

# CO 513 – Microprocessors and Microcontrollers

## Module 1: The ARM Assembly Language

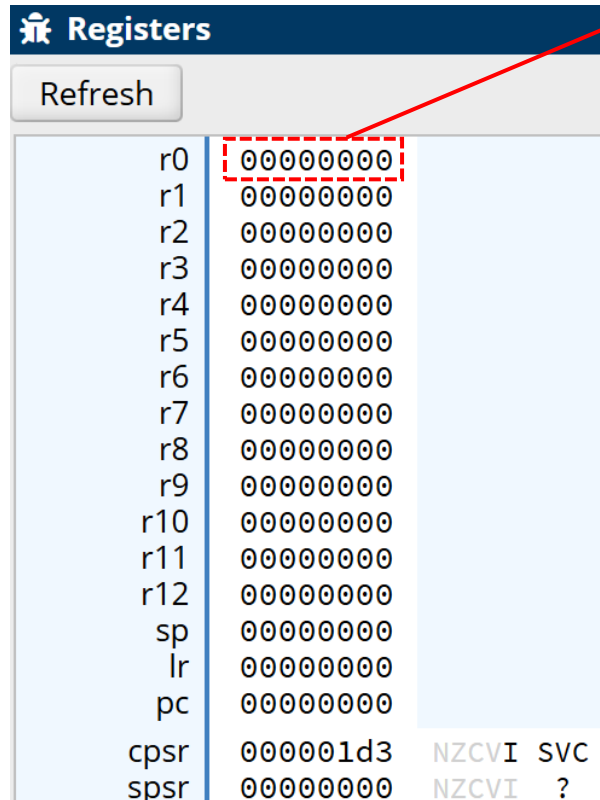
Electronics Engineering Department

College of Engineering

Technological University of the Philippines - Manila

# ARM v7 Registers

Registers are small storage areas located close to the processor for quick access



Registers			
Refresh			
r0	00000000		
r1	00000000		
r2	00000000		
r3	00000000		
r4	00000000		
r5	00000000		
r6	00000000		
r7	00000000		
r8	00000000		
r9	00000000		
r10	00000000		
r11	00000000		
r12	00000000		
sp	00000000		
lr	00000000		
pc	00000000		
cpsr	000001d3	NZCVI	SVC
spsr	00000000	NZCVI	?

Numbers are in hexadecimal (4 bits)

