




Fundamentals of Embedded & Real-Time Systems

EMBSYS CP100 – Module 4
Lawrence Lo / Dave Allegre
October 28, 2019

Today's Topics

- 
- > Embedded Programming Overview
 - > Introduction to ARM
 - > ARM Programmer's Model
 - > ARM Assembly Language
 - > A Design Challenge

Embedded Programming Overview



> Similar to Desktop or Server Development

> Differences:

- Cross-development

 - > Develop on a different platform than the target

 - > Executable uploaded to target

 - > Often (not always): no OS or memory manager (so no dynamic/run-time memory allocation)

- Platform Knowledge

 - > Need more knowledge of the platform hardware

 - > Need to know run-time architecture

- Linking and Loading

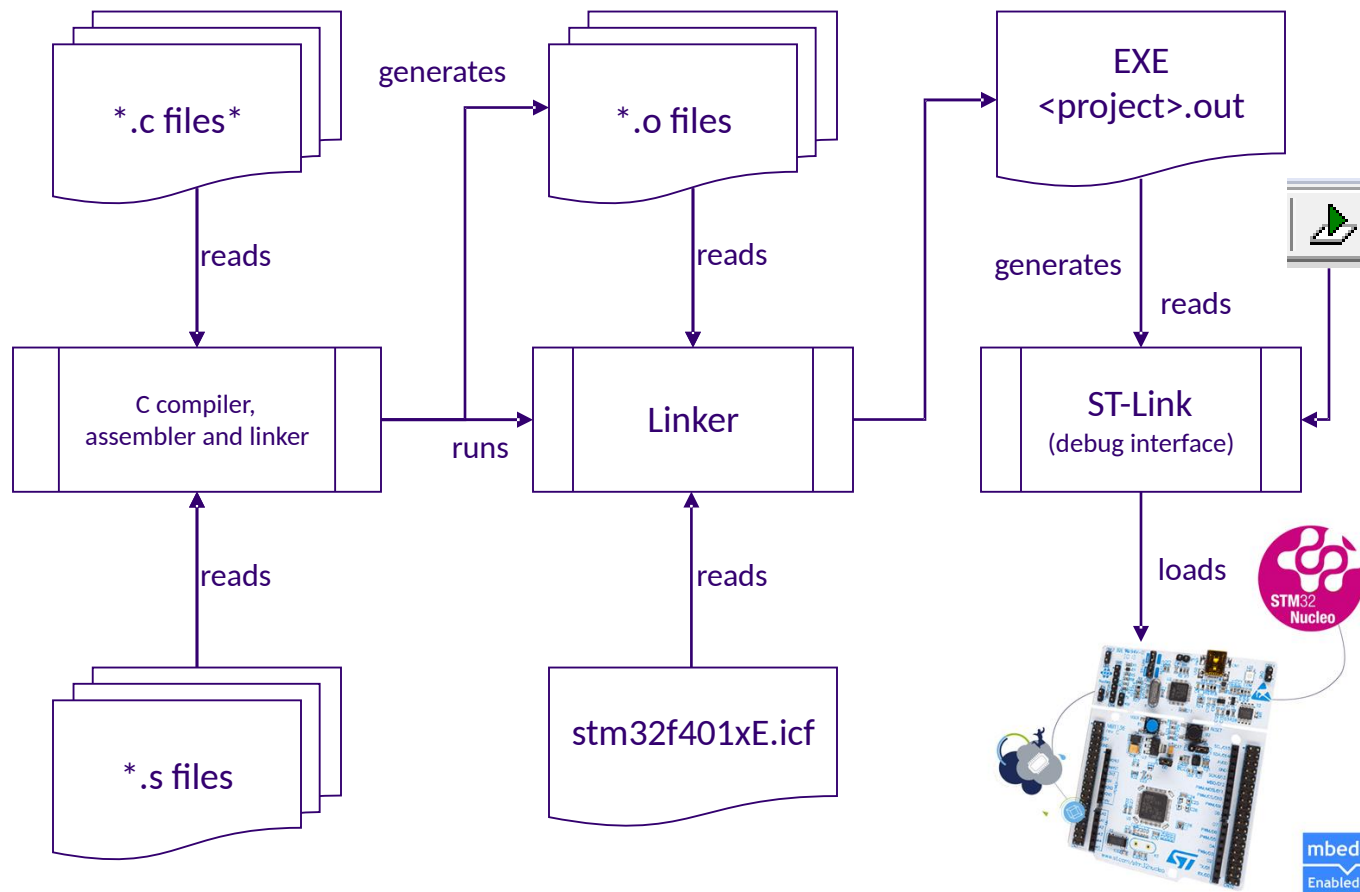
 - > Linking is static, at build time (no DLLs)

 - > Loading occurs at flash write or startup

- Debugging

 - > Need debug hardware support

Load-Build Process



Load-Build Process Tools (1)



> C Compiler

- Compiles C program into object files

> Assembler

- Assembles code into object files

> Linker

- Used to join multiple object files together and define memory configuration

> Flash Programmer

- Used to program the compiled image to the flash memory

Load-Build Process Tools (2)



> Debugger

- Control the operation of the microcontroller and to access internal operation information. The system can be examined and operations can be checked.

