DESN2000: Engineering Design & Professional Practice (EE&T)

Week 2

Data processing operations and memory access

David Tsai

School of Electrical Engineering & Telecommunications

Graduate School of Biomedical Engineering

d.tsai@unsw.edu.au

Biomedical Microsystems Lab

This week

- ARM data processing operations
  - Arithmetic instructions
  - Data move instructions
  - Logic instruction
  - Shifts and rotate options
  - Multiplication
- Memory access instructions
  - Load-store architecture
  - ARM load and store operations
  - Addressing modes
  - Load / store in byte and halfword levels
  - Endianness
  - Load and store multiple

ARM data processing instructions

- Data processing instructions:
  - Arithmetic operations ADD, SUB, ADC,...
  - Comparison CMP, CMN, TST, TEQ
  - Logical operations AND, EOR, ORR, BIC,...
  - Data movement MOV, MVN

ARM data processing instructions

- 4-field instruction format: <opcode> <destination> <source 1> <source 2>
  - Opcode name of the operation
  - Destination destination; a register
  - source 1 1st source; a register
  - source 2 2nd source; a register, shifted register, or an immediate

- Source 2 – shifted register: Shifting amount can be:
  - Immediate value: 5-bit unsigned number
    ```
    add a1, v1, v3, lsl #3
    ```
  - Specified in the lowest-8-bits of another register
    ```
    add a1, v1, v3, lsl v4
    ```
- Source 2 – immediate:
  - Must be representable by an 8-bit number, optionally right-rotated an even number of bits:
    ```
    add a1, v1, #5        ; permitted
    add a1, v1, #0xFE00   ; permitted
    add a1, v1, #0xFEA3   ; not permitted
    ```

Arithmetic instructions: add / subtract

- Examples: assuming 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×D

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

  - E = −A = 0 − A (use reverse subtraction, RSB)

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

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

CPSR

31	30	29	28	27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8	7	6	5	4	3	2	1	0
N	Z	C	V	Do not modify / Read as Zero	I	F	T	M	M	M	M	M

Arithmetic instructions: add / subtract

- Add /subtract with carry:
  - ADC add with carry
  - SBC subtract with carry
  - RSC reverse subtract with carry
- Examples:

```
ADC V1, V2, V3    ; V1 = V2 + V3 + C
SBC V1, V2, V3    ; V1 = V2 + !V3 + C
RSC V1, V2, V3    ; V1 = V3 + !V2 + C
```


Using adder circuit for subtraction:

A – B = A + not(B) + 1

Setting carry with subtraction:

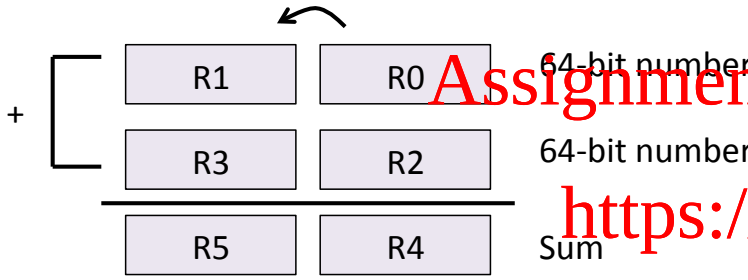
If A ≥ B ⇒ C=1

If A < B ⇒ C=0

- Useful for 64-bit arithmetic.

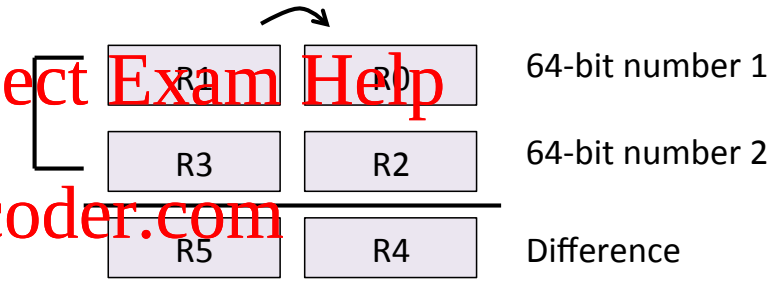
Arithmetic instructions: add / subtract

64-bit addition:




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

64-bit subtraction:




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

Data move instructions

- Moving data between regs?
  ```
  ADD V1, V2, #0  ; V1 ← V2
  ```

- Alternatively, `MOV` instruction can be used.
  ```
  MOV V1, V2      ; V1 ← V2
  ```
- The second operand can be an immediate (8-bit constraint), register or a shifted register.

