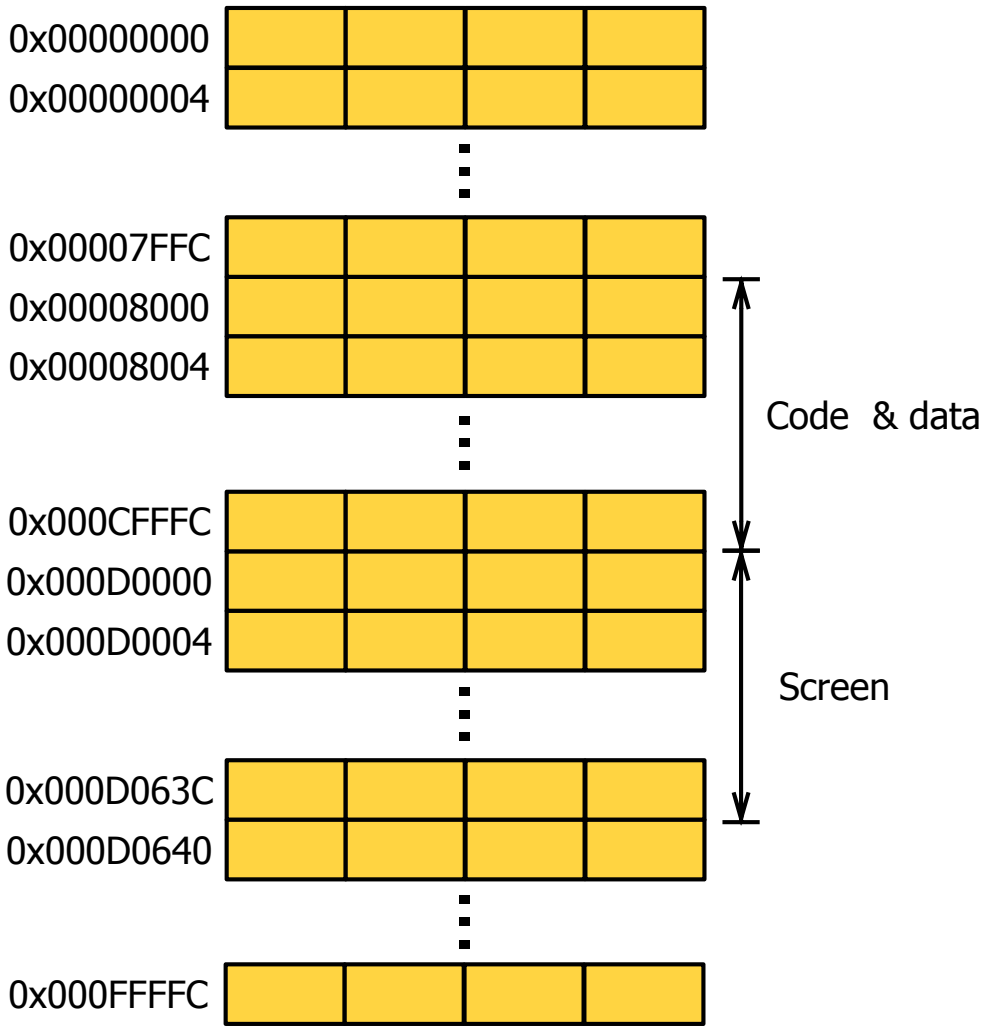
Assembly Programming

010.133
Digital Computer Concept and Practice
Spring 2013

Lecture 07

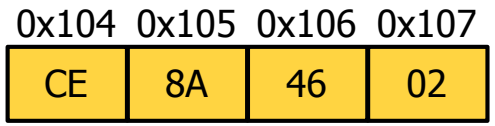
The Virtual Machine

- A virtual computer architecture
- A little-endian machine
- Word size: 32 bits
- Main memory size: 1MB
  - 0x00008000 ~ 0x000CFFFC: code and data
  - 0x000D0000 ~ 0x000D0640: screen

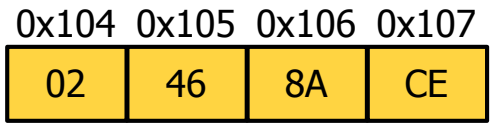


Little Endian and Big Endian

- A little-endian machine stores the most significant byte of a word at the lowest address
  - x86, old ARM, old Alpha
- A big-endian machine stores it at the lowest address
  - Old PowerPC, old SPARC
- Bi-endian machines
  - Configurable
  - PowerPC, SPARC v9, Alpha, MIPS, ARM v8
- For example, a 32-bit word 0x02468ACE is stored at address 0x00000104



