ARM Assembly Programming

Based on ARM Assembly Language and Architecture

By

Mohammad Maizidi and others

Chapter 2

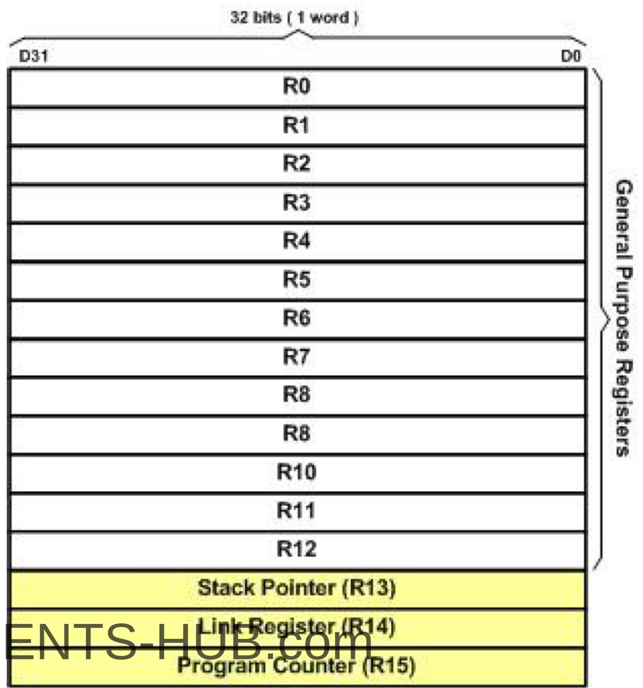
ARM Architecture and Assembly Language Programming

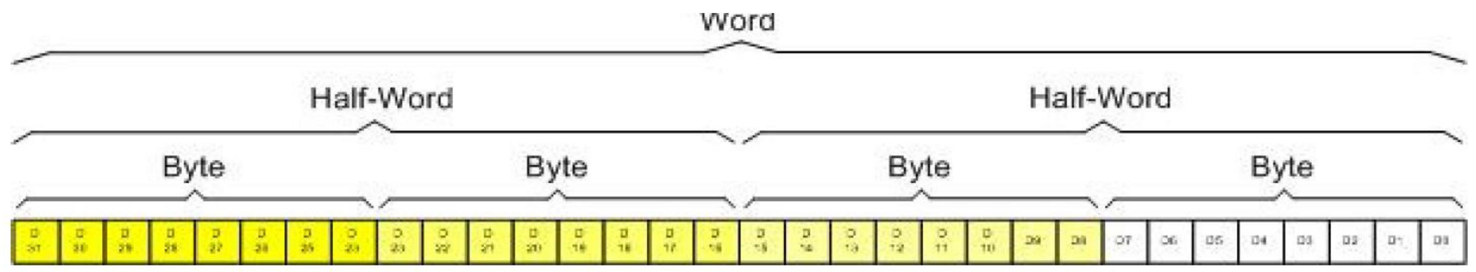
- 15 registers: R0 – R15
- R0 – R12: General Purpose Registers, R13 (SP), R14 (LR), R15 (PC)
- Register width is 32-bit (default size)
- Word: 32 bits, Half-word: 16 bits, Byte: 8 bits

RISC machine is a load-store machine

Note: Memory is byte addressable
ARM is little Indian

two instructions can access memory: load + store





ARM Instruction Format

- 3-address Instructions: instruction destination,source1,source2

source2 can be a register, immediate (constant) value, or memory.

dest + source1 can only be registers

source2 can be constant >> 8bits (signed/unsigned)

MOV instruction: MOV Rn,Op2 ;load Rn register with Op2 (Operand2).

;Op2 can be immediate (constant) number #K which is an 8-bit value that can be 0–255 in decimal, (00–FF in hex)

Op2 can also be a register Rm. Rn or Rm are any of the registers R0 to R15

Examples:

```
MOV R2,#0x25 ;load R2 with 0x25 (R2 = 0x25)
MOV R1,#0x87 ;copy 0x87 into R1 (R1 = 0x87)
MOV R5,R7 ;copy contents of R7 into R5 (R5 = R7)
```

To write a comment in Assembly language we use ‘;’.

Immediate constant notes:

1. We put # in front of every immediate value.

2. If we want to present a number in hex, we put a 0x in front of it. If we put nothing in front of a number, it is in decimal. For example, in "MOV R1,#50", R1 is loaded with 50 in decimal, whereas in "MOV R1,#0x50", R1 is loaded with 50 in hex (80 in decimal).

3. If values 0 to FF are moved into a 32-bit register, the rest of the bits are assumed to be all zeros. For example, in "MOV R1,#0x5" the result will be R1=0x00000005; that is, R1=00000000000000000000000000000101 in binary.

4. Moving an immediate value larger than 255 (FF in hex) into the register will cause an error.

ARM is not case sensitive

mov R5,#2_1110

Note!

We cannot load values larger than 0xFF (255) into registers R0 to R12 using the MOV instruction. For example, the following instruction is not valid:

```
MOV R5,#0x999999                          ;invalid instruction
```

The reason is the fact that although the ARM instruction is 32-bit wide, only 8 bits of MOV instruction can be used as an immediate value which can take values not larger than 0xFF (255).

ADD instruction:

ADD Rd,Rn,Op2 ;ADD Rn to Op2 and store the result in Rd
;Op2 can be Immediate value #K (K is between 0 and 255) ;or Register Rm

```
MOV     R1,#0x25 ;copy 0x25 into R1 (R1 = 0x25)
MOV     R7,#0x34 ;copy 0x34 into R1 (R7 = 0x34)
ADD      R5,R1,R7 ;add value R7 to R1 and put it in R5
   ;(R5 = R1 + R7)
     or
MOV     R1,#0x25            ;load (copy) 0x25 into R1 (R1 = 0x25)
ADD      R5,R1,#0x34        ;add 0x34 to R1 and put it in R5
                           ;(R5 = R1 + 0x34)
```

R5 = 0x59 (0x25 + 0x34 = 0x59)

