CS301
Embedded System and Microcomputer Principle

Lecture 3: ARM Assembly

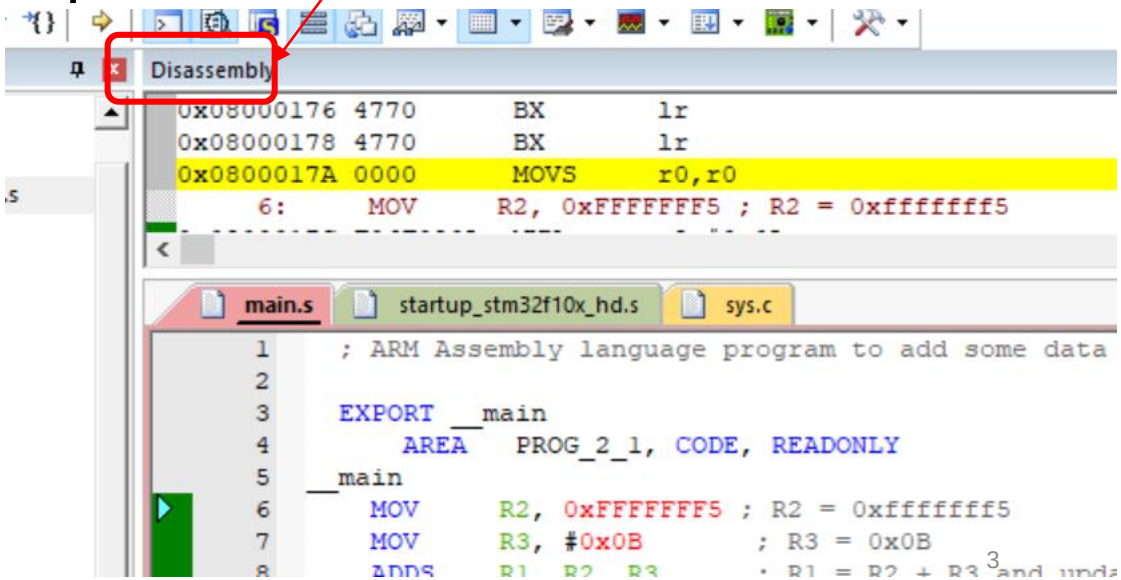
2023 Fall

Recap

- GPIO
  - CPU
    - ALU
    - Registers
      - R1~R12
      - SP, LR, PC
      - PSR (flags)
    - Control unit
  - GPIO
    - Memory mapped IO
      - CPU treats peripherals registers the same as data memory
    - GPIO Mode
      - default: floating input
      - input: pull up/pull down
      - output: push-pull
    - Registers
      - 1 port -> 16 pins -> 7 registers
  - Programming GPIO
    - 1.Enable GPIO Clock
      - RCC->APB2ENR
    - 2. Configure GPIO mode
      - CRL (pin0~pin7)
      - CRH (pin8~pin15)
      - ODR (if input, 1: pull up, 0: pull down
    - 3a. Output value setting
      - ODR
    - 3b. Input value reading
      - IDR

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		Machine Code		in Memory
MOV r12, #114		0xE3A0C072		0X00008000
MOV r7, #0xCB	assembler →	0xE3A070CB	linker →	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(助记符)
  - 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

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

- Condition Flags: NZCV
  - **Negative** bit
    - `N = 1` if most significant bit of result is `1`
  - **Zero** bit
    - `Z = 1` if all bits of result are `0`
  - **Carry** 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
  - **oVerflow** 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
    ---------------
        [1] 0 1 0 1 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
    -----------------------------
              1   0   1   1   0
```

$= (10)_2$

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

Overflow

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

```
   0 1 1 0 0      12
+  0 0 1 0 1   +   5
-----------   -------
   1 0 0 0 1     -15
```

5-bit result

