

# CS301

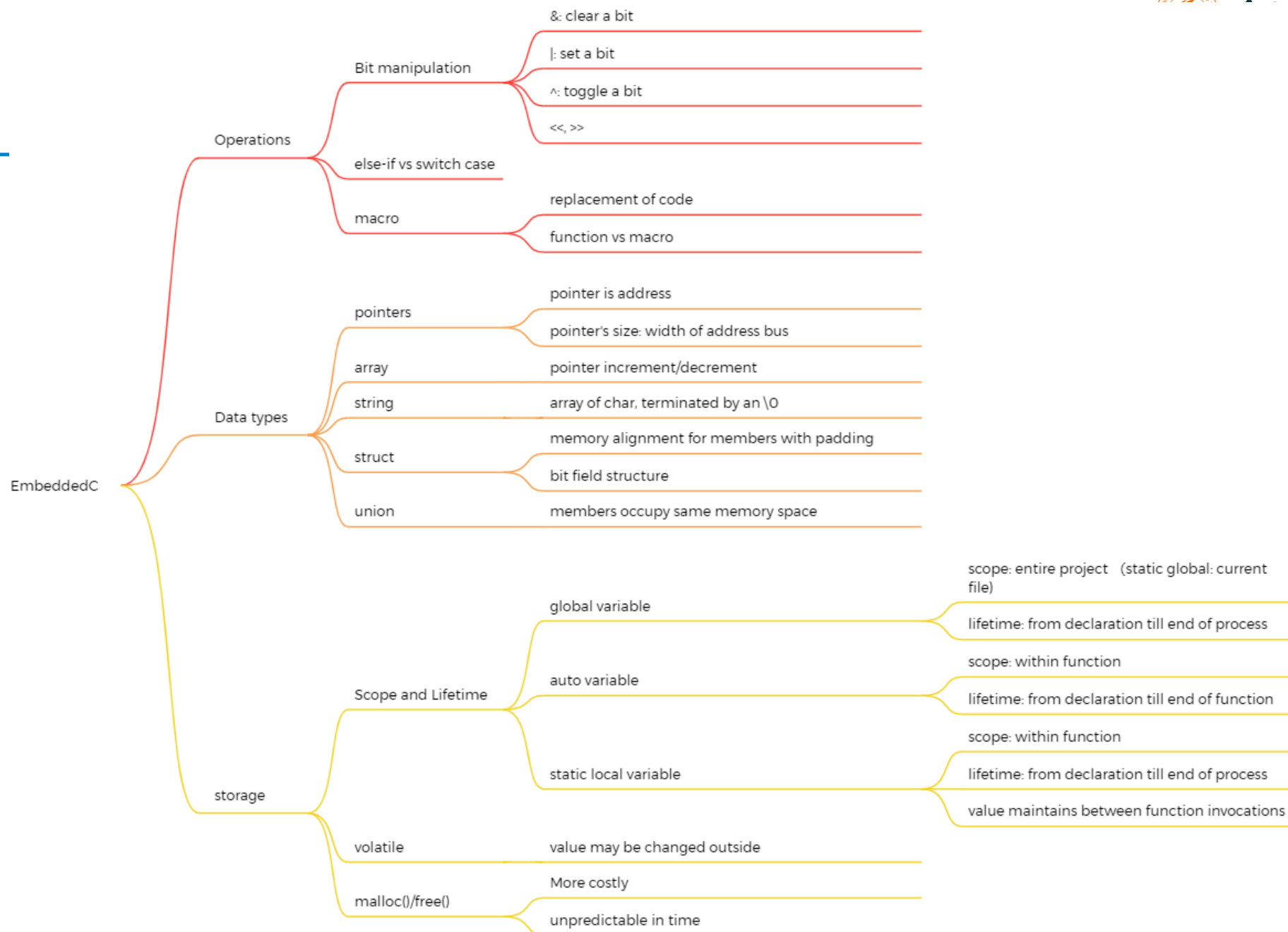
## Embedded System and Microcomputer Principle

### Lecture 4: ARM Assembly

2024 Fall

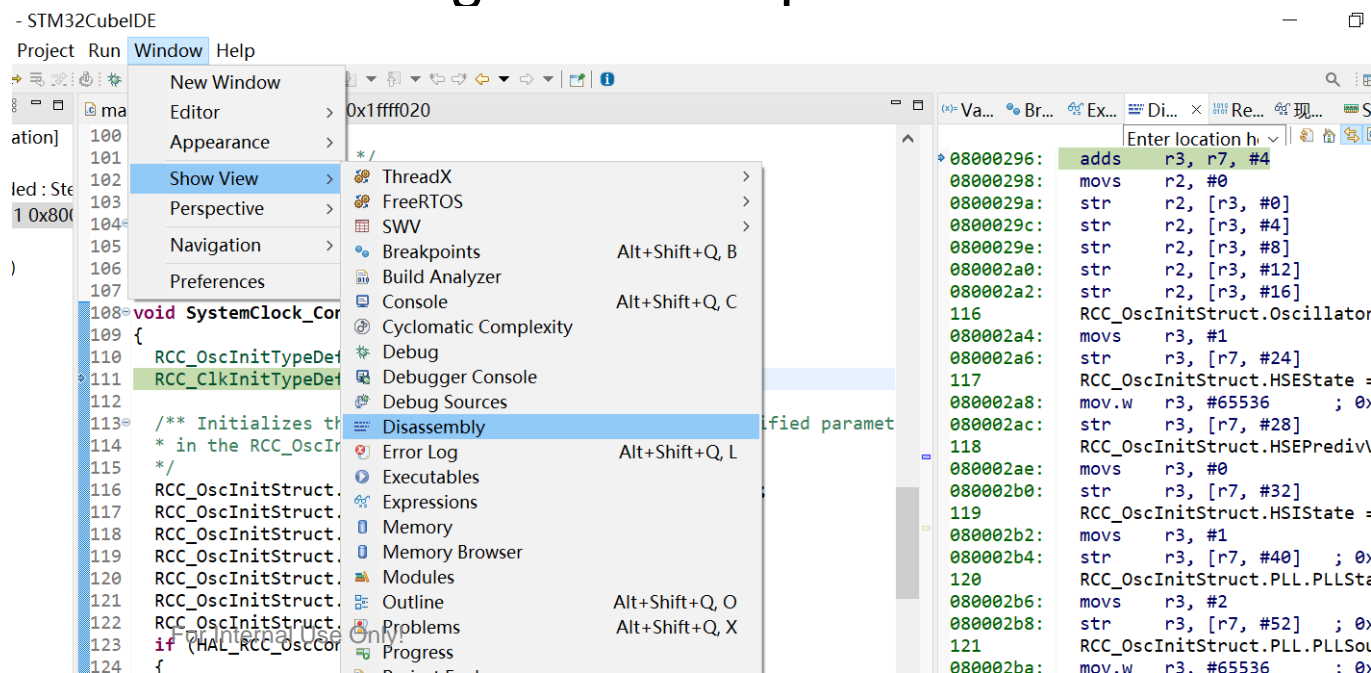
This PowerPoint is for internal use only at Southern University of Science and Technology.  
Please do not repost it on other platforms without permission from the instructor.

