Microprocessors

Fall 2020

3. Assembly Language Introduction

Tentative Weekly Schedule

- Week x1 - Introduction to Course
- Week x2 - Architecture
- **Week x3 - Assembly Language Introduction**
- Week x4 - Assembly Language Usage and Faults
- Week x5 - Embedded C, Toolchain and Debugging
- Week x6 - Interrupts
- Week x7 - Timers
- Week x8 - Modulation
- Week x9 - Serial Communications I
- Week xA - Serial Communications II
- Week xB - Analog Interfacing
- Week xC - Memory and DMA
- Week xD - RTOS
- Week xE - Wireless Communications

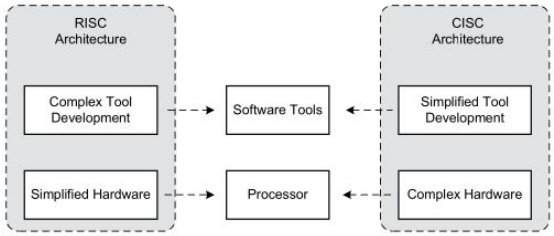
Review: ARM: Overview of Processor Families

There are three branches in ARM Cortex family cores:

- **Cortex-A** - Application processors. Designed as fully functional computers capable of running complex operating systems. Commonly used in mobile phones, tablets, and laptops such as Raspberry PI, Android phones, iPhones, iPads, ...
- **Cortex-R** - Real-time processors. Designed to be fast reacting to events. Have low interrupt latency and more deterministic in tight situations. Often used in critical systems where data interpretation is essential such as medical devices, car systems, basebands and low-level device controllers, such as hard drive controllers.
- **Cortex-M** - Microcontroller series. Designed to be ultra-low-power and small form-factor. Runs at a way slower clock speed then A and R series. They are used in robotic systems and small consumer electronics.

Review: Complexity based ISA Classification

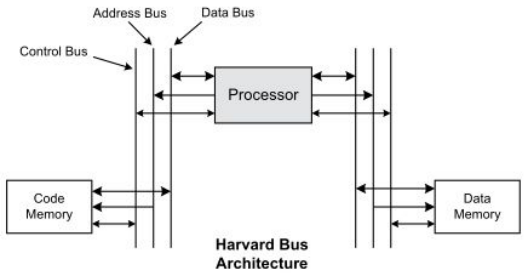
- **Complex instruction set computers (CISC)**
  - Uses single complex instruction that can perform many operations usually requiring multiple cycles
  - Usually supports varying instruction length
  - More complex hardware, simplified compiler
- **Reduced instruction set computers (RISC)**
  - Uses simplified instructions that can use many instructions to perform an operation usually completing one instruction per cycle
  - More simplified hardware, complex compiler to transform code

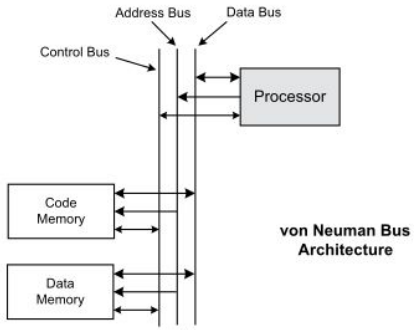


Review: Memory Interface-Based Architecture Classification

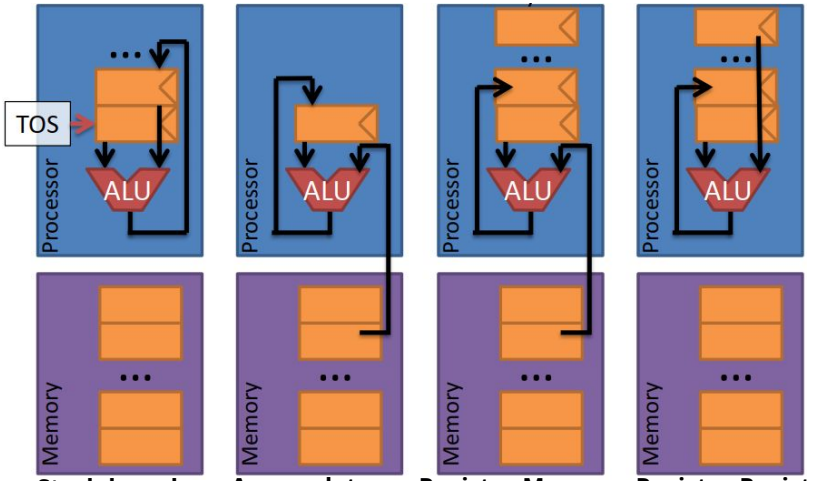
There are two widely used memory interface architectures

- **von Neumann** - uses a common bus for both data and code
- **Harvard** - uses separate buses for data and code





Review: Where do operands come and results go?



