EIE/ENE 334 Microprocessors

Lecture 10:

The Cortex-M0 Programming

Week #10 : Dejwoot KHAWPARISUTH

Adapted from

http://webstaff.kmutt.ac.th/~dejwoot.kha/

Program: swap register contents

To swap the contents of two registers without using an intermediate storage

Algorithm:

$$A = A \oplus B$$
$$B = A \oplus B$$
$$A = A \oplus B$$

```
; R0 <-> R1
   EORS R0,R0,R1
   EORS R1,R1,R0
   EORS R0,R0,R1
```

Program: factorial calculation

$$\mathbf{n!} = \prod_{i=1}^{n} \boldsymbol{i} = \boldsymbol{n(n-1)(n-2)\ldots(1)}$$

```
;
    MOVS R0,#10              ; n=10
    MOV R1,R0                ; copy to temp.
FLOOP
    SUBS R1,R1,#1
    BEQ ENDFACT
    MOVS R2,R1
    MULS R2,R0,R2
    MOV R0,R2
    MOVS R1,R1
    BNE FLOOP
ENDFACT
```

Program: arithmetic

Unsigned Addition

```
 95d =  01011111b          =   5Fh
189d =  10111101b          =   BDh
---------------------------------
284d   1)00011100b = 284d      1)1Ch
```

Signed Addition

```
-001d =  11111111b          =   FFh
+027d =  00011011b          =   1Bh
---------------------------------
+026d   1)00011010b = +026d    1)1Ah
```

Program: arithmetic

If positive numbers are added, there is the possibility that the sum will exceed + 127d, as demonstrated in the following example:

```
+100d  =   01100100b              =   64h
+050d  =   00110010b              =   32h
+150d      0)10010110b = −106d        0)96h
```

Program: arithmetic

An example of adding two positive numbers that do not exceed the positive limit is this:

+045d	=	00101101b	=	2Dh
+075d	=	01001011b	=	4Bh
+120d		0)01111000b = 120d		0)78h

Note: that there are no carries from bits 6 or 7 of the sum; the Carry and OV flags are both 0.

Program: arithmetic

The result of adding two negative numbers together for a sum that does not exceed the negative limit is shown in this example:

−030d	=	11100010b		=	E2h
−050d	=	11001110b		=	CEh
−080d		1)10110000b	= −080d		1)B0h

Here, there is a carry from bit 7 or the Carry flag is 1; there is a carry from bit 6 and the OV flag is 0.

Program: arithmetic

When adding two negative numbers whose sum does exceed -128d, we have:

```
-070d  =   10111010b          =   BAh
-070d  =   10111010b          =   BAh
-140d     1)01110100b = +116d     1)74h
```

In this example, there is a carry from bit position 7, and no carry from bit position 6, so the Carry and the OV flags are set to 1.

Program: arithmetic

Note: Overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{31}, or less than -2^{31}. The OV flag is set to 1 if there is a carry out of bit position 31, but not bit position 30 or if there is a carry out of bit position 30 but not bit position 31, which may be expressed as the logical operation:

OV = C31 XOR C30

Program: 64-bit addition

The following instructions add a 64-bit integer contained in [R3:R2] to another 64-bit integer contained in [R1:R0], and place the result in [R5:R4].

```
; [R5:R4] = [R3:R2]+[R1:R0]
   ADDS R4,R2,R0     ; add the Least Significant word first
   MOV R5,R3
   ADCS R5,R5,R1     ; Rd and Rn must be the same
```

Program: 96-bit subtraction

The following instructions subtract a 96-bit integer contained in [R3:R2:R1] to another 64-bit integer contained in [R6:R5:R4], and place the result in [R3:R2:R1].

