Chapter 2

Instructions: Language of the Computer

halfwords

- The MIPS instruction set has explicit instructions to load and store such 16- bit quantities, called *halfwords*.

- ***Load half*** **(lh)** loads a halfword from memory, placing it in the rightmost 16 bits of a register.

- Like load byte, ***load half*** **(lh)** treats the halfword as a signed number and thus sign-extends to fill the 16 leftmost bits of the register
- ***load halfword unsigned*** **(lhu)** works with unsigned integers.

halfwords

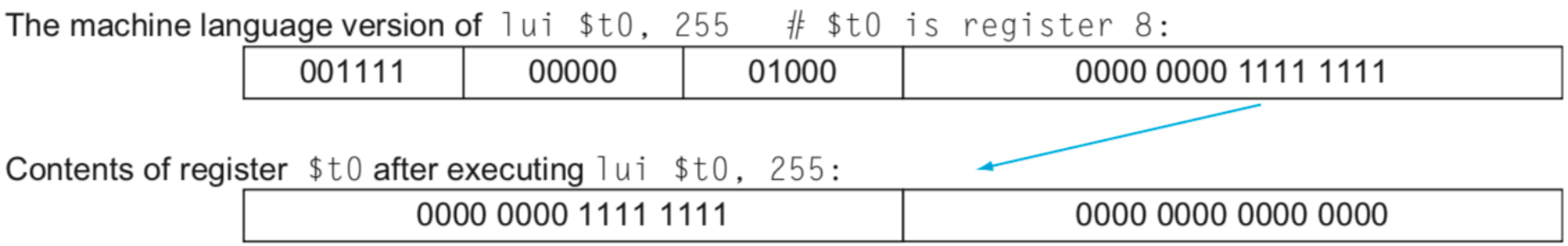
- The MIPS instruction set has explicit instructions to load and store such 16- bit quantities, called *halfwords*.

- ***Store half*** **(sh)** takes a halfword from the rightmost 16 bits of a register and writes it to memory.

```
lhu $t0,0($sp) # Read halfword (16 bits) from source
sh $t0,0($gp)  # Write halfword (16 bits) to destination
```

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
  - *load upper immediate* (lui) set the upper 16 bits of a constant in a register
  - a subsequent instruction specify the lower 16 bits of the constant



32-bit Constants

What is the MIPS assembly code to load this 32-bit constant into register `$s0`?

```
0000 0000 0011 1101 0000 1001 0000 0000
```

First, we would load the upper 16 bits, which is 61 in decimal, using `lui`:

```
lui $s0, 61   # 61 decimal = 0000 0000 0011 1101 binary
```

The value of register `$s0` afterward is

```
0000 0000 0011 1101 0000 0000 0000 0000
```

The next step is to insert the lower 16 bits, whose decimal value is 2304:

```
ori $s0, $s0, 2304 # 2304 decimal = 0000 1001 0000 0000
```

The final value in register `$s0` is the desired value:

```
0000 0000 0011 1101 0000 1001 0000 0000
```

MIPS provides the instructions *and immediate* (andi) and *or immediate* (ori).

Addressing in Branches and Jumps

- J type Instruction format:
  - Consists of 6 bits for the operation field and the rest of the bits for the address field

```
j    10000    # go to location 10000
```

- Unlike the jump instruction, the conditional branch instruction must specify two operands in addition to the branch address

```
bne  $s0,$s1,Exit  # go to Exit if $s0 ≠ $s1
```

Addressing in Branches and Jumps

- If addresses of the program had to fit in this 16-bit field, it would mean that no program could be bigger than 2^16, which is far too small to be a realistic option today.

- An alternative would be to specify a register that would always be added to the branch address, so that a branch instruction would calculate the following:

- Program counter= Register+ Branch address

Addressing in Branches and Jumps

- This sum allows the program to be as large as 2^32 and still be able to use conditional branches, solving the branch address size problem. Then the question is, which register?
- Since the *program counter* (PC) contains the address of the current instruction, **we can branch within 2^16 words of the current instruction** if we use the PC as the register to be added to the address.
- Almost all loops and *if* statements are much smaller than 2^16 words, so the PC is the ideal choice.
- This form of branch addressing is called **PC-relative addressing**

Addressing in Branches and Jumps

- MIPS uses **PC-relative addressing for all conditional branches**, because the destination of these instructions is likely to be close to the branch.
- On the other hand, **jump-and-link instructions invoke procedures** that have no reason to be near the call, so they normally use other forms of addressing.
- Hence, the MIPS architecture offers long addresses for procedure calls by using **the J-type format** for both **jump** and **jump-and-link instructions.**

Addressing in Branches and Jumps

The *while* loop on pages 92–93 was compiled into this MIPS assembler code:

```
Loop:sll $t1,$s3,2        # Temp reg $t1 = 4 * i
     add  $t1,$t1,$s6     # $t1 = address of save[i]
     lw   $t0,0($t1)      # Temp reg $t0 = save[i]
     bne  $t0,$s5, Exit   # go to Exit if save[i] ≠ k
     addi $s3,$s3,1       # i = i + 1
     j    Loop            # go to Loop
Exit:
```

If we assume we place the loop starting at location 80000 in memory, what is the MIPS machine code for this loop?