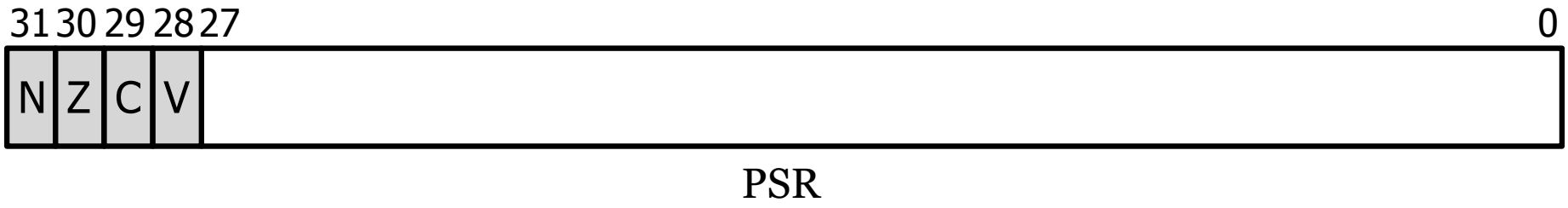
Little endian



Big endian

Registers

- A total of 17 registers
- 13 general-purpose registers
  - To store data and addresses
  - R0 ~ R12
- 4 special-purpose registers
  - Stack pointer (SP), link register (LR), program counter (PC), and processor status register (PSR)
  - R13: SP
  - R14: LR
  - R15: PC
  - The PSR contains condition flags



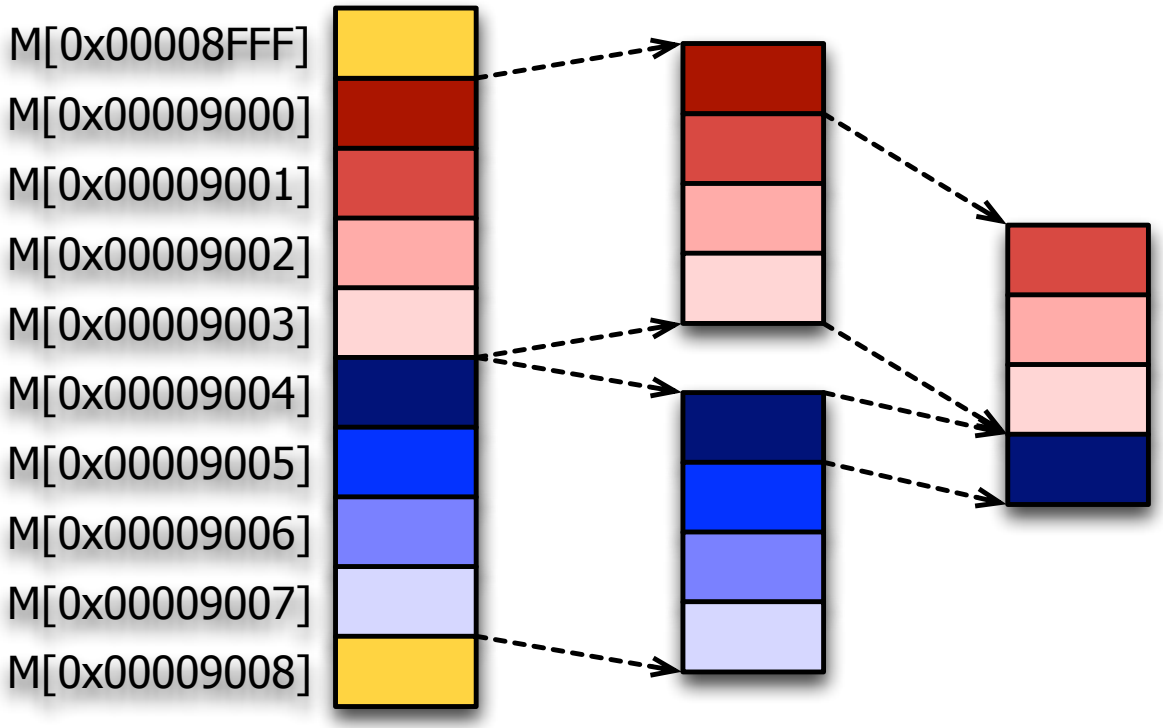
PSR

Alignment

- In general, CPUs do not read from and write to memory in byte-sized chunks
  - A CPU typically accesses memory in two-, four-, eight-, 16-, or even 32-byte chunks
- Instruction and data alignment requirements of the VM
  - Only aligned accesses are supported
  - Instruction
    - A 32-bit (4-byte) instruction must be aligned on a 4-byte boundary
  - Data
    - A 4-byte word must be aligned on a 4-byte boundary
    - A 2-byte word must be aligned on a 2-byte boundary

Alignment (contd.)