> Simulator

- Allow the program to execution to be simulated without real hardware.

> Other utilities

- File converters
- Memory maps

Introduction to ARM

> References

- [1] UM1724 User manual. STM32 Nucleo-64 board. DocID025833 Rev 11. November 2016. STMicroelectronics.
- [2] STM32F401xB STM32F401xC Reference Manual. RM0368 Rev 5. December 2018. STMicroelectronics.
- [3] Cortex™-M4 Devices Generic User Guide. ARM DUI 0553A (ID121610). 2010. ARM.
- [4] ARM Procedure Call Standard. ARM IHI 0042F. Nov 2015. ARM.
- [5] STM32F401xB STM32F401xC Datasheet. DocID024738 Rev 6. September 2016. STMicroelectronics.
- [6] ARMv7-M Architecture Reference Manual. ARM DDI 0403E.b (ID120114). 2014. ARM.

RISC-Based Design (1)

> Two primary CPU designs of instruction set architecture (ISA)

- CISC: Complex instruction set computer
- RISC: Reduced instruction set computer

> CISC Features

- Many instructions and variants
- Instructions can do complex tasks, so compiler has to do less work
- Operators can work directly on memory
- Instructions can take a varying number of CPU cycles to complete
- Instructions can vary in size

RISC-Based Design (2)

> RISC Features


- Few instructions
- Instructions do simple tasks quickly, but compiler has to do more work
- Load-Store operation: Operators can only work on values held in registers
- Instructions take the same number of cycles to execute (one)
- Instructions are all the same size

> ARM is a RISC-based (but not pure RISC) design

> Primary RISC Features in ARM

- Load-Store architecture (aka Read-Modify-Write)
- Instructions are either 32-bit or 16-bit

