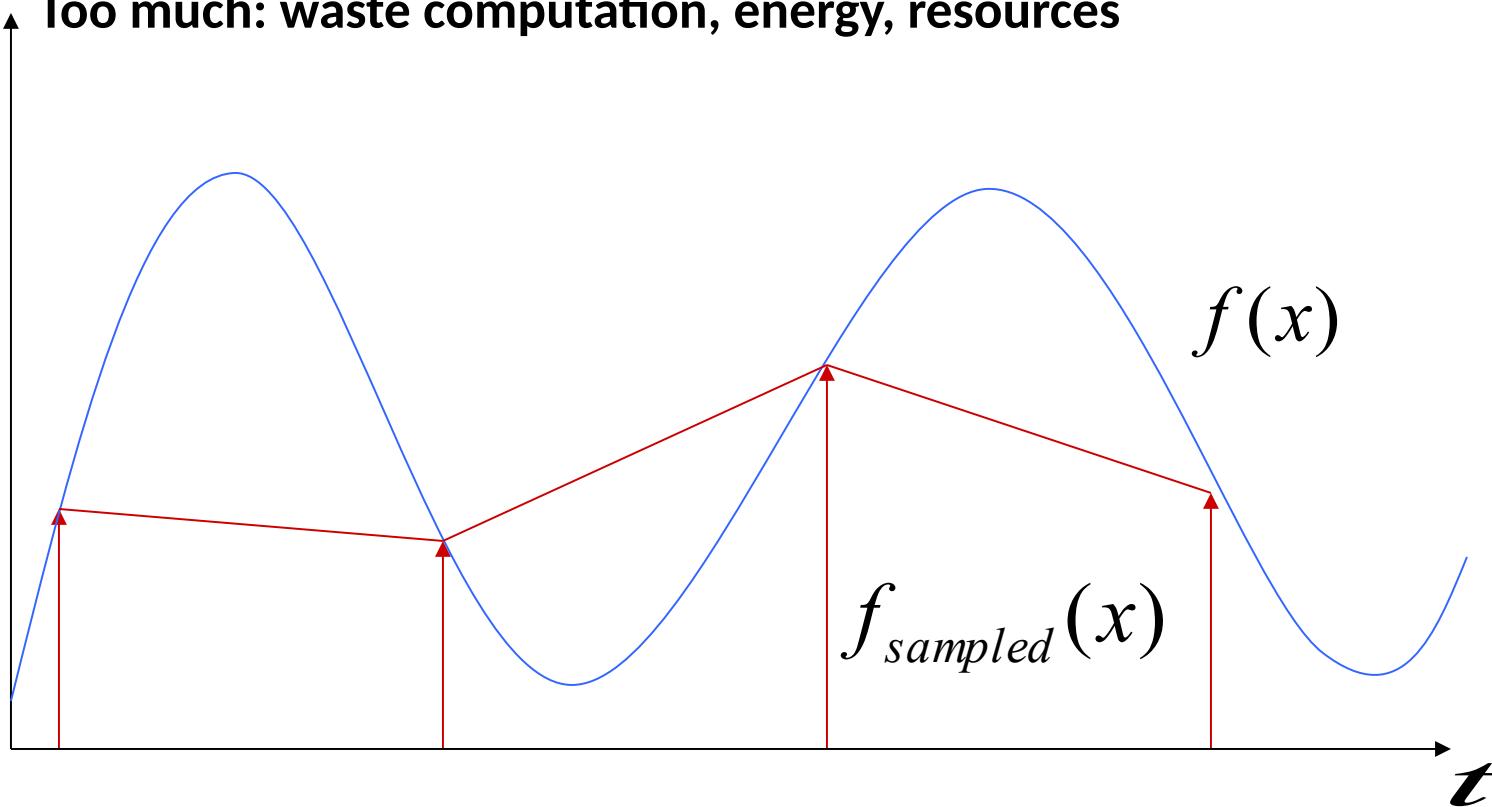
Choosing the horizontal granularity

- Resolution
 - Number of discrete values that represent a range of analog values
 - 12-bit ADC
 - 4096 values
 - Range / 4096 = Step
 - Quantization Error
 - How far off discrete value is from actual
 - $\frac{1}{2}$ LSB \Leftrightarrow Range / 8192
- Larger range \Leftrightarrow less information**
- Larger range \Leftrightarrow larger error**



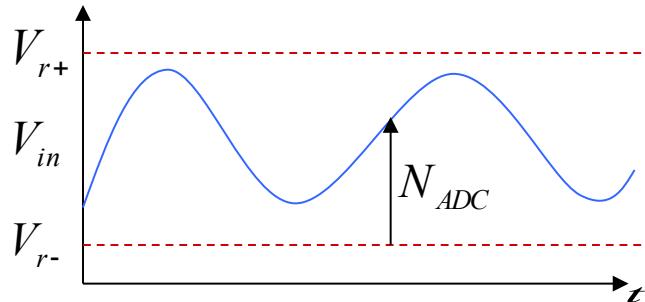
Choosing the sample rate

- What sample rate do we need?
 - Too little: we can't reconstruct the signal we care about
 - Too much: waste computation, energy, resources



Conversion: ADC

- Converting between voltages, ADC counts, and engineering units
 - Converting: ADC counts ☙ Voltage



$$N_{ADC} = 4095 \times \frac{V_{in} - V_{R-}}{V_{R+} - V_{R-}}$$
$$V_{in} = N_{ADC} \times \frac{V_{R+} - V_{R-}}{4095}$$

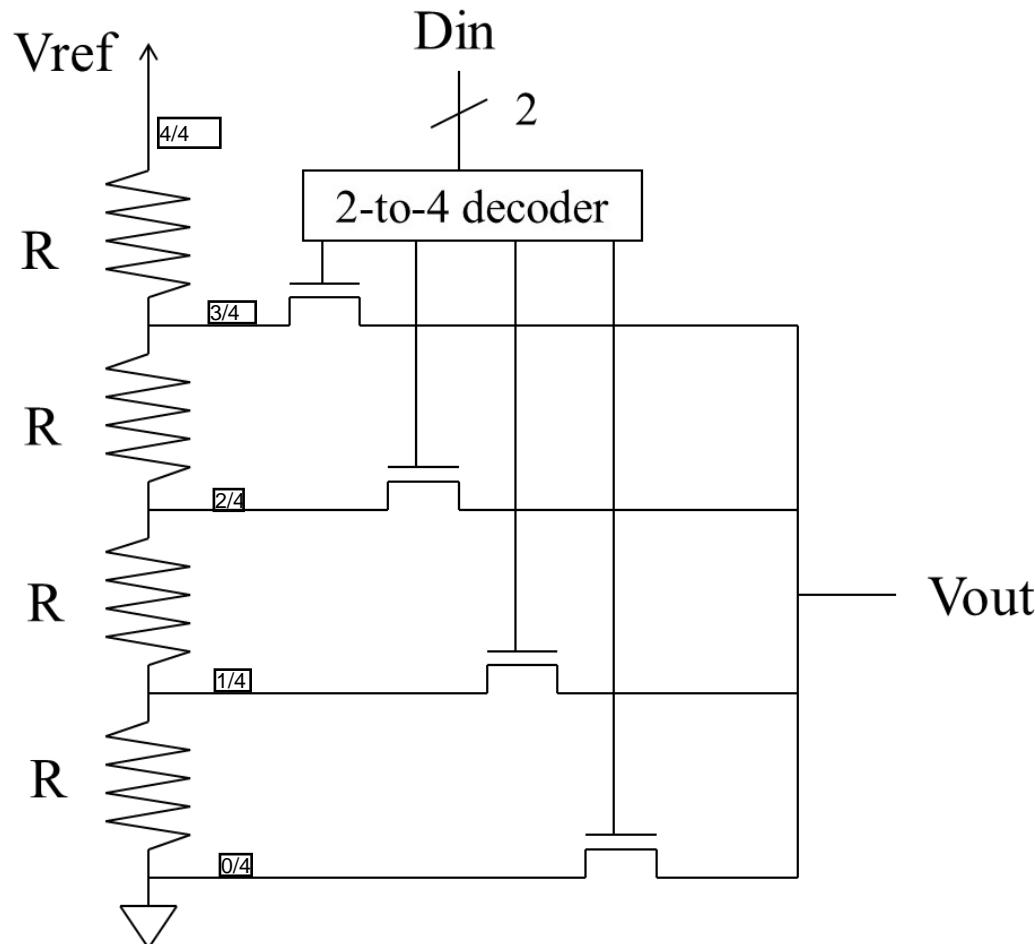
- Converting: Voltage ↗ Engineering Units

$$V_{TEMP} = 0.00355(TEMP_C) + 0.986$$

$$TEMP_C = \frac{V_{TEMP} - 0.986}{0.00355}$$

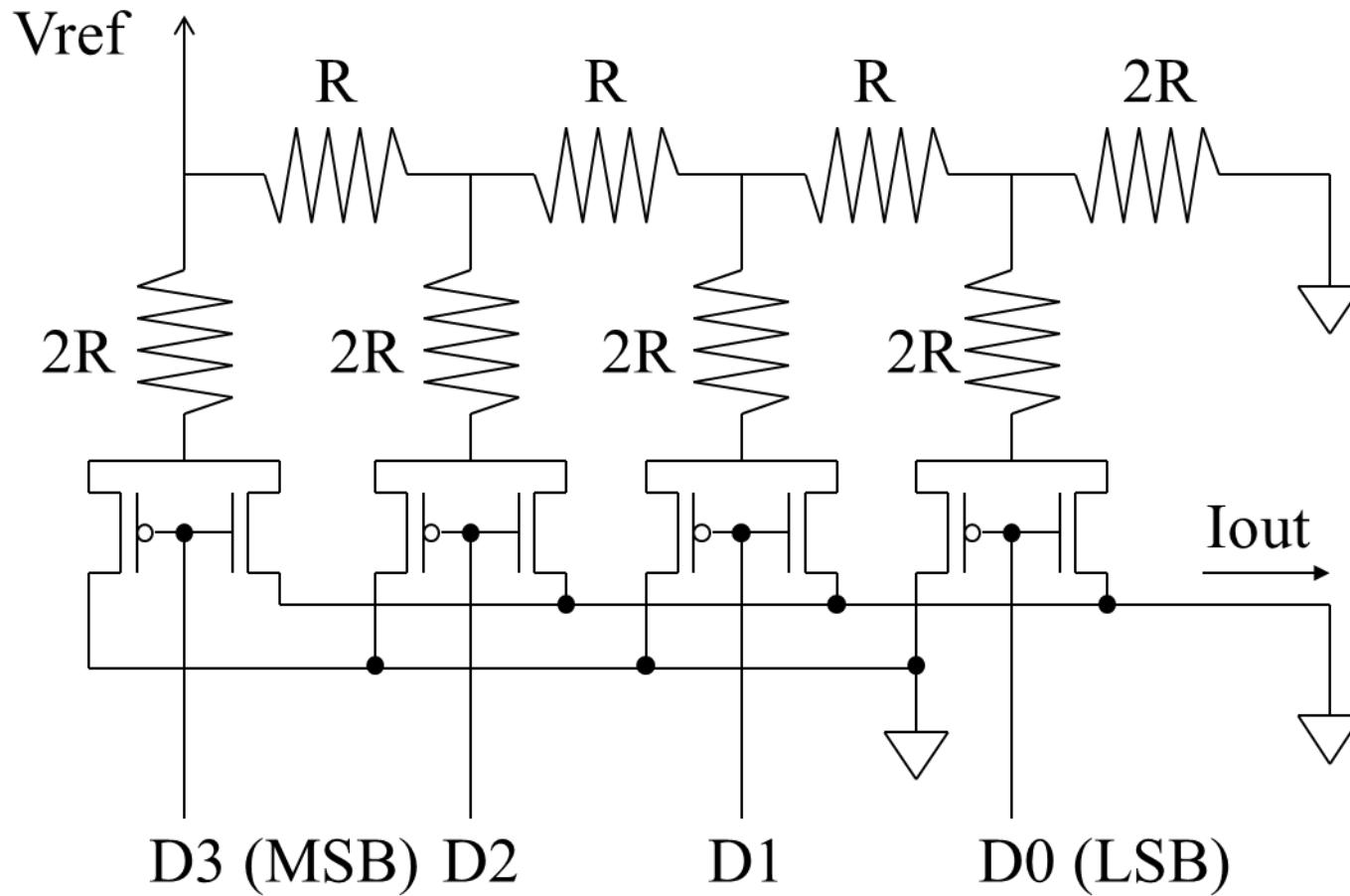
Conversion: DAC

- DAC 1: Voltage Divider



Conversion: DAC

- DAC 2: R/2R Ladder

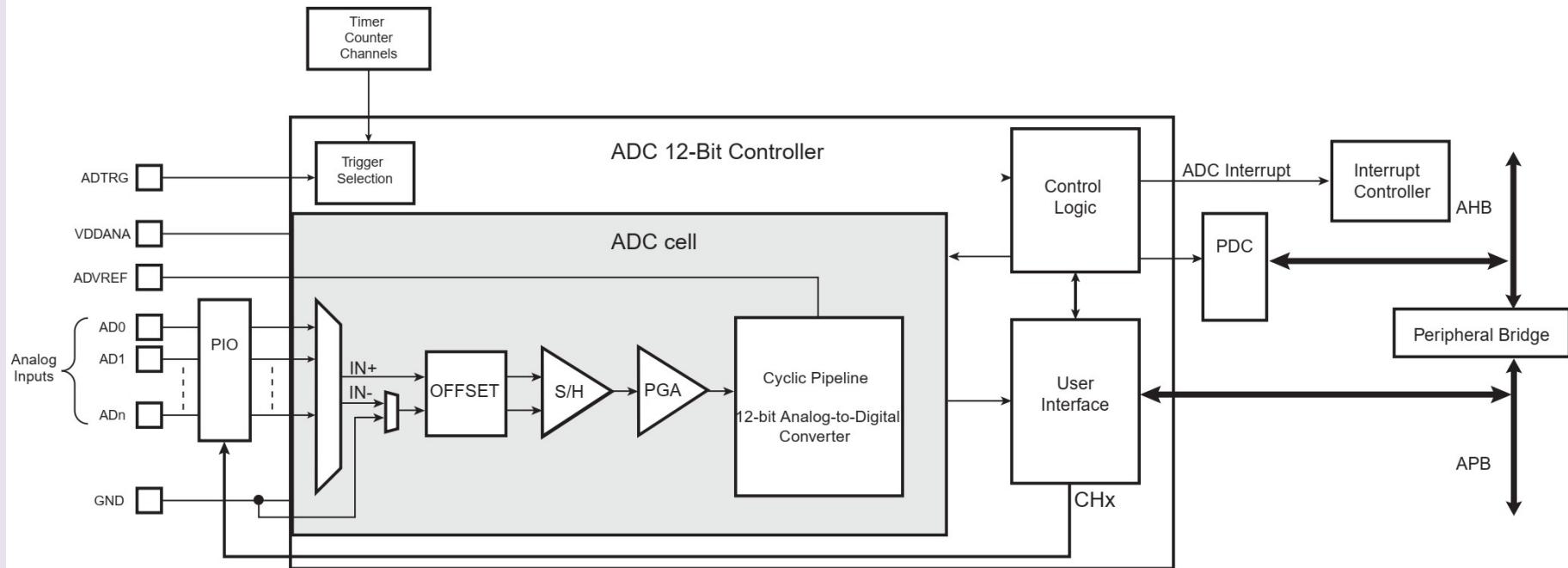


ADC in SAM3X

- A 16-to-1 analog multiplexer
 - Making possible the analog-to-digital conversions of 16 analog lines
 - The conversions extend from 0V to ADVREF
 - Supports an 10-bit or 12-bit resolution mode
 - Conversion results are reported in a common register for all channels
 - As well as in a channel-dedicated register
 - **1 MHz Conversion Rate**
 - **Individual Enable and Disable of Each Channel**

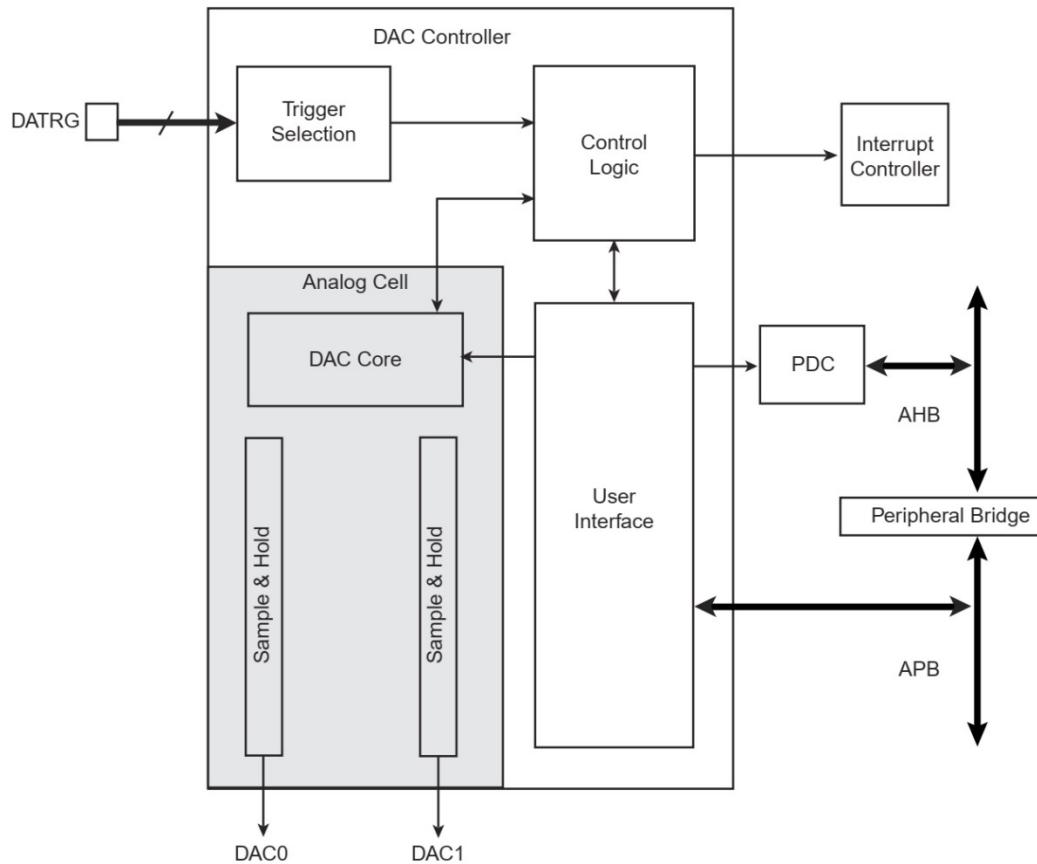
ADC in SAM3X

- Analog-to-Digital Converter Block Diagram
 - The temperature sensor is connected to Channel 15 of the ADC