- Assume that a CPU requires 4-byte words to be aligned on a 4-byte boundary
- Accessing a 4-byte word from an unaligned address 0x9001 requires two reads of 4-byte data



Instructions Set

- Instruction size
  - Fixed 32-bit instructions
- Supported instruction types
  - Move instructions (MOV)
  - Arithmetic instructions (ADD, SUB, MUL)
  - Logic instructions (AND, ORR, EOR)
  - Load/store instructions (LDR, LDRH, LDRB, STR, STRH, STRB)
  - Comparison instructions (CMP)
  - Branch instructions (B, BNE, BGT, BLT, BGE, BLE, BL)

MOV Instructions

- MOV <Rd>, <Operand>
  - <Rd>: R0 ~ R15
  - <Operand>: an 8-bit integer constant or a register
  - The 8-bit integer constant is prefixed with '#'
    - Examples
      - #0x24, #-0x13, and #127
  - Meaning
    - Moves the value of <Operand> to the destination register <Rd>
  - Examples
    - MOV R0, #32
    - MOV R1, #-25
    - MOV R2, R3
    - MOV R2, 3245 (not allowed)

Arithmetic Instructions

- ADD <Rd>, <Rn>, <Operand>
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - <Operand>: an 8-bit integer constant or a register
  - Meaning
    - Adds the value of <Operand> to the value of register <Rn>, and stores the result in the destination register <Rd>
  - Examples
    - ADD R2, R3, #12
    - ADD R2, R2, #-1
    - ADD R0, R2, R1

Arithmetic Instructions (contd.)

- SUB <Rd>, <Rn>, <Operand>
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - <Operand>: an 8-bit integer constant or a register
  - Meaning
    - Subtracts the value of <Operand> from the value contained <Rn>, and stores the result in the destination register <Rd>
  - Examples
    - SUB R2, R3, #12
    - SUB R2, R2, #-1
    - SUB R0, R2, R1

Arithmetic Instructions (contd.)

- MUL <Rd>, <Rm>, <Rs>
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - <Rs>: R0 ~ R15
  - Meaning
    - Multiplies the contents of <Rm> and <Rs>, and stores the result in the destination register <Rd>
    - The least significant 32 bits of the result are written to the destination register
  - Examples
    - MUL R12, R11, R10

Logic Instructions

- AND <Rd>, <Rn>, <Operand>
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - <Operand>: an 8-bit integer constant or a register
  - Meaning
    - Performs a bitwise AND of the value of <Operand> and the value of register <Rn>, and stores the result in the destination register <Rd>
  - Examples
    - AND R2, R3, R3
    - AND R2, R2, #0
    - AND R0, R2, #0x000000FF

Logic Instructions (contd.)

- ORR <Rd>, <Rn>, <Operand>
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - <Operand>: an 8-bit integer constant or a register
  - Meaning
    - Performs a bitwise OR of the value of <Operand> and the value of register <Rn>, and stores the result in the destination register <Rd>
  - Examples
    - ORR R2, R3, R3
    - ORR R2, R2, #-1
    - ORR R0, R2, #0x000000FF

Logic Instructions (contd.)

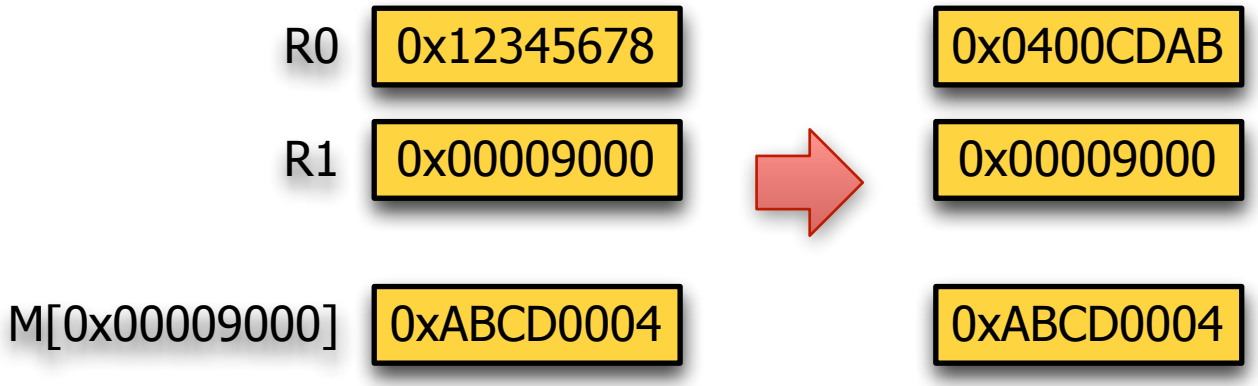
- EOR <Rd>, <Rn>, <Operand>
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - <Operand>: an 8-bit integer constant or a register
  - Meaning
    - Performs a bitwise exclusive OR of the value of <Operand> and the value of register <Rn>, and stores the result in the destination register <Rd>
  - Examples
    - EOR R2, R3, R3
    - EOR R2, R2, #-1
    - EOR R0, R2, #0x000000FF

