Introduction to Microcontrollers

中山 大 学

数据科学与计算机学院

郭雪梅

Tel:39943108

Email:guoxuem@mail.sysu.edu.cn

URL1：http://human-robot.sysu.edu.cn/course

http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C2_FundamentalConcepts.htm

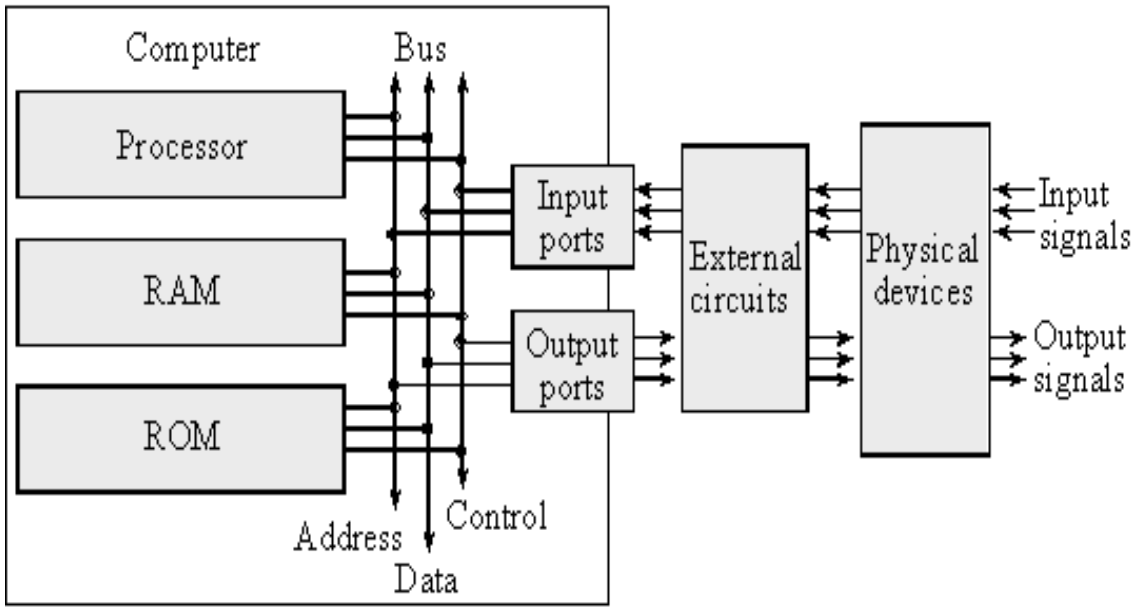
http://users.ece.utexas.edu/~valvano/Volume1/E-Book/C7_DesignDevelopment.htm

Agenda

- Course Description
  - Book, *INTRODUCTION TO ARM CORTEX-M MICROCONTROLLERS*
  - Lectures: Tuesday 8:00-9:40, 10:00-11:40(d203) 14:20-18:00, d203
- Embedded Systems
  - Microcontrollers
- ARM Architecture
  - Programming
  - Integrated Development Environment (IDE)

Introduction to Computers

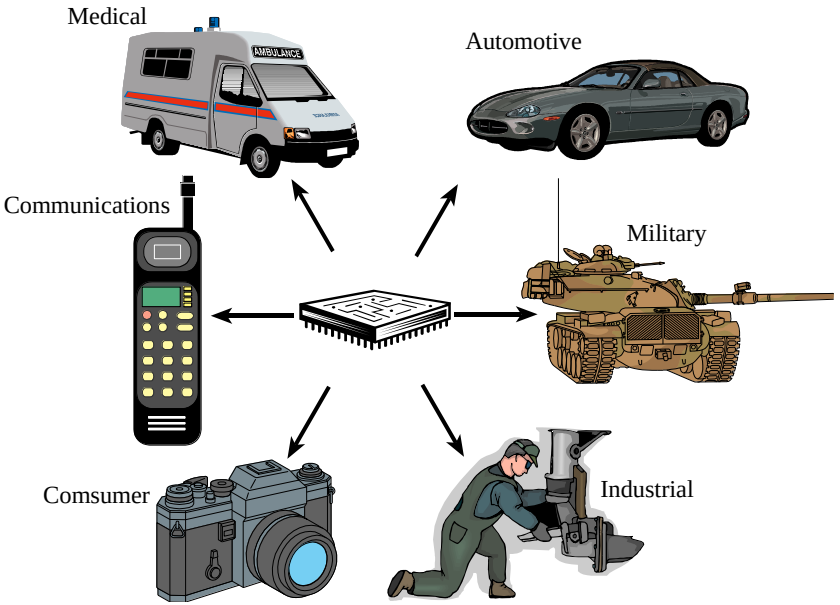
A **computer** combines a processor, random access memory (RAM), read only memory (ROM), and input/output (I/O) ports.

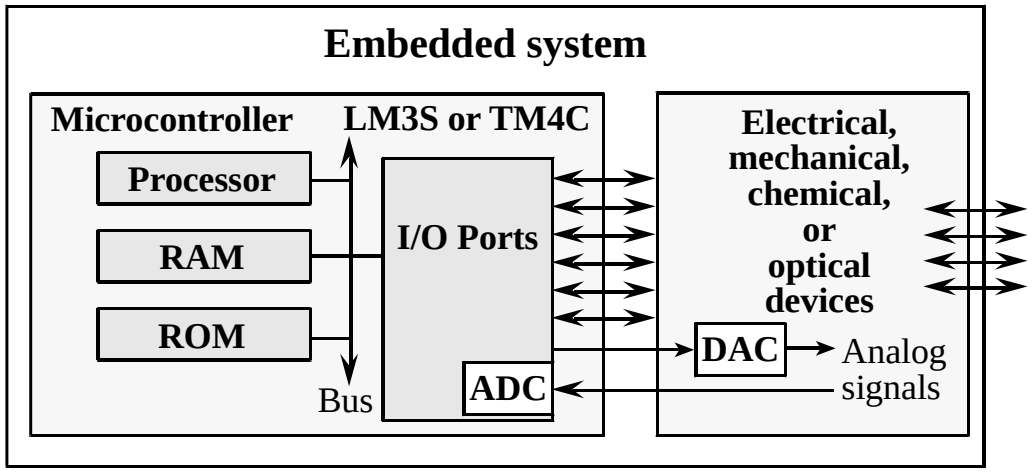


A **port** is a physical connection between the computer and its outside world. Ports allow information to enter and exit the system.

A **bus** is a collection of wires used to pass information between modules.

Embedded System





- Embedded Systems are everywhere
  - Ubiquitous, invisible
  - Hidden (computer inside)
  - Dedicated purpose
- MicroProcessor
  - Intel: 4004, ..8080,.. x86
  - Freescale: 6800, .. 9S12,.. PowerPC
  - ARM, DEC, SPARC, MIPS, PowerPC, Natl. Semi.,…
- MicroController
  - Processor+Memory+ I/O Ports (Interfaces)

A very small microcomputer, called a **microcontroller**, contains all the components of a computer (processor, memory, I/O) on a single chip.

