Chapters 3
ARM Assembly

Embedded Systems with ARM Cortex-M

Updated: Wednesday, February 7, 2018
Programming languages - Categories

- **Interpreted based on the machine**
  - High Level Language
  - Interpreter
    - Lower Level Language
    - Assembly Language
    - Binary Language

- **Compiler**
  - High Level Language
  - Lower Level Language

- **Assembler**
  - Computer Instruction
  - Binary 0's and 1's

Less complex, not as efficient

Efficient, more complex
Programming languages - Interpreter Vs. Compiler

<table>
<thead>
<tr>
<th>Interpreter</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translates program one statement at a time.</td>
<td>Scans the entire program and translates it as a whole into machine code.</td>
</tr>
<tr>
<td>It takes less amount of time to analyze the source code but the overall execution time is slower.</td>
<td>It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.</td>
</tr>
<tr>
<td>No intermediate object code is generated, hence are memory efficient.</td>
<td>Generates intermediate object code which further requires linking, hence requires more memory.</td>
</tr>
<tr>
<td>Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.</td>
<td>It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.</td>
</tr>
</tbody>
</table>

Assembler converts instructions into Machine Language 1s and 0s.
C Compiler

C Compiler

Preprocessor

Compiler

Assembly Source File

.C

C Header File

.H
Assemblers and C Compilers
Assemblers and C Compilers
ARM Assembly

- Modern ARM processors have several instruction sets:
  - The fully-featured 32-bit ARM instruction set,
  - The more restricted, but space efficient, 16-bit Thumb instruction set,
  - The newer mixed 16/32-bit Thumb-2 instruction set,
  - Jazelle DBX for Java byte codes,
  - The NEON 64/128-bit SIMD instruction set,
  - The VFP vector floating point instruction set.
- → Thumb-2 is the progression of Thumb (strictly it is Thumb v3). It improves performance whilst keeping the code density tight by allowing a mixture of 16- and 32-bit instructions.

http://www.davespace.co.uk/arm/introduction-to-arm/instruction-sets.html
Levels of Program Code
C Code → Assembly → Machine Language

- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability

- **Assembly language**
  - Textual representation of instructions

- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data

C Program
```c
int main(void)
{
    int i;
    int total = 0;
    for (i = 0; i < 10; i++)
    {
        total += i;
    }
    while(1); // Dead loop
}
```

Assembly Program
```assembly
MOVS r1, #0
MOVS r0, #0
B check
loop ADD r1, r1, r0
    ADDS r0, r0, #1
    CMP r0, #10
    BLT loop
    B self
```

Machine Program
```
0010001000000000
0010000000000000
1100000000000001
0100010000000001
0001100001000000
0001100000100010
1101111011111111
1011111100000000
1110011111111110
```
Levels of Program Code
C Code ➔ Assembly ➔ Machine Language

Machine Program

Hardware representation
- Binary digits (bits)
- Encoded instructions and data
Assembly Instruction Sets for Cortex-M

Examples:
- ADD
- AND
- CMP
- SUB
- MUL
- MOV
- etc.
Assembly Instructions Supported

• Arithmetic and logic
  • Add, Subtract, Multiply, Divide, Shift, Rotate

• Data movement
  • Load, Store, Move

• Compare and branch
  • Compare, Test, If-then, Branch, compare and branch on zero

• Miscellaneous
  • Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier
ARM Instruction Format

```
label    mnemonic operand1, operand2, operand3  ; comments
```

- Label is a reference to the **memory address** of this instruction.
- **Mnemonic** represents the operation to be performed (ADD, SUB, etc.).
- The number of **operands** varies, depending on each specific instruction. Some instructions have no operands at all.
  - Typically, operand1 is the **destination** register, and operand2 and operand3 are source operands.
  - operand2 is usually a register.
  - operand3 may be a register, an immediate number, a register shifted to a constant amount of bits, or a register plus an offset (used for memory access).
- Everything after the semicolon “;” is a comment, which is an annotation explicitly declaring programmers’ intentions or assumptions.
ARM Instruction Format

\[
\text{label} \quad \text{mnemonic operand1, operand2, operand3} \quad ; \text{comments}
\]

\[
\text{target} \quad \text{ADD} \quad r0, \quad r2, \quad r3 \quad ; \quad r0 = r2 + r3
\]
ARM Instruction Format

```
label         mnemonic operand1, operand2, operand3 ; comments
```

Examples: Variants of the ADD instruction

```
ADD r1, r2, r3 ; r1 = r2 + r3
ADD r1, r3     ; r1 = r1 + r3
ADD r1, r2, #4 ; r1 = r2 + 4
ADD r1, #15    ; r1 = r1 + 15
```