- Wait a jiffy (no-operation; nop)?
  ```
  MOV V1, V1
  ```

Data move instructions

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

```
MOV V1, #0xA3     ; V1 = 0x000000A3
MVN V2, V1        ; V2 = 0xFFFFFF5C
```

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

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

Logical instructions

- Logic instructions perform bit-wise logical operations associated with registers: AND, ORR, EOR, BIC
- Instruction format: <opcode> <destination> <source 1> <source 2>
  - Opcode name of the operation
  - Destination destination operand (register)
  - source 1 1st source, a register
  - source 2 2nd source, a register, shifted register, or an immediate
- Examples:

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


#F300 ✓

#F3A1 X

Logical instructions: AND

- `AND V1, V2, V3` ⇒ V1 = V2 AND V3
- Recall:
  A AND 0 = 0
  A AND 1 = A
- Bit-wise AND operation can be used to create a bit-mask.
- Example: if register V1 contains:
  1011 0110 1011 1001 1010 1110 0110 1010
  to extract bits 4-11, we can use the following mask:
  0000 0000 0000 0000 0000 1111 1111 0000

  `AND V1, V1, #0xFF0`

A	B	A and B
0	0	0
0	1	0
1	0	0
1	1	1

Logical instructions: BIC

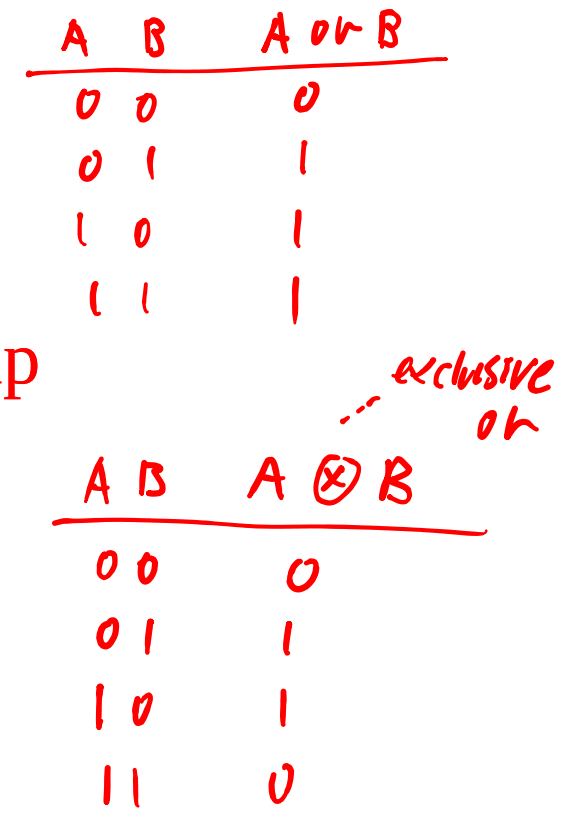
- Bit clear (BIC) is used to set certain bits to zero (and leaving other bits unchanged) in a register.
- `BIC V1, V2, V3` ⇒ V1 = V2 AND (NOT V3)
- Bits of source-1 operand (V2) in places where source-2 operand (V3) has ones will be cleared.
- Example: suppose V1 has:
  1010 1100 1010 0001 1000 0011 1100 1110
  To isolate the upper 24 bits of V1, can do a bit-clear with mask
  0000 0000 0000 0000 0000 0000 1111 1111