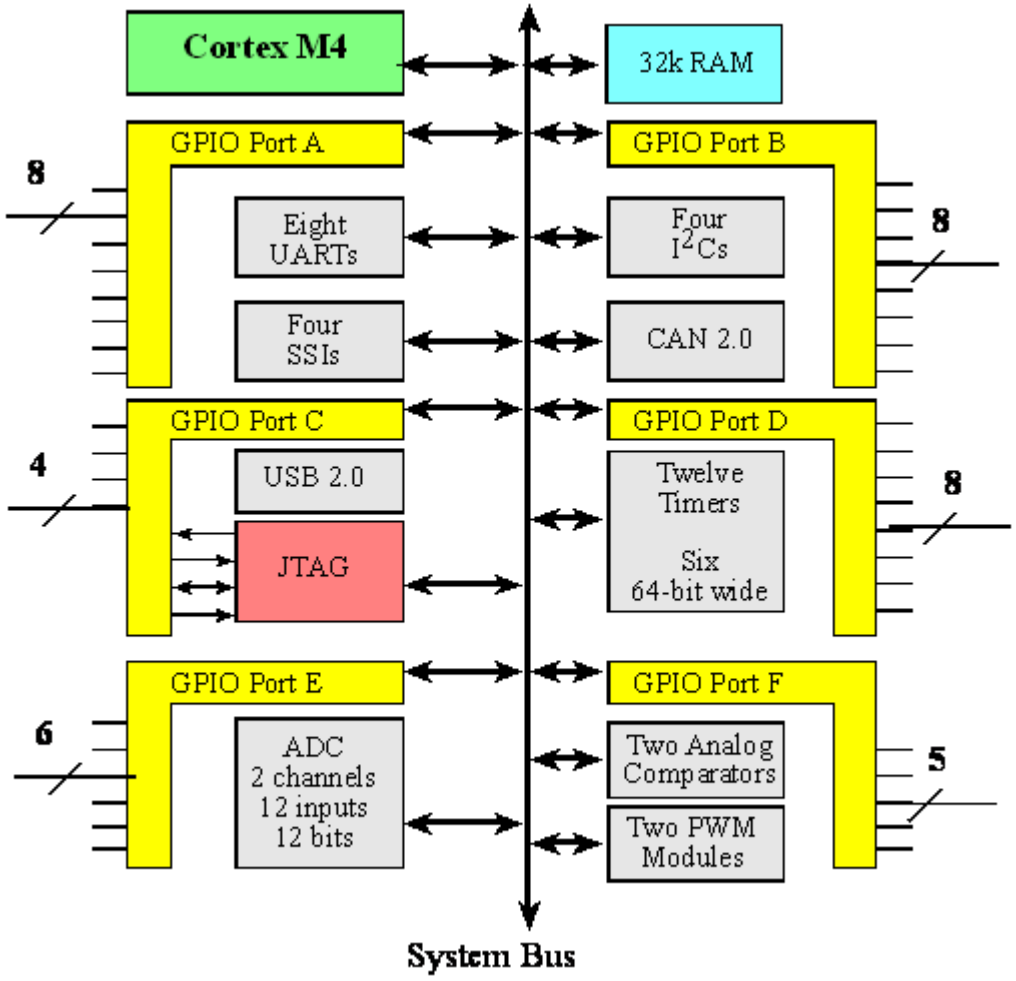
Embedded Systems

- A *reactive system* continuously
  - accepts inputs
  - performs calculations
  - generates outputs
- A *real time system*
  - Specifies an upper bound on the time required to perform the input/calculation/output in reaction to external events
  - Interacts with physical environment

Microcontroller

- Processor – Instruction Set + memory + accelerators
  - Ecosystem
- Memory
  - Non-Volatile
    - ROM
    - EPROM, EEPROM, Flash
  - Volatile
    - RAM (DRAM, SRAM)
- Interfaces
  - H/W: Ports
  - S/W: Device Driver
  - Parallel, Serial, Analog, Time
- I/O
  - Memory-mapped vs. I/O-instructions (I/O-mapped)

Texas Instruments TM4C123



ARM Cortex-M4
+ 256K EEPROM
+ 32K RAM
+ JTAG
+ Ports
+ SysTick
+ ADC
+ UART

ARM体系结构的技术特征及发展

ARM三层含义：

ARM是一个公司的名字,Advanced RISC Machines Ltd,主要设计**RISC**系列处理器内核

ARM代表一项技术

体积小、低功耗、低成本、高性能

- 支持**Thumb**（**16**位）**/ARM**（**32**位）双指令集，能很好的兼容**8**位**/16**位器件
- 大量使用寄存器，指令执行速度更快
- 大多数数据操作都在寄存器中完成
- 寻址方式灵活简单，执行效率高
- 指令长度固定

ARM是一类微处理器的统称

ARM架构

- ARM处理器版本指体系结构，产生ARM系列产品

  ARM架构主要是以指令集来区分，从开发先后顺序来看，ARM指令集目前已经发展到第七代

  ARM产品家族一般对应不同体系架构；

  - ✓ V1版架构
  - ✓ V2版架构
  - ✓ V3版架构
  - ✓ V4版架构
  - ✓ V5版架构
  - ✓ V6版架构
  - ✓ V7版架构

ARM Architecture versions and products

Key architecture revisions and products:

ARMv1-ARMv3: largely lost in the mists of time

ARMv4T: ARM7TDMI – first Thumb processor

ARMv5TEJ(+VFPv2): ARM926EJ-S

ARMv6K(+VFPv2): ARM1136JF-S, ARM1176JFZ-S, ARM11MPCore – first Multiprocessing Core

ARMv7-A+VFPv3 Cortex-A8

ARMv7-A+MPE+VFPv3: Cortex-A5, Cortex-A9

ARMv7-A+MPE+VE+LPAE+VFPv4 Cortex-A15

ARMv7-R : Cortex-R4, Cortex-R5

ARMv6-M Cortex–M0

ARMv7-M: Cortex-M3, Cortex-M4

ARMv7架构三种款式

ARM v7-A

设计用于高性能的“开放应用平台”——越来越接近电脑了,需要运行复杂应用程序的“应用处理器”支持大型嵌入式操作系统

ARM v7-R

实时且高性能的处理器,像高档轿车的组件，大型发电机控制器，机器手臂控制器等，处理器不但要很好很强大，还要极其可靠，对事件的反应也要极其敏捷.

ARM v7-M

单片机实时控制系统，低成本、低功耗、极速中断反应以及高处理效率，深度嵌入式。

Cortex-M3就是按款式M设计

Cortex-M3 Overview

ARM CorteX-M3处理器技术特点

Cortex-M3处理器为ARM Cortex系列处理器CPU核心的第一款产品。

此款处理器特别针对对价格敏感但又具备高系统效能需求的嵌入式应用设计，包括仪器仪表、汽车电子、家电及网络装置等应用。

典型产品：**TI（Luminary）LMS3XX系列**

NXP LPC1XX系列

ST STM32系列

Cortex-M3 Processor

- **Greater performance efficiency**: more work to be done without increasing the frequency or power requirements
  - –Implements the new Thumb-2 instruction set architecture
    - 70% more efficient per MHz than an ARM7TDMI-S processor executing Thumb instructions
    - 35% more efficient than the ARM7TDMI-S processor
- **Low power consumption**: longer battery life, especially critical in portable products including wireless networking applications
- **Improved code density**: code fits in even the smallest memory footprints
- Core pipeline has 3 stages

