Principles of Computer Architecture

*Miles Murdocca and Vincent Heuring*

Chapter 5: Languages and the Machine
Chapter Contents

5.1 The Compilation Process
5.2 The Assembly Process
5.3 Linking and Loading
5.4 Macros
5.5 Case Study: Extensions to the Instruction Set – The Intel MMX™ and Motorola AltiVec™ SIMD Instructions
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
```
### Instruction Formats and PSR Format for the ARC

#### SETHI Format
```
<table>
<thead>
<tr>
<th>op</th>
<th>Format</th>
<th>rd</th>
<th>op2</th>
<th>Inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>SETHI/Branch</td>
<td>0</td>
<td>0</td>
<td>branch</td>
</tr>
<tr>
<td>01</td>
<td>CALL</td>
<td>0</td>
<td>0</td>
<td>sethi</td>
</tr>
<tr>
<td>10</td>
<td>Arithmetic</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Memory</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
```

#### Branch Format
```
<table>
<thead>
<tr>
<th>op</th>
<th>Format</th>
<th>cond</th>
<th>op2</th>
<th>Inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>addcc</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td>0</td>
<td>andcc</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>1</td>
<td>orcc</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>srl</td>
</tr>
</tbody>
</table>
```

#### CALL Format
```
<table>
<thead>
<tr>
<th>op</th>
<th>Format</th>
<th>disp30</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>
```

#### Arithmetic Formats
```
<table>
<thead>
<tr>
<th>op</th>
<th>Format</th>
<th>rd</th>
<th>op3</th>
<th>rsi</th>
<th>imm13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000</td>
</tr>
</tbody>
</table>
```

#### Memory Formats
```
<table>
<thead>
<tr>
<th>op</th>
<th>Format</th>
<th>rd</th>
<th>op3</th>
<th>rsi</th>
<th>imm13</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00000</td>
</tr>
</tbody>
</table>
```

#### PSR Format
```
<table>
<thead>
<tr>
<th>n</th>
<th>z</th>
<th>v</th>
<th>c</th>
</tr>
</thead>
</table>
```

---

**Chapter 5: Languages and the Machine**
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
    .
    .
    .
```
This program sums LENGTH numbers

Register usage:
%r1 - Length of array a
%r2 - Starting address of array a
%r3 - The partial sum
%r4 - Pointer into array a
%r5 - Holds an element of a

.begin  ! Start assembling
.org 2048  ! Start program at 2048
a_start .equ 3000  ! Address of array a

ld [length], %r1 ! %r1 ← length of array a
ld [address],%r2 ! %r2 ← address of a
andcc %r3, %r0, %r3 ! %r3 ← 0

loop: andcc %r1, %r1, %r0 ! Test # remaining elements
be done  ! Finished when length=0
addcc %r1, -4, %r1  ! Decrement array length
addcc %r1, %r2, %r4 ! Address of next element
ld %r4, %r5  ! %r5 ← Memory[%r4]
addcc %r3, %r5, %r3 ! Sum new element into r3
ba loop  ! Repeat loop.

done: jmpl %r15 + 4, %r0 ! Return to calling routine

length: 20  ! 5 numbers (20 bytes) in a

address: a_start
.org a_start  ! Start of array a

a: 25  ! length/4 values follow
    -10
    33
    -5
    7
.end  ! Stop assembling

Symbol Table

<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>

(a)