```
      1 0 0 1 1     -13
+     1 1 0 0 1   + -7
-----------------  -----
  [1] 0 1 1 0 0      12
```

Extra bit is discarded.

5-bit result

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

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

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
0100 + 0010	0110		
0100 + 0110	1010		
1100 + 1110	1010		
1100 + 1010	0110		

Exercise

- Assume a four-bit system unsigned and signed operations

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

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
  - ADDS 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	⟶	ADDS
SUB	⟶	SUBS
MUL	⟶	MULS
AND	⟶	ANDS
ORR	⟶	ORRS
LSL	⟶	LSLS
MOV	⟶	MOVS

Example

```
MOV     R2, #0xFFFFFFF5 ; R2 = 0xfffffff5
MOV     R3, #0x0B       ; R3 = 0x0B
ADDS    R1, R2, R3      ; R1 = R2 + R3 and update the flags
```

```
  0xFFFFFFF5      1111 1111 1111 1111 1111 1111 1111 0101
+ 0x0000000B    + 0000 0000 0000 0000 0000 0000 0000 1011
 0x100000000    1 0000 0000 0000 0000 0000 0000 0000 0000
```

All zeros

C=1

Z=1

- 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        0111 1111 1111 1111 1111 1111 1111 1111
+ 0x00000002      + 0000 0000 0000 0000 0000 0000 0000 0010
  0x80000001        1000 0000 0000 0000 0000 0000 0000 0001
```

N=1 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:

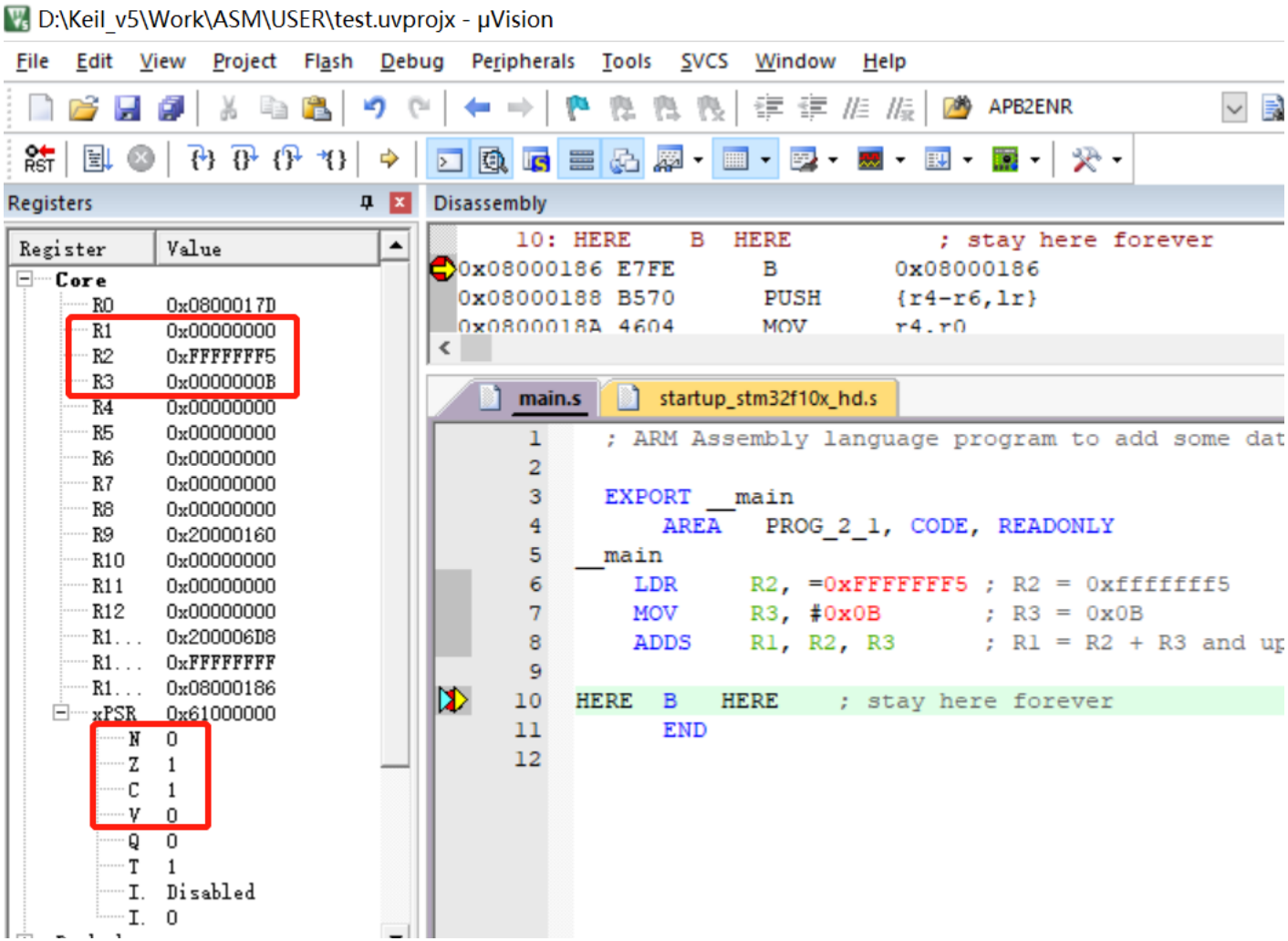
```
MOV     R6, #0x38    ;R6 = 0x38
MOV     R7, #0x2F    ;R17 = 0x2F
ADDS    R6, R6,R7    ;add R7 to R6
```