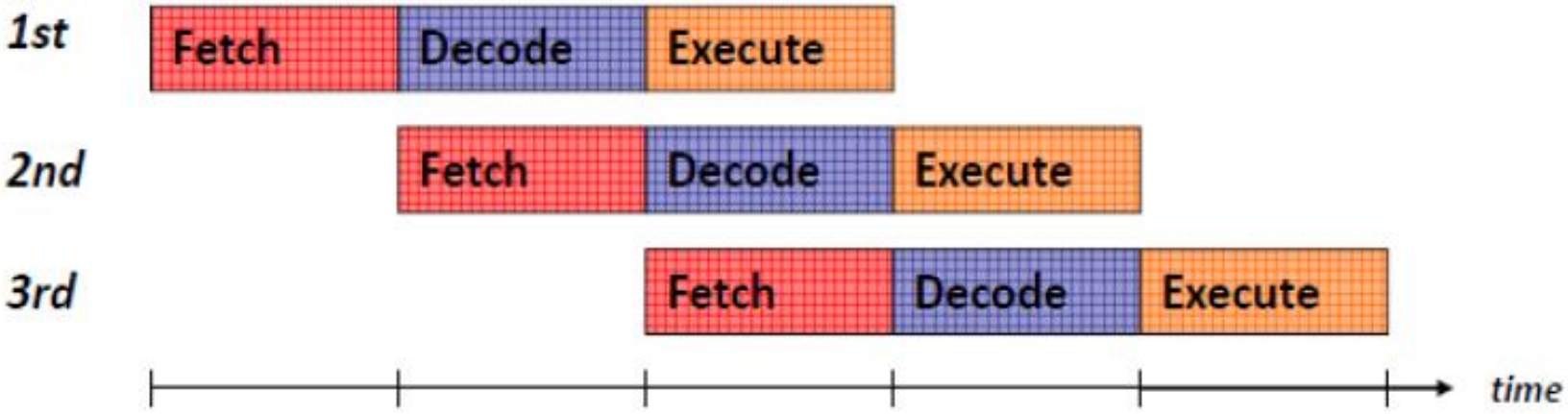
–Instruction Fetch

–Instruction Decode

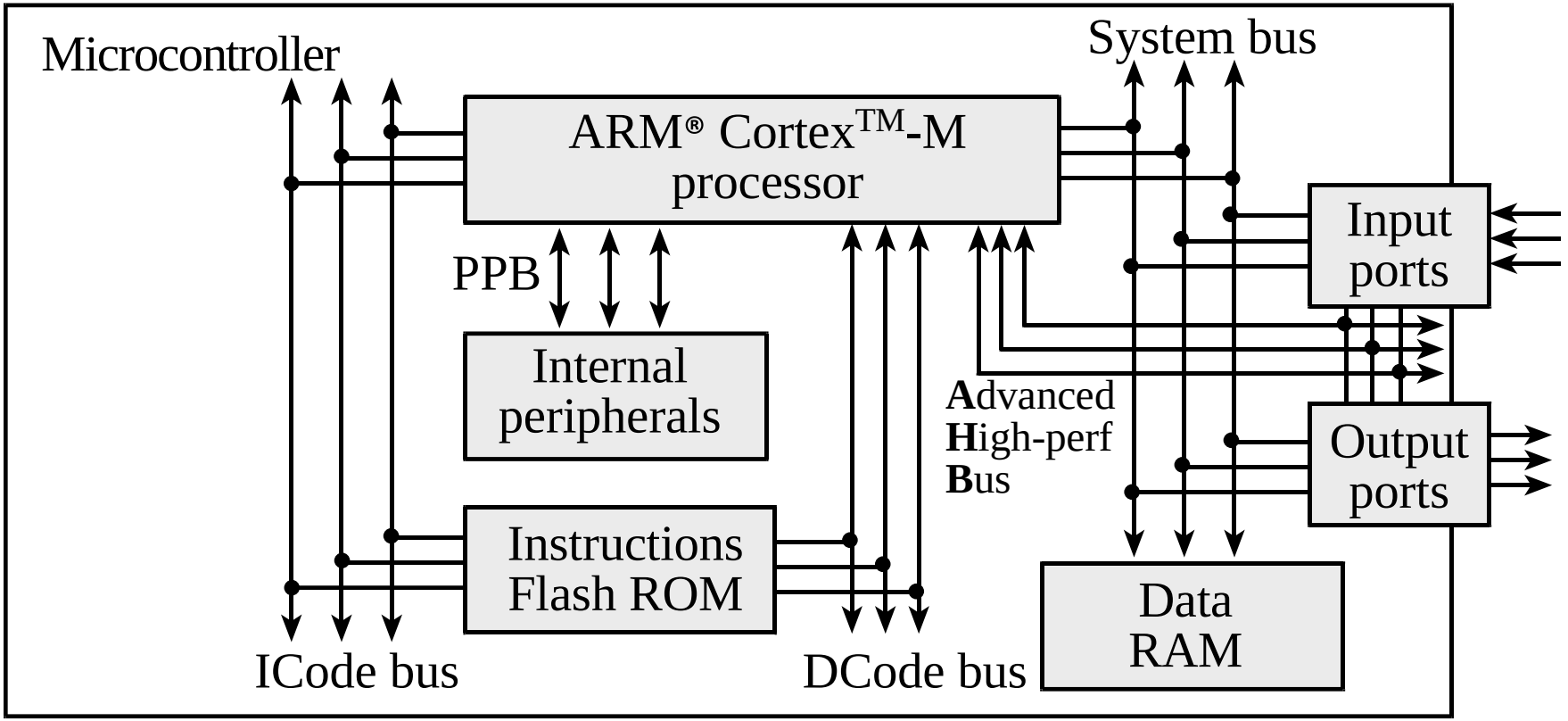
–Instruction Execute

Cortex-M3 Pipeline

- The Cortex-M3 Uses the 3-stage pipeline for instruction executions
  - Fetch ⇒ Decode ⇒ Execute
  - Pipeline design allows effective throughput to increase to one instruction per clock cycle
  - Allows the next instruction to be fetched while still decoding or executing the previous instructions



ARM Cortex M4-based System



- **ARM Cortex-M4 processor**
- ***Harvard* architecture**
  - **Different busses for instructions and data**

Instructions are fetched from flash ROM using the ICode bus. Data are exchanged with memory and I/O via the system bus interface.

I-Code 总线、D-Code 总线

- I-Code 总线是一条基于AHB总线协议的32 位总线，负责在 0x0000_0000 –0x1FFF_FFFF 之间的取指操作。
- 取指以字的长度执行，即使是对于16 位指令也如此。
- 因此CPU 内核可以一次取出两条16 位Thumb 指令。

D-Code 总线也是一条基于AHB 总线协议的32 位总线，负责在 0x0000_0000 –0x1FFF_FFFF 之间的数据访问操作。

M4比M3增加了浮点处理器FPU，优化的DSP指令也可在单周期内完成。其他二者并无实质上的区别。

Processor Modes

- The ARM has seven basic operating modes:
  - Each mode has access to:
    - Its own stack space and a different subset of registers
  - Some operations can only be carried out in a privileged mode

Exception modes

Mode	Description	
Supervisor (SVC)	Entered on reset and when a Software Interrupt instruction (SWI) is executed	**Privileged modes**
FIQ	Entered when a high priority (fast) interrupt is raised	
IRQ	Entered when a low priority (normal) interrupt is raised	
Abort	Used to handle memory access violations	
Undef	Used to handle undefined instructions	
System	Privileged mode using the same registers as User mode	
User	Mode under which most Applications / OS tasks run	**Unprivileged mode**

