Part II

Instruction-Set Architecture

<table>
<thead>
<tr>
<th>Parts</th>
<th>Chapters</th>
</tr>
</thead>
<tbody>
<tr>
<td>I. Background and Motivation</td>
<td>1. Combinational Digital Circuits</td>
</tr>
<tr>
<td></td>
<td>2. Digital Circuits with Memory</td>
</tr>
<tr>
<td></td>
<td>3. Computer System Technology</td>
</tr>
<tr>
<td></td>
<td>4. Computer Performance</td>
</tr>
<tr>
<td>II. Instruction-Set Architecture</td>
<td>5. Instructions and Addressing</td>
</tr>
<tr>
<td></td>
<td>6. Procedures and Data</td>
</tr>
<tr>
<td></td>
<td>7. Assembly Language Programs</td>
</tr>
<tr>
<td></td>
<td>8. Instruction-Set Variations</td>
</tr>
<tr>
<td>III. The Arithmetic/Logic Unit</td>
<td>9. Number Representation</td>
</tr>
<tr>
<td></td>
<td>10. Adders and Simple ALUs</td>
</tr>
<tr>
<td></td>
<td>11. Multipliers and Dividers</td>
</tr>
<tr>
<td></td>
<td>12. Floating-Point Arithmetic</td>
</tr>
<tr>
<td>IV. Data Path and Control</td>
<td>13. Instruction Execution Steps</td>
</tr>
<tr>
<td></td>
<td>14. Control Unit Synthesis</td>
</tr>
<tr>
<td></td>
<td>15. Pipelined Data Paths</td>
</tr>
<tr>
<td></td>
<td>16. Pipeline Performance Limits</td>
</tr>
<tr>
<td>V. Memory System Design</td>
<td>17. Main Memory Concepts</td>
</tr>
<tr>
<td></td>
<td>18. Cache Memory Organization</td>
</tr>
<tr>
<td></td>
<td>19. Mass Memory Concepts</td>
</tr>
<tr>
<td></td>
<td>20. Virtual Memory and Paging</td>
</tr>
<tr>
<td>VI. Input/Output and Interfacing</td>
<td>21. Input/Output Devices</td>
</tr>
<tr>
<td></td>
<td>22. Input/Output Programming</td>
</tr>
<tr>
<td></td>
<td>23. Buses, Links, and Interfacing</td>
</tr>
<tr>
<td></td>
<td>24. Context Switching and Interrupts</td>
</tr>
<tr>
<td>VII. Advanced Architectures</td>
<td>25. Road to Higher Performance</td>
</tr>
<tr>
<td></td>
<td>26. Vector and Array Processing</td>
</tr>
<tr>
<td></td>
<td>27. Shared-Memory Multiprocessing</td>
</tr>
<tr>
<td></td>
<td>28. Distributed Multicomputing</td>
</tr>
</tbody>
</table>
This presentation is intended to support the use of the textbook *Computer Architecture: From Microprocessors to Supercomputers*, Oxford University Press, 2005, ISBN 0-19-515455-X. It is updated regularly by the author as part of his teaching of the upper-division course ECE 154, Introduction to Computer Architecture, at the University of California, Santa Barbara. Instructors can use these slides freely in classroom teaching and for other educational purposes. Any other use is strictly prohibited. © Behrooz Parhami

<table>
<thead>
<tr>
<th>Edition</th>
<th>Released</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
<th>Revised</th>
</tr>
</thead>
</table>
A Few Words About Where We Are Headed

Performance = 1 / Execution time  

simplified to  

1 / CPU execution time

CPU execution time = Instructions × CPI / (Clock rate)

Performance = Clock rate / (Instructions × CPI)

Try to achieve CPI = 1 with clock that is as high as that for CPI > 1 designs; is CPI < 1 feasible? (Chap 15-16)

Define an instruction set; make it simple enough to require a small number of cycles and allow high clock rate, but not so simple that we need many instructions, even for very simple tasks (Chap 5-8)

Design hardware for CPI = 1; seek improvements with CPI > 1 (Chap 13-14)

Design memory & I/O structures to support ultrahigh-speed CPUs

Design ALU for arithmetic & logic ops (Chap 9-12)
II Instruction Set Architecture

Introduce machine “words” and its “vocabulary,” learning:
- A simple, yet realistic and useful instruction set
- Machine language programs; how they are executed
- RISC vs CISC instruction-set design philosophy

<table>
<thead>
<tr>
<th>Topics in This Part</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 5 Instructions and Addressing</td>
<td></td>
</tr>
<tr>
<td>Chapter 6 Procedures and Data</td>
<td></td>
</tr>
<tr>
<td>Chapter 7 Assembly Language Programs</td>
<td></td>
</tr>
<tr>
<td>Chapter 8 Instruction Set Variations</td>
<td></td>
</tr>
</tbody>
</table>
5 Instructions and Addressing

First of two chapters on the instruction set of MiniMIPS:
- Required for hardware concepts in later chapters
- Not aiming for proficiency in assembler programming

### Topics in This Chapter

<table>
<thead>
<tr>
<th>Section</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Abstract View of Hardware</td>
</tr>
<tr>
<td>5.2</td>
<td>Instruction Formats</td>
</tr>
<tr>
<td>5.3</td>
<td>Simple Arithmetic/Logic Instructions</td>
</tr>
<tr>
<td>5.4</td>
<td>Load and Store Instructions</td>
</tr>
<tr>
<td>5.5</td>
<td>Jump and Branch Instructions</td>
</tr>
<tr>
<td>5.6</td>
<td>Addressing Modes</td>
</tr>
</tbody>
</table>
5.1 Abstract View of Hardware

Figure 5.1 Memory and processing subsystems for MiniMIPS.
Data Types

Byte = 8 bits

Halfword = 2 bytes

Word = 4 bytes

Doubleword = 8 bytes

Quadword (16 bytes) also used occasionally

MiniMIPS registers hold 32-bit (4-byte) words. Other common data sizes include byte, halfword, and doubleword.
Register Conventions

A doubleword sits in consecutive memory addresses according to the big-endian order (most significant byte has the lowest address)

Byte numbering:

When loading a byte into a register, it goes in the low end

A 4-byte word sits in consecutive memory addresses according to the big-endian order (most significant byte has the lowest address)

When loading a byte into a register, it goes in the low end

A doubleword sits in consecutive registers or memory locations according to the big-endian order (most significant word comes first)

Figure 5.2
Registers and data sizes in MiniMIPS.
Registers Used in This Chapter

Figure 5.2 (partial)

10 temporary registers

Temporary values

8 operand registers

Saved across procedure calls

Operands

More temporaries

Wallet

Change

Analogy for register usage conventions

Wallet

Keys

Figure 5.2 (partial)
5.2 Instruction Formats

High-level language statement:

`a = b + c`

Assembly language instruction:

`add $t8, $s2, $s1`

Machine language instruction:

```
000000 10010 10001 11000 00000 100000
```

**Figure 5.3** A typical instruction for MiniMIPS and steps in its execution.
Add, Subtract, and Specification of Constants

MiniMIPS add & subtract instructions; e.g., compute:

\[ g = (b + c) - (e + f) \]

```assembly
add $t8,$s2,$s3  # put the sum b + c in $t8
add $t9,$s5,$s6  # put the sum e + f in $t9
sub $s7,$t8,$t9  # set g to ($t8) - ($t9)
```