- 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



Instructions using logic

- AND r0, r1, r2 ; Bitwise AND, r0 = r1 AND r2
  - clear a specific bit(s) of a byte
- ORR r0, r1, r2 ; Bitwise OR, r0 = r1 OR r2
  - set a specific bit(s) of a byte
- EOR r0, r1, r2 ; Bitwise Exclusive OR, r0 = r1 EOR r2
  - toggle a specific bit(s) of a byte
- BIC r0, r1, r2 ; Bit clear, r0 = r1 AND ~r2

	0x35	0	0	1	1	0	1	0	1
AND	0x0F	0	0	0	0	1	1	1	1
	0x05	0	0	0	0	0	1	0	1

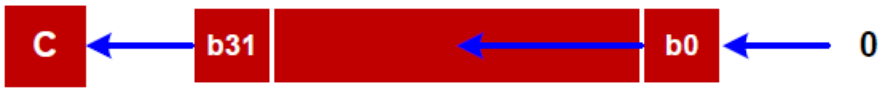
	0x44	0	1	0	0	0	1	0	0
EOR	0x06	0	0	0	0	0	1	1	0
	0x34	0	1	0	0	0	0	1	0

	0x04	0	0	0	0	0	1	0	0
ORR	0x30	0	0	1	1	0	0	0	0
	0x34	0	0	1	1	0	1	0	0

	0xFE	1	1	1	1	1	1	1	0
BIC	0x11	0	0	0	1	0	0	0	1