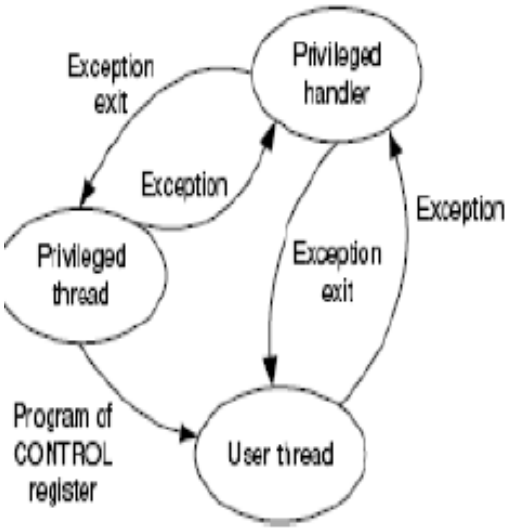
Operating Modes

User mode:

- Normal program execution mode
- System resources unavailable
- Mode changed by exception only

Exception modes:

- Entered upon exception
- Full access to system resources
- Mode changed freely



Modes (Thread out of reset)		Operations (privilege out of reset)	Stacks (Main out of reset)
	Handler - An exception is being processed	Privileged execution Full control	Main Stack Used by OS and Exceptions
	Thread - No exception is being processed - Normal code is executing	Privileged/Unprivileged	Main/Process

Exceptions

Exception	Mode	Priority	IV Address
Reset	Supervisor	1	0x00000000
Undefined instruction	Undefined	6	0x00000004
Software interrupt	Supervisor	6	0x00000008
Prefetch Abort	Abort	5	0x0000000C
Data Abort	Abort	2	0x00000010
Interrupt	IRQ	4	0x00000018
Fast interrupt	FIQ	3	0x0000001C

Table 1 - Exception types, sorted by Interrupt Vector addresses

Processor Register Set

- Cortex-M3 core has **16 user-visible registers**
  - –All processing takes place in these registers
- Three of these registers have dedicated functions
  - –**program counter (PC)** - holds the address of the next instruction to execute
  - –**link register (LR)** - holds the address from which the current procedure was called
  - – **"the" stack pointer (SP)** - holds the address of the current stack top (CM3 supports multiple execution modes, each with their own private stack pointer).
- **Processor status register (PSR)** which is implicitly accessed by many instructions

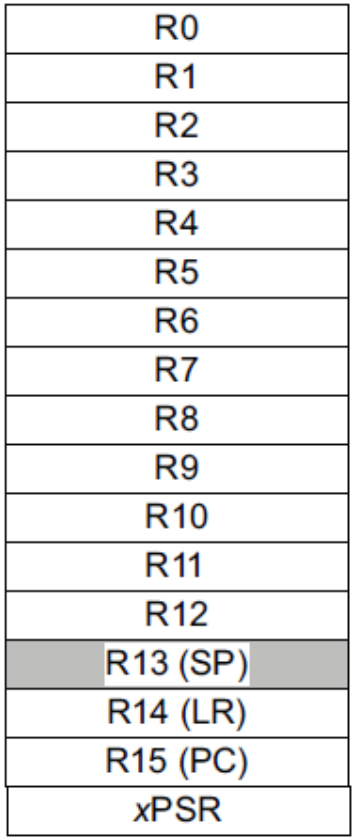
ARM Cortex-M3 ISA
Instruction Set Architecture

Instruction Set

```
ADD Rd, Rn, <op2>
```

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

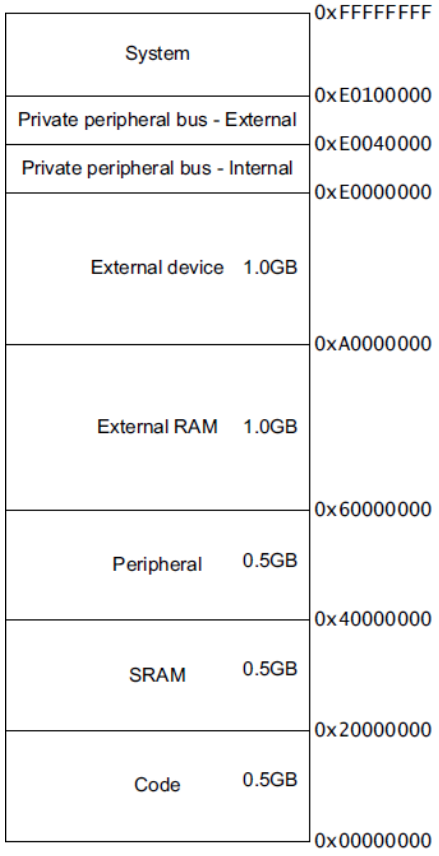
Register Set



32-bits

Endianess

Address Space

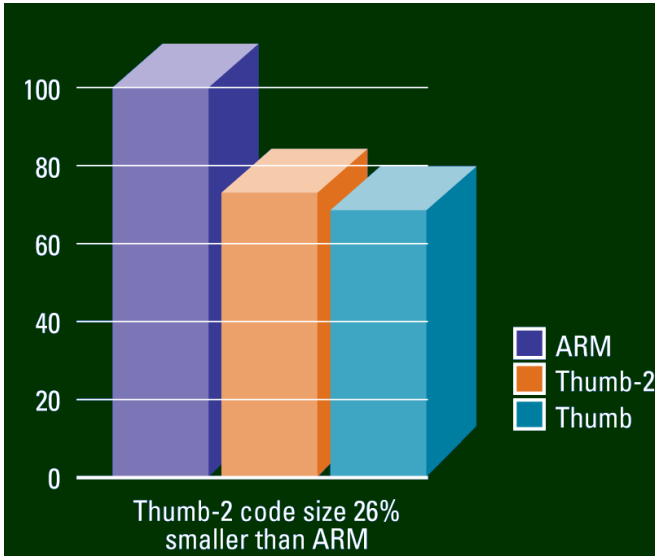


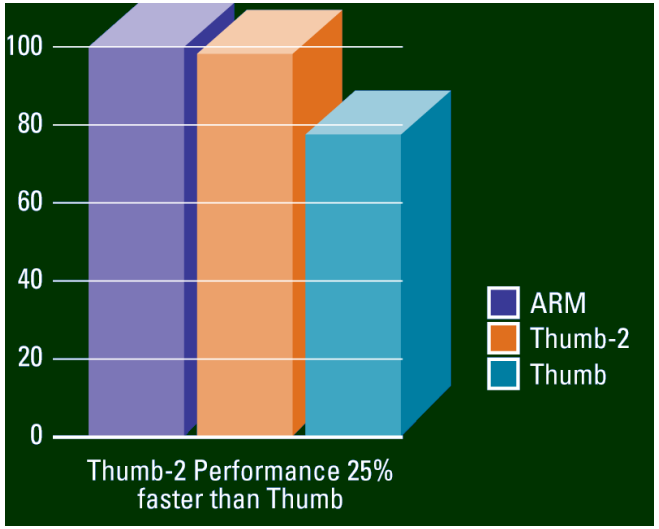
32-bits

Endianess

ARM ISA: Thumb2 Instruction Set

- Variable-length instructions
  - ARM instructions are a fixed length of 32 bits
  - Thumb instructions are a fixed length of 16 bits
  - Thumb-2 instructions can be either 16-bit or 32-bit
- Thumb-2 gives approximately 26% improvement in code density over ARM
- Thumb-2 gives approximately 25% improvement in performance over Thumb

