Lecture 5 Assembly Language Programming Basics

- 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
```


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

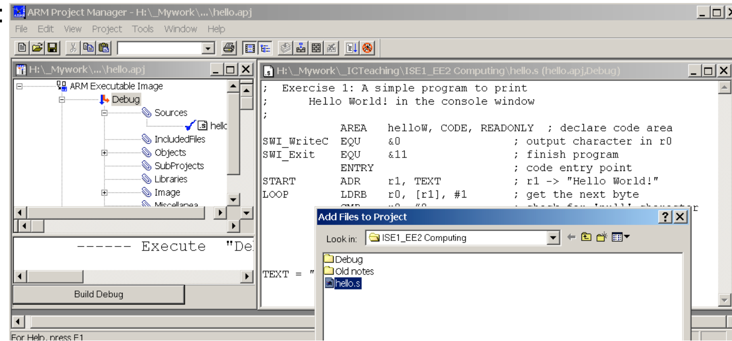
- The main routine of the program (labelled **start**) loads the values 15 and 20 into registers 0 and 1.
- The program then calls the subroutine **firstfunc** by using a branch with link instruction (**BL**).
- The subroutine adds together the two parameters it has received and places the result back 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

Creating program and project file

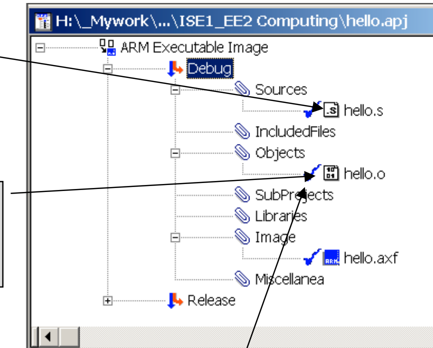
- Invoke ARM_SDK program and enter the program as hello.s in the directory **`H:\arm_work\hello.s\`**
- Use pulldown command >**`Project`** >**`New`** to create a new project called **`hello.apj`**
- Add all the files belonging to this project as shown (only one file here):



How to interpret the Project File (.apj)?

This project has only one module: example1.s

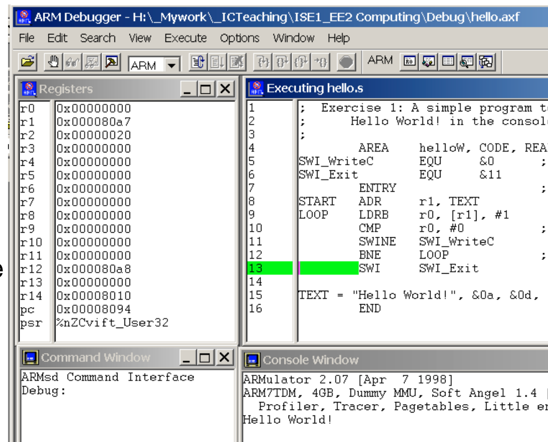
The assembler produces an object file (.o) for the linker to use.



Linker collects together all the assembled and compiled modules to form the .axf file for debugging.

Build and Run the program

- After building the project, you can invoke the debugger program and step through each instructions one at a time.
- The debugger program allows you to control the execution while viewing any register and memory location.
- You can also set breakpoints in the program.



Data Processing Instructions

- Three types of instructions:
  - Data Processing
  - Data Movement
  - Control Flow
- Rules apply to ARM data processing instructions:
  - All operands are 32 bits, come either from registers or are specified as constants (called literals) in the instruction itself
  - The result is also 32 bits and is placed in a register
  - 3 operands - 2 for inputs and 1 for result
- Example:

  ADD r0, r1, r2 ; r0 := r1 + r2
- Works for both unsigned and 2's complement signed
- This may produce carry out signal and overflow bits, but ignored by default
- Result register can be the same with input operand register

Data Processing Instructions - Arithmetic operations

- Here are ARM's arithmetic operations:

```
ADD   r0, r1, r2      ; r0 := r1 + r2
ADC   r0, r1, r2      ; r0 := r1 + r2 + C
SUB   r0, r1, r2      ; r0 := r1 - r2
SBC   r0, r1, r2      ; r0 := r1 - r2 + C - 1
RSB   r0, r1, r2      ; r0 := r2 - r1
RSC   r0, r1, r2      ; r0 := r2 - r1 + C - 1
```

- RSB stands for reverse subtraction
- Operands may be unsigned or 2's complement signed integers
- 'C' is the carry (C) bit in the CPSR - Current Program Status Reg

Data Processing Instructions - Logical operations

- Here are ARM's bit-wise logical operations:

```
AND   r0, r1, r2      ; r0 := r1 and r2 (bit-by-bit for 32 bits)
ORR   r0, r1, r2      ; r0 := r1 or r2
EOR   r0, r1, r2      ; r0 := r1 xor r2
BIC   r0, r1, r2      ; r0 := r1 and not r2
```

- BIC stands for 'bit clear', where every '1' in the second operand clears the corresponding bit in the first:

```
r1:   0101 0011 1010 1111 1101 1010 0110 1011
r2:   1111 1111 1111 1111 0000 0000 0000 0000
r0:   0000 0000 0000 0000 1101 1010 0110 1011
```


Data Processing Instructions - Register Moves

- Here are ARM's register move operations:

```
MOV   r0, r2      ; r0 := r2
MVN   r0, r2      ; r0 := not r2
```

- MVN stands for 'move negated'

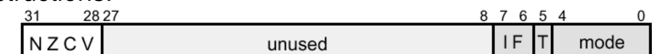
```
r2:   0101 0011 1010 1111 1101 1010 0110 1011
r0:   1010 1100 0101 0000 0010 0101 1001 0100
```


Data Processing Instructions - Comparison Operations

- Here are ARM's register comparison operations:

```
CMP   r1, r2      ; set cc on r1 - r2
CMN   r1, r2      ; set cc on r1 + r2
TST   r1, r2      ; set cc on r1 and r2
TEQ   r1, r2      ; set cc on r1 xor r2
```

- Results of subtract, add, and, xor are NOT stored in any registers
- Only the condition code bits (cc) in the CPSR are set or cleared by these instructions:



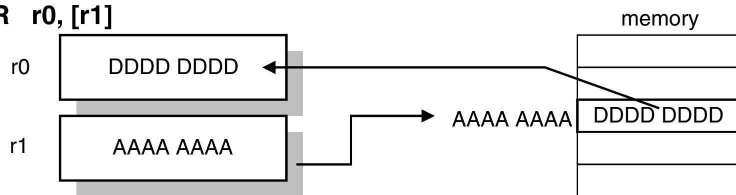
- Take CMP r1,r2 instruction:
  - N = 1 if MSB of (r1 - r2) is '1'
  - Z = 1 if (r1 - r2) = 0
  - C = 1 if (r1, r2) are both unsigned integers AND (r1 < r2)
  - V = 1 if (r1, r2) are signed integers AND (r1 < r2)

Data Transfer Instructions - single register load/store instructions

- Three basic forms of data transfer instructions:
  - Single register load/store instructions
  - Multiple register load/store instructions
  - Single register swap instructions
- Use a value in one register (called the **base** register) as a memory **address** and either loads the **data** value from that address into a destination register or stores the register value to memory:

  LDR   r0, [r1]   ; r0 := mem_{32}[r1]

  STR   r0, [r1]   ; mem_{32}[r1] := r0
- This is called **register-indirect addressing**
- **LDR r0, [r1]**

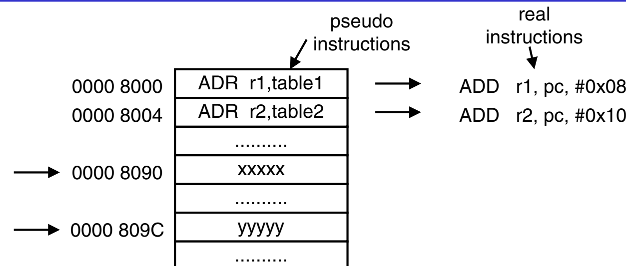


Data Transfer Instructions - Set up the address pointer

- Need to initialize address in r1 in the first place. How?
- Use ADR pseudo instruction - looks like normal instruction, but it does not really exists. Instead the assembler translates it to one or more real instructions.
- The following example copies data from TABLE 1 to TABLE2

