بِسْمِ ٱللَّهِ ٱلرَّحْمَٰنِ ٱلرَّحِيمِ

CSE211s:

Introduction to Embedded Systems

Functions & Assembler Directives

﴿يَرْفَعِ ٱللَّهُ ٱلَّذِينَ ءَامَنُوا۟ مِنكُمْ وَٱلَّذِينَ أُوتُوا۟ ٱلْعِلْمَ دَرَجَٰتٍ﴾

* Functions:

→ Functions, procedures & subroutines are code sequences that can be called to perform specific task.

→ We define a subroutine by giving it a name in the label field, followed by instructions, which when executed, perform the desired effect.

* In assembly language, we will use the BL instruction to call this subroutine. At run time, the BL instruction will save the return address in the LR register. the return address is the location of the instruction immediately after the BL instruction. At the end of the subroutine, the BX LR instruction will retrieve the return address from the LR register, returning the program to the place from which the subroutine was called. More precisely, it returns to the instruction immediately after the instruction that preformed the subroutine call.

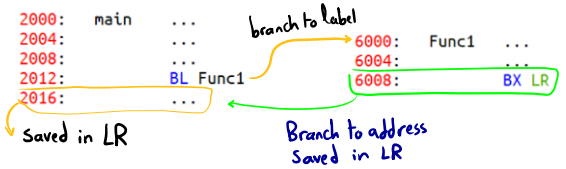
* Branch & link instruction : →BL Label

→ Step① : Save address of next instruction into LR

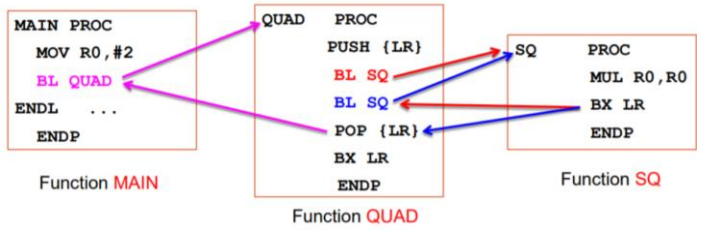
Step② : Branch to the given label

* Branch & exchange instruction : →BX register

→ Instead of providing a label to jump to, the BX instruction provides a register which contains an address to jump to



• in case of nested function : it's essential to save the contents of the link register in other some location before calling another subroutine. So, we push the link register onto the stack at entry then, an additional subroutine calls can safely be made without causing the return address to be lost



• if you do this, you will have two methods to return from a subroutine

① popping the PC off the stack at exit

```
PUSH {LR}
.
.
.
POP {PC}
```

② popping the LR then moving the value into the pc

```
PUSH {LR}
.
.
.
POP {LR}
BX LR
```

Scratch Register : it's a temporary register used to hold an intermediate value during a calculations with a routine

* ARM Procedure Call Standard (APCS) :

→ it defines a set of rules for procedure entry & exit

* the first four registers {R0-R3} : they are used to pass argument values into a subroutine & to return a result value from a function

→ they also can be used as scratch registers

→ they could be changed, caller needs to save them if needed after the call

→ Remaining parameters, if existed, are pushed into stack in the reverse order

* {R4-R11} : they are used to hold the values of a routine's local variable

→ they could be changed, callee needs to save them

* {R12} : it may be used by linker as a scratch register between a routine & any subroutine it calls. it can also be used within a routine to hold intermediate values between subroutine calls

Register	Synonym	Role in Procedure Call Standard
r0-r1	a1-a2	Argument/Result/Scratch Register
r2-r3	a3-a4	Argument/Scratch Register
r4-r8	v1-v5	Variable Register
r9	v6	Variable Register
r10-r11	v7-v8	Variable Register
r12	ip	Intra-Procedure Call Scratch Register
r13	sp	Stack Pointer
r14	lr	Link Register
r15	pc	Program Counter

* Rules for Procedures :

→ Called with a BL instruction, returns with a BX LR

→ Accepts up to 4 arguments in R0,R1,R2 & R3

→ Return value is always in R0 [and if necessary in R1,R2,R3]

* Assembler Directives :

→ We use assembler directives to assist & control the assembly process

→ these directives change the way the code is assembled

- **AREA** → `AREA sectionname {,attr①}{,attr②}`

→ it tells the assembler to define a new section of memory (SRAM or ROM)

* Attributes :

①
- **CODE** → Contains machine instructions / Const. data
- **DATA** → Contains data, no instructions
- **NOINIT** → Indicates the data section is uninitialized or initialized to zero

②
- **READONLY** → Indicates that this area should not be written to & placed in ROM (for code by default)
- **READWRITE** → Indicates that this area may be read from or written to placed in SRAM for variables

* Examples :
```
AREA RESET, DATA, READONLY
AREA myCode, CODE, READONLY
```

- **ENTRY** → indicates to the assembler the beginning of the executable code

- **END** → indicates to the assembler the end of the source file

- **EQU** → defines a constant value or a fixed address.

it doesn't set a side storage for a data item, but associates a constant number with a data or an address label

```
abc EQU 2               ; Assigns the value 2 to the symbol abc.
xyz EQU label+8         ; Assigns the address (label+8) to the symbol xyz.
```

- **SPACE** → it's used for uninitialized data (initialize it with zeroes)

```
data1   SPACE   255     ; defines 255 bytes of zeroed store to data1
```
label — no. of bytes

- **ALIGN** → it's used to ensure your data & code is aligned to appropriate boundaries (word / halfword aligned) → {2}

- **DCD** → Allocate aligned word memory location

```
data1   DCD     1,5,20  ; Defines 3 words containing
                        ; decimal values 1, 5, and 20
```

- **DCW** → Allocate aligned half word memory location

- **DCB** → Allocate aligned byte memory location

* Startup File :

→ it's a piece of code written in assembly or C language that executes before main() function of our embedded application. it performs various initialization steps by setting up the hardware of the micro controller so that the user application can run. therefore, a startup file always runs before the main() code of our embedded application

→ the startup file performs various initializations & contains code for interrupt vector routines

```
        AREA RESET , DATA , READONLY
        EXPORT __Vectors
__Vectors
        DCD 0x20008000 ; initializ sp
        DCD Reset_Handler
        ALIGN
        AREA mycode ,Code , READONLY
        ENTRY
        EXPORT Reset_Handler
Reset_Handler
```

* Test yourself → QUIZ