Addressing Mode

- Addressing mode define how a machine instruction refers to memory locations for its operands
- The addressing mode of the VM consists of two parts
  - Base register
  - Offset
- <Addressing_mode>
  - [<Rn>, <Offset>]
    - <Rn>: R0 ~ R15
    - <Offset>: a 12-bit integer constant
    - The address is determined by adding the value of <Offset> to the content of <Rn>
  - [<Rn>]
    - Equivalent to [<Rn>, #0]

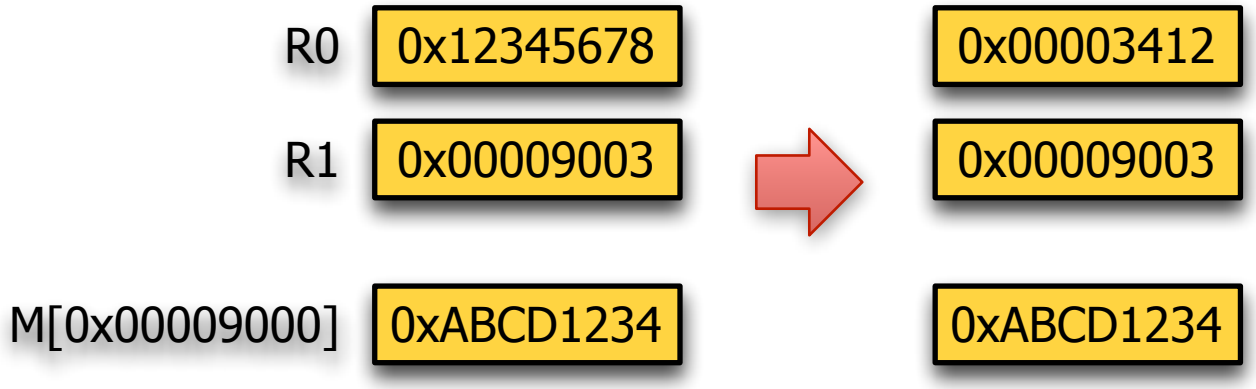
Load Instructions

- LDR <Rd>, <Addressing_mode>
  - <Rd>: R0 ~ R15
  - Meaning
    - A data word located at the address specified by <Addressing_mode> is loaded into the destination register <Rd>
    - The address must be a multiple of four
    - If PC is specified as <Rd>, the instruction the loaded data word as an address and branches to that address
  - Example
    - LDR R0, [R1]



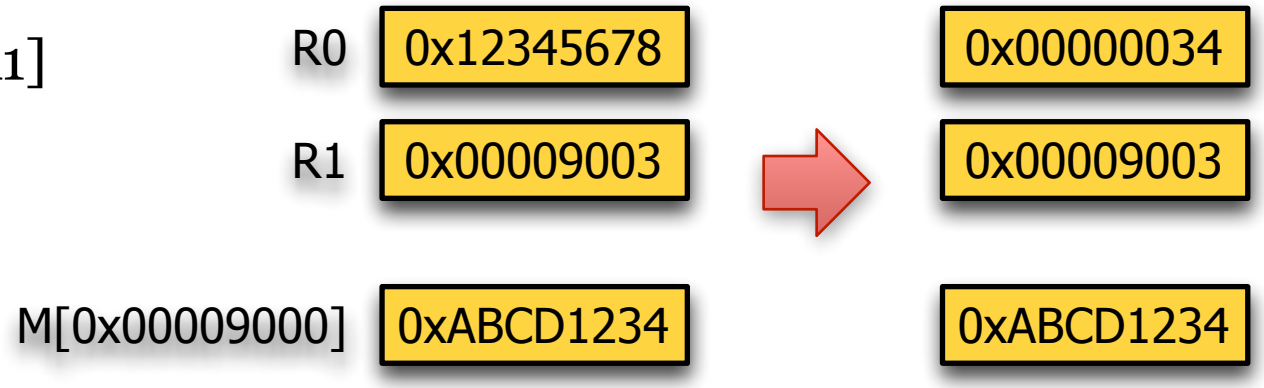
Load Instructions (contd.)

