Microprocessors

Fall 2020

4. Assembly Language Usage, Memory and Faults

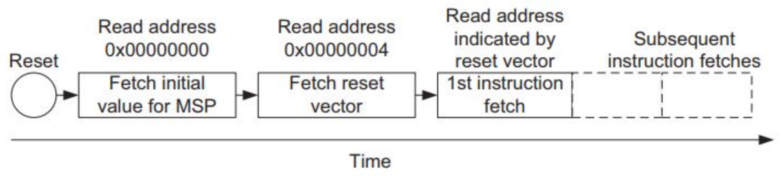
https://micro.furkan.space

Tentative Weekly Schedule

- Week x1 - Introduction to Course
- Week x2 - Architecture
- Week x3 - Assembly Language Introduction
- **Week x4 - Assembly Language Usage, Memory 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 - DMA
- Week xD - RTOS
- Week xE - Wireless Communications

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.

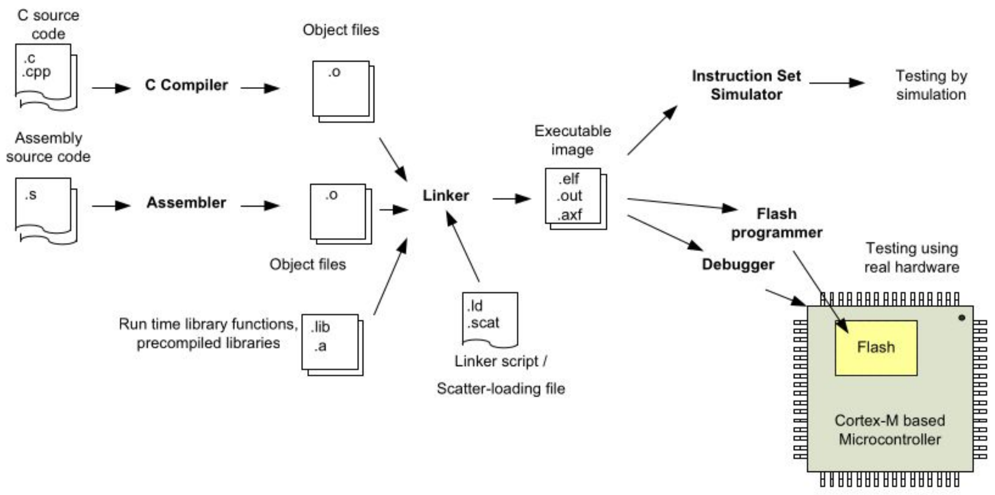

Reset
Read address 0x00000000
Fetch initial value for MSP
Read address 0x00000004
Fetch reset vector
Read address indicated by reset vector
1st instruction fetch
Subsequent instruction fetches
Time

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: Typical program generation flow



Review: 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 */
```

Program Control: if-then-else

- One of the most important functions is to handle conditional branches.
- Consider the given C code

```
if (counter > 10)
    counter = 0;
else
    counter += 1;
```

- In assembly, multiple ways.

```
  cmp r0, #10
  bgt zero_counter
  adds r0, r0, #1
  b counter_done
zero_counter:
  movs r0, #0
counter_done:
```

```
  cmp r0, #10
  ble incr_counter
  movs r0, #0
  b counter_done
incr_counter:
  adds r0, r0, #1
counter_done:
```

Program Control: loop

- Another important operation is looping.
- Consider the given C code