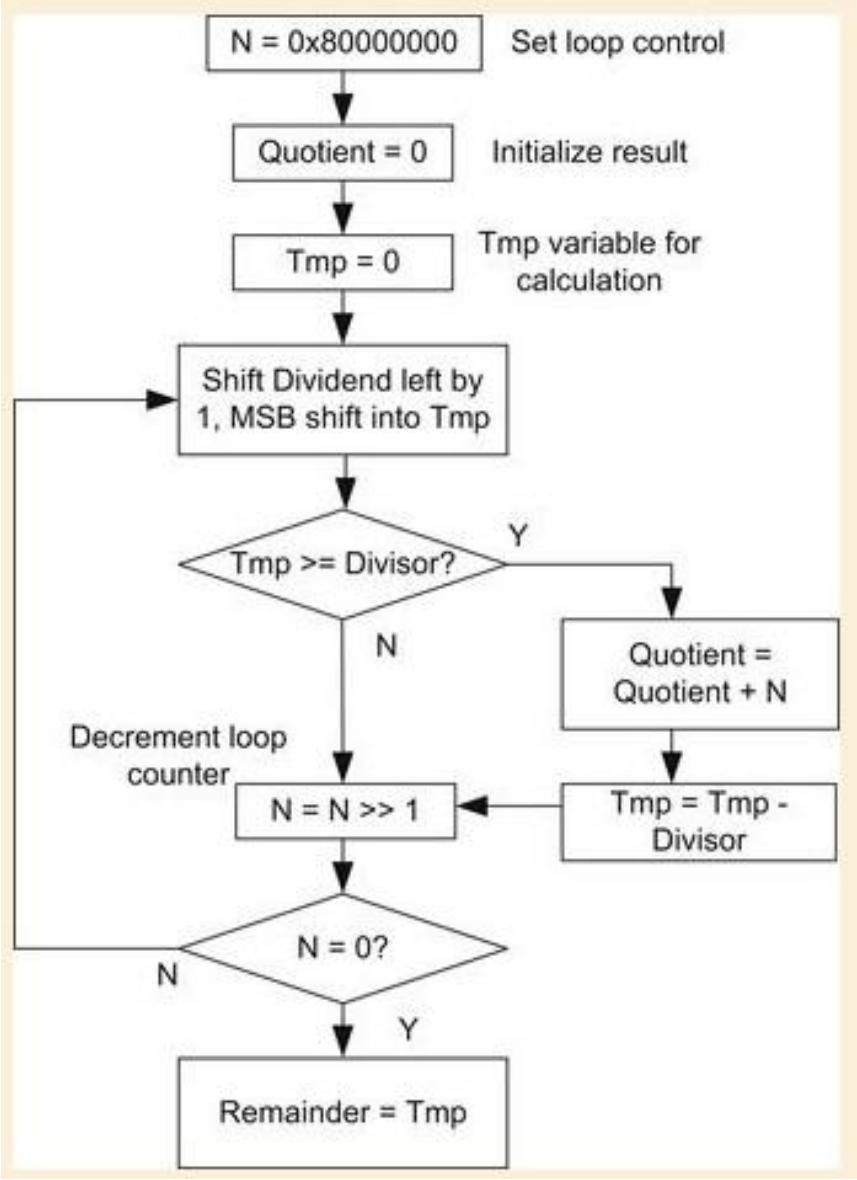
```
; [R3:R2:R1] = [R3:R2:R1] - [R6:R5:R4]
   SUBS R1,R1,R4     ; subtract the Least Significant word first
   SBCS R2,R2,R5
   SBCS R3,R3,R6     ; Rd and Rn must be the same
```

Program: multiplication by a constant

The following instructions multiply a 32-bit integer contained in R2 by 5 and place the result in R4 and R5.

```
; [R5:R4] = R2*5
   LSLS R4,R2,#2     ; R2*4
   LSRS R5,R2,#30
   MOVS R3,#0        ;  R2*(4+1)
   ADDS R4,R4,R2
   ADCS R5,R5,R3     ; Rd and Rn must be the same
```

Program: division



Program: division

```
simple_divide
; Inputs ; R0 = dividend ; R1 = divider
; Outputs ; R0 = quotient ; R1 = remainder
    PUSH {R2-R4}     ; Save registers to stack
    MOV R2, R0       ; Save dividend to R2 as R0 will be changed
    MOVS R3, #0x1    ; loop control
    LSLS R3, R3, #31 ; N = 0x80000000
    MOVS R0, #0      ; initial Quotient
    MOVS R4, #0      ; initial Tmp
simple_divide_loop
    LSLS R2, R2, #1 ; Shift dividend left by 1 bit, MSB go into carry
    ADCS R4, R4, R4 ; Shift Tmp left by 1 bit, carry move into LSB
    CMP R4, R1
    BCC simple_divide_lessthan
    ADDS R0, R0, R3 ; Increment quotient

    SUBS R4, R4, R1
simple_divide_lessthan
    LSRS R3, R3, #1 ; N = N > > 1
    BNE simple_divide_loop
    MOV R1, R4       ; Put remainder in R1, Quotient is already in R0
    POP {R2-R4}      ; Restore used register
    BX LR ; Return
```