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

Logic Instruction – ORR, EOR

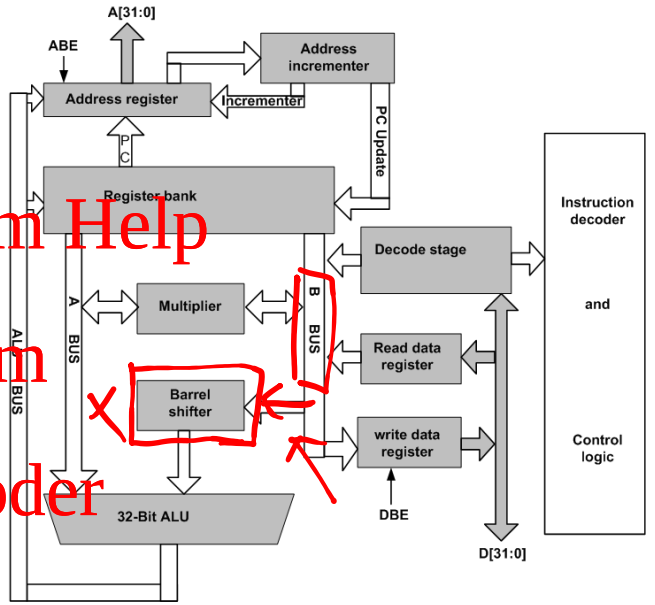
- ORR: bit-wise logical OR operation
- EOR: bit-wise logical XOR operation
- Recall:
  A OR 0 = A
  A OR 1 = 1
  A XOR 1 = NOT A
  A XOR 0 = A
  A XOR A = 0

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

Shift and rotate operations

- Shifting operation is embedded in almost all data processing instructions.
- Performed by the barrel shifter in the data path.



- ARM shift and rotate operations:
  - LSL — Logical shift to the left
  - LSR — Logical shift to the right
  - ASR — Arithmetic shift to the right
  - ROR — Rotate right
  - RRX — Rotate right through carry

- Only source-2 operand can be shifted. The shifting amount is specified as an immediate (5 bits) or in another register.

Shift and rotate operations

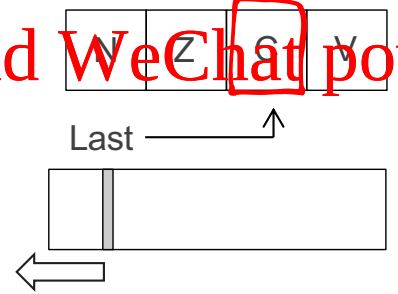
- Logical shift left (LSL) examples:

```
mov a1, v1, lsl #8   ; a1 := v1 << 8-bit
mov a1, v1, lsl v2   ; a1 := v1 << v2-bits
```

1001 0010 0011 0100 0101 0110 0111 1000
0011 0100 0101 0110 0111 1000 0000 0000

- Sets C to the value of the last bit to fall off the end of the shift. But this detail is not often significant.

CPSR



Shift and rotate operations

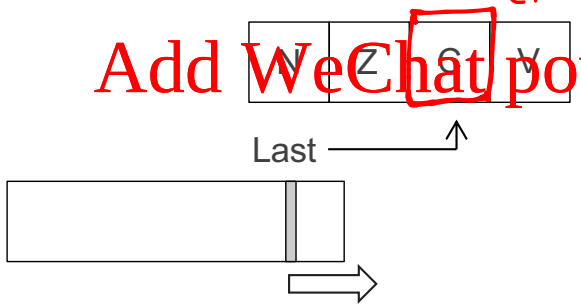
- Logical shift right (LSR) examples:

```
mov a1, v1, lsr #8   ; a1 := v1 >> 8-bit
mov a1, v1, lsr v2   ; a1 := v1 >> v2-bits
```

1001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0101 0110 0111 1000 0000 0000

- Sets C to the value of the last bit to fall off the end of the shift. But this detail is not often significant.



Shift and rotate operations

- Arithmetic shift right (ASR) examples:

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

#1 -ve

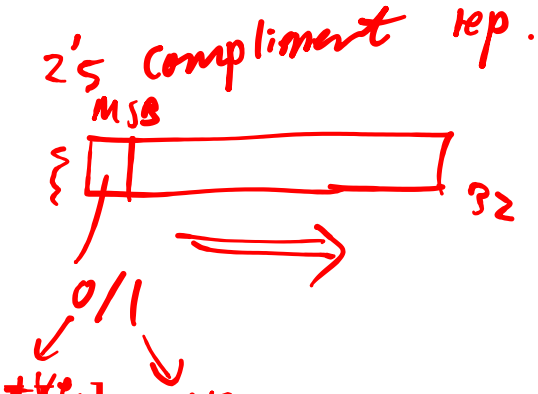
1001 0010 0011 0100 0101 0110 0111 1000

1111 1111 1001 0010 0011 0100 0101 0110

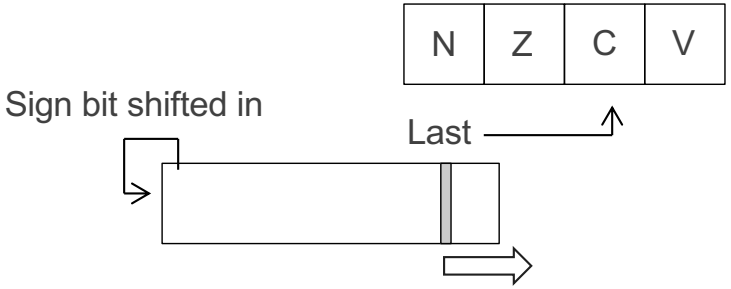
#2 +ve

0001 0010 0011 0100 0101 0110 0111 1000

0000 0000 0001 0010 0011 0100 0101 0110




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



Shift and rotate operations

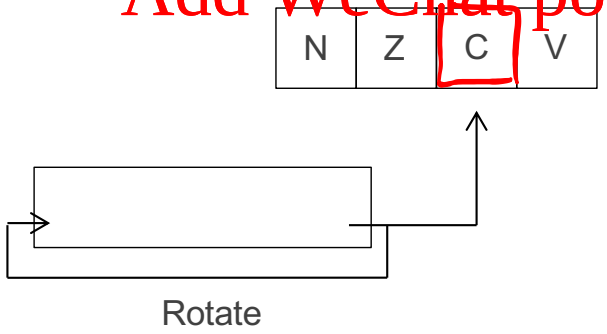
- Rotate right (ROR) examples:

```
mov a1, v1, ror #8   ; a1 := v1 >> 8-bits
                     ; a1[31:24] := v1[7:0]
```

1001 0010 0011 0100 0101 0110 0111 1000

0111 1000 1001 0010 0011 0100 0101 0110


```
mov a1, v1, ror v2   ; a1 := v1 >> v2-bits
                     ; a1[31:(31-v2)] := v1[v2:0]
```

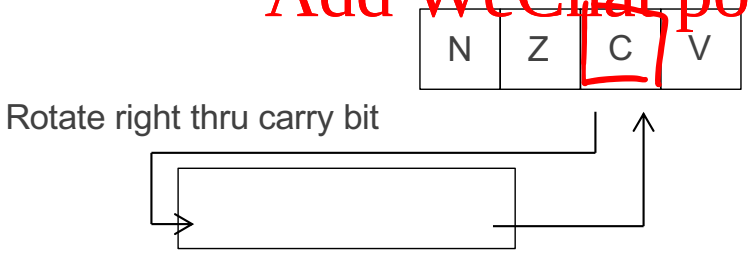



Shift and rotate operations

- Rotate right through carry (RRX) example:

```
mov a1, v1, rrx  ; a1 := v2 >> 1-bit
                 ; a1[31] := C-flag
                 ; C-flag := v1[0]
```

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



Shift and rotate operations

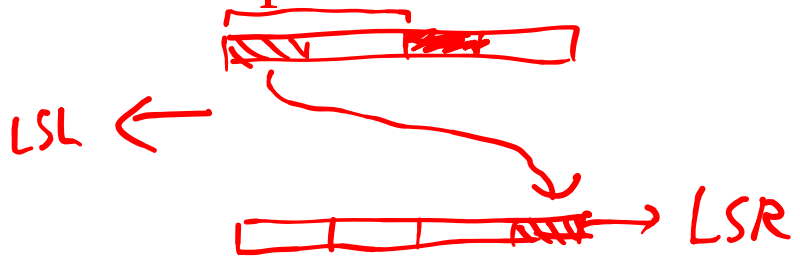
Q: Isolate the second byte of register V1 and move it to the first byte of V1.

Option 1: by bit-masking then shifting

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


Option 2: shift left (zero-fill right) then shift right (zero-fill left)

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



Multiplication

- Multiplication is expensive (clk cycles, power and circuit complexity).
- Multiplication by a constant may be achieved by shifting and add / sub.