Stack-based

PUSH A
PUSH B
ADD
POP C

Accumulator

LOAD A
ADD B
STORE C

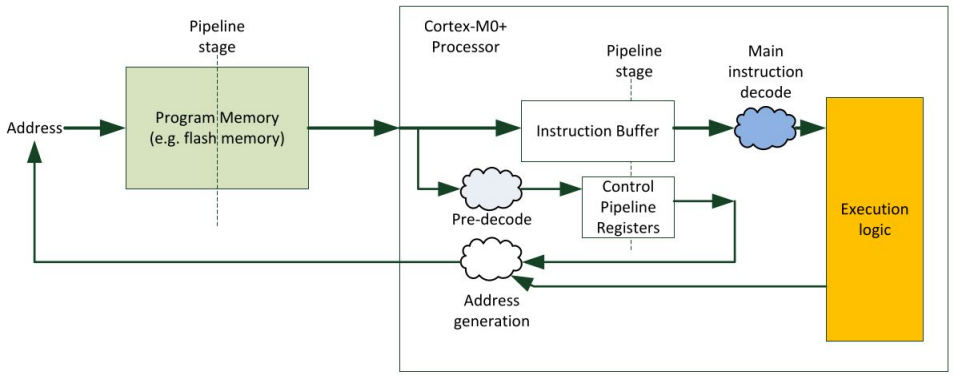
Register-Memory

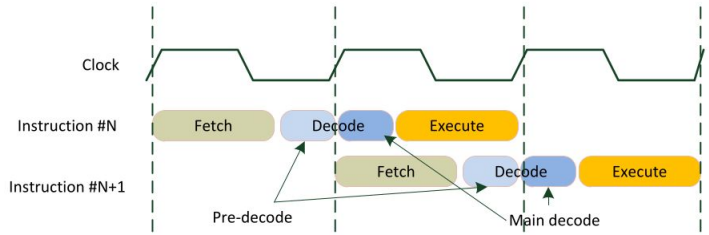
LOAD R1, A
ADD R3, R1, B
STORE R3, C

Register-Register

LOAD R1, A
LOAD R2, B
ADD R3, R1, R2
STORE R3, C

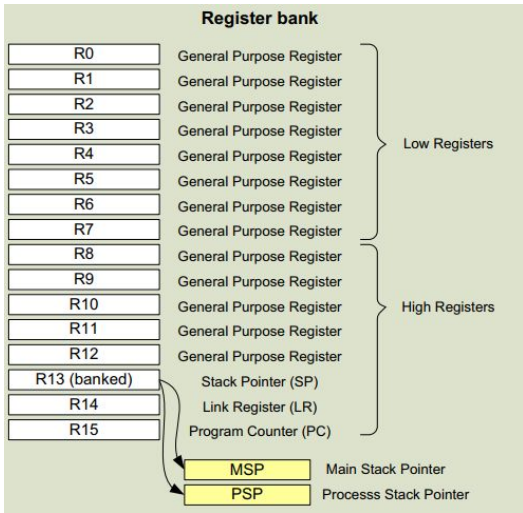
ARM Cortex M0+ Two stage pipeline





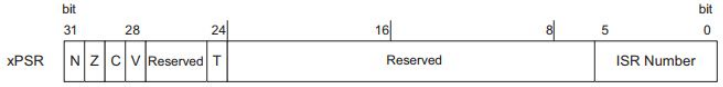
Review: Regular and Special Registers

- In Cortex-M0+, there are **16 32-bit** registers (13 general purpose and 3 specific use)
  - **R0 - R15** with **R0-R12** being general purpose registers.
  - due to size, *mostly* **only low registers** can be used with 16-bit Thumb code.
  - data needs to be loaded from memory to registers to be processed (**load/store**)

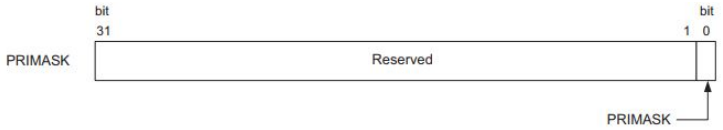


Review: Special Registers

- **xPSR** - Program Status Register - holds information about program execution, exception number and the ALU flags.



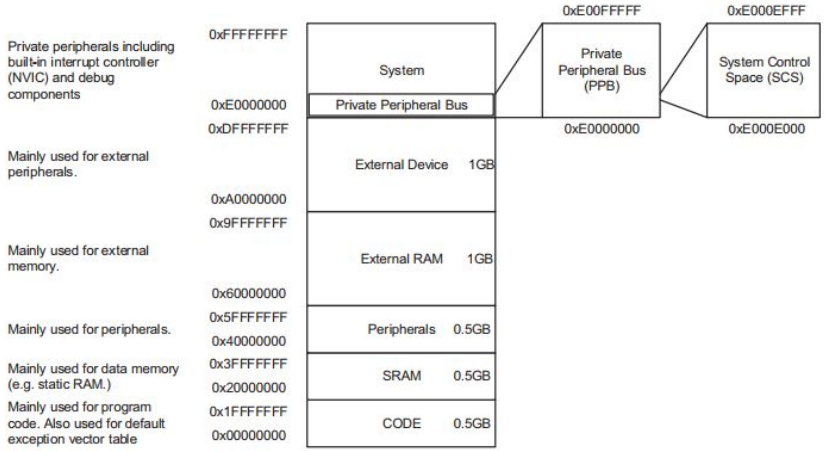
- **PRIMASK** - Interrupt Mask special Register is used to **block all interrupts except** *Non-Maskable Interrupt (NMI)* and the *HardFault* exception