Decimal and hex constants

- **Decimal**: 25, 123456, −2873
- **Hexadecimal**: 0x59, 0x12b4c6, 0xffffffff

Machine instruction typically contains

- an opcode
- one or more source operands
- possibly a destination operand
MiniMIPS Instruction Formats

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sh</th>
<th>fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
<tr>
<td></td>
<td>Opcode</td>
<td>Source register 1</td>
<td>Source register 2</td>
<td>Destination register</td>
<td>Shift amount</td>
<td>Opcode extension</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>operand / offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
<tr>
<td></td>
<td>Opcode</td>
<td>Source or base</td>
<td>Destination or data</td>
<td>Immediate operand or address offset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>op</th>
<th>jump target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>J</td>
<td>6 bits</td>
<td>26 bits</td>
</tr>
<tr>
<td></td>
<td>Opcode</td>
<td>Memory word address (byte address divided by 4)</td>
</tr>
</tbody>
</table>

Figure 5.4 MiniMIPS instructions come in only three formats: register (R), immediate (I), and jump (J).
5.3 Simple Arithmetic/Logic Instructions

Add and subtract already discussed; logical instructions are similar

```plaintext
add  $t0,$s0,$s1  # set $t0 to ($s0)+($s1)
sub  $t0,$s0,$s1  # set $t0 to ($s0)-(s1)
and  $t0,$s0,$s1  # set $t0 to ($s0)∧($s1)
or   $t0,$s0,$s1  # set $t0 to ($s0)∨($s1)
xor  $t0,$s0,$s1  # set $t0 to ($s0)⊕($s1)
nor  $t0,$s0,$s1  # set $t0 to (($s0)∨($s1))'
```

Figure 5.5 The arithmetic instructions add and sub have a format that is common to all two-operand ALU instructions. For these, the fn field specifies the arithmetic/logic operation to be performed.
Arithmetic/Logic with One ImmediateOperand

An operand in the range $[-32768, 32767]$, or $[0x0000, 0xffff]$, can be specified in the immediate field.

\[
\text{addi} \quad \text{\texttt{$t0,$s0,61}} \quad \# \text{ set } \texttt{$t0} \text{ to } (\texttt{$s0})+61 \\
\text{andi} \quad \text{\texttt{$t0,$s0,61}} \quad \# \text{ set } \texttt{$t0} \text{ to } (\texttt{$s0})\wedge 61 \\
\text{ori} \quad \text{\texttt{$t0,$s0,61}} \quad \# \text{ set } \texttt{$t0} \text{ to } (\texttt{$s0})\lor 61 \\
\text{xori} \quad \text{\texttt{$t0,$s0,0x00ff}} \quad \# \text{ set } \texttt{$t0} \text{ to } (\texttt{$s0})\oplus 0x00ff
\]

For arithmetic instructions, the immediate operand is sign-extended.

\[
\begin{array}{ccccccccc}
\hline
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1
\end{array}
\]

\text{addi} = 8 \quad \begin{array}{cccc}
& \text{op} & \text{rs} & \text{rt} & \text{operand / offset}
\end{array}
\begin{array}{cccc}
1 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
1 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
1 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
0 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
0 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
0 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
0 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
0 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
0 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
0 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
0 & 0 & 0 & 0
\end{array}
\begin{array}{cccc}
1 & 1 & 1 & 1
\end{array}
\begin{array}{cccc}
0 & 1
\end{array}
\begin{array}{cccc}
0 & 1
\end{array}
\begin{array}{cccc}
0 & 1
\end{array}
\begin{array}{cccc}
0 & 1
\end{array}
\begin{array}{cccc}
0 & 1
\end{array}

Figure 5.6 Instructions such as \text{addi} allow us to perform an arithmetic or logic operation for which one operand is a small constant.
5.4 Load and Store Instructions

Figure 5.7  MiniMIPS \texttt{lw} and \texttt{sw} instructions and their memory addressing convention that allows for simple access to array elements via a base address and an offset (offset = 4i leads us to the i\textsuperscript{th} word).

- \texttt{lw} $t0,40($s3)
- \texttt{lw} $t0,A($s3)

Note on base and offset:
The memory address is the sum of $rs$ and an immediate value. Calling one of these the base and the other the offset is quite arbitrary. It would make perfect sense to interpret the address $A($s3) as having the base $A$ and the offset ($s3). However, a 16-bit base confines us to a small portion of memory space.
**lw, sw, and lui Instructions**

### lw
\[ lw \quad $t0,40($s3) \quad \# \text{load mem}[40+($s3)] \text{ in } $t0 \]

### sw
\[ sw \quad $t0,A($s3) \quad \# \text{store } ($t0) \text{ in mem}[A+($s3)] \]

### lui
\[ lui \quad $s0,61 \quad \# \text{The immediate value 61 is loaded in upper half of } $s0 \quad \# \text{with lower 16b set to 0s} \]

---

**Figure 5.8** The **lui** instruction allows us to load an arbitrary 16-bit value into the upper half of a register while setting its lower half to 0s.
Initializing a Register

Example 5.2

Show how each of these bit patterns can be loaded into $s0:

0010 0001 0001 0000 0000 0000 0011 1101
1111 1111 1111 1111 1111 1111 1111 1111

Solution

The first bit pattern has the hex representation: 0x2110003d

    lui  $s0,0x2110 # put the upper half in $s0
    ori  $s0,0x003d # put the lower half in $s0

Same can be done, with immediate values changed to 0xffffffff for the second bit pattern. But, the following is simpler and faster:

    nor  $s0,$zero,$zero # because (0 ∨ 0)' = 1
5.5 Jump and Branch Instructions

Unconditional jump and jump through register instructions

\[
\begin{align*}
\text{j} & \quad \text{verify} & \quad \# \text{go to mem loc named “verify”} \\
\text{jr} & \quad \$ra & \quad \# \text{go to address that is in } \$ra; \\
& & \# \text{$ra may hold a return address}
\end{align*}
\]

$ra$ is the symbolic name for reg. $31$ (return address)

Figure 5.9 The jump instruction $j$ of MiniMIPS is a J-type instruction which is shown along with how its effective target address is obtained. The jump register ($jr$) instruction is R-type, with its specified register often being $\$ra$. 
Conditional Branch Instructions

Conditional branches use PC-relative addressing

\[
\begin{align*}
\text{bltz} & \quad \text{bltz } & \quad \text{branch on } (s_1) < 0 \\
\text{beq} & \quad \text{beq } & \quad \text{branch on } (s_1) = (s_2) \\
\text{bne} & \quad \text{bne } & \quad \text{branch on } (s_1) \neq (s_2)
\end{align*}
\]

Figure 5.10 (part 1) Conditional branch instructions of MiniMIPS.
Comparison Instructions for Conditional Branching

```
slt  $s1,$s2,$s3  # if ($s2)<($s3), set $s1 to 1
     # else set $s1 to 0;
     # often followed by beq/bne
slti $s1,$s2,61  # if ($s2)<61, set $s1 to 1
     # else set $s1 to 0
```

**Figure 5.10 (part 2)  Comparison instructions of MiniMIPS.**
Examples for Conditional Branching

If the branch target is too far to be reachable with a 16-bit offset (rare occurrence), the assembler automatically replaces the branch instruction `beq $s0,$s1,L1` with:

```
  bne $s1,$s2,L2  # skip jump if (s1)≠(s2)
  j L1           # goto L1 if (s1)=(s2)
L2: ...
```

Forming if-then constructs; e.g., `if (i == j) x = x + y`

```
  bne $s1,$s2,endif  # branch on i≠j
  add $t1,$t1,$t2    # execute the "then" part
endif: ...
```

