Chapter 6 – Languages and the Machine
Chapter Contents

6.1 The Compilation Process
6.2 The Assembly Process
6.3 Linking and Loading
6.4 Macros
6.5 Quantitative Analyses of Program Execution
6.6 From CISC to RISC
6.7 Pipelining the Datapath
6.8 Overlapping Register Windows
6.9 Low Power Coding
The Compilation Process

- **Compilation** translates a program written in a high level language into a functionally equivalent program in assembly language.

- Consider a simple high-level language assignment statement:

  \[ A = B + 4; \]

- Steps involved in compiling this statement into assembly code:

  - Reducing the program text to the basic symbols of the language (for example, into identifiers such as A and B), denotations such as the constant value 4, and program delimiters such as = and +. This portion of compilation is referred to as *lexical analysis*.

  - Parsing symbols to recognize the underlying program structure. For the statement above, the parser must recognize the form:

    Identifier “=” Expression,
    
    where Expression is further parsed into the form:

    Identifier “+” Constant.

    Parsing is sometimes called *syntactic analysis*. 

The Compilation Process

— Name analysis: associating the names A and B with particular program variables, and further associating them with particular memory locations where the variables are located at run time.

— Type analysis: determining the types of all data items. In the example above, variables A and B and constant 4 would be recognized as being of type int in some languages. Name and type analysis are sometimes referred to together as semantic analysis: determining the underlying meaning of program components.

— Action mapping and code generation: associating program statements with their appropriate assembly language sequence. In the statement above, the assembly language sequence might be as follows:

```
ld [B], %r0, %r1  # Get variable B into a register.
add %r1, 4, %r2  # Compute the value of the expression
st %r2, %r0, [A]  # Make the assignment.
```
The Assembly Process

• The process of translating an assembly language program into a machine language program is referred to as the assembly process.

• Production assemblers generally provide this support:
  — Allow programmer to specify locations of data and code.
  — Provide assembly-language mnemonics for all machine instructions and addressing modes, and translate valid assembly language statements into the equivalent machine language.
  — Permit symbolic labels to represent addresses and constants.
  — Provide a means for the programmer to specify the starting address of the program, if there is one; and provide a degree of assemble-time arithmetic.
  — Include a mechanism that allows variables to be defined in one assembly language program and used in another, separately assembled program.
  — Support macro expansion.
Assembly Example

• We explore how the assembly process proceeds by “hand assembling” a simple ARC assembly language program.

! This program adds two numbers

```
.begin
.org 2048
main:  ld    [x], %r1                   ! Load x into %r1
       ld    [y], %r2                   ! Load y into %r2
       addcc %r1, %r2, %r3            ! %r3 ← %r1 + %r2
       st    %r3, [z]                  ! Store %r3 into z
       jmpl  %r15 + 4, %r0            ! Return
```

x: 15
y: 9
z: 0

.end
Instruc- tion Formats and PSR Format for the ARC

<table>
<thead>
<tr>
<th>op</th>
<th>Format</th>
<th>op2</th>
<th>Inst.</th>
<th>op3 (op=10)</th>
<th>op3 (op=11)</th>
<th>cond</th>
<th>branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SETHI/Branch</td>
<td>010</td>
<td>branch</td>
<td>010000 addcc</td>
<td>000000 ld</td>
<td>0001</td>
<td>be</td>
</tr>
<tr>
<td></td>
<td></td>
<td>100</td>
<td>sethi</td>
<td>010001 addcc</td>
<td>000100 st</td>
<td>0101</td>
<td>bcs</td>
</tr>
<tr>
<td>01</td>
<td>CALL</td>
<td>010</td>
<td>branch</td>
<td>010010 crccc</td>
<td>000101 neg</td>
<td>0110</td>
<td>bne</td>
</tr>
<tr>
<td>10</td>
<td>Arithmetic</td>
<td>010</td>
<td>branch</td>
<td>010110 crcc</td>
<td>0011 bvs</td>
<td>0111</td>
<td>bvs</td>
</tr>
<tr>
<td>11</td>
<td>Memory</td>
<td>010</td>
<td>branch</td>
<td>110110 srl</td>
<td>1000 ba</td>
<td>11000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PSR</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>n</td>
<td>z</td>
<td>v</td>
<td>c</td>
<td>p</td>
<td>t</td>
</tr>
</tbody>
</table>
Assembled Code

```
ld [x], %r1
  1100 0010 0000 0000 0010 1000 0001 0100
ld [y], %r2
  1100 0100 0000 0000 0010 1000 0001 1000
addcc %r1,%r2,%,r3
  1000 0110 1000 0000 0100 0000 0000 0010
st %r3, [z]
  1100 0110 0010 0000 0010 1000 0001 1100
jmpl %r15+4, %r0
  1000 0001 1100 0011 1110 0000 0000 0100
15
  0000 0000 0000 0000 0000 0000 0000 1111
9
  0000 0000 0000 0000 0000 0000 0000 1001
0
  0000 0000 0000 0000 0000 0000 0000 0000
```
Forward Referencing

