Principles of Computer Architecture

Miles Murdocca and Vincent Heuring

Chapter 5: Languages and the Machine

Chapter Contents

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 assemby 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.**

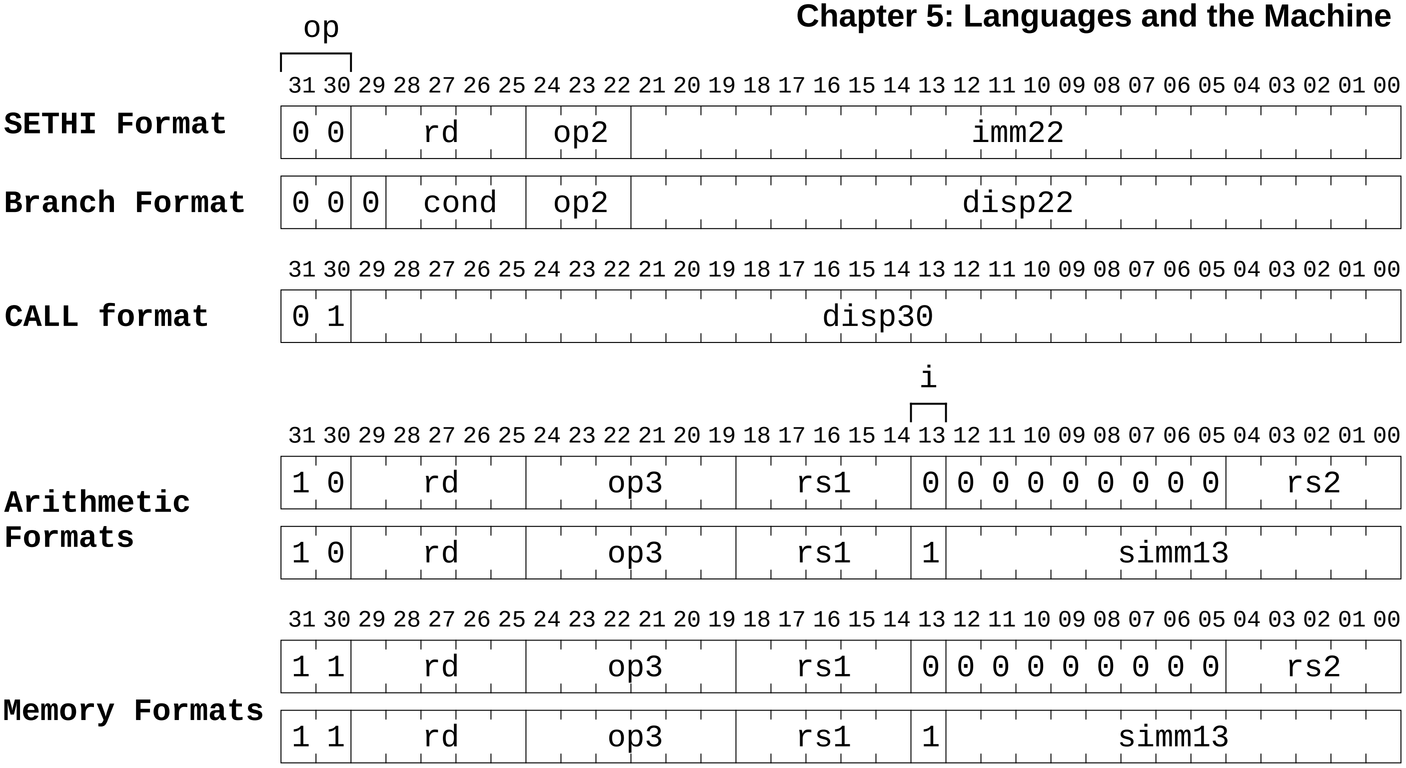
Assembly Example

- **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



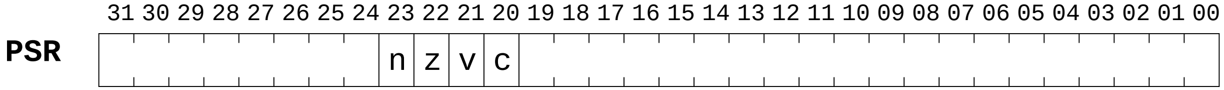
op	Format
00	SETHI/Branch
01	CALL
10	Arithmetic
11	Memory

op2	Inst.
010	branch
100	sethi

op3 (op=10)	
010000	addcc
010001	andcc
010010	orcc
010110	orncc
100110	srl
111000	jmpl

op3 (op=11)	
000000	ld
000100	st

cond	branch
0001	be
0101	bcs
0110	bneg
0111	bvs
1000	ba



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:**

```
              .
              .
              .
            call  sub_r       ! Subroutine is invoked here
              .
              .
              .
sub_r:      st    %r1, [w]    ! Subroutine is defined here
              .
              .
              .
```