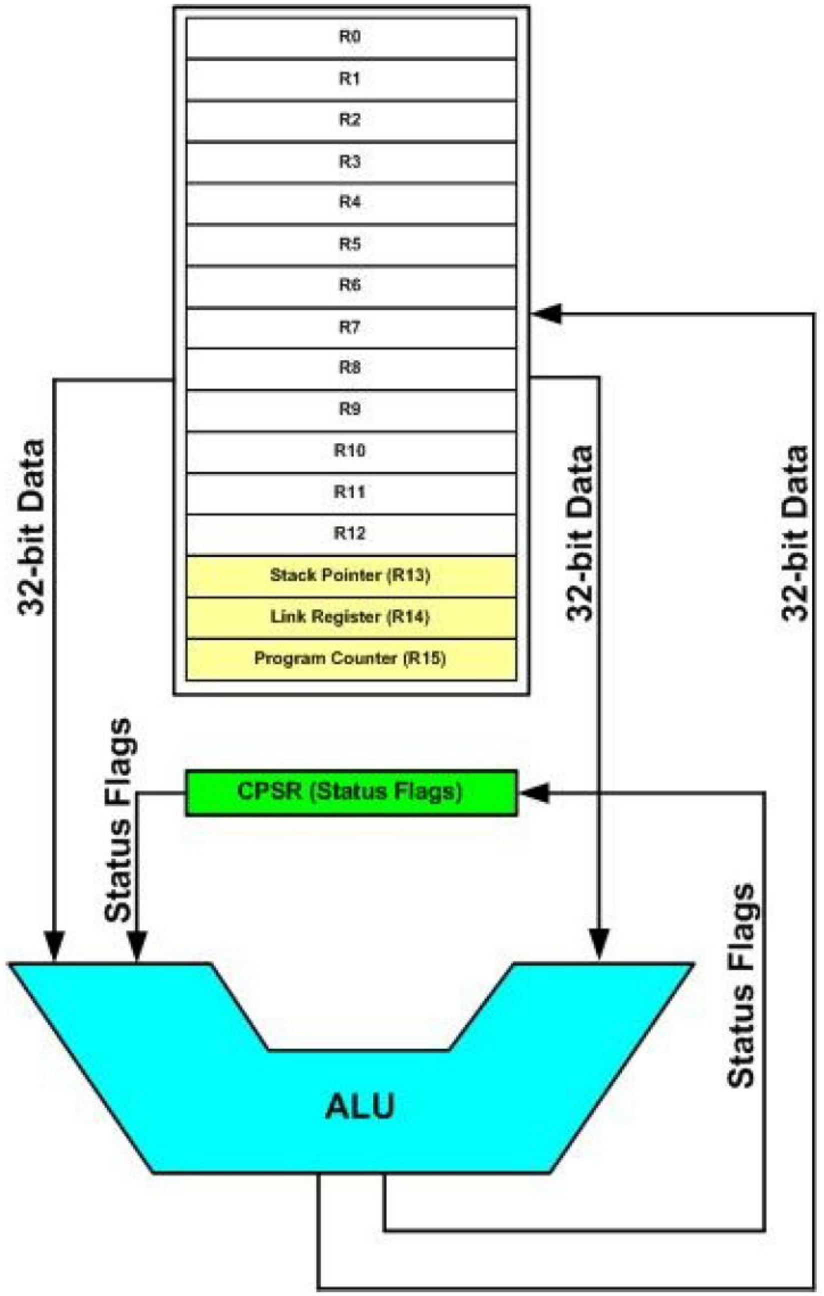
SUB instruction

```
SUB      Rd,Rn,Op2         ;Rd=Rn – Op2

MOV     R1,#0x34           ;load (copy) 0x34 into R1 (R1=0x34)
SUB      R5,R1,#0x25       ;R5 = R1 – 0x25 (R1 = 0x34 – 0x25)
```

The old format

```
SUB      R1,R1,#0x25      ;R1=R1-0x25
SUB      R1,#0x25          ;R1=R1-0x25


SUB      R1,R1,R2          ;R1=R1-R2
SUB      R1,R2              ;R1=R1-R2
```

Notice that in most of instructions like ADD and SUB, Rn can be omitted if Rd and Rn are the same.
This format is no longer recommended by Unified Assembler Language.



CPSR: Current Program Status Register

ADC: more than 32bit,we can use it. R1:R0 + R3:R2
ADD R4,R2,R0
ADC R5,R3,R1

```
MOV R0, #0xFF      ; Load R0 with 0xFF (binary: 11111111)
MOV R1, #0x0F      ; Load R1 with 0x0F (binary: 00001111)
BIC R2, R0, R1     ; Clear bits in R0 where R1 is 1, store result in R2
; After execution:
; R2 = 0xF0 (binary: 11110000)
```

CMP/CMN:
It updates the condition flags (N, Z, C, V) in the CPSR based on the result but does not store the result.

```
MOV R0, #5         ; Load R0 with 5
MOV R1, #10        ; Load R1 with 10
CMP R0, R1         ; Compare R0 with R1
BGT label_gt       ; Branch to label_gt if R0 > R1 (greater than)
```

Instruction	Description
ADD Rd, Rn,Op2*	ADD Rn to Op2 and place the result in Rd
ADC Rd, Rn,Op2	ADD Rn to Op2 with Carry and place the result in Rd
AND Rd, Rn,Op2	AND Rn with Op2 and place the result in Rd
BIC Rd, Rn,Op2	AND Rn with NOT of Op2 and place the result in Rd
CMP Rn,Op2	Compare Rn with Op2 and set the status bits of CPSR**
CMN Rn,Op2	Compare Rn with negative of Op2 and set the status bits
EOR Rd, Rn,Op2	Exclusive OR Rn with Op2 and place the result in Rd
MVN Rd,Op2	Place NOT of Op2 in Rd
MOV Rd,Op2	MOVE (Copy) Op2 to Rd
ORR Rd, Rn,Op2	OR Rn with Op2 and place the result in Rd
RSB Rd, Rn,Op2	Subtract Rn from Op2 and place the result in Rd
RSC Rd, Rn,Op2	Subtract Rn from Op2 with carry and place the result in Rd
SBC Rd, Rn,Op2	Subtract Op2 from Rn with carry and place the result in Rd
SUB Rd, Rn,Op2	Subtract Op2 from Rn and place the result in Rd
TEQ Rn,Op2	Exclusive-OR Rn with Op2 and set the status bits of CPSR
TST Rn,Op2	AND Rn with Op2 and set the status bits of CPSR

* *Op2 can be an immediate 8-bit value #K which can be 0–255 in decimal, (00–FF in hex). Op2 can also be a register Rm. Rd, Rn and Rm are any of the general purpose registers*

** *CPSR is discussed later in this chapter*

*** *The instructions are discussed in detail in the next chapters*

Table 2- 1: ALU Instructions Using GPRs

```
MOV R0, #5         ; Load R0 with 5
MOV R1, #10        ; Load R1 with 10
CLC                ; Clear the carry flag (C = 0)
RSC R2, R0, R1     ; Perform R2 = R1 - R0 - (1 - C)
; Carry = 0 → Borrow = 1
; R2 = R1 - R0 - 1
; R2 = 10 - 5 - 1 = 4
```

```
MOV R0, #0xFF      ; Load R0 with 0xFF (binary: 11111111)
MOV R1, #0x0F      ; Load R1 with 0x0F (binary: 00001111)
TEQ R0, R1         ; Perform bitwise XOR: R0 XOR R1
BEQ equal_label    ; Branch if result is zero (Z = 1)
BNE not_equal_label; Branch if result is not zero (Z = 0)
```

```
MOV R0, #0xF0      ; Load R0 with 0xF0 (binary: 11110000)
MOV R1, #0x80      ; Load R1 with 0x80 (binary: 10000000)
TST R0, R1         ; Test if the most significant bit (MSB) in R0 is set
BEQ bit_not_set    ; Branch if result is zero (Z = 1, bit not set)
BNE bit_set        ; Branch if result is non-zero (Z = 0, bit is set)
```

The ARM Memory Map

The Special Function Registers in ARM

The R13 is set aside for stack pointer.

The R14 is designated as link register which holds the return address when the CPU calls a subroutine

the R15 is the program counter (PC).

The CPSR (current program status register) is used for keeping condition flags among other things,

(the Thumb) have only R0-7 but every variation of ARM chip has R13-R15 SFRs.

special function register (SFR)

The Thumb instruction format is designed to compete with the 8- and 16-bit microcontrollers and increase code density.

a 32-bit program counter can access a maximum of 4G (2^{32} = 4G) bytes of program memory locations.

In ARM microcontrollers each memory location is a byte wide. 0x00000000–0xFFFFFFFF address range.