DAC in SAM3X

- Digital-to-Analog Converter Controller (DACC)



The End!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 12

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- <https://www.analogictips.com/pulse-width-modulation-pwm/>
- Atmel | SMART ARM-based MCU DATASHEET, SAM3X / SAM3A Series,
Atmel-11057C-ATARM-SAM3X-SAM3A-Datasheet_23-Mar-15

Pulse Width Modulation (PWM)

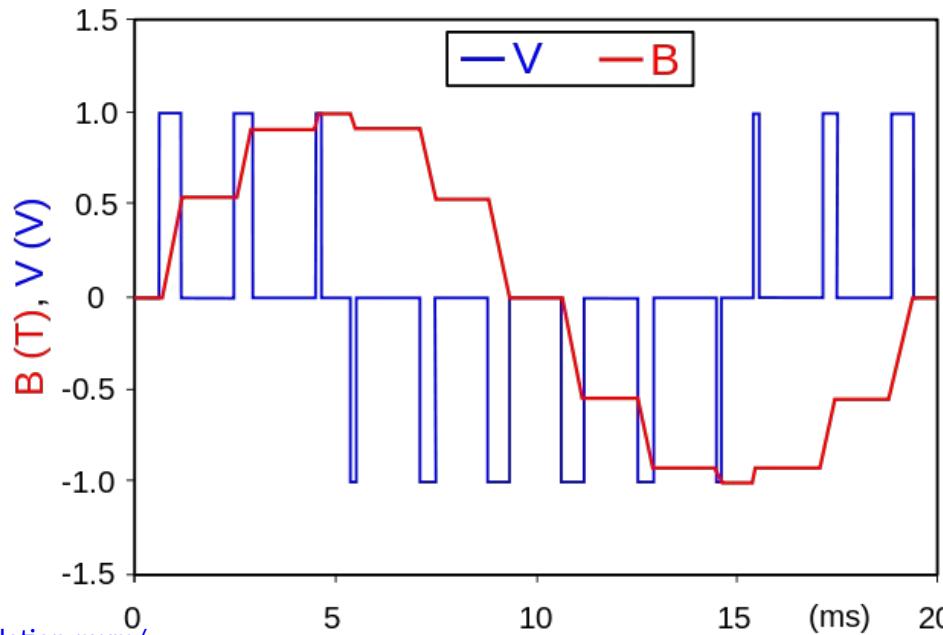
PWM: What is it and how does it work?

- Digital signals have two positions
 - on or off, interpreted in shorthand as 1 or 0
- Analog signals can be
 - on, off, half-way, two-thirds the way to on, and an infinite number of positions between 0 and 1
- To take an analog input signal (e.g., temperature) into a microcontroller
 - by using an analog-to-digital converter
 - But what about outputs?
- PWM is a way to control analog devices with a digital output
 - Drive analog devices like variable-speed motors, dimmable lights, actuators, and speakers

<https://www.analogictips.com/pulse-width-modulation-pwm/>

PWM: What is it and how does it work?

- PWM is not true analog output
 - PWM “fakes” an analog-like result by applying power in pulses, or short bursts of regulated voltage
 - Is a fancy term for describing a type of digital signal

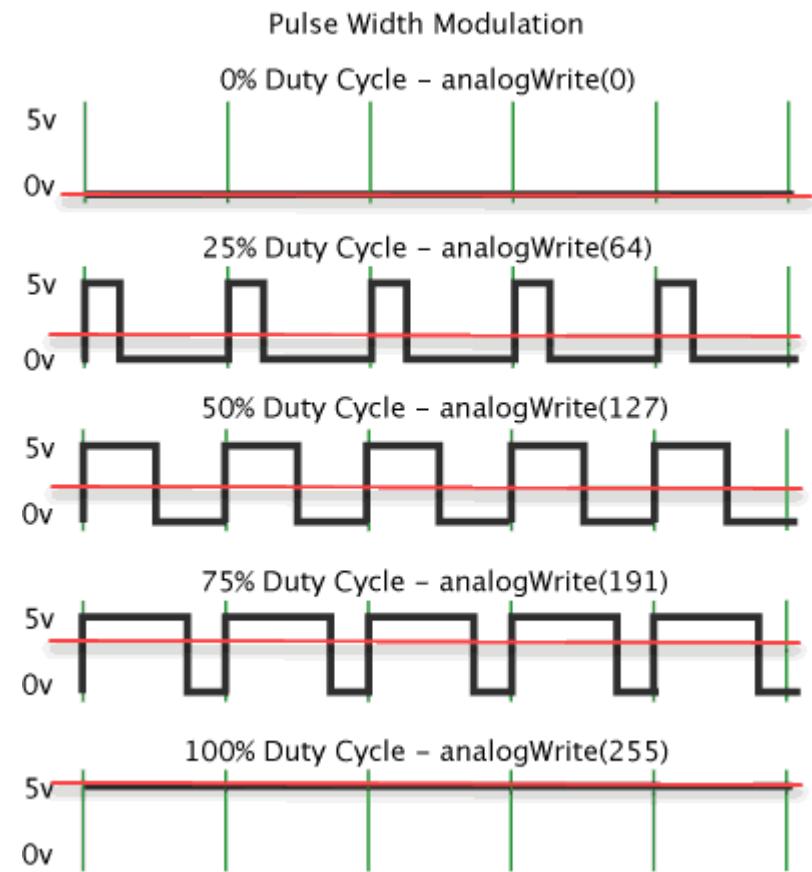


<https://www.analogictips.com/pulse-width-modulation-pwm/>

PWM: What is it and how does it work?

- A device that is driven by PWM ends up behaving like the average of the pulses
- The average voltage level can be a steady voltage or a moving target (dynamic/changing over time)

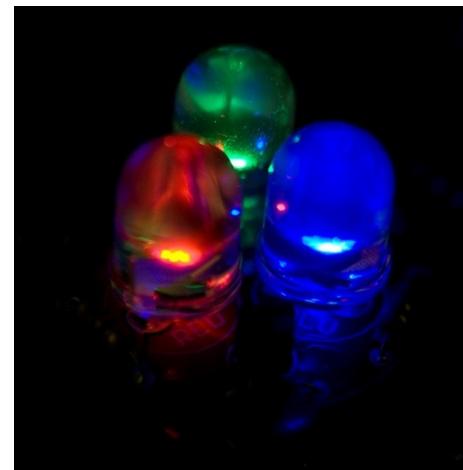
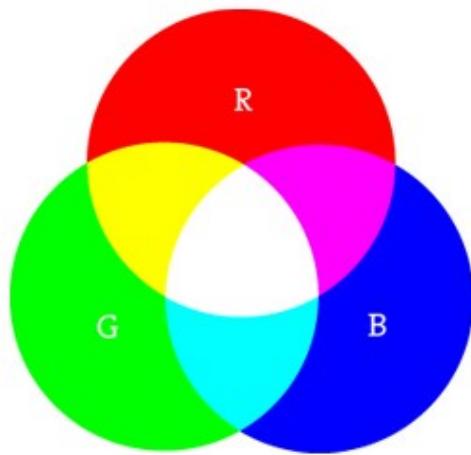
Average Voltage =
Duty Cycle x High Voltage Level



<https://www.analogictips.com/pulse-width-modulation-pwm/>

PWM: An Example

- Controlling the brightness of an LED by adjusting the duty cycle
- With an RGB (red green blue) LED,
 - you can control how much of each of the three colors you want in the mix of color by dimming them with various amounts



<https://www.analogictips.com/pulse-width-modulation-pwm/>

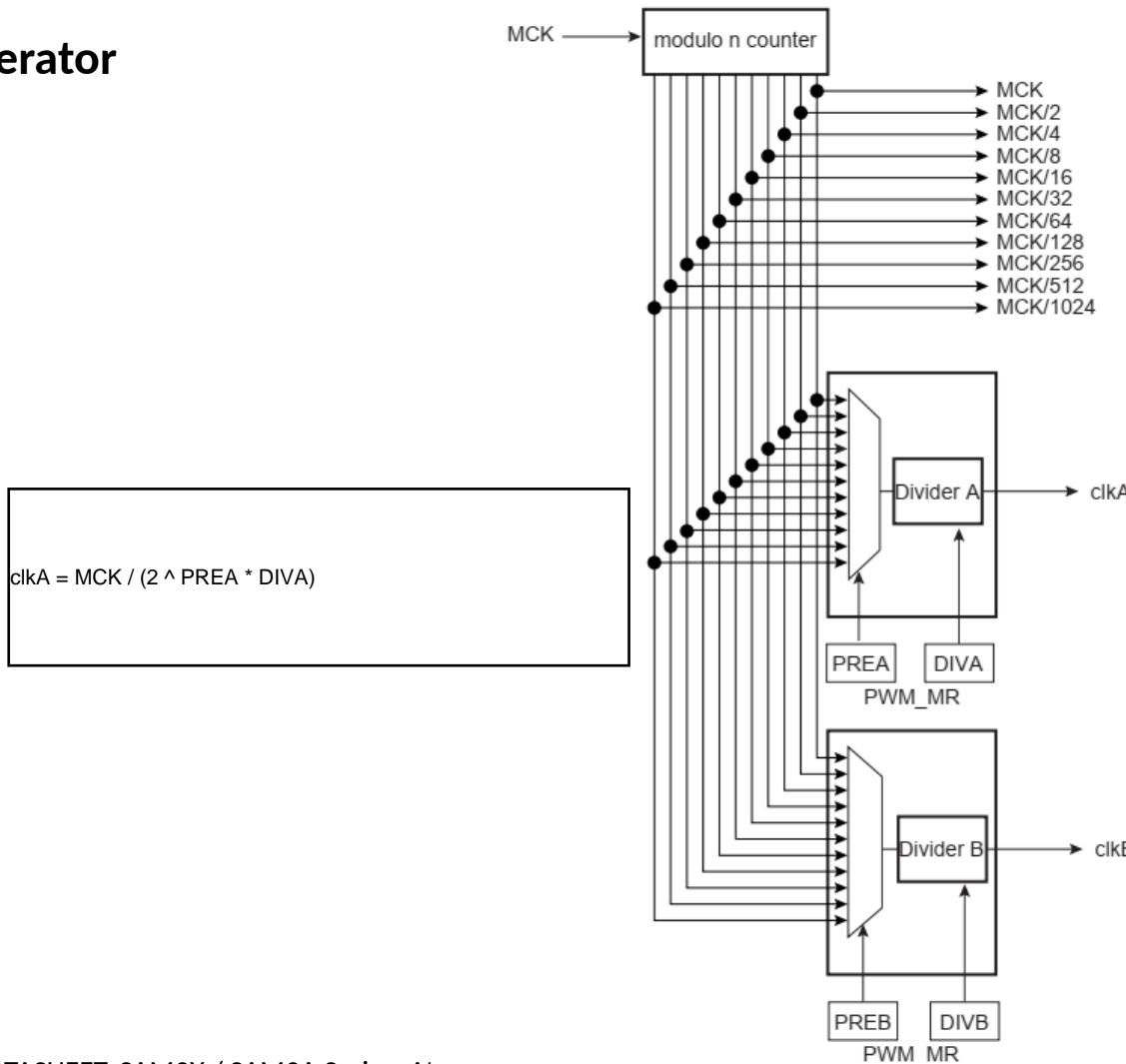
PWM in SAM3X

- The PWM macrocell controls 8 channels independently
 - Each channel controls two complementary square output waveforms
- Embedded Characteristics
 - Common Clock Generator Providing Thirteen Different Clocks
 - A Modulo n Counter Providing Eleven Clocks
 - Two Independent Linear Dividers Working on Modulo n Counter Outputs
 - Independent Clock Selection for Each Channel
 - Independent Period, Duty-Cycle and Dead-Time for Each Channel

dead time is the intervals that we don't want to pass duty cycle.

PWM in SAM3X

- PWM Clock Generator**



The End!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 13

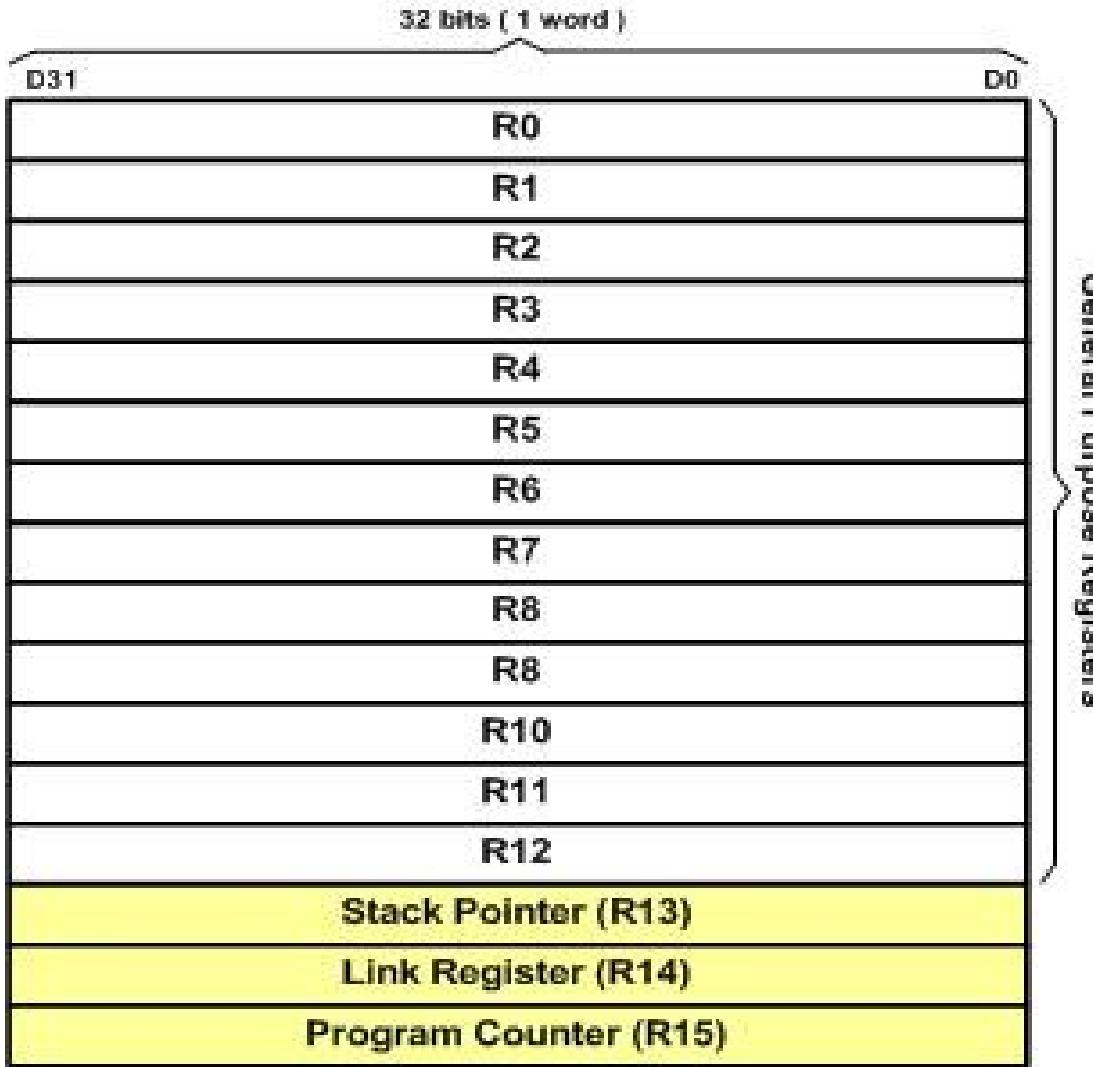
Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