Program: while loop

```
while (…){instructions}
```

```
;
   B TEST
LOOP
   …                    ; instruction
TEST
   …                    ; evaluate condition
   BNE LOOP
```

Program: do while loop

```
while (…){instructions}
```

```
;
LOOP      ; loop body
   …
   …      ; evaluate condition
   BNE LOOP
EXIT
   …
```

Program: for loop

```
for (i=0;i<10;i++){instructions}
```

```
;
   MOV R0,#10          ; i = 10
LOOP
   …
   SUBS R0,R0,#1       ; i=i-1
   BNE   LOOP
DONE
```

Program: for loop

```
for (i=0;i<10;i++){instructions}
```

```
;
   MOV R0,#0          ; i = 0
LOOP
   CMP R0,#10         ; i<10?
   BGE DONE
   …
   ADDS R0,R0,#1      ; i=i+1
   B    LOOP
DONE
```

Program: for loop (example)

```
for (i=0;i<8;i++){a[i]=b[7-i];}
; assume byte-wide array
SRAM_BASE_A   EQU   0x20000000
   MOVS R0,#0          ; i = 0
   LDR R1,=array_b
   LDR R2,=SRAM_BASE_A
   MOVS R3,#7
LOOP
   CMP R0,#8           ; i<8?
   BGE DONE
   SUBS R4,R3,R0       ; 7-i
   LDRB R5,[R1,R4]     ; b[7-i]
   STRB R5,[R2,R0]     ; a[i]=b[7-i]
   ADDS R0,R0,#1       ; i=i+1
   B     LOOP
DONE   B DONE
array_b DCB 0x12,0x34,0x56,0x78,0x9A,0xBC,0xDE,0xF0
```

Program: for loop

```
for (i=0;i<8;i++){a[i]=b[7-i];}
; assume byte-wide array
SRAM_BASE_A   EQU   0x20000000
   MOVS R0,#8          ; 8 rounds
   LDR R1,=array_b
   LDR R2,=SRAM_BASE_A
   MOVS R3,#0          ; R3=i
   MOVS R4,#7
LOOP
   SUBS R5,R4,R3       ; 7-i
   LDRB R6,[R1,R5]
   STRB R6,[R2,R3]
   ADDS R3,R3,#1       ; i=i+1
   SUBS R0,R0,#1
   BNE   LOOP

DONE  B DONE
array_b DCB 0x12,0x34,0x56,0x78,0x9A,0xBC,0xDE,0xF0
```

Program: while loop (example)

```
while (a!=b){
   if (a>b) a=a-b;
   else b=b-a;}
; the greatest common divisor algorithm
GCD
   CMP R0,R1          ; a>b?
   BEQ END            ; if a=b -> END
   BLT LESS
   SUBS R0,R0,R1      ; a=a-b
   B GCD
LESS
   SUBS R1,R1,R0      ; b=b-a
   B GCD
```

Program: while loop (example)

```
while (a!=b){
   if (a>b) a=a-b;
   else b=b-a;}
```

```
; the greatest common divisor algorithm
; NOT ARM Cortex-M0 instruction set
GCD
   CMP R0,R1          ; a>b?
   SUBGT R0,R0,R1     ; a=a-b
   SUBLT R1,R1,R0     ; b=b-a
   BNE GCD
```

Program: condition code suffixes

Suffix	Flags	Meaning
EQ	Z = 1	Equal, last flag setting result was zero
NE	Z = 0	Not equal, last flag setting result was non-zero
CS or HS	C = 1	Higher or same, unsigned
CC or LO	C = 0	Lower, unsigned
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned
LS	C = 0 or Z = 1	Lower or same, unsigned
GE	N = V	Greater than or equal, signed
LT	N != V	Less than, signed
GT	Z = 0 and N = V	Greater than, signed
LE	Z = 1 and N != V	Less than or equal, signed
AL	Can have any value	Always. This is the default when no suffix is specified.

Program: condition code suffixes

The condition flags

The APSR contains the following condition flags:

N Set to 1 when the result of the operation was negative, cleared to 0 otherwise.

Z Set to 1 when the result of the operation was zero, cleared to 0 otherwise.

C Set to 1 when the operation resulted in a carry, cleared to 0 otherwise.

V Set to 1 when the operation caused overflow, cleared to 0 otherwise.

For more information about the APSR see *Program Status Register* on page 2-4.