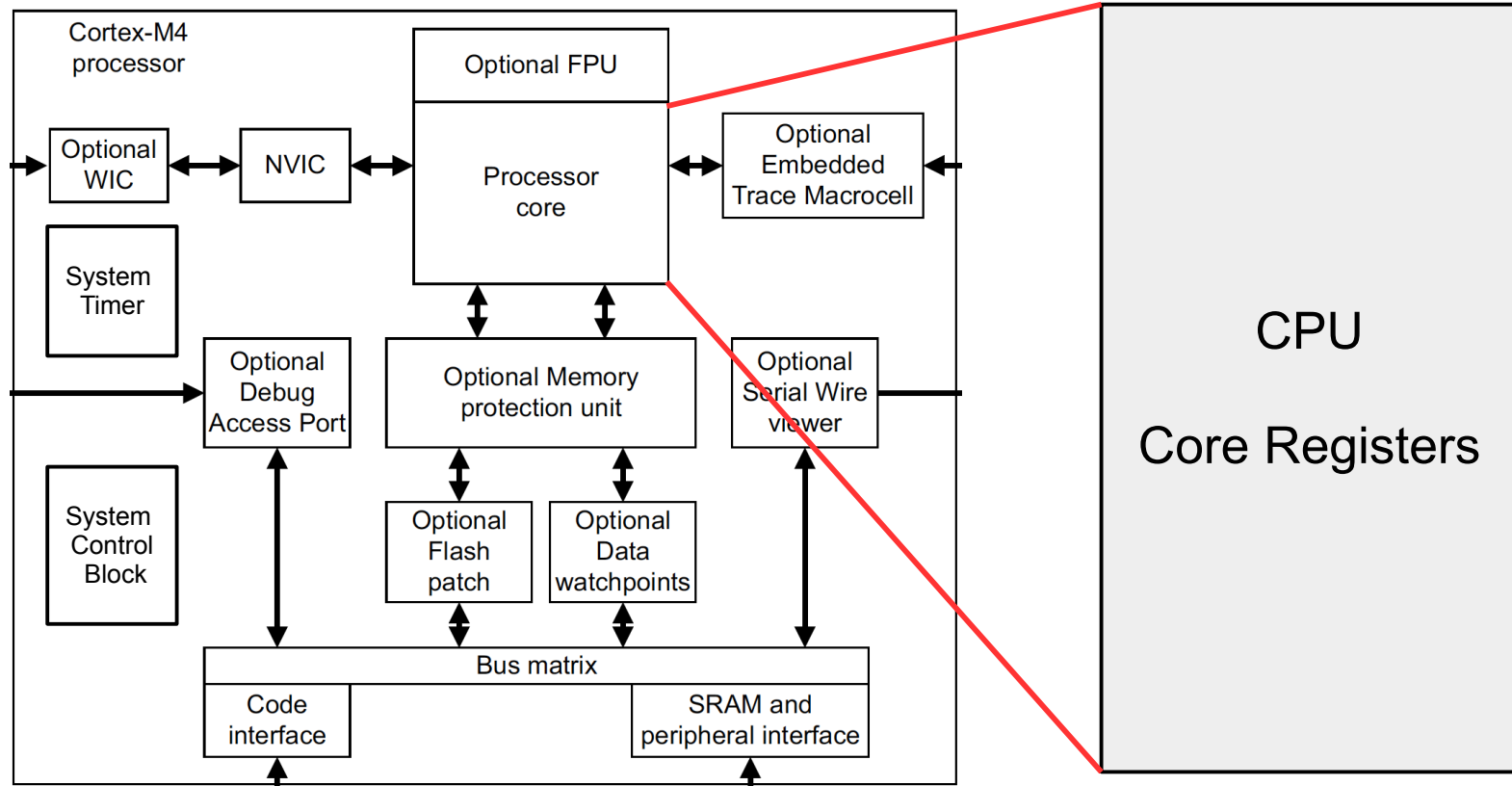
ARM Architecture

- 
- > The STM32F4 implements ARMv7E-M architecture
 - > The Cortex-M4 Core:
 - 32-bit registers
 - 32-bit internal data path and bus interface
 - 32-bit hardware multiply with 32 or 64-bit result.
 - 1 – 240 interrupts
 - Thumb and Thumb-2 instruction sets
 - Optional Floating-Point Unit (FPU)
 - Optional Memory Protection Unit (MPU)

ARM Programmer's Model (1)

- > A programmer's model is an abstraction of a computer architecture from the view of a programmer. It includes:
 - Register set
 - General purpose and control registers.
 - Memory map
 - Flash, SRAM and peripherals.
 - Exception handling
 - Entry to and exit from exceptions/interrupts. Exception vectors.
 - Instruction set
 - Mnemonics, operands and address modes.

ARM Programmer's Model (3)



ARM Programmer's Model (4)

Operation States

> Debug State

- When the processor is halted, it enters debug and stops executing instructions

> Thumb State

- When the process is running a program code, it is in Thumb state.

ARM Programmer's Model (5)

Operation Modes

- > ARM operates in one mode at a time
- > Handler Mode
 - When executing an exception handler such as an Interrupt Service Routine (ISR).
 - In handler mode, the processor always has privileged access level.
- > Thread Mode
 - When executing application code.
 - In thread mode, the processor can be with either privileged or unprivileged access level.
- > After reset, the processor is in thread mode with privileged access level.

ARM Programmer's Model (6)



> Privileged Access Level

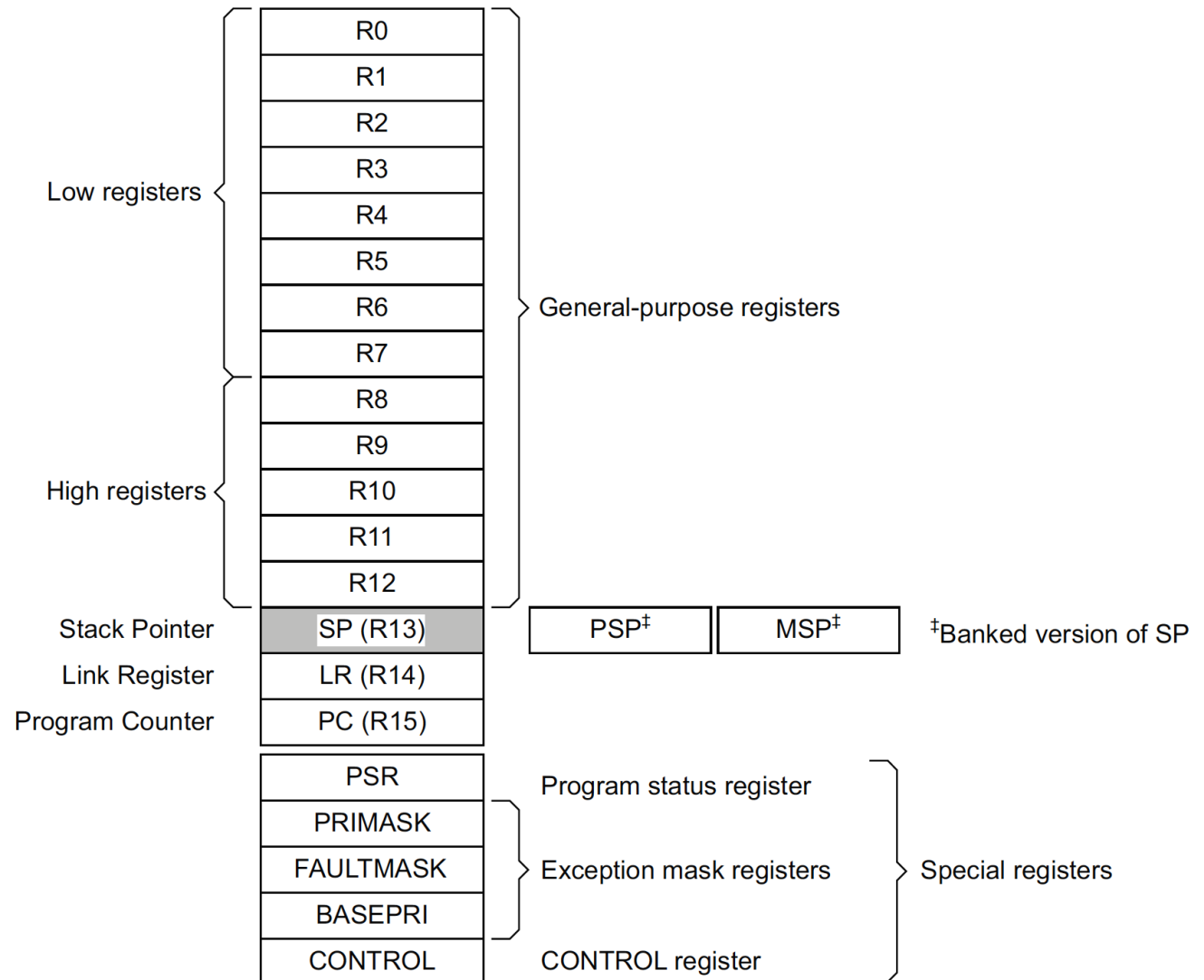
- Can access System Control Block, NVIC or System Timer.
- Can use special register access instructions (MRS and MSR).
- Software in this level can switch into unprivileged access level via the CONTROL register.
- On exception, processor always (and automatically) switches to privileged access level.