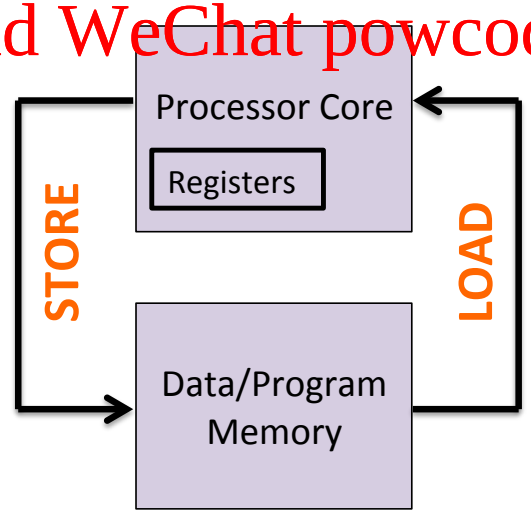
- Examples: assuming variables A and B corresponds to registers V1 and V2, respectively.
  1. $B = 5 \times A = (2^2 + 1)A = 2^2A + A$
     `ADD V2, V1, V1, LSL #2`
  2. $B = 105 \times A = (15 \times 7)A = (2^4 - 1)(2^3 - 1)A$
     `RSB V2, V1, V1, LSL #4`
     `RSB V2, V2, V2, LSL #3`

- Reverse subtract without carry, syntax:
  `RSB <Rd>, <Rn>, <operand2>`
  Rd = operand2 – Rn
  Useful because ALU operations permissible on operand2.

- $2^4 - 1$ achieved by LSL #4, then RSB reverse-subtracting 1.
- $2^3 - 1$ achieved by LSL #3, then RSB reverse-subtracting 1.

Load-store architecture

- Data processing instructions only work on registers.
  Must move memory-based data to register first, before processing.
- Remember:
  - Memory is organized as 8-bit blocks
  - Registers are 32-bits wide.
  - 32-bit data bus and 32-bit address bus.



Addressing modes

- **Pre-indexed** ++address
  - offset is added to the base register **before** the load / store.
  - The load/store takes place at the address pointed by (base + offset).
  - *Optionally* updates the base register with new address.
- **Post-indexed** address++
  - offset is added to the base **after** the load / store.
  - The load/store happens at the address pointed by the base register.
  - *Always* updates the base register with new address.

Pre-indexed load / store

- Syntax: <opcode>{cond} <rd> [<rn>, <offset>] {!}
  - opcode **LDR** (loading a word) and **STR** (storing a word).
  - Cond Optional conditional execution.
  - Rd Destination register for LDR and source register for STR.
  - Rn Base register.
  - Offset Offset from the base register: an immediate, a register, or a shifted register.
  - ! Optionally write effective address (base + offset) back to the base register.
- Load/store operation happens at the effective address (rn + offset).

Pre-indexed load / store - examples

- `LDR V1, [V2, #12]`
  A word located at the effective address V2 + 12 is loaded into register V1.
- `LDR V1, [V2, #12]!`
  A word located at the effective address V2 + 12 is loaded into register V1 and the base register V2 is updated to V2 + 12.
- `LDR V1, [V2, V3]`
  A word located at the effective address V2 + V3 is loaded into register V1.
- `LDR V1, [V2, V3, LSL #2]`
  A word located at the effective address V2 + 4*V3 is loaded into register V1 and the base register V2 is updated to V2 + 4*V3.

Post-indexed load / store

- Syntax: <opcode>{cond} <rd> [<rn>], <offset>
  - Opcode **LDR** (loading a word) and **STR** (storing a word).
  - Cond Optional conditional execution.
  - Rd Destination register for LDR and source register for STR.
  - Rn Base register. The address where load/store occurs.
  - Offset Added to base after load/store: immediate, register, or a shifted register.
- Load/store happens at the address in rn
- (rn + offset) becomes the base register for the next load/store.

Post-indexed load / store - examples

#1 loading mem.

- `LDR V1, [V2], #12` ..... #2 addr update : V2 = V2 + 12
  A word located at the effective address given by V2 is loaded into register V1, then base register V2 is updated to V2 + 12.
- `LDR V1, [V2], V3`
  A word located at the effective address given by V2 is loaded into register V1, then base register V2 is updated to V2 + V3.
- `LDR V1, [V2], V3, LSL #3` #2 addr update : V2 = V2 + V3 × 8
  A word located at the effective address given by V2 is loaded into register V1, then base register V2 is updated to V2 + 8*V3.

Load / store examples - C to assembly

- Compile C code: `C = A + B`
  - All variables are integers (words) located in memory locations having the following offsets from the base address 0x4000:
    A: offset*0
    B: offset*4
    C: offset*8


```
MOV V1, #0x4000     ; base address in V1
LDR V2, [V1]        ; load A
LDR V3, [V1, #4]    ; load B
ADD V2, V2, V3      ; Add
STR V2, [V1, #8]    ; store C
```

Load / store examples - C to assembly

- Compile C code: `G = F + my_array[i]`
  - All variables are integers. G and F correspond to registers V1 and V2, respectively. Base address for my_array (i.e. address of my_array[0]) is in register V3. Array index i is in register V4.

