Lecture 2: Assembler Directives and Data Processing Instructions

ELEC 2142 Embedded System Design
2018 Session 2 - Week 2

Outline

Assembler Directives

Recall, structure of a source line in your assembly code,

[label] [instruction/directive/pseudo-instruction] [;comments]

A couple of things to note,

- Instructions are actual ARM instructions.
- Directives are instructions for the assembler (i.e. not part of the machine instructions).
- Pseudo-instructions are "fake" instructions used to make things easier for the programmer, and are converted to actual ARM instructions during the time of assembly
- Labels are memory addresses that you can use as pointers to various locations in your program to do things like branching and sub-routine calls. Labels are mapped into actual memory addresses by the linker.

Assembler Directives

- Constants in an ARM assembly program
  - Numeric constants
    - If you don't specify - it is decimal - 255
    - Hexadecimal - 0x255
    - Any other base - n_xxxx - n is the base (from 2 to 9), xxxx is the number. E.g. 2_10101010
  - Character constants - 'A'
  - String constants - "Hello World". Note: string constants are not null-terminated.
- Examples: MOV R0,#2_10101101
  MOV R2,#'B'
- Directives, register names, assembly mnemonics are not case sensitive. However, you can not mix uppercase and lowercase.
  ADD R1, R2, R3 ; permitted
  Add R1, r2, R3 ; not permitted
- Don't forget the preceding white space for each line of code.

Assembler Directives - AREA

- Defines a block of data or code within your source file.
- Syntax:

  AREA <section name> {,attr} {,attr}...
- Section name is the name of the data or code section. If the section name starts with a digit, it has to be enclosed in bars. E.g. |1_test|
- Optional attributes can be
  - CODE: defines a section of code. Typically declared as read-only.
  - DATA: defines a section of data. Can be declared as read-write.
  - READONLY: declares that the section cannot modified.
  - READWRITE: declares that the section can be modified.
  - ALIGN=expr: aligns the section into a 2^{expr}-byte boundary.

Assembler Directives - RN

- Renames a register.
- Syntax:

```
name    RN    expr
```

- name: custom name you want to assign to a register
- expr: evaluates to a number from 0 to 15
- Example

```
alpha    RN    5; register R5 is renamed as alpha
coeff1   RN    0; register R0 is renamed as coeff1
```

Assembler Directives - EQU

- Equates a symbol to a numeric constant. Useful to give meaningful names to numeric constants used in the program. Similar to #define in C.
- Syntax:

```
name      EQU      expr {,type}
```

- name: symbolic name assigned to the constant
- expr could be
  - a 32-bit user defined constant
  - an absolute memory address
  - a relative memory address
- type: optional attribute. can be ARM, THUMB, CODE16, CODE32, DATA.
- Examples:

```
SRAM_BASE      EQU      0x40000000
table      EQU      label+8
```

; assigns the program relative 32-bit address label+8 to symbol table

Assembler Directives - ENTRY

- Declares an entry point to the program.
- Must have at least one ENTRY.
- In projects with multiple source files, each source will have an ENTRY directive.
- A single source file should not have more than one ENTRY directive.
- Example:

```
AREA     testprog, CODE, READONLY
ENTRY
```

Assembler Directives - DCB, DCW, DCD

These directives are used to declare initial run-time content in the memory at byte, halfword, and word level.

DCB

- Allocates and define run-time memory content at byte level.
- Syntax:

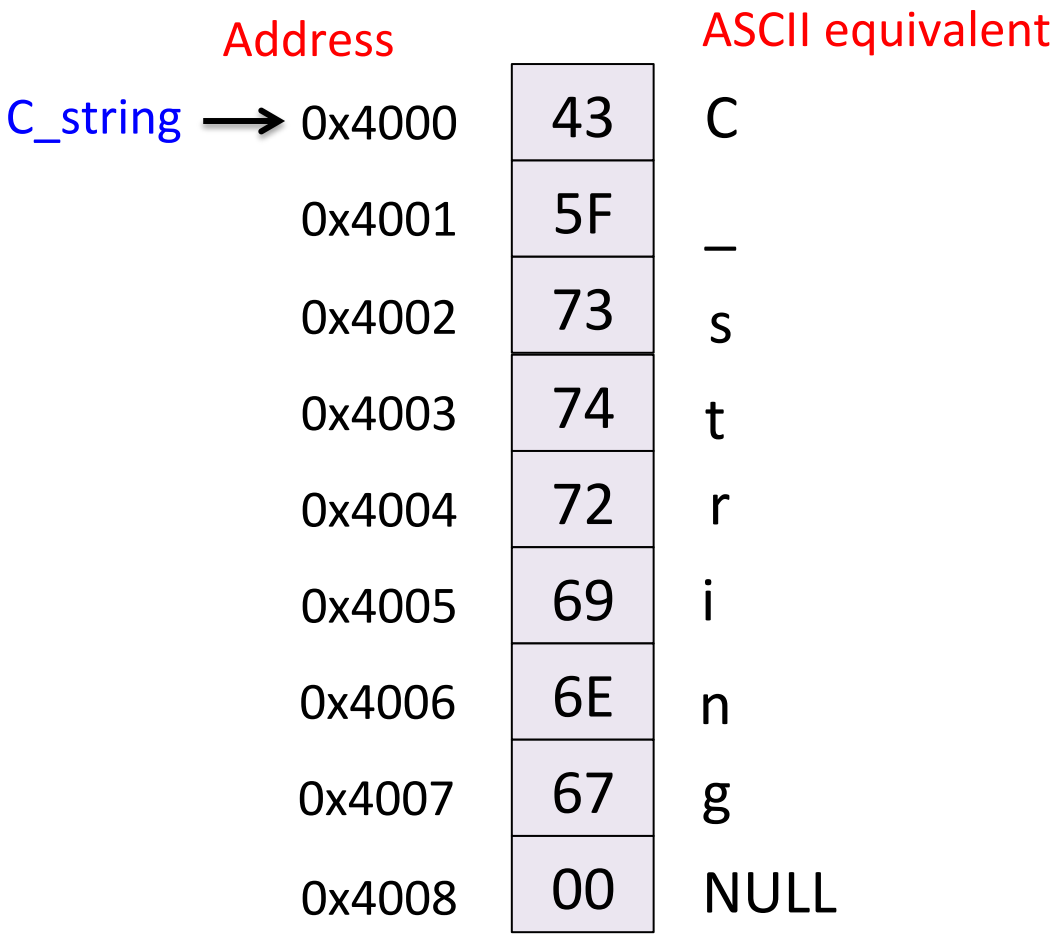
```
{label}    DCB    expr {,expr}...
```

- label: memory address pointing to the starting byte declared by DCB.
- expr: either a numeric expression that evaluates to an integer in the range -128 to 255 or a string constant. For a string constant the individual characters are stored in consecutive locations in the memory.
- DCB operates at byte level. Therefore, explicit memory alignment may be needed using the ALIGN directive.

Assembler Directives - DCB, DCW, DCD

DCB example:

C_string DCB "C_string", 0; null terminated string constant



Assembler Directives - DCB, DCW, DCD

DCW

- Defines run-time memory content at halfword level.
- Syntax:

```
{label}    DCW{U}    expr {,expr}...
```

- label: memory address pointing to the starting byte of the halfword declared by DCW.
- Optional U is to discard alignment. Otherwise aligned to 2-byte boundaries.
- expr: is a numeric expression that evaluates to an integer in the range -32768 to 65535.

Example:

```
coeff    DCW    0xFE33, 0xD3E5; defines two halfwords
```

Assembler Directives - DCB, DCW, DCD

DCD

- Defines run-time memory content at word level.
- Syntax:

  {label} DCD{U} expr {,expr}...

- label: memory address pointing to the starting byte of the word declared by DCD.
- Optional U is to discard alignment. Otherwise aligned to 4-byte boundaries. That is, DCD might insert upto 3 bytes of padding.
- expr: is a numeric expression or program relative address.
- Example:

```
coeff1  DCD  1,5,20
data1   DCD  data2+4
```

Assembler Directives - DCB, DCW, DCD

Example:

```
        DCB     255
        DCD     1,5
```

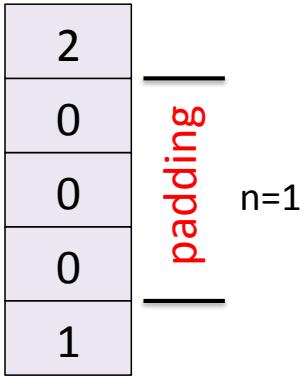
Example:

```
        DCB     255
        DCDU      1,5
```

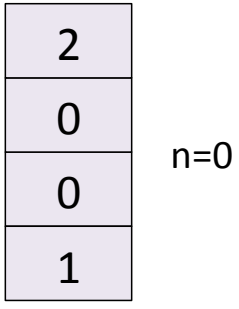
Assembler Directives - ALIGN

- Used to align the current location in memory to a specified boundary by padding with zeros.
- Syntax: ALIGN {expr{,offset}}
- expr: numeric expression evaluating to any power of two from 2^0 to 2^{31}.
- offset: any numeric expression
- Current memory location is aligned to the next lowest address of the form offset $+ n \times$ expr, where $n \in \mathbb{Z}$ selected to minimize padding.