# Recap





# Compiler

- A high level language (such as C, C++, Fortran, etc.) is converted into either machine code or mnemonics using a computer package called a **compiler**.
- Most programs are written in a high level language
- But assembly language programming is commonly used for engineering systems which must operate in real time e.g. a mobile phone.
- Nobody writes computer programs using machine code.



# Assembler

- A computer package called an **assembler** converts an assembly language program into a machine code program.
- E.g.

Mnemonic (助记符)		Machine Code		in Memory
MOV r12, #114		0xE3A0C072		0X00008000
MOV r7, #0xCB	<b>assembler</b>	0xE3A070CB	<b>linker</b>	0X00008004
MOV r6, r14		0xE1A0600E		0X00008008
MOV r7, r12		0xE1A0700C		0X0000800C

- In ARM mode, each instruction occupying 4 adjacent memory locations, as each instruction is 32 bits long. Cortex-M is Thumb-2 mode (mix of 16/32bits instructions)
- The machine code can be downloaded to the microprocessor memory.

# Assembly Language

- Mnemonics
  - In general nobody remembers all of the machine code for any particular processor (or indeed any).
  - Instead we use mnemonics
    - mnemonics are words or phrases which are easy to remember and can replace something which is difficult to remember.
- Assembly language
  - If the mnemonics for every instruction in a computer program were listed in the order that they were executed then the resulting list would be an assembly language program.
- Example:

```
MOV r6, r14
MOV r7, #0xCB
MOV r7, r12
MOV r12, #114
```

# Assembly Format

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

- label
  - Place marker, memory address of the current instruction
  - Used by branch instructions to implement if-then or goto
- opcode
  - The name of the instruction
  - Operation to be performed by processor core
- operands
  - Registers
  - Constants (called immediate values)
- comments
  - Everything after the semicolon (;) is a comment
  - Explain programmers' intentions or assumptions

# Assembly Instructions

- Arithmetic and logic
  - Add, Subtract, Multiply, Shift, Rotate
- Data movement
  - Load, Store, Move
- Compare and branch
  - Compare, Branch

# Instructions for Arithmetic

- The ARM7 can add, subtract and multiply numbers (but not divide).

- Opcode destination, source1, source2
  - Opcodes: ADD, SUB, MUL, etc.

- Examples:

- **ADD R5,R2,R1**

- $R5 = R2 + R1$

- **SUB R5,R1,#23**

- $R5 = R1 - 23$

- **RSB R5,R1,R2**

- $R5 = R2 - R1$  , reverse subtraction

- **MUL R5,R2,R1**

- $R5 = R2 * R1$

- If the result is more than 32 bits long, the destination register, R5 only holds the bottom 32 bits of the result and the rest is lost

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)



# Flags

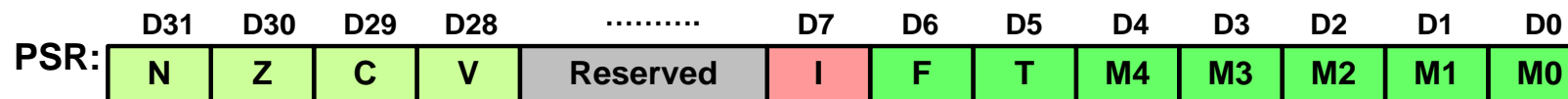
$a = 10000$

$b = 10000$

$c = a + b$

- Are  $a$  and  $b$  signed or unsigned numbers?
  - CPU does not know the answer at all.
  - Therefore, the hardware sets up both the carry flag and the overflow flag.
  - It is software's (programmers'/compilers') responsibility to interpret the flags.
  - Noted: In computers, numbers are stored in their two's complement representation.

# Condition Flags in Program Status Register



- Condition Flags: NZCV
  - **N**egative bit
    - N = 1 if most significant bit of result is 1
  - **Z**ero bit
    - Z = 1 if all bits of result are 0
  - **C**arry bit
    - For unsigned addition, C = 1 if carry takes place
    - For unsigned subtraction, C = 0 if borrow takes place (carry = not borrow)
    - For shift/rotation, C = last bit shifted out
  - **oV**erflow bit
    - V = 1 if adding 2 same-signed numbers produces a result with the opposite sign
      - Positive + Positive = Negative, or
      - Negative + negative = Positive
    - Non-arithmetic operations does not touch V bit, such as MOV, AND, LSL

# Carry

- Carry/borrow flag bit for **unsigned** numbers

carry	1	1	1	1	1
	1	1	1	1	1
+	0	1	0	1	1
<hr/>					
	1	0	1	0	1
	0				0

Extra bit is discarded

5-bit result

- **Carry flag = 1**, indicating carry has occurred on unsigned addition.
- Carry flag is 1 because the result crosses the boundary between 31 and 0.

borrow	1		2		
	2	2	0	0	2
	0	1	1	0	1
-	1	0	1	1	1
<hr/>					
	1	0	1	1	0

= (10)<sub>2</sub>

- **Carry flag = 0**, indicating borrow has occurred on unsigned subtraction.
- For subtraction, carry = NOT borrow.