A carry occurs:

- if the result of an addition is greater than or equal to 2^{32}
- if the result of a subtraction is positive or zero
- as the result of a shift or rotate instruction.

Overflow occurs when the sign of the result, in bit[31], does not match the sign of the result had the operation been performed at infinite precision, for example:

- if adding two negative values results in a positive value
- if adding two positive values results in a negative value
- if subtracting a positive value from a negative value generates a positive value
- if subtracting a negative value from a positive value generates a negative value.

The Compare operations are identical to subtracting, for `CMP`, or adding, for `CMN`, except that the result is discarded. See the instruction descriptions for more information.

Program: branch condition

Signed and unsigned data operations or compare (CMP, CMN, TST)

For example: CMP R0,R1

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

Program: branch condition

To detect value overflow in add or subtract operations

Required Branch Control	Unsigned Data	Signed Data
If (overflow (R0 + R1)) then branch	BCS label	BVS label
If (no_overflow (R0 + R1)) then branch	BCC label	BVC label
If (overflow (R0 - R1)) then branch	BCC label	BVS label
If (no_overflow (R0 ? R1)) then branch	BCS label	BVC label

To detect whether an operation result is a positive or negative value

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

Program: function

```
; Main program
   BL FUNC_A
   NOP
   …                ;
                    FUNC_A
                       NOP
                       BX LR
```

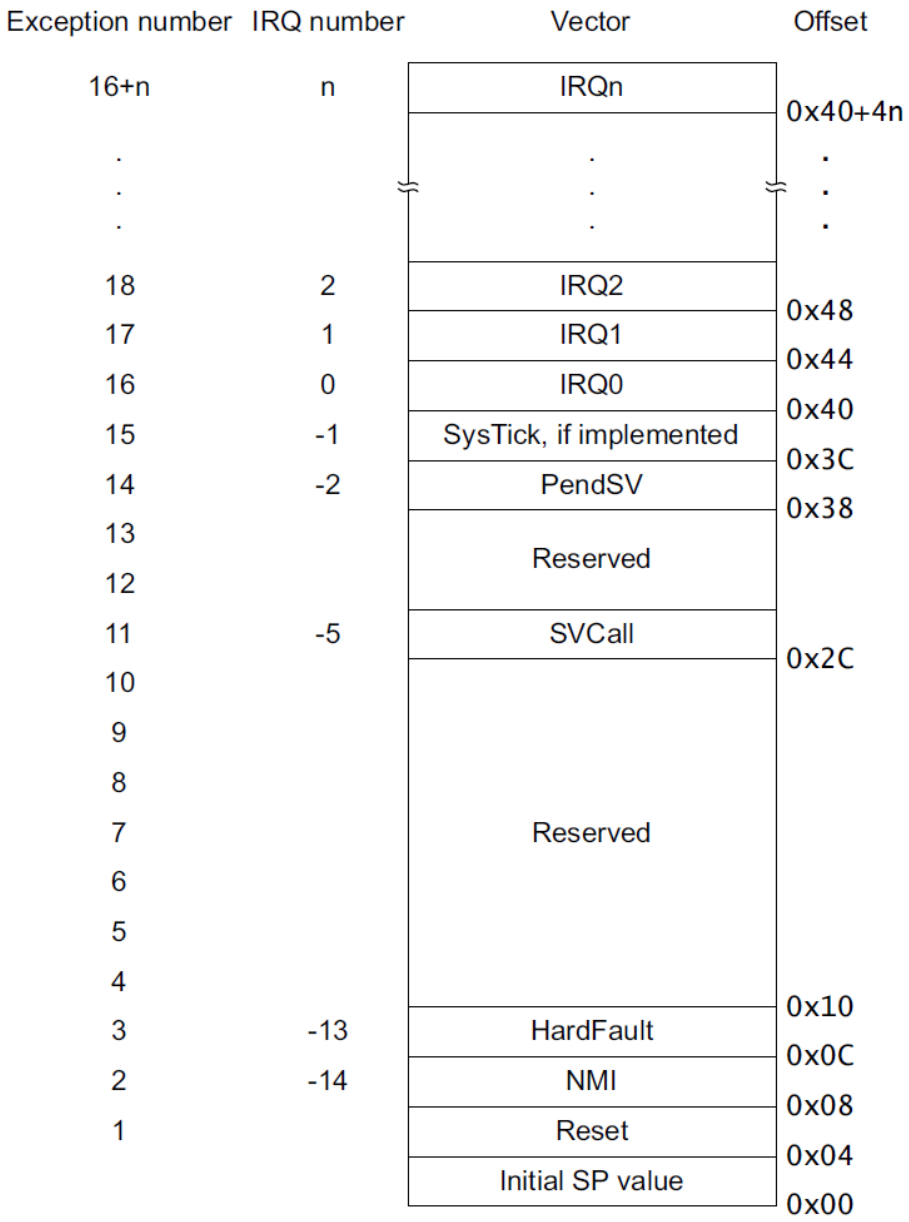
Program: function

```
; Main program
   BL FUNC_B
   NOP
   …                ;
                    FUNC_B
                       PUSH {R4-R6,LR}
                       NOP
                       BL FUNC_C
                       NOP
                       POP {R4-R6,PC}
                                          ;
                                          FUNC_C
                                             NOP
                                             BX LR
```