```
int sum = 0, i = 0;
for (; i<5; ++i)
   sum += i;
```

- In assembly

```
  movs r0, #0        // sum = 0
  movs r1, #0        // i = 0
loop:
  adds r0, r0, r1    // sum += i
  adds r1, r1, #1    // ++i
  cmp r1, #5         // compare r1 with 5
  blt loop           // if less than, b to loop
```

Branch Instructions

Branch type	Examples
Normal branch—branch always carry out.	B label (Branch to address marked as "label")
Conditional branch—branch depends on the current status of APSR and the condition specified in the instruction	BEQ label (Branch if Z flag is set, which is result from an equal comparison or ALU operation with result of zero.)
Branch and link—branch always carries out and updates the Link Register (LR, R14) with the instruction address following the executed BL instruction.	BL label (Branch to address "label," and Link Register updated to the instruction after this BL instruction.)
Branch and exchange state—Branch to address stored in a register. The LSB of the register should be set to 1 to indicate Thumb® state. (Cortex®-M0 and Cortex-M0+ processors do not support ARM® instructions so Thumb state must be used.)	BX LR (Branch to address stored in the Link register. This instruction is often used for function return.)
Branch and link with exchange state—Branch to address stored in a register, with the Link Register (LR/R14) updated to the instruction address following the executed BLX instruction. The LSB of the register should be set to 1 to indicate Thumb state. (Cortex-M0 and Cortex-M0+ processors do not support ARM instructions so Thumb state must be used.)	BLX R4 (Branch to address stored in the R4 and LR is updated to the instruction following the BLX instruction. This instruction is often used for calling functions addressed by function pointers.)

Typical Usages of Branch Conditions

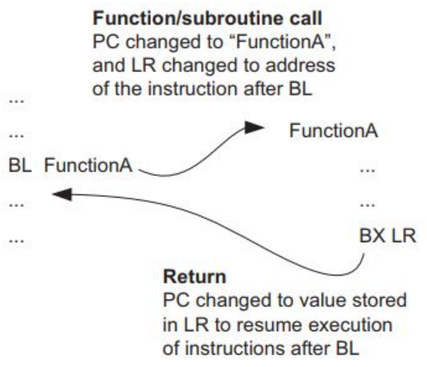
- A number of conditions are available for the conditional branches. After a `CMP R0, R1` instruction:

Required branch control	Unsigned data	Signed data
If (R0 equal R1) then branch	BEQ label	BEQ label
If (R0 not equal R1) then branch	BNE label	BNE label
If (R0 > R1) then branch	BHI label	BGT label
If (R0 >= R1) then branch	BCS label/BHS label	BGE label
If (R0 < R1) then branch	BCC label/BLO label	BLT label
If (R0 <= R1) then branch	BLS label	BLE label

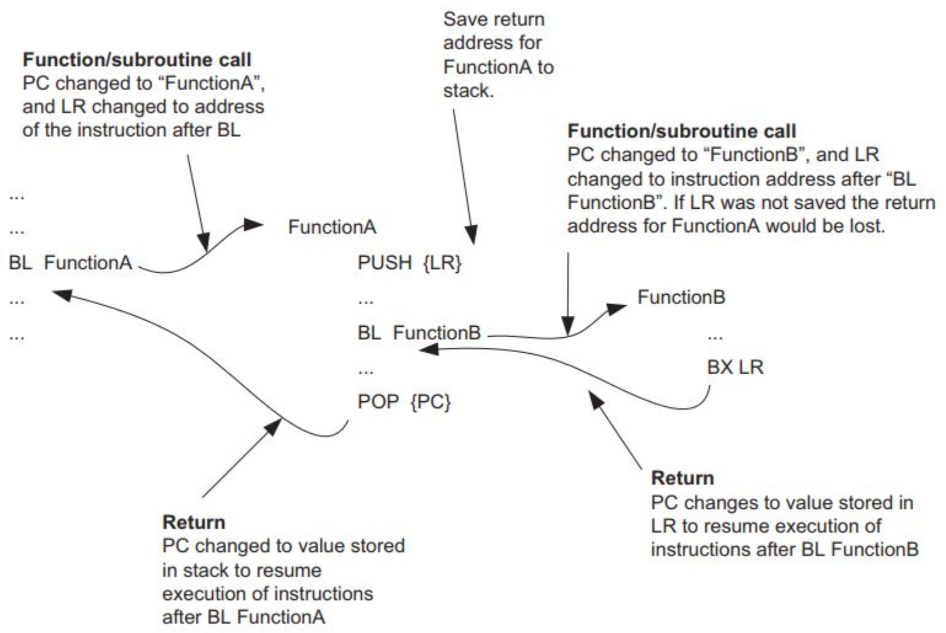
Required branch control	Unsigned data	Signed data
If (result >= 0) then branch	Not applicable	BPL label
If (result < 0) then branch	Not applicable	BMI label

Simple function or subroutine calls

Instruction example	Scenarios
BL function	Target function address is fixed and the offset is within +/−16 MB
LDR R0, =function; (other registers could also be used) **BLX R0**	Target function address can be changed during run time. No branch offset limitation.


Function/subroutine call
PC changed to "FunctionA", and LR changed to address of the instruction after BL
...
...
BL FunctionA
...
...
FunctionA
...
...
BX LR
Return
PC changed to value stored in LR to resume execution of instructions after BL

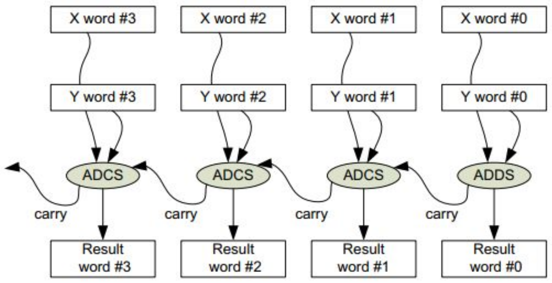
Nested function calls


Function/subroutine call
PC changed to "FunctionA", and LR changed to address of the instruction after BL
Save return address for FunctionA to stack.
Function/subroutine call
PC changed to "FunctionB", and LR changed to instruction address after "BL FunctionB". If LR was not saved the return address for FunctionA would be lost.
...
...
BL FunctionA
...
...
FunctionA
PUSH {LR}
...
BL FunctionB
...
POP {PC}
FunctionB
...
BX LR
Return
PC changed to value stored in stack to resume execution of instructions after BL FunctionA
Return
PC changes to value stored in LR to resume execution of instructions after BL FunctionB

64-Bit/128-Bit Addition

Adding two 64-bit values using 32-bit registers can be done with partitioning.

