

5

WORKING WITH ASSEMBLY LANGUAGE

In the last chapter the assembly language was presented as being functionally equivalent to machine language, but the translation process from assembly code to machine code was not covered. Likewise, linking and loading of assembled programs was mentioned, but the processes were not detailed. In this chapter the processes of assembly, linking, and loading are covered in detail. Several features commonly found in assemblers are also covered.

5.1 The Assembly Process

The process of translating an assembly language program into a machine language program is referred to as the **assembly process**. The assembly process is straightforward and rather simple, since there is an exact 1-to-1 mapping of assembly language statements to their machine language counterparts. This is in opposition to compilation, for example, in which a given high-level language statement may be translated into any number of computationally equivalent machine language statements.

While assembly is a straightforward process, it is tedious and quite error-prone if done by hand. In fact, the assembler was one of the first software tools developed after the invention of the digital electronic computer. All commercial assemblers provide at least the following capabilities:

- Allow the programmer to specify the run-time location of data values and programs.
- Provide a means for the programmer to initialize data values in memory prior to program execution.
- Provide assembly-language mnemonics for all machine instructions and ad-

dressing modes, and translate valid assembly language statements into their equivalent machine language binary values.

- Permit the use of 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.
- 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.
- Provide for the expansion of **macro routines**, that is, routines that can be defined once, and then instantiated as many times as needed.

We shall illustrate how the assembly process proceeds by using ARC assembly and machine language to “hand assemble” a simple ARC assembly program similar to Figure 4-13, reproduced below for convenience as Figure 5-1. In assem-

```

! 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
      jmp1  %r15 + 4, %r0 ! Return
x:    15
y:    9
z:    0
      .end

```

Figure 5-1 A simple ARC program that adds two numbers

bling this program we use the ARC encoding formats shown in Figure 4-10, reproduced here as Figure 5-2. The figure shows the encoding of ARC machine language. That is, it specifies the target binary machine language of the ARC computer that the assembler must generate from the assembly language text.

Assembly and two pass assemblers

Most assemblers pass over the assembly language text twice, and are referred to as

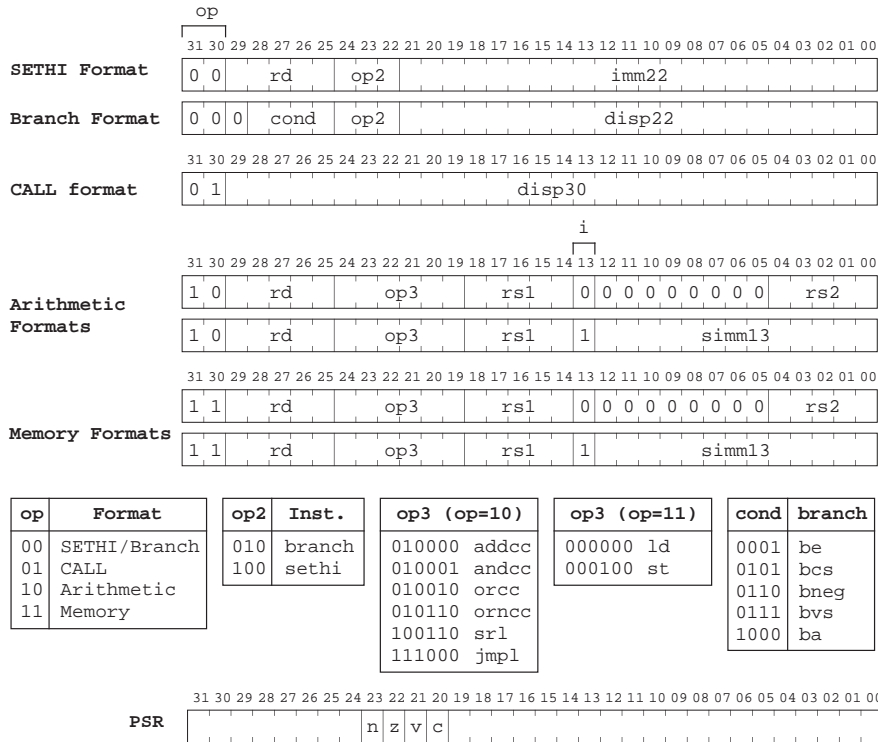


Figure 5-2 Instruction formats and PSR format for the ARC.

two-pass assemblers. The first pass is dedicated to determining the addresses of all data items and machine instructions, and selecting which machine instruction should be produced for each assembly language instruction. During this pass the assembler also performs any assembly-time arithmetic operations, and inserts all labels and constant values into a table, referred to as the **symbol table**. The primary reason for requiring a second pass is to allow symbols to be used in the program before they are defined, which is known as **forward referencing**. Thus the assembler may not know the value of a symbol at the time when it is first used. After the first pass, however, the assembler will have identified and entered all symbols into its symbol table, and, during a second pass it “fixes up” the machine language by inserting the values of unknown symbols.