Exception:

Exception number[a]	IRQ number[a]	Exception type	Priority	Vector address[b]	Activation
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	Synchronous
4-10	-	Reserved	-	-	-
11	-5	SVCall	Configurable[e]	0x0000002C	Synchronous
12-13	-	Reserved	-	-	-
14	-2	PendSV	Configurable[e]	0x00000038	Asynchronous
15	-1	SysTick[c]	Configurable[e]	0x0000003C	Asynchronous
15	-	Reserved	-	-	-
16 and above[d]	0 and above	IRQ	Configurable[e]	0x00000040 and above[f]	Asynchronous

Vector table:



Vector Table for ARMv7-M

- **First entry contains initial Main SP**
- **All other entries are addresses for exception handlers**
  - Must always have LSBit = 1 (for Thumb)
- **Table has up to 496 external interrupts**
  - Implementation-defined
  - Maximum table size is 2048 bytes
- **Table may be relocated**
  - Use Vector Table Offset Register
  - Still require minimal table entries at 0x0 for booting the core
- **Each exception has a vector number**
  - Used in Interrupt Control and State Register to indicate the active or pending exception type
- **Table can be generated using C code**
  - Example provided later

Address		Vector #
0x40 + 4*N	External N	16 + N
…	…	…
0x40	External 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVC	11
0x1C to 0x28	Reserved (x4)	7-10
0x18	Usage Fault	6
0x14	Bus Fault	5
0x10	Mem Manage Fault	4
0x0C	Hard Fault	3
0x08	NMI	2
0x04	Reset	1
0x00	Initial Main SP	N/A

Exception Handling

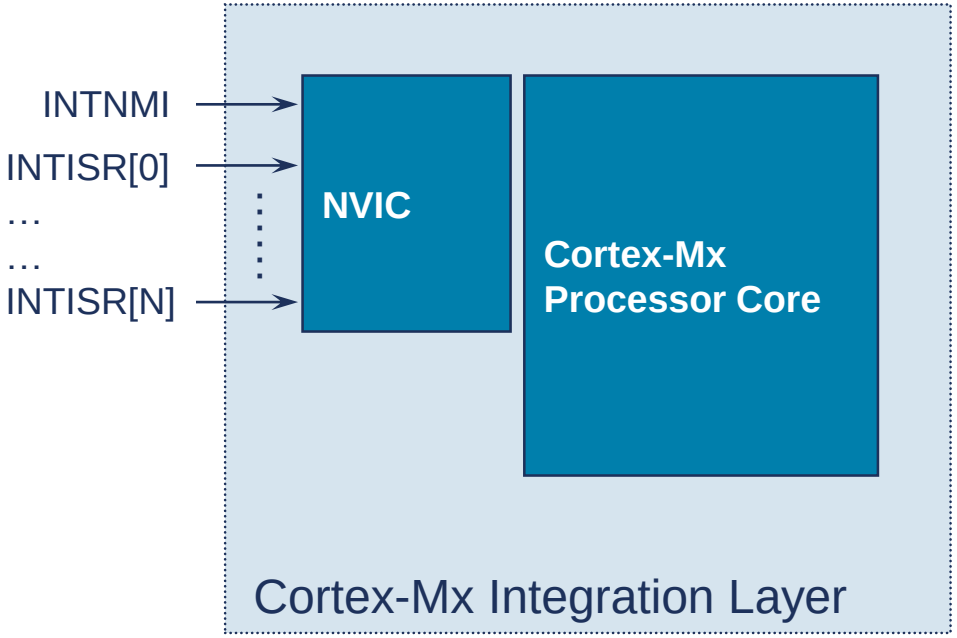
- **Exception types:**
  - Reset
  - Non-maskable Interrupts (NMI)
  - Faults
  - PendSV
  - SVCall
  - External Interrupt
  - SysTick Interrupt

