Chapter 5
Data Organization: The Load and Store Instructions

Arthur B. Maccabe
Department of Computer Science
The University of New Mexico


Overview

- Assembler directives
  - the location counter; the symbol table, allocation, initialization and alignment; assembler segments; and constant expressions
- SPARC load and store instructions
  - operand sizes; memory addresses; and encoding load and store instructions
- Data organization
  - pointers; arrays; structures; and strings
- Addresses in control: switch
- Addresses and integers
- Addressing modes on the HP-PA

Slide 1
Assembly Language Conventions

**Space** any number of spaces or tabs; cannot appear in a number or identifier; optional, unless specifically required or prohibited; readability

**Comments** “!” and everything following on the line.

**Identifier** an underscore ("_") or a letter followed by any number of underscores, letters, and digits.

**Label definition** an identifier followed by a colon (":").

**Instruction** an operation name followed by a list of operands; there must be a space between the operation name and the operand list

**Operand list** a comma-separated list of operands; number of operands depends on the operation.

**Line** empty (i.e., white space); a label definition; an instruction; or a label definition followed by an instruction.

**Program** a sequence of lines.

Assembler Directives—Overview

- Sometimes called pseudo-operations
- Like an instruction
  - specified on a single line; optional label; name followed by parameters

**Slide 3**

- Convention: directive names start with ‘.’
- Translation, the location counter, and the symbol table
- Allocation, initialization, and alignment
- Assembler segments
- Constant expressions
The Translation Process

Slide 4

The Location Counter

- Starts at zero; is incremented (by instruction size) as each instruction is translated
- Logical address of the instruction
- Contrast to program counter
  - translation time versus execution time
  - location counter is strictly increasing, program counter can decrease (loops)

Slide 5
The Symbol Table

Slide 6
- Operations: insert and lookup
- Labels
  \(<\textit{label}, \textit{address}>;\) uses the location counter; forward references are OK
- Symbolic constants
  e.g., \texttt{.equiv const, 15}; \(<\textit{name}, \textit{value}>;\) uses explicitly supplied value; no forward references; avoid the use of magic numbers

Allocation and Initialization

- Sizes:
  \begin{tabular}{lrr}
  \hline
  Type & Bytes & Bits \\
  \hline
  \texttt{char} & 1 & 8 \\
  \texttt{short int} & 2 & 16 \\
  \texttt{int} & 4 & 32 \\
  \texttt{long int} & 8 & 64 \\
  \hline
  \end{tabular}

Slide 7
- Allocation: \texttt{.skip}; e.g., \texttt{.skip} 2
- Initialization:
  \begin{tabular}{l}
  \hline
  \texttt{name} & \texttt{size} \\
  \hline
  \texttt{.byte} & \texttt{1 byte} \\
  \texttt{.hword} & \texttt{2 bytes} \\
  \texttt{.word} & \texttt{4 bytes} \\
  \texttt{xword} & \texttt{8 bytes} \\
  \hline
  \end{tabular}
  e.g., \texttt{byte ‘a’, ‘b’, ‘c’}
Example

short int short_one = 22;
char ch_one = 'a';
short int short_two = 33;
char ch_two = 'A';
int int_one = 0;

Slide 8

<table>
<thead>
<tr>
<th>short_one:</th>
<th>.hword 22</th>
</tr>
</thead>
<tbody>
<tr>
<td>ch_one:</td>
<td>.byte 'a'</td>
</tr>
<tr>
<td>short_two:</td>
<td>.hword 33</td>
</tr>
<tr>
<td>ch_two:</td>
<td>.byte 'A'</td>
</tr>
<tr>
<td>int_one:</td>
<td>.word 0</td>
</tr>
</tbody>
</table>

Slide 9

<table>
<thead>
<tr>
<th>.align 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>short_one: .hword 22</td>
</tr>
<tr>
<td>ch_one:    .byte 'a'</td>
</tr>
<tr>
<td>.align 2</td>
</tr>
<tr>
<td>short_two: .hword 33</td>
</tr>
<tr>
<td>ch_two:    .byte 'A'</td>
</tr>
<tr>
<td>.align 4</td>
</tr>
<tr>
<td>int_one:   .word 0</td>
</tr>
</tbody>
</table>

assume a is divisible by 2, but not divisible by 4
Assembler Segments

Source code → Translator → Object code

Symbol table

Text segment

Data segment

Explicit segment identification: e.g., `seg "data"