• An example of forward referencing:

```plaintext
... 
... 
  call sub_r  ! Subroutine is invoked here
  ... 
sub_r:    st  %r1, [w]  ! Subroutine is defined here
  ... 
... 
```
Creating a Symbol Table

(a)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_start</td>
<td>3000</td>
</tr>
<tr>
<td>length</td>
<td></td>
</tr>
</tbody>
</table>

(b)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a_start</td>
<td>3000</td>
</tr>
<tr>
<td>length</td>
<td>2092</td>
</tr>
<tr>
<td>address</td>
<td>2096</td>
</tr>
<tr>
<td>loop</td>
<td>2060</td>
</tr>
<tr>
<td>done</td>
<td>2088</td>
</tr>
<tr>
<td>a</td>
<td>3000</td>
</tr>
</tbody>
</table>
### Assembled Program

<table>
<thead>
<tr>
<th>Location counter</th>
<th>Instruction</th>
<th>Object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>2048</td>
<td>.begin</td>
<td>11000010 00000000 00101000 00101100</td>
</tr>
<tr>
<td></td>
<td>.org 2048</td>
<td>11000100 00000000 00101000 00110000</td>
</tr>
<tr>
<td></td>
<td>a_start .equ 3000</td>
<td>10000100 10001000 11000000 00000000</td>
</tr>
<tr>
<td>2048</td>
<td>ld [length],%r1</td>
<td>00000000 10000000 00000000 00000011</td>
</tr>
<tr>
<td>2052</td>
<td>ld [address],%r2</td>
<td>00000000 10000000 01111111 11111100</td>
</tr>
<tr>
<td>2056</td>
<td>andcc %r3,%r0,%r3</td>
<td>10000000 10001000 01000000 00000001</td>
</tr>
<tr>
<td>2060</td>
<td>loop: andcc %r1,%r1,%r0</td>
<td>00000000 10000000 00000000 00000011</td>
</tr>
<tr>
<td>2064</td>
<td>be done</td>
<td>00000010 10000000 00000000 00000110</td>
</tr>
<tr>
<td>2068</td>
<td>addcc %r1,-4,%r1</td>
<td>00000010 10000000 01111111 11111100</td>
</tr>
<tr>
<td>2072</td>
<td>addcc %r1,%r2,%r4</td>
<td>10001000 10000000 01000000 00000001</td>
</tr>
<tr>
<td>2076</td>
<td>ld %r4,%r5</td>
<td>00000000 10000000 00000000 00000000</td>
</tr>
<tr>
<td>2080</td>
<td>ba loop</td>
<td>00000000 10111111 11111111 11111011</td>
</tr>
<tr>
<td>2084</td>
<td>addcc %r3,%r5,%r3</td>
<td>10000010 10000000 11000000 00000101</td>
</tr>
<tr>
<td>2088</td>
<td>done: jmpl %r15+4,%r0</td>
<td>10000001 11000011 11100000 00000000</td>
</tr>
<tr>
<td>2092</td>
<td>length: 20</td>
<td>00000000 00000000 00000000 00101000</td>
</tr>
<tr>
<td>2096</td>
<td>address: a_start</td>
<td>00000000 00000000 00000000 00010111 10111000</td>
</tr>
<tr>
<td></td>
<td>.org a_start</td>
<td>00000000 00000000 00000000 00010111 10111000</td>
</tr>
<tr>
<td>3000</td>
<td>a: 25</td>
<td>00000000 00000000 00000000 00011001</td>
</tr>
<tr>
<td>3004</td>
<td>-10</td>
<td>00000000 00000000 00000000 00110110</td>
</tr>
<tr>
<td>3008</td>
<td>33</td>
<td>00000000 00000000 00000000 00100001</td>
</tr>
<tr>
<td>3012</td>
<td>-5</td>
<td>00000000 00000000 00000000 00110111</td>
</tr>
<tr>
<td>3016</td>
<td>7</td>
<td>00000000 00000000 00000000 00000111</td>
</tr>
<tr>
<td></td>
<td>.end</td>
<td>00000000 00000000 00000000 00000011</td>
</tr>
</tbody>
</table>
Linking: Using `.global` and `.extern`