- **Exceptions processed in Handler mode (except Reset)**
  - Exceptions always run privileged

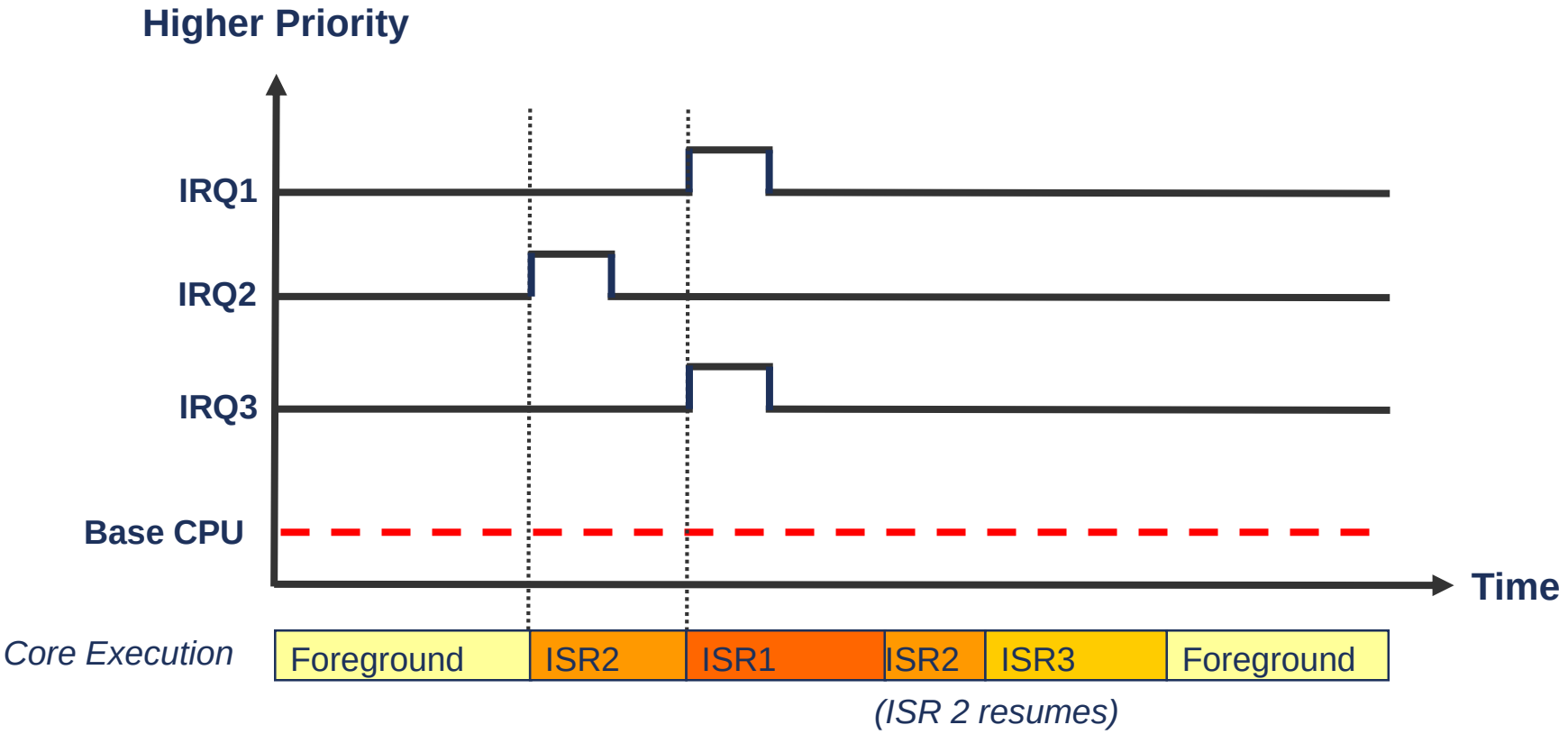
- **Interrupt handling**
  - Interrupts are a sub-class of exception
  - Automatic save and restore of processor registers (xPSR, PC, LR, R12, R3-R0)
  - Allows handler to be written entirely in ‘C’