	0xEE	1	1	1	0	1	1	1	0

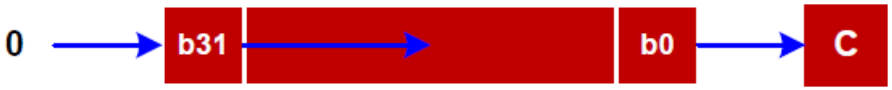
Instructions using logic

- 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

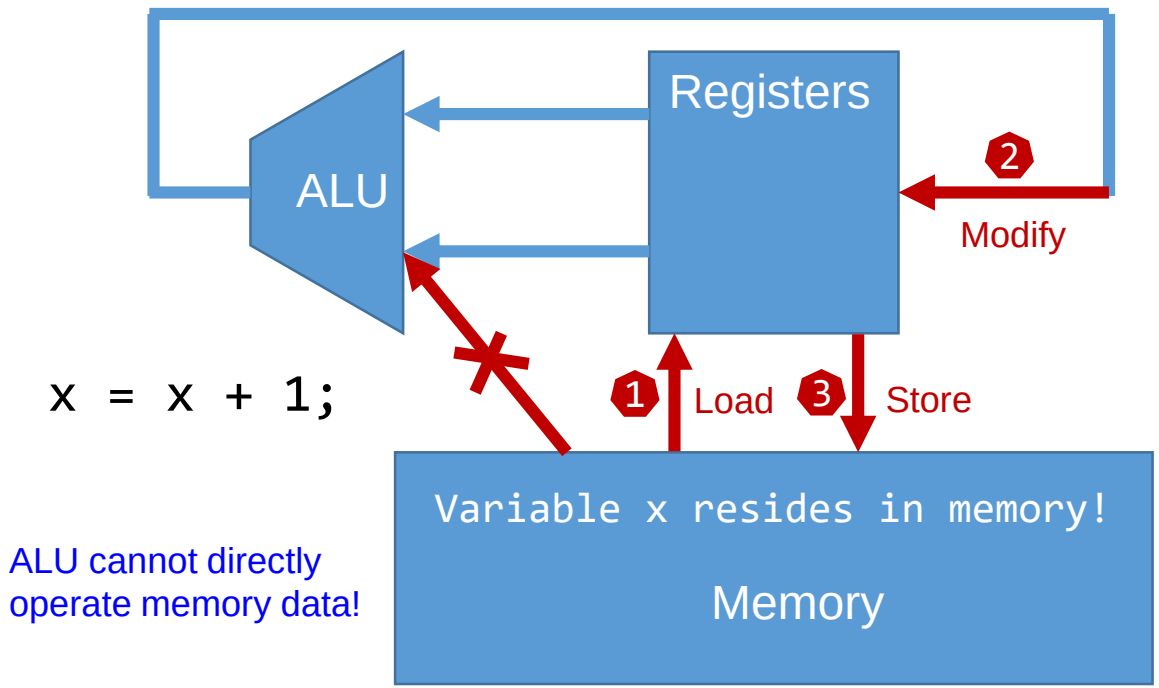
If rotate left by 12 bits

```
; 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

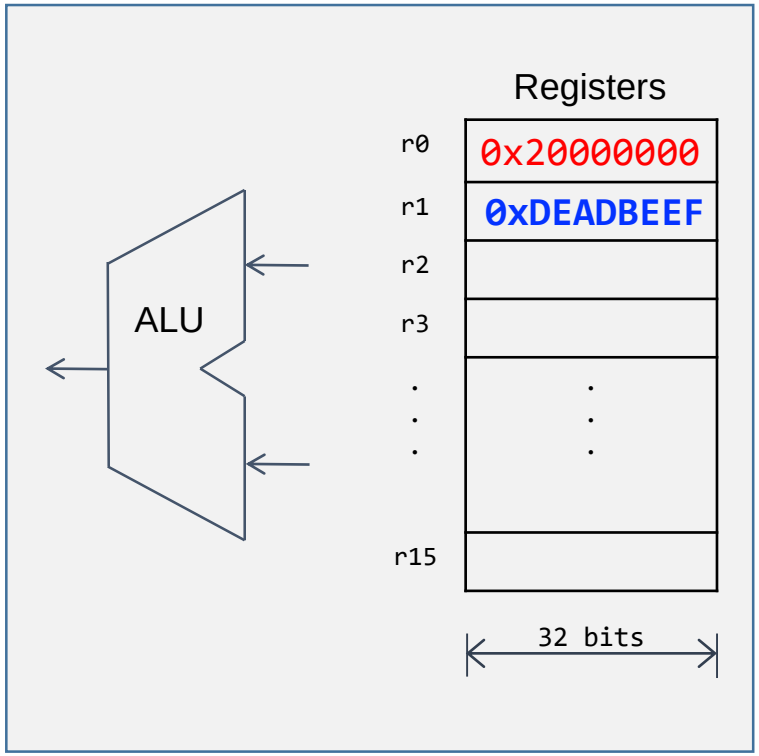
Data Transfer Instructions