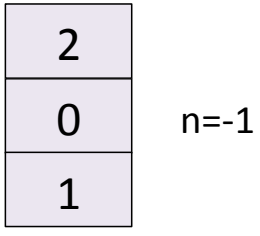
```
DCB     1
ALIGN   4
DCB     2
```




```
DCB     1
ALIGN   4,3
DCB     2
```




```
DCB     1
ALIGN   4,6
DCB     2
```



Assembler Directives - ALIGN

ALIGN (with no options) is same as ALIGN 4

```
                AREA Example, CODE, READONLY
start           ADR  r6, label1
                MOV  pc, lr
label1          DCB  1              ; pc now misaligned
                ALIGN               ;ensures that
                                    ;subroutine1 addresses
                                    ;the following instruction
subroutine1
                MOV r5, #0x5
```

Assembler Directives - SPACE & END

- SPACE allocates a block of memory initialized to zero.
- Syntax:

  {label} SPACE {expr}

- expr: specifies the number of bytes to reserved as zero.
- ALIGN directive may be used to align any code following the SPACE directive.
- Example:

```
data    SPACE    255
```

- END declares the end of the source file

Assembler Directives - IMPORT/EXTERN

- Informs the assembler/linker that a label is not defined within the current file, but defined in some other file within the project. Correct address for the label is resolved by the linker when combining multiple modules.
- Syntax:

IMPORT	{label}
EXTERN	{label}

```
AREA file1, CODE, READONLY
IMPORT display
.
.
BL display
```

Assembler Directives - EXPORT/GLOBAL

- Informs the assembler/linker that a label defined as EXPORT/GLOBAL is available (visible) to other files within the project.
- Syntax:

EXPORT	{label}
GLOBAL	{label}

```
                AREA  file2, CODE, READONLY
                EXPORT  display
                  .
display           .
                  .
                END
```

Assembler Operators

- Again, not part of actual ARM instructions.
- These are primitive operations (such as arithmetic and logical) that can be performed on data (operands) within an instruction, so that the particular selection of operands makes more sense to you as the programmer.

```
A:ROL:B  ; Rotate A left by B bits
A:ROR:B  ; Rotate A right by B bits
A:SHL:B  ; Shift A left by B bits
A:SHR:B  ; Shift A right by B bits
A:AND:B  ; Bitwise AND of A and
         ; B
A:OR:B   ; Bitwise OR of A and B
A:EOR:B  ; Bitwise Exclusive OR of
         ; A and B
A + B    ;Add A to B
A-B      ;Subtract B from A
```

```
EXAMPLE:
ORR r1,r1, #1:SHL:3 ; a 1
;is shifted left three bits.
;Set bit 3 to 1 rather than ;0x8

MOV r0,
#((1:SHL:14):OR:(1:SHL:12)) ;
;setting bit 12 and 14 to 1.
;similar to #0x5000
```

ARM Data Processing Instruction

- ARM instructions
  - Data processing - MOV, ADD, SUB, ADC, RSB, RSC, MUL,...
  - Data transfer - LDR, STR
  - Control flow - B, BL
  - Special - SWI, MRS, MSR,...
- Data processing instructions contain
  - Arithmetic operations - ADD, SUB, ADC,...
  - Comparison - CMP, CMN, TST, TEQ
  - Logical operations - AND, EOR, ORR, BIC,...
  - Data movement - MOV, MVN
- 4-field instruction format

  <opcode> < destination > <source1 > <source2>
- opcode: name of the operation
- destination: destination operand (register)
- source 1: source operand 1 - must be a register
- source 2: source operand 2 - register/shifted register/an immediate

ARM Data Processing Instruction

- Source 2 - shifted register operand - shift amount can be
  - Immediate value - 5 bit unsigned number
    ADD a1, v1, v3, lsl #3
  - Specified in the least significant byte of another register
    ADD a1, v1, v3, lsl v4
- Source 2 - immediate operand
  - 8-bit constant
  - or a constant that can be generated by rotating an 8-bit number to the right by an even number of bits
    add a1, v1, #5 - permitted
    add a1, v1, #0xFE00 - permitted
    add a1, v1, #0xFEA3 - not permitted

Arithmetic Instructions - Addition/Subtraction

Assume variables A, B, C correspond to registers V1, V2, V3, respectively.

