Chapter 2

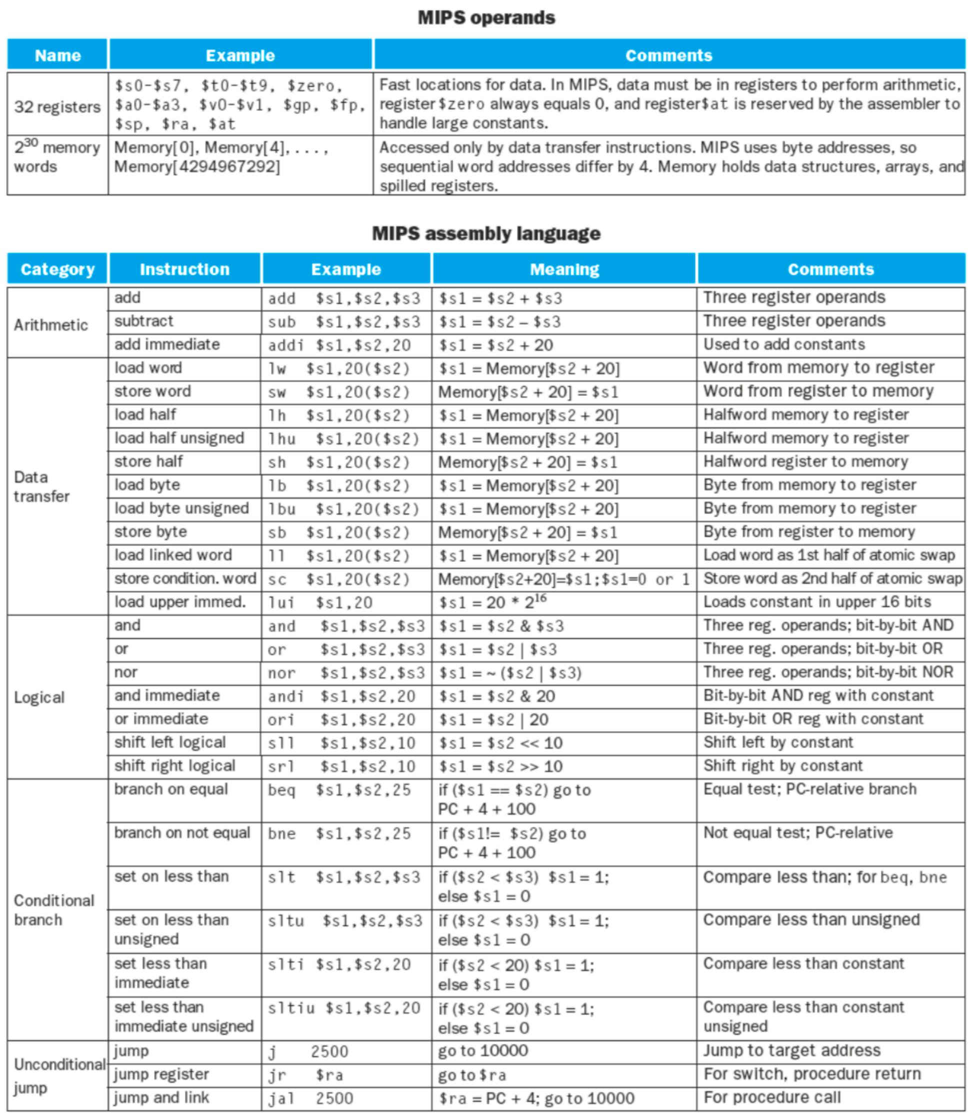
Instructions: Language of the Computer

Instruction Set

- To command a computer’s hardware, you must speak its language; the words of a computer’s language is called **instructions**
- Different computers have different instruction sets
    - But with many aspects in common
- Early computers had very simple instruction sets
    - Simplified implementation
- Many modern computers also have simple instruction sets

The MIPS Instruction Set

- The chosen instruction set is MIPS instruction set designed 1980.
- The following picture is a preview of the instruction set covered in the chapter



MIPS operands

Name	Example	Comments
32 registers	`$s0-$s7, $t0-$t9, $zero, $a0-$a3, $v0-$v1, $gp, $fp, $sp, $ra, $at`	Fast locations for data. In MIPS, data must be in registers to perform arithmetic, register `$zero` always equals 0, and register `$at` is reserved by the assembler to handle large constants.
2^{30} memory words	Memory[0], Memory[4], . . . , Memory[4294967292]	Accessed only by data transfer instructions. MIPS uses byte addresses, so sequential word addresses differ by 4. Memory holds data structures, arrays, and spilled registers.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1` = `$s2` + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half	`lh $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte	`lb $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[`$s2`+20]=`$s1`;`$s1`=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	`$s1` = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1` = ~ (`$s2` | `$s3`)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	`$s1` = `$s2` & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	`$s1` = `$s2` | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1` = `$s2` << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1` = `$s2` >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1`!= `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