- LDRH <Rd>, <Addressing_mode>
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - Meaning
    - A half word at the address specified by <Addressing_mode> is zero-extended to form a 32-bit word, and then the 32-bit result is stored to the destination register <Rd>
    - The address must be a multiple of two
  - Example
    - LDRH R0, [R1]



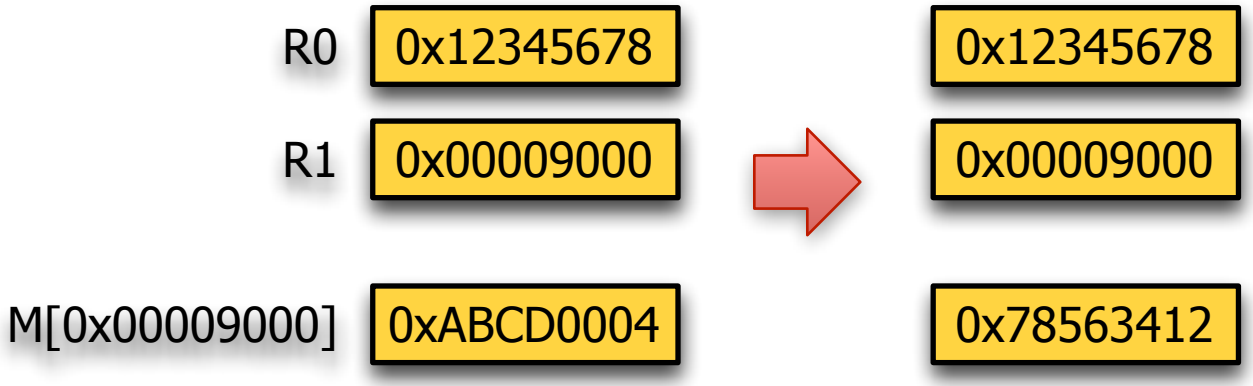
Load Instructions (contd.)

- LDRB <Rd>, <Addressing_mode>
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - Meaning
    - A byte at the address specified by <Addressing_mode> is zero-extended to form a 32-bit word, and then the 32-bit result is stored to the destination register <Rd>
  - Example
    - LDRB R0, [R1]



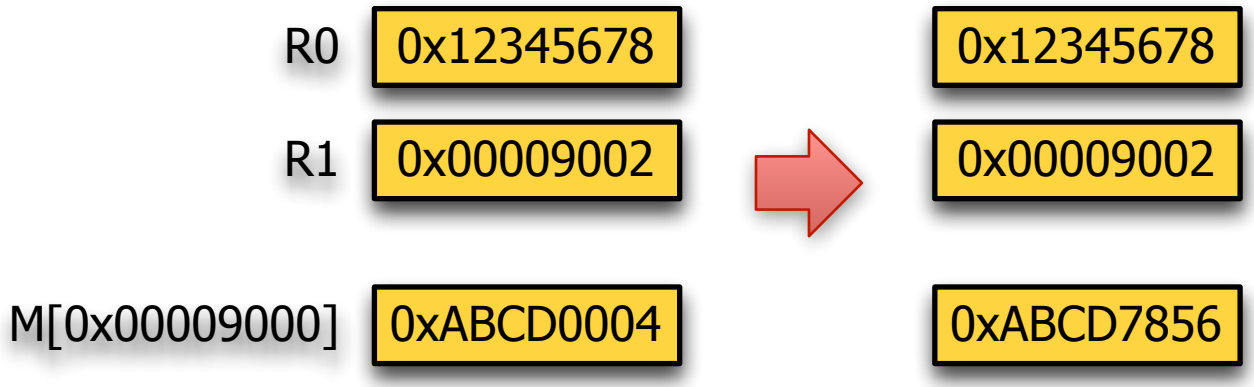
Store Instructions

- STR <Rd>, [<Rn>]
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - Meaning
    - A data word contained in <Rd> is stored to the memory location with the address specified by <Addressing_mode>
    - The address must be a multiple of four
  - Example
    - STR R0, [R1]