  - Using pre-indexed addressing:

```
LDR A1, [V3, V4, LSL #2]   ; load my_array[i]
ADD V1, V2, A1             ; perform addition
```


  - Using post-indexed addressing:


```
ADD V3, V3, V4, LSL #2
LDR A1, [V3]          ; load my_array[i]. V3 now points to my_array[i]
ADD V1, V2, A1        ; perform addition
```

Load / store in byte & halfword

- Load/store can happen at halfword or byte level.

Variants of load:

- LDR loading a word
- LDRB loading a byte
- LDRSB loading a signed byte
- LDRH loading a halfword
- LDRSH loading a signed halfword

Variants of store:

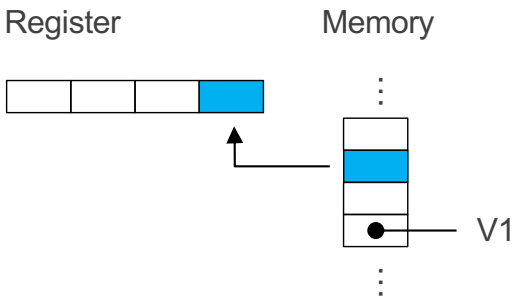
- STR storing a word
- STRB storing a byte
- STRH storing a halfword

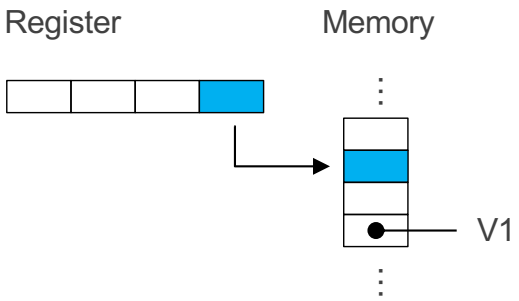
Load / store in byte & halfword

- This loads / stores the register's least-significant-byte.
- Example: loading a byte from memory to a register.

```
LDRB A1, [V1, #2]          ; V1 + 2-byte offset
LDRB A1, [V1, V2]
LDRB A1, [V1, V2, LSL #1]  ; OK, V1 + V2*2
```

- Use STRB to store.

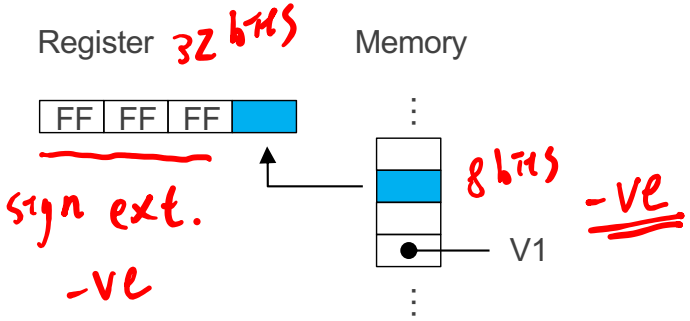




Load / store in signed byte & halfword

- Performs automatic sign bit extension
- Example: loading a signed byte from memory to a register.

```
LDRSB A1, [V1, #2]          ; V1 + 2-byte offset
LDRSB A1, [V1, V2]
LDRSB A1, [V1, V2, LSL #1]  ; illegal, not supported
```