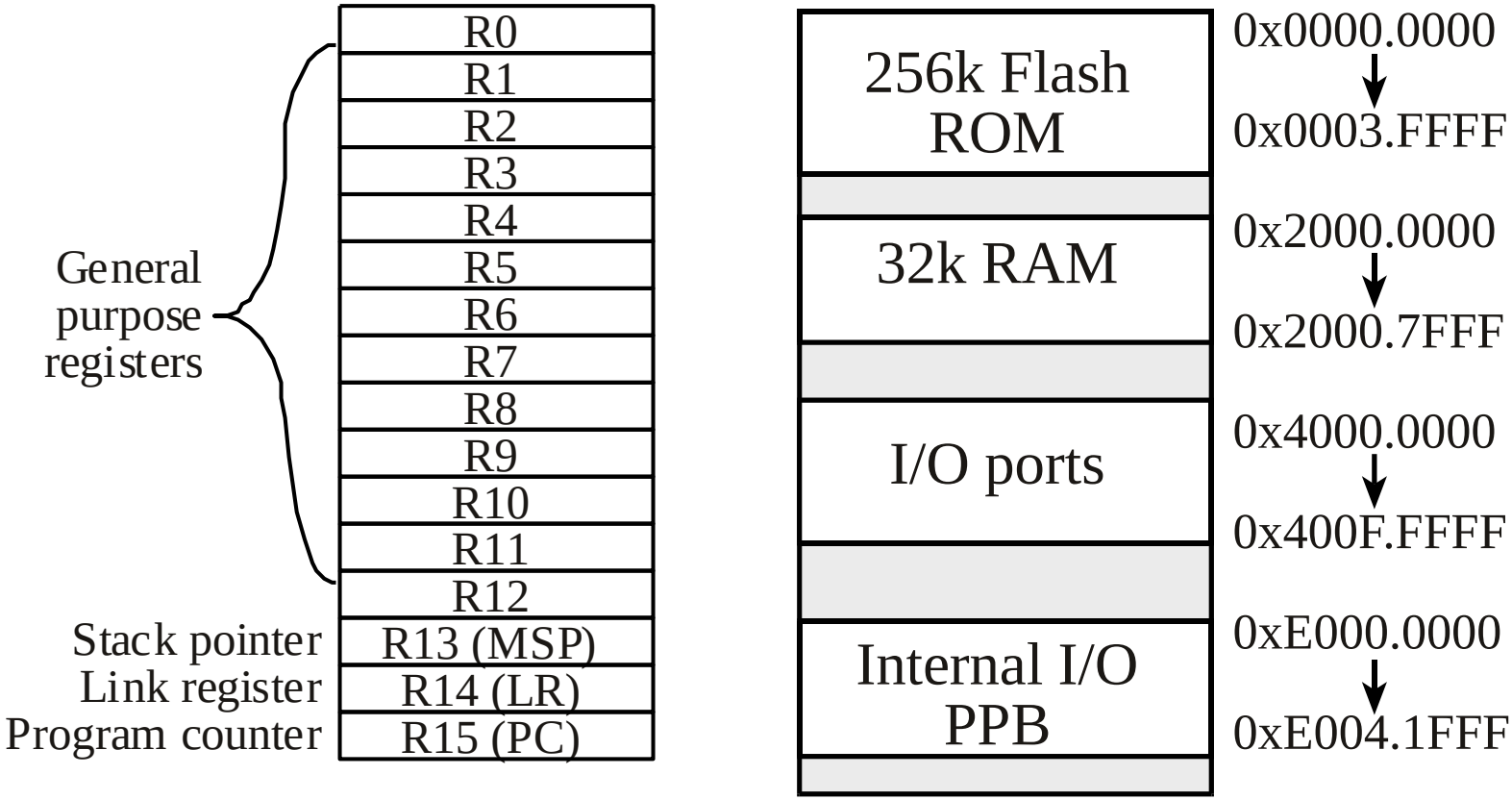




Outline of ARM Cortex-M4

1. 32-bit微处理器： 32-bit 数据路径、寄存器组、存储器接口。
2. 哈佛架构: 独立的指令总线和数据总线。这允许指令和数据在同一时间产生。
3. 存储空间: 4GB。
4. 寄存器: 寄存器(R0 到R15) 和特殊寄存器。
5. 运行模式: 线程模式和处理模式;特权级和用户级。
6. 中断和异常: 内置在嵌套向量中断控制器; 支持11 种系统异常外加240种外部IRQ。
7. 总线接口: 若干总线接口允许Cortex-M3 同时取指令和取数据。
8. MPU: 一个可选的存储器保护单元允许对特权访问和用户程序访问制定访问规则。
9. 指令集: Thumb-2 指令集; 允许32-位指令和16-位指令被同时使用。
10. 固定的内部调试组件: 提供调试操作支持和像断点调试这样的功能。

ARM ISA: Registers, Memory-map



Condition Code Bit	*s*	*Indicates*
N	**negative**	**Result is negative**
Z	**zero**	**Result is zero**
V	**overflow**	**Signed overflow**
C	**carry**	**Unsigned overflow**

TI TM4C123 Microcontroller

R13: 堆栈指针

- Cortex-M3 拥有两个堆栈指针，任一时刻只能使用其中的一个。
  - 主堆栈指针（MSP）：复位后缺省使用的堆栈指针，用于操作系统内核以及异常处理。
  - 进程堆栈指针（PSP）：由用户的应用程序代码使用。

R14：连接寄存器

- 当调用一个子程序时，为了减少访问内存的次数，由R14存储返回地址，把返回地址直接存储在寄存器中，这与其他大多数其它处理器都不一样。

- 只有1 级子程序调用的代码无需访问内存，从而提高了子程序调用的效率。如果多于1 级，则需要把前一级的R14值压到堆栈里。

- 在ARM编程时，应尽量只使用寄存器保存中间结果，迫不得以时才访问内存。

R15：程序计数寄存器（PC）

- 指向**当前的程序地址**。
- 如果**修改它的值**，就能改变程序的执行。

Data Sizes and Instruction Sets

- The ARM is a 32-bit architecture.
- When used in relation to the ARM:

  Byte means 8 bits

  Halfword means 16 bits (two bytes)

  Word means 32 bits (four bytes)