- **CONTROL** - special register that is used to change processor mode and stack pointer

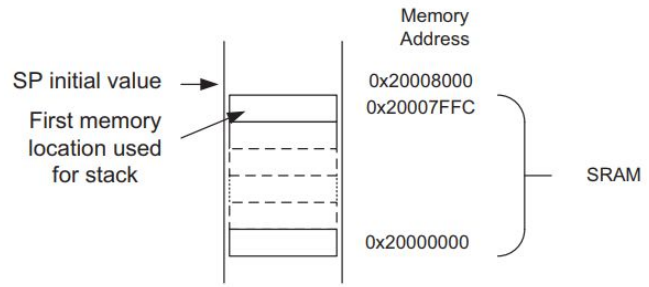
Review: Memory System

- Cortex-M processors have 4 GB of memory address space.
- Memory space is architecturally defined into a number of regions
- As a result all the Cortex-M devices have the same programming model for interrupt control and debug.



Review: Stack Memory Operation

- **Stack memory** is a memory usage mechanism that allows the system memory to be used as **temporary** data storage that behaves as a **first-in-last-out** (FILO) buffer.
- Content can be **added** with **PUSH** instruction and can be **acquired** using **POP** instruction
- Usually located at the top of SRAM, and **grows downwards** - meaning if data is added it gets **filled toward lower addresses** (**full-descending**)
- **Stack Pointer** shows the top of the stack

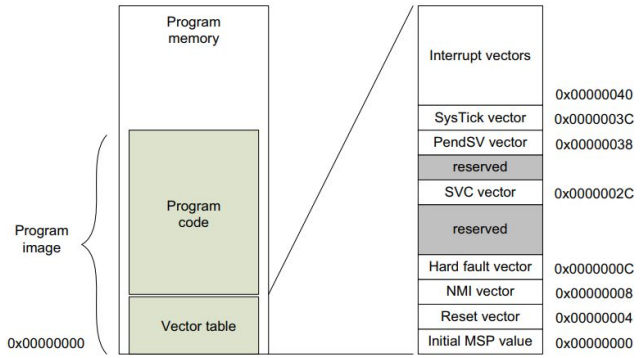


Review: Nested Vectored Interrupt Controller

- Used for **prioritizing** the interrupt requests and **handling** other exceptions
- The interrupt management function is controlled by registers in the **NVIC** located within the **System Control Space (SCS)**
- NVIC supports:
  - Flexible interrupt management
  - Nested interrupt support
  - Vectored exception entry
  - Interrupt masking

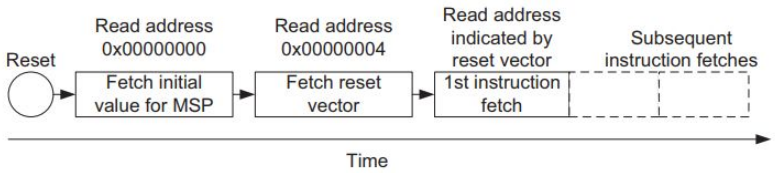
Review: Vector Table

- The beginning of the program image has the **vector table** which contains the starting address of exceptions.
- The size of the vector table depends on the number of interrupts implemented.
- First two addresses are important for reset operation. The next two are important for faulty recovery.