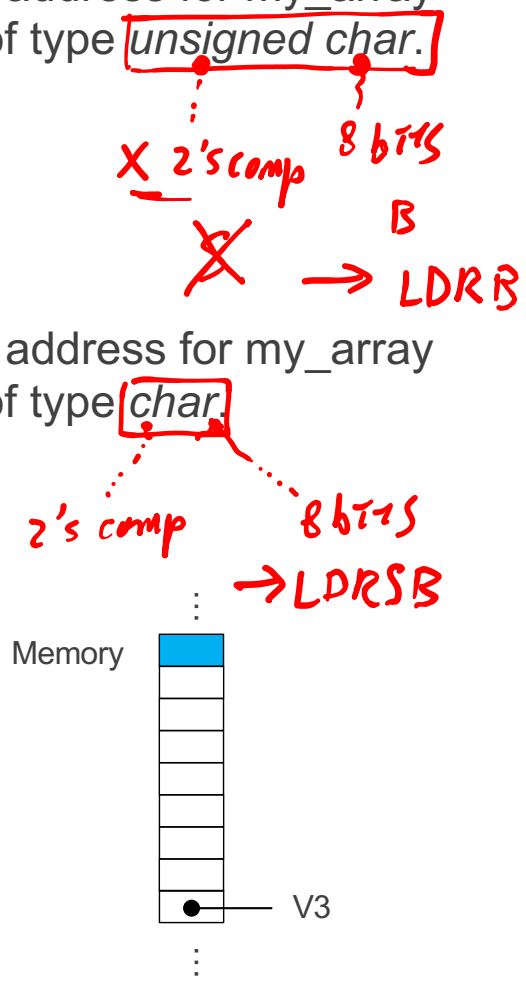
Load / store in byte & halfword

- Compile C code: `G = F + my_array[8]`
  - G and F correspond to registers V1 and V2, respectively. Base address for my_array (i.e. address of my_array[0]) is in register V3 and my_array is of type *unsigned char*.

```
LDRB A1, [V3, #8]
ADD V1, V2, A1
```

- Compile C code: `G = F + my_array[8]`
  - G and F correspond to registers V1 and V2, respectively. Base address for my_array (i.e. address of my_array[0]) is in register V3 and my_array is of type *char*.

```
LDRSB A1, [V3, #8]
ADD V1, V2, A1
```

Load / store in byte & halfword

- Compile C code: `G = F + my_array[8]`
  - G and F correspond to registers V1 and V2, respectively. Base address for my_array (i.e. address of my_array[0]) is in register V3 and my_array is of type *unsigned short*.

```
LDRH A1, [V3, #16]
ADD V1, V2, A1
```

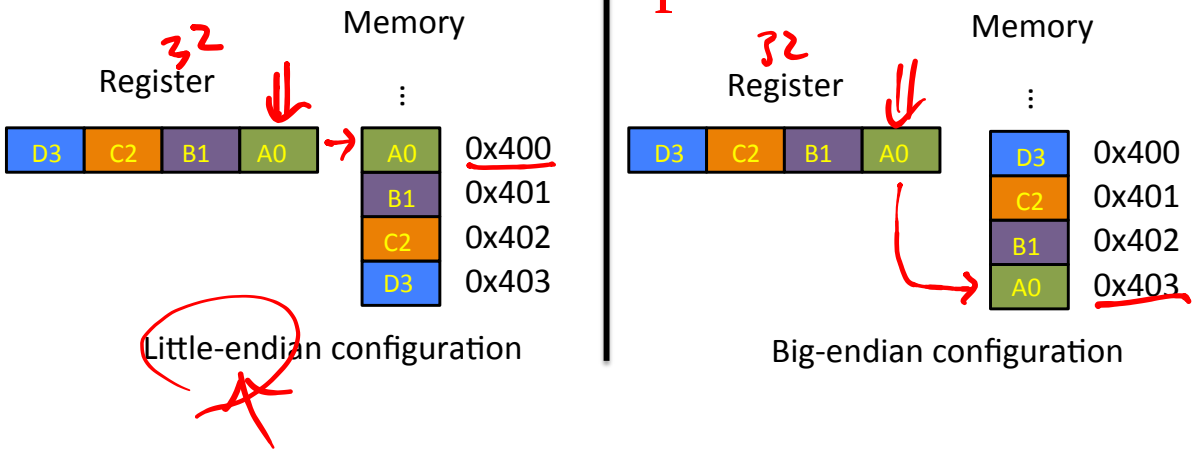
"short" is twice the size of "char" so offset twice the amount (8 x 2 = 16)

- Compile C code: `G = F + my_array[8]`
  - G and F correspond to registers V1 and V2, respectively. Base address for my_array (i.e. address of my_array[0]) is in register V3 and my_array is of type *short*.

```
LDRSH A1, [V3, #16]
ADD V1, V2, A1
```

Endianness

- Specifies the order of bytes stored in the memory.
- **Little-endian:** the least significant byte of register is stored at the lowest memory address.
- **Big-endian:** the least significant byte of register is stored at the highest memory address.
- ARM is little-endian by default.