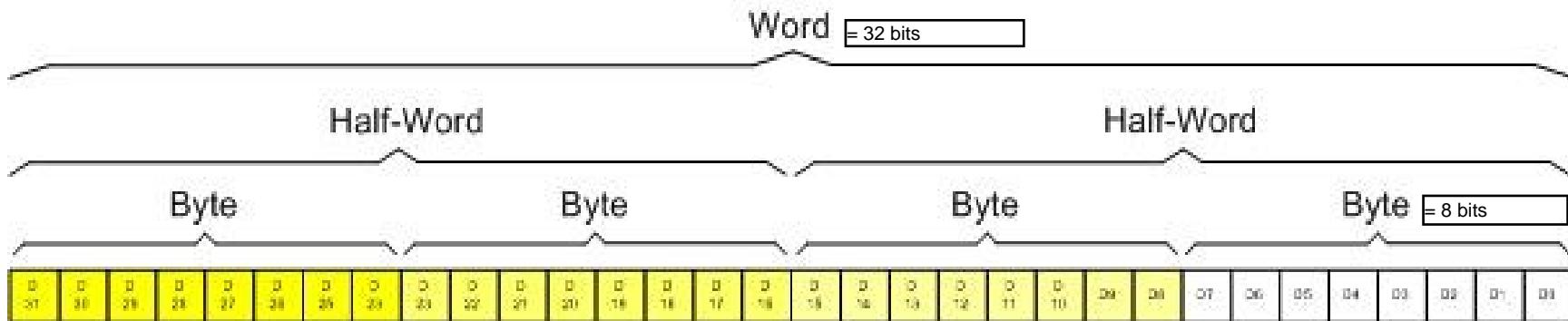
ARM Architecture and Assembly Language Programming

ARM Registers



Registers Data Size

- **MSB: D31**
- **LSB: D0**



General Purpose Registers

- **13 GPRs** = general purpose registers
 - **R0-R12 (32-bit)**
 - **Can be used by all arithmetic/logic instructions**

ARM Instruction Format

- A common format

all the instructions in arm are 32 bits length, because our registers are 32 bits :

Instruction destination, source1, source2

- **Source2 can be**
 - A register
 - Immediate (constant)
 - Memory
- **Destination is often**
 - A register
 - Memory

MOV Instruction

- Copies data into register or from register to register

MOV Rn, Op2

Rn is the destination register

- load **Rn** register with **Op2** (Operand2)
 - Op2 can be immediate (an 8-bit value)
 - Op2 can be a register **Rm**
 - **Rn** or **Rm** can be any of the registers R0 to R15

MOV Instruction: Examples

- Load R2 with 0x25 (R2 = 0x25)

MOV R2,#0x25; 25 in hex

- Notice the **#** before immediate values
- Use “**;**” for comment (same as “**//**” in C)
- For numbers in hex, put “**0x**” in front of the value
 - Put nothing for decimal

MOV R1,#50; 50 in decimal

- Copy contents of R7 into R5 (R5 = R7)

MOV R5,R7

MOV Instruction: Examples

- **On moving a constant,**
 - **What happens to the rest of the register's bits?**
 - **What about moving an immediate larger than 255?**

we fill 8 least significant bits with our constant, and put zero for other

MOV R1,#505

- **Will cause an error! Why?**

ADD Instruction

ADD Rd,Rn,Op2

- Add Rn to Op2 and store the result in Rd
 - Op2 can be immediate (an 8-bit value)
 - Op2 can be a register Rm

A Simple Program

- Add 0x25 and 0x34

MOV R1,#0x25

MOV R7,#0x34

ADD R5,R1,R7

OR

MOV R1,#0x25

ADD R5,R1,#0x34

add doesn't have two constants simultaneously

second one is better: one less instruction, one less registers used.30% decreases

WHY

To Learn Assembly?!!!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 14

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

Why to Learn Assembly Language

high level languages

- HLL can greatly enhance programmer productivity, **BUT**
- Writing assembly code is desirable or necessary for
 - The first steps in booting the computer
 - Code to handle interrupts
 - Low-level locking code for multi-threaded programs
 - Code for machines where no compiler exists
 - Code which needs to be optimized beyond the limits of the compiler
 - On computers with very limited memory
 - Code that requires low-level access to architectural and/or processor features
 - Write a compiler
 - Program device drivers

ARM Architecture and Assembly Language Programming

SUB Instruction

- We have already seen MOV and ADD instr.

SUB Rd, Rn, Op2 ; Rd = Rn – Op2

- A simple code

MOV R1, #0x34 ; load 0x34 into R1 (R1 = 0x34)

SUB R5, R1, #0x25 ; R5 = R1 – 0x25 (R5 = 0x34 – 0x25)

like add, sub doesn't have two constants too.

ALU Instructions Using GPRs

| Instruction | Description |
|---|--|
| ADD Rd, Rn, Op2* | ADD Rn to Op2 and place the result in Rd |
| ADC Rd, Rn, Op2 | ADD Rn to Op2 with Carry and place the result in Rd |
| AND Rd, Rn, Op2 | AND Rn with Op2 and place the result in Rd |
| BIC Rd, Rn, Op2 | AND Rn with NOT of Op2 and place the result in Rd |
| CMP Rn, Op2 | Compare Rn with Op2 and set the status bits of CPSR** |
| CMN Rn, Op2 | Compare Rn with negative of Op2 and set the status bits |
| EOR Rd, Rn, Op2 | Exclusive OR Rn with Op2 and place the result in Rd |
| MVN Rd, Op2 | Store the negative of Op2 in Rd |
| MOV Rd, Op2 | Move (Copy) Op2 to Rd |
| ORR Rd, Rn, Op2 | OR Rn with Op2 and place the result in Rd |
| RSB Rd, Rn, Op2 | Subtract Rn from Op2 and place the result in Rd |
| RSC Rd, Rn, Op2 | Subtract Rn from Op2 with carry and place the result in Rd |
| SBC Rd, Rn, Op2 | Subtract Op2 from Rn with carry and place the result in Rd |
| SUB Rd, Rn, Op2 | Subtract Op2 from Rn and place the result in Rd |
| TEQ Rn, Op2 | Exclusive-OR Rn with Op2 and set the status bits of CPSR |
| TST Rn, Op2 | AND Rn with Op2 and set the status bits of CPSR |
| * Op2 can be an immediate 8-bit value #K which can be 0–255 in decimal, (00–FF in hex). Op2 can also be a register Rm. Rd, Rn and Rm are any of the general purpose registers | |
| ** CPSR is discussed later in this chapter | |
| *** The instructions are discussed in detail in the next chapters | |

The ARM Memory Map

- **Memory space allocation in the ARM**
 - **4 gigabytes of directly accessible memory space**
from 0 to 0xFFFFFFFF
- **Memory space can be divided into five sections**
 - On-chip peripheral and I/O registers
 - On-chip data SRAM
 - On-chip EEPROM (for saving critical data)
 - On-chip Flash ROM (program space)
 - Off-chip DRAM space

The ARM Memory Map

- Why both EEPROM and Flash?

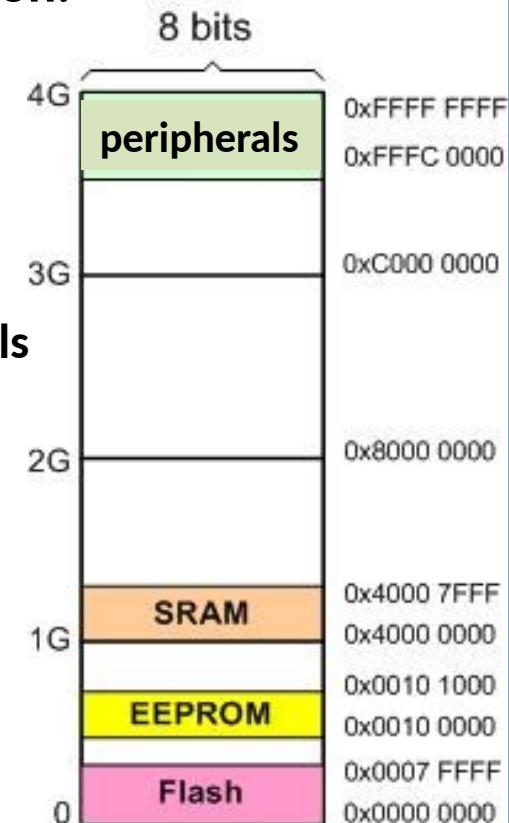
both are nonvolatile memory
- Why both SRAM and EEPROM?
- IO-mapped or Memory-Mapped IO?

our MCU is memory mapped io, because we have
- Memory space is 4GB; is it enough?

| Company | Device | Flash (K Bytes) | RAM (K Bytes) | I/O Pins |
|-----------|---------------|-----------------|---------------|----------|
| Atmel | AT91SAM7X512 | 512 | 128 | 62 |
| NXP | LPC2367 | 512 | 58 | 70 |
| ST | STR750FV2 | 256 | 16 | 72 |
| TI | TMS470R1A256 | 256 | 12 | 49 |
| Freescale | Mk10DX256VML7 | 256 | 64 | 74 |

The ARM Memory Map

- Q: A given ARM chip has the following address assignments. Calculate the space and the amount of memory given to each section.
 - Address range of **0x00100000 – 0x00100FFF** for EEPROM
 - Address range of **0x40000000 – 0x40007FFF** for SRAM
 - Address range of **0x00000000 – 0x0007FFFF** for Flash
 - Address range of **0xFFFFC0000 – 0xFFFFFFFF** for peripherals





Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 15

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

Load and Store Instructions

- Load Rd with the contents of location pointed to by Rx register

LDR Rd,[Rx]; Rx contains an address between 0x00000000 to 0xFFFFFFFF

- LDR reads one word (32-bit or 4-byte) of data

- from 4 consecutive memory locations

our MCU is byte addressable.

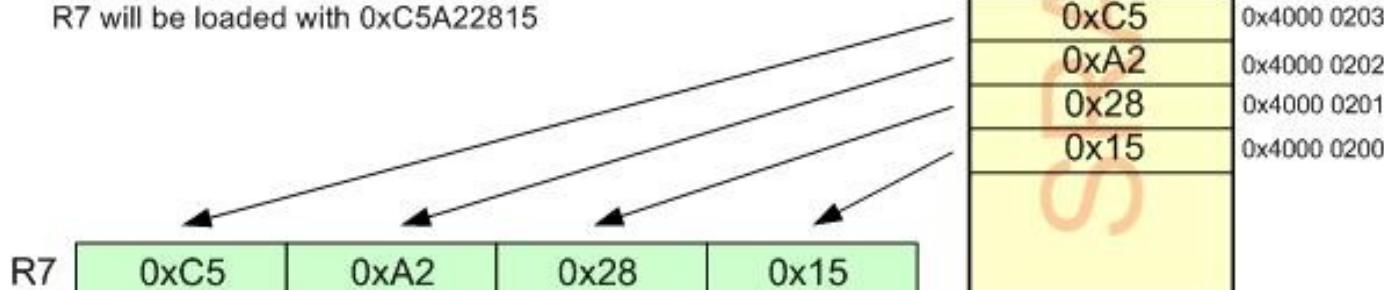
- The locations can be in the SRAM, a Flash memory or I/O registers

Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x15, 0x28, 0xA2 and 0xC5, respectively.

After running the following instruction:

`LDR R7, [R5]`

R7 will be loaded with 0xC5A22815



Load and Store Instructions

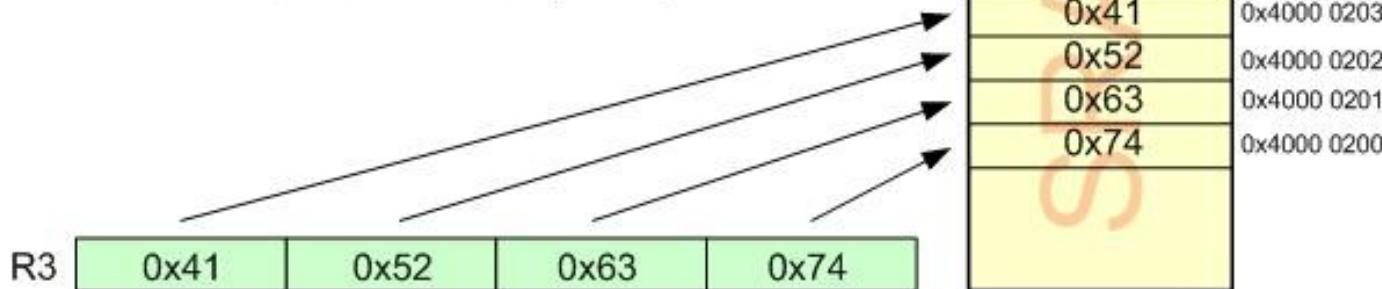
- Store register Rd into locations pointed to by Rx

STR Rd,[Rx]

Assume that R6=0x40000200, and R3 = 0x41526374. After running the following instruction:

STR R3, [R6]

locations 0x40000200 through 0x40000203 will be loaded with 0x74, 0x63, 0x52, and 0x41, respectively.



Load and Store Variations

- Load a byte

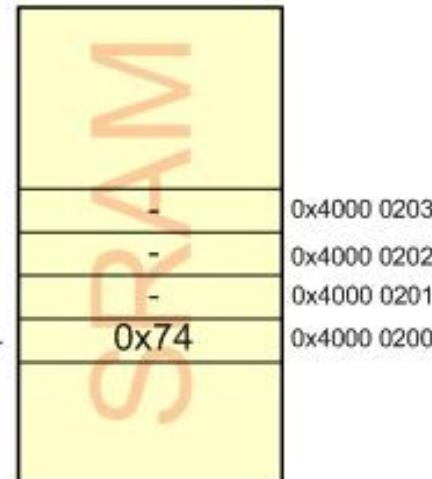
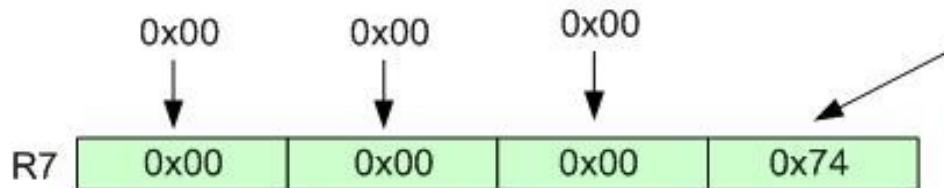
LDRB Rd, [Rx]

Assume that R5=0x40000200, and location 0x40000200 contains 0x74.

After running the following instruction:

`LDRB R7, [R5]`

R7 will be loaded with 0x00000074



Load and Store Variations

- Store a byte

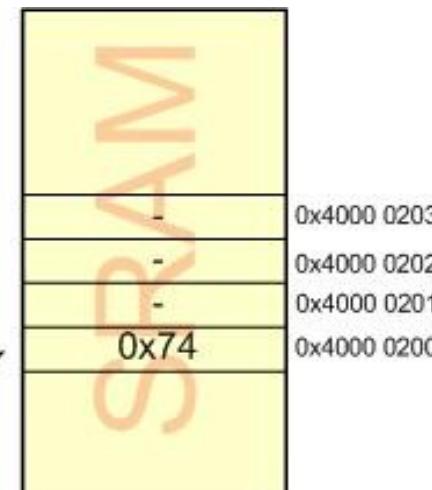
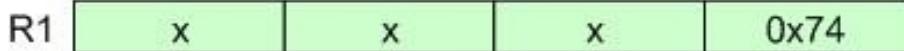
STRB Rd, [Rx]

Assume that R5=0x40000200, and R1 = 0x41526374.

After running the following instruction:

STRB R1, [R5]

locations 0x40000200 will be loaded with 0x74.



Load and Store Variations

- Load half-word (2-byte)

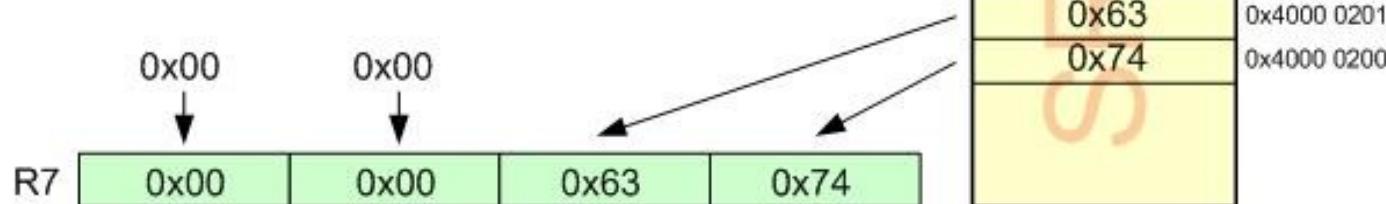
LDRH Rd, [Rx]

Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x74, 0x63, 0x52 ,and 0x41, respectively.

After running the following instruction:

LDRH R7, [R5]

R7 will be loaded with 0x00006374



Load and Store Variations

- Store half-word (2-byte)

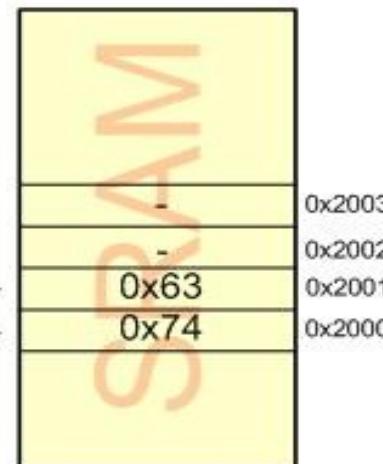
STRH Rd, [Rx]

Assume that R6=0x2000, and R3 = 0x41526374. After running the following instruction:

STRH R3 , [R6]

locations 0x2000 through 0x2001 will be loaded with 0x74 and 0x63, respectively.