```
copy      ADR   r1, TABLE1     ; r1 points to TABLE1
          ADR   r2, TABLE2     ; r2 points to TABLE2
          LDR   r0, [r1]       ; load first value ....
          STR   r0, [r2]       ;   and store it in TABLE2
          .......
TABLE1    ......               ; <source of data>
          ......
TABLE2    ......               ; <destination of data>
```


Data Transfer Instructions - ADR instruction



- How does **ADR** instruction works? Address is 32-bit, difficult to put a 32-bit address value in a register in the first place
- Solution: Program Counter PC (**r15**) is often close to the desired data address value
- **ADR r1, TABLE1** is translated into an instruction that add or subtract a constant to PC (**r15**), and put the results in r1
- This constant is known as **PC-relative offset**, and it is calculated as: addr_of_table1 - (PC_value + 8)

Data Transfer Instructions - Base plus offset addressing

- Extend the copy program further to copy NEXT word:

```
copy      ADR   r1, TABLE1     ; r1 points to TABLE1
          ADR   r2, TABLE2     ; r2 points to TABLE2
          LDR   r0, [r1]       ; load first value ....
          STR   r0, [r2]       ;   and store it in TABLE2
          ADD   r1, r1, #4     ; step r1 onto next word
          ADD   r2, r2, #4     ; step r2 onto next word
          LDR   r0, [r1]       ; load second value ...
          STR   r0, [r2]       ;   and store it
          ......
```

- Simplify with **pre-indexed addressing mode**

  LDR   r0, [r1, #4]   ; r0 := mem_{32} [r1 + 4]

Data Transfer Instructions - pre-indexed with auto-indexing

- A simplified version is:

```
copy    ADR   r1, TABLE1      ; r1 points to TABLE1
        ADR   r2, TABLE2      ; r2 points to TABLE2
        LDR   r0, [r1]        ; load first value ….
        STR   r0, [r2]        ;   and store it in TABLE2
        LDR   r0, [r1, #4]    ; load second value …
        STR   r0, [r2, #4]    ;   and store it
        ……
```

- Pre-indexed addressing does not change r1. Sometimes, it is useful to modify the base register to point to the new address. This is achieve by adding a '!', and is pre-indexed addressing with **auto-indexing**:

```
LDR   r0, [r1, #4]!    ; r0 : = mem32 [r1 + 4]
                       ; r1 := r1 + 4
```

- The '**!**' indicates that the instruction should update the base register after the data transfer

Data Transfer Instructions - post-indexed addressing

- Another useful form of the instruction is:

```
LDR   r0, [r1], #4    ; r0 : = mem32 [r1]
                      ; r1 := r1 + 4
```

- This is called: post-indexed addressing - the base address is used without an offset as the transfer address, after which it is auto-indexed.
- Using this, we can improve the copy program more:

```
copy    ADR   r1, TABLE1      ; r1 points to TABLE1
        ADR   r2, TABLE2      ; r2 points to TABLE2
loop    LDR   r0, [r1], #4    ; get TABLE1 1st word ….
        STR   r0, [r2], #4    ;   copy it to TABLE2
        ???                   ; if more, go back to loop
        ……
TABLE1  ……                    ; < source of data >
```


Data Transfer Instructions Summary

- Size of data can be reduced to 8-bit byte with:

```
LDRB   r0, [r1]        ; r0 : = mem8 [r1]
```

- Summary of addressing modes:

```
LDR   r0, [r1]               ; register-indirect addressing
LDR   r0, [r1 , # offset]    ; pre-indexed addressing
LDR   r0, [r1 , # offset]!   ; pre-indexed, auto-indexing
LDR   r0, [r1], # offset     ; post-indexed, auto-indexing
ADR   r0, address_label      ; PC relative addressing
```