Each register can hold  $4 \times 8 = 32$  bits of data

r0 – r6: general purpose registers

r7: special purpose register stores syscall codes

r8-r12: general purpose registers

sp: stack pointer register

lr: link register

pc: program counter register

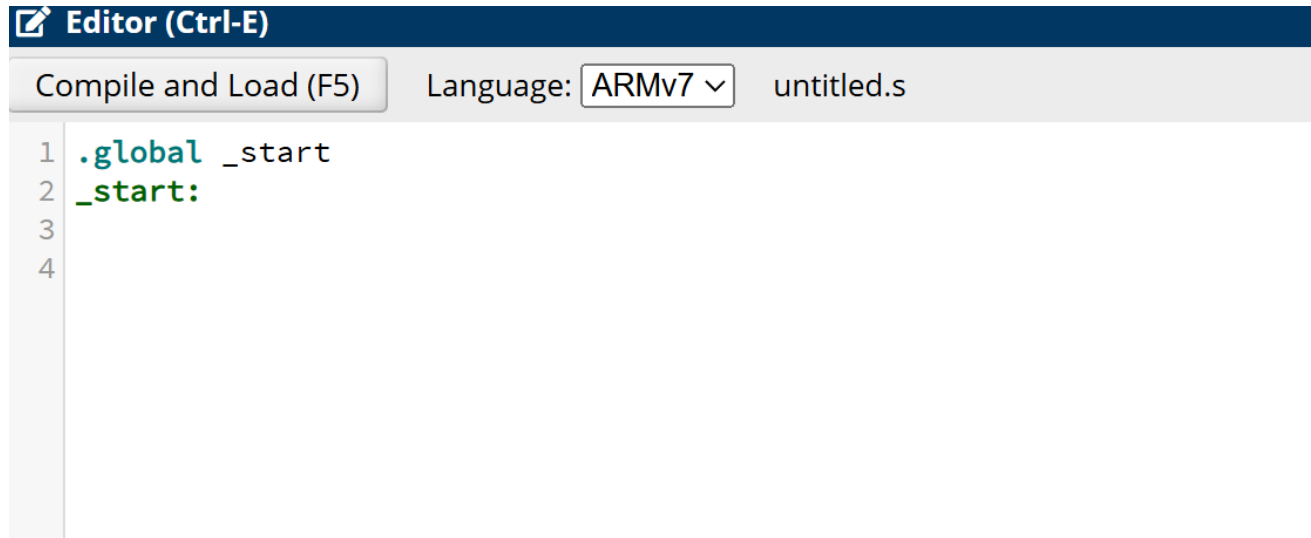
cpsr: current program status register

spsr: saved program status register

# Initialization

---

Every assembly program should be initialized with



The screenshot shows an IDE window titled "Editor (Ctrl-E)". Below the title bar is a toolbar with a button labeled "Compile and Load (F5)". To the right of the button, it says "Language: ARMv7" with a dropdown arrow, and "untitled.s". The main editing area contains the following assembly code:

```
1 .global _start
2 _start:
3
4
```

The **.global** command connects our program externally telling that **\_start** is entry point of our ARM program.

The **\_start** command indicates where the processor begins executing after reset.

# Your First Assembly Code

---

Let us write our first assembly code!

```
1 .global _start
2 _start:
3     MOV r0, #42
4     MOV r7, #0x1
5
6     SWI 0
7     |
```

*Code from demo1.s file*

The MOV command stores a value to a particular destination register.

In this case, we stored the decimal number 42 at register r0 and hexadecimal number 1 at register r7

SWI command calls for the system to do system interrupt

# Addressing Modes

Addressing refers to the different ways we can store and retrieve data from various memory locations that we have.

```
1  .global _start
2  _start:
3      MOV r0, #42 //data processing with immediate operand
4      MOV r1, r0 //data processing with register operand
5      LDR r2, =list //literal addressing (loads address of list in r2)
6      LDR r3, [r2] //register indirect addressing
7      LDR r4, [r2, #4] //register indirect addressing with increment
8      LDR r5, [r2, #4]! //reg.indirect addressing with pre-increment
9      LDR r6, [r2],#4 //reg.indirect addressing with post-increment
10
11  .data
12  list: .word 4, 5, -9, 1, 0, -2, 3
13
```

*Code from demo2.s file*

# Arithmetic Operations and CPSR Flags

Arithmetic operations can also be performed via immediate and register operands

```
1  .global _start
2  _start:
3      MOV r0, #4
4      ADD r1, r0, #5 //immediate ADD
5      ADD r2, r1, r0 //register ADD
6
7      SUB r3, r1, #2 //immediate SUB
8      SUB r3, r1, r0 //register SUB
9
10     SUB r4, r0, r1
11     SUBS r5, r0, r1 //subtract enabling CPSR
12
13     MUL r6, r0, r1 //register MUL (no immediate MUL operand)
```

The bits in the CPSR register comprise different flags that are updated when certain conditions occur

*Code from demo3.s file*

# Arithmetic Operations and CPSR Flags

---

## CPSR Condition Flags (*bits[31:28]*)

- N, bit[31]** Negative condition flag. Set to bit[31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then the processor sets N to 1 if the result is negative, and sets N to 0 if it is positive or zero.
- Z, bit[30]** Zero condition flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.
- C, bit[29]** Carry condition flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.
- V, bit[28]** Overflow condition flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

