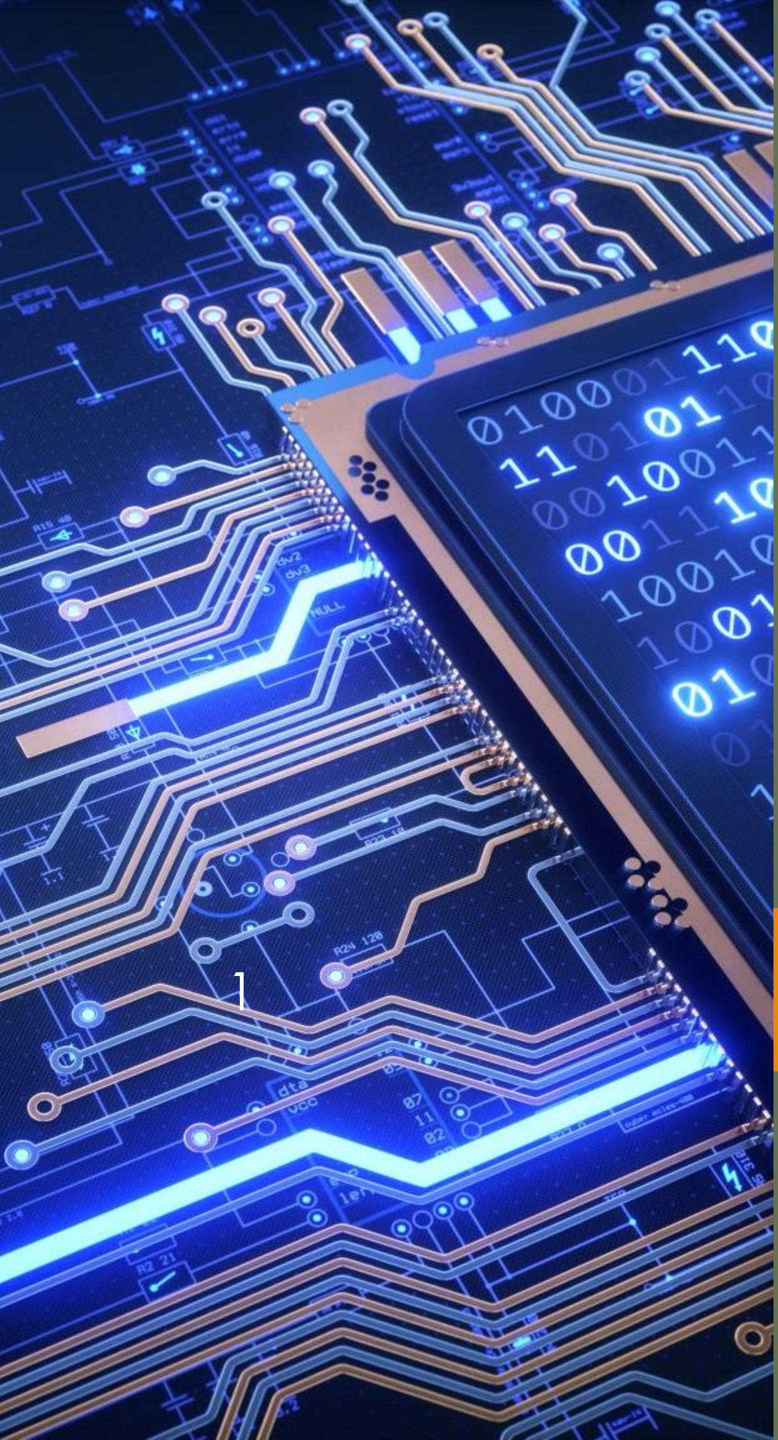


Embedded Controller programming for Real Time Systems: ECE-40097

Lesson 4

Vijay Kumar

Main Topics

- Big and Little Endian
- Load and Store
- Flow Control
- Code examples
- C and Assembly

Big and Little Endian

- Specifies how the bytes are stored in memory
- For little Endian, lsb is at low address
- For Big Endian, lsb is at high address