# Overflow

- Two's Complement **Signed** Integer Add/Sub

$$\begin{array}{r}
 0\ 1\ 1\ 0\ 0 \\
 +\ 0\ 0\ 1\ 0\ 1 \\
 \hline
 1\ 0\ 0\ 0\ 1
 \end{array}
 \qquad
 \begin{array}{r}
 12 \\
 +\ 5 \\
 \hline
 -15
 \end{array}$$

5-bit result

Overflow occurs if  $sum \geq 2^n$  when adding two positives, i.e. result becomes negative.

$$\begin{array}{r}
 1\ 0\ 0\ 1\ 1 \\
 +\ 1\ 1\ 0\ 0\ 1 \\
 \hline
 1\ 0\ 1\ 1\ 0\ 0
 \end{array}
 \qquad
 \begin{array}{r}
 -13 \\
 +\ -7 \\
 \hline
 12
 \end{array}$$

Extra bit is discarded.      5-bit result

overflow occurs if  $sum < -2^n$  when adding two negatives, i.e. result becomes positive

Overflow never occurs when adding two numbers with different signs

# Exercise

- Assume a four-bit system unsigned and signed operations

Expression	Result	Carry if unsigned	Overflow if signed
<b>0100 + 0010</b>	<b>0110</b>		
<b>0100 + 0110</b>	<b>1010</b>		
<b>1100 + 1110</b>	<b>1010</b>		
<b>1100 + 1010</b>	<b>0110</b>		

# Exercise

- Assume a four-bit system unsigned and signed operations

Expression	Result	Carry if unsigned	Overflow if signed
<b>0100 + 0010</b>	<b>0110</b>	<b>No</b>	<b>No</b>
<b>0100 + 0110</b>	<b>1010</b>	<b>No</b>	<b>Yes</b>
<b>1100 + 1110</b>	<b>1010</b>	<b>Yes</b>	<b>No</b>
<b>1100 + 1010</b>	<b>0110</b>	<b>Yes</b>	<b>Yes</b>

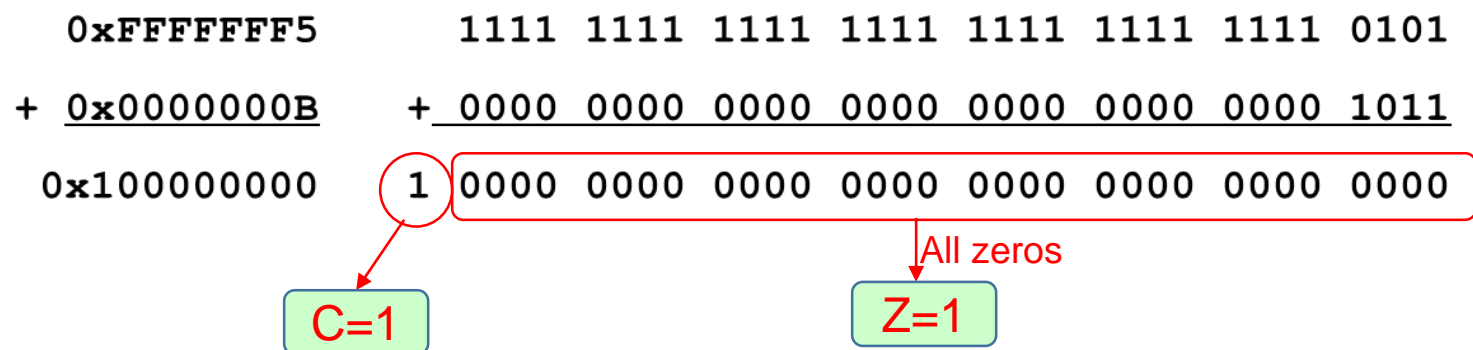
# Updating NZCV flags in PSR

- Most instructions update NZCV flags only if 'S' suffix is present
  - **ADD** r0, r1, r2 ; r0 = r1 + r2, NZCV flags **unchanged**
  - **ADD S** r0, r1, r2 ; r0 = r1 + r2, NZCV flags **updated**
- Some instructions update NZCV flags even if no S is specified.
  - **CMP**: Compare, like SUBS but without destination register
  - **CMP** r1, r2 vs **SUBS** r0, r1, r2

Flags not changed		Flags updated
ADD	→	ADD <b>S</b>
SUB	→	SUB <b>S</b>
MUL	→	MUL <b>S</b>
AND	→	AND <b>S</b>
ORR	→	ORR <b>S</b>
LSL	→	LSL <b>S</b>
MOV	→	MOV <b>S</b>

# Example

```
MOV    R2, #0xFFFFFFFF ; R2 = 0xffffffff5
MOV    R3, #0x0B        ; R3 = 0x0B
ADDS   R1, R2, R3        ; R1 = R2 + R3 and update the flags
```



- NZCV results:
  - N (Negative) = 0 ; bit 31 of result is 0
  - Z (Zero) = 1 ; IsZero(result)
  - C (Carry) = 1 ; carry, result crosses the boundary of 32 bits
  - V (oVerflow) = 0 ; adding +ve and -ve values, never overflow



# Example

```
MOV    R2, #0x7FFFFFFF ; R2 = 0x7fffffff
MOV    R3, #0x2         ; R3 = 0x2
ADDS   R1, R2, R3        ; R1 = R2 + R3 and update the flags
```

$0x7FFFFFFF$   
 $+ 0x00000002$   


---

 $0x80000001$

The addition is shown in binary:
   
 $0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$ 
  
 $+ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010$ 


---

 $1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$

The carry bit (1) is circled in red and labeled **N=1**.  
 The overflow bit (1) is circled in red and labeled **V=1**.

- NZCV results:
  - N (Negative) = 1 ; bit 31 of result is 1
  - Z (Zero) = 0 ; not zero
  - C (Carry) = 0 ; carry, result doesn't cross 32 bits boundary
  - V (oVerflow) = 1 ; overflow, +ve add +ve, result becomes -ve

# Example

- Show the status of the C and Z flags after the addition of 0x38 and 0x2F in the following instructions:

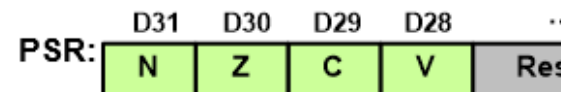
```
MOV    R6, #0x38    ;R6 = 0x38
MOV    R7, #0x2F    ;R17 = 0x2F
ADDS   R6, R6, R7    ;add R7 to R6
```

- Show the status of the Z flag after the subtraction of 0x23 from 0xA5 in the following instructions:

```
LDR    R0, =0xA5
LDR    R1, =0x23
SUBS   R0, R0, R1    ;subtract R1 from R0
```

# Flags in PSR Register

## • Debug in CubeIDE



Core/Src/flags.s - STM32CubeIDE

object Run Window Help

The screenshot shows the STM32CubeIDE interface. The main editor displays the assembly file `flags.s` with the following code:

```

1 /*
2  * flags.s
3  *
4  */
5
6 .syntax unified
7
8 .text
9 .global ASM_Flag_Function
10 .thumb_func
11
12 //-----
13 ASM_Flag_Function:
14     // show PSR Flags
15     MOV R2, #0xFFFFFFFF // R2 = 0xffffffff
16     MOV R3, #0x0B // R3 = 0x0B
17     ADDS R1, R2, R3 // R1 = R2 + R3 and update the flags
18     HERE: B HERE // stay here forever
19 //-----
20 END:
21

```

The register window on the right shows the following values:

Name	Value	Description
r1	0	
r2	-11	
r3	11	
r4	536870956	
r5	0	
r6	0	
r7	536920056	
r8	0	
r9	0	
r10	0	
r11	0	
r12	0	
sp	0x2000bfff	
lr	134218381	
pc	0x8000276 <HER...	
xpsr	0x61000000 (Hex)	
primask	0	

```

MOV     R2, #0xFFFFFFFF ; R2 = 0xffffffff
MOV     R3, #0x0B        ; R3 = 0x0B
ADDS    R1, R2, R3       ; R1 = R2 + R3 and update the flags

```

0xFFFFFFFF5	1111 1111 1111 1111 1111 1111 0101
+ 0x0000000B	+ 0000 0000 0000 0000 0000 0000 1011
0x100000000	1 0000 0000 0000 0000 0000 0000 0000

- NZCV results:
- N (Negative) = 0
  - Z (Zero) = 1
  - C (Carry) = 1
  - V (overflow) = 0

For Internal Use Only

# Instructions using logic

- **AND r0, r1, r2** ; Bitwise AND,  $r0 = r1 \text{ AND } r2$ 
  - clear a specific bit(s) of a byte
- **ORR r0, r1, r2** ; Bitwise OR,  $r0 = r1 \text{ OR } r2$ 
  - set a specific bit(s) of a byte
- **EOR r0, r1, r2** ; Bitwise Exclusive OR,  $r0 = r1 \text{ EOR } r2$ 
  - toggle a specific bit(s) of a byte
- **BIC r0, r1, r2** ; Bit clear,  $r0 = r1 \text{ AND } \sim r2$

<b>AND</b>	0x35	0	0	1	1	0	1	0	1
	0x0F	0	0	0	0	1	1	1	1
	0x05	0	0	0	0	0	1	0	1

<b>EOR</b>	0x44	0	1	0	0	0	1	0	0
	0x06	0	0	0	0	0	1	1	0
	0x34	0	1	0	0	0	0	1	0

<b>ORR</b>	0x04	0	0	0	0	0	1	0	0
	0x30	0	0	1	1	0	0	0	0
	0x34	0	0	1	1	0	1	0	0

<b>BIC</b>	0xFE	1	1	1	1	1	1	1	0
	0x11	0	0	0	1	0	0	0	1
	0xEE	1	1	1	0	1	1	1	0

# Instructions using logic

- LSL r3, r2, #3; Logical Shift Left

; r2 = 0x0000\_0003

LSL r3, r2, #3

; r3 = 0x0000\_0018 ( $24 = 2^3 * 3$ )

- LSR r1, r2, #3; Logical Shift Right , r1 = r2 >> 3

; r2 = 0x0000\_0010

LSR r1, r2, #3

; r1 = 0x0000\_0002 ( $2 = 16/2^3$ )

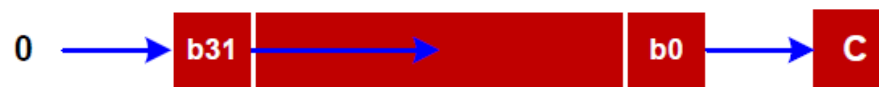
- ROR r2, r2, #10; Rotate Right

**If rotate left by 12 bits**

; r0 = 0xF000\_0000

ROR r2, r0, #20

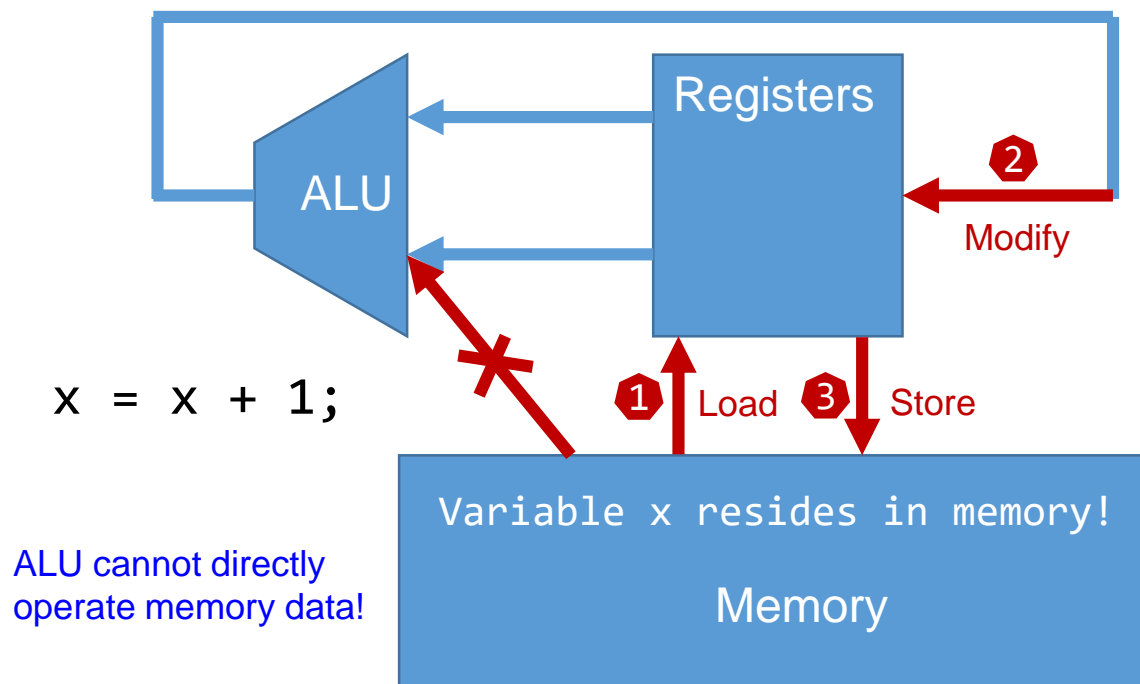
; r2 = 0x0000\_0F00 (Rotate left by m bits  
is equivalent to rotate right ROR by 32-m bits)