If the condition were `(i < j)`, we would change the first line to:

```
  slt $t0,$s1,$s2    # set $t0 to 1 if i<j
  beq $t0,$0,endif # branch if ($t0)=0; # i.e., i not< j or i≥j
```
Compiling if-then-else Statements

Example 5.3

Show a sequence of MiniMIPS instructions corresponding to:

if (i<=j) x = x+1; z = 1; else y = y-1; z = 2*z

Solution

Similar to the “if-then” statement, but we need instructions for the “else” part and a way of skipping the “else” part after the “then” part.

```
slt  $t0,$s2,$s1  # j<i? (inverse condition)
bne  $t0,$zero,else  # if j<i goto else part
addi $t1,$t1,1  # begin then part: x = x+1
addi $t3,$zero,1  # z = 1
j  endif  # skip the else part
else: addi $t2,$t2,-1  # begin else part: y = y-1
      add  $t3,$t3,$t3  # z = z+z
endif:...
```
5.6 Addressing Modes

Figure 5.11  Schematic representation of addressing modes in MiniMIPS.
Finding the Maximum Value in a List of Integers

Example 5.5

List \textit{A} is stored in memory beginning at the address given in \$s1. List length is given in \$s2. Find the largest integer in the list and copy it into \$t0.

Solution

Scan the list, holding the largest element identified thus far in \$t0.

\begin{verbatim}
  lw    \$t0,0($s1)  # initialize maximum to A[0]
  addi  \$t1,$zero,0  # initialize index i to 0

loop: add   \$t1,$t1,1  # increment index i by 1
  beq   \$t1,$s2,done  # if all elements examined, quit
  add   \$t2,$t1,$t1    # compute 2i in \$t2
  add   \$t2,$t2,$t2    # compute 4i in \$t2
  add   \$t2,$t2,$s1    # form address of A[i] in \$t2
  lw    \$t3,0($t2)    # load value of A[i] into \$t3
  slt   \$t4,$t0,$t3    # maximum < A[i]?
  beq   \$t4,$zero,loop # if not, repeat with no change
  addi  \$t0,$t3,0      # if so, A[i] is the new maximum

j     loop # change completed; now repeat

done: ... # continuation of the program
\end{verbatim}
### The 20 MiniMIPS Instructions Covered So Far

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
<th>op</th>
<th>fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load upper immediate</td>
<td>lui rt,imm</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td>add rd,rs,rt</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>Subtract</td>
<td>sub rd,rs,rt</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>Set less than</td>
<td>slt rd,rs,rt</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>Add immediate</td>
<td>addi rt,rs,imm</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>Set less than immediate</td>
<td>slti rd,rs,imm</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>AND</td>
<td>and rd,rs,rt</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>OR</td>
<td>or rd,rs,rt</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>XOR</td>
<td>xor rd,rs,rt</td>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>NOR</td>
<td>nor rd,rs,rt</td>
<td>0</td>
<td>39</td>
</tr>
<tr>
<td>AND immediate</td>
<td>andi rt,rs,imm</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>OR immediate</td>
<td>ori rt,rs,imm</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>XOR immediate</td>
<td>xori rt,rs,imm</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Load word</td>
<td>lw rt,imm(rs)</td>
<td>35</td>
<td>43</td>
</tr>
<tr>
<td>Store word</td>
<td>sw rt,imm(rs)</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>j L</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Jump register</td>
<td>jr rs</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Branch less than 0</td>
<td>bltz rs,L</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Branch equal</td>
<td>beq rs,rt,L</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>Branch not equal</td>
<td>bne rs,rt,L</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

#### Table 5.1

<table>
<thead>
<tr>
<th>op</th>
<th>fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>0</td>
<td>38</td>
</tr>
<tr>
<td>0</td>
<td>39</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>43</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>
6 Procedures and Data

Finish our study of MiniMIPS instructions and its data types:
• Instructions for procedure call/return, misc. instructions
• Procedure parameters and results, utility of stack

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Simple Procedure Calls</td>
</tr>
<tr>
<td>6.2 Using the Stack for Data Storage</td>
</tr>
<tr>
<td>6.3 Parameters and Results</td>
</tr>
<tr>
<td>6.4 Data Types</td>
</tr>
<tr>
<td>6.5 Arrays and Pointers</td>
</tr>
<tr>
<td>6.6 Additional Instructions</td>
</tr>
</tbody>
</table>
6.1 Simple Procedure Calls

Using a procedure involves the following sequence of actions:

1. Put arguments in places known to procedure (reg’s $a0–$a3)
2. Transfer control to procedure, saving the return address (jal)
3. Acquire storage space, if required, for use by the procedure
4. Perform the desired task
5. Put results in places known to calling program (reg’s $v0–$v1)
6. Return control to calling point (jr)

MiniMIPS instructions for procedure call and return from procedure:

jal proc  # jump to loc “proc” and link;
# “link” means “save the return
# address” (PC)+4 in $ra ($31)
jr rs  # go to loc addressed by rs
Illustrating a Procedure Call

Figure 6.1  Relationship between the main program and a procedure.
Recalling Register Conventions

Figure 5.2
Registers and data sizes in MiniMIPS.
Example 6.1

A Simple MiniMIPS Procedure

Procedure to find the absolute value of an integer.

\[ v0 \leftarrow |(a0)| \]

**Solution**

The absolute value of \( x \) is \(-x\) if \( x < 0 \) and \( x \) otherwise.

```assembly
abs: sub $v0, $zero, $a0  # put -(a0) in $v0;
    # in case (a0) < 0
bltz $a0, done       # if (a0)<0 then done
add $v0, $a0, $zero  # else put (a0) in $v0
done: jr $ra            # return to calling program
```

In practice, we seldom use such short procedures because of the overhead that they entail. In this example, we have 3-4 instructions of overhead for 3 instructions of useful computation.
Nested Procedure Calls

Figure 6.2 Example of nested procedure calls.
6.2 Using the Stack for Data Storage

Figure 6.4 Effects of push and pop operations on a stack.

Push c

sp = sp – 4
mem[sp] = c

Pop x

x = mem[sp]
sp = sp + 4

Analogy: Cafeteria stack of plates/trays

push: addi $sp,$sp,-4
sw $t4,0($sp)

pop: lw $t5,0($sp)
addi $sp,$sp,4

Cafeteria stack of plates/trays
Memory Map in MiniMIPS

Hex address

Reserved

Program

Static data

Dynamic data

Stack

Hex address
00000000
00400000
10000000
1000ffff
10000000
00000000
00400000
7ffffffc
7ffffffc
80000000
80000000

Addressable with 16-bit signed offset

$gp
$sp
$fp

Second half of address space reserved for memory-mapped I/O

1 M words
Text segment 63 M words
Data segment
448 M words
Stack segment

Text segment
63 M words

Stack segment

Figure 6.3 Overview of the memory address space in MiniMIPS.
6.3 Parameters and Results

Stack allows us to pass/return an arbitrary number of values

Figure 6.5 Use of the stack by a procedure.
Example of Using the Stack

Saving $fp, $ra, and $s0 onto the stack and restoring them at the end of the procedure

```
proc:    sw    $fp,-4($sp)     # save the old frame pointer
        addi   $fp,$sp,0        # save ($sp) into $fp
        addi   $sp,$sp,-12      # create 3 spaces on top of stack
        sw    $ra,-8($fp)       # save ($ra) in 2nd stack element
        sw    $s0,-12($fp)      # save ($s0) in top stack element
        lw    $s0,-12($fp)      # put top stack element in $s0
        lw    $ra,-8($fp)       # put 2nd stack element in $ra
        addi   $sp,$fp,0        # restore $sp to original state
        lw    $fp,-4($sp)       # restore $fp to original state
        jr    $ra              # return from procedure
```
6.4 Data Types