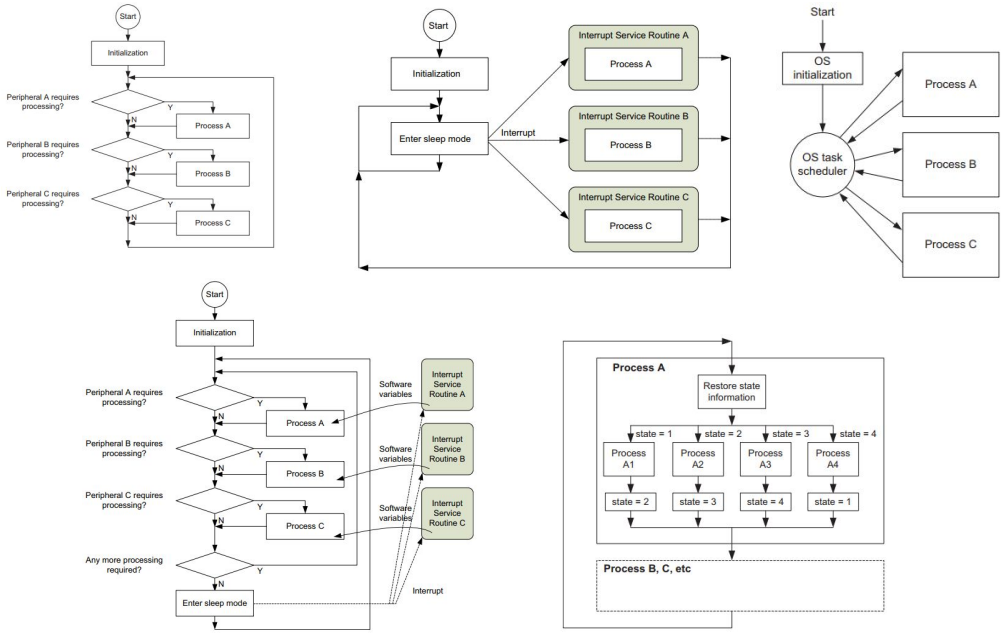


Review: Reset Sequence

- We have already seen what happens after reset in terms of software routines.
- First address in the vector table indicates the stack pointer value. (**SP** is set)
- Second address in the vector table is the Reset Handler. (**PC** is set)
- Then PC makes the jump to the pointing address.



Review: Software Program Flows



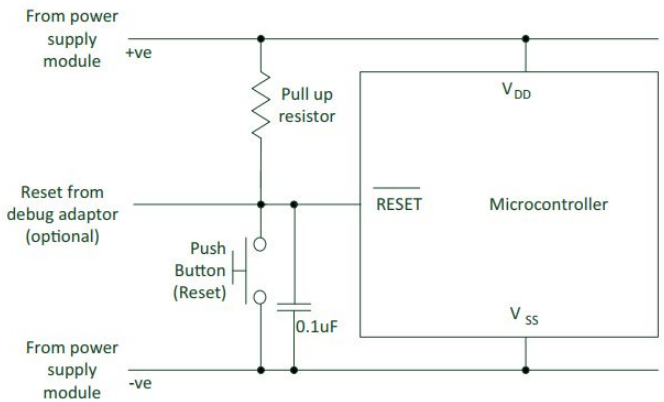
Review: Inside a program image

When we compile code for the target platform, there are different components:

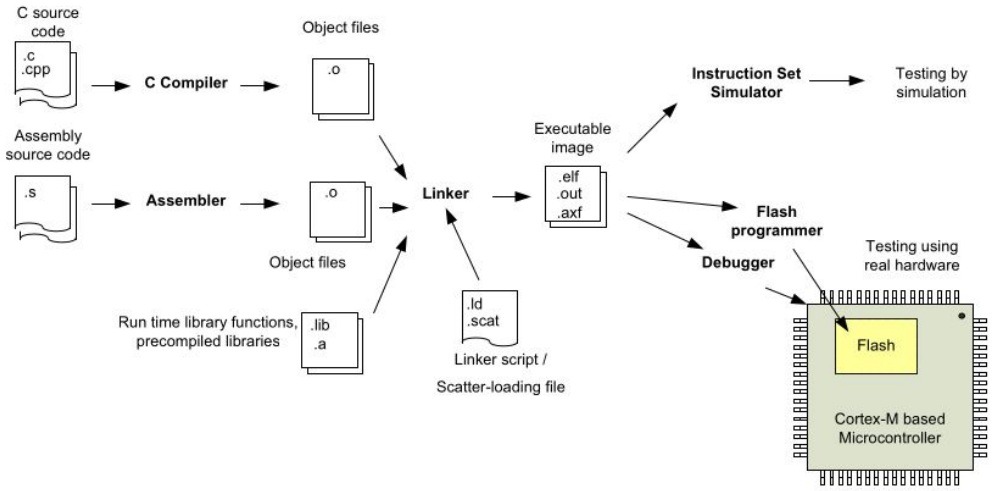
- **vector table** - contains the starting address of each **exception** and **interrupt**
- **reset handler** - some software and hardware initialization routines, clock initialization
- **startup code** - initialization of data and calling of **main()** function.
- **application code** - your code
- runtime library functions - any functions that you didn't implement
- **other data** - other global and static variables

Review: Programming Microcontrollers

A simple reset circuitry looks like this.



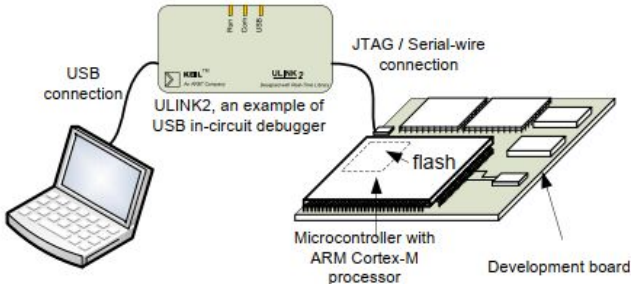
Review: Typical program generation flow



Review: Programming / Debugging

- After generating the binary (elf / bin) we can program the board using a **debugger** (or bootloader if present)
  - Usually are more expensive and has debugging capabilities.
- Program can be debugged using an **in-circuit debugger**.