  Most ARM's implement two instruction sets

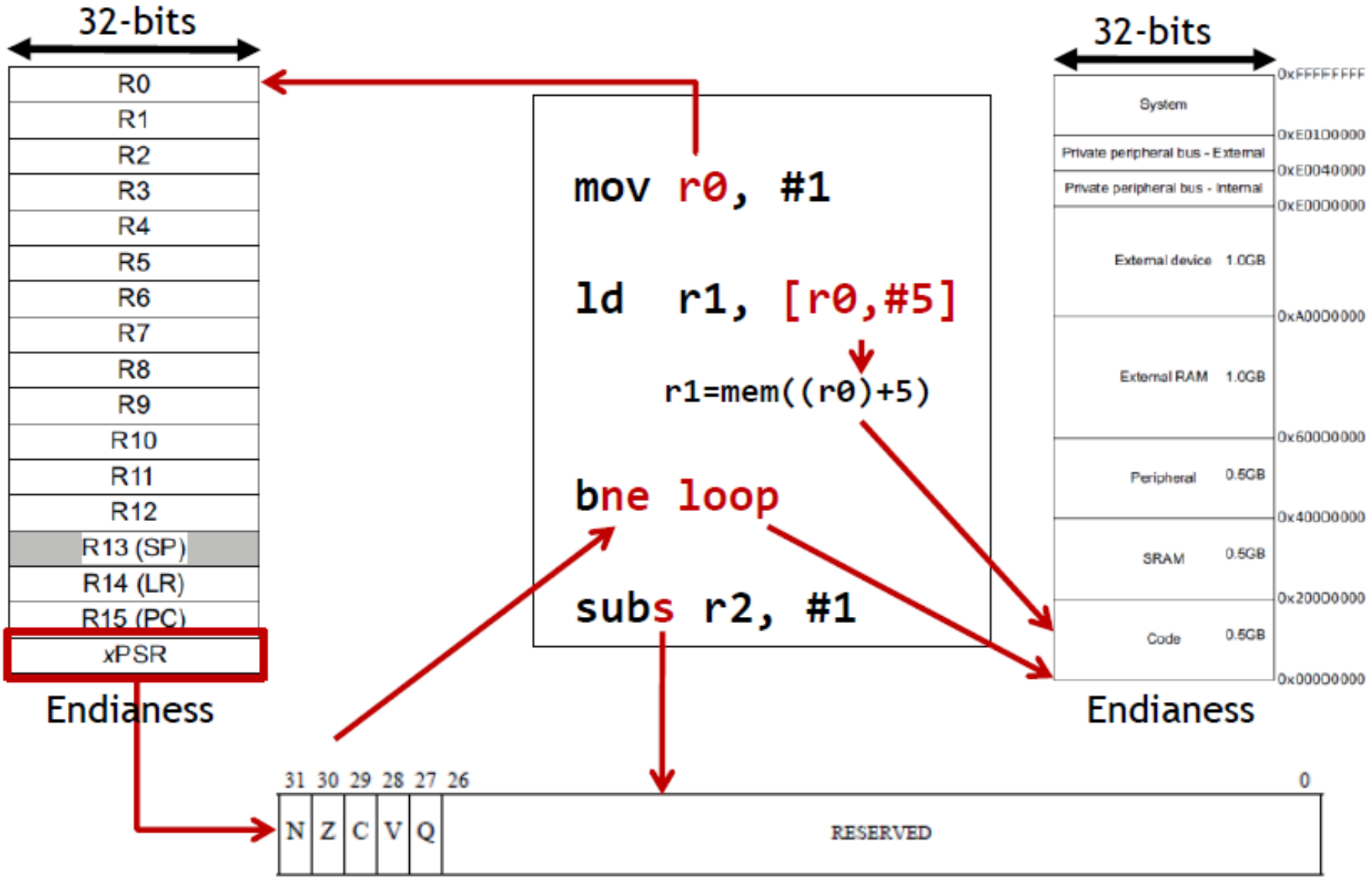
  32-bit ARM Instruction Set

  16-bit/32bit Thumb Instruction Set

- Cortex—M4

  Thumb-2 Blended 16-bit and 32-bit instruction set

Major Elements of ISA



Instruction Encoding

- Instructions are encoded in machine language opcodes

Instructions
movs r0, #10
movs r1, #0

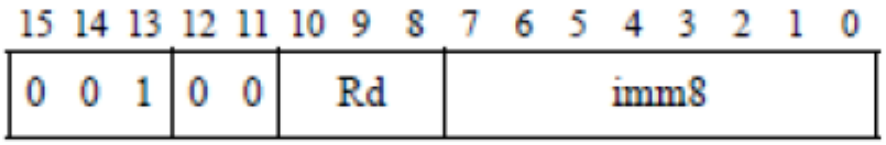
Register Value	Memory Value
001|00|000|00001010	(LSB) (MSB)
(msb) (lsb)	0a 20 00 21
001|00|001|00000000	

ARMv7 ARM

Encoding T1 All versions of the Thumb ISA.

MOVS <Rd>,#<imm8> Outside IT block.

MOV<c> <Rd>,#<imm8> Inside IT block.

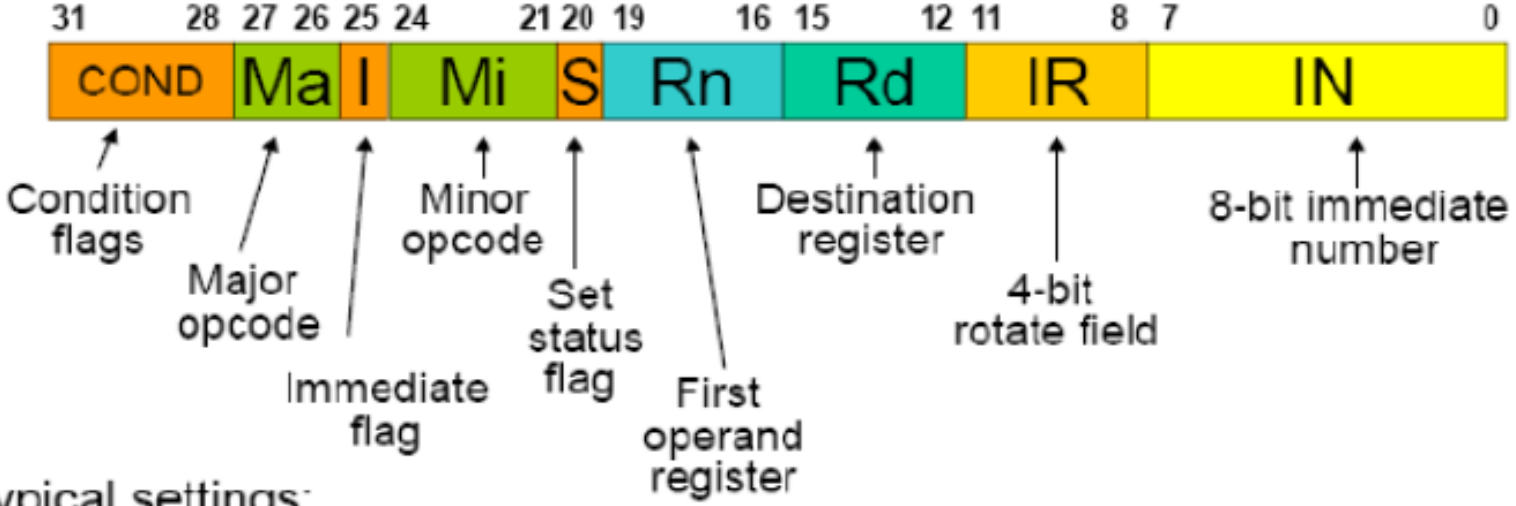



```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

32bit Instruction Encoding

Example: ADD instruction format

- ARM 32-bit encoding for ADD with immediate field



Typical settings:

Major opcode = 00 (this indicates data operation instructions)

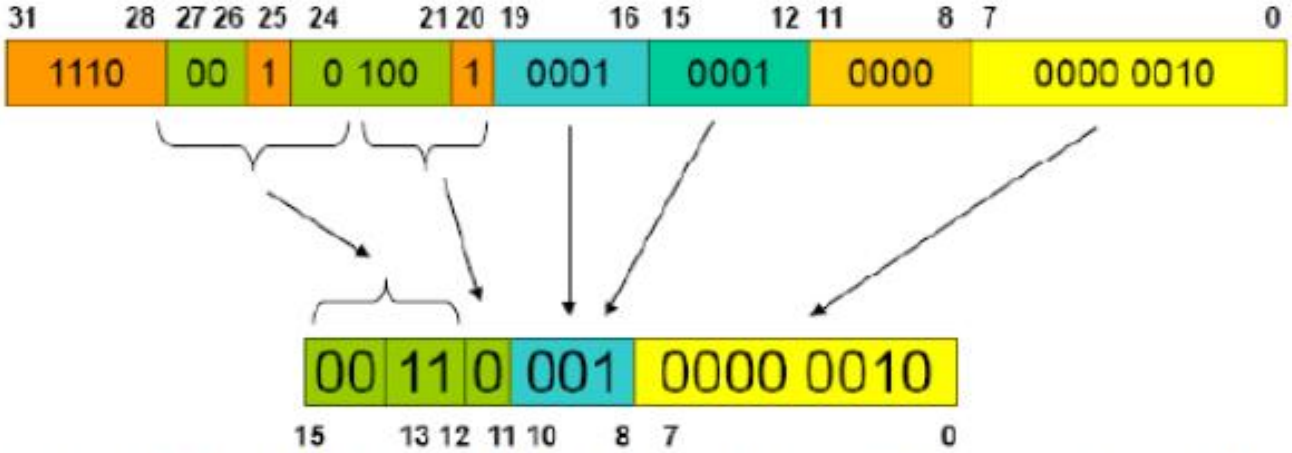
Minor opcode = 0100 (specifically, 100 ⇨ ADD instruction)

Immediate flag = 1 (immediate field in operand 2)

Set status flag = 1 (set carry flag after operation)

ARM and 16-bit Instruction Encoding

ARM 32-bit encoding: `ADDS r1, r1, #2`



- Equivalent 16-bit Thumb instruction: `ADD r1, #2`
  - No condition flag
  - No rotate field for the immediate number
  - Use 3-bit encoding for the register
  - Shorter opcode with implicit flag settings (e.g. the set status flag is always set)

Assembler Language: Unified Assembler Language