Symbol Table

<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>

(b)
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></td>
</tr>
<tr>
<td></td>
<td>.org 2048</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a_start .equ 3000</td>
<td></td>
</tr>
<tr>
<td>2048</td>
<td>ld  [length],%r1</td>
<td>11000010 00000000 00101000 00101100</td>
</tr>
<tr>
<td>2052</td>
<td>ld  [address],%r2</td>
<td>11000100 00000000 00101000 00110000</td>
</tr>
<tr>
<td>2056</td>
<td>andcc %r3,%r0,%r3</td>
<td>10000110 10001000 11000000 00000000</td>
</tr>
<tr>
<td>2060</td>
<td>loop: andcc %r1,%r1,%r0</td>
<td>10000000 10001000 01000000 00000000</td>
</tr>
<tr>
<td>2064</td>
<td>be  done</td>
<td>00000010 10000000 00000000 0000110</td>
</tr>
<tr>
<td>2068</td>
<td>addcc %r1,-4,%r1</td>
<td>10000010 10000000 00000000 00000000</td>
</tr>
<tr>
<td>2072</td>
<td>addcc %r1,%r2,%r4</td>
<td>10000010 10000000 00000000 00000000</td>
</tr>
<tr>
<td>2076</td>
<td>ld  %r4,%r5</td>
<td>11001010 00000001 00000000 00000000</td>
</tr>
<tr>
<td>2080</td>
<td>ba  loop</td>
<td>00000000 10111111 11111111 11111111</td>
</tr>
<tr>
<td>2084</td>
<td>addcc %r3,%r5,%r3</td>
<td>10000010 10000000 11000000 00000010</td>
</tr>
<tr>
<td>2088</td>
<td>done: jmpl  %r15+4,%r0</td>
<td>10000000 11000011 11100000 00000100</td>
</tr>
<tr>
<td>2092</td>
<td>length:  20</td>
<td>00000000 00000000 00000000 00010100</td>
</tr>
<tr>
<td>2096</td>
<td>address:  a_start</td>
<td>00000000 00000000 00010111 10111000</td>
</tr>
<tr>
<td></td>
<td>.org  a_start</td>
<td></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>11111111 11111111 11111111 11110110</td>
</tr>
<tr>
<td>3008</td>
<td>33</td>
<td>00000000 00000000 00000000 00100001</td>
</tr>
<tr>
<td>3012</td>
<td>-5</td>
<td>11111111 11111111 11111111 11111011</td>
</tr>
<tr>
<td>3016</td>
<td>7</td>
<td>00000000 00000000 00000000 00000111</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.

```
! Main program
   .begin
   .org 2048
   .extern sub
main: ld [x], %r2
     ld [y], %r3
     call sub
     jmpl %r15 + 4, %r0
x: 105
y: 92
  .end

! Subroutine library
   .begin
ONE .equ 1
   .org 2048
sub: orncc %r3, %r0, %r3
     addcc %r3, ONE, %r3
     jmpl %r15 + 4, %r0
   .end
```
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>

<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

Main Program
Example ARC Program

! 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
.global 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

Principles of Computer Architecture by M. Murdocca and V. Heuring
Macro Definition