Memory: 4GB (Byte addressable) for both on-chip and off-chip (RAM, Flash, etc)

The 4G bytes of memory space can be divided into five sections.
They are as follows

1- **<u>On-chip peripheral and I/O registers:</u>**
This area is dedicated to general purpose I/O (**GPIO**) and special function registers (**SFRs**) of peripherals such as **timers**, **serial communication**, **ADC**, and so on. ARM uses memory-mapped I/O.

2- **On-chip data SRAM:** A RAM space ranging from a few kilobytes to several hundred kilobytes is set aside mainly for data storage (e.g. variables, stack)

Electrically Erasable and Programmable ROM (EEPROM)

3-**On-chip EEPROM:** A block of memory from 1K bytes to several thousand bytes is set aside for EEPROM memory (program code storage, saving critical data). Not all ARM chips have on-chip EEPROM non-volatile

4-**On-chip Flash ROM:** A block of memory from a few kilobytes to several hundred kilobytes is set aside for program space. The program space is used for the program code.

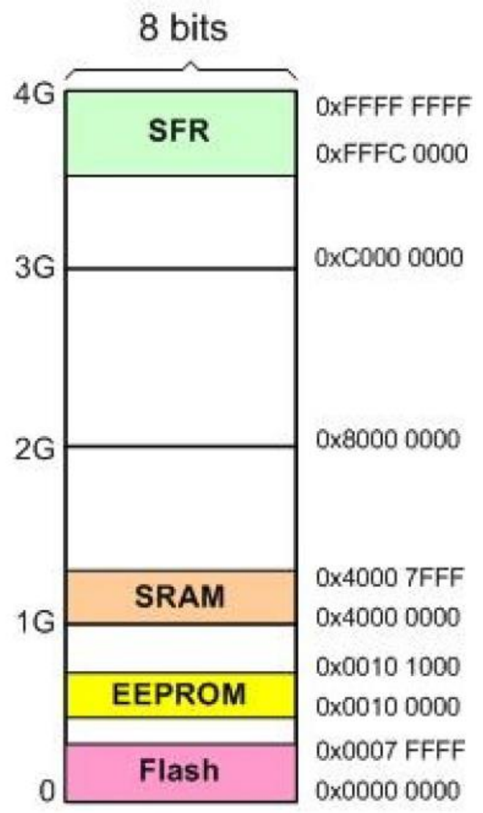
5-**Off-chip DRAM space:** A DRAM memory ranging from few megabytes to several hundred mega bytes can be implemented for external memory connection

Find the address space range of each of the following memory of an ARM chip:

(a) 2 KB of EEPROM starting at address 0x80000000

(b) 16 KB of SRAM starting at address 0x90000000

(c) 64 KB of Flash ROM starting at address 0xF0000000



Load and Store Instructions in ARM

LDR Rd, [Rx] instruction

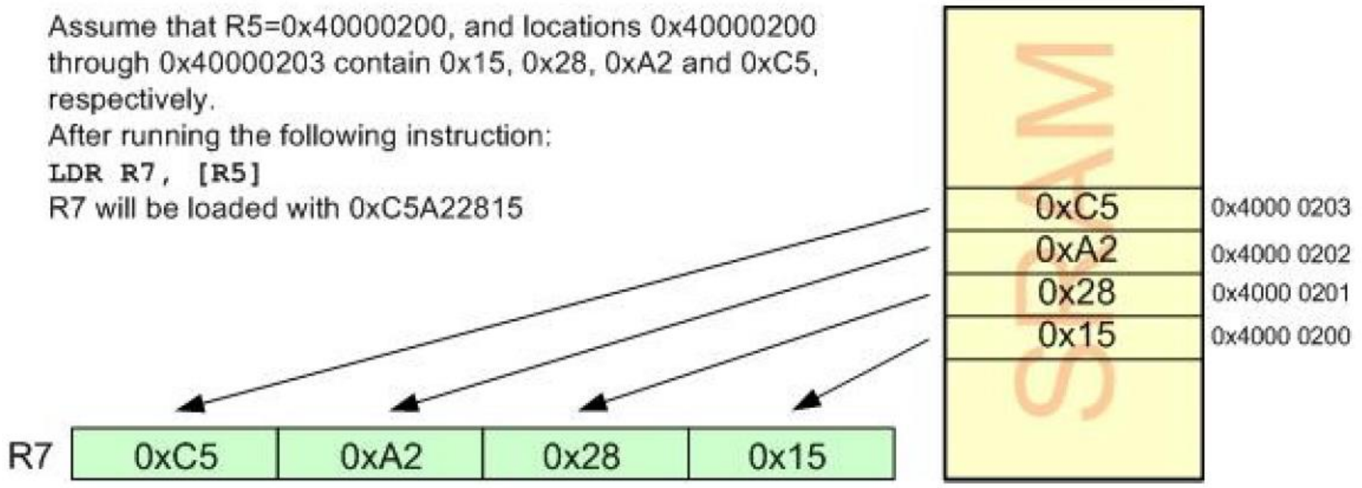
```
LDR   Rd,[Rx] ;load Rd with the contents of location pointed
              ;to by Rx register. Rx is an address between
              ;0x00000000 to 0xFFFFFFFF
```

The LDR instruction tells the CPU to load (bring in) one word (32-bit or 4 bytes)
from a base address pointed to by Rx into the GPR.
After this instruction is executed, the
Rd will have the same value as four consecutive locations
in the memory.

Example:

```
              ;assume R5 = 0x40000200
LDR   R7,[R5] ;load R7 with the contents of locations
              ;0x40000200-0x40000203
```

Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x15, 0x28, 0xA2 and 0xC5, respectively.
After running the following instruction:
`LDR R7, [R5]`
R7 will be loaded with 0xC5A22815



LDRB Rd, [Rx] instruction

LDRSB

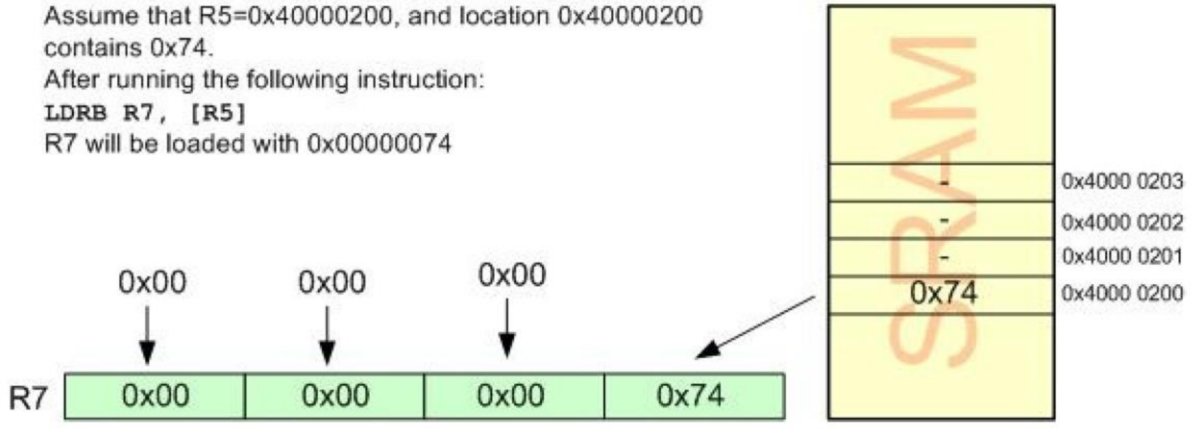
LDRB Rd, [Rx] ;load Rd with the contents of the location

; pointed to by Rx register.