- A `.global` is used in the module where a symbol is defined and a `.extern` is used in every other module that refers to it.

<table>
<thead>
<tr>
<th>! Main program</th>
<th>! Subroutine library</th>
</tr>
</thead>
<tbody>
<tr>
<td>.begin</td>
<td>.begin</td>
</tr>
<tr>
<td>.org 2048</td>
<td>.org 2048</td>
</tr>
<tr>
<td>.extern sub</td>
<td>.global sub</td>
</tr>
<tr>
<td>main: ld [x], %r2</td>
<td>sub: ornc %r3, %r0, %r3</td>
</tr>
<tr>
<td>ld [y], %r3</td>
<td>addcc %r3, ONE, %r3</td>
</tr>
<tr>
<td>call sub</td>
<td>jmpl %r15 + 4, %r0</td>
</tr>
<tr>
<td>jmpl %r15 + 4, %r0</td>
<td>.end</td>
</tr>
<tr>
<td>x: 105</td>
<td></td>
</tr>
<tr>
<td>y: 92</td>
<td></td>
</tr>
<tr>
<td>.end</td>
<td></td>
</tr>
</tbody>
</table>
Linking and Loading: Symbol Tables

- Symbol tables for the previous example:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Global/External</th>
<th>Relocatable</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub</td>
<td>–</td>
<td>External</td>
<td>–</td>
</tr>
<tr>
<td>main</td>
<td>2048</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>x</td>
<td>2064</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>y</td>
<td>2068</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Main Program

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Global/External</th>
<th>Relocatable</th>
</tr>
</thead>
<tbody>
<tr>
<td>ONE</td>
<td>1</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>sub</td>
<td>2048</td>
<td>Global</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Subroutine Library
Example ARC Program

```assembly
! Perform a 64-bit addition: C ← A + B
! Register usage: %r1 - Most significant 32 bits of A
! %r2 - Least significant 32 bits of A
! %r3 - Most significant 32 bits of B
! %r4 - Least significant 32 bits of B
! %r5 - Most significant 32 bits of C
! %r6 - Least significant 32 bits of C
! %r7 - Used for restoring carry bit

.begin     ! Start assembling
.globl main
.org 2048   ! Start program at 2048
main:
  ld [A], %r1 ! Get high word of A
  ld [A+4], %r2 ! Get low word of A
  ld [B], %r3 ! Get high word of B
  ld [B+4], %r4 ! Get low word of B
  call add_64 ! Perform 64-bit addition
  st %r5, [C] ! Store high word of C
  st %r6, [C+4] ! Store low word of C

.org 3072   ! Start add_64 at 3072
add_64:
  addcc %r2, %r4, %r6 ! Add low order words
  bcs lo_carry ! Branch if carry set
  addcc %r1, %r3, %r5 ! Add high order words
  jmpl %r15 + 4, %r0 ! Return to calling routine
lo_carry:
  addcc %r1, %r3, %r5 ! Add high order words
  bcs hi_carry ! Branch if carry set
  addcc %r5, 1, %r5 ! Add in carry
  jmpl %r15, 4, %r0 ! Return to calling routine
hi_carry:
  addcc %r5, 1, %r5 ! Add in carry
  sethi #3FFFFFF, %r7 ! Set up %r7 for carry
  addcc %r7, %r7, %r0 ! Generate a carry
  jmpl %r15 + 4, %r0 ! Return to calling routine

A: 0 ! High 32 bits of 25
    25 ! Low 32 bits of 25
B: #FFFFFFFE ! High 32 bits of -1
    #FFFFFFFE ! Low 32 bits of -1
C: 0 ! High 32 bits of result
    0 ! Low 32 bits of result
.end       ! Stop assembling
```
Macro Definition