```
// 64-bit addition X + Y
LDR r0, =0xFFFFFFFF  // X_Low ( X = 0x3333FFFFFFFFFFFF)
LDR r1, =0x3333FFFF  // X_High
LDR r2, =0x00000001  // Y_Low ( Y = 0x3333000000000001)
LDR r3, =0x33330000  // Y_High
ADDS r0, r0, r2 // lower 32
ADCS r1, r1, r3 // upper 32
```



Bit and Bit Field computations

- In microcontrollers we need to do lots of bit operations. A very basic example is to check whether a button is pressed.
- Consider a button is connected to Pin 5 of Portx

```
LDR  R0, =0x40021014  // GPIOx IDR register
LDR  R1, [R0]         // Read register to R1 0b 0000_0000_00x0_0000
LDR  R2, =0x0020      // R2 = 0b 0000_0000_0010_0000 = 0x 0020
ANDS R1, R1, R2       // R1 = 0b 0000_0000_00x0_0000 = 0x 0020
CMP  R1, R2           // Compare two registers
BEQ button_pressed   // If equal, button is pressed (x is 1)
```

Bit and Bit Field computations

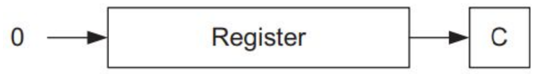
- Since memory access is usually a little slow, we can remove the second memory access by slight adjusting our code