$C = A + B$

```
ADD  V3, V1, V2;
```

$C = A + 2 \times B$

```
ADD  V3, V1, V2, LSL #1;
```

$C = A - B$

```
SUB  V3, V1, V2;
```

$C = A - 8 \times B$

```
SUB  V3, V1, V2, LSL #3;
```

$C = A + 10$

```
ADD  V3, V1, #10;
```

Arithmetic Instructions - Addition/Subtraction

Assume variables A, B, C, D, E correspond to registers V1, V2, V3, V4, V5 respectively.

$E = (A + B) - (C + D)$

```
ADD  V5, V1, V2;
ADD  A1, V3, V4; use intermediate register A1
SUB  V5, V5, A1;
```

$E = A + B + C - 2 \times D$

```
ADD  V5, V1, V2;
ADD  V5, V5, V3;
SUB  V5, V5, V4, LSL #1;
```

Use ADDS, SUBS if you want to save the condition code flags.

$E = -A = 0 - A$ - use reverse subtraction RSB

```
RSB  V5, V1, #0;
```

Arithmetic Instructions - Addition/Subtraction

Addition/subtraction with carry.

- Carry bit can be included in the computation using ADC (add with carry), SBC (subtraction with carry), and RSC (reverse subtraction with carry).
- This can be useful in performing 64-bit arithmetic.

ADC V1, V2, V3; V1 = V2 + V3 + C

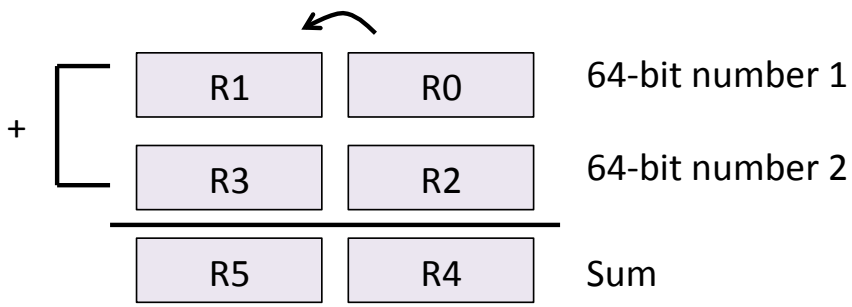
SBC V1, V2, V3; V1 =V2 - V3 + C - 1

RSC V1, V2, V3; V1 = V3 - V2 + C - 1

- Carry flag should be saved in CPSR by appending S

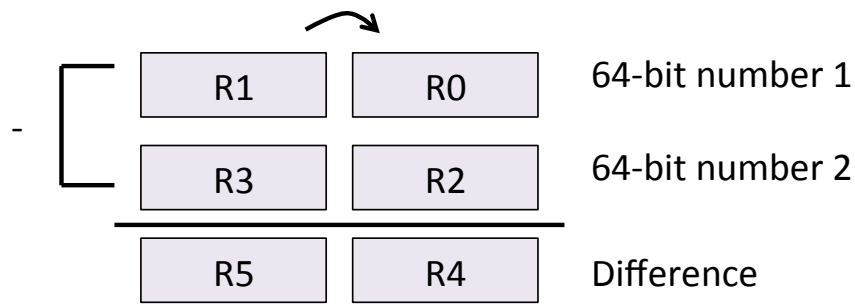
Arithmetic Instructions - Addition/Subtraction

64-bit addition example.




```
ADDS  R4, R0, R2
ADC   R5, R1, R3
```

64-bit subtraction example.




```
SUBS  R4, R0, R2
SBC   R5, R1, R3
```

Data Move Instructions

- Addition with zero can be used to move data between registers.

  ADD V1, V2, #0; V1 $\leftarrow$ V2

- Alternatively, MOV instruction can be used.

  MOV V1, V2; V1 $\leftarrow$ V2

- The second operand can be an immediate (8-bit constraint), register, or a shifted register.
- MOVS updates the condition code flags while moving data.
- A no-operation (nop) is equivalent to something like

  MOV V1, V1.