- A macro definition for `push`:

```assembly
! Macro definition for 'push'
.macro push arg1          ! Start macro definition
addcc  %r14, -4, %r14     ! Decrement stack pointer
st     arg1, %r14         ! Push arg1 onto stack
.endmacro                   ! End macro definition
```
Recursive Macro Expansion

! A recursive macro definition
.macro recurs_add X  ! Start macro definition
 .if X > 2  ! Assemble code if X > 2
   recurs_add X - 1  ! Recursive call
 .endif  ! End .if construct
       addcc %r1, %rX, %r1  ! Add argument into %r1
 .endmacro  ! End macro definition

recurs_add 4  ! Invoke the macro

Expands to:
addcc %r1, %r2, %r1
addcc %r1, %r3, %r1
addcc %r1, %r4, %r1
Instruction Frequency

- Frequency of occurrence of instruction types for a variety of languages. The percentages do not sum to 100 due to roundoff. (Adapted from Knuth, D. E., *An Empirical Study of FORTRAN Programs*, *Software—Practice and Experience*, 1, 105-133, 1971.)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Average Percent of Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>47</td>
</tr>
<tr>
<td>If</td>
<td>23</td>
</tr>
<tr>
<td>Call</td>
<td>15</td>
</tr>
<tr>
<td>Loop</td>
<td>6</td>
</tr>
<tr>
<td>Goto</td>
<td>3</td>
</tr>
<tr>
<td>Other</td>
<td>7</td>
</tr>
</tbody>
</table>
Complexity of Assignments

- Percentages showing complexity of assignments and procedure calls.


<table>
<thead>
<tr>
<th></th>
<th>Percentage of Number of Terms in Assignments</th>
<th>Percentage of Number of Locals in Procedures</th>
<th>Percentage of Number of Parameters in Procedure Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>—</td>
<td>22</td>
<td>41</td>
</tr>
<tr>
<td>1</td>
<td>80</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>15</td>
<td>20</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>≥ 5</td>
<td>0</td>
<td>20</td>
<td>8</td>
</tr>
</tbody>
</table>
Speedup and Efficiency

- Speedup $S$ is the ratio of the time needed to execute a program without an enhancement to the time required with an enhancement.

$$ S = \frac{T_{w0}}{T_w} \quad S = \frac{T_{w0} - T_w}{T_w} \times 100 $$

- Time $T$ is computed as the instruction count $IC$ times the number of cycles per instruction $CPI$ times the cycle time $\tau$.

$$ T = IC \times CPI \times \tau $$

- Substituting $T$ into the speedup percentage calculation above yields:

$$ S = \frac{IC_{w0} \times CPI_{w0} \times \tau_{w0} - IC_w \times CPI_w \times \tau_w}{IC_w \times CPI_w \times \tau_w} \times 100 $$
Example

• *Example:* Estimate the speedup obtained by replacing a CPU having an average CPI of 5 with another CPU having an average CPI of 3.5, with the clock period increased from 100 ns to 120 ns.

• The previous equation becomes:

\[
S = \frac{5 \times 100 - 3.5 \times 120}{3.5 \times 120} \times 100 = 19\%
\]
Four/Five-Stage Instruction Pipeline

- We used a five-step fetch-execute cycle earlier: (1) instruction fetch, (2) decode, (3) operand fetch, (4) ALU operation, (5) result writeback.

- These five phases can be thought of as only four phases in which the fourth phase, “execute,” has two subphases: ALU operation and writeback. A result writeback is not always needed and can be bypassed, thus the five phases are only four phases some of the time. For this discussion, we take a simple approach and force all instructions to go entirely through each phase whether or not that is actually needed, and so the ALU operation and writeback that are combined below will be implemented in five phases here.
Pipeline Behavior

- Pipeline behavior during a memory reference and during a branch.