Data size (number of bits), data type (meaning assigned to bits)

Signed integer: byte word
Unsigned integer: byte word
Floating-point number: word doubleword
Bit string: byte word doubleword

Converting from one size to another

<table>
<thead>
<tr>
<th>Type</th>
<th>8-bit number</th>
<th>Value</th>
<th>32-bit version of the number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsigned</td>
<td>0010 1011</td>
<td>43</td>
<td>0000 0000 0000 0000 0000 0000 0010 1011</td>
</tr>
<tr>
<td>Unsigned</td>
<td>1010 1011</td>
<td>171</td>
<td>0000 0000 0000 0000 0000 0000 1010 1011</td>
</tr>
<tr>
<td>Signed</td>
<td>0010 1011</td>
<td>+43</td>
<td>0000 0000 0000 0000 0000 0000 0010 1011</td>
</tr>
<tr>
<td>Signed</td>
<td>1010 1011</td>
<td>−85</td>
<td>1111 1111 1111 1111 1111 1111 1010 1011</td>
</tr>
</tbody>
</table>
**ASCII Characters**

Table 6.1  ASCII (American standard code for information interchange)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8-9</th>
<th>a-f</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUL</td>
<td>DLE</td>
<td>SP</td>
<td>0</td>
<td>@</td>
<td>P</td>
<td>`</td>
<td>p</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STX</td>
<td>DC2</td>
<td>&quot;</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENQ</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEL</td>
<td>ETB</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td></td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td></td>
<td>K</td>
<td>[</td>
<td>k</td>
<td>{</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td></td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CR</td>
<td>GS</td>
<td>-</td>
<td></td>
<td>M</td>
<td>]</td>
<td>m</td>
<td>}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SO</td>
<td>RS</td>
<td>.</td>
<td></td>
<td>N</td>
<td>^</td>
<td>n</td>
<td>~</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SI</td>
<td>US</td>
<td>/</td>
<td></td>
<td>O</td>
<td>_</td>
<td>o</td>
<td>DEL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

More controls
More symbols

8-bit ASCII code (col #, row #)\textsubscript{hex} e.g., code for + is (2b)\textsubscript{hex} or (0010 1011)\textsubscript{two}
Loading and Storing Bytes

Bytes can be used to store ASCII characters or small integers. MiniMIPS addresses refer to bytes, but registers hold words.

```
lb      $t0,8($s3)  # load rt with mem[8+($s3)]
         # sign-extend to fill reg
lbu     $t0,8($s3)  # load rt with mem[8+($s3)]
         # zero-extend to fill reg
sb      $t0,A($s3)  # LSB of rt to mem[A+($s3)]
```

Figure 6.6   Load and store instructions for byte-size data elements.
Meaning of a Word in Memory

A 32-bit word has no inherent meaning and can be interpreted in a number of equally valid ways in the absence of other cues (e.g., context) for the intended meaning.
6.5 Arrays and Pointers

Index: Use a register that holds the index \( i \) and increment the register in each step to effect moving from element \( i \) of the list to element \( i + 1 \)

Pointer: Use a register that points to (holds the address of) the list element being examined and update it in each step to point to the next element

Figure 6.8 Stepping through the elements of an array using the indexing method and the pointer updating method.
Selection Sort

Example 6.4

To sort a list of numbers, repeatedly perform the following:
Find the max element, swap it with the last item, move up the “last” pointer

Figure 6.9 One iteration of selection sort.
Selection Sort Using the Procedure $\text{max}$

Example 6.4 (continued)

```
sort: beq $a0,$a1,done  # single-element list is sorted
    jal max  # call the max procedure
    lw $t0,0($a1)    # load last element into $t0
    sw $t0,0($v0)    # copy the last element to max loc
    sw $v1,0($a1)    # copy max value to last element
    addi $a1,$a1,-4    # decrement pointer to last element
    j sort          # repeat sort for smaller list
done: ...                # continue with rest of program
```
6.6 Additional Instructions

MiniMIPS instructions for multiplication and division:

\[
\begin{align*}
\text{mult} & \quad s0, \ s1 \quad \# \text{set Hi, Lo to } (s0) \times (s1) \\
\text{div} & \quad s0, \ s1 \quad \# \text{set Hi to } (s0) \mod (s1) \\
& \quad \quad \# \text{and Lo to } (s0)/(s1) \\
\text{mfhi} & \quad t0 \quad \# \text{set } t0 \text{ to } (Hi) \\
\text{mflo} & \quad t0 \quad \# \text{set } t0 \text{ to } (Lo)
\end{align*}
\]

**Figure 6.10** The multiply (mult) and divide (div) instructions of MiniMIPS.

**Figure 6.11** MiniMIPS instructions for copying the contents of Hi and Lo registers into general registers.
Logical Shifts

MiniMIPS instructions for left and right shifting:

- \texttt{sll} $t0,$s1,2 \quad \# t0=(s1) left-shifted by 2
- \texttt{srl} $t0,$s1,2 \quad \# t0=(s1) right-shifted by 2
- \texttt{sllv} $t0,$s1,$s0 \quad \# t0=(s1) left-shifted by (s0)
- \texttt{srlv} $t0,$s1,$s0 \quad \# t0=(s1) right-shifted by (s0)

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig6.12}
\caption{The four logical shift instructions of MiniMIPS.}
\end{figure}
Unsigned Arithmetic and Miscellaneous Instructions

MiniMIPS instructions for unsigned arithmetic (no overflow exception):

- `addu $t0,$s0,$s1` # set $t0$ to ($s0)+($s1)
- `subu $t0,$s0,$s1` # set $t0$ to ($s0)$–($s1)
- `multu $s0,$s1` # set Hi,Lo to ($s0)×($s1)
- `divu $s0,$s1` # set Hi to ($s0)mod($s1)
  # and Lo to ($s0)/($s1)
- `addiu $t0,$s0,61` # set $t0$ to ($s0)+61;
  # the immediate operand is
  # sign extended

To make MiniMIPS more powerful and complete, we introduce later:

- `sra $t0,$s1,2` # sh. right arith (Sec. 10.5)
- `srav $t0,$s1,$s0` # shift right arith variable
- `syscall` # system call (Sec. 7.6)
The 20 MiniMIPS Instructions from Chapter 6 (40 in all so far)

### Table 6.2 (partial)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
<th>op</th>
<th>fn</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move from Hi</td>
<td>mfhi rd</td>
<td>0</td>
<td>16</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo rd</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>Add unsigned</td>
<td>addu rd,rs,rt</td>
<td>0</td>
<td>33</td>
</tr>
<tr>
<td>Subtract unsigned</td>
<td>subu rd,rs,rt</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>Multiply</td>
<td>mult rs,rt</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>Multiply unsigned</td>
<td>multu rs,rt</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Divide</td>
<td>div rs,rt</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>Divide unsigned</td>
<td>divu rs,rt</td>
<td>0</td>
<td>27</td>
</tr>
<tr>
<td>Add immediate unsigned</td>
<td>addiu rs,rt,imm</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Shift left logical</td>
<td>sll rd,rt,sh</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Shift right logical</td>
<td>srl rd,rt,sh</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Shift right arithmetic</td>
<td>sra rd,rt,sh</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Shift left logical variable</td>
<td>sllv rd,rt,rs</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Shift right logical variable</td>
<td>srlv rt,rd,rs</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Shift right arith variable</td>
<td>srav rd,rt,rd</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Load byte</td>
<td>lb rt,imm(rs)</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Load byte unsigned</td>
<td>lbu rt,imm(rs)</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Store byte</td>
<td>sb rt,imm(rs)</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>Jump and link</td>
<td>jal L</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>System call</td>
<td>syscall</td>
<td>0</td>
<td>12</td>
</tr>
</tbody>
</table>
### Table 6.2  The 37 + 3 MiniMIPS Instructions Covered So Far