- 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]

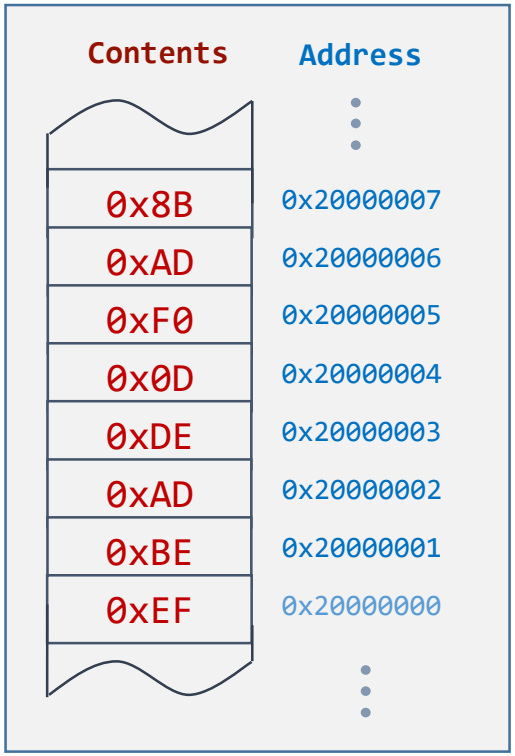


Load

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



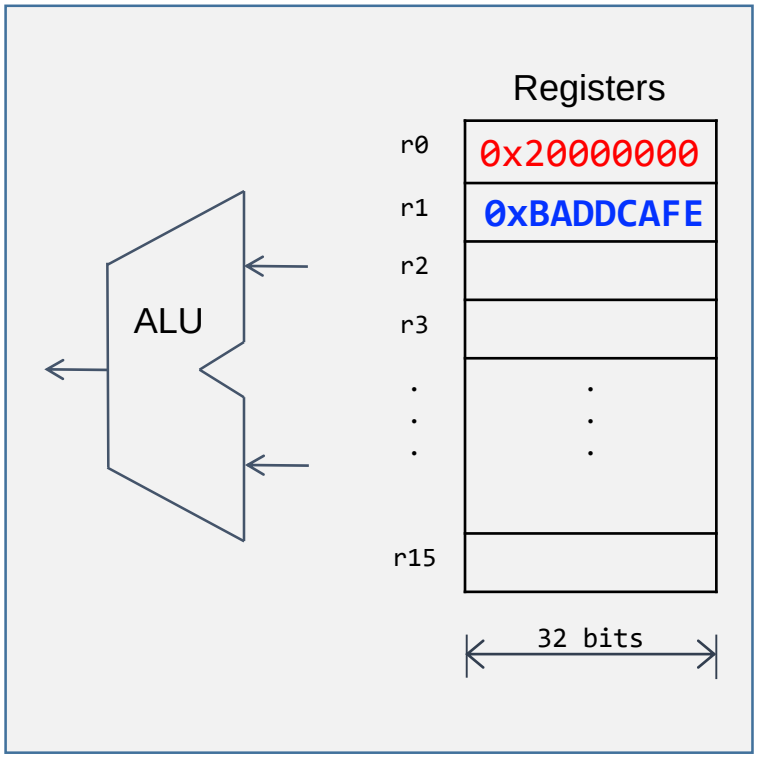
Processor Core



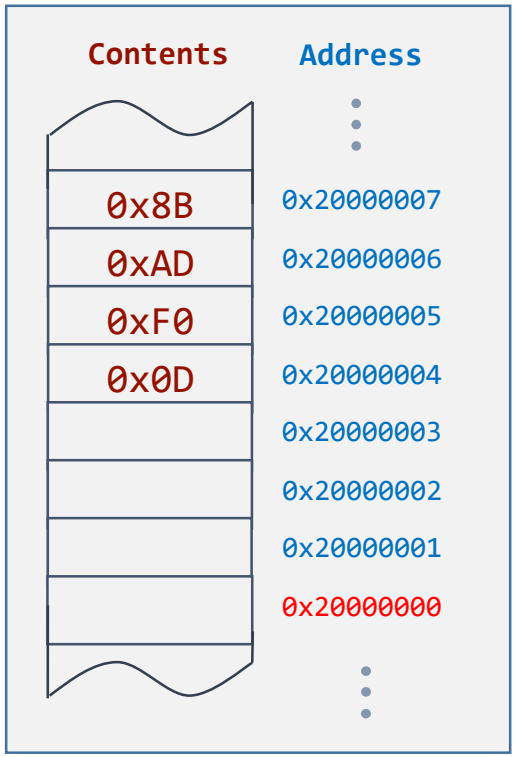
Memory

Store

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