Remember:
R has two components:
• Register Address
• Register Content
First Assembly

```assembly
AREA string_copy, CODE, READONLY
EXPORT __main
ALIGN ENTRY PROC

__main

strcpy LDR r1, =srcStr ; Retrieve address of the source string
LDR r0, =dstStr ; Retrieve address of the destination string
loop LDRB r2, [r1], #1 ; Load a byte & increase src address pointer
STRB r2, [r0], #1 ; Store a byte & increase dst address pointer
CMP r2, #0 ; Check for the null terminator
BNE loop ; Copy the next byte if string is not ended
stop B _stop ; Dead loop. Embedded program never exits.

ENDP

AREA myData, DATA, READWRITE
ALIGN

srcStr DB "The source string: ", 0 ; Strings are null terminated
dstStr DB "The destination string: ", 0 ; dstStr has more space than srcStr

END
```
Assembly Directives

- Directives are **NOT** instruction; allocate space and define types in many cases. They are used to provide key information for assembly.

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA</td>
<td>Make a new block of data or code</td>
</tr>
<tr>
<td>ENTRY</td>
<td>Declare an entry point where the program execution starts</td>
</tr>
<tr>
<td>ALIGN</td>
<td>Align data or code to a particular memory boundary</td>
</tr>
<tr>
<td>DCB</td>
<td>Allocate one or more bytes (8 bits) of data</td>
</tr>
<tr>
<td>DCW</td>
<td>Allocate one or more half-words (16 bits) of data</td>
</tr>
<tr>
<td>DCD</td>
<td>Allocate one or more words (32 bits) of data</td>
</tr>
<tr>
<td>SPACE</td>
<td>Allocate a zeroed block of memory with a particular size</td>
</tr>
<tr>
<td>FILL</td>
<td>Allocate a block of memory and fill with a given value.</td>
</tr>
<tr>
<td>EQU</td>
<td>Give a symbol name to a numeric constant</td>
</tr>
<tr>
<td>RN</td>
<td>Give a symbol name to a register</td>
</tr>
<tr>
<td>EXPORT</td>
<td>Declare a symbol and make it referable by other source files</td>
</tr>
<tr>
<td>IMPORT</td>
<td>Provide a symbol defined outside the current source file</td>
</tr>
<tr>
<td>INCLUDE/GET</td>
<td>Include a separate source file within the current source file</td>
</tr>
<tr>
<td>PROC</td>
<td>Declare the start of a procedure</td>
</tr>
<tr>
<td>ENDP</td>
<td>Designate the end of a procedure</td>
</tr>
<tr>
<td>END</td>
<td>Designate the end of a source file</td>
</tr>
</tbody>
</table>

- **INCLUDE constants.s** ; Load Constant Definitions
- Start of new data set; At least one code area is required per program; could be READONLY or READWRITE
- Typically $2^N$
- For Example **INCLUDE constants.s** ; Load Constant Definitions
- Refers to PROCEDURE and End of PROCEDURE; it defines the function; similar to main() in C
- END of source file
Let’s Practice First ....
Assembly Emulator

- Go to https://salmanarif.bitbucket.io/visual/index.html
- Download the appropriate version for your computer
- This is the list of supported instructions: https://salmanarif.bitbucket.io/visual/supported_instructions.html
Run your Assembly Code

Move ONE instruction at a time

What is your final PC value?
Try This & Answer the Questions.....

; The purpose of this program is to add: aa+bb=cc
; First we define the variables; The variables are stored in locations 0x100–0x108 in the MEMORY

; load (32 bit) value 0x00000001 into memory location 0x100 (by default) call it aa
aa DCD 0x00112233

; load (32 bit) value 0x00000002 into memory location 0x104 (by default)
bb DCD 2

; load (32 bit) value 0x00000000 into memory location 0x108 (by default)
cc DCD 0

main
LDR r1, =aa ; load memory address of aa into r1 After execution: PC = 0xC+4
LDR r2, [r1] ; load content of memory address in r1 into r2 (that is 0x0001)
LDR r3, =bb
LDR r4, [r3]
ADD r5, r2, r4 ; r2+r4 -> r5 and update flags
LDR r6, =cc ; load memory address of cc into r6
STR r5, [r6] ; store the sum into variable cc

; this section shows that the sum is in fact loaded into cc
LDR r7, =cc
LDR r8, [r7]

; let’s check cc+1---→ r9
ADD r9, r8, #1

END
Let’s Continue with Some Simple Commands....
Let’s Take a MOV & MVN Commands

• MOV r0, #42
  • Move the constant 42 into register R0.