> Unprivileged Access Level

- Access to System Control Block, etc. is blocked.
- Cannot access special register instructions (MRS and MSR).
- Cannot switch to privileged access level.

ARM Programmer's Model (7)

Registers (Ref [3])



Registers (2)

- R0 – R3, and R12 are scratch registers that can be used to hold intermediate values between subroutine calls.
R0 – R3 are also used to pass parameters and return value to and from a subroutine.
That is, they may be corrupted by a subroutine. It is the caller's responsibility to save their values in stack if they need to be preserved across a subroutine call.
- R4 – R11 are used for variables. A subroutine must preserve their values if it needs to use them (e.g. saving them in stack).
- We need to follow these rules when interfacing assembly code with C/C++ code.
- *Which registers need to be saved upon entry to ISR?*
(See p. 2-27 of [Ref 3])

Registers (3)

> R13, Stack Pointer (SP)

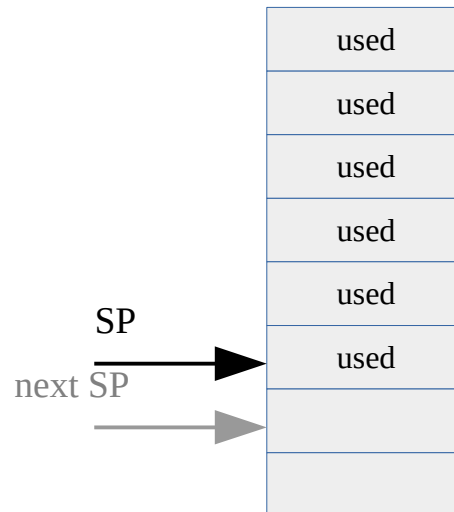
- Full descending stack.

- > SP points to the last stacked item in memory (full).

- > When pushing an item to stack, it decrements SP and then writes the item to the memory pointed to by SP.

- > What is the initial value of SP?

It is specified as the first exception vector.



Registers (4)

- Physically two different R13 registers
 - > Main Stack Pointer (MSP) or SP_main
 - Default Stack Pointer after power up
 - Used in Thread mode (Application) or Handler mode (Exception)
 - > Process Stack Pointer (PSP) or SP_process
 - Used in Thread mode (Application) only.
 - > Only one is visible at a time.
 - > R13 allows access to the current Stack Pointer.
Handler mode always uses MSP. Thread mode uses either MSP or PSP depending on CONTROL register setting.
- Lowest two bits of SP are always zero.
 - > Writes to these locations are ignored.
 - > Addresses of the transfers in stack operations must be aligned to 32-bit boundaries.

Registers (5)

> R14, Link Register (LR)

- Holds the return address when calling a function or subroutine.
- When a call is made, the value of LR is updated automatically
- If a nested function needs to be called, the value of the LR needs to be saved to the stack *first*.
- During an exception handling, the LR is updated automatically to a special exception return value which is used for triggering the exception return at the end of the handler (p. 2-28 of [Ref 3]).
 - > For example, the value 0xFFFFFFF9 tells the processor to return to Thread mode (application) using the non-floating point state (exception frame) saved in MSP (Main stack) and continue to use MSP after return.
 - > Where is the exception return address specified?