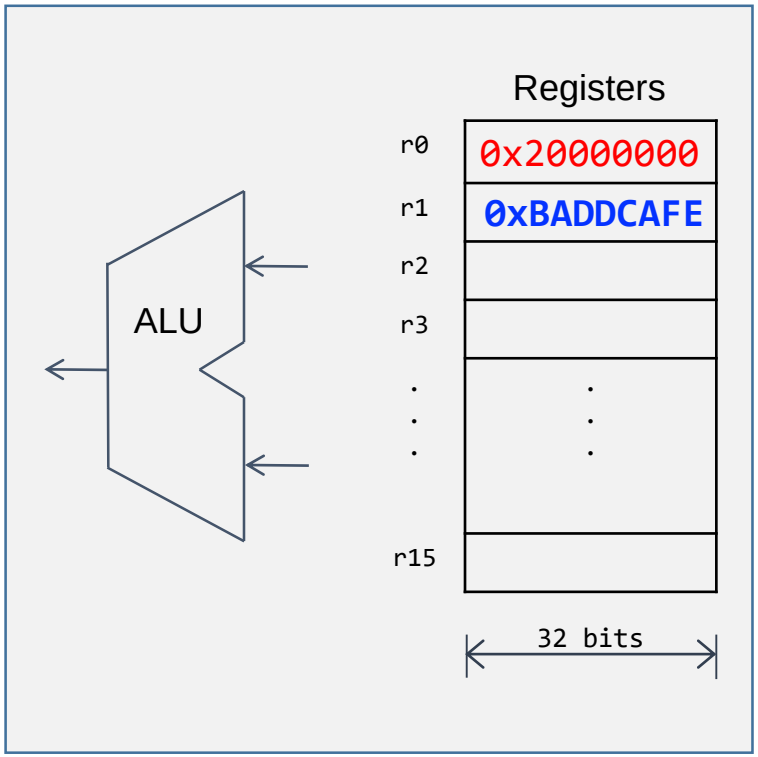
Processor Core



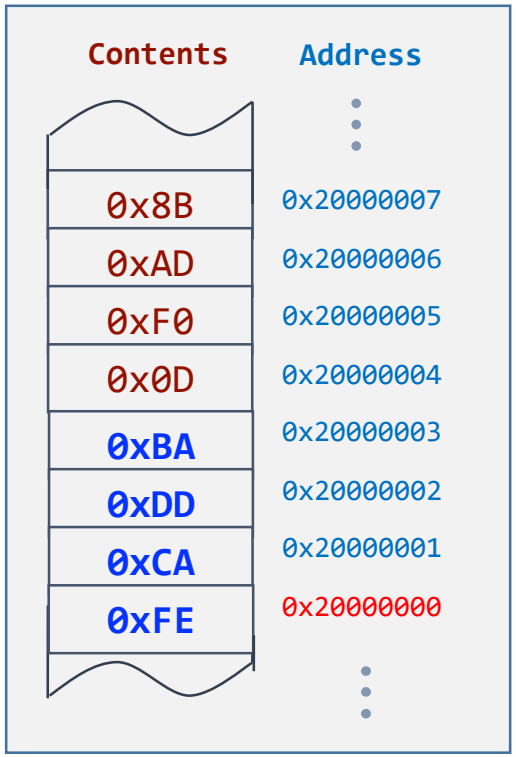
Memory

Store

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



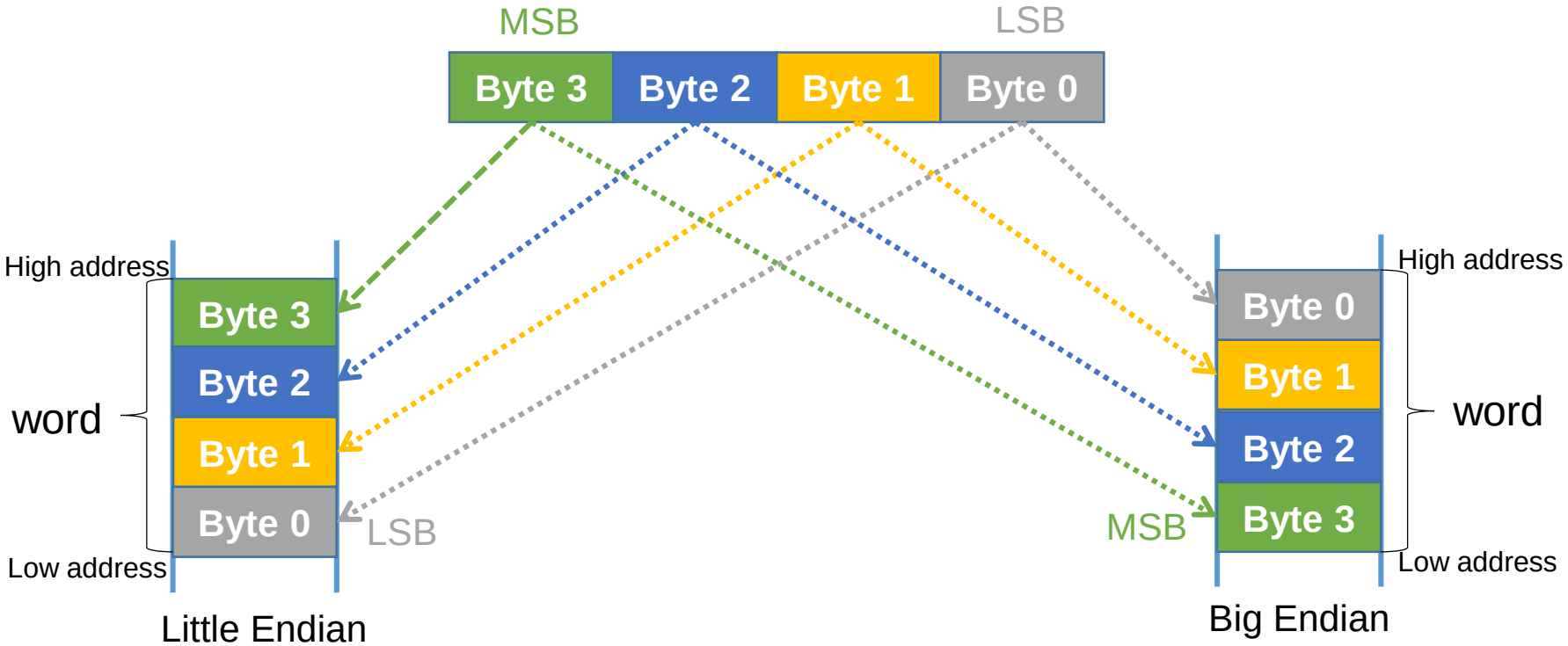
Processor Core



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!

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