# Storing to Memory

---

STR (store) instruction can be utilized in the same way as LDR (load) instruction

```
1  .global _start
2  _start:
3      LDR r0, =buff //loads the address of buff on r0
4      MOV r1, #3
5      MOV r2, #14
6
7      STR r1, [r0] //stores r1 to address pointed by r0
8      STR r2, [r0, #4] //stores r2 to address pointed by r0+1
9
10
11
12  .data
13  .align 4 // align the next label (buff) on a 16-byte boundary
14  buff: .space 8 //reserve 8 bytes of uninitialized space
15          // starting at label 'buff'
16
```

*Code from demo4.s file*



# Logical Operations

---

Bitwise logical operations can also be performed

```
1  .global _start
2  _start:
3      MOV r0, #0xA
4      MOV r1, #0x7
5
6      AND r2, r0, r1 //bitwise AND
7      ORR r3, r0, r1 //bitwise OR
8      EOR r4, r0, r1 //bitwise XOR
9      MVN r5, r0      //bitwise inversion
```

*Code from demo5.s file*

# Bit Manipulation

**Logical Shift Left (LSL)** shifts the bits of a register to the left by a specified number of positions, with the new rightmost bits being filled with zeros.

```
.global _start
_start:
    MOV r0, #0x032E // sample operand: 0000 0011 0010 1110
    MOV r1, #2      // shift amount for the register form

    LSL r2, r0, #2   // r2 = r0 << 2 (immediate)
    LSL r3, r0, r1   // r3 = r0 << r1 (register)
```

*Code from demo6.s file*

r0	0000032e
r1	00000002
r2	00000cb8
r3	00000cb8
r4	00000000
r5	00000000
r6	00000000
r7	00000000

Original Data: **0000 0011 0010 1110**

Data after logical shift left (LSL): **0000 1100 1011 1000**

# Bit Manipulation

**Arithmetic Shift Right (ASR)** shifts the bits of a register to the right and fills the vacated bits on the left with copies of the original sign bit.

```
1  .global _start
2  _start:
3      MOV r0, #-325 //1111 1111 1111 1111 1111 1110 1011 1011
4                      //0xffffebb
5
6      ASR r1, r0, #2 // r1 = r0 >> 2 (Arithmetic Shift)
7      LSR r2, r0, #2 // r1 = r0 >> 2 (Logical Shift)
```

*Code from demo7.s file*

r0	fffffebb
r1	fffffffae
r2	3ffffffae
r3	00000000
r4	00000000
r5	00000000
r6	00000000
r7	00000000

Original Data: **1111 1111 1111 1111 1111 1110 1011 1011**

Data after arithmetic shift right (ASR): **1111 1111 1111 1111 1111 1111 1010 1110**

Data after logical shift right (LSR): **0011 1111 1111 1111 1111 1111 1010 1110**

# Bit Manipulation

**Logical Shift Right (LSR)** shifts the bits of a register to the right and fills the vacated bits on the left with zeros.

```
1  .global _start
2  _start:
3      MOV r0, #-325 //1111 1111 1111 1111 1111 1110 1011 1011
4                      //0xffffebb
5
6      ASR r1, r0, #2 // r1 = r0 >> 2 (Arithmetic Shift)
7      LSR r2, r0, #2 // r1 = r0 >> 2 (Logical Shift)
```

*Code from demo7.s file*

r0	fffffebb
r1	fffffffae
r2	3ffffffae
r3	00000000
r4	00000000
r5	00000000
r6	00000000
r7	00000000

Original Data: **1111 1111 1111 1111 1111 1110 1011 1011**

Data after arithmetic shift right (ASR): **1111 1111 1111 1111 1111 1111 1010 1110**

Data after logical shift right (LSR): **0011 1111 1111 1111 1111 1111 1010 1110**

# Bit Manipulation

**Rotate Right (ROR)** instruction shifts all bits in a register to the right, with the bits shifted out from the least significant bit (LSB) side wrapping around and being inserted into the most significant bit (MSB) position

```
1  .global _start
2  _start:
3      MOV r0, #0b10010011 //0000 0000 0000 0000 0000 0000 1001 0011
4      ROR r1, r0, #4      // r4 = r0 ROR 4 (immediate)
```