Let us now assemble the program shown in Figure 5-1 into machine code. Using the Memory format shown in Figure 5-2, we find that the `op` field for the first executable instruction (`ld`) is 11. The destination of an `ld` instruction always goes in the `rd` field, which is 00001 for `%r1` in this case. The `op3` field follows

next, which is 000000 for `ld`. The `rs1` field identifies a register that is added to either the `simml3` field or the register indicated in the `rs2` field, which when added together identify an address for the source operand. For this case, label `x` appears five words after the first instruction, which is at location 2048. Since each word is composed of four bytes, the address of `x` is $5 \times 4 = 20$ bytes after the beginning of the program. The address of `x` is then $2048 + 20 = 2068$ which is represented by the bit pattern 0100000010100. This pattern fits into the signed 13-bit `simml3` field, and so we can use `%r0` in `rs1`.

The first line is thus assembled into the bit pattern shown below:

11	00001	000000	00000	1	0100000010100
<hr/>					
op	rd	op3	rs1	i	simml3

The next instruction is similar in form, and the corresponding bit pattern is:

11	00010	000000	00000	1	0100000011000
<hr/>					
op	rd	op3	rs1	i	simml3

The assembly process continues until all eight lines are assembled, as shown below:

<code>ld [x], %r1</code>	1100	0010	0000	0000	0010	1000	0001	0100
<code>ld [y], %r2</code>	1100	0100	0000	0000	0010	1000	0001	1000
<code>addcc %r1,%r2,%r3</code>	1000	0110	1000	0000	0100	0000	0000	0010
<code>st %r3, [z]</code>	1100	0110	0010	0000	0010	1000	0001	1100
<code>jmp1 %r15+4, %r0</code>	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

The need for two-pass assemblers

As a general approach, the assembly process is carried out by reading assembly language statements sequentially, from first to last, and generating machine code for each statement. As mentioned earlier, a difficulty with this approach is caused by forward referencing.

Consider the program fragment shown in Figure 5-3. When the assembler sees the `call` statement, it does not yet know the location of `sub_r` since the

```

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

```

Figure 5-3 An example of forward referencing.

`sub_r` label has not yet been seen. A solution to this problem is to mark the reference as unresolved, and then go back and fill in the unresolved reference when `sub_r` is defined later in the program. This is the way that a single-pass assembler would operate. An alternative solution is to make an initial pass through the code collecting labels and recording their positions, and then make a second pass for the actual translation. This is the way that a two-pass assembler operates.

The Symbol Table

In the first pass of a two-pass assembly process, a **symbol table** is created. A symbol is either a label or a symbolic name that refers to a value used during the assembly process, such as for a `.equ` statement. As an example of how a two-pass assembler operates, consider assembling the code in Figure 4-14. The symbol table is generated in the first pass of assembly. Starting from the `.begin` statement, the first symbol that is seen that is not an instruction, a pseudo-op, a comment, or a literal is `a_start`. An entry is created in the symbol table for `a_start`, which is given the value 3000. The next symbol that is encountered that has not been seen is `length`. An entry is created in the symbol table for `length`, which is initially assigned no value as shown in Figure 5-4a.

The next symbol that is seen is `address` which is not assigned a value since it has not yet been defined. The next symbol that is seen is `loop`, which is assigned a value of 2060, since the `.org` statement specifies a starting address of 2048 and the `loop` label is three instructions ($3 \times 4 = 12$ bytes) past the beginning of the program. The `.org` statement does not cause any code to be generated, and the location counter is thus unchanged until the first `ld` instruction is encountered. The next symbol that is encountered that is not in the symbol table is `done`, which is entered into the symbol table without a value since it has not yet appeared as a label.