Little Endian	Byte 3	Byte 2	Byte 1	Byte 0	
Big Endian	Byte 0	Byte 1	Byte 2	Byte 3	
	MSB			LSB	

Endian support in Cortex-M4

- Supports both Endian
  - Little Endian by default
- Instruction to change the endian
- Applicable for load and store

```
SETEND  BE        ; Set big-endian
SETEND  LE        ; Set little-endian
```

Examples

- 0x12345678 stored at address 0x20000000

Address	Data (little endian)	Data (Big endian)
0x20000003	0x12	0x78
0x20000002	0x34	0x56
0x20000001	0x56	0x34
0x20000000	0x78	0x12

Memory Addressing

- There are three addressing modes
  - Pre-index, Pre index with update and Post-index

Index Format	Example	Equivalent
Pre-index	`LDR r1, [r0, #4]`	r1 ← memory[r0 + 4], r0 is unchanged
Pre-index with update	`LDR r1, [r0, #4]!`	r1 ← memory[r0 + 4] r0 ← r0 + 4
Post-index	`LDR r1, [r0], #4`	r1 ← memory[r0] r0 ← r0 + 4

Examples

- STR r1, [r0, #4]! ; Post index with update

Assume: r0 = 0x20008000, r1=0x12345566

After the execution:

r1=0x12345566

r0 = 0x20008004

Memory Address	Memory Data
0x20008007	0x12
0x20008006	0x34
0x20008005	0x55
0x20008004	0x66
0x20008003	0x00
0x20008002	0x00
0x20008001	0x00
0x20008000	0x00

```
LDR     R8, [R10]               ; Loads R8 from the address in R10.
LDRNE   R2, [R5, #960]!         ; Loads (conditionally) R2 from a word
                                ; 960 bytes above the address in R5, and
                                ; increments R5 by 960
STR     R2, [R9,#const-struc]   ; const-struc is an expression evaluating
                                ; to a constant in the range 0-4095.
STRH    R3, [R4], #4            ; Store R3 as halfword data into address in
                                ; R4, then increment R4 by 4
LDRD    R8, R9, [R3, #0x20]     ; Load R8 from a word 32 bytes above the
                                ; address in R3, and load R9 from a word 36
                                ; bytes above the address in R3
STRD    R0, R1, [R8], #-16      ; Store R0 to address in R8, and store R1 to
                                ; a word 4 bytes above the address in R8,
                                ; and then decrement R8 by 16.
```


Memory addressing instructions

Load	Word	LDR Rd, [Rn, <op2>]	2
	To PC	LDR PC, [Rn, <op2>]	2 + P
	Halfword	LDRH Rd, [Rn, <op2>]	2
	Byte	LDRB Rd, [Rn, <op2>]	2
	Signed halfword	LDRSH Rd, [Rn, <op2>]	2
	Signed byte	LDRSB Rd, [Rn, <op2>]	2
	User word	LDRT Rd, [Rn, #<imm>]	2
	User halfword	LDRHT Rd, [Rn, #<imm>]	2
	User byte	LDRBT Rd, [Rn, #<imm>]	2
	User signed halfword	LDRSHT Rd, [Rn, #<imm>]	2
	User signed byte	LDRSBT Rd, [Rn, #<imm>]	2
	PC relative	LDR Rd,[PC, #<imm>]	2
	Doubleword	LDRD Rd, Rd, [Rn, #<imm>]	1 + N
	Multiple	LDM Rn, {<reglist>}	1 + N
	Multiple including PC	LDM Rn, {<reglist>, PC}	1 + N + P

Load instructions

Sign Extension

- LDRSB r1, [r0] ; Load signed Byte

; Assume r0 = 0x024000**04**

; Load a signed byte:

LDRSB r1, [r0]   ; r1 = 0xffffff04

Operation	Description	Assembler	Cycles
Store	Word	STR Rd, [Rn, <op2>]	2
	Halfword	STRH Rd, [Rn, <op2>]	2
	Byte	STRB Rd, [Rn, <op2>]	2
	Signed halfword	STRSH Rd, [Rn, <op2>]	2
	Signed byte	STRSB Rd, [Rn, <op2>]	2
	User word	STRT Rd, [Rn, #<imm>]	2
	User halfword	STRHT Rd, [Rn, #<imm>]	2
	User byte	STRBT Rd, [Rn, #<imm>]	2
	User signed halfword	STRSHT Rd, [Rn, #<imm>]	2
	User signed byte	STRSBT Rd, [Rn, #<imm>]	2
	Doubleword	STRD Rd, Rd, [Rn, #<imm>]	1 + N
	Multiple	STM Rn, {<reglist>}	1 + N

Store instructions

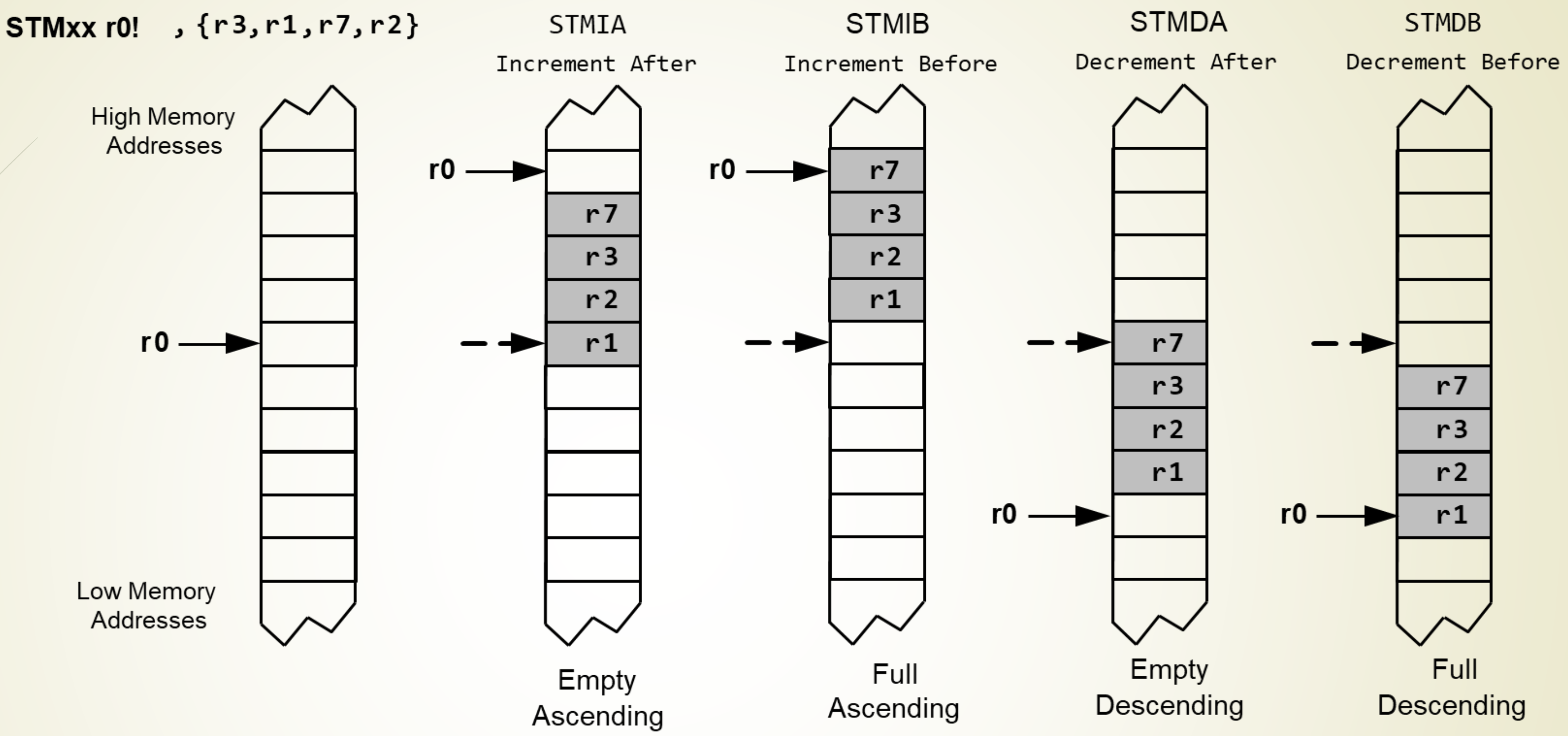
Loading and storing Multiple registers

- Single instructions to store from multiple register
- Order of registers doesn't matter
- Lowest numbered register, always stored at lowest address
- Rn contains the base address