*Code from demo8.s file*

r0	00000093
r1	30000009
r2	00000000
r3	00000000
r4	00000000
r5	00000000
r6	00000000
r7	00000000

Original Data: **0000 0000 0000 0000 0000 0000 1001 0011**

Data after rotate right (ROR): **0011 0000 0000 0000 0000 0000 0000 1001**

# The CMP Instruction

The **compare (CMP)** is an instruction that compares two values by performing a subtraction but discards the result, instead setting the processor's condition flags in the CPSR.

```
1  .global _start
2  _start:
3      MOV r0, #4
4      MOV r1, #5
5      MOV r2, #3
6      MOV r3, #4
7
8      cmp r0, r1 // r0 < r1, result is -ve, set N to 1
9      cmp r0, r2 // r0 > r2, result is +ve, set C to 1
10     cmp r0, r3 // r0 == r3, result is 0, set Z and C to 1
```

Since CMP is subtraction, the carry flag is being set. **C = 1** means “no borrow” is needed while **C = 0** means borrow is needed

24  
- 5  
----  
19

*Code from demo9.s file*

# Branching

**Branching** refers to the process of changing the Program Counter (PC), which dictates the next instruction to be executed, instead of following a sequential path.

## B

Branch causes a branch to a target address.

B<c> <label>

cond	Mnemonic extension	Meaning (integer)	Meaning (floating-point) <sup>a</sup>	Condition flags
0000	EQ	Equal	Equal	Z == 1
0001	NE	Not equal	Not equal, or unordered	Z == 0
0010	CS <sup>b</sup>	Carry set	Greater than, equal, or unordered	C == 1
0011	CC <sup>c</sup>	Carry clear	Less than	C == 0
0100	MI	Minus, negative	Less than	N == 1
0101	PL	Plus, positive or zero	Greater than, equal, or unordered	N == 0
0110	VS	Overflow	Unordered	V == 1
0111	VC	No overflow	Not unordered	V == 0
1000	HI	Unsigned higher	Greater than, or unordered	C == 1 and Z == 0
1001	LS	Unsigned lower or same	Less than or equal	C == 0 or Z == 1
1010	GE	Signed greater than or equal	Greater than or equal	N == V
1011	LT	Signed less than	Less than, or unordered	N != V
1100	GT	Signed greater than	Greater than	Z == 0 and N == V
1101	LE	Signed less than or equal	Less than, equal, or unordered	Z == 1 or N != V
1110	None (AL) <sup>d</sup>	Always (unconditional)	Always (unconditional)	Any

# Branching

---

When a branch condition is not satisfied, the program flow just continues sequentially.

```
1  .global _start
2  _start:
3      MOV r0, #4
4      MOV r1, #5
5
6      cmp r0, r1    // Compare r0 with r1 (sets condition flags)
7      beq cond1    // Branch to cond1 if r0 == r1
8      b cond2      // Otherwise, jump to cond2
9
10 cond1:
11     mov r2, #3
12
13 cond2:
14     mov r3, #3
```

*Code from demo10.s file*



# Branching

---

Without proper branching, the program just flows sequentially. This can lead to potential errors in program logic so be careful!

```
1  .global _start
2  _start:
3      MOV r0, #4
4      MOV r1, #5
5
6      cmp r0, r1    // Compare r0 with r1 (sets condition flags)
7      bne cond1    // Branch to cond1 if r0 != r1
8      b cond2      // Otherwise, jump to cond2
9
10 cond1:
11     mov r2, #3
12     b end        // Prevent falling through into cond2
13
14 cond2:
15     mov r3, #3
16
17 end: // Program flow continues here
```

*Code from demo11.s file*

# Branching

```
1 .global _start
2 _start:
3     LDR r0, =num1      // Load address of num1
4     LDR r1, [r0]       // Indirect addressing: load value of num1
5     LDR r0, =num2      // Load address of num2
6     LDR r2, [r0]       // Indirect addressing: load value of num2
7
8     CMP r1, r2         // Compare num1 with num2
9     BLT less_than      // If num1 < num2, branch to less_than
10
11 greater_equal:
12     ADD r3, r1, r2     // r3 = num1 + num2
13     ORR r4, r1, r2     // r4 = bitwise OR of num1 and num2
14     B end
15
16 less_than:
17     SUB r3, r2, r1     // r3 = num2 - num1
18     AND r4, r1, r2     // r4 = bitwise AND of num1 and num2
19
20 end:
21     // Program flow continues here
22
23 .data
24 num1: .word 4
25 num2: .word 7
```

*Code from demo12.s file*

Try to compile and load the following demo code.