R3



Current Program Status Register

- CPSR: Flag register



- N: Negative
- Z: Zero
- C: Carry
- V: Overflow
- T: Thumb
- I and F: Enable or disable the interrupt

S suffix and the status register

- By executing data processing instructions
 - By default, the status flags of CPSR are not updated
- To update the flags
 - put the 'S' suffix at the end of the opcode

ADD => ADDS

S Suffix and the Status Register

| Instruction | Flags Affected |
|-------------|----------------|
| ANDS | C, Z, N |
| ORRS | C, Z, N |
| MOVS | C, Z, N |
| ADDS | C, Z, N, V |
| SUBS | C, Z, N, V |
| B | No flags |

Note that we cannot put S after B instruction.

Flag Bits and Decision Making

- Conditional branch instructions

| Instruction | Flags Affecting the branch |
|-------------|----------------------------|
| BCS | Branch if C = 1 |
| BCC | Branch if C = 0 |
| BEQ | Branch if Z = 1 |
| BNE | Branch if Z = 0 |
| BMI | Branch if N = 1 |
| BPL | Branch if N = 0 |
| BVS | Branch if V = 1 |
| BVC | Branch if V = 0 |

branch if two numbers are equal.

branch if two numbers are not equal.



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh
farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 16

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

ARM Data Format, Pseudo-instructions and Directives

ARM Data Format

- Data format representation

- Hexadecimal numbers

MOV R1, #0x99

- Decimal numbers

MOV R7, #12

- Binary numbers

MOV R6, #2_10011001

- Numbers in any base between 2 and 9

MOV R7, #8_33

- ASCII characters

LDR R3, #'2' ; R3 = 00110010 or 32 in hex

Pseudo-instructions

- LDR pseudo-instruction
 - Limitation of loading immediate value: 8-bit

~~MOV Rd, 0x11223344~~

= 287454020 in decimal, but we can only move values to 8-bit registers

LDR Rd, =32-bit_immediate_value

LDR R7, =0x11223344

- Load registers with the addresses of memory locations

ADR Rn, label

Assembler Directives

- Directives give directions to the assembler
 - Some Widely Used ARM Directive

| Directive | Description |
|-----------|---|
| AREA | Instructs the assembler to assemble a new code or data section |
| END | Informs the assembler that it has reached the end of a source code. |
| EQU | Associate a symbolic name to a numeric constant. |
| INCLUDE | It adds the contents of a file to the current program. |

Assembler Directives

- Directives: **AREA**
 - Defines a new section of memory
AREA sectionname, attribute, attribute, ...
 - Some attributes: CODE, DATA, READONLY, READWRITE, and ALIGN
AREA MY_ASM_PROG1, CODE, READONLY

Assembler Directives

- Directives: **AREA**
 - Attributes: **READONLY**
 - All the **READONLY** sections of the same program are put next to each other in the **flash memory** by the **linker**
 - Attributes: **READWRITE**
 - The **linker** puts all the **READWRITE** sections of the same program next to each other in the **SRAM memory**
 - Attributes: **CODE**: is by default **READONLY** memory
 - Attributes: **DATA**: is by default **READWRITE** memory
 - Attributes: **ALIGN**: to align a section

Assembler Directives

- Directives: **END**
 - Indicates to the assembler the end of the source code
 - Is the last line of the ARM assembly program
 - Anything after the END directive in the source file is ignored by the assembler

AREA PROG_2_1, CODE, READONLY

MOV R1, #0x25 ; R1 = 0x25

MOV R2, #0x34 ; R2 = 0x34

ADD R3, R2, R1 ; R3 = R2 + R1

HERE B HERE ; stay here forever

END



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh
farbeh@aut.ac.ir

Department of Computer Engineering
Amirkabir University of Technology

Lecture 17

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

ARM Data Format, Pseudo-instructions and Directives (cont.)

Assembler Directives

- **Directives: EQU (equate)**
 - To define a constant value or a fixed address by a name
 - To make the program easier to read

COUNT EQU 0x25

MOV R2, #COUNT ; R2 = 0x25

we should put the # before the COUNT



Converted by Assembler to

MOV R2, #0x25

- Avoid searching the entire program trying to find and change every occurrence of a variable

Assembler Directives

- **Directives: EQU (equate)**

- **Using EQU for fixed data assignment**

DATA1 EQU 0x39 ; the way to define hex value

DATA2 EQU 2_00110101 ; the way to define binary value (35 in hex)

DATA3 EQU 39 ; decimal numbers (27 in hex)

DATA4 EQU '2' ; ASCII characters

- **Using EQU for special register address assignment**

FIO2SET0 EQU 0x3FFC058 ; PORT2 output set register 0 address

MOV R6, #0x01 ; R6 = 0x01

in this program we change output of port2 to 1

LDR R2, =FIO2SET0 ; R2 = 0x3FFC058

STRB R6, [R2] ; Write 0x01 to FIO2SET0

Assembler Directives

- Directives: **EQU (equate)**
 - Using EQU for RAM address assignment

```
SUM EQU 0x40000120 ; assign RAM location to SUM
MOV R2, #5 ; load R2 with 5
MOV R1, #2 ; load R1 with 2
ADD R2, R2, R1 ; R2 = R2 + R1
LDR R3, =SUM ; load R3 with 0x40000120
STRB R2, [R3] ; store the result SUM
```

- Directives: **RN (equate)**
 - To give a CPU register a name

```
VAL1 RN R1 ; define VAL1 as a name for R1
VAL2 RN R2 ; define VAL2 as a name for R2
SUM RN R3 ; define SUM as a name for R3
```

Assembler Directives

- Directives: **INCLUDE**
 - Tells the ARM assembler **to read in the content of a file to the current program file**

Assembler Data Allocation Directives

- Directives to allocate memory and initialize its value
- Directives: **DCB**
 - define constant byte
 - MYVALUE DCB 5 ; MYVALUE = 5
 - MYMSAGE DCB "HELLO WORLD" ; ASCII string gives 1 byte for each ascii character
 - Each alphanumeric letter in a string is converted to its ASCII encoding value
- Directives: **DCW => define constant half-word**
 - MYDATA DCW 0x20, 0xF230, 5000, 0x9CD7
- Directives: **DCD => define constant word**
 - MYDATA DCD 0x200000, 0x30F5, 5000000, 0xFFFF9CD7

Assembler Data Allocation Directives

- An sample program

AREA LOOKUP_EXAMPLE, READONLY, CODE

LDR R2, =OUR_FIXED_DATA ; point to OUR_FIXED_DATA

LDRB R0, [R2] ; load R0 with the contents of memory pointed to by R2

ADD R1, R1, R0 ; add R0 to R1

HERE B HERE ; stay here forever

OUR_FIXED_DATA

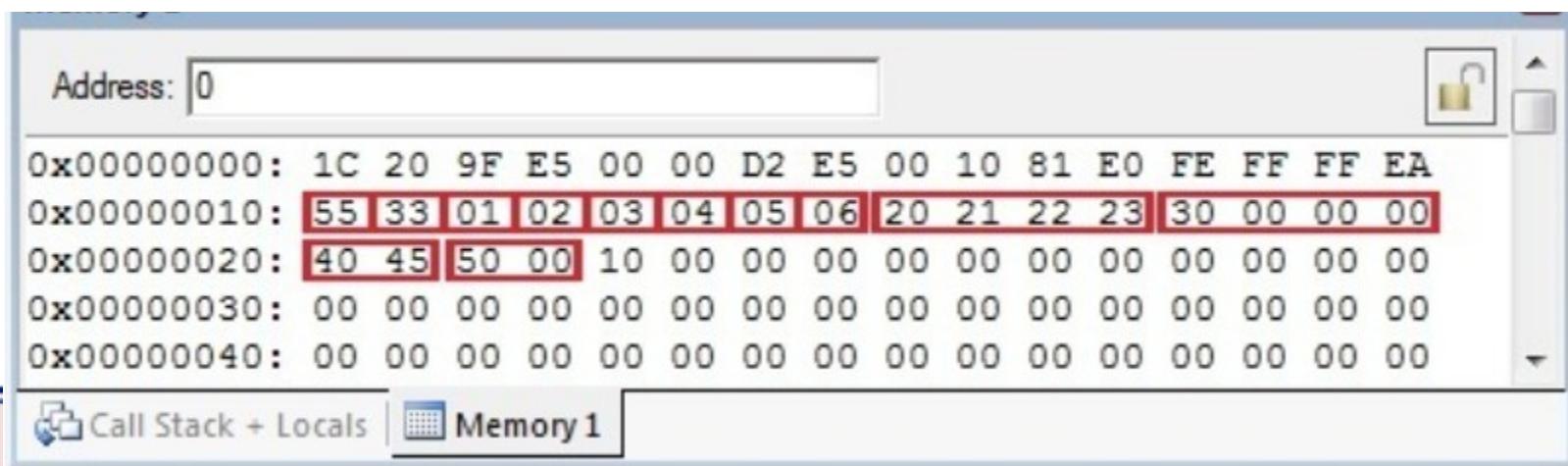
DCB 0x55, 0x33, 1, 2, 3, 4, 5, 6

DCD 0x23222120, 0x30

DCW 0x4540, 0x50

END

ADR R2, OUR_FIXED_DATA



Assembler Data Allocation Directives

- Directives to allocate memory and initialize its value
- Directives: **SPACE**
 - To allocate memory for variables without initial values
 - LONG_VAR SPACE 4 ; Allocate 4 bytes
 - OUR_ALFA SPACE 2 ; Allocate 2 bytes

- Directives: **ALIGN**
 - To make sure data is aligned on the 32-bit word or 16-bit half word address boundary
 - ALIGN 4 ; the next instruction is word (4 bytes) aligned
 - ...
 - ALIGN 2 ; the next instruction is half-word (2 bytes) aligned
 - ...

Assembler Data Allocation Directives

- Directives: **ALIGN**

AREA E2_7A, READONLY, CODE

ADR R2, DTA

LDRB R0, [R2]

ADD R1, R1, R0

H1 B H1

DTA DCB 0x55

DCB 0x22

END

DTA DCB 0x55

ALIGN 2

DCB 0x22

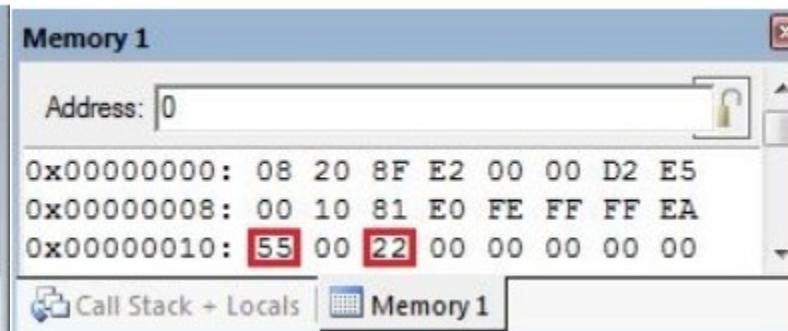
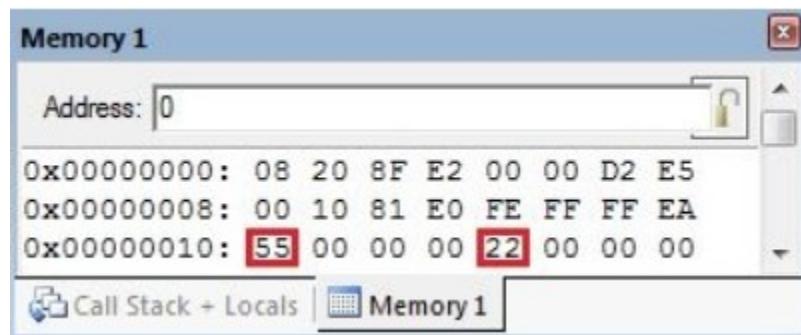
END

DTA DCB 0x55

ALIGN 4

DCB 0x22

END



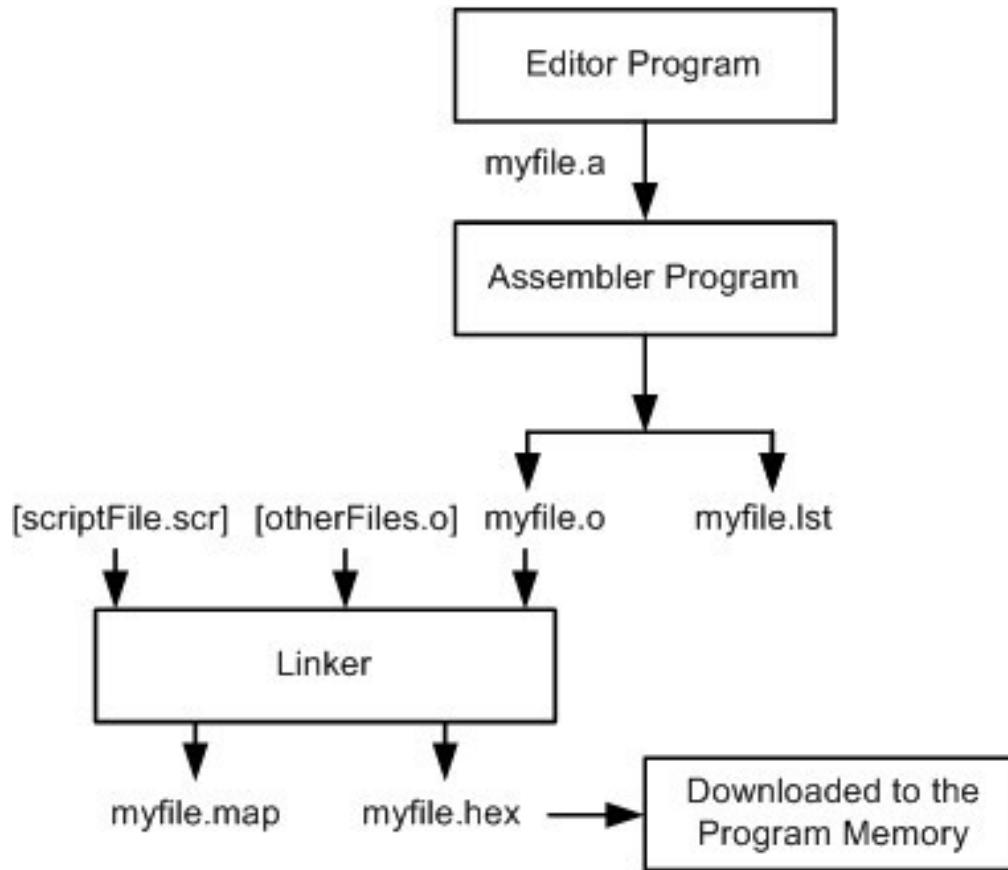
Rules for Labels in Assembly Language

- Each label name must be unique in the file
- Label names consist of
 - Alphabetic letters in both uppercase and lowercase
 - The digits 0 through 9, and
 - The special characters underscore ‘_’
- The first character of the label must be
 - An alphabetical letter or underscore
 - Cannot be a numeral
 - Cannot be reserved words (ADD, MOV, ...)

Creating an ARM Assembly Program

Creating an ARM Assembly Program

- Steps to create a program



Creating an ARM Assembly Program

- Sample of a Map File

```
Memory Map of the image
```

```
Image Entry point : 0x00000000
```

```
Load Region LR_1 (Base: 0x00000000, Size: 0x00000010, Max: 0xffffffff, ABSOLUTE)
```

```
Execution Region ER_RO (Base: 0x00000000, Size: 0x00000010, Max: 0xffffffff, ABSOLUTE)
```

| Base Addr | Size | Type | Attr | Idx | E Section Name | Object |
|------------|------------|------|------|-----|----------------|--------|
| 0x00000000 | 0x00000010 | Code | RO | 1 | * PROG_2_1 | a2.o |

```
Execution Region ER_RW (Base: 0x40000000, Size: 0x00000000, Max: 0xffffffff, ABSOLUTE)
```

```
**** No section assigned to this execution region ****
```

```
Execution Region ER_ZI (Base: 0x40000000, Size: 0x00000000, Max: 0xffffffff, ABSOLUTE)
```

```
**** No section assigned to this execution region ****
```

Creating an ARM Assembly Program

- Sample of a **List File**

| ARM Macro Assembler | | Page 1 | |
|---------------------|----------|---|------------------------------|
| 1 | 00000000 | ; ARM Assembly Language Program To Add Some Data and Store the SUM in R3. | |
| 2 | 00000000 | | |
| 3 | 00000000 | AREA PROG_2_4, CODE, READONLY | |
| 4 | 00000000 | ENTRY | |
| 5 | 00000000 | E3A01025 | MOV R1, #0x25 ; R1 = 0x25 |
| 6 | 00000004 | E3A02034 | MOV R2, #0x34 ; R2 = 0x34 |
| 7 | 00000008 | E0823001 | ADD R3, R2,R1 ; R3 = R2 + R1 |
| 8 | 0000000C | EAFFFFFE | B HERE |
| 9 | 00000010 | | END |

Power up Location for ARM

- Q: At what address does the CPU wake up to when power is applied or when the CPU is reset?
 - ARM7 microcontrollers: **0x00000000**
 - The first instruction is expected to be stored here
- **BUT, ARM Cortex-M is different**
 - Reads from **0x00000004-0x00000007** **and**
 - Put them into the program counter, **then**
 - CPU fetches the first instruction using the content of PC
 - The programmer (working with the software tools) shall put the starting address of the program at memory location **0x00000004**



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 18

Copyright Notice

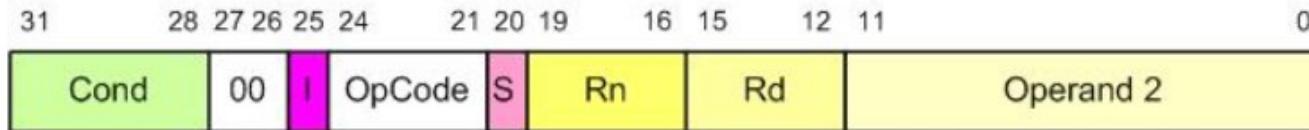
Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