For shift/rotation, C = last bit shifted out

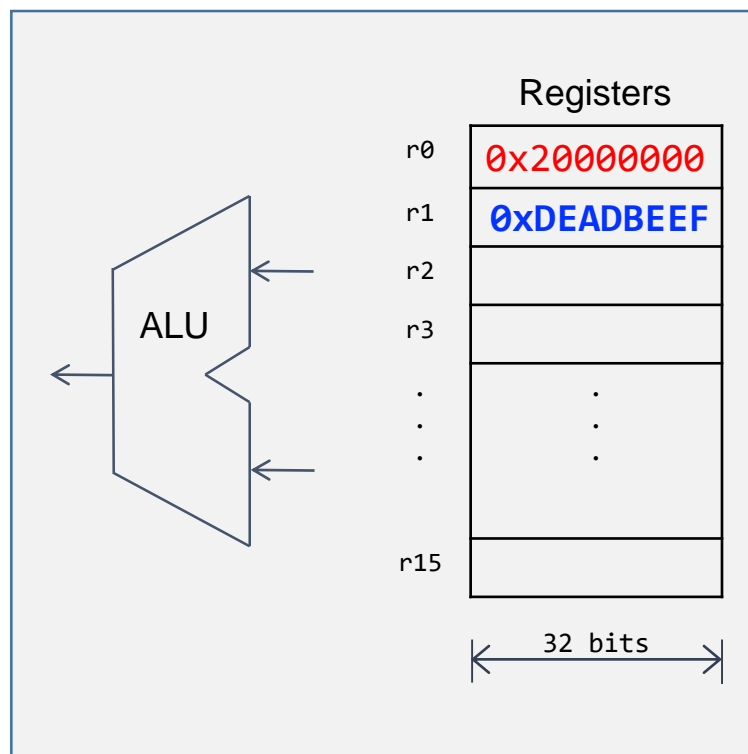
# Data Transfer Instructions

- **MOV** `r0, r1` ; Move,  $r0 = r1$
- **MVN** `r0, r1` ;  $r0 = 1$ 's Complement of  $r1$
- **LDR** `r0, [r1]` ; load value from memory location  $[r1]$  to  $r0$
- **STR** `r0, [r1]` ; store value  $r0$  into memory location  $[r1]$

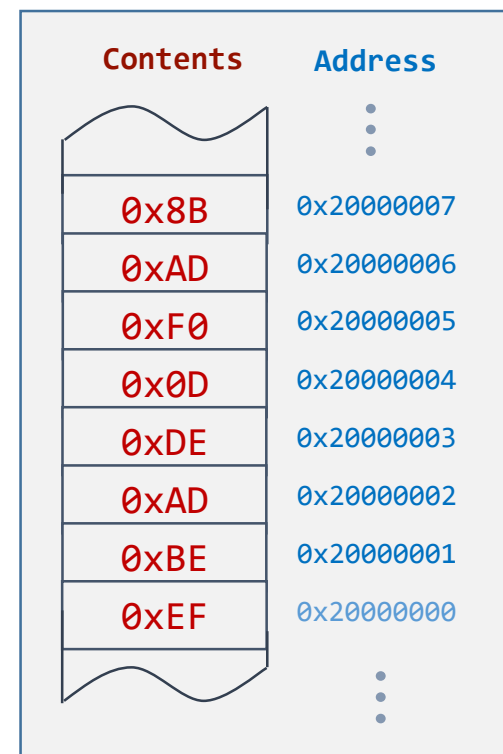


# Load

- Loading Word from Memory
  - `LDR r1, [r0] ; r1 = memory.word[r0]`
  - the data travels from memory to register



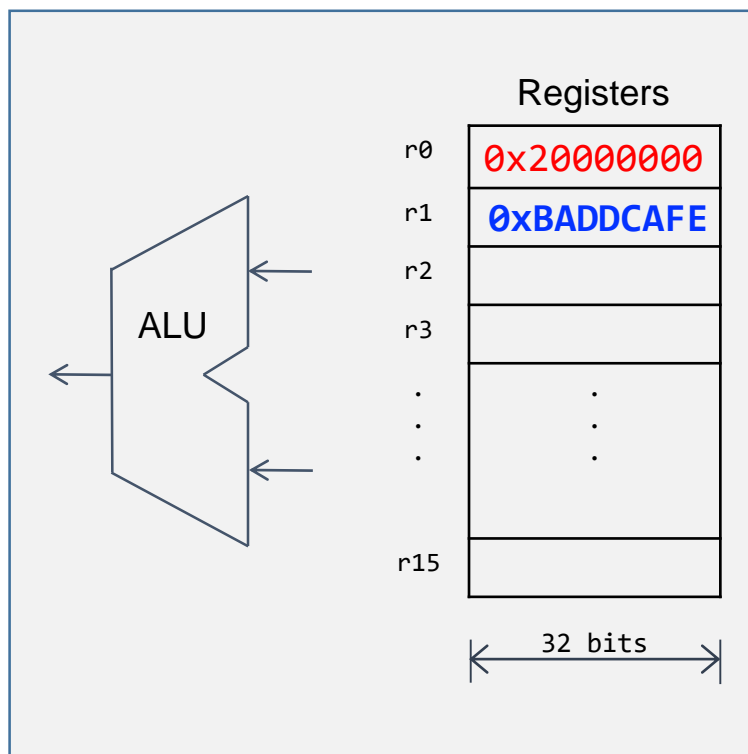
Processor  
Core



Memory

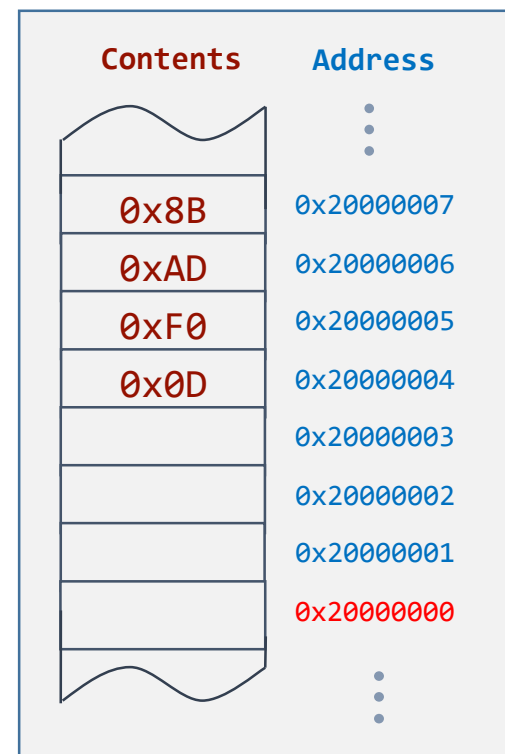
# Store

- Storing Word to Memory
  - `STR r1, [r0]` ; `memory.word[r0] = r1`
  - the data travels from register to memory



Processor  
Core

For Internal Use Only!