Arithmetic Operations

- Every computer must be able to perform arithmetic
- The MISP assembly language notation **add a, b, c** instructs a computer to add two variables b and c and put their sum in a.
- Suppose we want to **place sum of four variables into a**

```
add a, b, c     # The sum of b and c is placed in a
add a, a, d     # The sum of b, c, and d is now in a
add a, a, e     # The sum of b, c, d, and e is now in a
```

- Sharp symbol (#) one each line are comments. Comment always terminate at the end of line.
- **The number of operands are three for most of operations,** so there is simplicity.
- ***Design Principle 1:* Simplicity favors regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Operations- example

- Compiling C assignments to MIPS

```
a = b + c;
d = a - e;
```

$\longrightarrow$

```
add a, b, c
sub d, a, e
```

```
f = (g + h) - (i + j);
```

$\downarrow$

```
add t0,g,h # temporary variable t0 contains g + h
add t1,i,j # temporary variable t1 contains i + j
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)
```

Operands of the computer hardware

- The operands of arithmetic operations are restricted; They must be from a limited number of special locations built in hardware called **registers**
- The size of a register in the MIPS architecture is 32 bits
- **MIPS has a 32 × 32-bit register file**
  - Use for frequently accessed data
  - Numbered 0 to 31
  - <u>32-bit data called a “word”</u>
- Assembler names
  - **$t0, $t1, …, $t7 for temporary values**
  - **$s0, $s1, …, $s7 for saved variables**
- ***Design Principle 2:* Smaller is faster**
  - c.f. main memory: millions of locations

- **In MIPS assembly language**
- $s0 - $s7 maps onto register 16 to 23
- $t0 - $t7   maps onto register 8 to 15

- $s0   means register 16
- $s1   means register 17
- …….
- $t0   means register 8
- $t1   means register 9
- ……..

Example

- Compiling C assignments using registers to MIPS

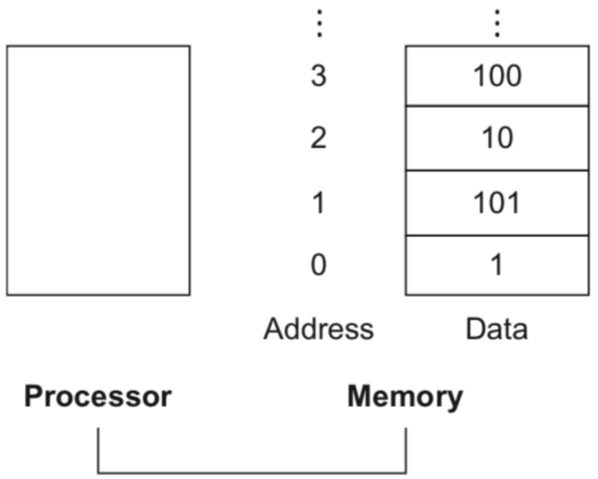
```
f = (g + h) - (i + j);
```

Variables f, g, h, i, and j are assigned to registers \$s0, \$s1, \$s2, \$s3, and \$s4 respectively. What is the complied MIPS code?

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

Memory Operands

- Processor can keep a small amount of data in registers; memory contains billion of data element; data structures, like array, are key in memory
- MIPS must include instructions that transfer data between memory and registers, called **data transfer instructions.**
- To access memory, the instruction must supply the memory address
- Memory is a single dimension array with the address acting as the index to that array



Memory Operands

- Data transfer instruction that **copies data from memory to a register is called <u>load</u>**
- **The format of the load instruction is**:

<u>The name of the operation</u> followed by<u> the register to be loaded</u>, then <u>a constant</u> and <u>register </u>*used to access memory.*

- The instruction complementary to load is **store**; **store instruction copies data from a register to memory.**
- **The format of the store instruction is:**

<u>The name of the operation</u> followed by<u> the register to be stored</u>, then <u>a constant</u> and <u>register </u>*used to access memory.*

constant: <u>offset to select the array element</u>

Register: has the address of first element of the array

Compiling an assignment when an operand is in memory

Let's assume that A is an array of 100 words and that the compiler has associated the variables g and h with the registers $s1 and $s2 as before. Let's also assume that the starting address, or *base address*, of the array is in $s3. Compile this C assignment statement:

```
g = h + A[8];
```