Track what happens to the registers and the memory as step over into the code .

# Loop Implementation

---

Loops in ARM assembly are implemented using a combination of conditional branch instructions and labels.

```
1  .global _start
2  _start:
3      mov r0, #0    // Store 0 to r0
4
5  loop:
6      cmp r0, #5    // Compare r0 with #5
7      bge end       // Branch to end if r0 >= 5
8      ADD r0, #1    // r0 = r0 + 1
9      b loop        // Branch to loop
10
11 end:
12                //Program flow continues here
```

*Code from demo13.s file*

# Function Calls

---

Functions can also be created in ARM through the use of bl and bx commands.

```
1 .global _start
2 _start:
3     mov r0, #2 //arg1
4     mov r1, #3 //arg 2
5
6     push {r0, r1}    // Save arguments r0 and r1 on the stack (to restore later)
7     bl add_nums      // Branch with link: call add_nums function
8
9     mov r2, r0
10    pop {r0, r1}      // Restore original arguments from the stack into r0 and r1
11
12    b end
13
14 add_nums:
15     add r0, r0, r1
16     bx lr            // Return from function, jump back to caller
17
18 end:
```

*Code from demo14.s file*

# Using Stack Memory

---

Stack memory is used for a variety of reasons:

- Function arguments can be passed through the stack when registers are not enough.
- The stack stores the return address when a function is called (similar to what `bx lr` does) .
- Memory efficient because stack grows/shrinks dynamically as functions are called and returned.

# Using Stack Memory

```
1 .global _start
2 _start:
3     // Prepare six arguments for function call
4     // First 4 arguments go in r0-r3 (ARM calling convention)
5     mov r0, #1        // First argument
6     mov r1, #2        // Second argument
7     mov r2, #3        // Third argument
8     mov r3, #4        // Fourth argument
9
10    // Arguments 5 and 6 must be pushed onto stack
11    mov r4, #6         // Sixth argument
12    push {r4}          // Push sixth argument on stack
13    mov r4, #5         // Fifth argument
14    push {r4}          // Push fifth argument on stack
15
16    // Call add_six function
17    bl add_six         // Branch with link to add_six
18
19    add sp, sp, #8     // Adjust stack pointer (2 words = 8 bytes)
20
21    // r0 contains the sum
22
23    // End program - infinite loop for CPUlator
24 end:
25    b end              // Branch to self (infinite loop)
```

```
26
27 add_six:
28     push {lr}         // Save link register
29
30     // Arguments 1-4 are already in r0-r3
31     // Arguments 5-6 are on the stack
32
33     // Add first four arguments
34     add r0, r0, r1     // r0 = arg1 + arg2
35     add r0, r0, r2     // r0 = r0 + arg3
36     add r0, r0, r3     // r0 = r0 + arg4
37
38     // Get arguments 5 and 6 from stack
39     // Stack layout: [top] -> return_addr, arg5, arg6
40     ldr r4, [sp, #4]   // Load fifth argument (offset 4)
41
42     add r0, r0, r4     // r0 = r0 + arg5
43     ldr r4, [sp, #8]   // Load sixth argument (offset 8)
44     add r0, r0, r4     // r0 = r0 + arg6
45
46     // Result is in r0 (return value)
47     pop {lr}          // Restore link register
48     bx lr             // Return to caller
```

*Code from demo15.s file*

# Thank You!

---

## Module 1: The ARM Assembly Language

Engr. Timothy M. Amado  
Faculty, Electronics Engineering Department  
Technological University of the Philippines – Manila