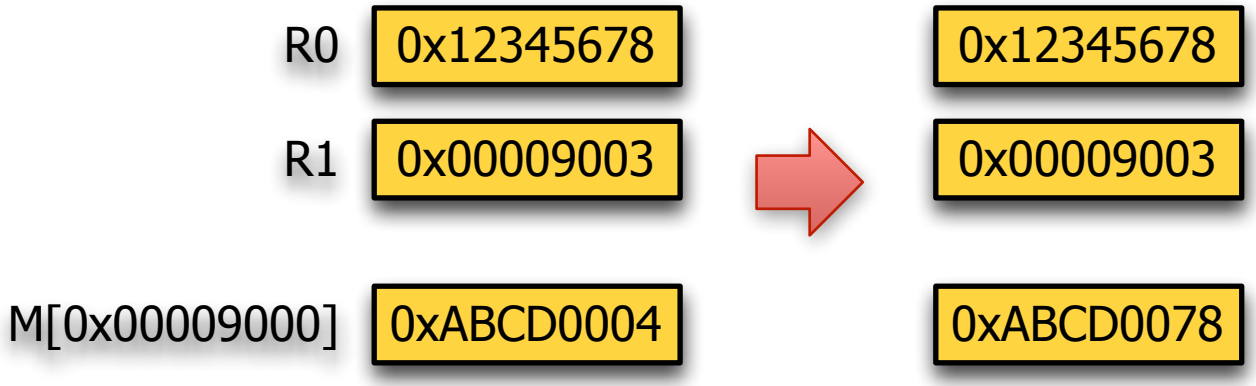
Store Instructions (contd.)

- STRH <Rd>, [<Rn>]
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - Meaning
    - The least significant two bytes of the word contained in <Rd> (Rd[15:0]) are stored to the memory location with the address specified by <Addressing_mode>
    - The address must be a multiple of two
  - Example
    - STRH R0, [R1]



Store Instructions (contd.)

- STRB <Rd>, [<Rn>]
  - <Rd>: R0 ~ R15
  - <Rn>: R0 ~ R15
  - Meaning
    - The least significant byte of the word contained in <Rd> (Rd[7:0]) is stored to the memory location with the address specified by <Addressing_mode>
  - Example
    - STRB R0, [R1]



Comparison Instructions

- CMP <Rn>, <Operand>
  - <Rn>: R0 ~ R15
  - <Operand>: an 8-bit integer constant or a register
  - Meaning
    - Compares the value of <Operand> with the value contained in <Rn> based on the result of subtracting the value of <Operand> from the value contained in <Rn>
    - Based on the subtraction, the condition flags (Negative, Zero, Carry-out, and oVerflow) in the PSR are updated
  - Examples
    - CMP R0, #0
    - CMP R2, R3

Unconditional Branch Instructions

- B <Label>
  - <Label> is a label
    - Some text followed by a colon, like "L1:"
    - A named location in the code
    - Converted by the assembler to the target address of the branch
  - Meaning
    - Cause a jump to the target address labeled with "<label>:"
  - Example code

```
        ADD   R2, R3, R0
        B     L1
        SUB   R1, R2, #4
L1:     ADD   R1, R2, #4
```

Unconditional Branch Instructions (contd.)

- BL <Label>
  - Meaning
    - Branch and link register instruction
    - Causes a jump to the target address labeled <Label>
    - Stores the return address in the link register, LR (R14)
      - The return address is obtained by adding 4 to the value of the current PC
  - Example

```
        ADD   R2, R3, R0
        BL    L1
        SUB   R1, R2, #4
L1:     ADD   R1, R2, #4
        MOV   PC, LR
```

M[0x00008100]	ADD R2, R3, R0
M[0x00008104]	BL L1
M[0x00008108]	SUB R1, R2, #4
M[0x0000810C]	ADD R1, R2, #4
M[0x00008110]	MOV PC, LR

Unconditional Branch Instructions (contd.)