The bss Segment

Object code file

Header

Code

Data segment

Memory image

Text

Data

bss
Constant Expressions

- Let the assembler do the math!
- e.g.:

  Slide 12
  .equiv CarSize, 14

  .skip 20 * CarSize

- Non relocatable expression

SPARC Load and Store Instructions—Overview

- Store instructions
- Big endian
- Load instructions

  Slide 13
  • Addressing modes
    - register indirect
    - register indirect with displacement
    - register indirect with index

- Encoding
### SPARC Store Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>stb</td>
<td>Store byte</td>
<td>stub, stsb</td>
</tr>
<tr>
<td>sth</td>
<td>Store halfword</td>
<td>stuh, stsh</td>
</tr>
<tr>
<td>stw</td>
<td>Store word</td>
<td>st, stuw, stsw</td>
</tr>
<tr>
<td>stx</td>
<td>Store extended (double) word</td>
<td></td>
</tr>
</tbody>
</table>

---

#### Sizes

- **Byte:**

  ![Byte Diagram]

- **Halfword:**

  ![Halfword Diagram]
Sizes (continued)

- Word:

Slide 16

```
a
a+1
a+2
a+3
```

- Extended word:

Slide 17

```
a
a+1
a+2
a+3
a+4
a+5
a+6
a+7
```
SPARC Load Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Operation</th>
<th>Synonyms</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldsb</td>
<td>Load signed byte</td>
<td></td>
</tr>
<tr>
<td>ldsh</td>
<td>Load signed halfword</td>
<td></td>
</tr>
<tr>
<td>ldsw</td>
<td>Load signed word</td>
<td></td>
</tr>
<tr>
<td>ldub</td>
<td>Load unsigned byte</td>
<td></td>
</tr>
<tr>
<td>lduh</td>
<td>Load unsigned halfword</td>
<td></td>
</tr>
<tr>
<td>lduw</td>
<td>Load unsigned word</td>
<td>ld</td>
</tr>
<tr>
<td>ldx</td>
<td>Load extended (double) word</td>
<td></td>
</tr>
</tbody>
</table>

Memory Addresses

- Address description
- Effective address calculation
- Three modes
  - register indirect
  - register indirect with displacement
  - register indirect with index
Register Indirect

- Syntax: register in square brackets (e.g., [%r2])
- Graphical interpretation:

```
    Memory
    ┌────────────────────────────────┐
    │                                 │
    │                                 │
    │                                 │
    │                                 │
    └────────────────────────────────┘
          Register
```
- Code example:
  
  ```
  kluw [%r4], [%r5]
  inc  [%r5]
  stw  [%r5], [%r4]
  ```

Using Symbolic Addresses

```
.seg  "data"    ! switch to the data segment
x: .skip 2      ! allocate 2 bytes

Slide 21
.seg  "text"    ! switch to the text segment
setx x, [%r3], [%r4]  ! put the address of x into %r4
ldsh [%r4], [%r5]
inc  [%r5]
sth  [%r5], [%r4]
```
Register Indirect with Displacement

- Syntax: register ‘+’ small constant (e.g., [%r4 + 24])
- Graphical interpretation:

```
Displacement

<table>
<thead>
<tr>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Base register
```

Expanding the “setx”

```
sethi %uhi(x), %r3        ! start of setx
or  %r3, %ulo(x), %r3
slx  %r3, 32, %r3
sethi %hi(x), %r4
or  %r4, %r3, %r4
or  %r4, %lo(x), %r4    ! end of setx
ldsh [%r4], %r5
inc  %r5
sth  %r5, [%r4]
```
Using Displacement Addressing

seti \%hi(x), \%r3
or \%r3, \%ulo(x), \%r3
slx \%r3, 32, \%r3
seti \%hi(x), \%r4
or \%r4, \%r3, \%r4
ldsh [\%r4+\%lo(x)], \%r5
inc \%r5
sth \%r5, [\%r4+\%lo(x)]