```
LDR R0, =0x40021014  // GPIOx IDR register
LDR R1, [R0]         // Read register to R1 0b 0000_0000_00x0_0000
MOVS R2, #1          // Assign 1 to R2
LSLS R2, R2, #5      // Shift 1, 5 times to the left
ANDS R1, R1, R2      // R1 = 0b 0000_0000_00x0_0000 = 0x 0020
CMP R1, R2           // Compare two registers
BEQ button_pressed   // If equal, button is pressed (x is 1)
```

- Instruction count increased by 1, but might be worth it

Bit and Bit Field computations

- We can improve this by using the carry flag APSR
- Recall **Logical Shift Right** instruction shifts LSB to carry bit



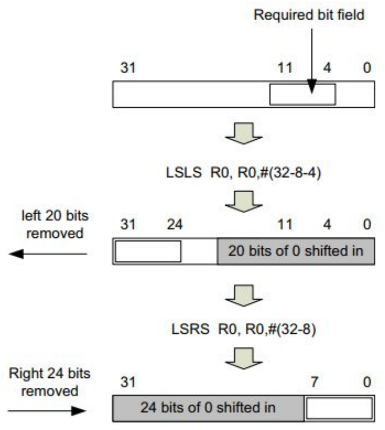
```
LDR R0, =0x40021014  // GPIOx IDR register
LDR R1, [R0]         // Read register to R1 0b 0000_0000_00x0_0000
LSRS R1, R1, #6      // Shift R0 6 times to right
                     //   moving bit 5 to the carry flag.
BCS button_pressed   // Branch if carry is set
```

- Instruction count decreased by 1, and 1 less memory access.

Bit and Bit Field computations

- If we want to extract a region of bits from a register, we can first shift left to remove unneeded op bits, then shift right to move required region lsb to bit 0
- Example, we want bits 11-4 for further processing

```
// Shift R0 20 times to left
LSLS R0, R0, #20
// Shift R0 24 times to right
LSRS R0, R0, #24
```



Array representation

- Arrays can be represented by inserting data **sequentially.**
- First element of the array can be accessed with a label given in the beginning.
- `.word`, `.byte` (or armcc equivalents `DCD` and `DCB` can be used.)

```
    ldr r0, =myarray     // Get the address of first element
    ldr r1, [r0]         // get the data in first element myarray[0]
    adds r0, r0, #4      // increment address by 4 (since word)