• MOV r2, r3
  • Move the contents of register R3 into register R2.

• MVN r1, r0
  • R1 = NOT(R0) = -43

• MOV r0, r0
  • A NOP (no operation) instruction.

• <operation>
  • MOV – move
    • Rd := Operand2
  • MVN – move NOT
    • Rd := 0xFFFFFFFF EOR Operand2
Arithmetic Operation

- `<operation>`
- **ADD** – Add
  - Rd := Rn + Operand2
- **ADC** – Add with Carry
  - Rd := Rn + Operand2 + Carry
- **SUB** – Subtract
  - Rd := Rn – Operand2
- **SBC** – Subtract with Carry
  - Rd := Rn – Operand2 – NOT(Carry)
- **RSB** – Reverse Subtract
  - Rd := Operand2 – Rn
- **RSC** – Reverse Subtract with Carry
  - Rd := Operand2 – Rn – NOT(Carry)

- **ADD r0, r1, r2**
  - R0 = R1 + R2
- **SUB r5, r3, #10**
  - R5 = R3 – 10
- **RSB r2, r5, #0xFF00**
  - R2 = 0xFF00 – R5

RSB and RSC subtract in reverse order (e.g. y - x not x - y).
Logical Instructions

• **AND** – *logical AND*
  • Rd := Rn AND Operand2

• **EOR** – *Exclusive OR*
  • Rd := Rn EOR Operand2

• **ORR** – *logical OR*
  • Rd := Rn OR Operand2

• **BIC** – *Bitwise Clear*
  • Rd := Rn AND NOT Operand2

• **AND r8, r7, r2**
  • R8 = R7 & R2

• **ORR r11, r11, #1**
  • R11 |= 1

• **BIC r11, r11, #1**
  • R11 &= ~1

• **EOR r11, r11, #1**
  • R11 ^= 1
Compare Instructions:
<operation>{cond} Rn,Operand2

- <operation>
- CMP – compare
  - Flags set to result of (Rn – Operand2).
- CMN – compare negative
  - Flags set to result of (Rn + Operand2).
- TST – bitwise test
  - Flags set to result of (Rn AND Operand2).
- TEQ – test equivalence
  - Flags set to result of (Rn EOR Operand2).
- CMP r0, #42
  - Compare R0 to 42.
- CMN r2, #42
  - Compare R2 to -42.
- TST r11, #1
  - Test bit zero.
- TEQ r8, r9
  - Test R8 equals R9.
- SUBS r1, r0, #42
  - Compare R0 to 42, with result.

- CMP is like SUB.
- CMN is like ADD – subtract of a negative number is the same as add.
- TST is like AND.
- TEQ is like EOR – exclusive or of identical numbers gives result of zero.
Directive: AREA

<table>
<thead>
<tr>
<th>AREA myData, DATA, READWRITE ; Define a data section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array DCD 1, 2, 3, 4, 5 ; Define an array with five integers</td>
</tr>
<tr>
<td>AREA myCode, CODE, READONLY ; Define a code section</td>
</tr>
<tr>
<td>EXPORT __main ; Make __main visible to the linker</td>
</tr>
<tr>
<td>ENTRY ; Mark the entrance to the entire program</td>
</tr>
<tr>
<td>__main PROC ; PROC marks the begin of a subroutine</td>
</tr>
<tr>
<td>... ; Assembly program starts here.</td>
</tr>
<tr>
<td>ENDP ; Mark the end of a subroutine</td>
</tr>
<tr>
<td>END ; Mark the end of a program</td>
</tr>
</tbody>
</table>

- The AREA directive indicates to the assembler the start of a new data or code section.
- Areas are the basic independent and indivisible unit processed by the linker.
- Each area is identified by a name and areas within the same source file cannot share the same name.
- An assembly program must have at least one code area.
- By default, a code area can only be read (READONLY) and a data area may be read from and written to (READWRITE).
### Directive: ENTRY

<table>
<thead>
<tr>
<th>Area</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA myData, DATA, READWRITE</td>
<td>Define a data section</td>
</tr>
<tr>
<td>DCD 1, 2, 3, 4, 5</td>
<td>Define an array with five integers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Area</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA myCode, CODE, READONLY</td>
<td>Define a code section</td>
</tr>
<tr>
<td>EXPORT __main</td>
<td>Make __main visible to the linker</td>
</tr>
<tr>
<td>ENTRY</td>
<td>Mark the entrance to the entire program</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Area</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>__main PROC</td>
<td>PROC marks the begin of a subroutine</td>
</tr>
<tr>
<td>...</td>
<td>Assembly program starts here.</td>
</tr>
<tr>
<td>ENDP</td>
<td>Mark the end of a subroutine</td>
</tr>
<tr>
<td>END</td>
<td>Mark the end of a program</td>
</tr>
</tbody>
</table>