Register Indirect with Index

- Syntax: base register ‘+’ index register (e.g., [\%r4 + \%r5])
- Graphical interpretation:

![Diagram showing Memory, Base register, Index register connections]
Assuming 32-bit Addresses

Slide 26

seth \%hi(x), \%r4
ldsh [\%r4+\%lo(x)], \%r5
inc \%r5
sth \%r5, [\%r4+\%lo(x)]

Encoding Load and Store Instructions

Slide 27

\[
\begin{array}{ccccccc}
11 & \text{rd} & \text{op}_3 & \text{rs}_1 & 0 & 00000000 & \text{rs}_2 \\
11 & \text{rd} & \text{op}_3 & \text{rs}_1 & 1 & \text{sim}13 \\
\end{array}
\]

Field | Meaning
--- | ---
rd | Destination register for load instructions, source register for store instructions
op\_3 | Op code
rs\_1 | Base register
rs\_2 | Index register
sim\_13 | Displacement value in 13-bit 2's complement representation
Opcodes

<table>
<thead>
<tr>
<th>Operation name</th>
<th>op3</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldsh</td>
<td>00 1010</td>
<td>Load signed halfword</td>
</tr>
<tr>
<td>ldsw</td>
<td>00 1000</td>
<td>Load signed word</td>
</tr>
<tr>
<td>ldub</td>
<td>00 0001</td>
<td>Load unsigned byte</td>
</tr>
<tr>
<td>lduh</td>
<td>00 0010</td>
<td>Load unsigned halfword</td>
</tr>
<tr>
<td>lduw</td>
<td>00 0000</td>
<td>Load unsigned word</td>
</tr>
<tr>
<td>ldx</td>
<td>00 1011</td>
<td>Load extended word</td>
</tr>
<tr>
<td>stb</td>
<td>00 0101</td>
<td>Store byte</td>
</tr>
<tr>
<td>sth</td>
<td>00 0110</td>
<td>Store halfword</td>
</tr>
<tr>
<td>stw</td>
<td>00 0100</td>
<td>Store word</td>
</tr>
<tr>
<td>stx</td>
<td>00 0111</td>
<td>Store extended word</td>
</tr>
</tbody>
</table>

Example Encoding 1

- SPARC instruction: ldsh [%r1+2], %r5

- Format

```
 31 30 29 25 24 23 22 21 20 19 18 14 13 12 11 0
```

- Field values

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Encoded value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>%r5</td>
<td>00101</td>
</tr>
<tr>
<td>op3</td>
<td>ldsh</td>
<td>001010</td>
</tr>
<tr>
<td>rs1</td>
<td>%r1</td>
<td>00001</td>
</tr>
<tr>
<td>simm13</td>
<td>2</td>
<td>0000000000010</td>
</tr>
</tbody>
</table>

- Encoding: 11 00101 001010 00001 1 0000000000010, or 0xA506002
Example Encoding 2

- SPARC instruction: ldsh [%r1+%r3], %r5

- Format

<table>
<thead>
<tr>
<th></th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>0</th>
<th>00000000</th>
<th>rs2</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Field values

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Encoded value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>%dr5</td>
<td>00101</td>
</tr>
<tr>
<td>op3</td>
<td>ldsh</td>
<td>001010</td>
</tr>
<tr>
<td>rs1</td>
<td>%r1</td>
<td>00001</td>
</tr>
<tr>
<td>rs2</td>
<td>%r3</td>
<td>00011</td>
</tr>
</tbody>
</table>

- Encoding: 11 00101 001010 00001 0 00000000 00011, or 0xCA504003

Example Encoding 3

- SPARC instruction: stw %r7, [%r1-12]

- Format

<table>
<thead>
<tr>
<th></th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>1</th>
<th>simm13</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Field values

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
<th>Encoded value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>%r7</td>
<td>00111</td>
</tr>
<tr>
<td>op3</td>
<td>stw</td>
<td>000100</td>
</tr>
<tr>
<td>rs1</td>
<td>%r1</td>
<td>00001</td>
</tr>
<tr>
<td>simm13</td>
<td>-12</td>
<td>11111111110100</td>
</tr>
</tbody>
</table>