Q. If register V1 contains variable x, compute $|x|$.

```
MOVS V1, V1
RSBLT V1, V1, #0; V1=0-V1 if V1 < 0
```

Data Move Instructions

MVN (move negative) moves the complement of the source operand into the destination register.

```
MOV  V1,#A3; V1 = 0x000000A3
MVN  V2,V1; V2 = 0xFFFFFF5C
```

MVN can be used to load -1 into a register.

```
MVN  V1, #0; V1 = 0xFFFFFFFF = -1
```

Logic Instruction

- Logic instructions perform bit-wise logical operations associated with registers.
- AND, ORR, EOR, BIC
- <opcode> <destination> <source 1> <source 2>
- opcode: name of the operation
- destination: destination operand (register)
- source 1: source operand 1 - must be a register
- source 2: source operand 2 - register/shifted register/an immediate

```
AND  V1, V2, V3
AND  V1, V2, V3, LSL #2
AND  V1, V2, V3, LSL A1
AND  V1, V2, #0xF3
```

Logic Instruction - AND

- AND V1, V2, V3 $\Rightarrow$ V1 = V2 AND V3
- Recall,

$$A \quad \text{AND} \quad 0 = 0$$
$$A \quad \text{AND} \quad 1 = A$$

- Bit-wise AND operation can be used to create a bit-mask. That is, by performing AND with a specified bit pattern one can extract certain bits in a register, making other bits 0.

Bit masking example

If register V1 contains

1011 0110 1011 1001 1010 1110 0110 1010

and you want to extract bits 4-11, we can use the following mask

0000 0000 0000 0000 0000 1111 1111 0000

AND V1, V1, #0xFF0

Logic Instruction - BIC

- Bit clear (BIC) is used to set certain bits to zero (and leaving other bits unchanged) in a register.
- BIC V1, V2, V3 $\Rightarrow$ V1 = V2 AND (NOT V3)
- Bits of source 1 operand (V2) in places where source 2 operand (V3) has ones will be cleared. Other bits of source 1 operand are not changed.

Bit masking example

Suppose V1 contains

1010 1100 1010 0001 1000 0011 1100 1110

and you want to isolate the upper 24 bits of V1. You can do a bit clear with the following mask

0000 0000 0000 0000 0000 0000 1111 1111

```
BIC  V1, V1, #0xFF
```

Logic Instruction - BIC

Q.

If V1 contains 0x200D00 and V2 contains 0x83000, find the value in V1 after executing BIC V1, V1, V2.

Logic Instruction - ORR/EOR

- ORR - bit-wise logical OR operation
- EOR - bit-wise logical XOR operation
- Recall,

$$A \text{ OR } 0 = A$$
$$A \text{ OR } 1 = 1$$
$$A \text{ XOR } 1 = \text{NOT } A$$
$$A \text{ XOR } 0 = A$$
$$A \text{ XOR } A = 0$$

```
ORR V1, V1, #0xFF
ORR V1, V1, V2
ORR V1, V1, V2, LSL #2
ORR V1, V1, V2, LSL V3
```

Shifts and Rotate Options

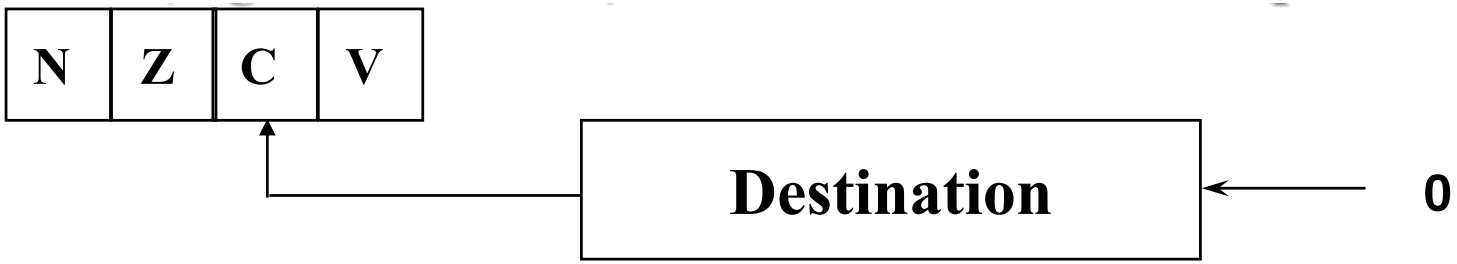
ARM shift and rotate operations include

- Logical shift to the left - LSL
- Logical shift to the right - LSR
- Arithmetic shift to the right - ASR
- Rotate right - ROR
- Rotate right through carry - RRX

Shifting operation is embedded in all most all data processing instructions. Only the source 2 operand is allowed to be shifted. The amount of shift can be specified as an immediate (5 bits) or in another register. Shifting happens within the barrel shifter in the data path.