Symbol	Value
a_start	3000
length	—

(a)

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

(b)

Figure 5-4 Symbol table for the ARC program shown in Figure 4-14, (a) after symbols `a_start` and `length` are seen; and (b) after completion.

If any labels are not defined at the end of the first pass, then an error exists in the program and the assembler can flag the errors without needing to continue to the second pass. At this point, the assembler does not yet know that `length`, `address`, and `done` will be defined later in the program.

The first pass of assembly continues, and the unresolved symbols `length`, `address`, and `done` are assigned the values 2092, 2096, and 2088, respectively. The label `a` is encountered, and is entered into the table with a value of 3000. The label `done` appears at location 2088 because there are 10 instructions (40 bytes) between the beginning of the program and `done`. Addresses for the remaining labels are computed in a similar manner.

After the symbol table is created, the second pass of assembly begins. The program is read a second time, starting from the `.begin` statement, but now object code is generated. The first statement that is encountered that causes code to be generated is `ld` at location 2048. The symbol table shows that the address portion of the `ld` instruction is $(2092)_{10}$ for the address of `length`, and so one word of code is generated using the Memory format as shown in Figure 5-5. The second pass continues in this manner until all of the code is translated. The assembled program is shown in Figure 5-5. Notice that the displacements for branch addresses are given in words, rather than in bytes, because the branch instructions multiply the displacements by four.

As mentioned earlier, for an assembler that supports macros, there must be a macro expansion phase that takes place prior to the two-pass assembly process. Macro expansion can also require two passes, in which the first pass records macro definitions, and the second pass generates assembly language statements.

Location counter	Instruction	Object code
	.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:	jmp1 %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	

Figure 5-5 Output from the second pass of the assembler for the ARC program shown in Figure 4-14.

The second pass of macro expansion can be very involved, however, if recursive macro definitions are supported. We will explore macro expansion later in the chapter. A more detailed description of macro expansion can be found in (Donovan, 1972).

One final task that the assembler must perform is to provide information about the starting location for a program. That is, the location that the PC should be initialized to when the program is loaded into memory prior to running. Most often the assembler provides for a special reserved label that the programmer can use to indicate where the program should start execution. In Figure 5-1 the label “main” is a signal to the assembler that execution should start at that location.

5.2 Linking and Loading

It may be the case that we use one file for the main portion of a program and another file for a subroutine library. In this scenario, we are forced to refer to labels in one file from code that exists in another file. A **linkage editor**, or **linker**, is a software program that combines separately assembled programs (called

object modules) into a single program, which is called a **load module**. The linker resolves any references in the symbol table of each object module that are made to symbols defined in the other object modules. The load module can then be loaded into memory by a **loader**, which may also need to modify addresses if the program is loaded at a location that differs from the loading origin used by the linker.

5.2.1 LINKING

A linker needs to distinguish local symbol names (used within a single source module) from global symbol names (used in more than one module). This can be accomplished by making use of the `.global` and `.extern` pseudo-ops. The `.global` pseudo-op instructs the assembler to mark a symbol as being available to other object modules during the linking phase. The `.extern` pseudo-op identifies a label that is used in one module but is defined in another. A `.global` is thus used in the module where a symbol is defined (such as where a subroutine is located) and a `.extern` is used in every other module that refers to it. Note that only address labels can be global or external: it would be meaningless to mark a `.equ` symbol as global or external, since `.equ` is a pseudo-op that is used during the assembly process only, and the assembly process is completed by the time that the linking process begins.

All labels referred to in one program by another, such as subroutine names, should have a line of the form shown below in the source module:

```
.global symbol1, symbol2, ...
```

All other labels are local, which means the same label can be used in more than one source module without risking confusion since local labels are not used after the assembly process finishes. A module that refers to global symbols should declare those symbols using the form:

```
.extern symbol1, symbol2, ...
```

As an example of how `.global` and `.extern` are used, consider the two assembly code source modules shown in Figure 5-6. Each module is separately assembled into an object module, each with its own symbol table as shown in Figure 5-7. The symbol tables now have an additional field that indicates if a symbol is global or external. Program `main` begins at location 2048, and each instruction is four bytes long, so `x` and `y` are at locations 2064 and 2068, respectively. The


```

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

! Subroutine library
    .begin
ONE .equ 1
    .org 2048
    .global sub
sub: ornc   %r3, %r0, %r3
      addcc %r3, ONE, %r3
      jmp   %r15 + 4, %r0
      .end

```

Figure 5-6 A program calls a subroutine that subtracts two integers.