Assembly examples

- Adding elements in an array
  - Let A[ ] be an integer array (each element is 32 bits). The base address (i.e. the address of A[0]) is in register A1. Compute:

$$\sum_{i=x}^{x+n-1} A[i]$$

and store the result in register V1. Starting index *x* is in register A2 and number of terms *n* ≥ 2 is stored in register A3. You should use post-indexed addressing mode.


```
    ADD A1, A1, A2, LSL #2  ; address of A[x] … start
    ADD A3, A1, A3, LSL #2  ; address of A[x+n] … end
    MOV V1, #0              ; initialize the sum
loop
    LDR A4, [A1], #4        ; load A[i++]
    ADD V1, V1, A4          ; summation
    CMP A1, A3              ; check whether i=x+n
    BLT loop                ; repeat while i<x+n
```

Assembly examples

- Type casting
  - Consider the following C statements:

```
int Var1;
char Var2;
Var1 = (int) Var2;
```

signed (2's comp)

  - If registers V1 and V2 hold the base addresses for Var1 and Var2, respectively, the above C code is equivalent to:

sign ext. #1 byte

```
LDRSB A1, [V2]
STR A1, [V1]
```

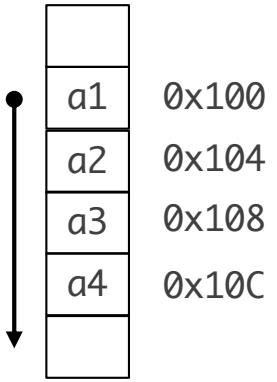
Load / store multiple, with address update

- The following code

```
str a1, [v1], #4
str a2, [v1], #4
str a3, [v1], #4
str a4, [v1], #4
```

can be done with STMIA (store multiple, **increment after**):

```
stmia v1!, {a1-a4}
```

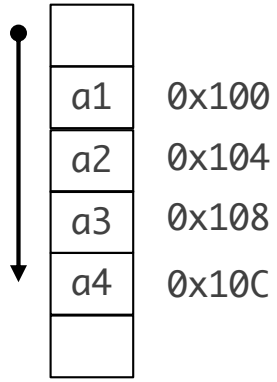


- The following code

```
str a1, [v1, #4]!
str a2, [v1, #4]!
str a3, [v1, #4]!
str a4, [v1, #4]!
```

Can be done with STMIB (store multiple, **increment before**):

```
stmib v1!, {a1-a4}
```



Ordering doesn't matter, the lowest numbered register maps to the lowest mem address

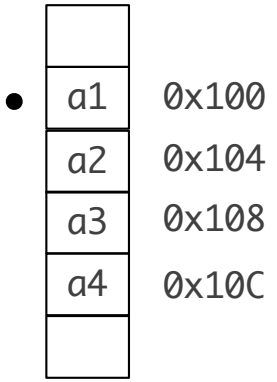
Load store multiple, no address update

- The following code

```
str a1, [v1]
str a2, [v1, #4]
str a3, [v1, #8]
str a4, [v1, #12]
```

can be done with STMIA (store multiple, **increment after**):

```
stmia v1, {a1-a4}
```

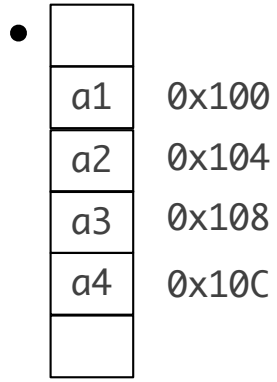


- The following code

```
str a1, [v1, #4]
str a2, [v1, #8]
str a3, [v1, #12]
str a4, [v1, #16]
```

can be done with STMIB (store multiple, **increment before**):

```
stmib v1, {a1-a4}
```



This week

- ARM data processing operations
  - Arithmetic instructions
  - Data move instructions
  - Logic instruction
  - Shifts and rotate options
  - Multiplication
- Memory access instructions
  - Load-store architecture
  - ARM load and store operations
  - Addressing modes
  - Load / store in byte and halfword levels
  - Endianness
  - Load and store multiple

In Moodle:

- **Start working on Lab 1 (Due: end of your 3-hr lab)**
- **Start doing Week 2 exercise**

References

[1] William Hohl, ARM Assembly Language: Fundamentals and Techniques, CRC Press, 2015 (2nd Edition).

[2] ARM Architecture Reference Manual.