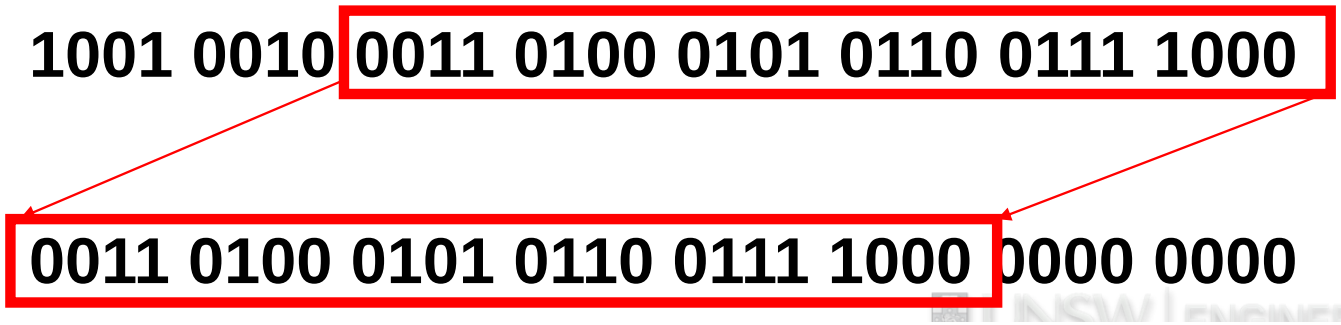
Shifts and Rotate Options

Logical shift to the left - LSL



mov a1, v1, lsl #8 ;a1 ← v1 << 8 bits

mov a1, v1, lsl v2 ;a1 ← v1 << v2 bits

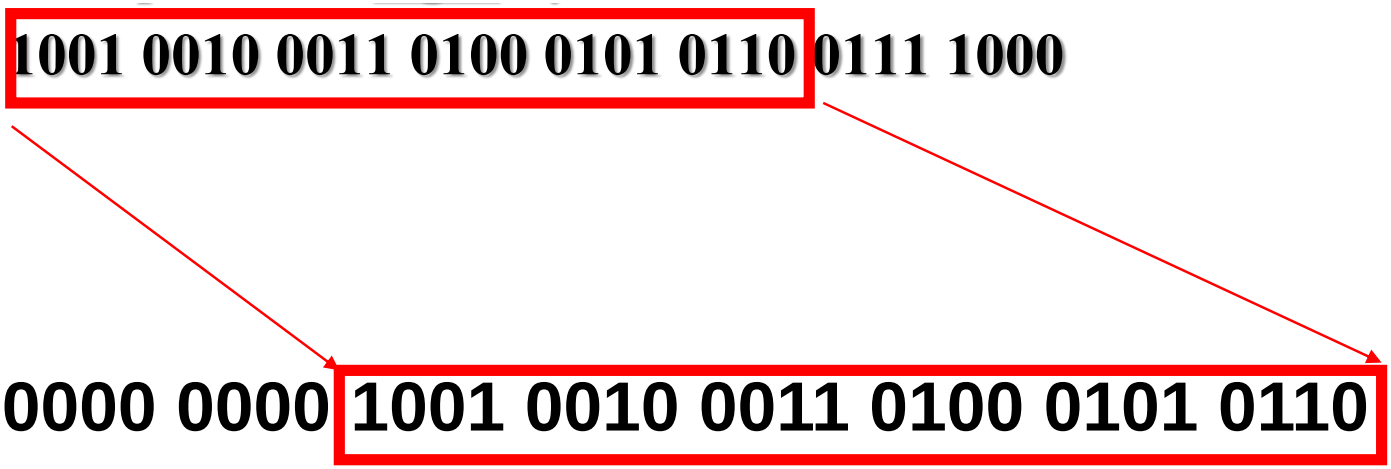


Shifts and Rotate Options

Logical shift to the right - LSR

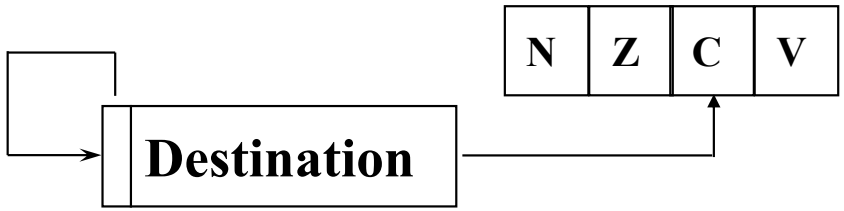
mov a1, v1, lsr #8 ;a1 ← v1 >> 8 bits

mov a1, v1, lsr v2 ;a1 ← v1 >> v2 bits



Shifts and Rotate Options