Assume that R5=0x40000200, and location 0x40000200 contains 0x74.
After running the following instruction:

```
LDRB R7, [R5]
```

R7 will be loaded with 0x00000074



Data Size	Bits	Decimal	Hexadecimal	Load instruction used
Byte	8	0 – 255	0 - 0xFF	LDRB
Half-word	16	0 – 65535	0 - 0xFFFF	LDRH
Word	32	0 – 2^{32}-1	0 - 0xFFFFFFFF	LDR

EX.
X DCB 3

LDR R0, [X]
LDR R1, =X
LDR R0, [R1]

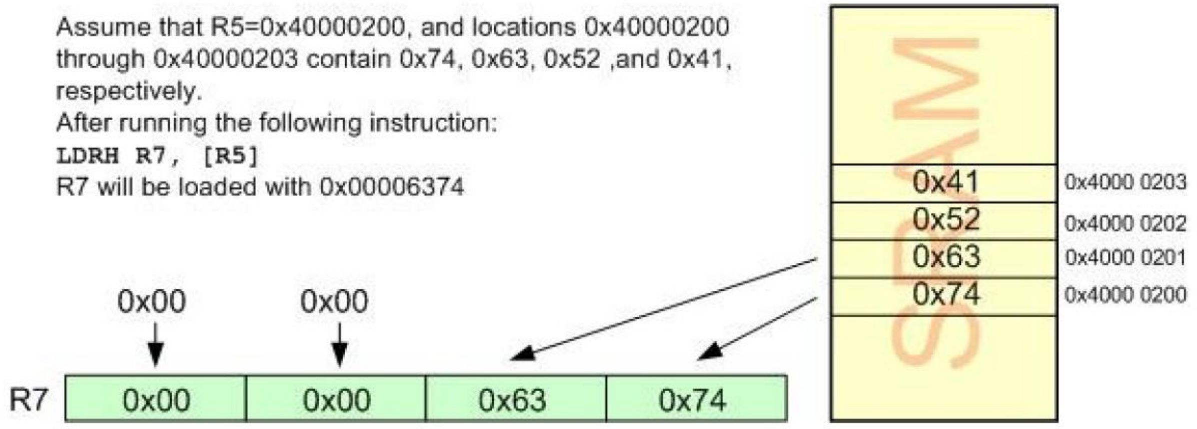
LDRH Rd, [Rx] instruction

LDRH Rd, [Rx] ;load Rd with the half-word pointed
; to by Rx register

Assume that R5=0x40000200, and locations 0x40000200 through 0x40000203 contain 0x74, 0x63, 0x52 ,and 0x41, respectively.
After running the following instruction:

```
LDRH R7, [R5]
```

R7 will be loaded with 0x00006374



Example 2-3, p.52

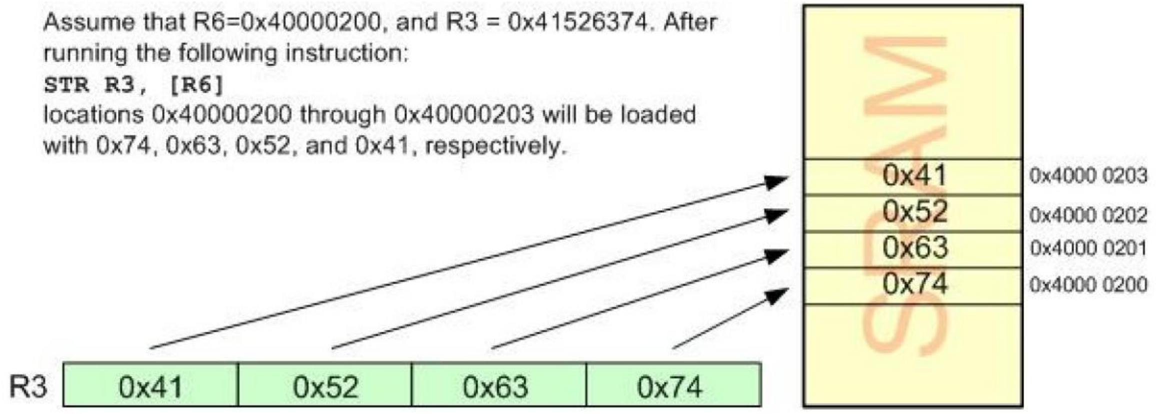
STR Rx,[Rd] instruction

STR Rx,[Rd] ;store register Rx into locations pointed to by Rd

Assume that R6=0x40000200, and R3 = 0x41526374. After running the following instruction:

```
STR R3, [R6]
```

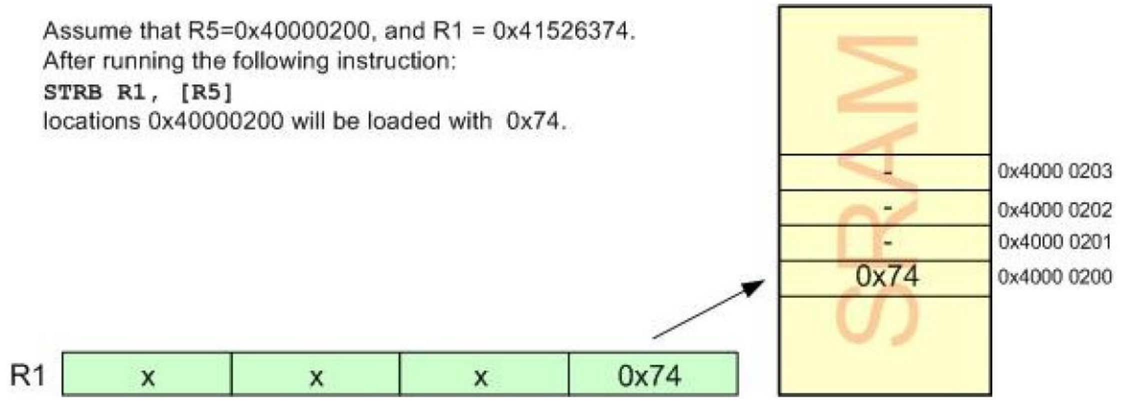
locations 0x40000200 through 0x40000203 will be loaded with 0x74, 0x63, 0x52, and 0x41, respectively.



Assume that R5=0x40000200, and R1 = 0x41526374. After running the following instruction:

```
STRB R1, [R5]
```

locations 0x40000200 will be loaded with 0x74.