myarray:
    .word 0x12345678
    .word 0xA3395679
    .word 0x62345613
    .word 0x423B5615
    .word 0x22345628
    .word 0x42345678
```

Memory System

In typical microcontrollers, we need some sort of

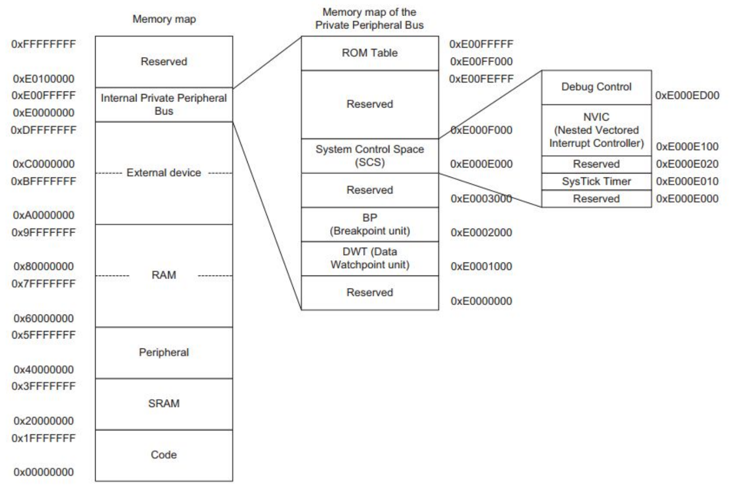
- Non-Volatile Memory for program storage
  - Flash, SD Card, ROM, HDD, SSD
- Scratch Memory for temporary data storage
  - SRAM

Most microcontrollers, these memories are integrated in the chip, however their size is limited.

- Depends on the manufacturer.
- Usually have a decent selection
  - Around 128 KB ROM and 32 KB RAM
- STM32G031T8 has 64KB of ROM and 8KB of RAM

Memory Map

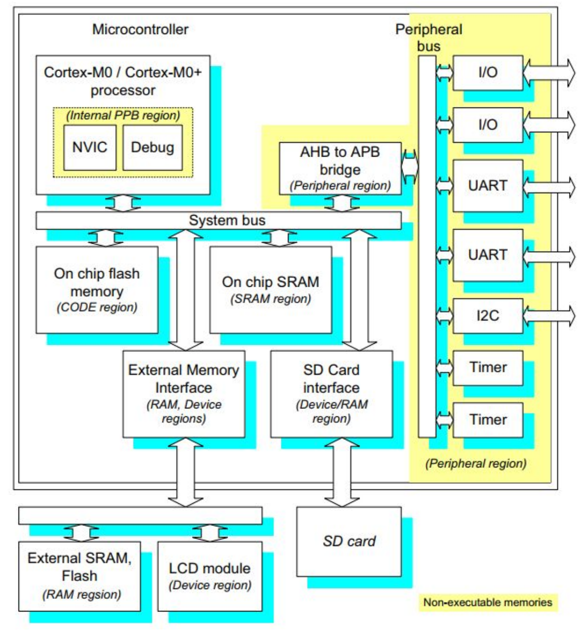
The memory space of Cortex-M0+ processors is architecturally divided into a number of regions.



Memory Map

- **Code Region (`0x00000000 - 0x1FFFFFFF`) - 512 MB**
  - Primarily used to store program code, including vector table.
- **SRAM Region (`0x20000000 - 0x3FFFFFFF`) - 512 MB**
  - Primarily used to store data, including stack.
  - It can also store program code.
  - It can be any read-write memory type
- **Peripheral Region (`0x40000000 - 0x5FFFFFFF`) - 512 MB**
  - Primarily used for peripherals.
  - It can also store data.
  - Execution is not allowed in this region
  - AHB-Lite or APB peripherals.
- **SRAM Region (`0x60000000 - 0x9FFFFFFF`) - 1GB**
- **Device Region (`0xA0000000 - 0xDFFFFFFF`) - 1GB**
- **Internal Private Peripheral Bus (`0xE0000000 - 0xE00FFFFF`) - 1MB**
  - Allocated for peripherals inside the processor (i.e. NVIC)
  - Program execution is not allowed
- **Reserved (`0xE0100000 - 0xFFFFFFFF`) - 511MB**

Example usage of memory regions

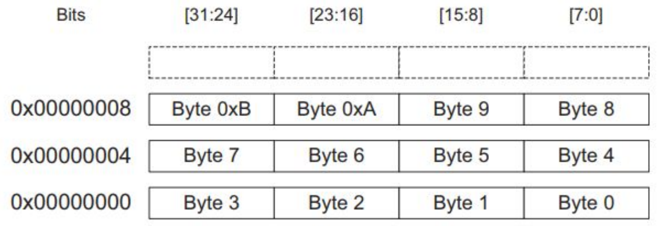


Memory Format

- Memory can be arranged in two different ways:
  - **little endian:** lowest byte is stored in lower address
  - **big endian:** lowest byte is stored in higher address
- Cortex-M0+ processors support either little or big endian format decided by the microcontroller vendor.
  - Most Cortex-M processors use little endian.

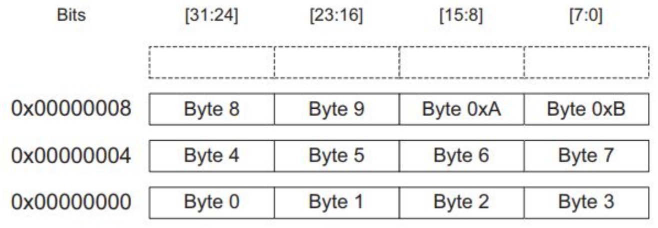
Memory Format

- Little endian memory



Bits	[31:24]	[23:16]	[15:8]	[7:0]
0x00000008	Byte 0xB	Byte 0xA	Byte 9	Byte 8
0x00000004	Byte 7	Byte 6	Byte 5	Byte 4
0x00000000	Byte 3	Byte 2	Byte 1	Byte 0

- Big endian memory



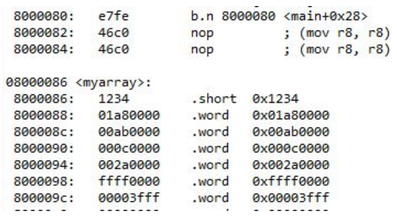
Bits	[31:24]	[23:16]	[15:8]	[7:0]
0x00000008	Byte 8	Byte 9	Byte 0xA	Byte 0xB
0x00000004	Byte 4	Byte 5	Byte 6	Byte 7
0x00000000	Byte 0	Byte 1	Byte 2	Byte 3

Memory alignment

- Cortex-M0+ is really picky about memory alignment
  - only 32-bit aligned (**word aligned**) memory accesses are supported
- We need to make sure compiler aligns our code, using `.align` directive.

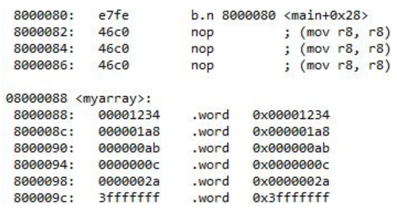
```
    b .
    nop
    nop