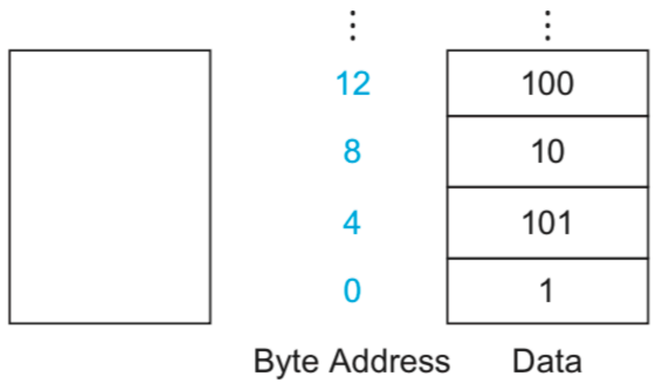
- One operand is in memory; we must transfer A[8] to a register. The address of A[8] is the sum of the base address of the array A, found in $s3, plus the number to select element 8. The constant in the data transfer (8) is called the offset

```
lw    $t0,8($s3) # Temporary reg $t0 gets A[8]
add   $s1,$s2,$t0 # g = h + A[8]
```

- The constant in a data transfer instruction (8) is called the *offset,* and the register added to form the address ($s3) is called the *base register*.

Memory Operand

- 8-bit bytes are useful in many programs,
- All architectures address individual bytes
- **The address of the word match the address of one of the four bytes within a word**, and **the address of sequential words differs by 4**
- In MIPS, words must start at addresses that are multiplies of 4. This requirement is called **an alignment restriction.**
- Computers divide into those use the address of the leftmost byte as the word address or those use the address of the rightmost byte as the word address. MIPS is leftmost
-



Compiling using load and store

Assume variable h is associated with register $s2 and the base address of the array A is in $s3. What is the MIPS assembly code for the C assignment statement below?

```
A[12] = h + A[8];
```

- The first two instructions are the same as the prior example, except this time we use the proper offset for byte addressing in load to select A[8]

```
lw    $t0,32($s3)  # Temporary reg $t0 gets A[8]
add   $t0,$s2,$t0  # Temporary reg $t0 gets h + A[8]
```

The final instruction stores the sum into A[12], using 48 (4 × 12) as the offset and register $s3 as the base register.

```
sw   $t0,48($s3)  # Stores h + A[8] back into A[12]
```

spilling registers

- Many programs have more variables than computers have registers.
- Consequently, the compiler tries to keep the most frequently used variables in registers and places the rest in memory, using loads and stores to move variables between registers and memory.
- The process of putting less commonly used variables (or those needed later) into memory is called ***spilling* registers.**

Registers vs. Memory

- Registers are faster to access than memory
- Registers take less time to access *and* have higher throughput than memory, making data in registers both faster to access and simpler to use
- Accessing registers also uses less energy than accessing memory
- To achieve highest performance and conserve energy, an instruction set architecture must have a sufficient number of registers, and compilers must use registers efficiently.

Immediate Operands

- Many programs use constant in an operation
- To add the constant 4 to register $s3, we could use the following code

```
lw $t0, AddrConstant4($s1)    # $t0 = constant 4
add $s3,$s3,$t0               # $s3 = $s3 + $t0 ($t0 == 4)
```

assuming that `$s1 + AddrConstant4` is the memory address of the constant 4.

- An alternative that avoid load instruction is an instruction called add immediate or addi

```
addi $s3, $s3, 4        # $s3=$s3+4
```

- ***Design Principle 3:*** **Make the common case fast**
  - Small constants are common
  - Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers

    ```
    add $t2, $s1, $zero
    ```

Signed and Unsigned Numbers

- 123 (base 10)= 1111011(base 2)
- Binary digit or binary bit: one of the two numbers in base 2, 0 or 1 that are components of information
- (on or off, true or false, or 1 or 0)
- In any number base, the value of its digit d is $d \times \text{Base}^i$

1011_{two}

represents

$$\begin{aligned}
& (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{\text{ten}} \\
= {} & (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{\text{ten}} \\
= {} & 8 + 0 + 2 + 1_{\text{ten}} \\
= {} & 11_{\text{ten}}
\end{aligned}$$