- A macro definition for `push`:

```
! Macro definition for 'push'
.macro    push arg1
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
    .if  X > 2
        recurs_add  X - 1
    .endif
    addcc  %r1, %rX, %r1
.endmacro

recurs_add  4  ! Invoke the macro

Expands to:
addcc  %r1, %r2, %r1
addcc  %r1, %r3, %r1
addcc  %r1, %r4, %r1
Intel MMX (MultiMedia eXtensions)

- Vector addition of eight bytes by the Intel PADDB mm0, mm1 instruction:

\[
\begin{array}{c}
\text{mm0} & \text{mm1} \\
11111111 & 00000000 & 01101001 & 10111111 & 00101010 & 01101010 & 10101111 & 10111101 \\
+ & + & + & + & + & + & + & + \\
11111110 & 11111111 & 00011111 & 10101010 & 11111111 & 00101011 & 11010101 & 00101010 \\
= & = & = & = & = & = & = & = \\
\text{mm0} & \text{mm0} \\
11111101 & 11111111 & 01111000 & 01101001 & 00101001 & 01111111 & 10000100 & 11100111 \\
\end{array}
\]
**Intel and Motorola Vector Registers**

- Intel “aliases” the floating point registers as MMX registers. This means that the Pentium’s 8 64-bit floating-point registers do double-duty as MMX registers.

- Motorola implements 32 128-bit vector registers as a new set, separate and distinct from the floating-point registers.

<table>
<thead>
<tr>
<th>Intel MMX Registers</th>
<th>Motorola AltiVec Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>63</td>
<td>127</td>
</tr>
<tr>
<td>MM7</td>
<td>VR31</td>
</tr>
<tr>
<td>•</td>
<td>VR30</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>MM0</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>VR1</td>
</tr>
<tr>
<td></td>
<td>VR0</td>
</tr>
</tbody>
</table>
### MMX and Altivec Arithmetic Instructions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operands (bits)</th>
<th>Arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Add, Subtract, signed and unsigned(B)</td>
<td>8, 16, 32, 64, 128</td>
<td>Modulo, Saturated</td>
</tr>
<tr>
<td>Integer Add, Subtract, store carry-out in vector reg.(M)</td>
<td>32</td>
<td>Modulo</td>
</tr>
<tr>
<td>Integer Multiply, store high- or low order half (I)</td>
<td>16←16×16</td>
<td>—</td>
</tr>
<tr>
<td>Integer multiply add: Vd = Va *Vb + Vc (B)</td>
<td>16←8×8 32←16×16</td>
<td>Modulo, Saturated</td>
</tr>
<tr>
<td>Shift Left, Right, Arithmetic Right(B)</td>
<td>8, 16, 32, 64(I)</td>
<td>—</td>
</tr>
<tr>
<td>Rotate Left, Right (M)</td>
<td>8, 16, 32</td>
<td>—</td>
</tr>
<tr>
<td>AND, AND NOT, OR, NOR, XOR(B)</td>
<td>64(I), 128(M)</td>
<td>—</td>
</tr>
<tr>
<td>Integer Multiply every other operand, store entire result, signed and unsigned(M)</td>
<td>16←8×8 32←16×16</td>
<td>Modulo, Saturated</td>
</tr>
<tr>
<td>Maximum, minimum. Vd←Max,Min(Va, Vb) (M)</td>
<td>8, 16, 32</td>
<td>Signed, Unsigned</td>
</tr>
<tr>
<td>Vector sum across word. Add objects in vector, add this sum to object in second vector, place result in third vector register.(M)</td>
<td>Various</td>
<td>Modulo, Saturated</td>
</tr>
<tr>
<td>Vector floating point operations, add, subtract, multiply-add, etc. (M)</td>
<td>32</td>
<td>IEEE Floating Point</td>
</tr>
</tbody>
</table>
Comparing Two MMX Byte Vectors for Equality

mm0 | 11111111 00000000 00000000 10101010 00101010 01101010 10101111 10111101 |
mm1 | 11111111 11111111 00000000 10101010 00101011 01101010 11010101 00101010 |

(T) (F) (T) (T) (F) (T) (F) (F)

mm0 | 11111111 00000000 11111111 11111111 00000000 11111111 00000000 00000000 |

(T) (F) (T) (T) (F) (T) (F) (F)
Conditional Assignment of an MMX Byte Vector

<table>
<thead>
<tr>
<th>mm0</th>
<th>11111111</th>
<th>00000000</th>
<th>11111111</th>
<th>11111111</th>
<th>00000000</th>
<th>11111111</th>
<th>00000000</th>
<th>00000000</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>10110011</td>
<td>10001101</td>
<td>01100110</td>
<td>10101010</td>
<td>00101011</td>
<td>01101010</td>
<td>11010101</td>
<td>00101010</td>
</tr>
<tr>
<td>mm2</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
</tr>
<tr>
<td>mm2</td>
<td>10110011</td>
<td>00000000</td>
<td>01100110</td>
<td>10101010</td>
<td>00000000</td>
<td>01101010</td>
<td>00000000</td>
<td>00000000</td>
</tr>
</tbody>
</table>