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load upper immediate</td>
<td>lui rt,imm</td>
</tr>
<tr>
<td>Add</td>
<td>add rd,rs,rt</td>
</tr>
<tr>
<td>Subtract</td>
<td>sub rd,rs,rt</td>
</tr>
<tr>
<td>Set less than</td>
<td>slt rd,rs,rt</td>
</tr>
<tr>
<td>Add immediate</td>
<td>addi rt,rs,imm</td>
</tr>
<tr>
<td>Set less than immediate</td>
<td>slti rd,rs,imm</td>
</tr>
<tr>
<td>AND</td>
<td>and rd,rs,rt</td>
</tr>
<tr>
<td>OR</td>
<td>or rd,rs,rt</td>
</tr>
<tr>
<td>XOR</td>
<td>xor rd,rs,rt</td>
</tr>
<tr>
<td>NOR</td>
<td>nor rd,rs,rt</td>
</tr>
<tr>
<td>AND immediate</td>
<td>andi rt,rs,imm</td>
</tr>
<tr>
<td>OR immediate</td>
<td>ori rt,rs,imm</td>
</tr>
<tr>
<td>XOR immediate</td>
<td>xori rt,rs,imm</td>
</tr>
<tr>
<td>Load word</td>
<td>lw rt,imm(rs)</td>
</tr>
<tr>
<td>Store word</td>
<td>sw rt,imm(rs)</td>
</tr>
<tr>
<td>Jump</td>
<td>j L</td>
</tr>
<tr>
<td>Jump register</td>
<td>jr rs</td>
</tr>
<tr>
<td>Branch less than 0</td>
<td>bltz rs,L</td>
</tr>
<tr>
<td>Branch equal</td>
<td>beq rs,rt,L</td>
</tr>
<tr>
<td>Branch not equal</td>
<td>bne rs,rt,L</td>
</tr>
<tr>
<td>Move from Hi</td>
<td>mfhi rd</td>
</tr>
<tr>
<td>Move from Lo</td>
<td>mflo rd</td>
</tr>
<tr>
<td>Add unsigned</td>
<td>addu rd,rs,rt</td>
</tr>
<tr>
<td>Subtract unsigned</td>
<td>subu rd,rs,rt</td>
</tr>
<tr>
<td>Multiply</td>
<td>mult rs,rt</td>
</tr>
<tr>
<td>Multiply unsigned</td>
<td>multu rs,rt</td>
</tr>
<tr>
<td>Divide</td>
<td>div rs,rt</td>
</tr>
<tr>
<td>Divide unsigned</td>
<td>divu rs,rt</td>
</tr>
<tr>
<td>Add immediate unsigned</td>
<td>addiu rs,rt,imm</td>
</tr>
<tr>
<td>Shift left logical</td>
<td>sll rd,rt,sh</td>
</tr>
<tr>
<td>Shift right logical</td>
<td>srl rd,rt,sh</td>
</tr>
<tr>
<td>Shift right arithmetic</td>
<td>sra rd,rt,sh</td>
</tr>
<tr>
<td>Shift left logical variable</td>
<td>sllv rd,rt,rs</td>
</tr>
<tr>
<td>Shift right logical variable</td>
<td>srlv rd,rt,rs</td>
</tr>
<tr>
<td>Shift right arith variable</td>
<td>srav rd,rt,rs</td>
</tr>
<tr>
<td>Load byte</td>
<td>lb rt,imm(rs)</td>
</tr>
<tr>
<td>Load byte unsigned</td>
<td>lbu rt,imm(rs)</td>
</tr>
<tr>
<td>Store byte</td>
<td>sb rt,imm(rs)</td>
</tr>
<tr>
<td>Jump and link</td>
<td>jal L</td>
</tr>
<tr>
<td>System call</td>
<td>syscall</td>
</tr>
</tbody>
</table>
7 Assembly Language Programs

Everything else needed to build and run assembly programs:
- Supply info to assembler about program and its data
- Non-hardware-supported instructions for convenience

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Machine and Assembly Languages</td>
</tr>
<tr>
<td>7.2 Assembler Directives</td>
</tr>
<tr>
<td>7.3 Pseudoinstructions</td>
</tr>
<tr>
<td>7.4 Macroinstructions</td>
</tr>
<tr>
<td>7.5 Linking and Loading</td>
</tr>
<tr>
<td>7.6 Running Assembler Programs</td>
</tr>
</tbody>
</table>
7.1 Machine and Assembly Languages

**Figure 7.1** Steps in transforming an assembly language program to an executable program residing in memory.

Assembly language program

- `add $2, $5, $5`
- `add $2, $2, $2`
- `add $2, $4, $2`
- `lw  $15, 0($2)`
- `lw  $16, 4($2)`
- `sw  $16, 0($2)`
- `sw  $15, 4($2)`
- `jr  $31`

Machine language program

```
00a51020 00421020
00821020 8c620000
8c620004 8cf20004
acf20000 ac620004
03e00008
```

Executable machine language program

Library routines (machine language)

MIPS, 80x86, PowerPC, etc.

Assembler

Linker

Loader

Memory content
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Location</th>
<th>Machine language program</th>
</tr>
</thead>
<tbody>
<tr>
<td>done</td>
<td>28</td>
<td>0:001000000010000000000000000000010001</td>
</tr>
<tr>
<td>result</td>
<td>248</td>
<td>4:00000010000100000000000000000100010</td>
</tr>
<tr>
<td>test</td>
<td>12</td>
<td>8:00000010001000000000000000000100000</td>
</tr>
</tbody>
</table>

Determined from assembler directives not shown here

**Figure 7.2** An assembly-language program, its machine-language version, and the symbol table created during the assembly process.
7.2 Assembler Directives

Assembler directives provide the assembler with info on how to translate the program but do not lead to the generation of machine instructions.

```assembly
.macro # start macro (see Section 7.4)
.end_macro  # end macro (see Section 7.4)
.text # start program’s text segment
...  # program text goes here
.data # start program’s data segment
	tiny: .byte 156,0x7a # name & initialize data byte(s)
	max: .word 35000   # name & initialize data word(s)
	small: .float 2E-3  # name short float (see Chapter 12)
	big: .double 2E-3   # name long float (see Chapter 12)
	.align 2           # align next item on word boundary
.array: .space 600  # reserve 600 bytes = 150 words
.str1: .ascii “a*b” # name & initialize ASCII string
.str2: .asciiz “xyz” # null-terminated ASCII string
.global main        # consider “main” a global name
```
Composing Simple Assembler Directives

Example 7.1

Write assembler directive to achieve each of the following objectives:

a. Put the error message “Warning: The printer is out of paper!” in memory.
b. Set up a constant called “size” with the value 4.
c. Set up an integer variable called “width” and initialize it to 4.
d. Set up a constant called “mill” with the value 1,000,000 (one million).
e. Reserve space for an integer vector “vect” of length 250.