.align
myarray:
    .word 0x1234
    .word 424
    .word 0b0010101011
    .word 12
    .word 42
    .word 0x3FFFFFFF
```

w/o .align



```
 8000080:   e7fe        b.n 8000080 <main+0x28>
 8000082:   46c0        nop             ; (mov r8, r8)
 8000084:   46c0        nop             ; (mov r8, r8)

08000086 <myarray>:
 8000086:   1234        .short  0x1234
 8000088:   01a80000    .word   0x01a80000
 800008c:   00ab0000    .word   0x00ab0000
 8000090:   000c0000    .word   0x000c0000
 8000094:   002a0000    .word   0x002a0000
 8000098:   ffff0000    .word   0xffff0000
 800009c:   00003fff    .word   0x00003fff
```

with .align



```
 8000080:   e7fe        b.n 8000080 <main+0x28>
 8000082:   46c0        nop             ; (mov r8, r8)
 8000084:   46c0        nop             ; (mov r8, r8)
 8000086:   46c0        nop             ; (mov r8, r8)

08000088 <myarray>:
 8000088:   00001234    .word   0x00001234
 800008c:   000001a8    .word   0x000001a8
 8000090:   000000ab    .word   0x000000ab
 8000094:   0000000c    .word   0x0000000c
 8000098:   0000002a    .word   0x0000002a
 800009c:   3fffffff    .word   0x3fffffff
```

Memory layout - Sections

- The program memory is divided into areas for organization.
  - **text, data, bss, heap** and **stack**.
- **text** segment holds the code and usually found in ROM. It is read-only memory (when program is executing) and code can be executed from here.
- **data, bss, heap** and **stack** sections holds the variables, and temporary variables and they are located in RAM.