Nucleo G031K8 board has built-in debugger on the back. No additional hardware required



Instruction Set

- Processors carry out operations by executing sequences of instructions.
  - web browsing, audio / video playback, text editing, etc.
- Each instruction defines a simple operation, and processors require a minimum of the following types of instructions
  - data processing, memory access, program flow control, etc.
- The instruction set supported by the Cortex-M Processors is called Thumb, and Cortex-M0+ supports only a subset of them (56). Most of them are 16-bits and a couple of them 32-bits.

Thumb instruction set for Cortex-M0+

16-bit Thumb instructions supported on Cortex-M0/M0+ processors									
ADC	ADD	ADR	AND	ASR	B	BIC	BLX	BKPT	BX
CMN	CMP	CPS	EOR	LDM	LDR	LDRH	LDRSH	LDRB	LDRSB
LSL	LSR	MOV	MVN	MUL	NOP	ORR	POP	PUSH	REV
REV16	REVSH	ROR	RSB	SBC	SEV	STM	STR	STRH	STRB
SUB	SVC	SXTB	SXTH	TST	UXTB	UXTH	WFE	WFI	YIELD

32-bit Thumb instructions supported on Cortex-M0/M0+ processors					
BL	DSB	DMB	ISB	MRS	MSR

- **Thumb** is designed for having good code density.
- Very limited set of instructions, not intended for high performance computing.
- Sometimes implemented with ARM instruction set in processors for both performance / code density.

Assembly Syntax

Based on different tools and assemblers, syntax will look different. **We will use GNU assembler syntax**

For armcc

```
label
   mnemonic operand1, operand2, ... ; comments
```

for GNU assembler

```
label:
   mnemonic operand1, operand2, ... // comments
```


Labels

```
label:
   mnemonic operand1, operand2, ... // comments
```

- **label** is used as a reference to an address or location
- can be used to get the instruction address to be used as branch target
- can also be used as a reference point for arrays
- will get removed after assembler

Mnemonic and operands

- Usually **instruction, destination, sources**.
- Number of operands depend on the instruction
- Immediate data usually prefixed with **#**

```
MOVS    R0, #0x12       /* Set R0 to 18       */
MOVS    R1, #'A'        // Set R1 to ASCII A
MOVS    R2, #22         @ Set R2 to 22
```

- Multi line comments can be made with `/*  */` pairs
- Single line comments can be made with `//` or `@` symbols.

Definitions

- definitions can be made with **.equ** keyword
- and later can be used with instructions
- **<= 8-bit** ones can be used with **MOV**
- **> 8-bit** ones needs to be used with **LDR**, which gets converter to *PC-relative addressing*.

```
.equ NVIC_IRQ_SETEN,    0xE000E100
.equ NVIC_IRQ0_ENABLE,  0x1

LDR  R0,=NVIC_IRQ_SETEN /* Put 0xE000E100 in R0 */
MOVS R1, #NVIC_IRQ0_ENABLE /* Put immediate data
(0x1) into register R1 */
```


Inserting data

There are different directives for inserting data with different sizes

Type of data to insert	ARM® assembler (e.g., Keil® MDK-ARM)	GNU assembler
Byte	DCB e.g., DCB 0x12	.byte e.g., .byte 0x012
Half word	DCW e.g., DCW 0x1234	.hword/.2byte e.g., .hword 0x01234
Word	DCD e.g., DCD 0x01234567	.word/.4byte e.g., .word 0x01234567
Double word	DCQ e.g., DCQ 0x12345678FF0055AA	.quad/.octa e.g., .quad 0x12345678FF0055AA
Floating point (Single precision)	DCFS e.g., DCFS 1E3	.float e.g., .float 1E3
Floating point (Double precision)	DCFD e.g., DCFD 3.14159	.double e.g., .double 3f14159
String	DCB e.g., DCB “Hello\n”, 0	.ascii/.asciz (with NULL termination) e.g., .ascii “Hello\n” .byte 0/*add NULL character */ e.g., .asciz “Hello\n”
Instruction	DCI e.g., DCI 0xBE00 ; Breakpoint (BKPT 0)	.inst/.inst.w e.g., .inst 0xbe00 /*Breakpoint (BKPT 0) */

Suffixes

- Some instructions can follow suffixes as shown below.
- Most data processing instructions will update APSR flags.

```
MOV  R0, R1 /* Move R1 into R0 */
MOVS R0, R1 /* Move R1 into R0, update NZ flags */
```

- Other group can be combined with branch operation to alter program flow.