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

Figure 5-7 Symbol tables for the assembly code source modules shown in Figure 5-6.

symbol `sub` is marked as external as a result of the `.extern` pseudo-op.

Unfortunately, subroutine `sub` also has a starting address of 2048. If the two modules are assembled separately, then there is no way for an assembler to know about the conflicting starting addresses during the assembly phase. In order to resolve this problem, the assembler marks symbols that may need to move as a result of changing the loading address for the module as **relocatable**, as shown in the Relocatable fields of the symbol tables shown in Figure 5-7. The idea is that a program that is assembled at a starting address of 2048 can be loaded at address 3000 instead, for instance, as long as all references to relocatable addresses within the program are increased by $3000 - 2048 = 952$. Relocation is performed by the linker so that relocatable addresses are changed by the same amount that the loading origin is changed, but an absolute address (such as the highest possible stack address, which is $2^{31} - 4$ for 32-bit words) stays the same regardless of the loading origin. The linker now has two jobs:

- (1) Modify addresses affected by relocation.

(2) Supply addresses for `.extern` symbols.

The assembler is responsible for determining which labels are relocatable when it builds the symbol table. It has no meaning to call an external label relocatable, since the label is defined in another module, so `sub` has no relocatable entry in the symbol table in Figure 5-7 for program `main`, but it is marked as relocatable in the subroutine library. The assembler must also identify code in the object module that needs to be modified as a result of relocation. The linker then has all of the information it needs to relocate a module.

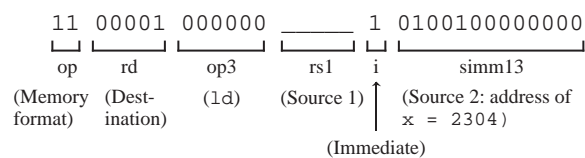
5.2.2 LOADING

The **loader** is a software program that places the load module into main memory. The loader sets the stack pointer `%sp` to its initial value and starts the execution of the load module at the starting address specified by the assembler. If there is only one load module that will execute at any time, then this model will work well. If more than one program is loaded at a time, then the different modules may overlap in memory if the starting addresses used by the linker are not far enough apart. In order to resolve this problem, a loader can add an offset to all of the relocatable code in a module so that an overlap does not occur. This type of loader does not simply repeat the job of the linker: the linker has to combine several object modules into a single load module, whereas the loader simply modifies relocatable addresses within a single load module.

It takes time for a loader to modify all of the references to relocatable addresses. A faster method is to designate a **base register**, which is added into every memory reference. Consider the line of code shown below:

```
ld [x], %r1
```

This line causes the contents of location `x` to be copied into `%r1` at run time. Let us assume that `x` is assigned to memory location 2304. There is one source (`[x]`) and one destination (`%r1`), but there is room for two sources in the Memory format for `ld` as shown below:



We can fill in `rs1` with the pattern 00000, since `%r0` always contains zero. If we designate a base register, say `%r13`, then the `rs1` field of every `ld` and `st` instruction (which are the only instructions that access memory) will have the bit pattern 01101. The source assembly line would then look like:

```
ld %r13, [x], %r1
```

Now, since the base register `%r13` is added to the address of every memory reference, a **relocating loader** only needs to change the base register in order to relocate an entire load module, rather than changing all of the references that are made to relocatable symbols. Note that we have only used the two-operand form of `ld` up to this point, but in its generic form, the first two operands of an `ld` instruction determine the memory location, and the third address identifies the destination register. For `st`, the first operand is the source register and the remaining two operands determine the memory location.

A PROGRAMMING EXAMPLE

Consider the problem of adding two 64-bit numbers using the ARC assembly language. We can store the 64-bit numbers in successive words in memory and then separately add the low and high order words. If a carry is generated from adding the low order words, then the carry is added into the high order word of the result.