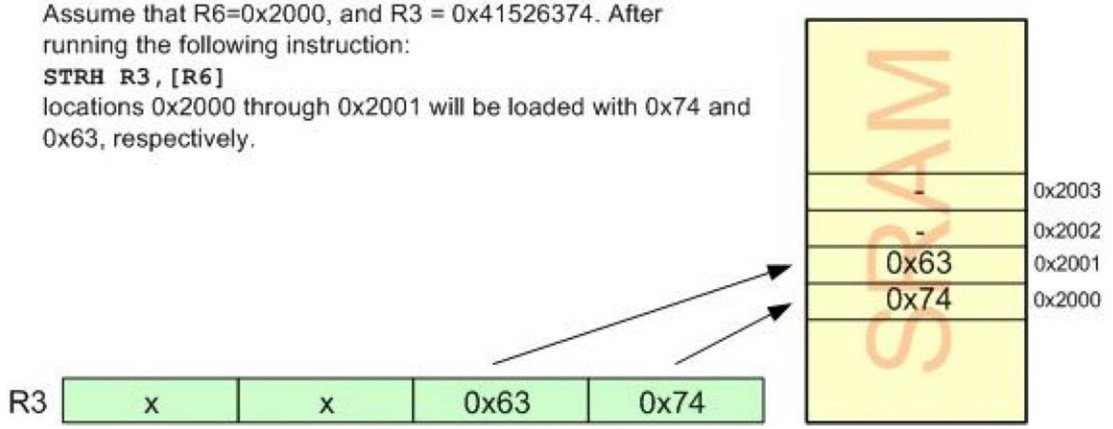
STRB Rx,[Rd] instruction

STRB Rx, [Rd] ;store the byte in register Rx into ;location pointed to by Rd

Assume that R6=0x2000, and R3 = 0x41526374. After running the following instruction:

```
STRH R3,[R6]
```

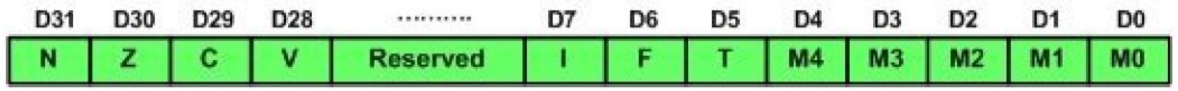
locations 0x2000 through 0x2001 will be loaded with 0x74 and 0x63, respectively.



STRH Rx,[Rd] instruction

STRH Rx, [Rd] ;store half-word (2-byte) in register Rx ;into locations pointed to by Rd

ARM CPSR (Current Program Status Register)



D31	D30	D29	D28	....	D7	D6	D5	D4	D3	D2	D1	D0
N	Z	C	V	Reserved	I	F	T	M4	M3	M2	M1	M0

C, the carry flag
This flag is set whenever there is a carry out from the D31 bit. This flag bit is affected after a 32-bit addition or subtraction.
Z, the zero flag
The zero flag reflects the result of an arithmetic or logic operation. If the result is zero, then Z = 1. Therefore, Z = 0 if the result is not zero.
N, the negative flag
Binary representation of signed numbers uses D31 as the sign bit. The negative flag reflects the result of an arithmetic operation. If the D31 bit of the result is zero, then N = 0 and the result is positive. If the D31 bit is one, then N = 1 and the result is negative. The negative and V flag bits are used for the signed number arithmetic operations and are discussed in Chapter 5.
V, the overflow flag
This flag is set whenever the result of a signed number operation is too large, causing the high-order bit to overflow into the sign bit. In general, the carry flag is used to detect errors in unsigned arithmetic operations while the overflow flag is used to detect errors in signed arithmetic operations. The V and N flag bits are used for signed number arithmetic operations

The T flag bit is used to indicate the ARM is in Thumb state. The I and F flags are
used to enable or disable the interrupt. See the ARM manual

S suffix and the status register

ADDS and SUBS instruction affects flag C, Z, V, and N.

Instruction	Flags Affected
ANDS	C, Z, N
ORRS	C, Z, N
MOVS	C, Z, N
ADDS	C, Z, N, V
SUBS	C, Z, N, V
B	No flags
Note that we cannot put S after B instruction.	

Example 1:
Find C and Z flags after executing the following instruction:

```
;assume R1 = 0x0000009C and R2 = 0xFFFFFF64
ADDS R2,R1,R2 ;add R1 to R2 and place the result in R2
```

Example2:
Show the status of the Z flag during the execution of the following program:

```
MOV R2,#4 ;R2 = 4
MOV R3,#2 ;R3 = 2
MOV R4,#4 ;R4 = 4
SUBS R5,R2,R3 ;R5 = R2 - R3 (R5 = 4 - 2 = 2)
SUBS R5,R2,R4 ;R5 = R2 - R4 (R5 = 4 - 4 = 0)
```

Flag bits and decision making

conditional jump (branch) based on the status of the flag bits.

Instruction	Flags Affected
BCS	Branch if C = 1
BCC	Branch if C = 0
BEQ	Branch if Z = 1
BNE	Branch if Z = 0
BMI	Branch if N = 1
BPL	Branch if N = 0
BVS	Branch if V = 1
BVC	Branch if V = 0

ARM Data Format and Directives

ARM has four data types. They are bit, byte (8-bit), half-word (16-bit) and word (32bit).

Hex numbers

To represent Hex numbers in an ARM assembler we put 0x (or 0X) in front of the number like this:

```
MOV R1,#0x99
```

Decimal numbers

To indicate decimal numbers in some ARM assemblers such as Keil we simply use the decimal (e.g., 12) and nothing before or after it. Here are some examples of how to use it:

```
MOV R7,#12 ;R7 = 00001100 or 0C in hex
MOV R1,#32 ;R1 = 32 = 0x20
```

Binary numbers

To represent binary numbers in an ARM assembler we put 2_ in front of the number. It is as follows:

```
MOV R6,#2_10011001 ;R6 = 10011001 in binary or 99 in hex
```

ASCII characters

To represent ASCII data in an ARM assembler we use single quotes as follows:

```
LDR R3,#‘2’ ;R3 = 00110010 or 32 in hex (See Appendix F)
```

Assembler directives

Directive	Description
AREA	Instructs the assembler to assemble a new code or data section
END	Informs the assembler that it has reached the end of a source file.
ENTRY	Declares an entry point to a program.
EQU	Gives a symbolic name to a numeric constant, a register-relative value or a PC-relative value.
INCLUDE	It adds the contents of a file to our program.

NOTE:
MOV and ADD instructions are commands to the CPU, but EQU, END, and ENTRY are directives to the assembler.

```
PI EQU 3.14

MOV R2,#PI
```

```
AREA sectionname, attribute, attribute, …

AREA MY_ASM_PROG1, CODE, READONLY
```

Among widely used attributes are CODE, DATA, READONLY, READWRITE, COMMON, and ALIGN

```
AREA OUR_VARIABLES, DATA, READWRITE

AREA OUR_CONSTS, DATA, READONLY
```

Program 2-1

```
;ARM Assembly Language Program To Add Some Data and Store the SUM in R3.

          AREA   PROG_2_1, CODE, READONLY
   ENTRY
   MOV     R1, #0x25          ;R1 = 0x25
   MOV     R2, #0x34          ;R2 = 0x34
   ADD      R3, R2,R1          ;R3 = R2 + R1
HERE      B           HERE               ;stay here forever
          END
```

LDR

The ARM assembler provide us a pseudo-instruction of “LDR Rd,=32-bit_immidiate_vlaue” to load value greater than 0xFF

the = sign used in the syntax

```
LDR R7,=0x112233

COUNT EQU 0x25
… … ….
MOV R2, #COUNT ;R2 = 0x25
```

```
DATA2 EQU 2_00110101 ;the way to define binary value (35 in hex)
DATA3 EQU 39 ;decimal numbers (27 in hex)
DATA4 EQU ‘2’
```

RN (equate)

Program 2-2: An ARM Assembly Language Program Using RN Directive

```
    ;ARM Assembly Language Program To Add Some Data
   ;and store the SUM in R3.

VAL1      RN          R1          ;define VAL1 as a name for R1
VAL2      RN          R2          ;define VAL2 as a name for R2
SUM       RN          R3          ;define SUM as a name for R3

               AREA     PROG_2_2, CODE, READONLY
   ENTRY
   MOV     VAL1, #0x25                        ;R1 = 0x25
   MOV     VAL2, #0x34                        ;R2 = 0x34
   ADD       SUM, VAL1,VAL2                  ;R3 = R2 + R1
HERE      B             HERE
               END
```

RN: rename reg.
This is used to define a name for a register.
The RN directive does not set aside a seperate storage for the name, but associates a register with that name.