- The ENTRY directive marks **the first instruction to be executed** within an application program.
- There must be **exactly one** ENTRY directive in an application, no matter how many source files the application has.
Directive: END

| AREA myData, DATA, READWRITE ; Define a data section |
| Array |
| DCD 1, 2, 3, 4, 5 ; Define an array with five integers |
| AREA myCode, CODE, READONLY ; Define a code section |
| EXPORT __main ; Make __main visible to the linker |
| ENTRY ; Mark the entrance to the entire program |
| __main PROC |
| ... ; Assembly program starts here. |
| ENDP ; Mark the end of a subroutine |
| END ; Mark the end of a program |

- The END directive indicates the end of a source file.
- Each assembly program must end with this directive.
## Directive: PROC and ENDP

<table>
<thead>
<tr>
<th>AREA myData, DATA, READWRITE ; Define a data section</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCD 1, 2, 3, 4, 5 ; Define an array with five integers</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AREA myCode, CODE, READONLY ; Define a code section</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPORT __main ; Make __main visible to the linker</td>
</tr>
<tr>
<td>ENTRY ; Mark the entrance to the entire program</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>__main PROC</th>
</tr>
</thead>
<tbody>
<tr>
<td>... ; Assembly program starts here.</td>
</tr>
<tr>
<td>ENDP ; Mark the end of a subroutine</td>
</tr>
<tr>
<td>END ; Mark the end of a program</td>
</tr>
</tbody>
</table>

- PROC and ENDP are to mark the start and end of a function (also called subroutine or procedure).
- A single source file can contain multiple subroutines, with each of them defined by a pair of PROC and ENDP.
- PROC and ENDP cannot be nested. We cannot define a function within another function.
Directive: EXPORT and IMPORT

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA myData, DATA, READWRITE ; Define a data section</td>
<td></td>
</tr>
<tr>
<td>Array</td>
<td>DCD 1, 2, 3, 4, 5; Define an array with five integers</td>
</tr>
<tr>
<td>AREA myCode, CODE, READONLY ; Define a code section</td>
<td></td>
</tr>
<tr>
<td>EXPORT __main ; Make __main visible to the linker</td>
<td></td>
</tr>
<tr>
<td>ENTRY</td>
<td>; Mark the entrance to the entire program</td>
</tr>
<tr>
<td>__main</td>
<td>PROC ; PROC marks the begin of a subroutine</td>
</tr>
<tr>
<td>...</td>
<td>; Assembly program starts here.</td>
</tr>
<tr>
<td>ENDP</td>
<td>; Mark the end of a subroutine</td>
</tr>
<tr>
<td>END</td>
<td>; Mark the end of a program</td>
</tr>
</tbody>
</table>

- The EXPORT declares a symbol and makes this **symbol visible** to the linker.
- The IMPORT gives the assembler a symbol that is **not defined locally** in the current assembly file. The symbol must be defined in another file.
- The IMPORT is similar to the “extern” keyword in C.
## Directive: Data Allocation

<table>
<thead>
<tr>
<th>Directive</th>
<th>Description</th>
<th>Memory Space</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DCB</strong></td>
<td>Define Constant Byte</td>
<td>Reserve 8-bit values</td>
</tr>
<tr>
<td><strong>DCW</strong></td>
<td>Define Constant Half-word</td>
<td>Reserve 16-bit values</td>
</tr>
<tr>
<td><strong>DCD</strong></td>
<td>Define Constant Word</td>
<td>Reserve 32-bit values</td>
</tr>
<tr>
<td><strong>DCQ</strong></td>
<td>Define Constant</td>
<td>Reserve 64-bit values</td>
</tr>
<tr>
<td><strong>DCFS</strong></td>
<td>Define single-precision floating-point numbers</td>
<td>Reserve 32-bit values</td>
</tr>
<tr>
<td><strong>DCFD</strong></td>
<td>Define double-precision floating-point numbers</td>
<td>Reserve 64-bit values</td>
</tr>
<tr>
<td><strong>SPACE</strong></td>
<td>Defined Zeroed Bytes</td>
<td>Reserve a number of zeroed bytes</td>
</tr>
<tr>
<td><strong>FILL</strong></td>
<td>Defined Initialized Bytes</td>
<td>Reserve and fill each byte with a value</td>
</tr>
</tbody>
</table>
### Directive: Data Allocation