Instruction Formation of the ARM

Instruction Formation

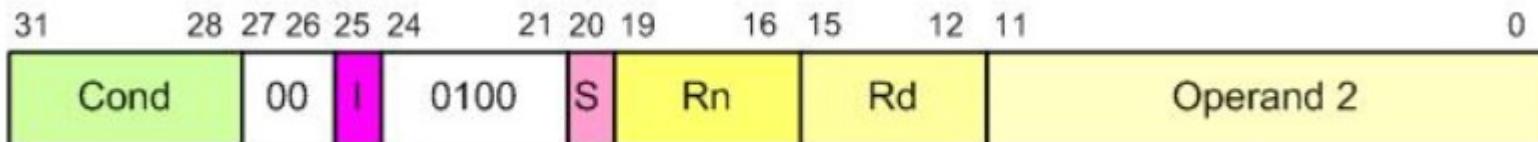
- General formation of data processing instructions



- Cond [31:28]: Condition field (will be discussed in Ch. 4)
- [27:26]: Always "00"
- I [25]: Defines the type of second operand
 - I = 1: Op2 is an immediate value
- [24:21]: Opcode
- S [20]: Update CPSR

Instruction Formation

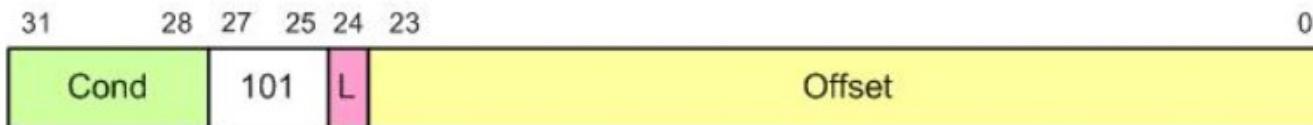
- **ADD instruction format**



- **SUB instruction format**



- **Branch (B) instruction format**



Little Endian vs. Big Endian War

• Little endian

- Low byte goes to the low memory location
- High byte goes to the high memory address
- All Intel processors

• Big endian

- Low byte goes to the high memory location
- High byte goes to the low memory address
- Freescale (formerly Motorola and now NXP)

- ## • ARM allows the software designer to choose!
- A hardware switch that is controlled by software

| Value | Memory | Address |
|-------------|----------------------|--|
| E0 82 30 01 | E0 82 30 01 | 0000 000B 0000 000A 0000 0009 0000 0008 |
| E3 A0 20 34 | E3 A0 20 34 | 0000 0007 0000 0006 0000 0005 0000 0004 |
| E3 A0 10 25 | E3 A0 10 25 | 0000 0003 0000 0002 0000 0001 0000 0000 |

| Value | Memory | Address |
|-------------|----------------------|--|
| E0 82 30 01 | 01 30 82 E0 | 0000 000B 0000 000A 0000 0009 0000 0008 |
| E3 A0 20 34 | 34 20 A0 E3 | 0000 0007 0000 0006 0000 0005 0000 0004 |
| E3 A0 10 25 | 25 10 A0 E3 | 0000 0003 0000 0002 0000 0001 0000 0000 |

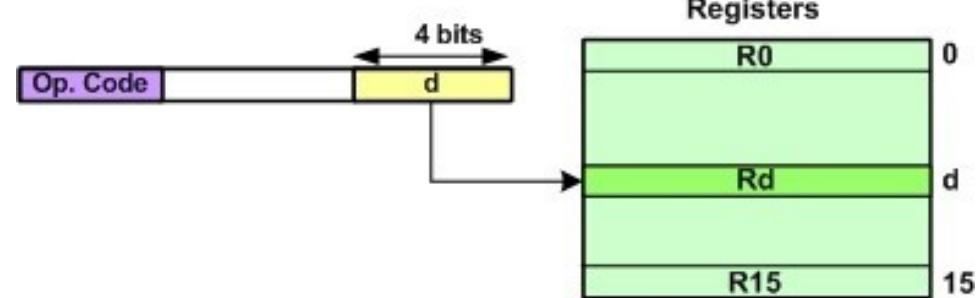
Some ARM Addressing Modes

Addressing Modes

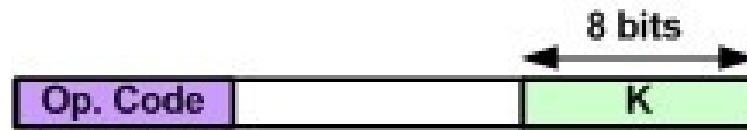
- **Addressing Mode**
 - The various ways operands are specified in the instruction
 - Is the way CPU generates address from instruction to read/write the operands
- **Three basic modes**
 - Register
 - Immediate
 - Register indirect (indexed addressing mode)

Addressing Modes

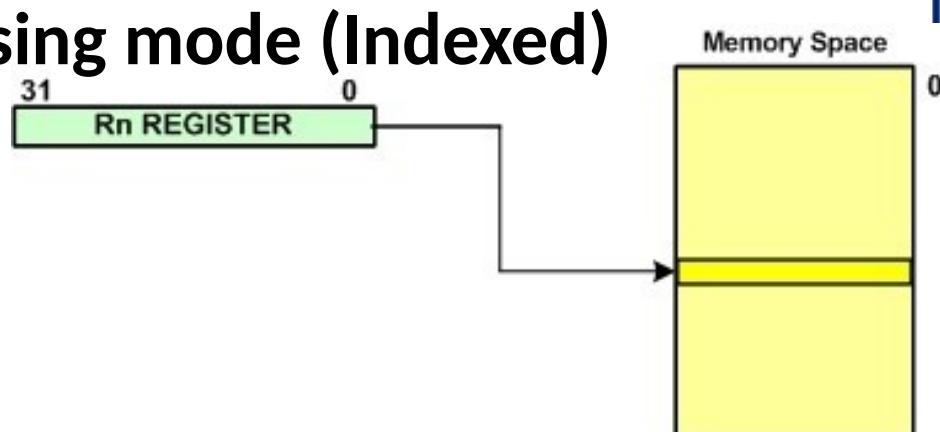
- Register Addressing Mode



- Immediate addressing mode



- Register Indirect addressing mode (Indexed)



Addressing Modes

- Register Indirect addressing mode (Indexed)
 - Can be used for **pointer** implementation

`char *ourPointer;`

`ourPointer = (char*) 0x12456; //Point to location 12456`

`*ourPointer = 25; //store 25 in location 0x12456`

`ourPointer++; //point to next location`

`LDR R2, =0x12456 ; point to location 0x12456`

`MOV R0, #25 ; R0 = 25`

`STRB R0, [R2] ; store R0 in location 0x12456`

`ADD R2, R2, #1 ; increment R2 to point to next location`

End of Chapter 2!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh
farbeh@aut.ac.ir

Department of Computer Engineering
Amirkabir University of Technology

Lecture 19

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

Arithmetic and Logic Instructions and Programs

Arith. Instr. and Flag Bits for Unsigned Data

| Instruction (Flags unchanged) | | Instruction (Flags updated) | |
|-------------------------------|-----------------------------|-----------------------------|---|
| ADD | Add | ADDS | Add and set flags |
| ADC | Add with carry | ADCS | Add with carry and set flags |
| SUB | SUBS | SUBS | Subtract and set flags |
| SBC | Subtract with carry | SBCS | Subtract with carry and set flags |
| MUL | Multiply | MULS | Multiply and set flags |
| UMULL | Multiply long | UMULLS | Multiply Long and set flags |
| RSB | Reverse subtract | RSBS | Reverse subtract and set flags |
| RSC | Reverse subtract with carry | RSCS | Reverse subtract with carry and set flags |

Note: The above instruction affect all the N, Z, C, and V flag bits of CPSR (current program status register) but the N and V flags are for signed data and are discussed in Chapter 5.

- No increment instr. In ARM
 - Use the available ADD instr.

Multiword ADD and SUB

1 word

- Add 0x35F62562FA to 0x21F412963B

40 bits

LDR R0, =0xF62562FA; R0 = 0xF62562FA

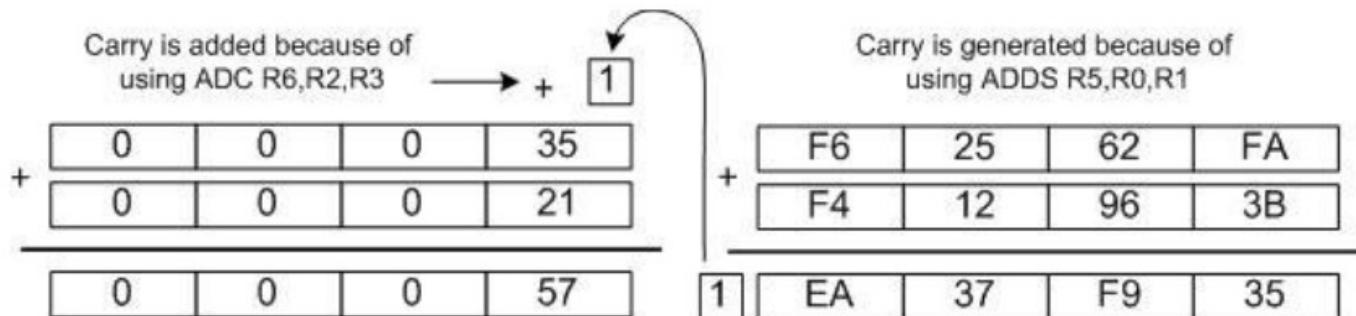
LDR R1, =0xF412963B; R1 = 0xF412963B

MOV R2, #0x35; R2 = 0x35

MOV R3, #0x21; R3 = 0x21

ADDS R5, R1, R0; R5 = 0xF62562FA + 0xF412963B ; now C = 1

ADC R6, R2, R3; R6 = R2 + R3 + C; = 0x35 + 21 + 1 = 0x57



Multiword ADD and SUB

- Sub 0x21F62562FA from 0x35F412963B

LDR R0, =0xF62562FA; R0 = 0xF62562FA ; notice the syntax for LDR

LDR R1, =0xF412963B; R1 = 0xF412963B

MOV R2, #0x21; R2 = 0x21

MOV R3, #0x35; R3 = 0x35

SUBS R5, R1, R0; R5 = R1 - R0; = 0xF412963B - 0xF62562FA, and C = 0

SBC R6, R3, R2; R6 = R3 - R2 - 1 + C; = 0x35 - 0x21 - 1 + 0 = 0x13

we add it with the carry bit.

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|--|--|------------------------------------|--|---|----|----|----|----|----|-------|--|--|--|---|---|---|----|--|---|----|----|----|----|----|----|----|----|---|----|----|----|--|--|--|----|
| $\begin{array}{l} \text{SBC R6,R3,R2} \Rightarrow R6 = C - 1 + R3 - R2 \\ R6 = C - 1 + R3 + (\text{2's complement of R2}) \end{array}$ | $\begin{array}{r} + 0 \\ - 1 \end{array}$ | $C = 0 \text{ so there is borrow}$ | $\begin{array}{l} \text{SUBS R5,R1,R0} \Rightarrow \\ R5 = R1 + (\text{2's complement of R0}) \end{array}$ | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| $\begin{array}{r} + \\ \hline \end{array}$ | <table border="1"><tr><td>0</td><td>0</td><td>0</td><td>35</td></tr><tr><td>FF</td><td>FF</td><td>FF</td><td>DF</td></tr><tr><td colspan="4"><hr/></td></tr><tr><td>0</td><td>0</td><td>0</td><td>13</td></tr></table> | 0 | 0 | 0 | 35 | FF | FF | FF | DF | <hr/> | | | | 0 | 0 | 0 | 13 | $\begin{array}{r} + \\ \hline \end{array}$ | <table border="1"><tr><td>F4</td><td>12</td><td>96</td><td>3B</td></tr><tr><td>09</td><td>DA</td><td>9D</td><td>06</td></tr><tr><td>0</td><td>FD</td><td>ED</td><td>33</td></tr><tr><td></td><td></td><td></td><td>41</td></tr></table> | F4 | 12 | 96 | 3B | 09 | DA | 9D | 06 | 0 | FD | ED | 33 | | | | 41 |
| 0 | 0 | 0 | 35 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| FF | FF | FF | DF | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| <hr/> | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 0 | 0 | 13 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| F4 | 12 | 96 | 3B | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 09 | DA | 9D | 06 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | FD | ED | 33 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| | | | 41 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

Multiplication and Division of Unsigned Numbers

- Not all CPUs have instr. for mult. and div.
 - All the ARM processors have mult., but not all have div.
 - ARM Cortex-M3 and M4 have both mult. and div.
- MUL:** Regular multiplication
- MULL:** long multiplication

| Instruction | Source 1 | Source 2 | Destination | Result |
|--------------|----------|----------|----------------------|---------------------------|
| MUL | Rn | Op2 | Rd (32 bits) | $Rd=Rn \times Op2$ |
| UMULL | Rn | Op2 | RdLo, RdHi (64 bits) | $RdLo:RdHi=Rn \times Op2$ |

Note 1: Using MUL for word \times word multiplication preserves only the lower 32 bit result in Rd and the rest are dropped. If the result is greater than 0xFFFFFFFF, then we must use UMULL (unsigned Multiply Long) instruction.

Note 2: In some CPUs the C flag is used to indicate the result is greater than 32-bit but this is not the case with ARM MUL instruction.

Multiplication of Unsigned Numbers

- **MUL**: Regular multiplication
- **MULL**: long multiplication

LDR R1, =100000; R1=100,000

LDR R2, =150000; R2=150,000

MUL R3, R2, R1; R3 is not 15,000,000,000; it cannot fit in 32 bits.

32 bits --> max ~= 4,300,000,000

LDR R1, =0x54000000; R1 = 0x54000000

LDR R2, =0x10000002; R2 = 0x10000002

LSB MSB

UMULL R3, R4, R2, R1; $0x54000000 \times 0x10000002 =$

$0x05400000A8000000$; R3 = 0xA8000000, the lower 32 bits

; R4 = 0x05400000, the higher 32 bits

Multiplication of Unsigned Numbers

- Multiply and Accumulate Instructions in ARM

MLA Rd, Rm, Rs, Rn ; Rd = Rm × Rs + Rn

Rd can't be bigger than 32 bits.

MOV R1, #100; R1 = 100

MOV R2, #5; R2 = 5

MOV R3, #40; R3 = 40

MLA R4, R1, R2, R3; R4 = R1 × R2 + R3 = 100 × 5 + 40 = 540

- To accumulate the products of the multiplication

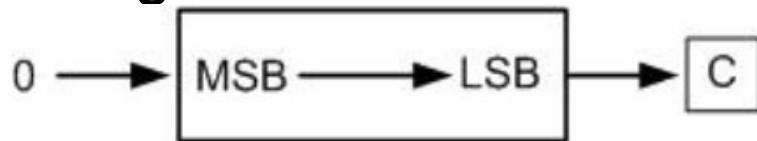
MLA R3, R1, R2, R3; R3 = R1 × R2 + R3 or R3 += R1 × R2

- UMLAL: unsigned multiply and accumulate long

UMLAL RdLo, RdHi, Rn, Op2; RdHi:RdLo = Rn × Op2 + RdHi:RdLo

Rotate and Barrel Shifter

- LSR: Logical shift right



MOV R0, #0x9A; R0 = 0x9A

MOVS R1, R0, LSR #3; shiftR R0 3 times, then store the result in R1

MOV R0, #0x9A

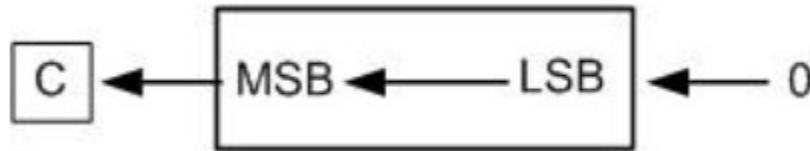
MOV R2, #0x03

MOVS R1, R0, LSR R2; shiftR R0 R2 times; and move the result to R1

Rotate and Barrel Shifter

- LSL: Logical shift left

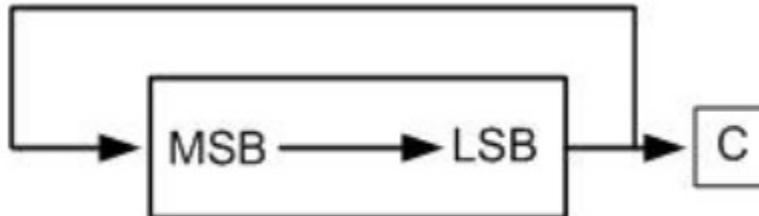
MOV R1, R1, LSL R2



- ROR: Rotate right

MOVS R1, R1, ROR #1

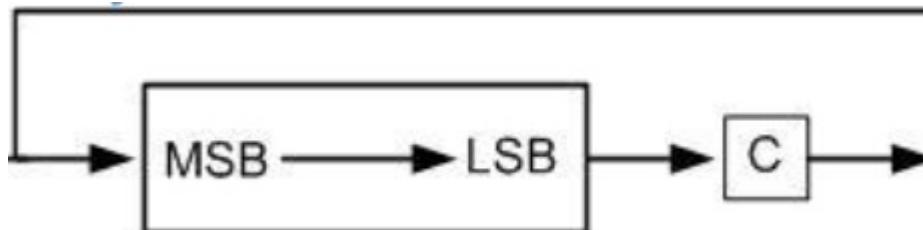
MOVS R1, R1, ROR R0



Rotate and Barrel Shifter

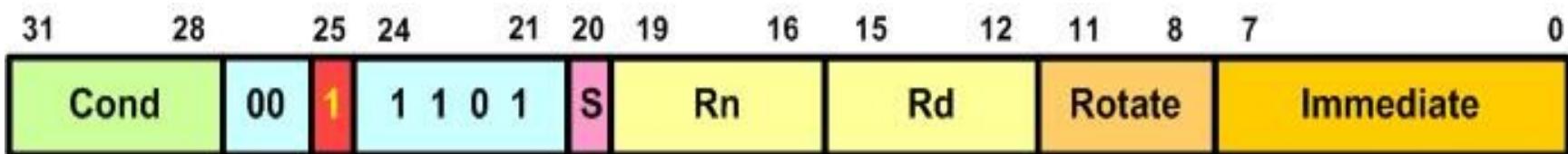
- **NO ROL: Rotate left**
 - Use ROR for rotating left
 - n-bit ROL ↪ (32 - n)-bit ROR
 - Take care of **carry**
- RRX: Rotate right through carry
 - Takes no arguments

MOVS R2, R2, RRX; Rotate 1 bit



Rotating Immediate Arguments

- MOV Instruction



MOV R0, #0xFF, #2; R0 = 0xFF is rotated right 2 times. R0 = 0xC000003F

MOV R0, #0xFF, #12; R0 = 0xFF is rotated right 12 times. R0 = 0x0FF00000

MOV R0, #0xFF, #28; R0 = 0xFF is rotated right 28 times. R0 = 0x00000FF0

we put half of the rotate number in Rotate register, for example in this example

- BUT, Rotate field (bit₁₁-bit₈) is 4-bit

Rotating Immediate Arguments

- Shift instruction

MOV R0, R2, LSL #8 ↗ LSL R0, R2, #8 LSL ==> pseudo instruction

- ASR: Arithmetic Shift right

ASR Rd, Rm, Rn

ASRS Rd, Rm, Rn

- LSL: Logical Shift Left

LSL Rd, Rm, Rn

- ...