- Encoding: 11 00111 000100 00001 1 1111111110100, or 0xCE207FF4
Data Organization—Overview

- How is data organized in memory?
- Pointers
- Arrays
- Structures
- Strings

Pointers

- “pointer to” and “address of” (usually “pointer to” is associated with a type)
- Example
  
  ```
  char ch1, *ptr;
  ```

Slide 33

  ```
  ptr = &ch;
  *ptr = 'A';
  ```
- Graphical interpretation

  ![Graphical interpretation](attachment://pointer_diagram.png)
Pointers in Assembly Language

- C code
  
  ```c
  char ch1, *ptr;
  
  ptr = &ch1;
  ```

- SPARC code
  
  ```asm
  .seg  "data"
  .align 8
  ptr: .skip 8       ! an address is 64 bits
  ch1: .skip 1       ! a character is one byte
  
  .seg  "text"
  set  ch1, %r2      ! %r2 = &ch1
  sethi %hi(ptr), %r3
  stx %r2, [%r3+lo(ptr)] ! store the value in ptr
  ```

Graphical Interpretation

Slide 35

```
Memory

ptr

ch1
```
Pointer Dereferencing

- C code

  ```
  *ptr = 'a';
  ```

- SPARC code

  ```
  seti  %hi(ptr), %r2
  ldx  [%r2+%lo(ptr)], %r3  ! load the pointer value
  set  'a', %r4
  stb  %r4, [%r3]  ! use register indirect addressing
  ```

Array Layout and Indexing

- Basics

Slide 37

- Iterating through an array
- Nonzero-based arrays
- Multidimensional arrays
Arrays: Layout

- Allocation
  - C code:
    ```c
    int int_arr[24];
    ```
  - SPARC directive:
    ```asm
    .reserve int_arr, 24 * 4, "bss" ! 4 bytes per integer
    ```
- Graphical interpretation

Arrays: Indexing

- Scaled addition
  ```plaintext
  addr = base + elem_size * index
  ```
- Graphical interpretation
Indexing Example 1

- C code
  
  \[ \text{int}_\text{arr}[5] = 47; \]

- SPARC code
  
  set int_arr+5*4, %r2 \quad \text{the indexed address calculation}
  set 47, %r3 \quad \text{put the value 47 in a register}
  stw %r3, [%r2] \quad \text{the store instruction}

Indexing Example 2

- C code
  
  \[ \text{int}_\text{arr}[i] = 47; \]

- SPARC code
  
  Slide 41
  
  set int_arr, %r2 \quad \text{put base address in a register}
  sethi %hi(i), %r3
  ldsw [%r3+%lo(i)], %r3 \quad \text{load the value of i}
  sll %r3, 2, %r3 \quad \text{scale the index}
  set 47, %r4 \quad \text{put the value 47 in a register}
  stw %r4, [%r2+%r3] \quad \text{store the value}
Sum Array Elements

```c
int arr[20];
int sum, i;

Slide 42

sum = 0;
for( i = 0 ; i < 20 ; i++ ) {
    sum = sum + arr[i];
}
```

Using Simple Control Transfers

```c
int arr[20];
int sum, i;

Slide 43

sum = 0;
i = 0;
top:
    sum = sum + arr[i];
i = i + 1;
if( i < 20 ) goto top
```
SPARC code

```
.reserve arr, 20*4, "bss"
.seg "data"
sum: .skip 4
```

Slide 44
```
.seg "text"
clr %r2                      ! sum = 0;
clr %r3                      ! i = 0;
set arr, %r4                 ! %r4 holds the base address
```

SPARC code (continued)

```
top:
    ksw [r4+r3], r5              ! arr[i]
    add r5, r2, r2               ! sum = sum + arr[i];
    inc 4, r3                    ! i = i + 1;
    cmp r3, 80                   ! i < 20
    bl top
    nop

    seti hi(sum), r5
    stw r2, [r5+lo(sum)]
```

23
Nonzero Arrays

- Basic indexing expression
  \[ \text{addr} = \text{base} + \text{elem.size} \times (\text{index} - \text{lower}) \]