Figure 5-8 shows one possible coding. The 64-bit operands `A` and `B` are stored in memory in a high endian format, in which the most significant 32 bits are stored in lower memory addresses than the least significant 32 bits. The program begins by loading the high and low order words of `A` into `%r1` and `%r2`, respectively, and then loading the high and low order words of `B` into `%r3` and `%r4`, respectively. Subroutine `add_64` is called, which adds `A` and `B` and places the high order word of the result in `%r5` and the low order word of the result in `%r6`. The 64-bit result is then stored in `C`, and the program returns.

Subroutine `add_64` starts by adding the low order words. If a carry is not generated, then the high order words are added and the subroutine finishes. If a carry is generated from adding the low order words, then it must be added into the high order word of the result. If a carry is not generated when the high order words are added, then the carry from the low order word of the result is simply added into the high order word of the result and the subroutine finishes. If, how-

```

! 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
        .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
        jmppl %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
        jmppl %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
        jmppl %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

```

Figure 5-8 An ARC program adds two 64-bit integers.

ever, a carry is generated when the high order words are added, then when the carry from the low order word is added into the high order word, the final state of the condition codes will show that there is no carry out of the high order word, which is incorrect. The condition code for the carry is restored by placing a large number in %r7 and then adding it to itself. The condition codes for n, z, and v may not have correct values at this point, however. A complete solution is not detailed here, but in short, the remaining condition codes can be set to their

proper values by repeating the `addcc` just prior to the `%r7` operation, taking into account the fact that the `c` condition code must still be preserved. ■

5.3 Macros

If a stack based calling convention is used, then a number of registers may frequently need to be pushed and popped from the stack during calls and returns. In order to push ARC register `%r15` onto the stack, we need to first decrement the stack pointer (which is in `%r14`) and then copy `%r15` to the memory location pointed to by `%r14` as shown in the code below:

```
addcc %r14, -4, %r14 ! Decrement stack pointer
st    %r15, %r14    ! Push %r15 onto stack
```

A more compact notation for accomplishing this might be:

```
push  %r15          ! Push %r15 onto stack
```

The compact form assigns a new label (`push`) to the sequence of statements that actually carry out the command. The `push` label is referred to as a **macro**, and the process of translating a macro into its assembly language equivalent is referred to as **macro expansion**.

A macro can be created through the use of a **macro definition**, as shown for `push` in Figure 5-9. The macro begins with a `.macro` pseudo-op, and termi-

```
! 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
```

Figure 5-9 A macro definition for `push`.

ates with a `.endmacro` pseudo-op. On the `.macro` line, the first symbol is the name of the macro (`push` here), and the remaining symbols are command line arguments that are used within the macro. There is only one argument for macro `push`, which is `arg1`. This corresponds to `%r15` in the statement “`push %r15`,” or to `%r1` in the statement “`push %r1`,” *etc.* The argument (`%r15` or `%r1`) for each case is said to be “bound” to `arg1` during the assembly process.

Additional formal parameters can be used, separated by commas as in:

```
.macro name arg1, arg2, arg3, ...
```

and the macro is then invoked with the same number of actual parameters:

```
name %r1, %r2, %r3, ...
```

The body of the macro follows the `.macro` pseudo-op. Any commands can follow, including other macros, or even calls to the same macro, which allows for a recursive expansion at assembly time. The parameters that appear in the `.macro` line can replace any text within the macro body, and so they can be used for labels, instructions, or operands.

It should be noted that during macro expansion formal parameters are replaced by actual parameters using a simple textual substitution. Thus one can invoke the `push` macro with either memory or register arguments:

```
push %r1
```

or

```
push foo
```

Obviously the programmer needs to be aware of this feature of macro expansion when the macro is defined, lest the expanded macro contain illegal statements.

Additional pseudo-ops are needed for recursive macro expansion. The `.if` and `.endif` pseudo-ops open and close a conditional assembly section, respectively. If the argument to `.if` is true (at macro expansion time) then the code that follows, up to the corresponding `.endif`, is assembled. If the argument to `.if` is false, then the code between `.if` and `.endif` is ignored by the assembler. The conditional operator for the `.if` pseudo-op can be any member of the set $\{<, =, >, \geq, \neq, \text{ or } \leq\}$.