Solution:

a. 

b. 

c. 

d. 

e. 

# 250 words = 1000 bytes
7.3 Pseudoinstructions

Example of one-to-one pseudoinstruction: The following

\[
\text{not } \$s0 \quad \# \text{ complement } (\$s0)
\]

is converted to the real instruction:

\[
\text{nor } \$s0,\$s0,\$zero \quad \# \text{ complement } (\$s0)
\]

Example of one-to-several pseudoinstruction: The following

\[
\text{abs } \$t0,\$s0 \quad \# \text{ put } |(\$s0)| \text{ into } \$t0
\]

is converted to the sequence of real instructions:

\[
\begin{align*}
\text{add } \& \$t0,\$s0,\$zero & \quad \# \text{ copy } x \text{ into } \$t0 \\
\text{slt } \& \$at,\$t0,\$zero & \quad \# \text{ is } x \text{ negative?} \\
\text{beq } \& \$at,\$zero,+4 & \quad \# \text{ if not, skip next instr} \\
\text{sub } \& \$t0,\$zero,\$s0 & \quad \# \text{ the result is } 0 - x
\end{align*}
\]
### MiniMIPS Pseudo-instructions

<table>
<thead>
<tr>
<th>Pseudoinstruction</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Move</td>
<td>move regd,regs</td>
</tr>
<tr>
<td>Load address</td>
<td>la regd,address</td>
</tr>
<tr>
<td>Load immediate</td>
<td>li regd,anyimm</td>
</tr>
<tr>
<td>Absolute value</td>
<td>abs regd,regs</td>
</tr>
<tr>
<td>Negate</td>
<td>neg regd,regs</td>
</tr>
<tr>
<td>Multiply (into register)</td>
<td>mul regd,reg1,reg2</td>
</tr>
<tr>
<td>Divide (into register)</td>
<td>div regd,reg1,reg2</td>
</tr>
<tr>
<td>Remainder</td>
<td>rem regd,reg1,reg2</td>
</tr>
<tr>
<td>Set greater than</td>
<td>sgt regd,reg1,reg2</td>
</tr>
<tr>
<td>Set less or equal</td>
<td>sle regd,reg1,reg2</td>
</tr>
<tr>
<td>Set greater or equal</td>
<td>sge regd,reg1,reg2</td>
</tr>
<tr>
<td>Rotate left</td>
<td>rol regd,reg1,reg2</td>
</tr>
<tr>
<td>Rotate right</td>
<td>ror regd,reg1,reg2</td>
</tr>
<tr>
<td>NOT</td>
<td>not reg</td>
</tr>
<tr>
<td>Load doubleword</td>
<td>ld regd,address</td>
</tr>
<tr>
<td>Store doubleword</td>
<td>sd regd,address</td>
</tr>
<tr>
<td>Branch less than</td>
<td>blt reg1,reg2,L</td>
</tr>
<tr>
<td>Branch greater than</td>
<td>bgt reg1,reg2,L</td>
</tr>
<tr>
<td>Branch less or equal</td>
<td>ble reg1,reg2,L</td>
</tr>
<tr>
<td>Branch greater or equal</td>
<td>bge reg1,reg2,L</td>
</tr>
</tbody>
</table>

#### Table 7.1

- **Copy**
- **Arithmetic**
- **Shift**
- **Logic**
- **Memory access**
- **Control transfer**
7.4 Macroinstructions

A macro is a mechanism to give a name to an oft-used sequence of instructions (shorthand notation)

```assembly
.macro  name(args) # macro and arguments named
... # instr’s defining the macro
.end_macro # macro terminator
```

How is a macro different from a pseudoinstruction?

Pseudos are predefined, fixed, and look like machine instructions
Macros are user-defined and resemble procedures (have arguments)

How is a macro different from a procedure?

Control is transferred to and returns from a procedure
After a macro has been replaced, no trace of it remains
Macro to Find the Largest of Three Values

Write a macro to determine the largest of three values in registers and to put the result in a fourth register.

Solution:

```assembly
.macro mx3r(m,a1,a2,a3)  # macro and arguments named
    move   m,a1              # assume (a1) is largest; m = (a1)
    bge m,a2,+4           # if (a2) is not larger, ignore it
    move   m,a2              # else set m = (a2)
    bge m,a3,+4           # if (a3) is not larger, ignore it
    move   m,a3              # else set m = (a3)
.endmacro # macro terminator
```

If the macro is used as `mx3r($t0,$s0,$s4,$s3)`, the assembler replaces the arguments `m, a1, a2, a3` with `$t0, $s0, $s4, $s3`, respectively.
7.5 Linking and Loading

The linker has the following responsibilities:

- Ensuring correct interpretation (resolution) of labels in all modules
- Determining the placement of text and data segments in memory
- Evaluating all data addresses and instruction labels
- Forming an executable program with no unresolved references

The loader is in charge of the following:

- Determining the memory needs of the program from its header
- Copying text and data from the executable program file into memory
- Modifying (shifting) addresses, where needed, during copying
- Placing program parameters onto the stack (as in a procedure call)
- Initializing all machine registers, including the stack pointer
- Jumping to a start-up routine that calls the program’s main routine
7.6 Running Assembler Programs

Spim is a simulator that can run MiniMIPS programs.
The name Spim comes from reversing MIPS.
Three versions of Spim are available for free downloading:

- PCSpim for Windows machines
- xspim for X-windows
- spim for Unix systems

You can download SPIM by visiting:

http://www.cs.wisc.edu/~larus/spim.html
## Input/Output Conventions for MiniMIPS

### Table 7.2: Input/output and control functions of `syscall` in PCSpim.

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>($v0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Print integer</td>
<td>Integer in $a0</td>
<td>Integer displayed</td>
</tr>
<tr>
<td>2 Print floating-point</td>
<td>Float in $f12</td>
<td>Float displayed</td>
</tr>
<tr>
<td>3 Print double-float</td>
<td>Double-float in $f12,$f13</td>
<td>Double-float displayed</td>
</tr>
<tr>
<td>4 Print string</td>
<td>Pointer in $a0</td>
<td>Null-terminated string displayed</td>
</tr>
<tr>
<td>5 Read integer</td>
<td>Pointer in $a0</td>
<td>Integer returned in $v0</td>
</tr>
<tr>
<td>6 Read floating-point</td>
<td>Float in $f0</td>
<td>Float returned in $f0</td>
</tr>
<tr>
<td>7 Read double-float</td>
<td>Pointer in $a0, length in $a1</td>
<td>Double-float returned in $f0,$f1</td>
</tr>
<tr>
<td>8 Read string</td>
<td>Pointer in $a0, length in $a1</td>
<td>String returned in buffer at pointer</td>
</tr>
<tr>
<td>9 Allocate memory</td>
<td>Number of bytes in $a0</td>
<td>Pointer to memory block in $v0</td>
</tr>
<tr>
<td>10 Exit from program</td>
<td></td>
<td>Program execution terminated</td>
</tr>
</tbody>
</table>
Figure 7.3

PCSpim User Interface

Menu bar
Tools bar

File
Open
Save Log File
Exit

Simulator
Clear Registers
Reinitialize
Reload
Go
Break
Continue
Single Step
Multiple Step ...
Breakpoints ...
Set Value ...
Disp Symbol Table
Settings ...