统一的汇编语言(UAL) 允许16-bit 和32-bit 指令的选择。

ADD R0, R1 ; R0 = R0 + R1, using Traditional Thumb syntax

ADD R0, R0, R1 ; Equivalent instruction using UAL syntax

当统一的汇编语言的语法被使用时, 指令是否改变标志取决于S后缀。

AND R0, R1 ; Traditional Thumb syntax

ANDS R0, R0, R1 ; Equivalent UAL syntax (S suffix is added)

使用UAL, 你可以通过加后缀来指定你想要的指令:

ADDS R0, #1 ; Use 16-bit Thumb instruction by default

; for smaller size

ADDS.N R0, #1 ; Use 16-bit Thumb instruction (N=Narrow)

ADDS.W R0, #1 ; Use 32-bit Thumb-2 instruction (W=wide)

32-bit Thumb-2 指令可以半字对齐。

0x1000 : LDR r0,[r1] ; a 16-bit instructions (occupy 0x1000-0x1001)

0x1002 : RBIT.W r0 ; a 32-bit Thumb-2 instruction (occupy 0x1002-0x1005)

Assembly language instructions

Assembly language instructions have four fields separated by spaces or tabs. The **label field**、The **opcode field** 、
The **Operands** 、**comment field.**

Label Opcode Operands Comment

Func MOV R0, #100 ; this sets R0 to 100

For example, the general description of the addition instruction

ADD{cond} {Rd,} Rn, #imm12

could refer to either of the following examples.

ADD R0,#1 ; R0=R0+1

ADD R0,R1,#10 ; R0=R1+10

The **assembler** translates assembly source code into **object code**, which are the machine instructions executed by the processor. All object code is halfword-aligned. This means instructions can be 16 or 32 bits wide, and the program counter bit 0 will always be 0. The **listing** is a text file containing a mixture of the object code generated by the assembler together with our original source code.