STMxx rn{!}, {register_list}
LDMxx rn{!}, {register_list}

;xx = IA, IB, DA, or DB
;Rn = R0 – R7

Addressing Modes	Description	Instructions
IA	Increment After	STMIA, LDMIA
IB	Increment Before	STMIB, LDMIB
DA	Decrement After	STMDA, LDMDA
DB	Decrement Before	STMDB, LDMDB



STMxx r0! , {r3,r1,r7,r2}
STMIA
Increment After
STMIB
Increment Before
STMDA
Decrement After
STMDB
Decrement Before
High Memory Addresses
Low Memory Addresses
r0
r7
r3
r2
r1
Empty Ascending
Full Ascending
Empty Descending
Full Descending

Examples

Incorrect

LDMIA r3!,{r0,r9} ; high registers not allowed

STMIA r5!, {} ; must be at least one register ; in list

STMIA r5!,{r1-r6} ; value stored from r5 is unpredictable

Branch Instructions

- Used to change the flow of control
  - Conditional
  - Unconditional
- Conditional branch instructions checks for Z, C, N, V flags
- Depends on signed or unsigned

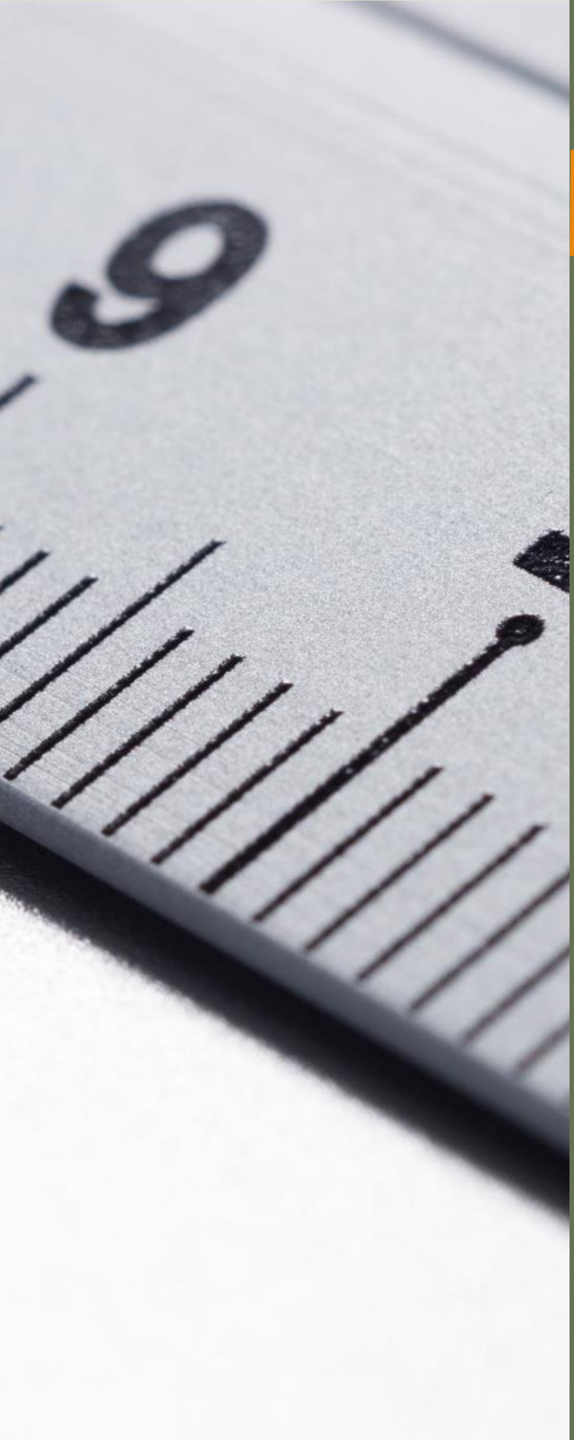
	Instruction	Description	Flags tested
Unconditional Branch	B label	Branch to label	
Conditional Branch	BEQ label	Branch if EQual	Z = 1
	BNE label	Branch if Not Equal	Z = 0
	BCS/BHS label	Branch if unsigned Higher or Same	C = 1
	BCC/BLO label	Branch if unsigned LOwer	C = 0
	BMI label	Branch if MInus (Negative)	N = 1
	BPL label	Branch if PLus (Positive or Zero)	N = 0
	BVS label	Branch if oVerflow Set	V = 1
	BVC label	Branch if oVerflow Clear	V = 0
	BHI label	Branch if unsigned HIgher	C = 1 & Z = 0
	BLS label	Branch if unsigned Lower or Same	C = 0 or Z = 1
	BGE label	Branch if signed Greater or Equal	N = V
	BLT label	Branch if signed Less Than	N != V
	BGT label	Branch if signed Greater Than	Z = 0 & N = V
	BLE label	Branch if signed Less than or Equal	Z = 1 or N = !V