PCSpim

File Simulator Window Help

Registers
PC = 00400000  EPC = 00000000  Cause = 00000000
Status = 00000000  HI = 00000000  LO = 00000000

General Registers
R0 (r0) = 0  R8 (t0) = 0  R16 (s0) = 0  R24
R1 (at) = 0  R9 (t1) = 0  R17 (s1) = 0  R25

Text Segment
[0x00400000] 0xc0100008 jal 0x00400020 [main] ; 43
[0x00400004] 0x00000021 addu $0, $0, $0 ; 44
[0x00400008] 0x2402000a addiu $2, $0, 10 ; 45
[0x0040000c] 0x0000000c syscall ; 46
[0x00400010] 0x00000021 addu $0, $0, $0 ; 47

Data Segment
DATA
[0x10000000] 0x00000000 0x6c696146 0x20206465
[0x10000010] 0x676e6974 0x44444120 0x6554000a
[0x10000020] 0x44412067 0x000a4944 0x74736554

Messages
See the file README for a full copyright notice.
Memory and registers have been cleared, and the simulator re
D:\temp\dos\TESTS\Alubare.s has been successfully loaded

For Help, press F1
Base=1; Pseudo=1, Mapped=1; LoadTrap=0
8 Instruction Set Variations

The MiniMIPS instruction set is only one example
• How instruction sets may differ from that of MiniMIPS
• RISC and CISC instruction set design philosophies

<table>
<thead>
<tr>
<th>Topics in This Chapter</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.1    Complex Instructions</td>
</tr>
<tr>
<td>8.2    Alternative Addressing Modes</td>
</tr>
<tr>
<td>8.3    Variations in Instruction Formats</td>
</tr>
<tr>
<td>8.4    Instruction Set Design and Evolution</td>
</tr>
<tr>
<td>8.5    The RISC/CISC Dichotomy</td>
</tr>
<tr>
<td>8.6    Where to Draw the Line</td>
</tr>
</tbody>
</table>
### 8.1 Complex Instructions

Table 8.1 (partial)  Examples of complex instructions in two popular modern microprocessors and two computer families of historical significance

<table>
<thead>
<tr>
<th>Machine</th>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium</td>
<td>MOVS</td>
<td>Move one element in a string of bytes, words, or doublewords using addresses specified in two pointer registers; after the operation, increment or decrement the registers to point to the next element of the string</td>
</tr>
<tr>
<td>PowerPC</td>
<td>cntlzd</td>
<td>Count the number of consecutive 0s in a specified source register beginning with bit position 0 and place the count in a destination register</td>
</tr>
<tr>
<td>IBM 360-370</td>
<td>CS</td>
<td>Compare and swap: Compare the content of a register to that of a memory location; if unequal, load the memory word into the register, else store the content of a different register into the same memory location</td>
</tr>
<tr>
<td>Digital VAX</td>
<td>POLYD</td>
<td>Polynomial evaluation with double flp arithmetic: Evaluate a polynomial in ( x ), with very high precision in intermediate results, using a coefficient table whose location in memory is given within the instruction</td>
</tr>
</tbody>
</table>
## Benefits and Drawbacks of Complex Instructions

<table>
<thead>
<tr>
<th>Benefits</th>
<th>Drawbacks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fewer instructions in program (less memory)</td>
<td>More complex format (slower decoding)</td>
</tr>
<tr>
<td>Fewer memory accesses for instructions</td>
<td>Less flexible (one algorithm for polynomial evaluation or sorting may not be the best in all cases)</td>
</tr>
<tr>
<td>Programs may become easier to write/read/understand</td>
<td>If interrupts are processed at the end of instruction cycle, machine may become less responsive to time-critical events (interrupt handling)</td>
</tr>
<tr>
<td>Potentially faster execution (complex steps are still done sequentially in multiple cycles, but hardware control can be faster than software loops)</td>
<td></td>
</tr>
</tbody>
</table>
8.2 Alternative Addressing Modes

Let’s refresh our memory (from Chap. 5)

<table>
<thead>
<tr>
<th>Addressing</th>
<th>Instruction</th>
<th>Other elements involved</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>Implied</td>
<td></td>
<td>Some place in the machine</td>
<td></td>
</tr>
<tr>
<td>Immediate</td>
<td></td>
<td>Extend, if required</td>
<td></td>
</tr>
<tr>
<td>Register</td>
<td>Reg spec</td>
<td>Reg file</td>
<td>Reg data</td>
</tr>
<tr>
<td>Base</td>
<td>Constant offset</td>
<td>Reg file</td>
<td>Mem addr</td>
</tr>
<tr>
<td>PC-relative</td>
<td>Constant offset</td>
<td>Add</td>
<td>Memory</td>
</tr>
<tr>
<td>Pseudodirect</td>
<td></td>
<td>PC</td>
<td>Mem addr</td>
</tr>
</tbody>
</table>

Figure 5.11  Schematic representation of addressing modes in MiniMIPS.
More Elaborate Addressing Modes

More Elaborate Addressing Modes are not supported in MiniMIPS.

Figure 8.1 Schematic representation of more elaborate addressing modes not supported in MiniMIPS.
Usefulness of Some Elaborate Addressing Modes

Update mode: XORing a string of bytes

```assembly
loop:  lb    $t0, A($s0)
xor  $s1, $s1, $t0
addi $s0, $s0, -1
bne  $s0, $zero, loop
```

One instruction with update addressing

Indirect mode: Case statement

```assembly
case:  lw   $t0, 0($s0)  # get s
      add  $t0, $t0, $t0  # form 2s
      add  $t0, $t0, $t0  # form 4s
      la   $t1, T        # base T
      add  $t1, $t0, $t1
      lw   $t2, 0($t1)   # entry
      jr    $t2
```

Branch to location $Li$
if $s = i$ (switch var.)

<table>
<thead>
<tr>
<th>$T$</th>
<th>L0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T+4$</td>
<td>L1</td>
</tr>
<tr>
<td>$T+8$</td>
<td>L2</td>
</tr>
<tr>
<td>$T+12$</td>
<td>L3</td>
</tr>
<tr>
<td>$T+16$</td>
<td>L4</td>
</tr>
<tr>
<td>$T+20$</td>
<td>L5</td>
</tr>
</tbody>
</table>
### 8.3 Variations in Instruction Formats

0-, 1-, 2-, and 3-address instructions in MiniMIPS

<table>
<thead>
<tr>
<th>Category</th>
<th>Format</th>
<th>Opcode</th>
<th>Description of operand(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-address</td>
<td>0</td>
<td>syscall</td>
<td>One implied operand in register $v0</td>
</tr>
<tr>
<td>1-address</td>
<td>2 Address</td>
<td>j</td>
<td>Jump target addressed (in pseudodirect form)</td>
</tr>
<tr>
<td>2-address</td>
<td>0 rs rt</td>
<td>mult</td>
<td>Two source registers addressed, destination implied</td>
</tr>
<tr>
<td>3-address</td>
<td>0 rs rt rd</td>
<td>add</td>
<td>Destination and two source registers addressed</td>
</tr>
</tbody>
</table>

**Figure 8.2** Examples of MiniMIPS instructions with 0 to 3 addresses; shaded fields are unused.
Zero-Address Architecture: Stack Machine

Stack holds all the operands (replaces our register file)
Load/Store operations become push/pop

Arithmetic/logic operations need only an opcode: they pop operand(s) from the top of the stack and push the result onto the stack

Example: Evaluating the expression \((a + b) \times (c - d)\)

<table>
<thead>
<tr>
<th>Push a</th>
<th>Push b</th>
<th>Add</th>
<th>Push d</th>
<th>Push c</th>
<th>Subtract</th>
<th>Multiply</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>b</td>
<td>a + b</td>
<td>d</td>
<td>c</td>
<td>c – d</td>
<td>a + b</td>
</tr>
<tr>
<td>a</td>
<td>a + b</td>
<td>d</td>
<td>a + b</td>
<td>Result</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Polish string: \(a \ b + d \ c - \times\)

If a variable is used again, you may have to push it multiple times

Special instructions such as “Duplicate” and “Swap” are helpful
One-Address Architecture: Accumulator Machine

The accumulator, a special register attached to the ALU, always holds operand 1 and the operation result.

