CEG2400 - Microcomputer Systems

Lecture 3: ARM Assembly Language

Philip Leong

Tools

- Compiler translates a high level language such as C or C++ into assembly language
- Assembler turns assembly language into object code
  - Allows programmer to use mnemonics, symbolic labels, error checking, pseudo-instructions etc
  - Object code not directly executable as it may call other object code in different files and/or libraries
- Linker turns object code into an executable, assigning addresses for jumps, code/data areas etc

Assembly Programming

- The following is a simple example which illustrates some of the core constituents of an ARM assembler module:

```
        AREA Example, CODE, READONLY      ; name this block of code
        ENTRY                             ; mark first instruction
                                          ; to execute
start
        MOV     r0, #15                   ; Set up parameters
        MOV     r1, #20
        BL      firstfunc                 ; Call subroutine
        SWI     0x11                      ; terminate
firstfunc
                                          ; Subroutine firstfunc
        ADD     r0, r0, r1                ; r0 = r0 + r1
        MOV     pc, lr                    ; Return from subroutine
                                          ; with result in r0
        END                               ; mark end of file
```

label, opcode, operands, comment

General Layout of an Assembly Program

- The general form of lines in an assembler module is:

  label <space> opcode <space> operands <space> ; comment

- Each field must be separated by one or more <whitespace> (such as a space or a tab).
- Actual instructions never start in the first column, since they must be preceded by whitespace, even if there is no label.
- All three sections are optional and the assembler will also accept blank lines to improve the clarity of the code.

Description of Module

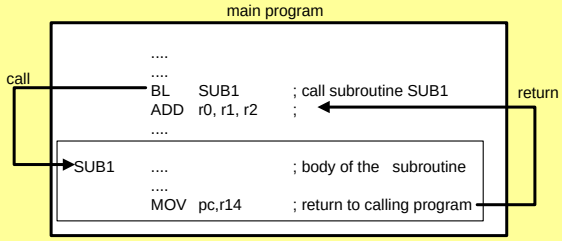
- The main routine of the program (labelled **start**) loads the values 15 and 20 into registers 0 and 1.
- The program then calls the subroutine **firstfunc** by using a branch with link instruction (**BL**).
- The subroutine adds together the two parameters it has received and places the result back into r0.
- It then returns by simply restoring the program counter to the address which was stored in the **link register** (r14) on entry.
- Upon return from the subroutine, the main program simply terminates using software interrupt (**SWI**) 11. This instructs the program to exit cleanly and return control to the debugger.

AREA, ENTRY & END Assembly Directives

- Directives are instructions to the assembler program, NOT to the microprocessors
- AREA Directive - specifies chunks of data or code that are manipulated by the linker.
  - A complete application will consist of one or more areas. The example above consists of a single area which contains code and is marked as being read-only. A single CODE area is the minimum required to produce an application.
- ENTRY Directive - marks the first instruction to be executed within an application
  - An application can contain only a single entry point and so in a multi-source-module application, only a single module will contain an ENTRY directive.
- END directive - marks the end of the module

Subroutines

- Subroutines allow you to modularize your code so that they are more reusable.
- The general structure of a subroutine in a program is:



Subroutine (con't)

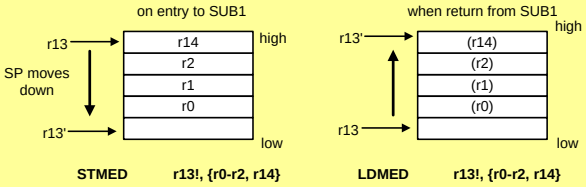
- **`BL subroutine_name`** (Branch-and-Link) is the instruction to jump to subroutine. It performs the following operations:
  - 1) It saves the **PC** value (which points to the next instruction) in r14. This is has the return address.
  - 2) It loads PC with the address of the subroutine. This effective performs a branch.
- BL always uses r14 to store the return address. r14 is called the **link register.**
- Return from subroutine is simple: - just put r14 back into PC (r15).

Nested Subroutines

- Since the return address is held in register r14, you should not call a further subroutine without first saving r14.
- It is also a good software engineering practice that a subroutine does not change any register values except when passing results back to the calling program.
- This is the principle of information hiding: try to hide what the subroutine does from the calling program.
- How do you achieve these two goals? Use a stack to:
  - Preserve r14
  - Save, then retrieve, the values of registers used inside subroutine

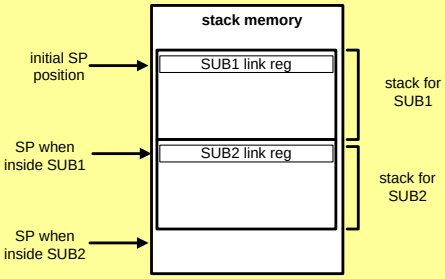
Preserve things inside subroutine with STACK

```
        BL      SUB1
        .....
SUB1    STMED   r13!, {r0-r2, r14}      ; push work & link registers
        ....
        BL      SUB2                    ; jump to a nested subroutine
        ...
        LDMED   r13!, {r0-r2, r14}      ; pop work & link registers
        MOV     pc, r14                 ; return to calling program
```



Effect of subroutine nesting

- SUB1 calls another subroutine SUB2. Assuming that SUB2 also saves its link register (r14) and its working registers on the stack, a snap-shot of the stack will look like:-



HexOut - example of a well-written subroutine

```
; Subroutine HexOut - Output 32-bit word as 8 hex digits as ASCII characters
; Input parameters:       r1 contains the 32-bit word to output
; Return parameters:      none
HexOut  STMED   r13!, {r0-r2, r14}  ; save working registers on stack
        MOV     r2, #8              ; r2 has nibble (4-bit digit) count = 8
Loop    MOV     r0, r1, LSR #28     ; get top nibble by shifting right 28 bits
        CMP     r0, #9              ; if nibble <= 9, then
        ADDLE   r0, r0, #"0"                ;     convert to ASCII numeric char
        ADDGT   r0, r0, #"A"-10     ; else  convert to ASCII alphabet char
        BL      WriteC              ; print character
        MOV     r1, r1, LSL #4      ; shift left 4 bits to get to next nibble
        SUBS    r2, r2, #1          ; decrement nibble count
        BNE     Loop                ; if more, do next nibble
        LDMED   r13!, {r0-r2, pc}   ; retrieve working registers from stack
                                    ;   ... and return to calling program
        END
```