BCD and ASCII Conversion

- **Unpacked BCD:** each decimal digit is represented by a byte
0000 1001 0000 0101 ↗ 95
- **Packed BCD:** two decimal digits are packed in one byte
1001 0101 ↗ 95
- **ASCII to unpacked BCD conversion**
 - Get rid of the "011" in the upper 3 bits of the 7-bit ASCII
 - ASCII number is ANDed with "0000 1111"
- **ASCII to packed BCD conversion**
 - First convert to unpacked BCD (remove the upper 3 bits)
 - Combine every two digits to make a packed BCD

BCD and ASCII Conversion

- Packed BCD to ASCII conversion
 - Convert packed BCD to unpacked BCD
 - Tagged with 011 0000 (0x30)

| Packed BCD | Unpacked BCD | ASCII |
|------------------|-----------------------|---------------------|
| 0x29 | 0x02 & 0x09 | 0x32 & 0x39 |
| 0010 1001 | 0000 0010 & 0000 1001 | 011 0010 & 011 1001 |

End of Chapter 3!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 20

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

Branch, Call, and Looping in ARM

Looping in ARM

branch if not equal

- Using instruction BNE for looping

```
BACK ..... ; start of the loop
..... ; body of the loop
..... ; body of the loop
SUBS Rn, Rn, #1 ; Rn = Rn - 1, set the flag Z = 1 if Rn = 0
BNE BACK ; branch if Z = 0
```

; --- this program adds value 9 to the R0 a 1000 times ---

```
AREA EXAMPLE4_1, CODE, READONLY
LDR R2, =1000 ; R2 = 1000 (decimal) for counter
MOV R0, #0 ; R0 = 0 (sum)
AGAIN ADD R0, R0, #9 ; R0 = R0 + 9 (add 09 to R1, R1 = sum)
SUBS R2, R2, #1 ; Decrement counter and set the flags.
BNE AGAIN ; repeat until COUNT = 0 (when Z = 1)
MOV R4, R0 ; store the sum in R4
HERE B HERE ; stay here
END
```

Looping in ARM

- Loop inside a loop

AREA EXAMPLE4_3, CODE, READONLY

MOV R0, #0x55 ; R0 = 0x55

MOV R2, #16 ; load 16 into R2 (outer loop count)

L1 LDR R1, =1000000000 ; R1 = 1,000,000,000 (inner loop count)

L2 EOR R0, R0, #0xFF ; complement R0 (R0 = R0 Ex-OR 0xFF) everything being xored by all 11..11

SUBS R1, R1, #1 ; R1 = R1 - 1, decrement R1 (inner loop)

BNE L2 ; repeat it until R1 = 0

SUBS R2, R2, #1 ; R2 = R2 - 1, decrement R2 (outer loop)

BNE L1 ; repeat it until R2 = 0

HERE B HERE ; stay here

END

Looping in ARM

- Conditional Branch Instructions for Unsigned Data

| Instruction | | Action |
|----------------|--|---------------------------|
| BCS/BHS | branch if carry set/branch if higher or same | Branch if C = 1 |
| BCC/BLO | branch if carry clear/branch if lower | Branch if C = 0 |
| BEQ | branch if equal | Branch if Z = 1 |
| BNE | branch if not equal | Branch if Z = 0 |
| BLS | branch if lower or same | Branch if Z = 1 or C = 0 |
| BHI | branch if higher | Branch if Z = 0 and C = 1 |

BEQ ==BNE !=BHS >=BHI >BLO <BLS <=

Looping in ARM

- Comparison of unsigned numbers

CMP Rn, Op2

| Instruction | C | Z |
|-------------|---|---|
| Rn > Op2 | 1 | 0 |
| Rn = Op2 | 1 | 1 |
| Rn < Op2 | 0 | 0 |

LDR R1, =0x35F ; R1 = 0x35F

LDR R2, =0xCC ; R2 = 0xCC

CMP R1, R2 ; compare 0x35F with 0xCC

BCC OVER ; branch if C = 0

MOV R1, #0 ; if C = 1, then clear R1

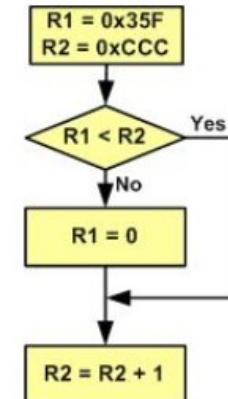
OVER ADD R2, R2, #1

; R2 = R2 + 1 = 0xCC + 1 = 0xCD

In C:
R1 = 0x35F;
R2 = 0xCC;

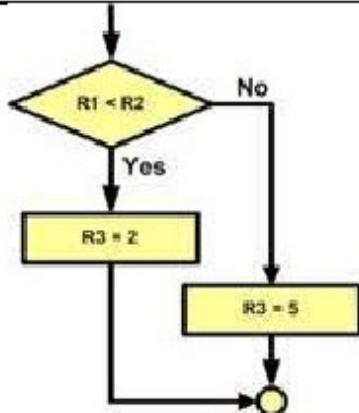
if (R1 >= R2)
{
 R1 = 0;
}

R2 = R2 + 1;



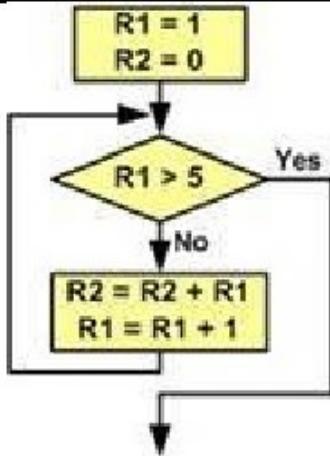
Branch Instruction

```
CMP R1, R2  
BHS L1  
MOV R3, #2  
B OVER  
L1 MOV R3, #5  
OVER
```



```
//in C  
if(R1 < R2)  
{  
    R3 = 2;  
}  
else  
{  
    R3 = 5;  
}
```

```
MOV R1, #1  
MOV R2, #0  
L1 CMP R1, #5  
BHI L2  
ADD R2, R2, R1  
ADD R1, R1, #1  
B L1  
L2 MOV R3, #5
```

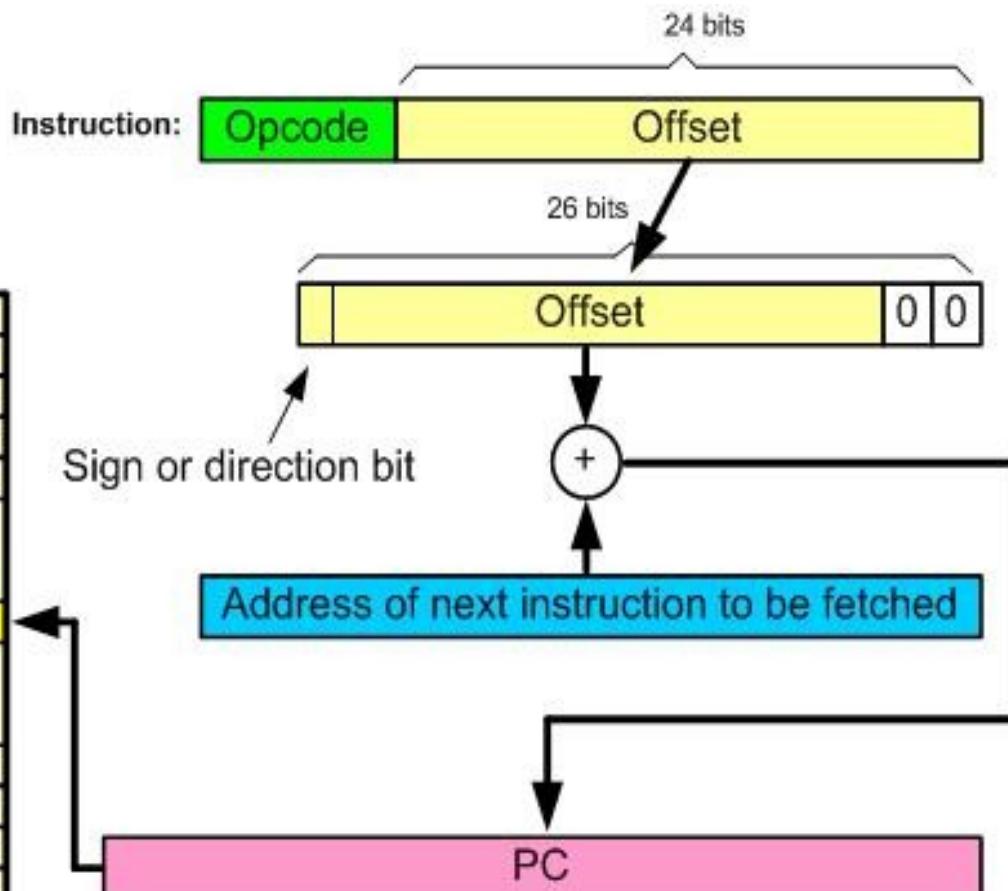


```
//in C  
unsigned int R1 = 1;  
unsigned int R2 = 0;  
while (R1 <= 5)  
{  
    R2 = R2 + R1;  
    R1 = R1 + 1;  
}
```

Branch Instruction

| | | | |
|------------|------------|------------|------------|
| 0x00000003 | 0x00000002 | 0x00000001 | 0x00000000 |
| 0x00000007 | 0x00000006 | 0x00000005 | 0x00000004 |
| 0x0000000B | 0x0000000A | 0x00000009 | 0x00000008 |
| 0x0000000F | 0x0000000E | 0x0000000D | 0x0000000C |
| 0x00000013 | 0x00000012 | 0x00000011 | 0x00000010 |
| ⋮ | ⋮ | ⋮ | ⋮ |
| ⋮ | ⋮ | ⋮ | ⋮ |
| 0xFFFFFFF3 | 0xFFFFFFF2 | 0xFFFFFFF1 | 0xFFFFFFF0 |
| 0xFFFFFFF7 | 0xFFFFFFF6 | 0xFFFFFFF5 | 0xFFFFFFF4 |
| 0xFFFFFFFB | 0xFFFFFFF9 | 0xFFFFFFF9 | 0xFFFFFFF8 |
| 0xFFFFFFFF | 0xFFFFFFFF | 0xFFFFFFFF | 0xFFFFFFFF |

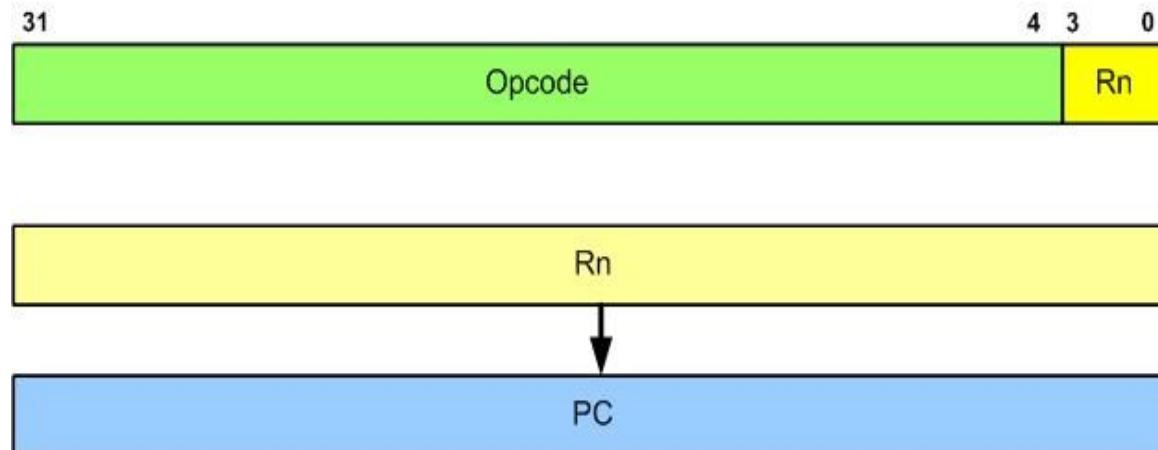
Program memory (Addresses are word-aligned)



Branching beyond 32MB byte limit

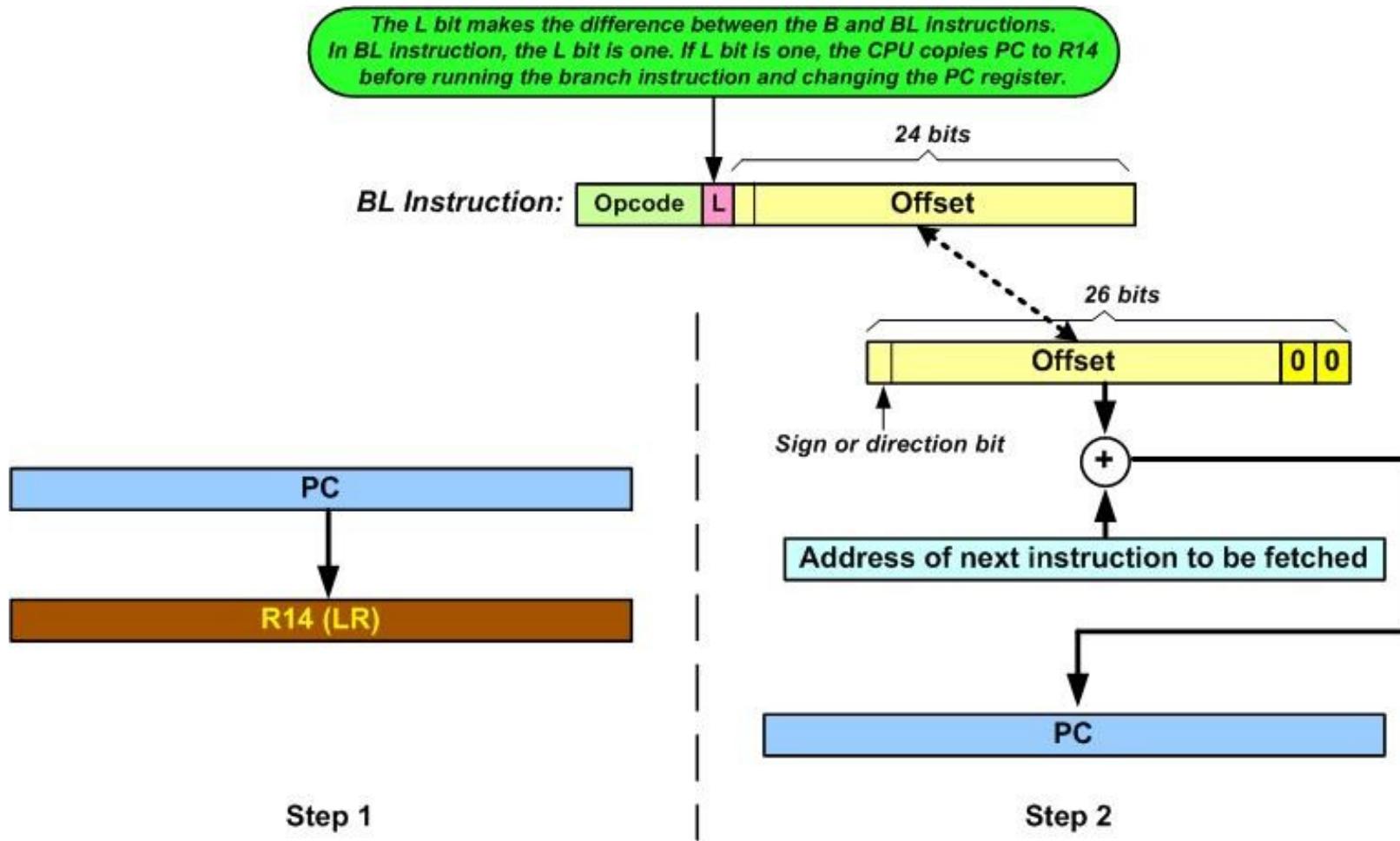
- Branch and exchange

BX Rn



Calling Subroutine with BL

- BL: Branch and Link



Calling Subroutine with BL

- Delay subroutine

AREA EXAMPLE4_8, CODE, READONLY

```
RAM_ADDR EQU 0x40000000 ; change the address for your ARM
LDR    R1, =RAM_ADDR ; R1 = RAM address
AGAIN MOV    R0, #0x55 ; R0 = 0x55
STRB   R0, [R1] ; send it to RAM
BL     DELAY ; call delay (R14 = PC of next instruction)
MOV    R0, #0xAA ; R0 = 0xAA
STRB   R0, [R1] ; send it to RAM
BL     DELAY ; call delay
B      AGAIN ; keep doing it
; -----DELAY SUBROUTINE
DELAY LDR R3, =5 ; R3 = 5, modify this value for different delay
L1    SUBS   R3, R3, #1 ; R3 = R3 - 1
      BNE L1
      BX    LR ; return to caller; -----end of DELAY subroutine
END ; notice the place for END directive
```