- B halt
  - "halt" is a reserved name
  - Meaning
    - Tells the underlying system (e.g., operating system) that the program execution has been finished
    - Place this instruction at the end of the program

Conditional Branch Instructions

- B<Cond> <label>
  - <Cond>: the condition mnemonic under which the instruction is executed
  - Meaning
    - Causes a branch to the target address if the condition matches the result of the previous CMP (the condition flags in the PSR)
  - Example
    - CMP R1, R2 makes BLT L1 taken if R1 < R2

Mnemonic	Description	Condition flags
EQ	Equal	Z = 1
NE	Not equal	Z = 0
GE	Signed greater than or equal to	(N = 1, V = 1) or (N = 0, V = 0)
LT	Signed less than	(N = 1, V = 0) or (N = 0, V = 1)
GT	Signed greater than	(N = 1, Z = 0, V = 1) or (N = 0, Z = 0, V = 0)
LE	Signed less than or equal to	(Z = 1) or (N = 1, V = 0) or (N = 0, V = 1)

Adding 10 Numbers

- Adding ten arbitrary integers stored in memory from the address labeled "Numbers"

```
            MOV  R0, #0
            MOV  R3, #0
            MOV  R2, =Numbers
L:          ADD  R3, R3, #1
            LDR  R1, [R2]
            ADD  R0, R0, R1
            ADD  R2, R2, #4
            CMP  R3, #10
            BNE  L
            B    halt
Numbers:
```

Pseudo-instructions

- There is no real single machine instruction that corresponds to an assembly pseudo-instruction
  - The assembler will translate the pseudo-instruction into an equivalent sequence of machine instructions
- LDR <Rd>, =<Value>
  - Loads a 32-bit word <Value> to <Rd>
- LDR <Rd>, <Label>
  - Loads the value stored at <Label> to <Rd>
- LDR <Rd>, =<Label>
  - Loads the address labeled <Label> to <Rd>
- ADR <Rd>, <Label>
  - Loads the address labeled <Label> to <Rd>

Assembler Directives

- Allow you to take special programming actions during the assembly (i.e., translation) process
- .align <number>
  - Aligns the next piece of code or data to a $2^{<number>}$ boundary
- .word <number>
  - <number> is optional
  - Creates a (4-byte) word in the location where ".word" is placed
  - If <number> exists, the number is stored in the location when the word is created

Assembler Directives (contd.)

- .hword and .byte are similar to the case of .word except they create a half word (2 bytes) and a byte in the location they are placed, respectively

```
            ...
            .align 2
var_word:
            .word 0xABCDE
array_word:
            .word 1, 2, 3, 4
var_byte
            .byte 255
var_hword
            .hword 0xBAD   @ bad alignment
            ...
```

0x00009000	DE	BC	0A	00
0x00009004	01	00	00	00
0x00009008	02	00	00	00
0x0000900C	03	00	00	00
0x00009010	04	00	00	00
0x00009014	FF			
0x00009018				

Assembler Directives (contd.)

```
            ...
            .align 2
var_word:
            .word 0xABCDE
array_word:
            .word 1, 2, 3, 4
var_byte
            .byte 255
            .align 1
var_hword
            .hword 0xBAD   @ OK
            ...
```

0x00009000	DE	BC	0A	00
0x00009004	01	00	00	00
0x00009008	02	00	00	00
0x0000900C	03	00	00	00
0x00009010	04	00	00	00
0x00009014	FF		AD	0B
0x00009018				

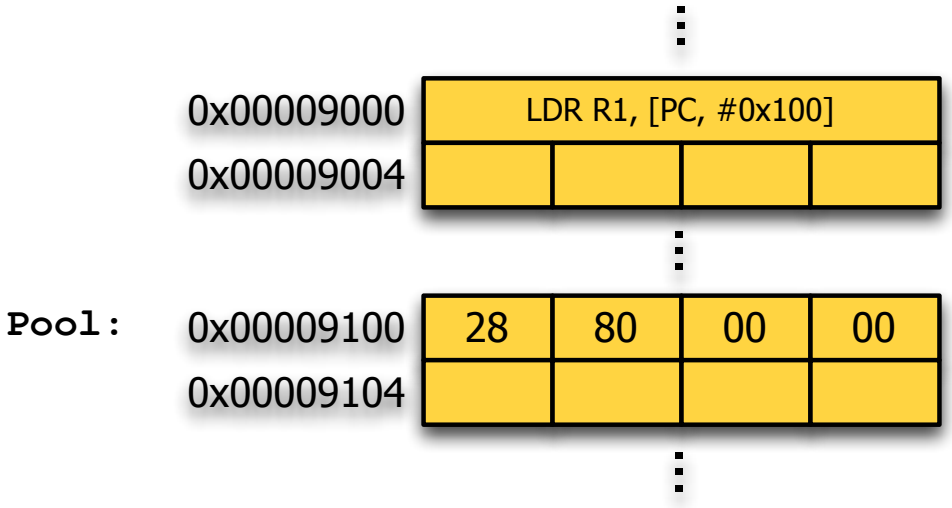
Literal Pools

- A literal is a fixed value embedded in the source code
  - A variable is a name that can take on a value any time during program execution
- A literal pool is a table that contains literals during assembly process and execution
  - The assembler uses literal pools to keep certain constant values that are to be loaded into registers
  - Literal pools are placed where the CPU does not attempt to execute them as instructions
    - After an unconditional branch instruction