Creating a Symbol Table

```
! 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	Value
a_start	3000
length	—

(a)

Symbol	Value
a_start	3000
length	2092
address	2096
loop	2060
done	2088
a	3000

(b)

Assembled Program

```
Location       Instruction                     Object code
 counter
               .begin
               .org  2048
     a_start   .equ  3000
2048           ld    [length],%r1   11000010 00000000 00101000 00101100
2052           ld   [address],%r2   11000100 00000000 00101000 00110000
2056           andcc %r3,%r0,%r3    10000110 10001000 11000000 00000000
2060 loop:     andcc %r1,%r1,%r0    10000000 10001000 01000000 00000001
2064           be    done           00000010 10000000 00000000 00000110
2068           addcc %r1,-4,%r1     10000010 10000000 01111111 11111100
2072           addcc %r1,%r2,%r4    10001000 10000000 01000000 00000010
2076           ld    %r4,%r5        11001010 00000001 00000000 00000000
2080           ba    loop           00010000 10111111 11111111 11111011
2084           addcc %r3,%r5,%r3    10000110 10000000 11000000 00000101
2088 done:     jmpl  %r15+4,%r0     10000001 11000011 11100000 00000100
2092 length:          20            00000000 00000000 00000000 00010100
2096 address:         a_start       00000000 00000000 00001011 10111000
               .org   a_start
3000 a:               25            00000000 00000000 00000000 00011001
3004                 -10            11111111 11111111 11111111 11110110
3008                  33            00000000 00000000 00000000 00100001
3012                  -5            11111111 11111111 11111111 11111011
3016                   7            00000000 00000000 00000000 00000111
               .end
```

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
      .global sub
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:**

Symbol	Value	Global/ External	Reloc- atable
`sub`	–	External	–
`main`	2048	No	Yes
`x`	2064	No	Yes
`y`	2068	No	Yes

Main Program

Symbol	Value	Global/ External	Reloc- atable
`ONE`	1	No	No
`sub`	2048	Global	Yes

Subroutine Library

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 #3FFFFF, %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:              #FFFFFFFF     ! High 32 bits of -1
                #FFFFFFFF     ! 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:**