Main Program and Calling Subroutines

```
; MAIN program calling subroutines
AREA PogramName, CODE, READONLY

MAIN BL SUBR_1           ; Call Subroutine 1
      BL SUBR_2           ; Call Subroutine 1
      BL SUBR_3           ; Call Subroutine 1
HERE BAL HERE            ; stay here. BAL is the same as B
; -----end of MAIN

; -----SUBROUTINE 1
SUBR_1 ....
.....
BX LR ; return to main
; ----- end of subroutine 1

; -----SUBROUTINE 2
SUBR_2 ....
.....
BX LR ; return to main
; ----- end of subroutine 2

; -----SUBROUTINE 3
SUBR_3 ....
.....
BX LR ; return to main
; ----- end of subroutine 3
END      ; notice the END of file
```

Conditional Execution

- A unique feature of ARM
 - conditional execution for ALL instructions

| Cond | Instruction | | |
|------|--------------------|--------------------------|-----------------|
| Bits | Mnemonic Extension | Meaning | Flag |
| 0000 | EQ | Equal | Z = 1 |
| 0001 | NE | Not equal | Z = 0 |
| 0010 | CS/HS | Carry Set/Higher or Same | C = 1 |
| 0011 | CC/LO | Carry Clear/Lower | C = 0 |
| 0100 | MI | Minus/Negative | N = 1 |
| 0101 | PL | Plus | N = 0 |
| 0110 | VS | V Set (Overflow) | V = 1 |
| 0111 | VC | V Clear (No Overflow) | V = 0 |
| 1000 | HI | Higher | C = 1 and Z = 0 |
| 1001 | HS | Lower or Same | C = 1 and Z = 1 |
| 1010 | GE | Greater than or Equal | N = V |
| 1011 | LT | Less than | N ≠ V |
| 1100 | GT | Greater than | Z = 0 and N = V |
| 1101 | LE | Less than or Equal | Z = 0 or N ≠ V |
| 1110 | AL | Always (unconditional) | |
| 1111 | --- | Not Valid | |

Conditional Execution

- **Conditional MOV**

MOV R1, #10; R1 = 10

MOV R2, #12; R2 = 12

CMP R2, R1; compare 12 with 10, Z=0 because they are not equal

MOVEQ R4, #20; this line is not executed because the condition EQ is not met

- **Conditional ADD with 'S' suffix**

ADDNES R1, R1, #10 ; this line is executed and set the flags if Z = 0

End of Chapter 4!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh
farbeh@aut.ac.ir

Department of Computer Engineering
Amirkabir University of Technology

Lecture 21

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

Signed Integer Numbers Arithmetic

Signed number

- **Sign extension and avoiding erroneous results**

```
; assume memory location 0x80000 has +96 = 0110 0000 and R1=0x80000  
LDRSB    R0, [R1] ; now R0 = 000000000000000000000000000000001100000  
; assume memory location 0x80000 contains -2 = 1111 1110 and R2=0x80000  
LDRSB    R4, [R2] ; now R4 = 11111111111111111111111111111110  
  
; assume 0x80000 contains +260 = 0000 0001 0000 0100 and  
R1=0x80000  
LDRSH R0, [R1] ; R0=0000 0000 0000 0000 0000 0001 0000 0100  
  
; assume location 0x20000 has -327660=0x8002 and R2=0x20000  
LDRSH R1, [R2] ; R1=FFFF8002
```

Signed number

- Signed number multiplication: **SMULL**

LDR R1, =-3500 ; R1 = -3500 (0xFFFFF254)

LDR R0, =-100 ; R0 = -100 (0xFFFFF9C)

SMULL R2, R3, R0, R1

- Signed number comparison

CMP Rn, Op2

Op2 > Rn V = N

Op2 = Rn Z = 1

Op2 < Rn N ≠ V

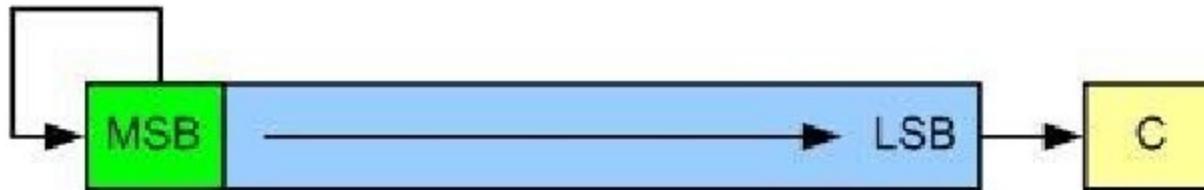
Arithmetic Shift

- ASR (arithmetic shift right)

MOV Rn, Op2, ASR count

or

ASR Rn, Op2, count



End of Chapter 5!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 22

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

ARM Memory Map, Memory Access, and Stack

Memory Addressing

- **Memory Byte Addressing in ARM**

| D31 | D24 D23 | D16 D15 | D8 D7 | D0 |
|-------------|-------------|-------------|-------------|-------------|
| 0x00000003 | 0x00000002 | 0x00000001 | 0x00000000 | 0x00000000 |
| 0x00000007 | 0x00000006 | 0x00000005 | 0x00000004 | 0x00000004 |
| 0x0000000B | 0x0000000A | 0x00000009 | 0x00000008 | 0x00000008 |
| 0x0000000F | 0x0000000E | 0x0000000D | 0x0000000C | 0x0000000C |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |
| 0xFFFFFFF3 | 0xFFFFFFF2 | 0xFFFFFFF1 | 0xFFFFFFF0 | 0xFFFFFFF0 |
| 0xFFFFFFF7 | 0xFFFFFFF6 | 0xFFFFFFF5 | 0xFFFFFFF4 | 0xFFFFFFF4 |
| 0xFFFFFFFFB | 0xFFFFFFFFA | 0xFFFFFFFF9 | 0xFFFFFFFF8 | 0xFFFFFFFF8 |
| 0xFFFFFFFFF | 0xFFFFFFFFE | 0xFFFFFFFFD | 0xFFFFFFFFC | 0xFFFFFFFFC |

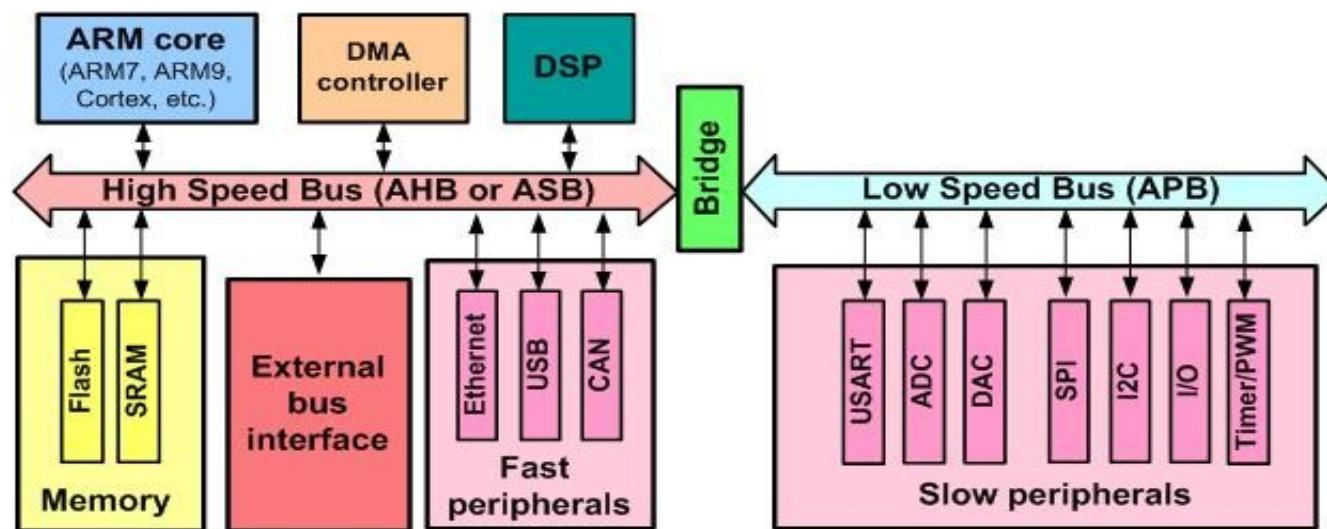
Memory Addressing

- Sample Memory Space Allocation in ARM

| Address range | Name | Description |
|---|------------|------------------------------------|
| 0x00000000-0x1FFFFFFF | Code | ROM or Flash memory |
| 0x20000000-0x3FFFFFFF [Z zeros] [Z Fs] | SRAM | SRAM region used for on-chip RAM |
| 0x40000000-0x5FFFFFFF | Peripheral | On-chip peripheral address space |
| 0x60000000-0x9FFFFFFF | RAM | Memory, cache support |
| 0xA0000000-0xDFFFFFFF | Device | Shared and non-shared device space |
| 0xE0000000-0xFFFFFFFF | System | PPB and vendor system peripherals |

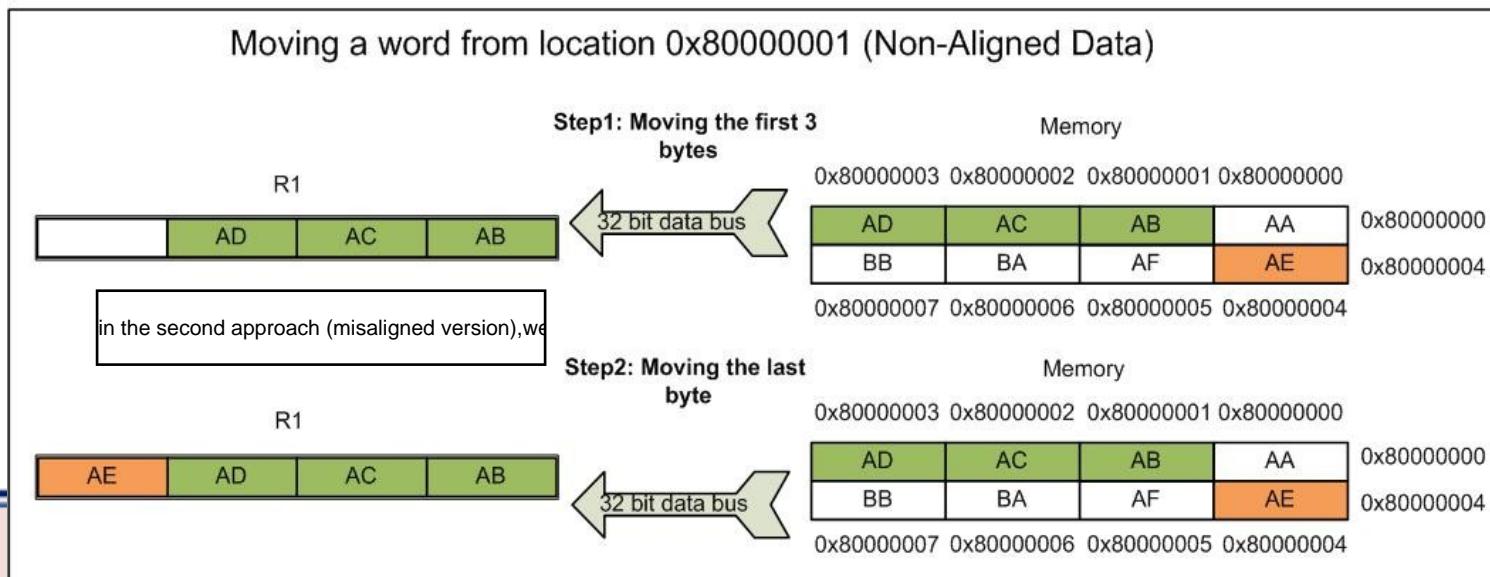
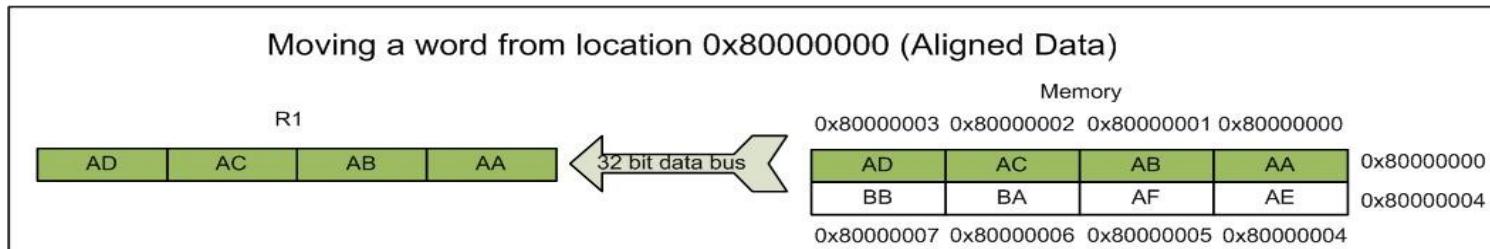
AHB and APB buses

- AHB: advanced high-performance bus
 - Connects CPU to RAM, ROM, ...
- APB: advanced peripherals bus
 - Dedicated for communication with the on-chip peripherals
 - timers, ADC, UART, SPI, I2C, ...



Data Misalignment in SRAM

- Compilers make sure that instructions are always aligned
- Placement of data in SRAM can be nonaligned
 - Memory access penalty



Data Misalignment in SRAM

LDR R1, =0x40000000 ; R1=0x40000000

LDR R2, =0x4598F31E ; R2=0x4598F31E

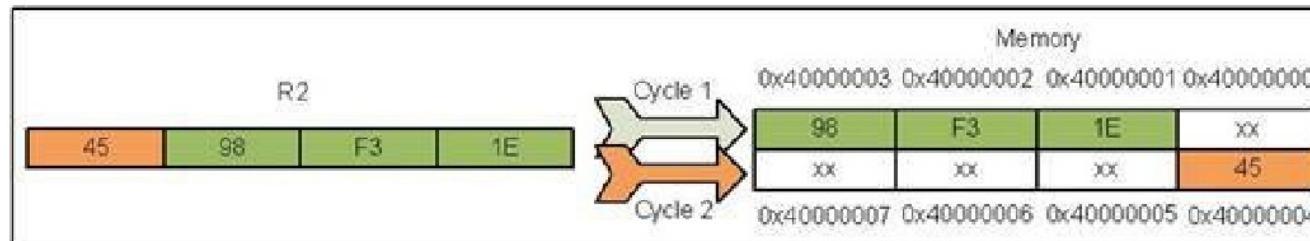
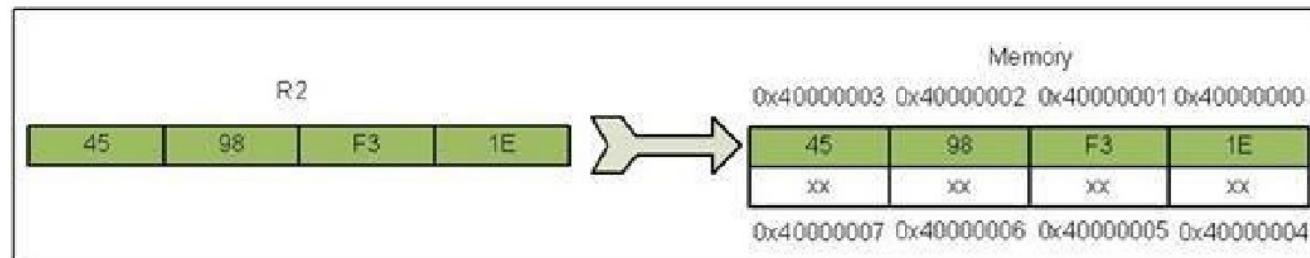
aligned

STR R2, [R1] ; Store R2 to location 0x40000000

ADD R1, R1, #1 ; R1 = R1 + 1 = 0x40000001

misaligned

STR R2, [R1] ; Store R2 to location 0x40000001



Advanced Indexed Addressing Mode

- Base plus offset addressing modes
 - Pre-indexed addressing mode with fixed offset

LDR R5, =0x55667788

LDR R1, =0x10000000; load the address of first location

STR R5, [R1] ; store R5 to location 0x10000000

STR R5, [R1, #4]; store R5 to location 0x10000000 + 4 (0x10000004)

STR R5, [R1, #8]; store R5 to location 0x10000000 + 8 (0x10000008)

STR R5, [R1, #0x0C]; store R5 to location 0x10000000 + 0x0C (0x1000000C)

Advanced Indexed Addressing Mode

- Base plus offset addressing modes
 - Pre-indexed addressing mode with writeback and fixed offset
 - The calculated pointer is written back to the pointing register
- LDR R1, =0x10000000 ; load the address of first location**
- STR R5, [R1] ; store R5 to location 0x10000000**
- STR R5, [R1, #4]!; store R5 to location 0x10000000 + 4 (0x10000004)**
; writeback makes R1 = 0x10000004
- STR R5, [R1, #4]!; store R5 to location 0x10000004 + 4 (0x10000008)**
; writeback makes R1 = 0x10000008
- STR R5, [R1, #4]!; store R5 to location 0x10000008 + 4 (0x1000000C)**
; writeback makes R1 = 0x1000000C

Advanced Indexed Addressing Mode

- Base plus offset addressing modes
 - Post-indexed addressing mode with fixed offset
 - Update pointer after the load/store operation
- STR R1, [R2], #4 ; store R1 into memory pointed to by R2 and then write back R2 + 4 to R2**
- LDRB R5, [R3], #1 ; load a byte from memory pointed to by R3 and then write back R3 + 1 to R3**

Advanced Indexed Addressing Mode

- Pre-indexed address mode with offset of a shifted register
 - Simple format

LDR Rd, [Rm, Rn] ; Rd is loaded from location Rm + Rn of memory
STR Rs, [Rm, Rn] ; Rs is stored to location Rm + Rn of memory
 - General format

LDR Rd, [Rm, Rn, <shift>] ; (Shifted Rn) + Rm is used as the address
STR Rd, [Rm, Rn, <shift>] ; (Shifted Rn) + Rm is used as the address

contents of Rm and Rn do not change!