Assembler data allocation directives

DCB directive (define constant byte)

The DCB directive allocates a byte size memory and initializes the values.

```
MYVALUE DCB 5 ;MYVALUE = 5
MYMSAGE DCB “HELLO WORLD” ;string
```

DCW directive (define constant half-word)

The DCW directive allocates a half-word size memory and initializes the values.

```
MYDATA DCW 0x20, 0xF230, 5000, 0x9CD7
```

DCD directive (define constant word)

The DCD directive allocates a word size memory and initializes the values.

```
MYDATA DCD 0x200000, 0xF30F5, 5000000, 0xFFFF9CD7
```

Directive	Description
DCB	Allocates one or more bytes of memory, and defines the initial runtime contents of the memory
DCW	Allocates one or more halfwords of memory, aligned on two-byte boundaries, and defines the initial runtime contents of the memory.
DCWU	Allocates one or more halfwords of memory, and defines the initial runtime contents of the memory. The data is not aligned.
DCD	Allocates one or more words of memory, aligned on four-byte boundaries, and defines the initial runtime contents of the memory.
DCDU	Allocates one or more words of memory and defines the initial runtime contents of the memory. The data is not aligned.

in Strings which is the LSB

Data Size	Bits	Decimal	Hexadecimal	Directive	Instruction
Byte	8	0 – 255	0 - 0xFF	DCB	STRB/LDRB
Half-word	16	0 – 65535	0 - 0xFFFF	DCW	STRH/LDRH
Word	32	0 – 2^{32}-1	0 - 0xFFFFFFFF	DCD	STR/LDR

ADR directive

To load registers with the addresses of memory locations we can also use the ADR pseudo-instruction which has a better performance

ADR Rn,label

ADR R2, OUR_FIXED_DATA ;point to OUR_FIXED_DATA

ALIGN

This is used to make sure data is aligned in 32-bit word or 16-bit half word memory address. The following uses ALIGN to make the data 32-bit word aligned:

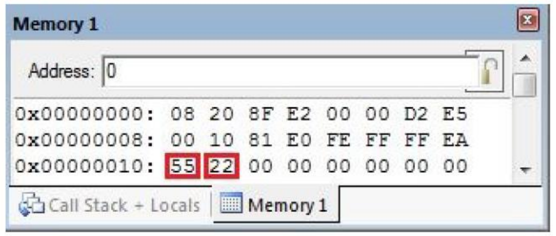
Page 74 Book

ALIGN 4 ;the next instruction is word (4 bytes) aligned

…

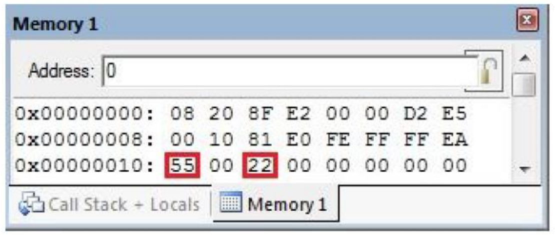
ALIGN 2 ;the next instruction is half-word (2 bytes) aligned

No Align



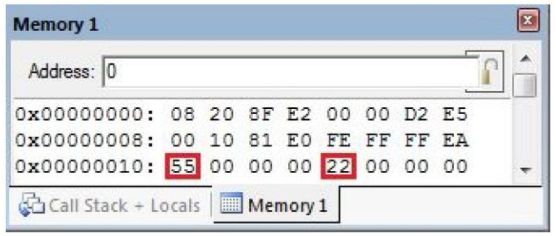
```
0x00000000: 08 20 8F E2 00 00 D2 E5
0x00000008: 00 10 81 E0 FE FF FF EA
0x00000010: 55 22 00 00 00 00 00 00
```

Align 2



```
0x00000000: 08 20 8F E2 00 00 D2 E5
0x00000008: 00 10 81 E0 FE FF FF EA
0x00000010: 55 00 22 00 00 00 00 00
```

Align 4



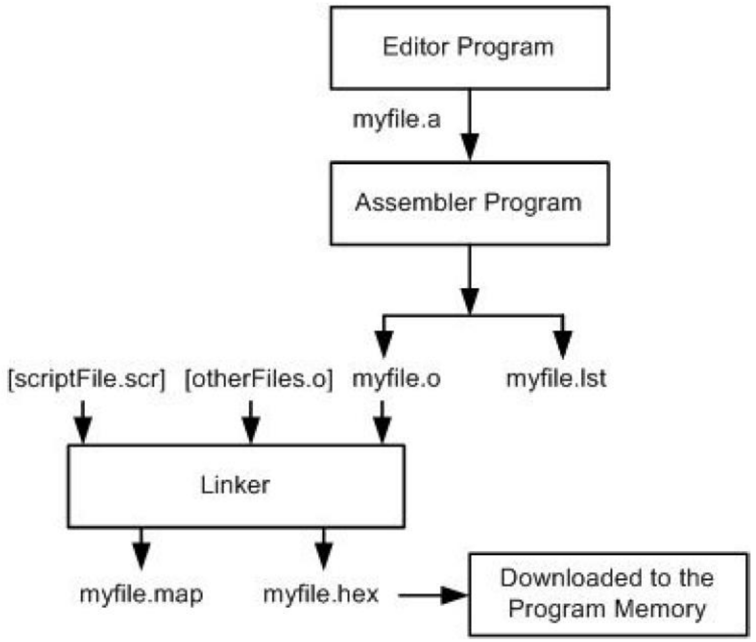
```
0x00000000: 08 20 8F E2 00 00 D2 E5
0x00000008: 00 10 81 E0 FE FF FF EA
0x00000010: 55 00 00 00 22 00 00 00
```

Instruction structure:

[label] mnemonic [operands] [;comment]

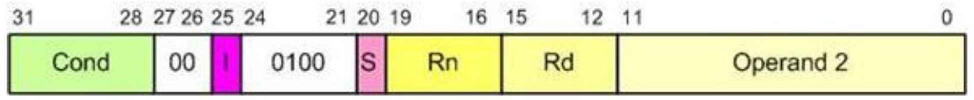
Note!

The first column of each line is always considered as label. Thus, be careful to press a Tab at the beginning of each line that does not have label; otherwise, your instruction is considered as a label and an error message will appear when compiling.

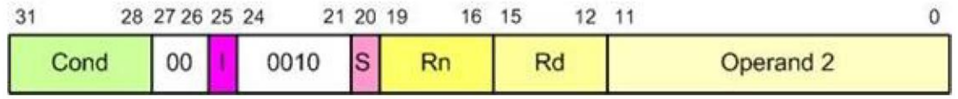


Keil IDE, which has a text editor, assembler, simulator, and much more all in one software package.