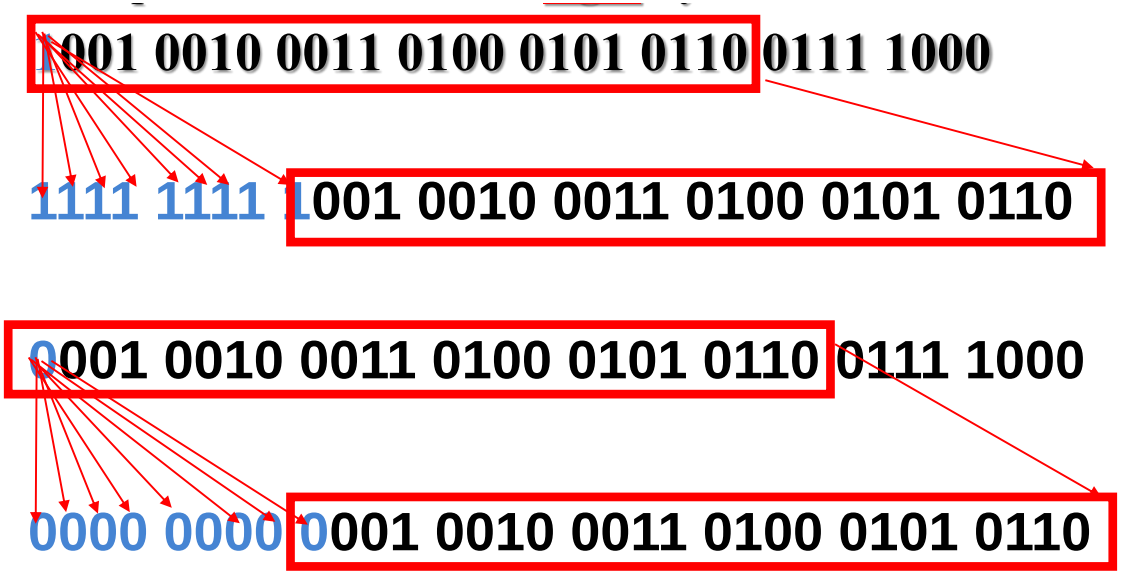
Arithmetic shift to the right - ASR



Sign bit shifted in

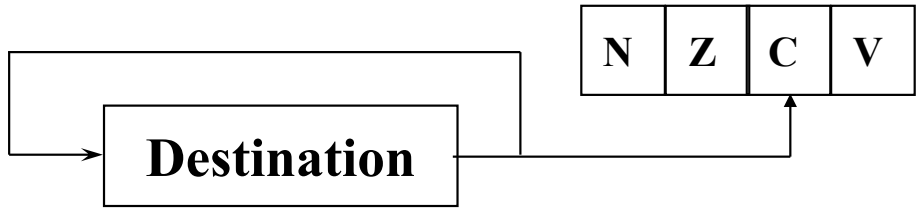
mov a1, v1, asr #8 ;a1 ← v1 >> 8 bits ;a1[31:24]=v1[31]

mov a1, v1, asr v2 ;a1 ← v1 >> v2 bits ;a1[31:24]=v1[31]



Shifts and Rotate Options

Rotate right - ROR

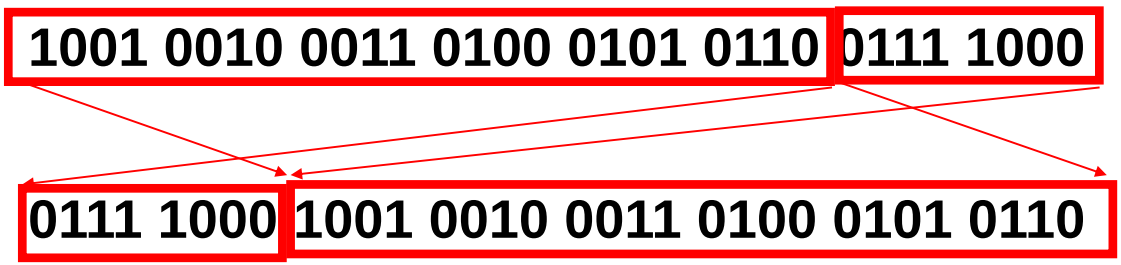


mov a1, v1, ror #8 **;a1 ← v1 >> 8 bits**

;a1[31:24]← v1[7:0]