```
.section .vectors /* place following into .vectors section */
.section .text    /* place following into .text section */
.section .data    /* place following into .data section */
```

- In C these happen automatically, in assembly we need to place things.
- We will see these sections in detail next week.

Section placements

- The program memory is divided into areas for organization.
  - **text, data, bss, heap** and **stack**.
- **How does the compiler know where things go?**

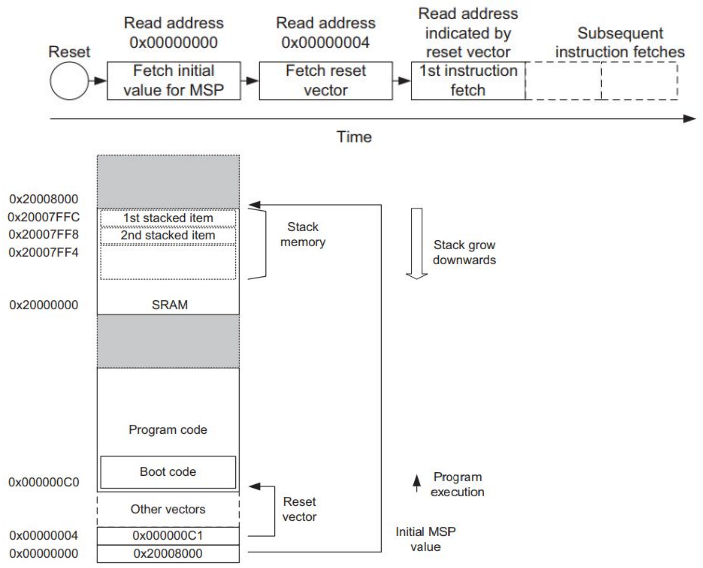
```
MEMORY
{
 ROM  (rx) : ORIGIN = 0x08000000, LENGTH = 64K
 RAM (rwx) : ORIGIN = 0x20000000, LENGTH =  8K
}

ENTRY(Reset_Handler)

/* end of RAM */
_estack = ORIGIN(RAM) + LENGTH(RAM);
```

```
SECTIONS
{
  .text :
  {
    KEEP(*(.vectors))
    *(.text*)
  } >ROM

  .data :
  {
    *(.data*)
  } >RAM AT> ROM

  .bss :
  {
    *(.bss*)
  } >RAM
}
```

Linker Script

Reset Sequence

We talked about Reset Sequence and what happens.



Reset Sequence - Assembly listing

Let's look at how it is handled in assembly.
First embed some values in an address.

```
/* vector table, +1 thumb mode */
.section .vectors
vector_table:
    .word _estack              /*     Stack pointer */
    .word Reset_Handler +1     /*     Reset handler */
    .word NMI_Handler +1       /*       NMI handler */
    .word HardFault_Handler +1 /* HardFault handler */
    /* rest of the vector table */
```

ResetHandler

- **ResetHandler** is the first code that gets executed.
- Usually hardware initializations are done here
- Finally calls **main** function.

```
.section .text
Reset_Handler:
    ldr r0, =_estack    /* set stack pointer */
    mov sp, r0          /*  ... */

    bl init_data        /* initialize data (and bss) sections */
    bl main             /* call main function */
    b .                 /* just in case main exits */
```

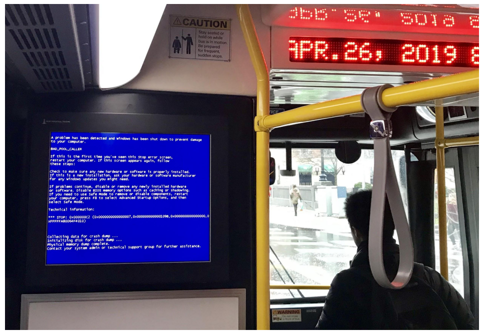
How to see the assembly listing of an elf file

```
$ arm-none-eabi-objdump -d .\asm.elf

.\asm.elf:     file format elf32-littlearm

Disassembly of section .text:

08000000 <vector_table>:
 8000000:       20002000        andcs   r2, r0, r0
 8000004:       08000013        stmdaeq r0, {r0, r1, r4}
 8000008:       08000011        stmdaeq r0, {r0, r4}
 800000c:       08000011        stmdaeq r0, {r0, r4}

08000010 <Default_Handler>:
 8000010:       e7fe            b.n     8000010 <Default_Handler>