Registers (6)

> R15, Program Counter (PC)

- Is readable and writeable
- A read returns the current instruction address, plus 4
 - > Due to the pipeline nature of the design
 - > Compatibility requirement with the ARM7TDMI processor
- Writing to PC causes a branch or return-from-branch operation.

> A note about R14 and R15 registers

- Addresses are always even (bit 0 is zero)
- Some branch/call operations (BX and BLX) require bit 0 to be set to indicate Thumb state or else a fault exception will be triggered (p. 3-119 of [Ref 3]).
It doesn't mean an instruction is stored at an odd address.

Registers (7)

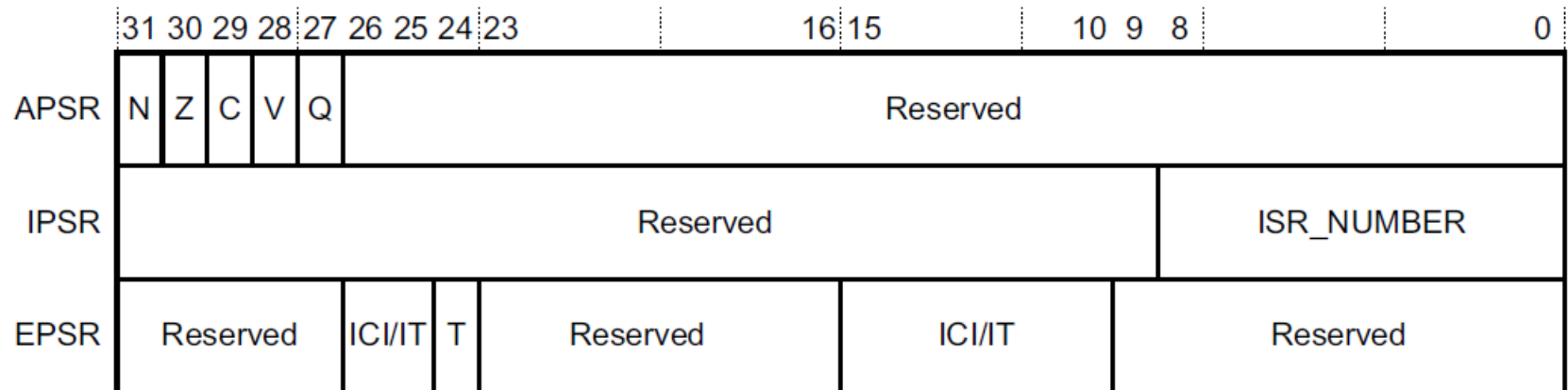
> Aliases of register names used in assembly code

R0 – R12:	R0, R1...R12, r0, r1...r12
R13:	R13, r13, SP, sp, MSP, PSP
R14:	R14, r14, LR, lr
R15:	R15, r15, PC, pc

Special Registers (1)

> Program Status Registers

- Composed of three status registers
 - > Application PSR (APSR)
 - > Interrupt PSR (IPSR)
 - > Execution PSR (EPSR)



Ref [3]

Special Registers (2)

- > All PSR's can be accessed together, or separately
- > Combined PSR is named as xPSR and while accessing, use name PSR
- > Requires special register access instructions (MRS and MSR)
 - MRS means “move from special register to general register”
 - MSR means “move from general register to special register”

*From ARM website

Registers (1)

> Registers R0 – R12

- General purpose registers
- Stores data or addresses
- Initial values are undefined
- Two subsets: Low and High
 - > Depending on 16-bit Thumb or 32-bit Thumb-2 Instructions
- Their use in C/C++ is governed by APCS (ARM Procedure Call Standard), also known as ABI (Application Binary Interface) (Ref [4]).

Special Registers (3)

> Bit fields in the PSR

- Contains the condition flags resulting from the execution of previous instructions (if enabled).
 - > [31] N – Negative flag
 - > [30] Z – Zero flag
 - > [29] C – Carry or borrow flag
 - > [28] V – Overflow flag
 - > [27] Q – DSP overflow and saturation flag
 - > [26:25][15:10] ICI/IT – *Interrupt-Continuable* Instruction bits (LDM, STM), or *If-Then* instruction status bit for conditional execution (p.2-7 of [Ref 3] for details)
 - > [24] T – Thumb state. Always 1; clearing will cause a fault exception
 - > [8:0] ISR Number – Number of the current exception.
 - Also known as “Exception number” (p. 2-24 of [Ref 3]).
 - 0 – Thread mode, 2 – NMI, 3 HardFault ... 16 – IRQ0
 - “IRQ number” is offset by -16 and is used by CMSIS.