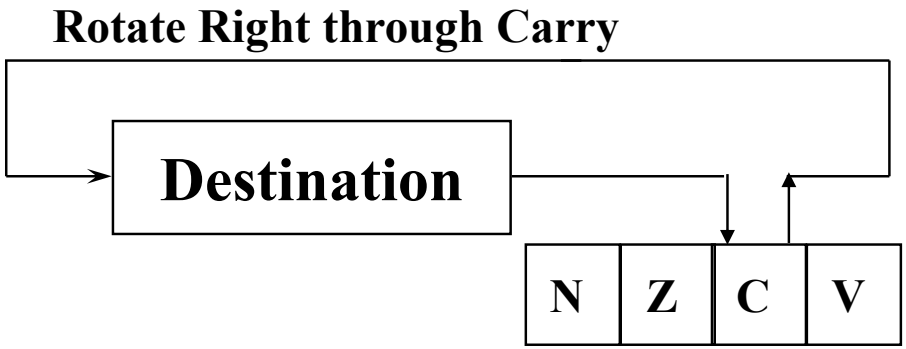
mov a1, v1, ror v2 **;a1 ← v1 >> v2 bits**

;a1[31:(31-v2)] ← v1[v2:0]



Shifts and Rotate Options

Rotate right through carry - RRX



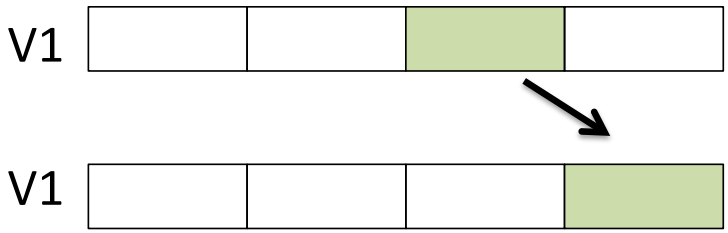

```
mov a1, v1, rrx    ;a1 ← v1 >> 1 bit
                   ;a1[31]← CF
                   ;CF ← v1[0]
```

Rotation happens through the carry flag in CPSR. Rotation by 1-bit only.

Shifts and Logical Operations Examples

Q.

Isolate the second byte of register V1 and move it to the first byte of V1.




```
AND V1, V1, #0xFF00
MOV V1, V1, LSR #8
```

Alternatively,

```
MOV V1, V1, LSL #16
MOV V1, V1, LSR #24
```

Shifts and Logical Operations Examples

Q.

Extract bit 2 to bit 9 of register V1 and place it in bit 0 to bit 7 of register V2.

Shifts and Logical Operations Examples

Q.

V1 contains 1011 1100 1000 0101 0001 111 0 1011 111 0

V2 contains 1110 1110 0101 1001 1101 1111 0101 0110

Replace bits 1-8 of V1 with bits 0-7 of V2. Other bits of V1 should not be affected.

```
AND A1, V2, #0xFF
MOV A1, A1, LSL #1
BIC V1, V1, #0x1EF
ORR V1, V1, A1
```

Multiplication

- Multiplication is expensive in terms of speed, power, and hardware complexity.
- Multiplication by a constant may be achieved by means of shits and add/sub.

Assume variables A and B corresponds to registers V1 and V2, respectively.

$B = 5 \times A = (2^2 + 1)A = 2^2 A + A$

```
ADD V2, V1, V1, LSL #2
```

$B = 105 \times A = (15 \times 7)A = (2^4 - 1)(2^3 - 1)A$

```
RSB V2, V1, V1, LSL #4
RSB V2, V2, V2, LSL #3
```

Multiplication

- MUL - multiplication of two registers. Truncates the result by discarding upper 32 bits of the 64-bit result.

  MUL Rd, Rm, Rs; Rd = Rm * Rs
- MLA - multiply and accumulate. Addition is embedded within the operation. Useful in implementing digital filters.

  MLA Rd, Rm, Rs, Rn; Rd = Rn + (Rm * Rs)
- Rd and Rm cannot be the same register.
- Cannot use PC.

Multiplication

- Multiply long (MULL) and multiply and accumulate long (MLAL) keep the entire 64-bit result in two destination registers RdHi and RdLo.
- Four versions depending on whether the operands are considered as signed or unsigned.
- Unsigned multiply long

  UMULL RdLo, RdHi, Rm, Rs; RdHi:RdLo = Rm * Rs
- Signed multiply long

  SMULL RdLo, RdHi, Rm, Rs; RdHi:RdLo = Rm * Rs (signed)
- Unsigned multiply-accumulate long

  UMLAL RdLo, RdHi, Rm, Rs; RdHi:RdLo = RdHi:RdLo + Rm * Rs
- Signed multiply-accumulate long

  SMLAL RdLo, RdHi, Rm, Rs; RdHi:RdLo = RdHi:RdLo + Rm * Rs (signed)