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Instruction Fetch</td>
<td>addcc</td>
</tr>
<tr>
<td>Decode</td>
<td></td>
</tr>
<tr>
<td></td>
<td>addcc</td>
</tr>
<tr>
<td>Operand Fetch</td>
<td></td>
</tr>
<tr>
<td>Execute</td>
<td></td>
</tr>
<tr>
<td>Memory Reference</td>
<td></td>
</tr>
</tbody>
</table>

`addcc`, `ld`, `srl`, `subcc`, and `be` are instructions, and `nop` is a no-operation (NOP) instruction.
Filling the Load Delay Slot

- SPARC code, (a) with a `nop` inserted, and (b) with `srl` migrated to `nop` position.

```
srl   %r3, %r5
addcc %r1, 10, %r1
ld    %r1, %r2
nop
subcc %r2, %r4, %r4
be    label

(a)
```

```
addcc %r1, 10, %r1
ld    %r1, %r2
srl   %r3, %r5
subcc %r2, %r4, %r4
be    label

(b)
```
Call-Return Behavior


Time in Units of Calls/Returns

Nesting Depth

Window depth = 5
SPARC Registers

- User view of RISC I registers.

32 bits

<table>
<thead>
<tr>
<th>%g0</th>
<th>R0</th>
<th>Global Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>%g7</td>
<td>R7</td>
<td>Incoming Parameters</td>
</tr>
<tr>
<td>%i0</td>
<td>R8</td>
<td>Local Variables</td>
</tr>
<tr>
<td>%i7</td>
<td>R15</td>
<td>Outcoming Parameters</td>
</tr>
<tr>
<td>%i10</td>
<td>R16</td>
<td></td>
</tr>
<tr>
<td>%i17</td>
<td>R23</td>
<td></td>
</tr>
<tr>
<td>%o0</td>
<td>R24</td>
<td></td>
</tr>
<tr>
<td>%o7</td>
<td>R31</td>
<td></td>
</tr>
</tbody>
</table>
Overlapping Register Windows

CWP = 8

R0
R7

R8
R15
R16
R23
R24
R31

R0
R7

R8
R15
R16
R23
R24
R31

Globals

Ins
Locals
Outs

Overlap

Procedure A

CWP = 24

Ins
Locals
Outs

Procedure B
Example: Compiled C Program

- Source code for C program to be compiled with gcc.

```c
/* Example C program to be compiled with gcc */

#include
<stdio.h>

main ()
{
    int a, b, c;
    a = 10;
    b = 4;
    c = add_two(a, b);
    printf("c = %d\n", c);
}

int add_two(a,b)
int a, b;
{
    int result;
    result = a + b;
    return(result);
}
```
gcc
Generated
SPARC
Code
gcc
Generated
SPARC
Code (cont’)

call printf, 0
nop ! A nop is needed here because of the pipeline flush
! that follows a transfer of control.
.LL1
ret ! Return to calling routine (Solaris for this case)
restore ! The complement to save. Although it follows the
! return, it is still in the pipeline and gets executed.
.LLfe1
.size main, .LLfe1-main ! Size of
.align 4
global add_two
type add_two, #function
.proc 04
add_two:
!#PROLOGUE# 0
save %sp, -120, %sp
!#PROLOGUE# 1
st %t0, [%fp+68] !Same as %c0 in calling routine (variable a)
st %t1, [%fp+72] !Same as %c1 in calling routine (variable b)
ld [%fp+68], %c0
ld [%fp+72], %c1
add %c0, %c1, %c0 ! Perform the addition
st %c0, [%fp-20] ! Store result in stack frame
ld [%fp-20], %t0 ! %t0 (result) is %c0 in called routine
b .LL2
nop
.LL2:
ret
restore
.LLfe2:
.size add_two, .LLfe2-add_two
.ident "GCC: (GNU) 2.5.8"
Effect of Compiler Optimization