Branch Instructions

Compare Instructions

- Branch instructions follow the compare instructions
- Examples:
  - CMP r1, r2; Compare two numbers
  - BEQ label ; go to label if they are equal

Instruction	Operands	Brief description	Flags
CMP	Rn, Op2	Compare	N,Z,C,V
CMN	Rn, Op2	Compare Negative	N,Z,C,V
TEQ	Rn, Op2	Test Equivalence	N,Z,C
TST	Rn, Op2	Test	N,Z,C



Number Comparision

- Which number is greater
  - 0xffffffff or 0x00000023
- Depends if the number is signed or not
  - If signed, then later is greater
- Developer responsibility to indicate
  - In C, use signed vs unsigned
  - In Assembly, use signed vs unsigned branch instructions
    - BLE or BLS

Combinations Instructions

- Combines two instructions
  - **CMP Rn, #0**
  - **BEQ label** ;
  - **CBZ Rn, label ; combined together**

Instruction	Operands	Brief description	Flags
CBZ	Rn, label	Compare and Branch if Zero	-
CBNZ	Rn, label	Compare and Branch if Non Zero	-

C Program	Assembly Program

`// Find string length`
`char str[] = "hello";`
`int len = 0;`
 	
`for( ; ; ) {`
`    if (*str == '\0')`
`        break;`
`    str++;`
`    len++;`
`}`
	`        ; r0 = string memory address`
	`        ; r1 = string length`
	`        MOV  r1, #0          ; len = 0`
	
	`loop    LDRB r2, [r1]`
	`        CBNZ r2, notZero`
	`        B    endloop`
	`notZero ADD  r0, r0, #1     ; str++`
	`        ADD  r1, r1, #1     ; len++`
	`        B    loop`
	`endloop`

Example

Synchronization primitives

- Provides a non-blocking mechanism that a thread or process can use to obtain exclusive access to a memory location.
- Software can use them to perform a guaranteed read-modify-write memory update sequence, or for a semaphore mechanism
- For reasons of performance, keep the number of instructions between corresponding Load-Exclusive and Store-Exclusive instruction to a minimum
- The pairs of Load-Exclusive and Store-Exclusive instructions are:
  - Word instructions LDREX and STREX
  - Halfword instructions LDREXH and STREXH
  - Byte instructions LDREXB and STREXB

Examples

```
MOV r1, #0x1                ; load the ‘lock taken’ value
try
    LDREX r0, [LockAddr]        ; load the lock value
    CMP r0, #0                  ; is the lock free?
    STREXEQ r0, r1, [LockAddr]  ; try and claim the lock
    CMPEQ r0, #0                ; did this succeed?
    BNE try                     ; no – try again
    ....                        ; yes – we have the lock
```

C equivalent statements

- Branch and conditional statements are used for followings
  - If-then
  - If-then-else
  - While loop
  - For loop
  - Break/Continue