LDR R1, [R2, R3, LSL #2] ; R2 + (R3 × 4) is used as the address
STR R1, [R2, R3, LSL #1] ; R2 + (R3 × 2) is used as the address
STRB R1, [R2, R3, LSL #2] ; R2 + (R3 × 4) is used as the address
; least significant byte of R1 is stored at location R2 + (R3 × 4)
LDR R1, [R2, R3, LSR #2] ; R2 + (R3 / 4) is used as the address

Advanced Indexed Addressing Mode

- Writeback sign (!) in pre-indexed ld/st with scaled register
 - LDR R1, [R2, R3, LSL #2]!;** R2 + (R3 × 4) is used as the address,
; content of location R2 + (R3 × 4) is loaded to R1
; R2 = R2 + (R3 × 4) (R2 is updated.)
 - STR R1, [R2, R3, LSL #1]!;** R2 + (R3 × 2) is used as the address
; R1 is stored to location R2 + (R3 × 2)
; R2 = R2 + (R3 × 2) (R2 is updated)
- Scaled register post-indexed
 - STR R1, [R2], R3, LSL #2;** store R1 at location R2 of memory
; and write back R2 + (R3 × 4) to R2.
 - LDR R1, [R2], R3, LSL #2;** load location R2 of memory to R1
; and write back R2 + (R3 × 4) to R2

To be Continued!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh

farbeh@aut.ac.ir

Department of Computer Engineering

Amirkabir University of Technology

Lecture 23

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

Stack and Stack Usage in ARM

- **Stack:** A data structure that allows easy access to the top of the stack
- **Stack in assembly:** A section of memory to store data temporarily
 - Hardware support to facilitate the creation and maintenance of stack
- **All the general registers can be used as a stack pointer**

but we don't use it in PS
- **Descending stack:** pointing to a lower address after push
- **Ascending stack:** pointing to a higher address after push
- **Empty stack:** point to where the new data will be stored
- **Full Stack:** point to where the old data will be taken off
- **instructions PUSH, POP and interrupt handling assume the stack to be full descending**

push and pop ==> pseudo instructions

Stack and Stack Usage in ARM

- Full descending stack
 - Stack pointer is pointing to the **last word** of data put onto the stack
- If the stack is **empty** (no data stored in the stack yet)
 - SP is pointing to the word **immediately below** the stack
- Initializing the stack pointer in ARM
 - The **R13 (SP)** register contains value 0 at power up
 - We must **initialize** the SP at the beginning of the program
 - Point to somewhere in the **internal SRAM**

Stack and Stack Usage in ARM

```
Stack_Top equ 0x40008000
AREA EXAMPLE_6_17, CODE, READONLY
LDR R13, =Stack_Top ; load SP
LDR R0, =0x125 ; R0 = 0x125
LDR R1, =0x144 ; R1 = 0x144
MOV R2, #0x56 ; R2 = 0x56
BL MY_SUB ; call a subroutine
ADD R3, R0, R1
ADD R3, R3, R2
HERE B HERE ; stay here
MY_SUB
; save R0, R1, and R2 on stack before use
SUB R13, R13, #4
STR R0, [R13] ; save R0 on stack
SUB R13, R13, #4
STR R1, [R13] ; save R1 on stack
SUB R13, R13, #4
STR R2, [R13] ; save R2 on stack
```

when stack pointer points to 0 means ou

0xFF=255

; ----- modify R0, R1, and R2
MOV R0, #15
MOV R1, #25
SUB R2, R0, R1
STR R2, [LR, #4]
; restore the original registers contents from stack
LDR R2, [R13] ; restore R2 from stack
ADD R13, R13, #4 ; R13 = R13 + 4 to increment the stack pointer
LDR R1, [R13] ; restore R1 from stack
ADD R13, R13, #4 ;
LDR R0, [R13] ; restore R0 from stack
ADD R13, R13, #4
BX LR ; return to caller
END

Stack and Stack Usage in ARM

- STM and LDM
 - To store and load multiple registers with a single instruction

MY_SUB

```
; -----save R0, R1, and R2 on stack before they are used by a loop
STMFA R13, {R0-R2} ; save R0, R1, R2 on stack using Full Ascending
; -----R0, R1, and R2 are changed
MOV R0, #0 ; R0=0
MOV R1, #0 ; R1=0
MOV R2, #0 ; R2=0
; -----restore the original registers contents from stack
LDMFA R13, {R0-R2}
; restore R0, R1, and R2 from stack using Full Ascending
BX LR ; return to caller
```

Stack and Stack Usage in ARM

- Options for LDM and STM instructions

| Option | Description |
|--------|------------------|
| IA | Increment After |
| IB | Increment Before |
| DA | Decrement After |
| DB | Decrement Before |

| Stack Structure | Load | Store | Load (alternate Names) | Store (alternate Names) |
|------------------|-------|-------|---------------------------|----------------------------|
| Full Ascending | LDMDA | STMIB | LDMFA | STMFA |
| Full Descending | LDMIA | STMDB | LDMFD | STMF D |
| Empty Ascending | LDMDB | STMIA | LDMEA | STMEA |
| Empty Descending | LDMIB | STMDA | LDMED | STMED |

PUSH is an alias of “STMDB R13!”

POP is an alias of “LDMIA R13!”

Subroutine Call

- **Stack frame**
 - A block of memory on the stack for a subroutine
 - Parameters are pushed onto the stack by the caller
 - The return address is pushed onto the stack from **R14**
- Subroutine moves the stack pointer to leave a block of memory space for the **local variables**
- The stack pointer is copied to **another register** to be used for access into the **stack frame**
- Each subroutine should **preserve** any register it is going to use and **restore** them before return

ARM Bit-Addressable Memory Region

- **Bit-banding option**
 - Generally available in M3 and M4
 - To mitigate the issues of Read-Modify-Write
 - Only few small regions are bit-banded

| SRAM Byte addresses | SRAM Bit addresses (We use these addresses to access the individual bits) | | | | | | | |
|---------------------|--|-----|-----|-----|-----|-----|-----|------------|
| | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 |
| 200FFFFF | 23FFFFFFC | F8 | F4 | F0 | EC | E8 | E4 | 23FFFFFFE0 |
| 200FFFFE | 23FFFFFFDC | D8 | D4 | D0 | CC | C8 | C4 | 23FFFFFFC0 |
| 200FFFFD | 23FFFFFFBC | B8 | B4 | B0 | AC | A8 | A4 | 23FFFFFFA0 |
| 200FFFFC | 23FFFF9C | 98 | 94 | 90 | 8C | 88 | 84 | 23FFFF80 |
| 200FFFFB | 23FFFF7C | 78 | 74 | 70 | 6C | 68 | 64 | 23FFFF60 |
| 200FFFFA | 23FFFF5C | x8 | x4 | x0 | xC | x8 | x4 | 23FFFF40 |
| 200XXXXX | 2XXXXXXC | X8 | X4 | X0 | XC | X8 | X4 | 2XXXXXXX0 |
| 20000008 | 2200011C | 118 | 114 | ... | 104 | 102 | 100 | 22000100 |
| 20000007 | 220000FC | F8 | F4 | F0 | EC | E8 | E4 | 220000E0 |
| 20000006 | 220000DC | D8 | D4 | D0 | CC | C8 | C4 | 220000C0 |
| 20000005 | 220000BC | B8 | B4 | B0 | AC | A8 | A4 | 220000A0 |
| 20000004 | 2200009C | 98 | 94 | 90 | 8C | 88 | 84 | 22000080 |
| 20000003 | 2200007C | 78 | 74 | 70 | 6C | 68 | 64 | 22000060 |
| 20000002 | 2200005C | 58 | 54 | 50 | 4C | 48 | 44 | 20000040 |
| 20000001 | 2200003C | 38 | 34 | 30 | 2C | 28 | 24 | 22000020 |
| 20000000 | 2200001C | 18 | 14 | 10 | 0C | 08 | 04 | 22000000 |

ARM Bit-Addressable Memory Region

- A program to set HIGH the D6 of the SRAM location 0x20000001

LDR R1, =0x20000001 ; load the address of the byte

LDRB R2, [R1] ; get the byte

ORR R2, R2, #2_01000000 ; make D6 bit high

; (binary representation in Keil for 0b01000000)

STRB R2, [R1] ; write it back

LDR R1, =0x22000038 ; load the alias address of the bit

MOV R2, #1 ; R2 = 1

STR R2, [R1] ; Write one to D6

ADR, LDR, and PC Relative Addressing

- Using PC (R15) register as the pointer register
LDR R0, [PC, #4]
- What is the value of R0 if LDR is in address 0x00000004?
 $R0 = 0x00000004 + 4 + 8$
- The ADR Pseudo-instruction
ADR Rn, Label ↗ ADD Rn, PC, #offset
- Implementing the LDR Pseudo-instruction
LDR R2, =0x12345678
 - Assembler stores the value as a constant in program memory
LDR R2, [PC, #0x0008]

End of Chapter 6!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh
farbeh@aut.ac.ir

Department of Computer Engineering
Amirkabir University of Technology

Lecture 24

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

Appendix C: Macros

Macro

- Write the task once and invoke it whenever it is needed
- MACRO definition

MACRO

[\$label] macroName parameter1, parameter2, ..., parameterN

... ...

... ...

MEND

MACRO

ADD3VAL \$DEST, \$ARG1, \$ARG2, \$ARG3

ADD \$DEST, \$ARG1, \$ARG2

ADD \$DEST, \$DEST, \$ARG3

MEND

AREA CODE1, READONLY,
CODE

MOV R1, #5

MOV R2, #2

ADD3VAL R0, R1, R2, #5

- To distinguish parameters, they must start with \$

Macro

- Default Values for parameters

MACRO

ADD3VAL \$DEST, \$ARG1=R3, \$ARG2, \$ARG3=#5

ADD \$DEST, \$ARG1, \$ARG2

ADD \$DEST, \$DEST, \$ARG3

MEND

- To use the default value, put a ‘|’ instead of the parameter

ADD3VAL R0, R1, R2, |

Using Labels in Macros

- Labels must be unique

MACRO

\$lbl OUR_MACRO

CMP R1,#5

BEQ \$lbl

MOV R1, #1

\$lbl

MEND

AREA OURCODE, READONLY, CODE ENTRY

MOV R1, #3

Label1 OUR_MACRO

MOV R1, #5

Label2 OUR_MACRO

HERE B HERE

Using Labels in Macros

- Multiple labels inside a macro

| | |
|------------------------------|-------------------------------|
| MACRO | MOV R1, #3 |
| \$lbl OUR_MACRO | CMP R1, #5 |
| CMP R1, #5 | BEQ \$lbl.equal |
| MOV R1, #1 | MOV R1, #1 |
| B \$lbl.next | B label1next |
| \$lbl.equal | label1equal MOV R1, #2 |
| MOV R1, #2 | label1next MOV R1, #5 |
| \$lbl.next | CMP R1, #5 |
| MEND | BEQ label2equal |
| AREA OURCODE, READONLY, CODE | MOV R1, #1 |
| MOV R1, #3 | B label2next |
| label1 OUR_MACRO | label2equal MOV R1, #2 |
| MOV R1, #5 | label2next HERE B HERE |
| Label2 OUR_MACRO | |
| HERE B HERE | |

Conditional Macros

- We can pass condition into macros

MACRO

\$lbl OurMacro\$cond

CMP R1, #5

B\$cond \$lbl.equal

MOV R1, #1

\$lbl.equal

MEND

AREA OURCODE, READONLY, CODE

MOV R1, #3

label1 OurMacroEQ ; in the macro check equality

MOV R1, #3

label2 OurMacroLO ; in the macro check if is lower

HERE B HERE

MOV R1, #3

label1 OurMacroEQ

CMP R1, #5

BEQ label1equal

MOV R1, #1

label1equal

MOV R1, #3

label2 OurMacroLO

CMP R1, #5

BLO label2equal

MOV R1, #1

label2equal HERE B HERE

MACRO from Files

- Include Directive

The image shows a debugger interface with two windows side-by-side.

prog.asm:

```
1 AREA OURCODE, READONLY, CODE
2 INCLUDE MyMacro.s
3 ENTRY
4 MOV R1, #5
5 MOV R2, #2
6 ADD3VAL R0, R1, R2, #5
7 H1 B H1
8 END
```

MyMacro.s:

```
1 MACRO
2 ADD3VAL $DEST, $ARG1, $ARG2, $ARG3
3 ADD $DEST, $ARG1, $ARG2
4 ADD $DEST, $DEST, $ARG3
5 MEND
6 END
```

End of Appendix C!



Microprocessors and Assembly Language

Spring 2020

Hamed Farbeh
farbeh@aut.ac.ir

Department of Computer Engineering
Amirkabir University of Technology

Lecture 25

Copyright Notice

Parts (text & figures) of this lecture are adopted from:

- **Arm Assembly Language Programming and Architecture, Volume 1, 1st edition, Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi, MicroDigitalEd, 2013.**

Appendix E: Passing Arguments into Functions

Passing Arguments

- Three ways to pass arguments (parameters) to functions
 - Through registers
 - Through memory using references
 - Using stack

Passing Arguments

- Passing arguments through registers

```
AREA OUR_PROG, CODE, READONLY
    MOV R0, #5 ; R0 = 5
    MOV R1, #7 ; R1 = 7
    BL BIGGER ; BIGGER(5, 7)
    HERE B HERE ; stay here
; =====
; BIGGER returns the bigger value
; Parameters: R0 and R1: the values to be compared
; Returns: R0: containing the bigger value
; =====
BIGGER CMP R0, R1
    BHI L1 ; if R0 > R1 go to L1
    MOV R0, R1 ; R0 = R1
L1      BX LR ; return
END
```

Passing Arguments

- Passing through memory using references
 - Store the data in memory and pass its address through a register

```
ADR    R0, OUR_STR ; R0 = addr. of OUR_STR
BL    STR_LENGTH ; STR_LENGTH(&OUR_STR)
HERE   B HERE ; stay here
OUR_STR  DCB    "HELLO!"
; =====
STR_LENGTH MOV R1, R0 ; move string pointer to R1
              MOV R0, #0 ; use R0 as string length counter
L_BEGIN    LDRB R2, [R1] ; fetch a character from string
              CMP R2, #0
              BXEQ LR ; return if character is null (end of string)
              ADD R1, R1, #1 ; point to next character in string
              ADD R0, R0, #1 ; increment the counter
              B L_BEGIN
```

we end our string with zero

Passing Arguments

- **Passing arguments through stack**
 - The arguments are pushed onto the stack just before calling the function and popped off after returning

```
LDR SP, =(0x40000000+(16*1024)) ; init stack pointer
MOV R0, #5
PUSH {R0} ; push Arg1
MOV R0, #7
PUSH {R0} ; push Arg2
BL BIGGER ; BIGGER(5, 7)
ADD SP, SP, #8 ; adjust the stack pointer to remove the arguments
HERE B HERE ; stay here
; =====
BIGGER
LDR R0, [SP, #4] ; R0 = arg1
LDR R1, [SP, #0] ; R1 = arg2
CMP R0, R1
MOVLO R0, R1 ; if R0 < R1 move R1 into R0
L1 BX LR ; return
```

Passing Arguments

- In ARM CPU, the arguments are passed in the **first four registers**
 - If there are **four** or fewer arguments
- If there are **more than four** arguments
 - The **first four** are passed in the first four **registers**
 - The **rest** are passed on the **stack**

ARM Application Procedure Call Standard

- AAPCS provides a standard for implementing the functions
- Some rules
 - The arguments must be sent through R0 to R3
 - The return value must be returned in R0 (and R1 for return a 64-bit)
 - The functions can use R4 to R8, R10 and R11 for temporary storage
 - Their values must be saved upon entering the function and restored before returning
 - The stack must be used as Full Descending

End of Appendix E!