Loading Constants into a Register

If the constant will fit into 16 bits, use \texttt{li} (load immediate)
\begin{verbatim}
li $14,8 \# $14 = 8
\end{verbatim}

- \texttt{li} is a pseudoinstruction for something like:
\begin{verbatim}
addi $14,$0,8
\end{verbatim}

\begin{verbatim}
or
\end{verbatim}

or \begin{verbatim}
$14,$0,8
\end{verbatim}

If the constant does not fit into 16 bits, use \texttt{lui} (load upper immediate)
- \texttt{lui} puts a constant in the most significant halfword
\begin{verbatim}
lui rt, immed \# rt<31,16> = immed
\end{verbatim}
\begin{verbatim}
\# rt<15,0> = 0
\end{verbatim}

- \texttt{addi} (or \texttt{ori}) puts a constant into the least significant halfword

Example: load the constant 0x1b236723 into $t0
\begin{verbatim}
lui $t0,0x1b23
ori $t0,$t0,0x6723
\end{verbatim}
Getting the Base Address into a Register

Method 1: address is a value in memory

```assembly
.data # define the data section
xyz: .word 1 # store the value 1 here
... # some other data
.text # define the code section
...
# lines of code
lw $5,xyz # loads contents of xyz into $5
```

(the assembler generates `lw $5,offset($gp)`)

Method 2: use `la` & the symbolic name for the location

- loads an address rather than the contents of the address
- `la` is a pseudoinstruction, but `lui` followed by `addi`
- example:

```assembly
la $6,xyz # $6 contains the address of memory location xyz
lw $5,0($6) # $5 contains the contents of memory location xyz
```

Method 3: the address is a constant & you know what it is

- use `li` (if $\pm 32K$
- use `lui` and `addi` (or `ori`) otherwise
Masking with Logical Instructions

Use masks
  • to extract smaller information units from a word
  • to set certain bits to 0 or 1 while retaining other bits as they are

Example: create a mask of all 1’s for the low-order byte of $6 --- don’t care about the other bits
ori $6,$6,0x000000ff       # set $6<7:0> to 1’s

Example: use a mask to clear the high-order byte of $6 but leave the 3 other bytes the same
lui $5,0x000000ff           # set $5<23:16> to 1’s,
# $5<31:24> and the other
# bits to 0’s
ori $5,$5,0x0000ffff        # set $5<15:0> to 1’s
and $6,$6,$5                # clear the high-order byte
Shifting

Arithmetic shifts to the right: the sign bit is extended
Logical shifts & arithmetic shifts to the left: zeros are shifted in

Examples:

$5$ contains: 1111 1111 0000 0000 0000 0000 0000 0000

srl $5,5,6$  
# shift right logical 6 bits
# $5 = 0000 0011 1111 1100 0000...

sra $5,5,6$  
# shift right arithmetic 6 bits
# $5 = 1111 1111 1111 1100 0000...
HI & LO

Used for holding the product of a multiply (multiplying two 32-bit numbers may yield a 64-bit product)

- HI gets the upper 32 result bits
- LO gets the lower 32

Used for the quotient and remainder of a divide

- LO gets the quotient
- HI gets the remainder
- if an operand is negative, the remainder is not specified by the MIPS architecture

Instructions to move between HI/LO & the GPRs.

\begin{verbatim}

mfhi rd       # move from HI to rd
mflo rd       # move from LO to rd
mthi rd       # move to HI from rd
mtlo rd       # move to LO from rd
mul rd,rs,rt  # a pseudoinstruction for mult rs,rt
mflo rd

\end{verbatim}
Addressing Modes

A function to calculate the address of an operand

**operand specifier** vs. **operand**

MIPS has few (RISC again)

- **register** addressing
  - operand specifier is a register number
  - operand is the register contents

- **immediate** addressing
  - operand specifier/operand is a constant in the instruction stream

- **base or displacement** addressing
  - operand specifier is a register contents plus a constant in the instruction
  - operand is the contents of the memory location whose address is that specifier
Addressing Modes