Loading a 32-bit Value to a Register

- Cannot use a MOV instruction
  - Only 8-bit constant is allowed in the MOV instruction
- Use a pseudo LDR instruction and a value prefixed with =
  - The assembler checks if the value is available and addressable in any previous literal pools
  - If not, it places the value in the next literal pool
- Example
  - LDR R1, =0x00008028

```
            ...
            LDR R1,[PC, #0x100]
            ...
            B foo
            ...
Pool:
            .word 0x00008028
            ...
```



Directives and Pseudo-Instructions

```
            LDR R0, =0xABCDEABC  @ load a value
            LDR R0, =fvar        @ load fvar's address
            LDR R0, [R0]         @ load fvar's content
            LDR R0, foo          @ load a value stored at foo
            LDR R0, foo+4        @ load fvar's address
            LDR R0, [R0]         @ load fvar's content
            ...
foo:        .word 0x00001000
            .word fvar           @ contains fvar's address
fvar:       .word 0x000ABCDE
```

Variables

- If you add a label in front of ".word", you have a variable
  - Variable - a named memory location in which a program can store intermediate results and from which it can read them
- The value of a label is the address where it is located

Strings

- String
  - A sequence of characters (bytes)
  - Literal
- Null character
  - '\0' in C, 0x00 in ASCII
  - A character with the value zero
  - Present in the ASCII (NULL) and Unicode character sets
- Null terminated string
  - A string that is terminated with a null character
- C string
  - Null terminated string written between double quotes
  - For example, "Hello world!" is a C string

```
.byte 0x48, 0x65, 0x6C, 0x6C, 0x6F, 0x2C,
      0x20, 0x77, 0x6F, 0x72, 0x6C, 0x64,
      0x21, 0x00
```

0x00009000	48	65	6C	6C
0x00009004	6F	2C	20	77
0x00009008	6F	72	6C	64
0x0000900C	21	00		
0x00009010				

Fibonacci Numbers

- Fibonacci numbers are defined by

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

  - 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
- We would like to compute K^{th} (K > 1) Fibonacci number
  - K is stored at the address labeled K
  - The result is stored in R2

Fibonacci Numbers (contd.)

```
        MOV  R0, #0          @ F(0)
        MOV  R1, #1          @ F(1)
        MOV  R3, #1          @ i
        LDR  R4, K
L:      ADD  R3, R3, 1       @ i = i + 1
        ADD  R2, R0, R1      @ F(i)
        MOV  R0, R1
        MOV  R1, R2
        CMP  R3, R4
        BLT  L
        B    halt
K:       .word 100
```

Finding the Maximum

- N 32-bit integers are consecutively stored at addresses starting at the address labeled “intArray”
  - N > 0 and N is stored at the address labeled “N”
- We want to find the maximum among those N integers
- The result will be stored at the address labeled “max”
- Algorithm
  - Initialize Rx with the first integer
  - Compare the content of Rx with each integer Y and update Rx with Y if Y is bigger than the content of Rx
  - Store the content of Rx at the location labeled “max”

Linear Search

- Finding a particular number X in a sequence of numbers
  - Checking every element, one at a time sequentially, until the desired one is found

- N 32-bit integers are consecutively stored at addresses starting at the address labeled “intArray”
  - N > 0 and N is stored at the address labeled “N”
- X is stored at the address labeled “X”
- If X is found, store the position of X (starting from 0) in the number sequence at the address labeled “found”
- Otherwise store -1 to “found”

Linear Search (contd.)

```
            LDR R0, X
            LDR R1, N
            LDR R2, =intArray   @ load the value of intArray
            MOV R3, #0          @ position i
L:          LDR R4, [R2]
            CMP R0, R4
            BEQ X_found
            ADD R2, R2, #4
            ADD R3, R3, 1       @ i = i + 1
            CMP R3, R1
            BLT L
Not_found:  MOV R5, #-1         @ X not found
            STR R5, found
            B   halt
X_found:    STR R3, found       @ X found
            B   halt

found:      .word
X:          .word 14
N:          .word 9
intArray:   .word -24, 34, 92, 234, 659, -145, -789, 14, 19
```