Special Registers (4)

Exception number	IRQ number	Offset	Vector
16+n	n	0x0040+4n	IRQn
.	.	.	.
.	.	.	.
.	.	.	.
18	2	0x004C	IRQ2
17	1	0x0048	IRQ1
16	0	0x0044	IRQ0
15	-1	0x0040	Systick
14	-2	0x003C	PendSV
13		0x0038	Reserved
12			Reserved for Debug
11	-5	0x002C	SVCall
10			Reserved
9			
8			
7			
6	-10	0x0018	Usage fault
5	-11	0x0014	Bus fault
4	-12	0x0010	Memory management fault
3	-13	0x000C	Hard fault
2	-14	0x0008	NMI
1		0x0004	Reset
		0x0000	Initial SP value

Ref [3]

Special Registers (3)

- > Each exception, including interrupts has a priority level.
 - Smaller the number, higher the priority.
 - Configurable (≥ 0) except Reset (-3), NMI (-2) and Hardfault (-1)
- > Interrupt Mask Registers
 - PRIMASK
 - > 1-bit wide
 - > Effectively it raises the current exception priority level to 0.
 - Highest level for a programmable exception/interrupt
 - > Prevents the activation of all exceptions with configurable priority.

Special Registers (4)

– FAULTMASK

- > Similar to PRIMASK, but also blocks HardFault exception (-1).
 - Can be used by fault handling code to suppress triggering of further faults during fault handling.

– BASEPRI

- > Allows more flexible interrupt masking.
- > When set to 0, is disabled.
- > When set to a non-zero, it blocks exceptions and interrupts that have the same or lower priority level as that in BASEPRI.
- > Still allows exceptions with a higher priority to be accepted by the processor.

Special Registers (5)

> CONTROL[2:0] register

- Selection of the stack pointer (Main Stack Pointer/Process Stack Pointer)
- Access level in Thread Mode (Privileged/Unprivileged)

> [0] nPriv – Defines privilege level

- 0 (default) It is privileged level when in Thread mode
- 1 is unprivileged level when in Thread mode
- In Handler mode, processor is always in privileged mode

> [1] SPSEL – Defines Stack Pointer

- 0 (default) Thread mode uses Main Stack Pointer (MSP)
- 1 Thread mode uses Process Stack Pointer (PSP)
- In Handler mode, this bit is always 0 and write to this bit is ignored.

> [2] FPCA – Floating Point Context Active

- Determines if registers in floating point unit need to be saved
- 0 (default) Floating point unit has not been used in the current context (no save)
- 1 FP context active; Cortex-M4 needs to preserve FP state when processing an exception.

ARM Assembly Language (1)



Cortex Microcontroller Software Interface Standard (CMSIS)

- > Developed by ARM to allow microcontroller and software vendors to use a consistent software infrastructure.
- > Provides a form of standardization
- > Enhanced software reusability and compatibility
- > Toolchain independent

ARM Assembly Language (2)

Instruction	CMSIS function
CPSIE I	void __enable_irq(void)
CPSID I	void __disable_irq(void)
CPSIE F	void __enable_fault_irq(void)
CPSID F	void __disable_fault_irq(void)

Special register	Access	CMSIS function
PRIMASK	Read	uint32_t __get_PRIMASK (void)
	Write	void __set_PRIMASK (uint32_t value)
FAULTMASK	Read	uint32_t __get_FAULTMASK (void)
	Write	void __set_FAULTMASK (uint32_t value)
BASEPRI	Read	uint32_t __get_BASEPRI (void)
	Write	void __set_BASEPRI (uint32_t value)
CONTROL	Read	uint32_t __get_CONTROL (void)
	Write	void __set_CONTROL (uint32_t value)

Ref [3]

ARM Assembly Language (3)

- > Short of writing code in binary, this is the closest a programmer gets to the machine's language
- > Most compilers compile C to assembly, then assemble the resulting code
- > Requires precise thinking and planning, and knowledge of the processor architecture
- > ARM Assembly Language has some very interesting and powerful features
 - Ability to control which operations set status flags
 - > except for instructions whose purpose is setting the flags
 - Ability to conditionally execute any instruction
 - > many other processors only offer conditional branching
 - Flexible manipulation of second operand

ARM Assembly Language (4)

- > Some operations cannot be easily performed in high-level languages:
 - e.g., Rotate through carry
- > Registers are not addressable and therefore not accessible to high-level languages
- > Interrupt support *often* requires assembly language (Cortex-M makes it possible to just use C)
- > Operating Systems
 - Task switching often requires direct register access
 - Critical section entry/exit functions need access to interrupt control flags
- > Debugging
 - Debuggers can allow you to step through assembly
 - Need to be able to read the code generated by the compiler