Figure 5-10 shows a recursive macro definition and its expansion during the assembly process. The argument `x` is tested in the `.if` line. If `x` is greater than 1, then the macro is called again, but with the argument `x - 1`. If the macro `recurs_add` is invoked with an argument of 3, then three lines of code are generated as shown in the bottom of the figure. The first time that `recurs_add` is invoked, `x` has a value of 3. The macro is invoked again with `x = 2` and `x = 1`, at which point the first `addcc` statement is generated. The second and third

! A recursive macro definition		
.macro	recurs_add X	! Start macro definition
.if	X > 1	! Assemble code if X > 0
	recurs_add X - 1	! Recursive call
.endif		! End .if construct
	addcc %r1, %rX, %r1	! Add argument into %r1
.endmacro		! End macro definition
<hr/>		
recurs_add	3	! Invoke the macro
<i>Expands to:</i>		
addcc	%r1, %r1, %r1	
addcc	%r1, %r2, %r1	
addcc	%r1, %r3, %r1	

Figure 5-10 A recursive macro definition, and the corresponding macro expansion.

`addcc` statements are then generated as the recursion unwinds.

Macro expansion is normally performed by a **macro preprocessor** before the program is assembled. The macro expansion process may be invisible to a programmer, however, since it may be invoked by the assembler itself.

5.4 Case Study: Extensions to the Instruction Set – The Intel MMX™ and Motorola AltiVec™ SIMD instructions.

As integrated circuit technology provides ever increasing capacity within the processor, processor vendors search for new ways to use that capacity. One way that both Intel and Motorola capitalized on the additional capacity was to extend their ISAs with new registers and instructions that are specialized for processing streams or blocks of data. Intel provides the MMX extension to their Pentium processors and Motorola provides the AltiVec extension to their PowerPC processors. In this section we will discuss why the extensions are useful, and how the two companies implemented them.

5.4.1 BACKGROUND

The processing of graphics, audio, and communications streams requires that the same repetitive operations be performed on large blocks of data. For example a graphic image may be several megabytes in size, with repetitive operations required on the entire image for filtering, image enhancement, or other processing. So-called streaming audio (audio that is transmitted over a network in real time) may require continuous operation on the stream as it arrives. Likewise 3-D image generation, virtual reality environments, and even computer games require

extraordinary amounts of processing power. In the past the solution adopted by many computer system manufacturers was to include special purpose processors explicitly for handling these kinds of operations.

Although Intel and Motorola took slightly different approaches, the results are quite similar. Both instruction sets were extended with **SIMD** (Single Instruction stream / Multiple Data stream) instructions and data types. The SIMD approach applies the same instruction to a **vector** of data items simultaneously. The term “vector” refers to a collection of data items, usually bytes or words.

Vector processors and processor extensions are by no means a new concept. The earliest **CRAY** and **IBM 370** series computers had vector operations or extensions. In fact these machines had much more powerful vector processing capabilities than these first microprocessor-based offerings from Intel and Motorola. Nevertheless, the Intel and Motorola extensions provide a considerable speedup in the localized, recurring operations for which they were designed. These extensions are covered in more detail farther down, but Figure 5-11 gives a flavor for

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

Figure 5-11 The vector addition of eight bytes by the Intel PADDB mm0, mm1 instruction.

the process. The figure shows the Intel PADDB (Packed Add Bytes) instruction, which performs 8-bit addition on the vector of eight bytes in register MM0 with the vector of eight bytes in register MM1, storing the results in register MM0.

5.4.2 THE BASE ARCHITECTURES

Before we cover the SIMD extensions to the two processors, we will take a look at the base architectures of the two machines. Surprisingly, the two processors could hardly be more different in their ISAs.

The Intel Pentium

Aside from special-purpose registers that are used in operating system-related matters, the Pentium ISA contains eight 32-bit integer registers, with each register having its own “personality.” For example, the Pentium ISA contains a single accumulator (EAX) which holds arithmetic operands and results. The processor also includes eight 80-bit floating-point registers, which, as we will see, also serve

as vector registers for the MMX instructions. The Pentium instruction set would be characterized as **CISC** (Complicated Instruction Set Computer). We will discuss CISC vs. RISC (Reduced Instruction Set Computer) in more detail in Chapter 9, but for now, suffice it to say that the Pentium instructions vary in size from a single byte to 9 bytes in length, and many Pentium instructions accomplish very complicated actions. The Pentium has many addressing modes, and most of its arithmetic instructions allow one operand or the result to be in either memory or a register. Much of the Intel ISA was shaped by the decision to make it binary-compatible with the earliest member of the family, the 8086/8088, introduced in 1978. (The 8086 ISA was itself shaped by Intel's decision to make it assembly-language compatible with the venerable 8-bit 8080, introduced in 1973.)