- **PC-relative** addressing
  - operand specifier is the contents of the PC plus a constant in the instruction
  - operand is the instruction at the memory location whose address is that specifier

- **pseudodirect** addressing
  - operand specifier is the address in the jump instruction
  - operand is the instruction at the memory location whose address is that specifier concatenated with the upper bits of the PC & 2 low-order 0s
Addressing Modes

User-generated addressing modes:
- register, immediate, displacement, pseudodirect

Compiler & assembler-generated addressing modes
- PC-relative
- example:

```assembly
loop:    lw  $8, offset($9)
         bne $8, $21, exit    # 2 instructions
         add $19,$19,20
         j    loop          # -4 instructions

exit:
```

+ need fewer bits to specify the operand address
+ **position-independent code**: can load anywhere in memory

- why programmers don’t use PC-relative
  ```assembly
  bne    $8, $21, 2($pc)
  ```

If you insert additional code here, you **must change the hardcoded displacement**...... Ack!
Other Addressing Modes

Indexed addressing
- use 2 registers as the operand specifier
- \texttt{lw \ $t1, $s1, $s2} \quad \# \ $t1 \text{ gets Memory}[\$s1+\$s2]
- in MIPS : \texttt{add \ $s0, $s1, $s2}
  \quad \texttt{lw \ $t1, 0($s0)}

Update addressing
- increment the memory address as part of a data transfer
  - autoincrement, autodecrement
  - useful when marching through an array
- \texttt{lwu \ $t1, 0($s0)} \quad \# \ $t1 \text{ gets Memory}[$s0] ; \quad \$s0 = \$s0 + 4
- in MIPS : \texttt{lw \ $t1, 0($s0)}
  \quad \texttt{addi \ $s0, $s0, 4}
A Longer Example

High-level language version

```c
int a[100];
int i;
for (i=0; i<100; i++) {
    a[i] = 5;
}
```

Assembly language version

- base address of array a in $15
- $8 contains the value of i, $9 the value 5

```
add $8,$0,$0 # initialize i
li $9,5 # $9 has the constant 5
loop:
    sla $10,$8,2 # $10 has i in bytes
    addu $14,$10,$15 # address of a[i]
    sw $9,0($14) # store 5 in a[i]
    addiu $8,$8,1 # increment i
    blt $8,100,loop # branch if loop not finished
```
A Longer Example

Machine-language version generated by a compiler

[0x00400020] 0x00004020   add $8,$0,$0
[0x00400024] 0x34090005   ori $9,$0,55
[0x00400028] 0x34010004   ori $1,$0,4
[0x0040002c] 0x01010018   mult $8,$1
[0x00400030] 0x00005012   mflo $10
[0x00400034] 0x014f7021   addu $14,$10,$15
[0x00400038] 0xadc90000   sw $9,0($14)
[0x0040003c] 0x25080001   addiu $8,$8,1
[0x00400040] 0x20100064   slti $2,$8,100
[0x00400044] 0x1420fff9   bne $2,$0,-28
A Longer Example

Machine-language version generated by a compiler

[0x00400020] 0x00004020 add $8,$0,$0 ; same
[0x00400024] 0x34090005 ori $9,$0,5 ; li $9,5
[0x00400028] 0x34010004 ori $1,$0,4 ; sla $10,$8,2
[0x0040002c] 0x01010018 mult $8,$1 ; (loop head)
[0x00400030] 0x00005012 mflo $10
[0x00400034] 0x014f7021 addu $14,$10,$15; same
[0x00400038] 0xad90000 sw $9,0($14) ; same
[0x0040003c] 0x25080001 addiu $8,$8,1 ; same
[0x00400040] 0x2010064 slti $2,$8,100 ; blt $8,100,Loop
[0x00400044] 0x1420fff9 bne $2,$0,-28
Assembly Language Programming
or
How to be Nice to Your TA

• Use lots of detailed comments
• Don’t be too fancy
• Use lots of detailed comments
• Use words whenever possible
• Use lots of detailed comments
• Remember that the address of a word is evenly divisible by 4
• Use lots of detailed comments
• The word following the word at address is at address + 4.
• Use lots of detailed comments