08000012 <main>:
 8000012:       4e01            ldr     r6, [pc, #4]    ; (8000018 <main+0x6>)
 8000014:       6835            ldr     r5, [r6, #0]
 8000016:       e7fe            b.n     8000016 <main+0x4>
 8000018:       80000101        .word   0x80000101
```

What happens if this code is executed?

Faults

- In a processor, if a program goes wrong and the processor detects a **fault**, then a **fault exception** occurs.
- On Cortex M0+ processors, there is only one exception type that handles faults - **HardFault** exception.
- Sometimes there is none available. We will see what to do in those situations.



HardFault Exception

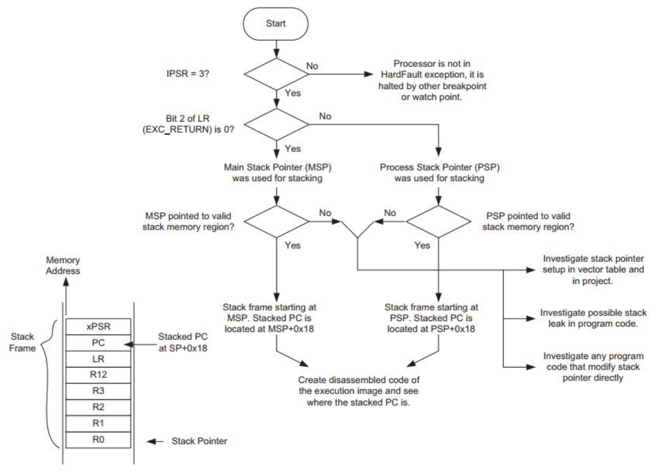
- HardFault exception is second highest priority exception type (with priority level -1)
- When a HardFault is triggered, we know that the microcontroller is in trouble.
  - Some kind of reset mechanism needs to be implemented **for production code** in case of **any unforeseen bugs**.
- It is also useful in development when debugging software to try to catch the bugs.

What can cause a Fault?

- There are a number of possible reasons for a fault to occur:
- **Memory related**
  - Bus error - an attempt to access an invalid address (generated by bus slave)
  - attempt to execute program from non-executable memory region
- **Program errors**
  - Undefined instruction execution
  - Switch to ARM state (from Thumb)
  - Unaligned memory access

Analyze a Fault

Depending on the type of fault, very often it is straightforward to locate the instruction that caused the HardFault exception.


Start
IPSR = 3?
No
Processor is not in HardFault exception, it is halted by other breakpoint or watch point.
Yes
Bit 2 of LR (EXC_RETURN) is 0?
No
Yes
Main Stack Pointer (MSP) was used for stacking
Process Stack Pointer (PSP) was used for stacking
MSP pointed to valid stack memory region?
No
No
PSP pointed to valid stack memory region?
Yes
Yes
Investigate stack pointer setup in vector table and in project.
Investigate possible stack leak in program code.
Investigate any program code that modify stack pointer directly
Stack frame starting at MSP. Stacked PC is located at MSP+0x18
Stack frame starting at PSP. Stacked PC is located at PSP+0x18
Create disassembled code of the execution image and see where the stacked PC is.
Memory Address
Stack Frame
xPSR
PC
LR
R12
R3
R2
R1
R0
Stacked PC at SP+0x18
Stack Pointer

Error Handling in Real Applications

- In real applications, embedded systems will be running **without a debugger**.
- In most cases HardFault exception handler can be used to carry out safety actions then reset the processor.
  - Perform application specific safety actions (e.g. perform shut down sequence in a motor controller)
  - Optionally the system can report the error within a user interface, then reset the system.

Error Handling in Real Applications

- Since a HardFault could be caused by an error in the **stack pointer** value, a HardFault handler programmed in **C language might not perform correctly**.
- It might depend on the stack memory to operate.
  - Write the HardFault in assembly language.
  - Write an assembly language routine to check if the stack pointer is in a valid memory range before entering a C routine to handle the fault exception.

This week

- Boards should be available.
- Read Chapter 6, 7, 11 from Yiu
- Project 1 due on 7th week
- Assignment 3
- Lab 2

Links

- `Linker manual - https://sourceware.org/binutils/docs/ld/`
- `Linker script output section - https://sourceware.org/binutils/docs/ld/Output-Section-Attributes.html#Output-Section-Attributes`
- `GNU Debugger - https://sourceware.org/gdb/current/onlinedocs/gdb`