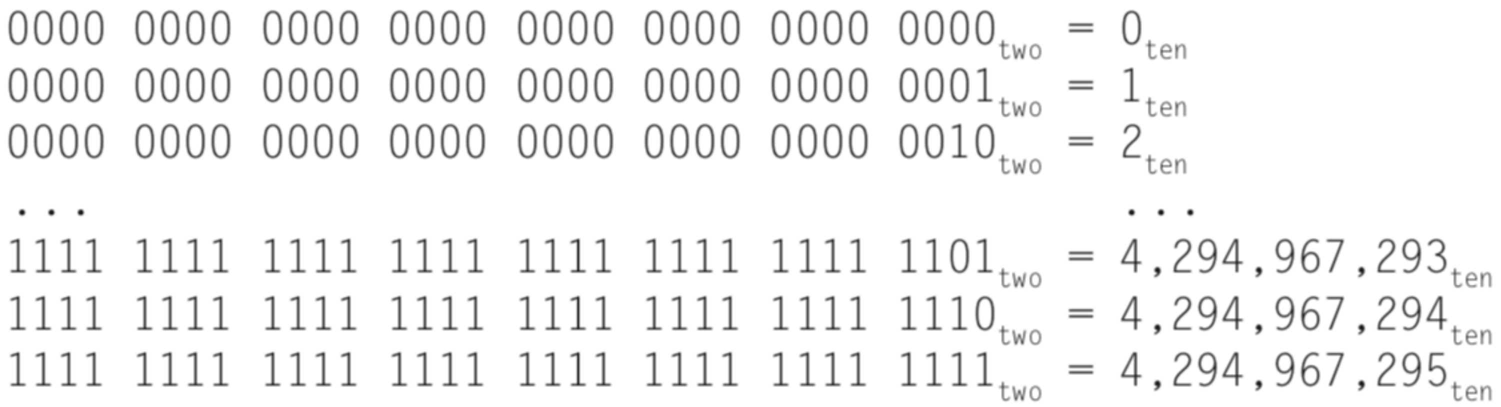
- 11 in a MISP word

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
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1

(32 bits wide)

Signed and Unsigned Numbers

- **Least significant bit:** is used to refer to the rightmost bit
- **Most significant bit:** is used to refer to the leftmost bit
- The MIPS word is 32 bits long, so we can represent 2^32 different 32-bit patterns



$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

$\ldots \qquad \ldots$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 4{,}294{,}967{,}293_{ten}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 4{,}294{,}967{,}294_{ten}$

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 4{,}294{,}967{,}295_{ten}$

2s-Complement Signed Integers

- We need a representation that distinguish positive numbers from negative number
- Sign bit: consider a bit as a sign bit (leftmost or rightmost bit); 0 means positive, 1 means negative
- Two ’s complement representation:

$$
\begin{aligned}
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} &&= 0_{ten}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} &&= 1_{ten}\\
&0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} &&= 2_{ten}\\
&\ldots && \ldots\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} &&= 2{,}147{,}483{,}645_{ten}\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} &&= 2{,}147{,}483{,}646_{ten}\\
&0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} &&= 2{,}147{,}483{,}647_{ten}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} &&= -2{,}147{,}483{,}648_{ten}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} &&= -2{,}147{,}483{,}647_{ten}\\
&1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} &&= -2{,}147{,}483{,}646_{ten}\\
&\ldots && \ldots\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} &&= -3_{ten}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} &&= -2_{ten}\\
&1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} &&= -1_{ten}
\end{aligned}
$$

2s-Complement Signed Integers

- We can represent positive and negative 32-bit numbers in terms of the bit value times a power of 2:

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \ldots + (x1 \times 2^{1}) + (x0 \times 2^{0})$$

- Example:

What is the decimal value of this 32-bit two's complement number?

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_{\text{two}}$$

Substituting the number's bit values into the formula above:

$$\begin{aligned}&(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{1}) + (0 \times 2^{1}) + (0 \times 2^{0})\\&= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0\\&= -2{,}147{,}483{,}648_{\text{ten}} + 2{,}147{,}483{,}644_{\text{ten}}\\&= -4_{\text{ten}}\end{aligned}$$

Negation shortcut

Negate 2_{ten}, and then check the result by negating -2_{ten}.

2_{ten} = $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two}$

Negating this number by inverting the bits and adding one,

$$\begin{array}{lr} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} \\ + & 1_{two} \\ \hline = & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} \\ = & -2_{ten} \end{array}$$

Going the other direction,

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two}$

is first inverted and then incremented:

$$\begin{array}{lr} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} \\ + & 1_{two} \\ \hline = & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} \\ = & 2_{ten} \end{array}$$

Signed extension shortcut

Convert 16-bit binary versions of 2_{ten} and -2_{ten} to 32-bit binary numbers.

The 16-bit binary version of the number 2 is

$0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

It is converted to a 32-bit number by making 16 copies of the value in the most significant bit (0) and placing that in the left-hand half of the word. The right half gets the old value:

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$

Let's negate the 16-bit version of 2 using the earlier shortcut. Thus,

$0000\ 0000\ 0000\ 0010_{two}$

becomes

$$\begin{array}{lr} & 1111\ 1111\ 1111\ 1101_{two} \\ + & 1_{two} \\ \hline \end{array}$$

$= 1111\ 1111\ 1111\ 1110_{two}$

Creating a 32-bit version of the negative number means copying the sign bit 16 times and placing it on the left:

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$

Reading assignment

- Read 2.1, 2.2, 2.3, and 2.4 of the text book