Assembly equivalent

- Using BNE to implement if-then-else

```
r1 = a, r2 = b
       CMP r1, #5    ; compare a and 5
       BNE else      ; go to else if a ≠ 5
then   MOV r2, #3    ; b = 3
       B   endif     ; go to endif
else   MOV r2, #2    ; b = 2
endif
```

Examples

C Program	Assembly Program

`int main(void) {`
`  int result, n;`
	`        AREA factorial, CODE, READONLY`
	`        EXPORT __main`
	`        ENTRY`
	`__main  PROC`

`  result = 1;` `  n = 6;`	`        MOV   r0, #1        ; r0 = result` `        MOV   r1, #6        ; r1 = n`

`  for (int i = 1; i <= n; i++)`
`    result = result * i;`
	`        MOV   r2, #1        ; r2 = i = 1`
	`loop    CMP   r2, r1        ; compare i and n`
	`        BGT   stop          ; if i > n, stop`
	`        MULS  r0, r2, r0    ; result *= i`
	`        ADD   r2, r2, #1    ; i++`
	`        B     loop`

`  while(1);`
`}`
	`stop    B     stop`
	`        ENDP`
	`        END`

Subroutine

- A subroutines, also called a function or a procedure,
  - single-entry, single-exit
  - Return to caller after it exits
- When a subroutine is called, the Link Register (LR) holds the memory address of the next instruction to be executed after the subroutine exits.
- And PC holds the address of the subroutine

Example

- Shows the usage of PC and LR

Main Program	Subroutine
_main …. BL test1 ; PC CMP r0, #1 ; PC +4	Test1 MOV r2, #3; LR = PC+4 ….. BX LR ; PC = LR

What is CMSIS

- Stands for **Cortex Microcontroller software interface standard**
- Enables consistent device support
- Some Instructions can’t be accessed directly
    - CMIS provides function to use those instructions
    - And special registers
- Simplifies development with software interfaces
- Reduces the learning curve
- Reduces time to market for new devices by creating reusable software
- Supported on our development board
- A device-independent interface for RTOS kernels
- Support from many vendors
- Will use some of these functions in our programming assignments

CMSIS Functions

- Access instructions are not accessible directly
  - CMSIS provides the exclusive access

Instruction	CMSIS function
LDREX	uint32_t __LDREXW (uint32_t *addr)
LDREXH	uint16_t __LDREXH (uint16_t *addr)
LDREXB	uint8_t __LDREXB (uint8_t *addr)
STREX	uint32_t __STREXW (uint32_t value, uint32_t *addr)
STREXH	uint32_t __STREXH (uint16_t value, uint16_t *addr)
STREXB	uint32_t __STREXB (uint8_t value, uint8_t *addr)
CLREX	void __CLREX (void)

Instruction	CMSIS intrinsic 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)
ISB	void __ISB(void)
DSB	void __DSB(void)
DMB	void __DMB(void)
REV	uint32_t __REV(uint32_t int value)
REV16	uint32_t __REV16(uint32_t int value)
REVSH	uint32_t __REVSH(uint32_t int value)
RBIT	uint32_t __RBIT(uint32_t int value)
SEV	void __SEV(void)
WFE	void __WFE(void)
WFI	void __WFI(void)

CMSIS Functions

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)
MSP	Read	uint32_t __get_MSP (void)
	Write	void __set_MSP (uint32_t TopOfMainStack)
PSP	Read	uint32_t __get_PSP (void)
	Write	void __set_PSP (uint32_t TopOfProcStack)

CMSIS Functions

C and Assembly

- Leveraging C and Assembly in same project
- Very realistic and suitable for optimal design
- Calling C routines from Assembly and vice – versa
- Embedding Assembly in C
  - Assembly instructions
  - Assembly functions
- Do we need to embed C in Assembly ??