Suffix	Descriptions
S	Update APSR (flags); for example, ADDS R0, R1 ; this ADD operation will update APSR
EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE	Conditional execution. EQ = Equal, NE = Not Equal, LT = Less Than, GT = Greater Than, etc. On the Cortex-M0 processor these conditions can only be applied to conditional branches. For example, BEQ label ; Branch to label if equal

Unified Assembler Language (UAL)

- With the ARM, Thumb, and Thumb-2 technology, Unified Assembler Language (UAL) syntax is developed to have a consistent syntax across all codes.
- `.syntax unified` directive in GNU assembler.

```
ADDS R0, R0, R1  /* R0 = R0 + R1, update APSR */
```

in most cases second **R0** can be omitted if tools allow it.

```
ADDS R0, R1
```

Instructions: Moving data within Processor

```
MOV  <Rd>, <Rm> /* Rd = Rm. Both can be high or low registers */
MOVS <Rd>, <Rm> /* Rd = Rm, update flags (Z, N). Both low registers */
ADDS <Rd>, <Rm>, #0 /* Rd = Rm, update flags (Z, N, C). Both low registers */
MOVS <Rd>, #imm8 /* Rd = imm8, update flags.  (Z, N). Immediate data range 0 to +255. low register */
```

Note: If we want to load immediate data that is more than 8-bits, we first need to store the data in memory space, then use *memory access instructions*.

Questions

Q1. R1 holds **0x13** and **R2** holds **0x24**. Swap them.

Q2. Write **0x24** into **R9**.

In Thumb instruction set, it is very tricky to play with high registers (**R8 - R12**). So avoid using them.

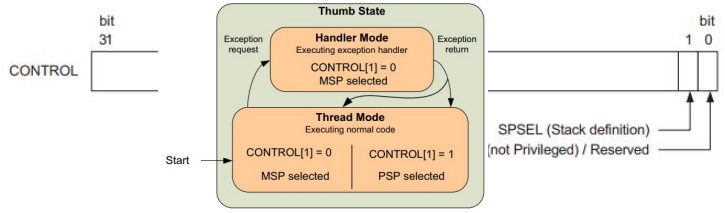
Instructions: Special Register Access

Special registers require special instructions to access.

```
MRS <Rd>, <SpecialReg> /* Read special register into Rd */
MSR <SpecialReg>, <Rd> /* Move Rd into special register */
```

Symbol	Register	Access type
APSR	Application Program Status Register (PSR)	Read/Write
EPSR	Execution PSR	No accesses (read as zero)
IPSR	Interrupt PSR	Read only
IAPSR	Composition of IPSR and APSR	Read only
EAPSR	Composition of EPSR and APSR	Read only (EPSR read as zero)
IEPSR	Composition of IPSR and EPSR	Read only (EPSR read as zero)
XPSR	Composition of APSR, EPSR, and IPSR	Read only (EPSR read as zero)
MSP	Main Stack Pointer	Read/Write
PSP	Process Stack Pointer	Read/Write
PRIMASK	Primary Exception Mask register	Read/Write
CONTROL	CONTROL register	Read/Write

Questions



Q1. Change **Main Stack Pointer** to **Process Stack Pointer**

Q2. Change **Process Stack Pointer** to **Main Stack Pointer**

Instructions: Memory access

There are a number of memory access instructions supporting various sizes.

```
LDR  <Rt>, [<Rn>, <Rm>] /* Word read single memory
data into register. Rt = memory[Rn + Rm]. All are
low registers */
LDRH <Rt>, [<Rn>, <Rm>] /* Half read */
LDRB <Rt>, [<Rn>, <Rm>] /* Byte read */
```

Transfer size	Unsigned load	Signed load	Signed/ Unsigned store
Word	LDR	LDR	STR
Half word	LDRH	LDRSH	STRH
Byte	LDRB	LDRSB	STRB

Instructions: Memory access - Note

- Memory accesses should always be aligned.
- Unaligned accesses are not supported and will throw a **HardFault exception**.
  - Word accesses should have `bits[1:0] = 00`
  - Half word accesses should have `bit[0] = 0`

For example:

```
LDR  R4, [R3, R2]
/* R3 + R2 should be 0xXXXXXX00 */
```


Instructions: Memory access - Immediate

- We can also use immediate addressing
- All are low registers

```
LDR  <Rt>, [<Rn>, #imm5] /* Word read single memory
data into register.
Rt = memory[Rn +  ZeroExtend (#imm5 << 2)] */

LDRH <Rt>, [<Rn>, #imm5] /* Half read
Rt = memory[Rn +  ZeroExtend (#imm5 << 1)] */

LDRB <Rt>, [<Rn>, #imm5] /* Byte read
Rt = memory[Rn +  ZeroExtend (#imm5)]      */
```


Question

Q. Read from memory location 0x24 to R0 using different offset methods

Instructions: Memory access - PC Relative

- Cortex-M processors support PC relative addressing, that is generated by the pseudo instruction.
- The data is stored in literal data blocks along side instructions (*literal pool*).
- **Rt** should be low register.
- **+4** is used for pipelining nature of the processor.

```
LDR  <Rt>, [PC, #imm8] /* Word read
   Rt = memory[WordAligned(PC + 4) +
ZeroExtend(#immed8 << 2)] */
LDR  R0,=0x12345678 /* 0x12345678 will be stored
somewhere in the memory (literal pool) and will be
transferred to R0 */
```


Instructions: Memory access - Store

- Store instructions follow the same syntax as load instructions.