<table>
<thead>
<tr>
<th>AREA</th>
<th>myData, DATA, READWRITE</th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>DCB &quot;Hello World!&quot;,0</td>
</tr>
</tbody>
</table>

; Allocate a string that is null-terminated

<table>
<thead>
<tr>
<th>dollar</th>
<th>DCB 2,10,0,200</th>
</tr>
</thead>
</table>

; Allocate integers ranging from -128 to 255

<table>
<thead>
<tr>
<th>scores</th>
<th>DCD 2,3.5,-0.8,4.0</th>
</tr>
</thead>
</table>

; Allocate 4 words containing decimal values

<table>
<thead>
<tr>
<th>miles</th>
<th>DCW 100,200,50,0</th>
</tr>
</thead>
</table>

; Allocate integers between -32768 and 65535

<table>
<thead>
<tr>
<th>Pi</th>
<th>DCFS 3.14</th>
</tr>
</thead>
</table>

; Allocate a single-precision floating number

<table>
<thead>
<tr>
<th>Pi</th>
<th>DCFD 3.14</th>
</tr>
</thead>
</table>

; Allocate a double-precision floating number

<table>
<thead>
<tr>
<th>p</th>
<th>SPACE 255</th>
</tr>
</thead>
</table>

; Allocate 255 bytes of zeroed memory space

<table>
<thead>
<tr>
<th>f</th>
<th>FILL 20,0xFF,1</th>
</tr>
</thead>
</table>

; Allocate 20 bytes and set each byte to 0xFF

<table>
<thead>
<tr>
<th>binary</th>
<th>DCB 2_01010101</th>
</tr>
</thead>
</table>

; Allocate a byte in binary

<table>
<thead>
<tr>
<th>octal</th>
<th>DCB 8_73</th>
</tr>
</thead>
</table>

; Allocate a byte in octal

<table>
<thead>
<tr>
<th>char</th>
<th>DCB ‘A’</th>
</tr>
</thead>
</table>

; Allocate a byte initialized to ASCII of ‘A’
Directive: EQU and RN

<table>
<thead>
<tr>
<th>; Interrupt Number Definition (IRQn)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BusFault_IRQn  EQU  -11            ; Cortex-M3 Bus Fault Interrupt</td>
</tr>
<tr>
<td>SVCall_IRQn    EQU  -5             ; Cortex-M3 SV Call Interrupt</td>
</tr>
<tr>
<td>PendSV_IRQn    EQU  -2             ; Cortex-M3 Pend SV Interrupt</td>
</tr>
<tr>
<td>SysTick_IRQn   EQU  -1             ; Cortex-M3 System Tick Interrupt</td>
</tr>
</tbody>
</table>

Dividend  RN  6                     ; Defines dividend for register 6
Divisor   RN  5                      ; Defines divisor for register 5

- The EQU directive associates a symbolic name to a numeric constant. Similar to the use of #define in a C program, the EQU can be used to define a constant in an assembly code.
- The RN directive gives a symbolic name to a specific register.
**Directive: ALIGN**

<table>
<thead>
<tr>
<th>Area</th>
<th>Code</th>
<th>Align</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AREA example, CODE</td>
<td>ALIGN = 3</td>
<td>; Memory address begins at a multiple of 8</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>r0, r1, r2</td>
<td>; Instructions start at a multiple of 8</td>
<td></td>
</tr>
<tr>
<td>AREA myData, DATA</td>
<td>ALIGN = 2</td>
<td>; Address starts at a multiple of four</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>DCB 0xFF</td>
<td>; The first byte of a 4-byte word</td>
<td></td>
</tr>
<tr>
<td>ALIGN 4, 3</td>
<td>; Align to the last byte (3) of a word (4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>DCB 0x33</td>
<td>; Set the fourth byte of a 4-byte word</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>DCB 0x44</td>
<td>; Add a byte to make next data misaligned</td>
<td></td>
</tr>
<tr>
<td>ALIGN</td>
<td>; Force the next data to be aligned</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>DCD 12345</td>
<td>; Skip three bytes and store the word</td>
<td></td>
</tr>
</tbody>
</table>
Directive: INCLUDE or GET

- The INCLUDE or GET directive is to include an assembly source file within another source file.
- It is useful to include constant symbols defined by using EQU and stored in a separate source file.

```
INCLUDE constants.s ; Load Constant Definitions
AREA main, CODE, READONLY
EXPORT __main
ENTRY
__main PROC
...
ENDP
END
```