The assembled instructions and their addresses are:

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

Addressing in Branches and Jumps

```
Loop:sll $t1,$s3,2      # Temp reg $t1 = 4 * i
     add  $t1,$t1,$s6   # $t1 = address of save[i]
     lw   $t0,0($t1)    # Temp reg $t0 = save[i]
     bne  $t0,$s5, Exit # go to Exit if save[i] ≠ k
     addi $s3,$s3,1     # i = i + 1
     j    Loop          # go to Loop
Exit:
```

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024	...					

Remember that MIPS instructions have byte addresses, so addresses of sequential words differ by 4, the number of bytes in a word. The `bne` instruction on the fourth line adds 2 words or 8 bytes to the address of the *following* instruction (80016), specifying the branch destination relative to that following instruction (8 + 80016) instead of relative to the branch instruction (12 + 80012) or using the full destination address (80024). The jump instruction on the last line does use the full address ($20000 \times 4 = 80000$), corresponding to the label `Loop`.

Branching Far Away

Given a branch on register `$s0` being equal to register `$s1`,

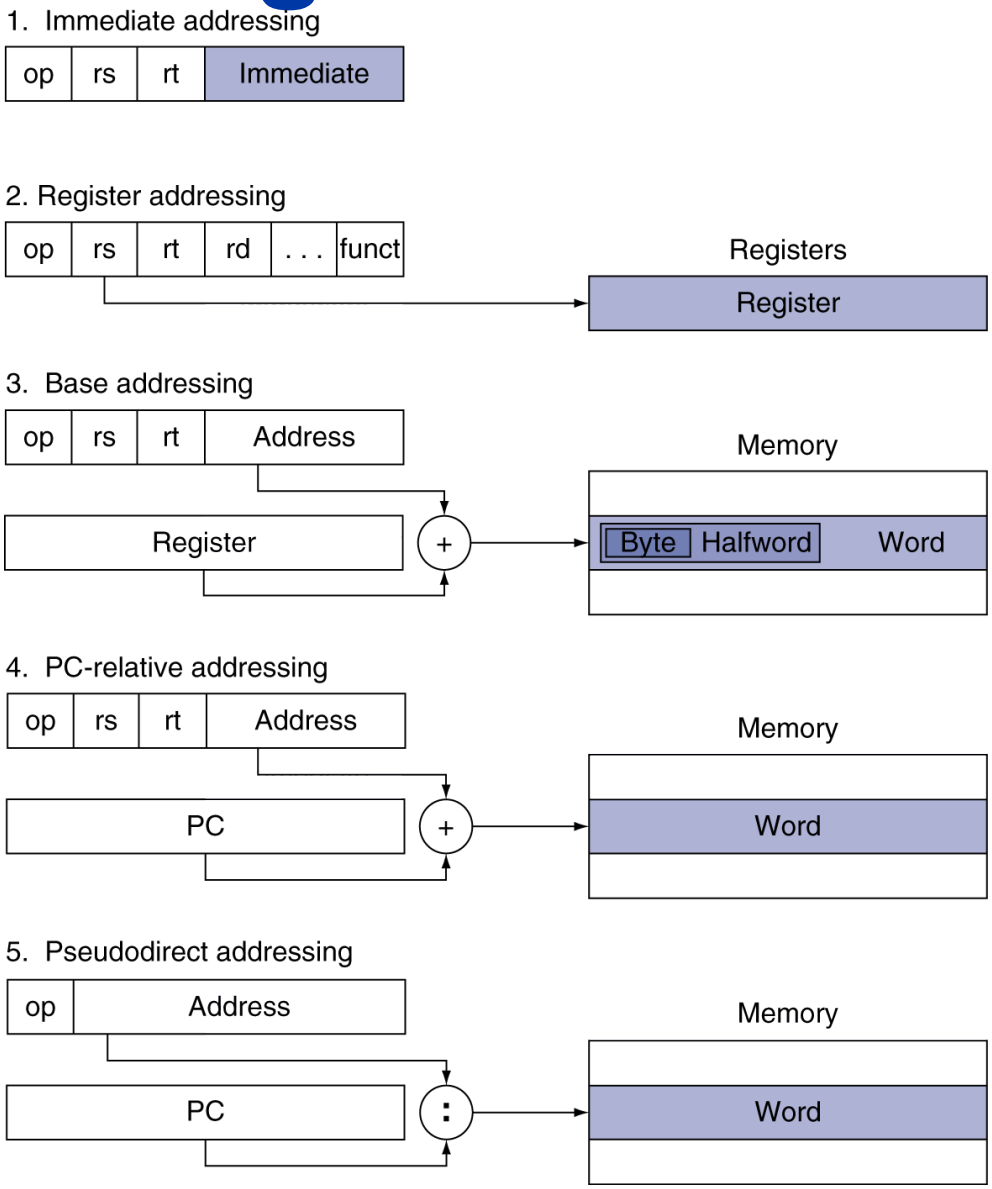
```
beq     $s0, $s1, L1
```

replace it by a pair of instructions that offers a much greater branching distance.

These instructions replace the short-address conditional branch:

```
        bne     $s0, $s1, L2
        j       L1
L2:
```

Addressing Mode Summary



Reading assignment

- Read 2.9 and 2.10 of the text book