External Interrupts

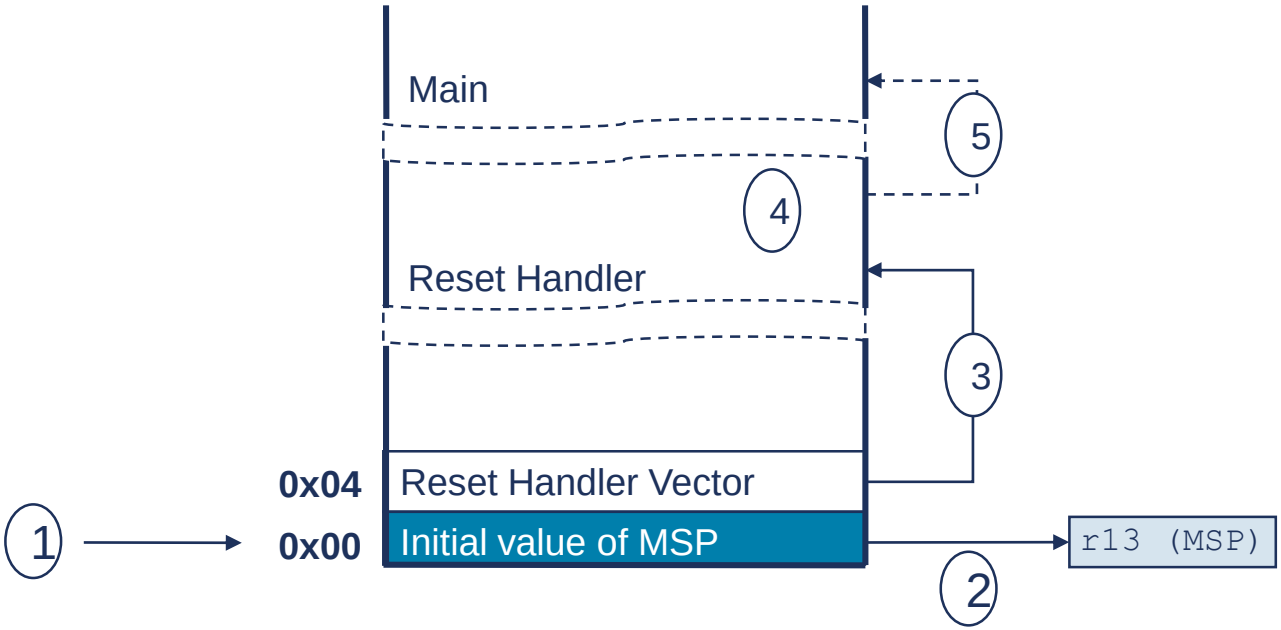
- **External Interrupts handled by Nested Vectored Interrupt Controller (NVIC)**
  - Tightly coupled with processor core
- **One Non-Maskable Interrupt (NMI) supported**
- **Number of external interrupts is implementation-defined**
  - ARMv7-M supports up to 496 interrupts



Exception Handling Example

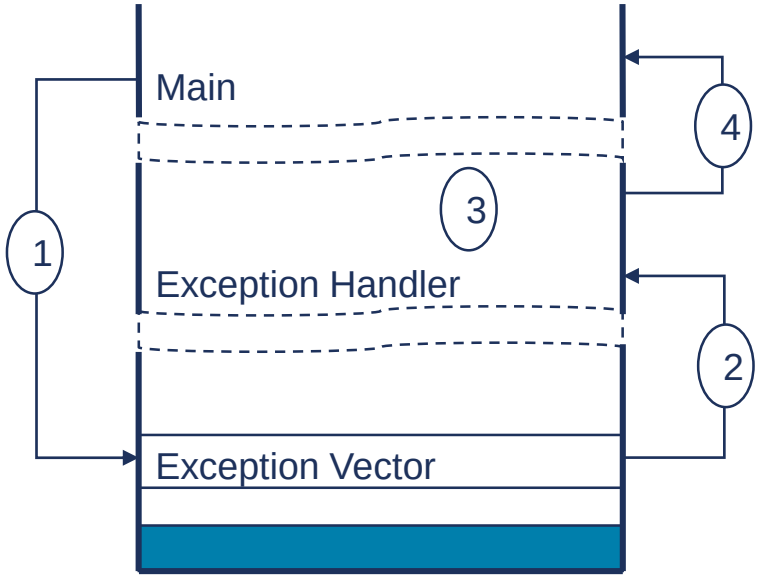


Reset Behavior



1. A reset occurs (Reset input was asserted)
2. Load MSP (Main Stack Pointer) register initial value from address 0x00
3. Load reset handler vector address from address 0x04
4. Reset handler executes in Thread Mode
5. Optional: Reset handler branches to the main program