Data Processing Instructions

- > Data processing instructions typically have the following syntax:
 $\text{op}\{\text{S}\}\{\text{cond}\} \{\text{Rd},\} \text{Rn}, \text{Operand2}$
 $\text{op}\{\text{cond}\} \{\text{Rd},\} \text{Rn}, \#imm12$
- > **op** specifies the operation to perform (ADD, CMP, etc)
- > **{S}** specifies that the instruction should update the condition flags in APSR (N,Z,V,C,Q)
- > **{cond}** specifies the optional condition to test
 - Only executes if the condition is true
 - For example LE means “less than or equal to”, BLE means “branch if less than or equal to”
- > **Rd** specifies the destination register to hold the result of an operation. If omitted, Rn is used as destination register as well.
- > **Rn** specifies the first source operand register
- > **Operand2** specifies the second source operand. It is very flexible and it therefore called a “**flexible second operand**”.
- > **#imm12** is a constant (immediate value) from 0 – 4095 (ADD, SUB)

Flexible Second Operand

- > Used for
 - Constant
 - Register with optional shift
 - > It can have one of the following formats:
 - **#<constant>**
 - Any constant that can be produced by shifting an 8-bit value left by any number of bits within a 32-bit word
 - Any constant of the form 0x00XY00XY
 - Any constant of the form 0xXY00XY00
 - Any constant of the form 0xXYXYXYXY
 - Some instructions take a wider range of constant values.
 - **Rm {, Shift}**
 - Rm is the register holding the data for the second operand
 - Shift is an optional shift to be applied to Rm. See Ref [3] for details.
 - ASR #n Arithmetic shift right n bits, $1 \leq n \leq 32$
 - LSL #n Logical shift left n bits, $1 \leq n \leq 31$
 - LSR #n Logical shift right n bits, $1 \leq n \leq 32$
 - ROR #n Rotate right n bits, $1 \leq n \leq 31$
 - RRX Rotate right one bit, with extend
- ```
- CMP r7, #1000 ; value=0xFA, shift=2 (1000 = 0x3E8)
- BIC r9, r8, #0xFF00 ; value=0xFF, shift=8
- ADD r0, r1, r2, LSL #2 ; r0 = r1 + r2 * 4
```

# Memory Access Instructions (1)

- > Define how an address is computed
- > Unlike data processing instructions, which must operate on registers or immediate values, load and store works with memory
- > Fall into one of two categories:
  - **Immediate offset**
  - **Register offset**
- > **Immediate offset:**

|                                     |                           |
|-------------------------------------|---------------------------|
| op{type}{cond} Rt, [Rn {, #offset}] | ; offset addressing       |
| op{type}{cond} Rt, [Rn, #offset]!   | ; pre-indexed addressing  |
| op{type}{cond} Rt, [Rn], #offset    | ; post-indexed addressing |
- > **type** specifies the size of transfer (byte, halfword).  
If omitted, default is word.
- > **Rt** specifies the register to load or store
- > **Rn** specifies the register storing the base address for memory access
- > **offset** specifies the immediate offset (+/-) from the base address

# Memory Access Instructions (2)

## > Offset addressing

The immediate offset value is added to the address stored in Rn to yield the address for memory access. The content of Rn is not changed.

## > Pre-indexed addressing

Similar to offset addressing, with the exception that the resulting memory address is written back to Rn.

## > Post-indexed addressing

The address stored in Rn is used as the address for memory access. Afterwards the immediate offset value is added to the address and written back to Rn.

## > For example:

```
LDRNE R2, [R5, #100]! ; address = R5 + 100. Update R5.
```

# Memory Access Instructions (3)

## > **Register offset:**

op{type}{cond} Rt, [Rn, Rm {, LSL #n}]

## > **type** specifies the size of transfer (byte, halfword)

B – unsigned byte                  SB – signed byte

H – unsigned halfword      SH – signed halfword

If omitted, default is word.

## > **Rt** specifies the register to load or store

## > **Rn** specifies the register storing the base address for memory access.

## > **Rm** specifies the register storing an offset (+/-) value from the base address

## > **LSL #n** specifies an optional shift (0 to 3) of the content in Rm

## > For example:

```
STR R0, [R1, R2, LSL #2] ; address = R1 + R2 * 4
```



# Conditional Codes

- > Specify which values of the status flags enable execution of the instruction
- > Conditional instructions, except for conditional branches, must be inside an If-Then block
- > Below is a list of some possible values of {cond}:

| {cond} | Meaning                      | Status Flag Values                                |
|--------|------------------------------|---------------------------------------------------|
| EQ     | Equal                        | Z set                                             |
| NE     | Not Equal                    | Z clear                                           |
| CS     | Unsigned higher or same      | C == 1                                            |
| CC     | Unsigned lower               | C == 0                                            |
| GE     | Signed greater than or equal | N set and V set, or<br>N clear and V clear (N==V) |
| LT     | Signed less than             | N set and V clear, or<br>N clear and V set (N!=V) |
| GT     | Signed greater than          | Z == 0 and N == V                                 |
| LE     | Signed less than or equal    | Z == 1 or N != V                                  |