```
12:                 LDR         R5, [R11]
13:
14:
0x08000188 F8DB5000  LDR        r5,[r11,#0x00]
```

Memory 1

Address: 0x0800018A

```
0x0800018A: 00 50 FE E7 00 00 70
0x0800019F: 00 CE 17 01 EB 96 76
0x080001B4: 43 F0 01 03 1A 4E B3
```

Call Stack + Locals | Memory 1

Registers

Register	Value
	0x0800017D
	0x00000000
	0xFFFFFFF5
	0x0000000B
R4	0x00000000
R5	0xE7FE5000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x0800018A
R12	0x00000000
R13 (SP)	0x200006D8
R14 (LR)	0xFFFFFFFF

Addressing Mode

- Immediate
  - MOV R1, #0x25
  - ADD R6, R6, #0x40

- Register addressing mode
  - MOV R2, R4
  - ADD R3, R2, R1

- 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 and (0…0 1111 1111 0000)
  - MOV r11, #0x3FC0000 (0011 1111 1100 …..)
    - Will put 0x03FC0000 into r11. (0011 1111 1100 0….0)

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 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 **!**, ‘pling’.

Pre-indexed and post-indexed

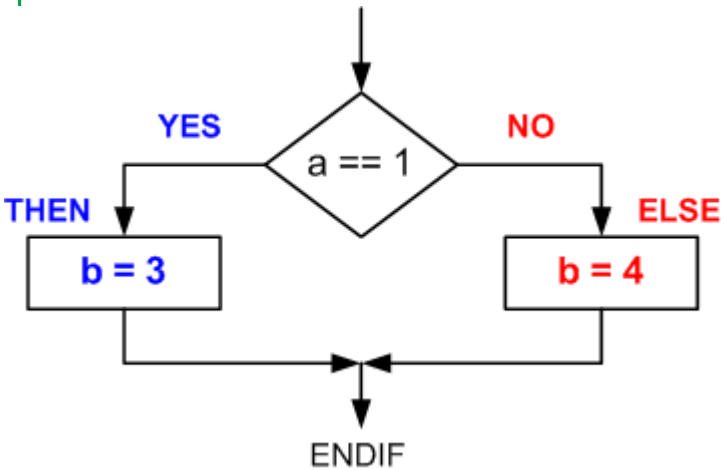
- Pre-indexing:
  - LDR r6, [r11, #12]!
  - Offset 12 is added to the base register r11, before r11 is used as a memory address.
- Post-indexing
  - LDR r6, [r11], #12
  - Offset 12 is added to the base register r11, after r11 is used for the memory address.
- There is no pling, !, for post-indexing because the base register is always updated.

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.)

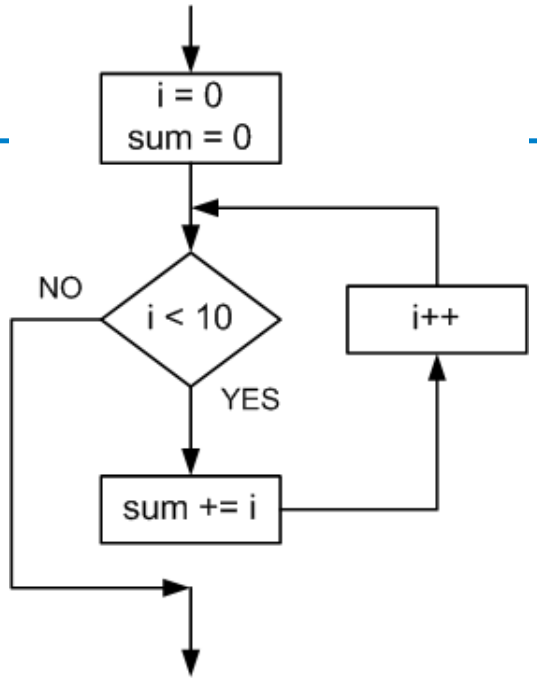
Compare	Signed	Unsigned
>	BGT	BHI
>=	BGE	BHS
<	BLT	BLO
<=	BLE	BLS
==	BEQ	
!=	BNE	

Branch Instruction

- For Loop

C Program

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



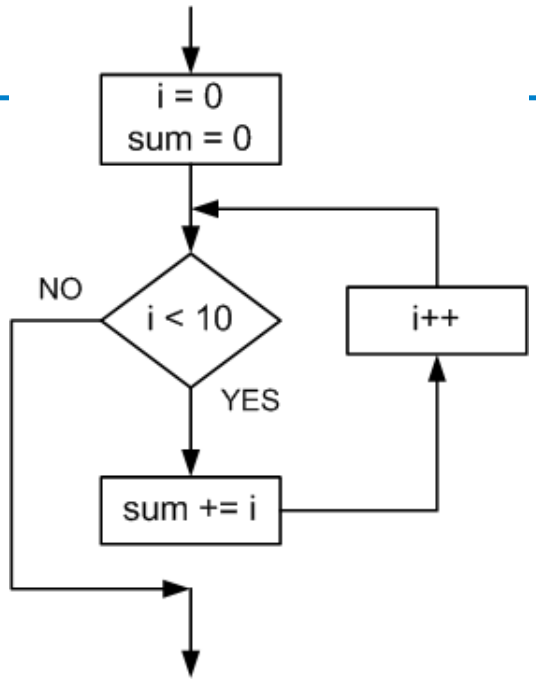

```
        MOV r0, #0   ; i
        MOV r1, #0   ; sum

        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
```