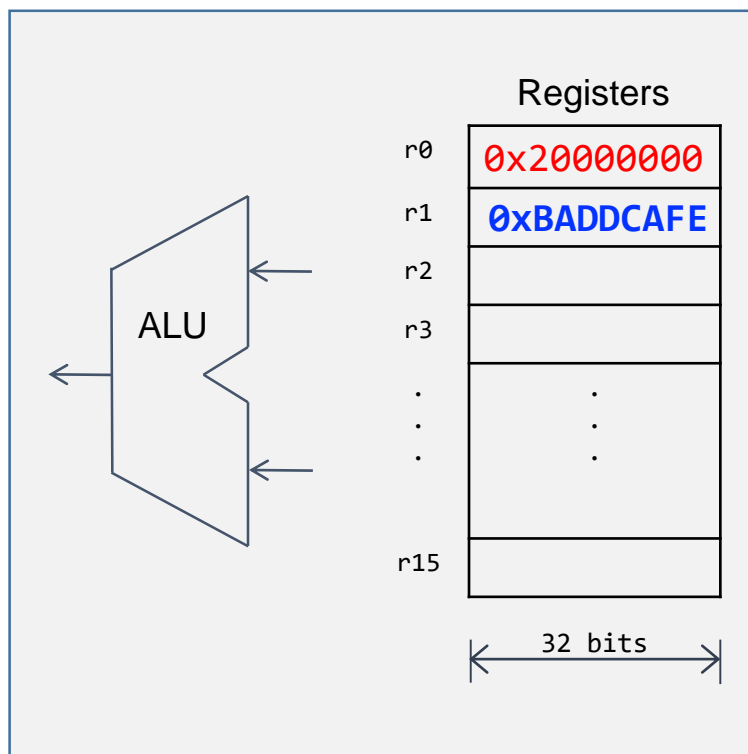


Memory



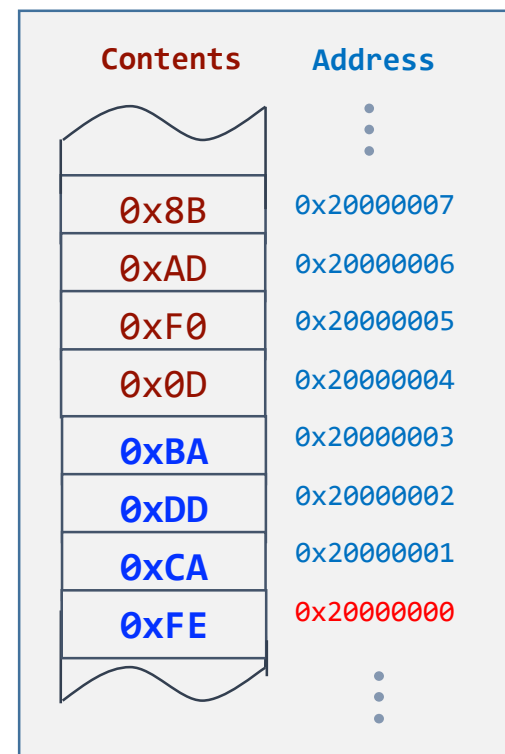
# Store

- Storing Word to Memory
  - `STR r1, [r0]` ; `memory.word[r0] = r1`
  - the data travels from register to memory



Processor  
Core

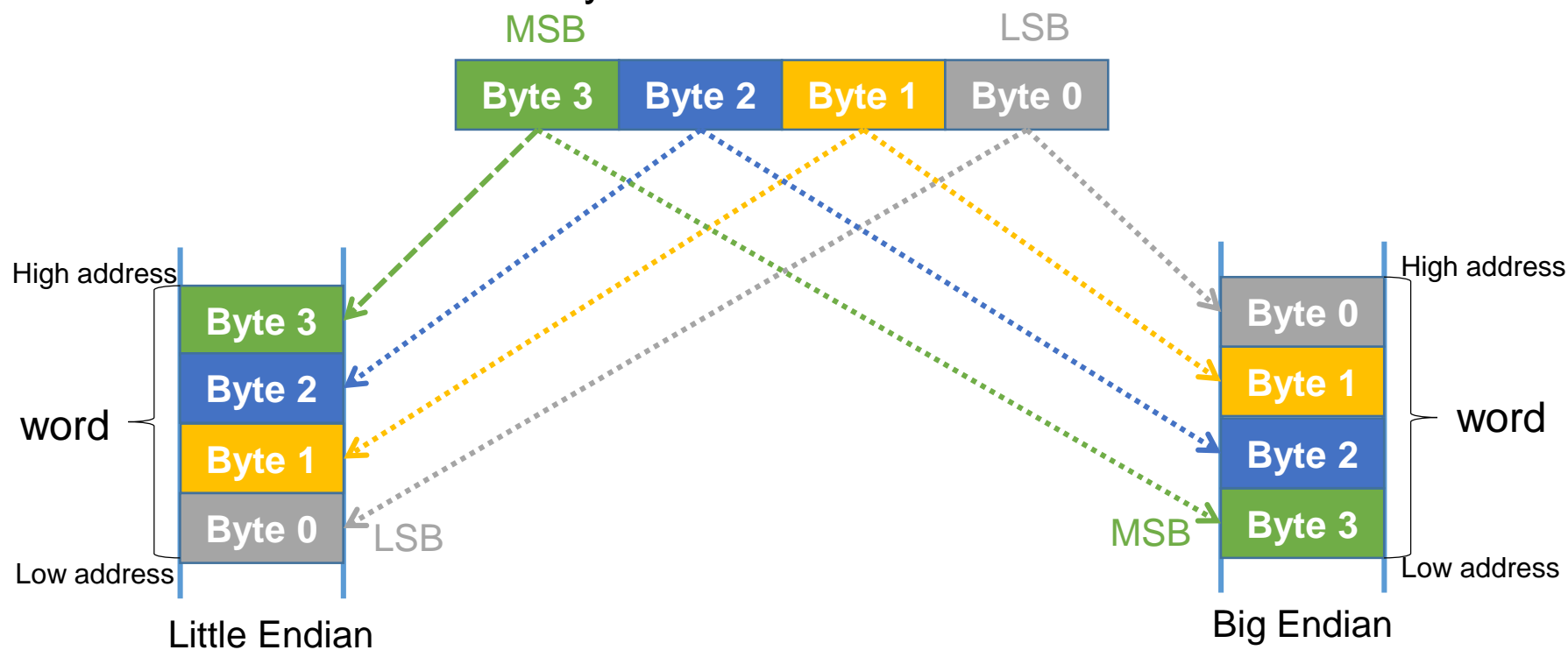
For Internal Use Only!