C Program (main.c)	Assembly Program (test.s)

<pre>char str[25] = "Embedded!";
 	
extern void strlength(char* s);
 	
int main(void){
    int i;
    i = strlength(str);
    while(1);
}</pre>
	<pre> AREA stringLength, CODE
	          EXPORT strlength ; make strlength visible
	          ALIGN
	strlength PROC
	          PUSH {r4, lr} ; preserve r4 and lr
	          MOV r4, #0 ; initialize length
	goback LDRB r1, [r0, r4] ; r0 = string address
	          CBZ r1, exit ; branch if zero
	          ADD r4, r4, #1 ; length++
	          B goback ; do it again
	exit MOV r0, r4 ; place result in r0
	          POP {r4, pc} ; exit
	          ENDP</pre>

C and Assembly

Assembly Program	C Program

<pre>   AREA main, CODE
        EXPORT __main
        IMPORT getValue
        IMPORT increment
        IMPORT setValue
        ALIGN
        ENTRY
 	
__main MOVS r2,#0
        MOVS r0,#1
        BL   setValue
        BL   increment
        BL   getValue
        MOV   r2,r0
 	
stop   B   stop
 	
        AREA myData, DATA
        EXPORT **counter**
counter DCD   0
        END</pre>
	<pre>
	
	extern int **counter**;
	
	int getValue() {
	   return counter;
	}
	
	void increment() {
	   counter++;
	}
	
	void setValue(int c) {
	   counter = c;
	}</pre>

From Assembly to C values

Assembly Program (main.s)	C Program (strlen.c)

<pre>   AREA my_strlen, CODE
        EXPORT __main
        IMPORT strlen
        ALIGN
        ENTRY
 	
__main PROC
        LDR   r0, =str
        BL   strlen
stop   B   stop
        ENDP
 	
        AREA myData, DATA
        ALIGN
Str   DCB   "12345678",0
        END</pre>
	<pre>
	
	int strlen(char *s){
	   int i = 0;
	
	   while(s[i] != '\0')
	      i++;
	
	   return i;
	}</pre>

From Assembly to C

Assembly File

- Has extension .s file
- We do have startup_stm32l475xx.s file under startup folder
- You could always add file in IDE by right click on the folder and then select file
    - Enter the file name

Code example – GNU assembler

- File name mySum.s

```
.globl   mySum
     .p2align 2
.type    mySum,%function

mySum:                    // Function "mySum" entry point.
.fnstart

    add r0, r1, r0   // Add r0 and r1 and store the result in r0

    bx lr            // Return from function call
.fnend
```

- Need to define extern in the file you will use.

```
// Extern function from assembly code
extern int mySum(int, int);
```

- https://developer.arm.com/documentation/dui0497/a/the-cortex-m0-instruction-set/branch-and-control-instructions/b--bl--bx--and-blx

Code example – GNU assembler

- Shows how to use branch

```
myFunc:    // increment until 10
.fnstart

      MOVS  r2,#10            // Set R2 to 10
loop:
      CMP   r0,r2           // compare R0 and R2
      BGT   stop             // If greater, jmp to stop

      ADD  r0,r0,#1          // increment R0
      b     loop             //The final result is saved in
      register r0
stop:
       bx lr                 // return
      .fnend
```

- **What value will this return ?**

Assembly in C

```
// Add 31 to a given umber
int Add31(uint32_t i) {
        asm (
        "mov    r1,#31 \n\t"    // start with r1 and store 31
        "add r0, r0, r1\n\t"   // Add  r1 to passed value i
        "MOV r0,r0 \n\t"       // Move the value to R0
        "BX lr \n\t"           // Return from function call
        );
}
```

If passed 4, will return 35

Assembly in C

```
// Increment until 10

int Inc10(uint32_t i) {

        asm (

        "mov    r1,#10   \n\t"    // start with r1 =10

        "loop1:                \n\t"
        "CMP r0,r1            \n\t"   // compare r0, r1
          "bgt stop            \n\t"        //  go to end if greater
        "add r0, r0,#1        \n\t“     // increment r0,
        "b loop1               \n\t“       // go back to loop

        "stop:                      \n\t"
        "MOV r0,r0                  \n\t"
        "BX lr                       \n\t"

        );
}
```

Next Lesson Topic

- What is Interrupt
- Interrupt Request Number
- Interrupt Service Routine
- Interrupt Priority
- Interrupt Flow
- Examples