Chapter II: Instruction - Language of Computer:

"The world in language is **Instruction**, and its vocabulary is **Instructon set**"

Operation of the Computer Hardware:

- Every computer has to perform arithmetic. The MIPS assembly language notation:

```
add a, b, c
```

- This instruction will perform addition of b and c then put their sum in a.
- This notation is rigid in that each MIPS arithmetic instruction perform only one operation and must always have exactly 3 variables.

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

→ It take three instructions to perform instruction adds of 4 variables

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	`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

- Each line of Assembly language can contain at most 1 instruction.
- Comments always terminate at the end of the line

Design Priciple I: Simplicity favors Regularity

Example:

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

Solution:
At first the compiler would break this statement into several MIPS instructions:

```
add t0, g, h # tempory variable t0 contain g + h:
add t1, i, j # tempory varialbe t1 contain j + j:
```

Then we perform the subtraction:

```
sub f, t0, t1 # get t0 - t1 equals to (g + h) - (i + j):
```

Operand of the Computer Hardware:

"Unlike others high-level programming language, the operand of arithmetic instructions are restricted; they must be a limited number of special location built directly in hardware called *register*".

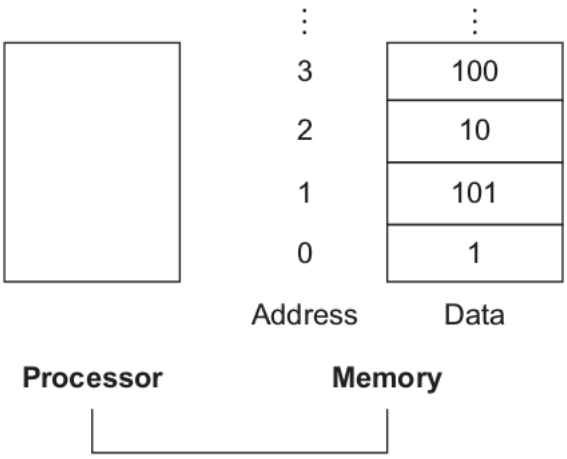
- *Registers* are primitives used in hardware design that are also visable to programmer when the computer is completed.
- The size of *a register* in the MIPS architecture is 32 bits. Groups of 32 bits occur so frequently that they are called **word**.
- The major difference between variable and register is the limited number of registers, typically 32 on current computer like MIPS.

Design Priciple II: Smaller is faster

- A very large number of registers may increase the clock cycle time supply because it take electric signals longer when they make travel farther.

Memory Operand:

- Since arithmetic operations occur only in MIPS instructions, thus MIPS must include instructions that transfer data between memory and register. Such instructions are called **data transfer instructions**.
- To access the word in memory, the instruction must suppply the memory **memory address**.



- The data transfer instruction that copies data from memory to register is called **load**. The actual name for this instruction in MIPS is *lw*, standing for *load word*.
- In MIPS, words must start at address that are multiples of 4. This requirement is called **alignment restriction**.

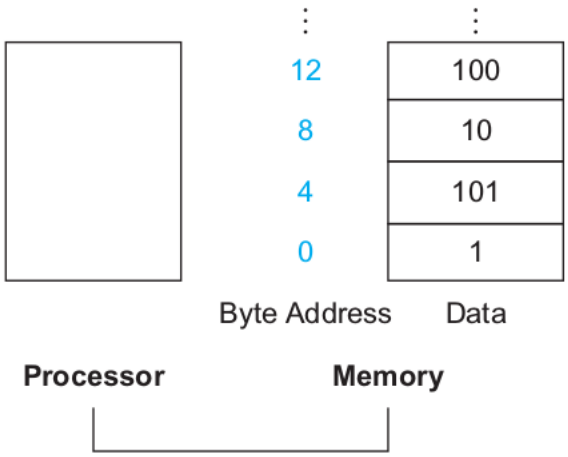


FIGURE 2.3 Actual MIPS memory addresses and contents of memory for those words. The changed addresses are highlighted to contrast with Figure 2.2. Since MIPS addresses each byte, word addresses are multiples of 4: there are 4 bytes in a word.

Constant or Immediate Operand:

- To use constant as an operand we woudl have to load a constant from memory to use:

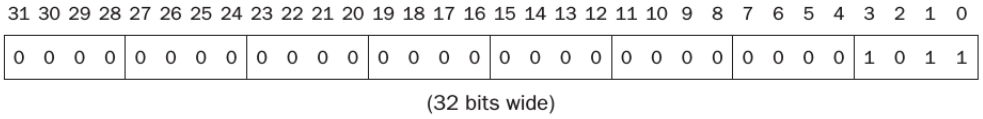
```
lw $t0, AddrConstant4($s1) # $t0 = constant 4
add $s3, $s3, $t0 # $s3 = $s3 + $t0 ($t0 = 4)
```

- Alternative method to add constant is using *add immediate* or *ad*:

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

Signed and Unsigned Number:

- Number of bits within a MIPS word and the placement of the the number 1011_{two}:



(32 bits wide)

- The MIPS word is 32 bits long, so we can represent 2^{32} different 32-bit patterns.
- The way to represent the signed and unsigned number is called *two's complement* representation:

  - The leading 0s represent for *positive*
  - THe leading 1s represent for *negative*

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$$

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$$
$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$$

Binary to Decimal Conversion

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

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

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

Ex:
$$(1 \times -2^{31}) + (1 \times 2^{30}) + (1 \times 2^{29}) + \ldots + (1 \times 2^{1}) + (0 \times 2^{1}) + (0 \times 2^{0})$$
$$= -2^{31} + 2^{30} + 2^{29} + \ldots + 2^{2} + 0 + 0$$
$$= -2{,}147{,}483{,}648_{ten} + 2{,}147{,}483{,}644_{ten}$$
$$= -4_{ten}$$

Revision: Binary Addition

a	b	a + b	carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1