Slide 46

- Gathering constants
  \[ \text{addr} = (\text{base} - \text{elem.size} \times \text{lower}) + \text{elem.size} \times \text{index} \]

- Savings:
  - when from: subtract, multiply, add
  - to: multiply, add

Example: Allocation and Definitions

- Context
  - an array of integers
  - upper bound 34
  - lower bound 15

Slide 47

- SPARC code
  .seg "data"
  .align 4
  i: .skip 4
  .reserve real_base, (34-15+1)*4, "bss" ! 4 bytes per integer
  .equiv img_base, real_base - 15*4

24
Example: Indexing

- set element with index \( i \) to 432
- SPARC code

```
seg "text"
set img_base, %r2 ! put base address in a register
sethi %hi(i), %r3
ldsw [%r3+%lo(i)], %r3 ! load the value of \( i \)
sll %r3, 2, %r3 ! scale the index
set 432, %r4
stw %r4, [%r2+%r3] ! store the value
```

Graphical Interpretation
Multidimensional Arrays

- Array of arrays
- C declaration: int two_arr[5][3];
- Row major layout (FORTRAN uses column major layout)

Graphical Interpretation
Example: Declarations

- C code
  
  ```c
  int i, j;
  int two_arr[5][3];
  ```

- SPARC code
  
  ```sparc
  .seg "data"
  .align 4
  i: .skip 4
  j: .skip 4
  .reserve two_arr, (5*3)*4, "bss" ! 5 arrays of 3 elements
  ! of 4 bytes
  ```

Example: Access

- C code
  
  ```c
  two_arr[i][j] = 47;
  ```

- SPARC code
  
  ```sparc
  .seg "text"
  set two_arr, %r2 ! %r2 holds the array base address
  sethi %hi(i), %r3
  ldsw [%r3+%lo(i)], %r3 ! load i
  mulp %r3, 3*4, %r3 ! scale by the size of a row
  add %r2, %r3, %r2 ! %r2 holds the row base address
  sethi %hi(j), %r3
  ldsw [%r3+%lo(j)], %r3 ! load j
  sll %r3, 2, %r1 ! scale by the element size
  set 47, %r4 ! the base address
  stw %r4, [%r2+%r3]
  ```
Structures

- C declarations
  ```
  struct person 
  { /* declare a structure */
    char name[14];
    int salary, age;
  }
  ```

- Template

- Padding introduced to support tiling (following one after the other with no alignment problems)

SPARC Template

! defines for the structure person

- `.equiv` SALARY, 0 ! the salary member starts at offset 0
- `.equiv` AGE, 4 ! the age member starts at offset 4
- `.equiv` NAME, 8 ! the name member starts at offset 8
- `.equiv` P_SIZE, 24 ! each structure is 24 bytes
Access: Using ‘.’

- C code
  ```c
  struct person joe;  // joe is a structure variable
  
joe.age = 25;        // joe is 25
  ```

- SPARC code
  ```asm
  .seg   "data"
  .align 4
  joe:  .skip P_SIZE   ! joe is a structure
  
  .seg   "text"
  set   25, %r2
  sethi %hi(joe+AGE), %r3
  stw   %r2, [%r3+%lo(joe+AGE)]   ! joe.age = 25
  ```

Slide 56

Access: Using ‘→’

- C code
  ```c
  struct person *sam;   // sam is a pointer to a structure
  
sam→age = 32;        // sam is 32
  ```

- SPARC code
  ```asm
  .seg   "data"
  .align 8
  sam:  .skip 8   ! sam is a pointer
  
  .seg   "text"
  set   32, %r2
  sethi %hi(sam), %r3
  ldx   [%r3+%lo(sam)], %r4   ! %r4 points to the structure
  stw   %r2, [%r4+AGE]        ! sam→age = 32
  ```

Slide 57
Strings

- Directives: .ascii and .asciz
- Representations
  - Length plus value
  - NULL terminated
- An Example

Length Plus Value

- Length in first byte
- Value in remaining bytes
- Example

```assembly
my_str: .byte last_ch - first_ch ! just the length
first_ch: .asci "Hello, world."
lst_ch: ! marking the end
```