# Assembly Language Examples (1)

- > Let's look at some ARM assembly language instructions.  
(.N suffix forces 16-bit instruction encoding)

```
uint32_t Fibonacci(uint32_t n) {
 if (n <= 1) {
 return 1;
 }
 uint32_t prev2 = 1;
 uint32_t prev1 = 1;
 uint32_t result;
 while (--n) {
 result = prev2 + prev1;
 prev2 = prev1;
 prev1 = result;
 }
 return result;
}
```

|                         |       |            |  |
|-------------------------|-------|------------|--|
| Fibonacci:              |       |            |  |
| 0x8002004: 0xb410       | PUSH  | {R4}       |  |
| 0x8002006: 0x0001       | MOVS  | R1, R0     |  |
| if (n <= 1) {           |       |            |  |
| 0x8002008: 0x2902       | CMP   | R1, #2     |  |
| 0x800200a: 0xd201       | BCS.N | 0x8002010  |  |
| return 1;               |       |            |  |
| 0x800200c: 0x2001       | MOVS  | R0, #1     |  |
| 0x800200e: 0xe009       | B.N   | 0x8002024  |  |
| uint32_t prev2 = 1;     |       |            |  |
| 0x8002010: 0x2201       | MOVS  | R2, #1     |  |
| uint32_t prev1 = 1;     |       |            |  |
| 0x8002012: 0x2301       | MOVS  | R3, #1     |  |
| while (--n) {           |       |            |  |
| 0x8002014: 0x1e49       | SUBS  | R1, R1, #1 |  |
| 0x8002016: 0x2900       | CMP   | R1, #0     |  |
| 0x8002018: 0xd004       | BEQ.N | 0x8002024  |  |
| result = prev2 + prev1; |       |            |  |
| 0x800201a: 0x189c       | ADDS  | R4, R3, R2 |  |
| 0x800201c: 0x0020       | MOVS  | R0, R4     |  |
| prev2 = prev1;          |       |            |  |
| 0x800201e: 0x001a       | MOVS  | R2, R3     |  |
| prev1 = result;         |       |            |  |
| 0x8002020: 0x0003       | MOVS  | R3, R0     |  |
| 0x8002022: 0xe7f7       | B.N   | 0x8002014  |  |
| return result;          |       |            |  |
| 0x8002024: 0xbc10       | POP   | {R4}       |  |
| 0x8002026: 0x4770       | BX    | LR         |  |

# Assembly Language Examples (2)

```
uint32_t Fibonacci2(uint32_t n) {
 if (n <= 1) {
 return 1;
 }
 uint32_t result = Fibonacci2(n-2) + Fibonacci2(n-1);
 return result;
}
```

|                                                      |                |       |                  |
|------------------------------------------------------|----------------|-------|------------------|
| Fibonacci2:                                          |                |       |                  |
| 0x8002028:                                           | 0xb538         | PUSH  | {R3-R5, LR}      |
| 0x800202a:                                           | 0x0004         | MOVS  | R4, R0           |
| if (n <= 1) {                                        |                |       |                  |
| 0x800202c:                                           | 0x2c02         | CMP   | R4, #2           |
| 0x800202e:                                           | 0xd201         | BCS.N | 0x8002034        |
| return 1;                                            |                |       |                  |
| 0x8002030:                                           | 0x2001         | MOVS  | R0, #1           |
| 0x8002032:                                           | 0xe007         | B.N   | 0x8002044        |
| uint32_t result = Fibonacci2(n-2) + Fibonacci2(n-1); |                |       |                  |
| 0x8002034:                                           | 0x1ea0         | SUBS  | R0, R4, #2       |
| 0x8002036:                                           | 0xf7ff 0xffff7 | BL    | Fibonacci2       |
| 0x800203a:                                           | 0x0005         | MOVS  | R5, R0           |
| 0x800203c:                                           | 0x1e60         | SUBS  | R0, R4, #1       |
| 0x800203e:                                           | 0xf7ff 0xffff3 | BL    | Fibonacci2       |
| 0x8002042:                                           | 0x1940         | ADDS  | R0, R0, R5       |
| return result;                                       |                |       |                  |
| 0x8002044:                                           | 0xbd32         | POP   | {R1, R4, R5, PC} |
| 0x8002046:                                           | 0x0000         | MOVS  | R0, R0           |

# Week 4 Reading Assignments



- > Reading assignment in [Ref 3]
  - Section 2.1 (ARM programmers model)
  - Section 2.3 (Exception model)
  - Section 3.3.3 and 3.3.7 (2<sup>nd</sup> operand, conditional)
  - Section 3.4.2 and 3.4.3 (LDR, STR)