Example:

```
STR <Rt>, [<Rn>, #imm5] /* Word write single
register into memory.
memory[Rn +  ZeroExtend (#imm5 << 2)] = Rt */
```

Note: There are also multiple store / load instructions but we will not really use them when writing assembly code. Make sure to check them out for debugging purposes.

Question

Q.
Memory address 0x24 holds data 1124
Memory address 0x28 holds data 749812

mem[0x24] = 1124
mem[0x28] = 749812

Swap them.

Question

Q.
mem[0x13424] = 1124
mem[0x13428] = 749812

Swap them.

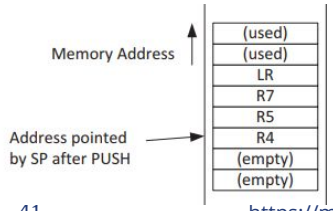
Instructions: Stack memory access

- There are two memory instructions dedicated for stack memory access. **PUSH** and **POP**
- Only low registers and LR, PC are supported

```
PUSH {<Ra>} /* Push Ra into Stack */
POP  {<Ra>} /* Pop stack value to Ra */
```

- Multiple registers can be pushed / popped
- Register range should be ascending order
- Higher register will be pushed first

```
PUSH {R1, R2, R5 - R7, LR} /* Store R1, R2, R5, R6, R7, and LR to stack. */
PUSH {R4, R5, R7, LR}
```



Instructions: Stack memory access

- A common combinations happen with function calls.
- Save processor state and return address.
- Then after the function completes, restore the processor state, and update the program counter to the return address.

```
my_func:
   PUSH {R4, R5, R7, LR} /* Store R4, R5, R7, LR */
   /* function body */
   POP {R4, R5, R7, PC} /* Restore R4, R5, R7, and pop LR value to PC */
```


Instructions: Arithmetic Operations – Add

- Most basic arithmetic operations are supported
- Can be register - register, or register - immediate
- All are low registers

```
ADD  <Rd>, <Rn>, <Rm> /* Rd = Rn + Rm */
ADDS <Rd>, <Rn>, <Rm> /* Rd = Rn + Rm, flags */
ADDS <Rd>, <Rn>, #imm3 /* Rd = Rn + #imm3, flags */
ADDS <Rd>, #imm8 /* Rd = Rd + #imm8, flags */
ADR  <Rd>, <label> /* Pseudo - same as above */
ADCS <Rd>, <Rm> /* Rd = Rd + Rm + Carry, flags */
```


Instructions: Arithmetic Operations – Sub

- Most basic arithmetic operations are supported
- Can be register - register, or register - immediate
- All are low registers

```
SUB  <Rd>, <Rn>, <Rm> /* Rd = Rn - Rm */
SUBS <Rd>, <Rn>, <Rm> /* Rd = Rn - Rm, flags */
SUBS <Rd>, <Rn>, #imm3 /* Rd = Rn - #imm3, flags */
SUBS <Rd>, #imm8 /* Rd = Rd - #imm8, flags */
SBCS <Rd>, <Rd>, <Rm> /* Rd = Rd - Rm - Borrow, flags */
RSBS <Rd>, <Rn>, #0 /* Rd = 0 - Rn, flags */
```

Question

Q.
mem[0x24] = 15
mem[0x28] = 18
mem[0x2C] = 17
mem[0x30] = 16

Find their **sum** and save it in **mem[0x34]**

Instructions: Arithmetic Operations - Compare

- There are also instructions that compare two operands (using subtract) and update APSR register
- Result is not stored

```
CMP  <Rn>, <Rm> /* Calculate Rn - Rm, flags */
CMP  <Rn>, #imm8 /* Calculate Rn - #imm8, flags. Rn is low register */
CMN  <Rn>, <Rm> /* Calculate negative, Rn - NEG(Rm), flags */
```


Instructions: Arithmetic Operations - Multiply

- There are is a multiplication instruction that multiplies two 32-bit numbers and returns 32-bit result.

```
MULS <Rd>, <Rn>, <Rm> /* Rd = Rn * Rm, N and Z update. All are low registers */
```

There is NO division instruction...

Instructions: Logical Operations

- Logical operations update NZ flags
- All use low registers

```
ANDS <Rd>, <Rn>, <Rm> /* And, Rd = Rn & Rm, NZ */
ORRS <Rd>, <Rn>, <Rm> /* Or,  Rd = Rn | Rm, NZ */
EORS <Rd>, <Rn>, <Rm> /* Xor, Rd = Rn ^ Rm, NZ */
BICS <Rd>, <Rn>, <Rm> /* Bit clear, Rd = Rn & ~Rm, NZ */
MVN <Rd>, <Rm> /* Not, Rd = ~Rm, NZ */
TST <Rn>, <Rm> /* And, Rn & Rm, NZ, result is not stored */
```