Exception Behaviour



1. **Exception occurs**
   - Current instruction stream stops
   - Processor accesses vector table
2. **Vector address for the exception loaded from the vector table**
3. **Exception handler executes in Handler Mode**
4. **Exception handler returns to main**

Interrupt Service Routine Entry

- **When receiving an interrupt the processor will finish the current instruction for most instructions**
  - To minimize interrupt latency, the processor can take an interrupt during the execution of a multi-cycle instruction - see next slide
- **Processor state automatically saved to the current stack**
  - 8 registers are pushed: PC, R0-R3, R12, LR, xPSR
  - Follows ARM Architecture Procedure Calling Standard (AAPCS)
- **During (or after) state saving the address of the ISR is read from the Vector Table**
- **Link Register is modified for interrupt return**
- **First instruction of ISR executed**
  - For Cortex-M3 or Cortex-M4 the total latency is normally 12 cycles, however, interrupt late-arrival and interrupt tail-chaining can improve IRQ latency
- **ISR executes from Handler mode with Main stack**

Returning From Interrupt

- **Can return from interrupt with the following instructions when the PC is loaded with "magic" value of 0xFFFF_FFFX (same format as EXC_RETURN)**
  - LDR PC, …..
  - LDM/POP which includes loading the PC
  - BX LR (most common)

- **If no interrupts are pending, foreground state is restored**
  - Stack and state specified by EXC_RETURN is used
  - Context restore on Cortex-M3 and Cortex-M4 requires 10 cycles

- **If other interrupts are pending, the highest priority may be serviced**
  - Serviced if interrupt priority is higher than the foreground's base priority
  - Process is called Tail-Chaining as foreground state is not yet restored
  - Latency for servicing new interrupt is only 6 cycles on M3/M4 (state already saved)

- **If state restore is interrupted, it is abandoned**
  - New ISR executed without state saving (original state still intact and valid)
  - Must still fetch new vector and refill pipeline (6-cycle latency on M3/M4)

Vector Table in C

```
typedef void(* const ExecFuncPtr)(void) __irq;

#pragma arm section rodata="exceptions_area"

ExecFuncPtr exception_table[] = {
   (ExecFuncPtr)&Image$$ARM_LIB_STACK$$ZI$$Limit,     /* Initial SP */
   (ExecFuncPtr)__main,                               /* Initial PC */
   NMIException,
   HardFaultException,
   MemManageException,
   BusFaultException,
   UsageFaultException,
   0, 0, 0, 0,                          /* Reserved */
   SVCHandler,
   DebugMonitor,
   0,                                   /* Reserved */
   PendSVC,
   SysTickHandler
   /* Configurable interrupts start here...*/
};
#pragma arm section
```

The vector table at address 0x0 is minimally required to have 4 values: stack top, reset routine location, NMI ISR location, HardFault ISR location

The SVCall ISR location must be populated if the SVC instruction will be used

Once interrupts are enabled, the vector table (whether at 0 or in SRAM) must then have pointers to all enabled (by mask) exceptions

Vector Table in Assembly

```
                PRESERVE8
                THUMB
                IMPORT ||Image$$ARM_LIB_STACK$$ZI$$Limit||
                AREA    RESET, DATA, READONLY
                EXPORT  __Vectors

__Vectors       DCD     ||Image$$ARM_LIB_STACK$$ZI$$Limit||  ; Top of Stack
                DCD     Reset_Handler                ; Reset Handler
                DCD     NMI_Handler                  ; NMI Handler
                DCD     HardFault_Handler            ; Hard Fault Handler
                DCD     MemManage_Handler            ; MemManage Fault Handler
                DCD     BusFault_Handler             ; Bus Fault Handler
                DCD     UsageFault_Handler           ; Usage Fault Handler
                DCD     0, 0, 0, 0,                  ; Reserved x4
                DCD     SVC_Handler,                 ; SVCall Handler
                DCD     Debug_Monitor                ; Debug Monitor Handler
                DCD     0                            ; Reserved
                DCD     PendSV_Handler               ; PendSV Handler
                DCD     SysTick_Handler              ; SysTick Handler
                ; External vectors start here
```