```
! 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
```

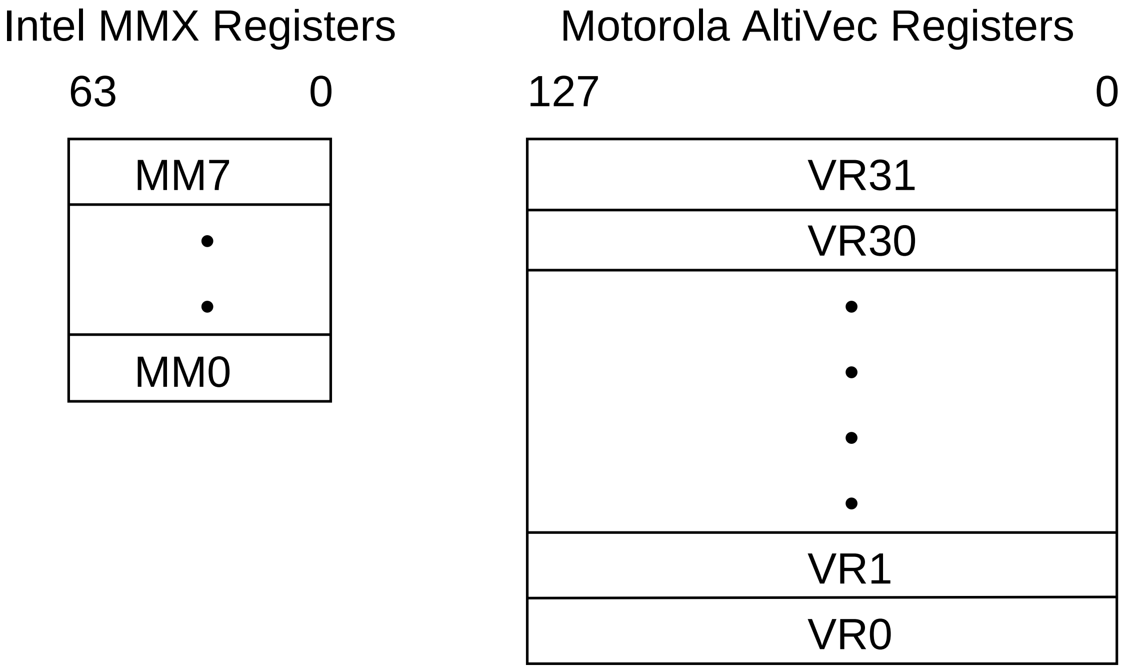
Intel MMX (MultiMedia eXtensions)

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

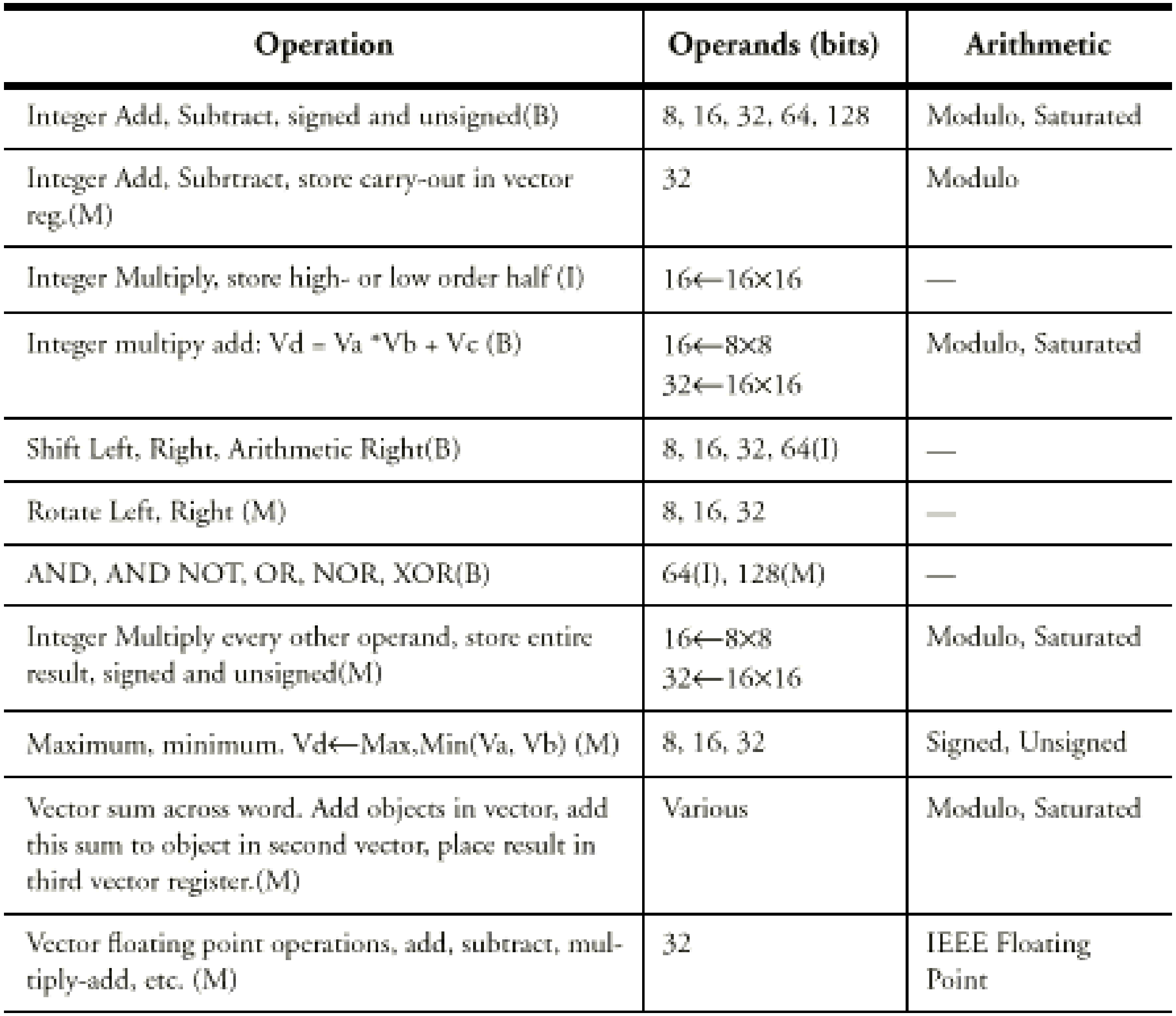
mm0	11111111	00000000	01101001	10111111	00101010	01101010	10101111	10111101
	+	+	+	+	+	+	+	+
mm1	11111110	11111111	00001111	10101010	11111111	00010101	11010101	00101010
	=	=	=	=	=	=	=	=
mm0	11111101	11111111	01111000	01101001	00101001	01111111	10000100	11100111

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.**



MMX and AltiVec Arithmetic Instructions



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

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
	↓	↓	↓	↓	↓	↓	↓	↓
mm0	11111111	00000000	11111111	11111111	00000000	11111111	00000000	00000000
	(T)	**(F)**	**(T)**	**(T)**	**(F)**	**(T)**	**(F)**	**(F)**

Conditional Assignment of an MMX Byte Vector

mm0	11111111	00000000	11111111	11111111	00000000	11111111	00000000	00000000
AND								
mm2	10110011	10001101	01100110	10101010	00101011	01101010	11010101	00101010
	↓	↓	↓	↓	↓	↓	↓	↓
mm2	10110011	00000000	01100110	10101010	00000000	01101010	00000000	00000000