The Motorola PowerPC

The PowerPC, in contrast, was developed by a consortium of IBM, Motorola and Apple, “from the ground up,” forsaking backward compatibility for the ability to incorporate the latest in RISC technology. The result was an ISA with fewer, simpler instructions, all instructions exactly one 32-bit word wide, 32 32-bit general purpose integer registers and 32 64-bit floating point registers. The ISA employs the “load/store” approach to memory access: memory operands have to be loaded into registers by load and store instructions before they can be used. All other instructions must have their operands and results in registers.

As we shall see below, the primary influence that the core ISAs described above have on the vector operations is in the way they access memory.

5.4.3 VECTOR REGISTERS

Both architectures provide an additional set of dedicated registers in which vector operands and results are stored. Figure 5-12 shows the vector register sets for the two processors. Intel, perhaps for reasons of space, “aliases” their floating point registers as MMX registers. This means that the Pentium's 8 64-bit floating-point registers also do double-duty as MMX registers. This approach has the disadvantage that the registers can be used for only one kind of operation at a time. The register set must be “flushed” with a special instruction, EMMS (Empty MMX State) after executing MMX instructions and before executing floating-point instructions.

Motorola, perhaps because their PowerPC processor occupies less silicon, imple-

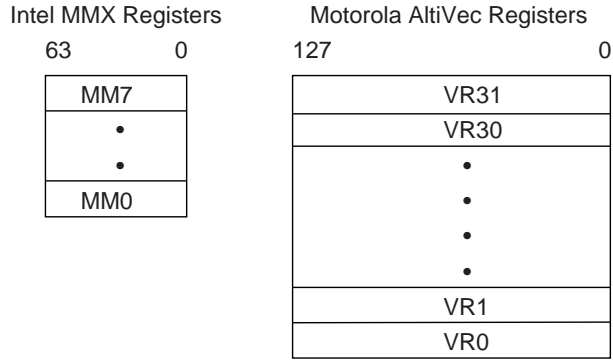


Figure 5-12 Intel and Motorola vector registers.

mented 32 128-bit vector registers as a new set, separate and distinct from their floating-point registers.

Vector operands

Both Intel and Motorola's vector operations can operate on 8- 16- 32- 64- and, in Motorola's case, 128-bit integers. Unlike Intel, which supports only integer vectors, Motorola also supports 32-bit floating point numbers and operations.

Both Intel and Motorola's vector registers can be filled, or packed, with 8- 16- 32- 64- and in the Motorola case, 128-bit data values. For byte operands, this results in 8- or 16-way parallelism, as 8 or 16 bytes are operated on simultaneously. This is how the SIMD nature of the vector operation is expressed: the same operation is performed on all the objects in a given vector register.

Loading to and storing from the vector registers

Intel continues their CISC approach in the way they load operands into their vector registers. There are two instructions for loading and storing values to and from the vector registers, `MOVD` and `MOVQ`, which move 32-bit doublewords and 64-bit quadwords, respectively. (The Intel word is 16-bits in size.) Syntax is:

```
MOVD    mm, mm/m32    ;move doubleword to a vector reg.
MOVD    mm/m32, mm    ;move doubleword from a vector reg.
MOVQ    mm, mm/m64    ;move quadword to a vector reg.
```

```
MOVQ    mm/m64, mm    ;move quadword from a vector reg.
```

- **mm** stands for one of the 8 MM vector registers,
- **mm/mm32** stands for either one of the integer registers, an MM register, or a memory location, and
- **mm/m64** stands for either an MM register or a memory location.

In addition, in the Intel vector arithmetic operations one of the operands can be in memory, as we will see below.

Motorola likewise remained true to their professed RISC philosophy in their load and store operations. The only way to access an operand in memory is through the vector load and store operations. There is no way to move an operand between any of the other internal registers and the vector registers. All operands must be loaded from memory and stored to memory. Typical load opcodes are:

```
lvebx    vD, rA|0, rB    ;load byte to vector reg vD, indexed.
lvehx    vD, rA|0, rB    ;move halfword to vector reg vD indexed.
lvewx    vD, rA|0, rB    ;move word to vector reg vD indexed.
lvx      vD, rA|0, rB    ;move doubleword to vector reg vD.
```

vD stands for one of the 32 vector registers. The memory address of the operand is computed from (rA|0 + rB), where rA and rB represent any two of the integer registers r0-r32, and the “|0” symbol means that the value zero may be substituted for rA. The byte, half word, word, or doubleword is fetched from that address. (PowerPC words are 32 bits in size.)