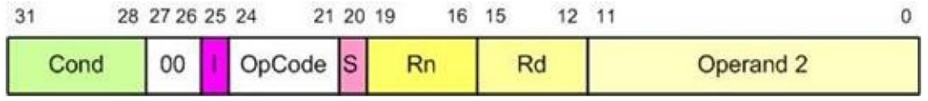
ADD instruction formation



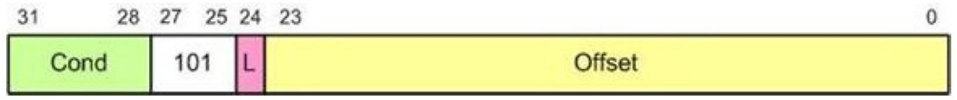
SUB instruction formation



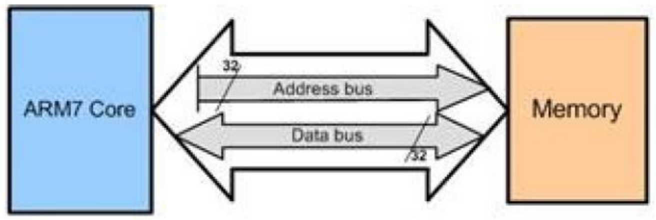
General formation of data processing instructions



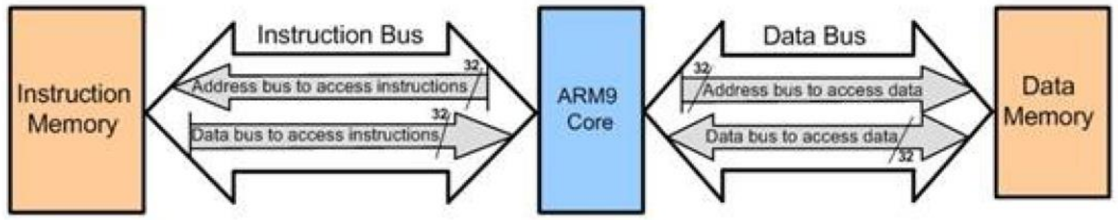
Branch instruction formation



Harvard and von Neumann architectures in the ARM

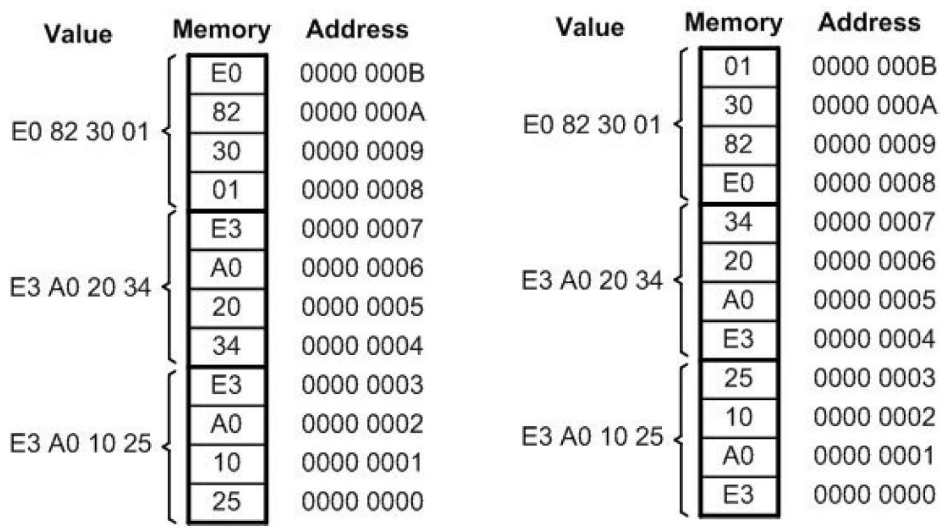


(a) Von Neumann



(b) Harvard

Little endian vs. big endian war

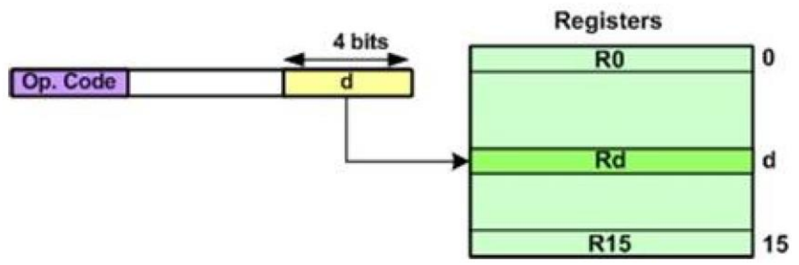


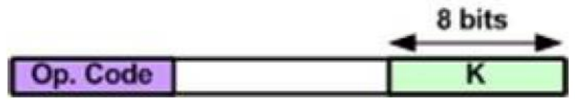
Value	Memory	Address
E0 82 30 01	E0	0000 000B
	82	0000 000A
	30	0000 0009
	01	0000 0008
E3 A0 20 34	E3	0000 0007
	A0	0000 0006
	20	0000 0005
	34	0000 0004
E3 A0 10 25	E3	0000 0003
	A0	0000 0002
	10	0000 0001
	25	0000 0000

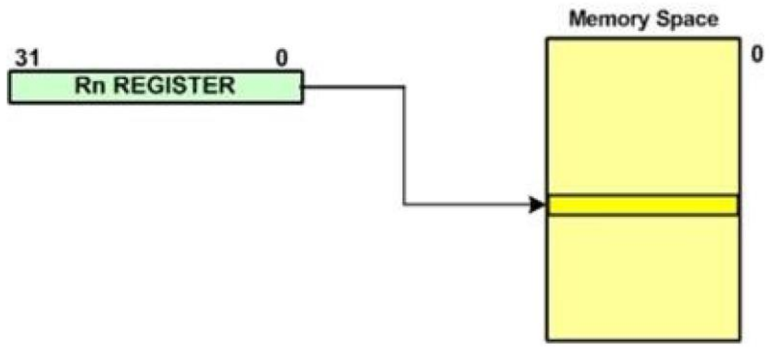
Value	Memory	Address
E0 82 30 01	01	0000 000B
	30	0000 000A
	82	0000 0009
	E0	0000 0008
E3 A0 20 34	34	0000 0007
	20	0000 0006
	A0	0000 0005
	E3	0000 0004
E3 A0 10 25	25	0000 0003
	10	0000 0002
	A0	0000 0001
	E3	0000 0000