Memory

# Little Endian vs Big Endian

- Little-endian
  - **LSB** of a word is at **least** memory address
- Big-endian
  - **MSB** of a word is at **least** memory address



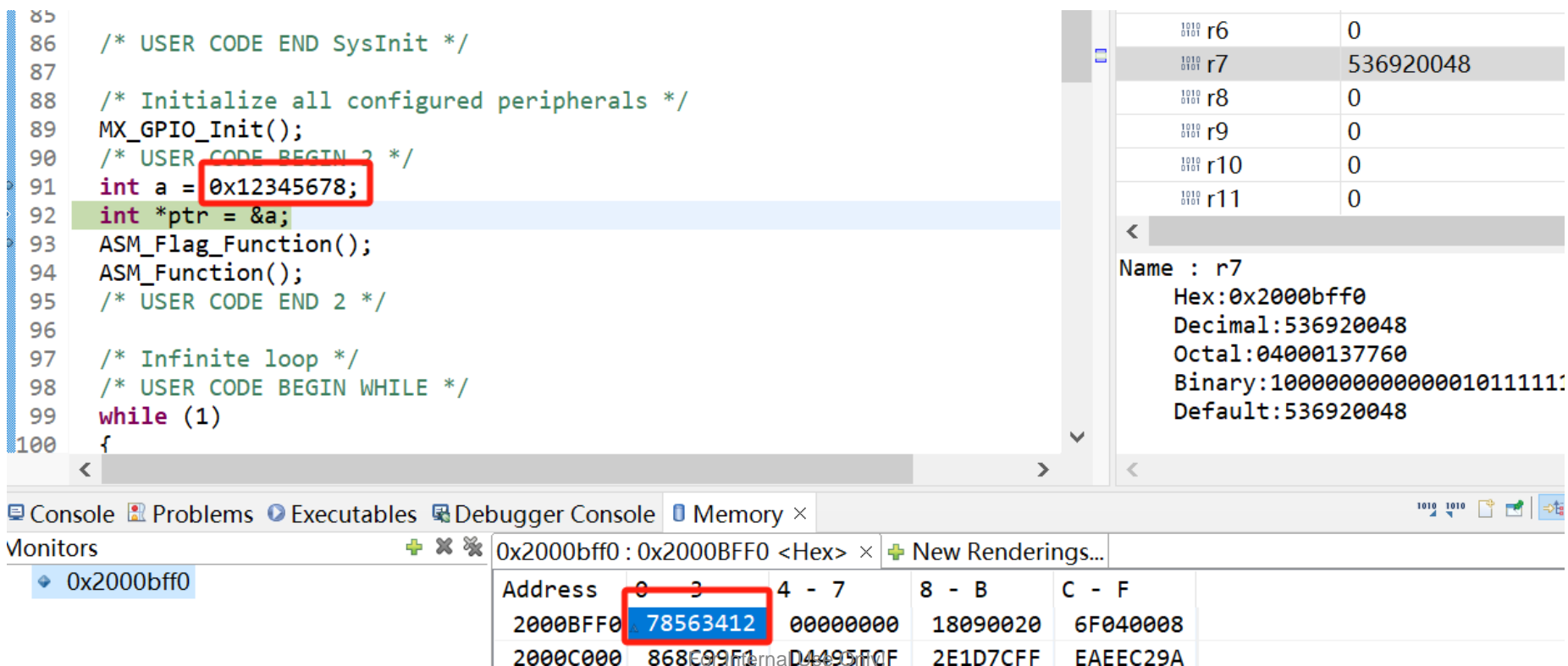
*LSB is at least address!*

For Internal Use Only!

*MSB is at least address!*

# Little endian or Big endian

- Microprocessors can be either 'little endian' or 'big endian'
- The ARM7 processor can be configured as either little endian or big endian.
  - Intel (e.g. the Pentium) uses little endian whereas MIPS uses big endian.
  - **Cortex M** uses little endian by default



```

85
86  /* USER CODE END SysInit */
87
88  /* Initialize all configured peripherals */
89  MX_GPIO_Init();
90  /* USER CODE BEGIN 2 */
91  int a = 0x12345678;
92  int *ptr = &a;
93  ASM_Flag_Function();
94  ASM_Function();
95  /* USER CODE END 2 */
96
97  /* Infinite loop */
98  /* USER CODE BEGIN WHILE */
99  while (1)
100 {
    
```

Debugger Console: Memory

Monitors: 0x2000bff0

Address	0 - 3	4 - 7	8 - B	C - F
2000BFF0	78563412	00000000	18090020	6F040008
2000C000	868C99F0	D4495F0F	2E1D7CFF	EAECE29A

Register r7: 536920048

Register r7 details:

- Name: r7
- Hex: 0x2000bff0
- Decimal: 536920048
- Octal: 04000137760
- Binary: 100000000000001011111
- Default: 536920048

# Addressing Mode

- Register addressing mode
  - MOV R2, R4
  - ADD R3, R2, R1
- Immediate addressing
  - MOV R1, #0x25
  - ADD R6, R6, #0x40
- Register indirect (indexed)
  - STR R5, [R6]
  - LDR R10, [R3]

# Immediate Addressing

- Immediate addressing means that the instruction code contains a value to be used.
- Restrictions
  - The immediate value has to be specified by 12 bits
  - but it does not have to be the least significant byte, and the remaining 4 bits to specify the location of the 8 bits
  - E.g.
  - **MOV r4, #0xFF0**
    - Will put 0x00000FF0 into r4 (the #0xFF0 is the immediate)
  - **ADD r2, r1, #10**
    - add 10 to value in r1 and put the sum in r2
  - **SUB r8, r8, #99**
    - subtract 99 from value in r8

# Valid Immediate

- The instruction code is always 32 bits and it must include information about the type of instruction
  - (e.g. ADD, MOV, EOR, etc.) and the destination register as well as the immediate value.
- So a 32 bit value can not be put into a 32 bit register
  - using immediate addressing and MOV.
  - E.G. `MOV r3, #0xF97D5EC5` is not allowed
- The immediate can be one byte (8 bits) but it does not have to be the least significant byte, and the remaining 4 bits to specify the location of the 8 bits
  - `MOV r11, #0x3FC0000` will put 0x03FC0000 into r11
  - `<immediate>=immed_8 rotate right (2*rotate_imm)`
  - The Immediate in instruction machine code is 0x7FF

# Indirect Addressing

- Base plus offset addressing
- Uses a value in a register (the 'base') plus a binary number (the 'offset') to identify a memory address.
- E.g.
  - `LDR r6, [r11, #12]`
    - means load into r6 the data held in the memory location that has the address given by the value in register (base) r11 added to 12.