Question

Q1. mem[0x50002024] = 0x14142424

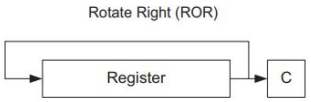
Set bit 4

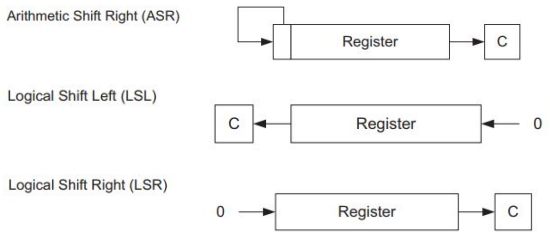
Q2. mem[0x50002024] = 0x14142424

Clear bit 19

Instructions: Shift and Rotate Operations

```
ASRS <Rd>, <Rd>, <Rm> /* Rd = Rd >> Rm, CNZ */
ASRS <Rd>, <Rd>, #imm5 /* Rd = Rd >> #imm5, CNZ */
LSLS <Rd>, <Rd>, <Rm> /* Rd = Rd << Rm, CNZ */
LSLS <Rd>, <Rd>, #imm5 /* Rd = Rd << #imm5, CNZ */
LSRS <Rd>, <Rd>, <Rm> /* Rd = Rd >> Rm, CNZ */
LSRS <Rd>, <Rd>, #imm5 /* Rd = Rd >> #imm5, CNZ */
RORS <Rd>, <Rd>, <Rm> /* Rd = Rd rotate by Rm bits,
CNZ */
```





Question

Q1. mem[0x50002024] = 0x14142424

Set bit 12

Q2. mem[0x50002024] = 0x14142424

Clear bit 19

Instructions: Program Flow Control

```
B <label> /* Branch to address. -+ 2046 bytes
offset of PC range */
B<cond> <label> /* Depending on APSR, branch to
address. -+ 254 bytes of PC range */
```

Example:

```
  MOVS R0, #3
loop:
  SUBS R0, #1
  BGT loop    // jump to loop if greater than
  NOP         // No/empty instruction
```

Question

Q.
mem[0x24] = 15
mem[0x28] = 18
...
mem[0x50] = 16

Find their **sum** and save it in **mem[0x54]**

Instructions: Branch <cond> suffixes

Suffix	Branch condition	Flags (APSR)
EQ	Equal	Z flag is set
NE	Not equal	Z flag is cleared
CS/HS	Carry set/unsigned higher or same	C flag is set
CC/LO	Carry clear/unsigned lower	C flag is cleared
MI	Minus/negative	N flag is set (minus)
PL	Plus/positive or zero	N flag is cleared
VS	Overflow	V flag is set
VC	No overflow	V flag is cleared
HI	Unsigned higher	C flag is set and Z is cleared
LS	Unsigned lower or same	C flag is cleared or Z is set
GE	Signed greater than or equal	N flag is set and V flag is set, or N flag is cleared and V flag is cleared (N == V)
LT	Signed less than	N flag is set and V flag is cleared, or N flag is cleared and V flag is set (N != V)
GT	Signed greater then	Z flag is cleared, and either both N flag and V flag are set, or both N flag and V flag are cleared (Z == 0 and N == V)
LE	Signed less than or equal	Z flag is set, or either N flag set with V flag cleared, or N flag cleared and V flag set (Z == 1 or N != V)

Instructions: Program Flow Control

- **`BL <label>` Branch and Link.** Branch to address and store return address to Link Register. Use for function calls. range is -+ 16MB of PC
- **`BX <Rm>` Branch and Exchange.** Branch to register value, and change processor state depending on register bit[0]. Since Cortex-M0+ only supports Thumb mode, Register bit[0] should be 1. Otherwise fault exception.
- **`BLX <Rm>` Branch, Link and Exchange.** Branch to register value, save return address to Link Register, and change processor state depending on register bit[0].

Instructions: Function calls

- Use `BL <label>` for function calls. It will update LR with LSB set to 1, and `BX LR` to return back from function.

```
main:
  ...
  BL func1 /* b to func1, set lr to next instr */
  MOVS R0, R2
  ...

func1:
  ...
  BX LR /* return to where lr points to */
```

Exception and Sleep mode instructions

Sometimes, for critical sections, we don't want to be interrupted:

```
CPSIE I /* Enable all interrupts (Clear PRIMASK) */
CPSID I /* Disable all interrupts (Set PRIMASK) */
```

Sleep mode instructions:

```
WFI /* Wait for Interrupt */
WFE /* Wait for Event (or interrupt) */
```


Question

mem[0x50000800] = 0xXXXXXXXX
mem[0x50000814] = 0xXXXXXXXX

Q1. Write **01** to **bits[13:12]** to **0x50000800** without changing the rest of the bits.

Q2. Write **1** to **bit[6]** to **0x50000814** without changing the rest of the bits.

This week

- Find / order hardware components
- Read Chapter 5 from Yiu
- Project 1 due on 6th week
- Assignment 2, Lab 1

Links

- ARMv6-M Architecture Reference Manual - https://developer.arm.com/documentation/ddi0419/c/