- Size of length establishes maximum string length
NULL Terminated

- Value followed by terminating character
- Example

```c
my_str: .asciz "Hello, world."
```
- Inefficiency of string length and concatenation

Calculating String Length

```c
char str[128];
int len;

len = 0;
while( str[len] != 0 ) {
    len = len + 1;
}
```
Simple Control Transfers

```
len = 0;
goto test;

Slide 62
```

top:
```
  len = len + 1;
```
test:
```
if( str[len] ≠ 0 ) goto top;
```

SPARC Code

```
Slide 63
.seg   "data"
.align 8
.reserve str, 128, "bss"
len: .skip 4
```
SPARC Code (continued)

```
Seg "text"
clr %r2      ! len = 0;
set str, %r3
ba %xcc, test  ! goto test
nop
```

Slide 64

top:
inc %r2      ! len = len + 1;
test:
lhb [,%r3+%r2], %r4  ! load str[len]
brnz %r4, top     ! if( str[len] != 0 ) goto top
nop
seti %hi(len), %r4
stw %r2, [%r4+%do(len)]  ! store len

Addresses in Control—switch

- Sample switch statement

```
switch (i) {
    case 1:
        /* statement 1 */
        break;
    case 2:
        /* statement 2 */
        break;
    case 4:
        /* statement 3 */
    }
```

- Cascaded comparisons
- Branch table
Cascaded Comparisons

```
.seg "text"
sethi %hi(i), %r2
ldw [%r2+%lo(i)], %r3
cmp %r3, 1
bne %xcc, try2
nop ! branch delay slot
:
! code for stmt1
bra %xcc, last
nop ! branch delay slot
```

Cascaded Comparisons (continued)

```
try2:
cmp %r3, 2
bre %xcc, lab2
nop ! branch delay slot
ncmp %r3, 4
bne try3
nop ! branch delay slot
lab2:
:
! code for stmt2
bra %xcc, last
nop ! branch delay slot
```
Cascaded Comparisons (continued)

try3:
\[
\begin{align*}
\text{cmp} & \; \%r3, 6 \\
\text{bne} & \; \%xcc, \text{last} \\
\text{nop} & \quad \text{! branch delay slot} \\
\text{;} & \quad \text{! code for stmt3} \\
\text{last}: \\
\end{align*}
\]

Branch Table (the table)

.tab
.data
 Sexe ; initialize the branch table
.word lab1 ; (i = 1) offset 0 \rightarrow \text{stmt1}
.word lab2 ; (i = 2) offset 1 \rightarrow \text{stmt2}
.word last ; (i = 3) offset 2 \rightarrow \text{last}
.word lab2 ; (i = 4) offset 3 \rightarrow \text{stmt2}
.word last ; (i = 5) offset 4 \rightarrow \text{last}
.word lab3 ; (i = 6) offset 5 \rightarrow \text{stmt3}
Branch Table (branching)

```
-seg "text"
sethi %hi(i), %r2
ldsw [%r2+%lo(i)], %r3
cmp %r3, 1
blt %xcc, last  ! nothing to do
cmp %r3, 6
bgt last       ! nothing to do
set tab-4, %r2 ! normalized table base address
add %r3, 2, %r3 ! scale the index
ldw [%r2+%r3], %r4 ! %r4 holds target address
jmp %r4         ! branch delay slot
nop
```

Slide 70

Branch Table (the statements)

```
lab1:
  ...
  bra  %xcc, last      ! code for stmt1
  nop ! branch delay slot
lab2:
  ...
  bra  %xcc, last      ! code for stmt2
  nop ! branch delay slot
lab3:
  ...
  ! code for stmt3
last:
```

Slide 71
Addresses and Integers

- Add an integer to an address
- Subtract an integer from an address
- Subtract two addresses

Addressing Modes on the HP PA

- Displacement

Slide 73

- pre and post modification
- Indexed
  - post modification
  - scaling
Displacement Addressing on the HP PA

- Base register not modified

Slide 74

- Base register modified, post-displacement

- Base register modified, pre-displacement

Indexed Addressing on the HP PA

- Simple indexing

Slide 75

- Base register modification
Indexed Addressing on the HP PA (continued)

- Indexing with scaling

- Base register modification with scaling