# Automatic updating

- In many applications there is a great deal of data movement between the CPU and memory and it can be very useful if the base register is updated on each load or store.
- The instruction:
  - `LDR r6, [r11, #12]!`
    - does the same as the instruction on the previous slide
    - but 12 is added to the value in r11.
    - The automatic updating is identified by the !



# Pre-indexed and post-indexed

- Pre-indexing:
  - `LDR r6, [r11, #12]!`
    - Offset 12 is added to the base register r11, **before** r11 is used as a memory address.
- Post-indexing
  - `LDR r6, [r11], #12`
    - Offset 12 is added to the base register r11, **after** r11 is used for the memory address.

# Example

- What values are held in r4, r7 and r8 and is memory modified or not after the execution of the following? (assume little endian)

<u>Programme:</u>	<u>Memory Address</u>	<u>Contents</u>
MOV r4, #0x8000	0x00008000	0xF5
LDR r7, [r4], #4	0x00008001	0x04
STR r7, [r4], #4	0x00008002	0x4C
LDR r8, [r4, #-4]!	0x00008003	0x82

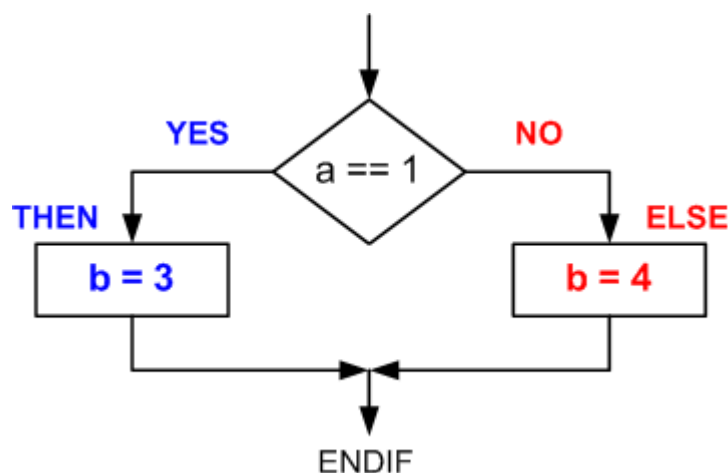
# Branch Instruction

- If-then-else

## C Program

```
if (a == 1)
    b = 3;
else
    b = 4;
```

```
; r1 = a, r2 = b
CMP r1, #1    ; compare a and 1
BNE else     ; go to else if a ≠ 1
then MOV r2, #3 ; b = 3
      B      endif ; go to endif
else MOV r2, #4 ; b = 4
endif
```



**CMP Rn, Op2** (Rn – Op2, Same as SUBS, except result is discarded.)

**B label** (branch to label.)

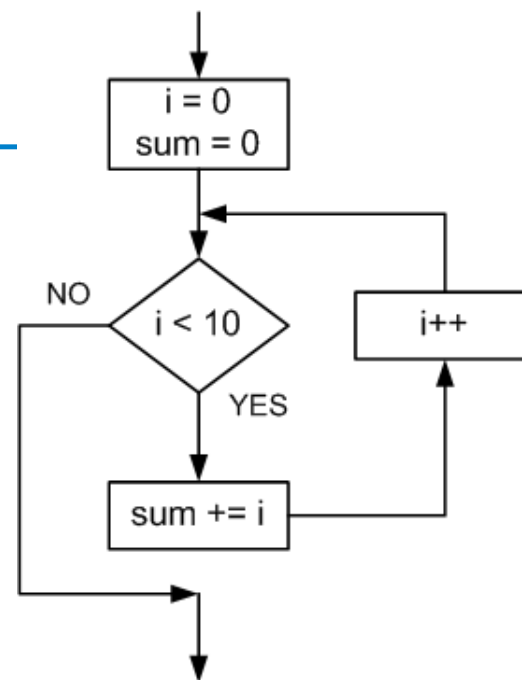
Compare	Signed	Unsigned
>	BGT	BHI
>=	BGE	BHS
<	BLT	BLO
<=	BLE	BLS
==	BEQ	
!=	BNE	

# Branch Instruction

- For Loop

## C Program

```
int i;  
int sum = 0;  
for(i = 0; i < 10; i++){  
    sum += i;  
}
```



```
MOV r0, #0    ; i  
MOV r1, #0    ; sum
```

**B**    **check**

```
loop    ADD r1, r1, r0    ; sum += i  
        ADD r0, r0, #1    ; i++
```

```
check    CMP r0, #10     ; check whether i < 10  
        BLT loop        ; loop if signed less than
```

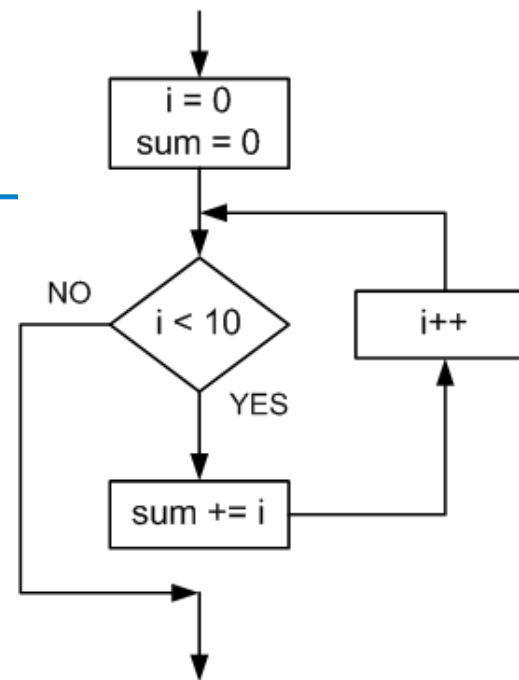
```
endloop
```

# Branch Instruction

- While Loop

## C Program

```
int i;
int sum = 0;
while (i < 10){
    sum += i;
    i++;
}
```



```

MOV r0, #0 ; i
MOV r1, #0 ; sum

loop  CMP r0, #10 ; check whether i < 10
      BGE endloop ; skip if ≥
      ADD r1, r1, r0 ; sum += i
      ADD r0, r0, #1 ; i++
      B loop
endloop
```

# CISC v.s. RISC

- CISC = Complex Instruction Set Computer
  1. Complicated CPU
  2. Each instruction takes longer to execute
  3. Fewer machine code instructions for each high level instruction
  4. Good code density
  5. Smaller semantic gap
  6. Simple compiler
- RISC = Reduced Instruction Set Computer
  1. Simple CPU
  2. Machine code instructions execute quickly
  3. More machine code instructions for each high level instruction
  4. Poor code density
  5. Larger semantic gap
  6. Complicated compiler