```
Address     Object code  Label  Opcode  Operand     comment
0x000005E2 F04F0164     Func  MOV     R1,#0x64    ; R1=100
0x000005E6 FB00F001           MUL     R0,R0,R1    ; R0=100*input
0x000005EA F100000A           ADD     R0,R0,#0x0A ;
                                      R0=100*input+10
0x000005EE 4770               BX    LR          ; return 100*input+10
```

ARM - Data Movement

- LEA R0, Label ;R0 <- PC + Offset to Label
- ADR R0,Label or LDR R0,=Label
- LD R1,Label ; R1 <- M[PC + Offset]
- LDR R0,=Label ; Two steps: (i) Get address into R0
  LDRH R1,[R0] ; (ii) Get content of address [R0] into R1
- LDR R1,R0,n ; R1 <- M[R0+n]
- LDRH R1,[R0,#n]
- LDI R1,Label ; R1 <- M[M[PC + Offset]]
- ; Three steps!!
- ST R1,Label ; R1 -> M[PC + Offset]
- LDR R0,=Label ; Two steps: (i)Get address into R0
  STRH R1,[R0] ; (ii) Put R1 contents into address in R0
- STR R1,R0,n ; R1 -> M[R0+n]
- STRH R1,[R0,#n]
- STI R1,Label ; R1 -> M[M[PC + Offset]]
- ; Three steps!!

ARM – Arithmetic/Logic

- ADD R1, R2, R3 ; R1 <- R2 + R3
- ADD R1,R2,R3 ; 32-bit only
- ADD R1,R2,#5 ; R1 <- R2 + 5
- ADD R1,R2,#5 ; 32-bit only, Immediate is 12-bit
- AND R1,R2,R3 ; R1 <- R2 & R3
- AND R1, R2, R3 ; 32-bit only
- AND R1,R2,#1 ; R1 <- Bit 0 of R2
  AND R1, R2, #1 ; 32-bit only
- NOT R1,R2 ; R1 -> ~(R2)
- EOR R1,R2,#-1 ; -1 is 0xFFFFFFFF,
- ; so bit XOR with 1 gives complement

ARM – Control

- BR Target ; PC <- Address of Target
- B Target
- BRnzp Target ; PC <- Address of Target
- B Target
- BRn Target ; PC <- Address of Target if N=1
- BMI Target ; Branch on Minus
- BRz Target ; PC <- Address of Target if Z=1
- BEQ Target
- BRp Target ; PC <- Address of Target if P=1
- No Equivalent
- BRnp Target ; PC <- Address of Target if Z=0
- BNE Target
- BRzp Target ; PC <- Address of Target if N=0
- BPL Target ; Branch on positive or zero (Plus)
- BRnz Target ; PC <- Address of Target if P=0
- No Equivalent

ARM – Subs,TRAP,Interrupt

- JSR Sub ; PC <- Address of Sub, Return address in R7
- BL Sub (Link Reg) ; PC<-Address of Sub, Ret. Addr in R14
- JSRR R4 ; PC <- R4, Return address in R7
- BLX R4 ; PC <-R4, Return address in R14 (Link Reg)
- RET ; PC <- R7 (Implicit JMP to address in R7)
- BX LR ; PC <- R14 (Link Reg)
- JMP R2 ; PC <- R2
- BX R2 ; PC <- R14 (Link Reg)
- TRAP x25 ; PC <- M[x0025], Return address in R7
- SVC #0x25 ; Similar in concept but not implementation
- RTI ; Pop PC and PSR from Supervisor Stack…
- BX LR ; PC <- R14 (Link Reg) [same as RET]

Install Keil

How to install Keil uVision for the Arm, MDK-Lite (32KB) Edition. It does not require a serial number or license key. If you are developing code for a Texas Instruments microcontroller, I suggest you do NOT upgrade to Keil version 5. Rather I suggest you stay with version 4.73:

1) Go to https://www.keil.com/demo/eval/armv4.htm
   Enter your contact information with valid address, phone and email (use your official UT email address),
   Company = University of Texas at Austin,
   devices=TI,
   architecture=Cortex-M4
   Click submit
2) Right-click on MDK473_A.EXE (571,320K) and save it to your computer
3) Install

Execute the mdk473.exe file, installing the application some place easy to find, like C:\Keil or D:\Keil. This is the first screen:

4)Read license agreement, agree to terms, and click Next.

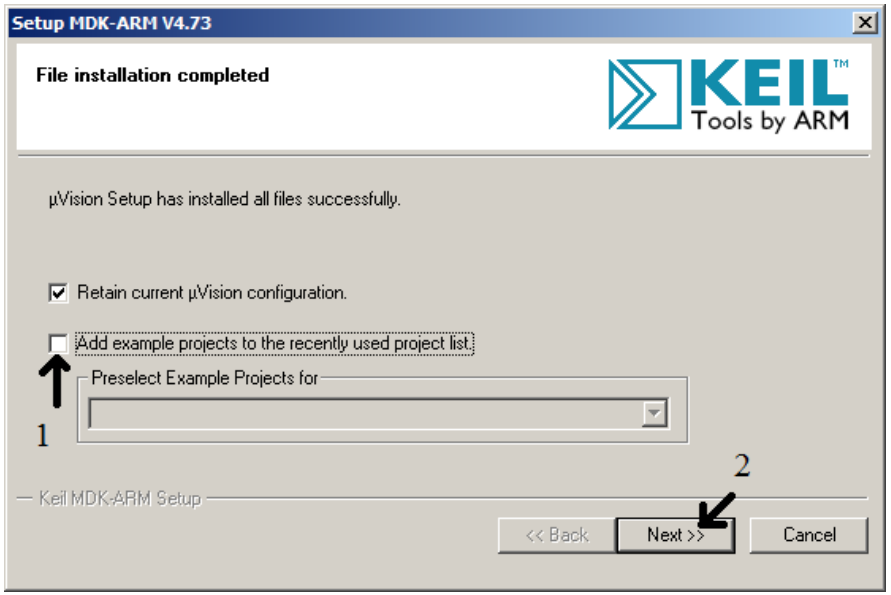
Select a place to install Keil (I chose D: because there was more room on the computer on drive D), and click Next.

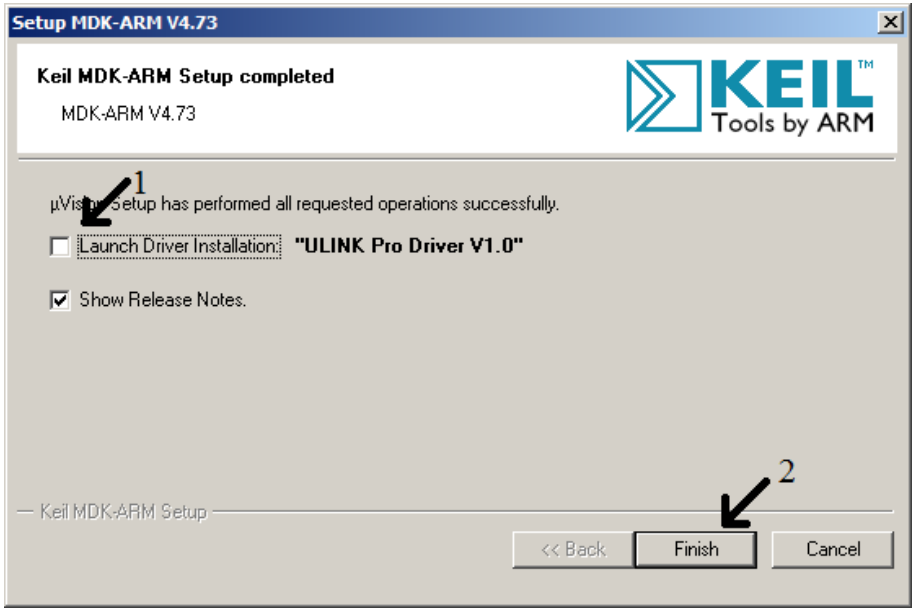
Update these fields with your correct information, and click Next. Set your Company to University of Texas at Austin.

Wait while it installs

Deselect example projects (the examples will come from the class web site), and click Next.

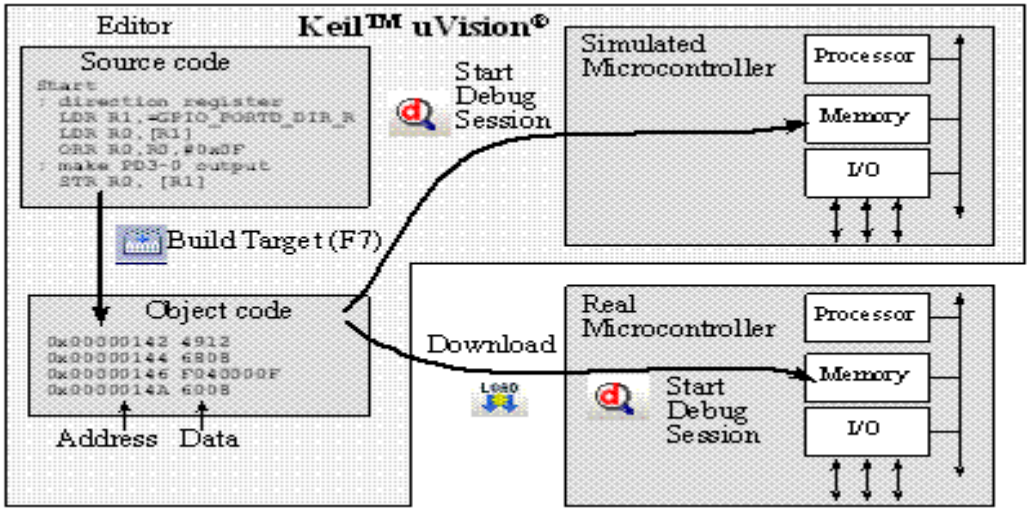
Deselect ULINK Pro Driver V1.0 (the drivers you need will be installed later), and click Finish.





Software Development Process

In this class we will begin with assembly language, and then introduce C. Either the ARM Keil™ **uVision®** or the Texas Instruments **Code Composer Studio™ (CCStudio)** integrated development environment (IDE) can be used to develop software for the Texas Instruments microcontrollers. Both include an editor, assembler, compiler, and simulator. Furthermore, both can be used to download and debug software on a real microcontroller. The entire development process is contained in one application, as shown. In this course, we will use ARM Keil™ **uVision.**



Software Development Process

first use an **editor** to create our **source code**. Next, we use an **assembler** or **compiler** to translate our source code into **object code**. On ARM Keil™ uVision® we compile/assemble by executing the command **Project->Build Target** (short cut F7). The **target** specifies the platform on which we will be running the object code. When testing software with the simulator, we choose the **Simulator** as the target. When simulating, there is no need to download, we simply launch the simulator by executing the **Debug->Start Debug Session** command. The simulator is an easy and inexpensive way to get started on a project. However, its usefulness will diminish as the I/O becomes more complex.

In a real system, we choose the real microcontroller via its JTAG debugger as the target. The JTAG is both a **loader** and a **debugger.** After downloading we can start the system by hitting the reset button on the board or we can debug it by executing **Debug->Start Debug Session** command in the uVision® IDE.

For embedded systems, we typically perform initial testing on a simulator. The process for developing applications on real hardware is identical except the target is switched from a simulated microcontroller to the real microcontroller.