- SPARC code generated with the -O optimization flag:

```assembly
.file "add.c"
.section ".rodata"
.align 8
.LLC0:
.asciz "c = \n"
.section ".text"
.align 4
.global main
.type main,#function
.proc 04
main:
!#PROLOGUE# 0
save %sp,-112,%sp
!#PROLOGUE# 1
mov 10,%0
call add_two,0
mov 4,%0
mov %0,%1
sethi %hi(.LLC0),%0
call printf,0
or %0,%lo(.LLC0),%0
ret
restore
.LLfe1:
.size main,LLfe1-main
.align 4
.global add_two
.type add_two,#function
.proc 04
add_two:
!#PROLOGUE# 0
!#PROLOGUE# 1
retl
add %0,%0,1,%0
.LLfe2:
.size add_two,LLfe2-add_two
.ident "GCC: (GNU) 2.7.2"
```
Low Power Coding

• Consider the ARC sequence shown below:

\[ \text{ld [2096], \%r16} \]
\[ \text{ld [2100], \%r17} \]
\[ \text{ld [2104], \%r18} \]
\[ \text{ld [2108], \%r19} \]

• The corresponding machine code is:

| op  | rd  | op3 | rs1 | i  | simm13 | No. Transitions
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10000</td>
<td>00000</td>
<td>00000</td>
<td>1</td>
<td>0100000110000</td>
<td>ld [2096],%r16</td>
</tr>
<tr>
<td>11</td>
<td>10001</td>
<td>00000</td>
<td>00000</td>
<td>1</td>
<td>0100000110100</td>
<td>ld [2100],%r17</td>
</tr>
<tr>
<td>11</td>
<td>10010</td>
<td>00000</td>
<td>00000</td>
<td>1</td>
<td>0100000111000</td>
<td>ld [2104],%r18</td>
</tr>
<tr>
<td>11</td>
<td>10011</td>
<td>00000</td>
<td>00000</td>
<td>1</td>
<td>0100000111100</td>
<td>ld [2108],%r19</td>
</tr>
</tbody>
</table>

Total: 8

(Continued on next slide)
Low Power Coding (Continued)

- The total number of transitions for the code in the previous slide is eight. However, if we reorder the last two instructions, then the total number of transitions is reduced to only six:

<table>
<thead>
<tr>
<th>op</th>
<th>rd</th>
<th>op3</th>
<th>rs1</th>
<th>i</th>
<th>simm13</th>
<th>No. Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>10000</td>
<td>00000</td>
<td>00000</td>
<td>1</td>
<td>0100000110000</td>
<td>ld [2096], %r16 -</td>
</tr>
<tr>
<td>11</td>
<td>10001</td>
<td>00000</td>
<td>00000</td>
<td>1</td>
<td>01000001101000</td>
<td>ld [2100], %r17 2</td>
</tr>
<tr>
<td>11</td>
<td>10011</td>
<td>00000</td>
<td>00000</td>
<td>1</td>
<td>01000001111000</td>
<td>ld [2108], %r19 2</td>
</tr>
<tr>
<td>11</td>
<td>10010</td>
<td>00000</td>
<td>00000</td>
<td>1</td>
<td>01000001110000</td>
<td>ld [2104], %r18 2</td>
</tr>
</tbody>
</table>

Total: 6

- There are several other optimizations that can be made to code sequences based on choosing instructions and parameters from functionally equivalent possibilities in such a way that the number of transitions are reduced.

*(Continued on next slide)*
Low Power Coding (continued)

- The use of a Gray code reduces the number of transitions in sequences of instructions, and in sequences of addresses:

<table>
<thead>
<tr>
<th>Normal encoding</th>
<th>No. of transitions</th>
<th>Gray code</th>
<th>No. of transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>–</td>
<td>000</td>
<td>–</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>001</td>
<td>1</td>
</tr>
<tr>
<td>010</td>
<td>2</td>
<td>011</td>
<td>1</td>
</tr>
<tr>
<td>011</td>
<td>1</td>
<td>010</td>
<td>1</td>
</tr>
<tr>
<td>100</td>
<td>3</td>
<td>110</td>
<td>1</td>
</tr>
<tr>
<td>101</td>
<td>1</td>
<td>111</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>2</td>
<td>101</td>
<td>1</td>
</tr>
<tr>
<td>111</td>
<td>1</td>
<td>100</td>
<td>1</td>
</tr>
</tbody>
</table>

Total: 11

Total: 7