ARM Addressing Modes

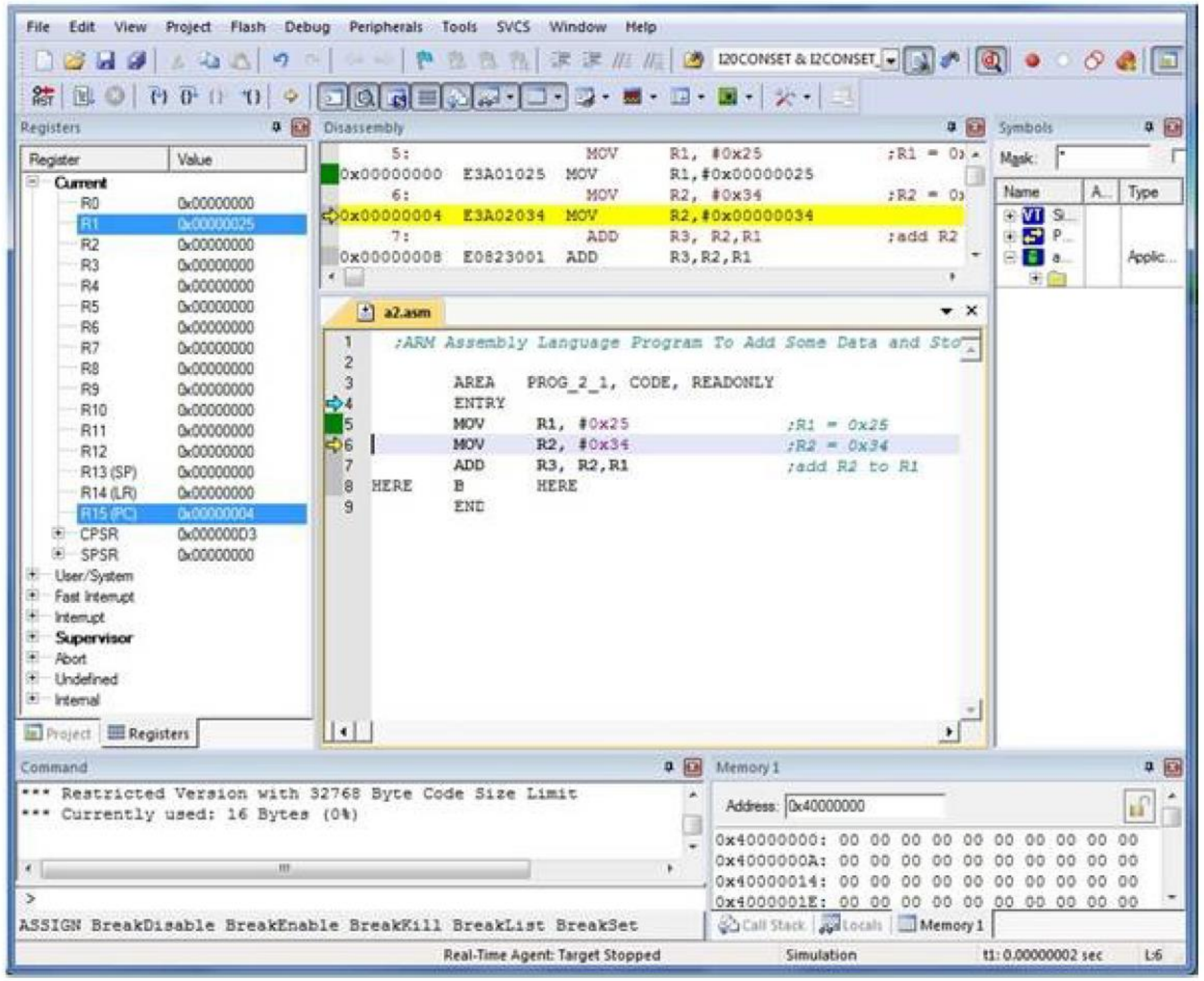
1. register
2. immediate
3. register indirect (indexed addressing mode)







Viewing Registers and Memory with ARM Keil IDE



Register	Value
Current	
R0	0x00000000
R1	0x00000025
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000004
CPSR	0x000000D3
SPSR	0x00000000
User/System	
Fast Interrupt	
Interrupt	
Supervisor	
Abort	
Undefined	
Internal	

Disassembly

```
     5:                      MOV       R1, #0x25                ;R1 = 0x
0x00000000  E3A01025  MOV       R1,#0x00000025
     6:                      MOV       R2, #0x34                ;R2 = 0x
0x00000004  E3A02034  MOV       R2,#0x00000034
     7:                      ADD       R3, R2,R1                ;add R2
0x00000008  E0823001  ADD       R3,R2,R1
```

a2.asm

```
 ;ARM Assembly Language Program To Add Some Data and Sto

        AREA   PROG_2_1, CODE, READONLY
        ENTRY
        MOV     R1, #0x25                      ;R1 = 0x25
        MOV     R2, #0x34                      ;R2 = 0x34
        ADD     R3, R2,R1                      ;add R2 to R1
HERE    B       HERE
        END
```

Command

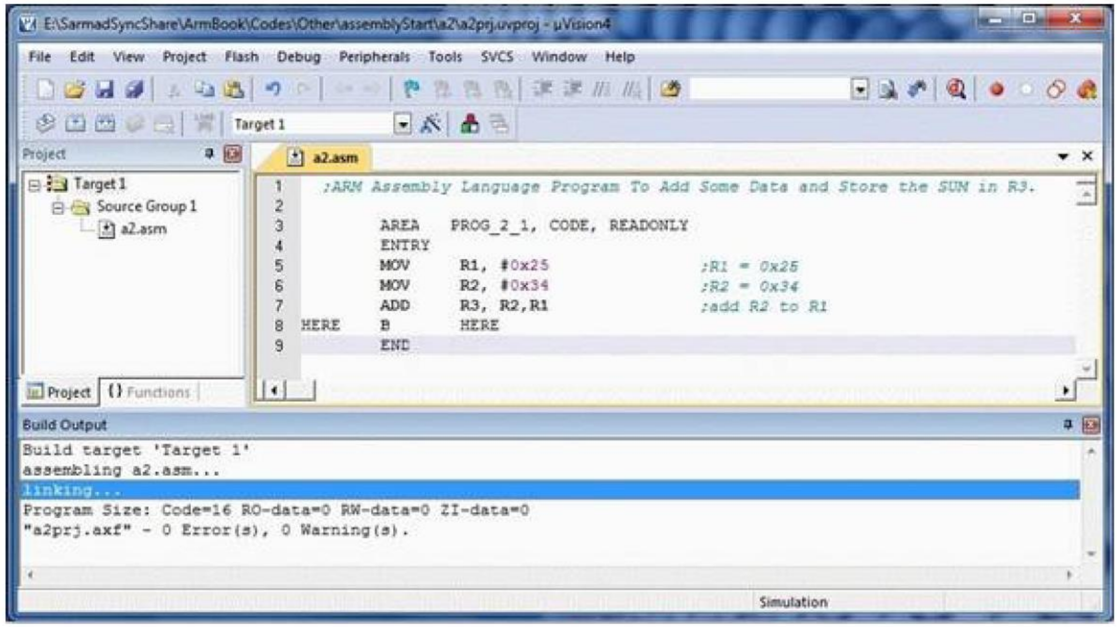
```
*** Restricted Version with 32768 Byte Code Size Limit
*** Currently used: 16 Bytes (0%)
>
ASSIGN BreakDisable BreakEnable BreakKill BreakList BreakSet
```

Memory 1

Address: 0x40000000

```
0x40000000: 00 00 00 00 00 00 00 00 00 00
0x4000000A: 00 00 00 00 00 00 00 00 00 00
0x40000014: 00 00 00 00 00 00 00 00 00 00
0x4000001E: 00 00 00 00 00 00 00 00 00 00
```

Real-Time Agent: Target Stopped Simulation t1: 0.00000002 sec L:6



E:\SarmadSyncShare\ArmBook\Codes\Other\assemblyStart\a2\a2prj.uvproj - µVision4

Target 1 / Source Group 1 / a2.asm

```
 ;ARM Assembly Language Program To Add Some Data and Store the SUN in R3.

        AREA   PROG_2_1, CODE, READONLY
        ENTRY
        MOV     R1, #0x25                      ;R1 = 0x25
        MOV     R2, #0x34                      ;R2 = 0x34
        ADD     R3, R2,R1                      ;add R2 to R1
HERE    B       HERE
        END
```

Build Output

```
Build target 'Target 1'
assembling a2.asm...
linking...
Program Size: Code=16 RO-data=0 RW-data=0 ZI-data=0
"a2prj.axf" - 0 Error(s), 0 Warning(s).
```