Only one operand needs to be specified by the instruction.

**Example:** Evaluating the expression \((a + b) \times (c - d)\)

- Load \(a\)
- Add \(b\)
- Store \(t\)
- Load \(c\)
- Subtract \(d\)
- Multiply \(t\)

Within branch instructions, the condition or target address must be implied:
- Branch to L if acc negative
- If register \(x\) is negative skip the next instruction

May have to store accumulator contents in memory (example above)

No store needed for \(a + b + c + d + \ldots\) ("accumulator")
Two-Address Architectures

Two addresses may be used in different ways:
Operand1/result and operand 2
Condition to be checked and branch target address

**Example:** Evaluating the expression \((a + b) \times (c - d)\)

```
load $1,a
add  $1,b
load $2,c
subtract $2,d
multiply $1,$2
```

Instructions of a hypothetical two-address machine

A variation is to use one of the addresses as in a one-address machine and the second one to specify a branch in every instruction.
Components that form a variable-length IA-32 (80x86) instruction.

Example of a Complex Instruction Format

Instruction prefixes (zero to four, 1 B each)

Operand/address size overwrites and other modifiers

Opcode (1-2 B)

ModR/M

Most memory operands need these 2 bytes

Offset or displacement (0, 1, 2, or 4 B)

Immediate (0, 1, 2, or 4 B)

Instructions can contain up to 15 bytes
Some of IA-32’s Variable-Width Instructions

<table>
<thead>
<tr>
<th>Type</th>
<th>Format (field widths shown)</th>
<th>Opcode</th>
<th>Description of operand(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-byte</td>
<td>5 3</td>
<td>PUSH</td>
<td>3-bit register specification</td>
</tr>
<tr>
<td>2-byte</td>
<td>4 4 8</td>
<td>JE</td>
<td>4-bit condition, 8-bit jump offset</td>
</tr>
<tr>
<td>3-byte</td>
<td>6 8 8</td>
<td>MOV</td>
<td>8-bit register/mode, 8-bit offset</td>
</tr>
<tr>
<td>4-byte</td>
<td>8 8 8 8</td>
<td>XOR</td>
<td>8-bit register/mode, 8-bit base/index, 8-bit offset</td>
</tr>
<tr>
<td>5-byte</td>
<td>4 3 32</td>
<td>ADD</td>
<td>3-bit register spec, 32-bit immediate</td>
</tr>
<tr>
<td>6-byte</td>
<td>7 8 32</td>
<td>TEST</td>
<td>8-bit register/mode, 32-bit immediate</td>
</tr>
</tbody>
</table>

Figure 8.3  Example 80x86 instructions ranging in width from 1 to 6 bytes; much wider instructions (up to 15 bytes) also exist
8.4 Instruction Set Design and Evolution

Desirable attributes of an instruction set:

Consistent, with uniform and generally applicable rules
Orthogonal, with independent features noninterfering
Transparent, with no visible side effect due to implementation details
Easy to learn/use (often a byproduct of the three attributes above)
Extensible, so as to allow the addition of future capabilities
Efficient, in terms of both memory needs and hardware realization

Figure 8.4 Processor design and implementation process.
8.5 The RISC/CISC Dichotomy

The RISC (reduced instruction set computer) philosophy:
Complex instruction sets are undesirable because inclusion of mechanisms to interpret all the possible combinations of opcodes and operands might slow down even very simple operations.

Ad hoc extension of instruction sets, while maintaining backward compatibility, leads to CISC; imagine modern English containing every English word that has been used through the ages

Features of RISC architecture

1. Small set of inst’s, each executable in roughly the same time
2. Load/store architecture (leading to more registers)
3. Limited addressing mode to simplify address calculations
4. Simple, uniform instruction formats (ease of decoding)
RISC/CISC Comparison via Generalized Amdahl’s Law

Example 8.1

An ISA has two classes of simple (S) and complex (C) instructions. On a reference implementation of the ISA, class-S instructions account for 95% of the running time for programs of interest. A RISC version of the machine is being considered that executes only class-S instructions directly in hardware, with class-C instructions treated as pseudoinstructions. It is estimated that in the RISC version, class-S instructions will run 20% faster while class-C instructions will be slowed down by a factor of 3. Does the RISC approach offer better or worse performance compared to the reference implementation?

Solution

Per assumptions, 0.95 of the work is speeded up by a factor of $1.0 / 0.8 = 1.25$, while the remaining 5% is slowed down by a factor of 3. The RISC speedup is $1 / [0.95 / 1.25 + 0.05 \times 3] = 1.1$. Thus, a 10% improvement in performance can be expected in the RISC version.
Some Hidden Benefits of RISC

In Example 8.1, we established that a speedup factor of 1.1 can be expected from the RISC version of a hypothetical machine.

This is not the entire story, however!

If the speedup of 1.1 came with some additional cost, then one might legitimately wonder whether it is worth the expense and/or time.

The RISC version of the architecture also:

- Reduces the effort and team size for design
- Shortens the testing and debugging phase
- Simplifies documentation and maintenance

\[ \text{Cheaper product and shorter time-to-market} \]
8.6 Where to Draw the Line

The ultimate reduced instruction set computer (URISC):
How many instructions are absolutely needed for useful computation?

Only one!
subtract source1 from source2, replace source2 with the result, and jump to target address if result is negative

Assembly language form:

label: urisc dest,srcl,target

Pseudoinstructions can be synthesized using the single instruction:

stop: .word 0
start: urisc dest,dest,+1  # dest = 0
       urisc temp,temp,+1   # temp = 0
       urisc temp,srcl,+1   # temp = -(srcl)
       urisc dest,temp,+1  # dest = -(temp); i.e. (srcl)
       ...  # rest of program

Corrected version

This is the move pseudoinstruction
Some Useful Pseudo Instructions for URISC

Example 8.2 (2 parts of 5)

Write the sequence of instructions that are produced by the URISC assembler for each of the following pseudoinstructions.

parta:  uadd    dest,src1,src2  # dest=(src1)+(src2)
        urisc   at1,at1,+1     # at1 = 0
        urisc   at1,src1,+1    # at1 = -(src1)
        urisc   at1,src2,+1    # at1 = -(src1)–(src2)
        urisc   dest,dest,+1   # dest = 0
        urisc   dest,at1,+1    # dest = -(at1)

partc:  uj         label       # goto label
        urisc   at1,at1,+1     # at1 = 0
        urisc   at1,one,label  # at1 = -1 to force jump

Solution

at1 and at2 are temporary memory locations for assembler’s use

parta:  urisc   at1,at1,+1     # at1 = 0
        urisc   at1,src1,+1    # at1 = -(src1)
        urisc   at1,src2,+1    # at1 = -(src1)–(src2)
        urisc   dest,dest,+1   # dest = 0
        urisc   dest,at1,+1    # dest = -(at1)

partc:  urisc   at1,at1,+1     # at1 = 0
        urisc   at1,one,label  # at1 = -1 to force jump
Figure 8.5  Instruction format and hardware structure for URISC.