The term “indexed” in the list above refers to the location where the byte, half-word or word will be stored in the vector register. The least significant bits of the memory address specify the index into the vector register. For example, LSB’s 011 would specify that the byte should be loaded into the third byte of the register. Other bytes in the vector register are undefined.

The store operations work exactly like the load instructions above except that the value from one of the vector registers is stored in memory.

5.4.4 VECTOR ARITHMETIC OPERATIONS

These operations form the heart of the SIMD process. We shall see that there is a new form of arithmetic, saturation arithmetic, and several new and exotic operations.

Saturation arithmetic

Both vector processors provide the option of doing **saturation arithmetic** instead of the more familiar modulo wraparound kind that we studied in Chapters 2 and 3. Saturation arithmetic works exactly like two's complement arithmetic as long as the results do not overflow or underflow. When results do overflow or underflow, in saturation arithmetic the result is held at the maximum or minimum allowable value rather than being allowed to wrap around. For example 2's complement bytes are saturated on the high end at +127 and on the low end at -128. Unsigned bytes are saturated at 255 and 0. If an arithmetic result overflows or underflows these bounds the result is clipped, or "saturated" at the boundary.

The reason for saturation arithmetic can be seen in the processing of color information. If color is represented by a byte in which 0 represents black and 255 represents white, then saturation allows the color to remain pure black or pure white after an operation rather than inverting upon overflow or underflow.

Instruction formats

As the two architectures have different approaches to addressing modes, so their SIMD instruction formats differ. Intel continues their process of using two-address instructions, where the one source operand can be in an MM register, an integer register, or memory, and the second operand and destination is an MM register:

OP mm, mm32or64 ;mm \leftarrow mm OP mm/mm32/64

Motorola requires all operands to be in vector registers, and it employs three-operand instructions:

OP Vd, Va, Vb [,Vc] ; Vd \leftarrow Va OP Vb [OP Vc]

This approach has the advantage that no vector register need be overwritten. In

addition, some instructions can employ a third operand, Vc.

Arithmetic operations

Perhaps not too surprisingly, the MMX and AltiVec instructions are quite similar. Both provide operations on 8- 16- 32- 64- and in the AltiVec case, 128-bit operands. In Table 5.1 below we see examples of the variety of operations pro-

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 multiply 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

Table 5.1 MMX and AltiVec arithmetic instructions.

vided by the two technologies. The primary driving forces for providing these particular operations is a combination of wanting to provide potential users of the technology with operations that they will find needed and useful in their particular application, amount of silicon available for the extension, and the base ISA.

5.4.5 VECTOR COMPARE OPERATIONS

The ordinary paradigm for conditional operations, compare and branch on condition, will not work for vector operations, because each operand undergoing the comparison can yield different results. For example, comparing two word vectors for equality could yield TRUE, FALSE, FALSE, TRUE. There is no good way to employ branches to select different code blocks depending upon the truth or falsity of the comparisons. As a result, vector comparisons in both MMX and AltiVec technologies result in the explicit generation of TRUE or FALSE. In both cases, TRUE is represented by all 1's, and FALSE by all 0's in the destination operand. For example byte comparisons yield FFH or 00H, 16-bit comparisons yield FFFFH or 0000H, and so on for other operands. These values, all 1's or all 0's, can then be used as masks to update values.

Example: comparing two byte vectors for equality

Consider comparing two MMX byte vectors for equality. Figure 5-13 shows the

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)

Figure 5-13 Comparing two MMX byte vectors for equality.

results of the comparison: strings of 1's where the comparison succeeded, and 0's where it failed. This comparison can be used in subsequent operations. Consider the high-level language conditional statement:

```
if (mm0 == mm1) mm2 = mm2 else mm2 = 0;
```

The comparison in Figure 5-13 above yields the mask that can be used to control the byte-wise assignment. Register mm2 is ANDed with the mask in mm0 and the result stored in mm2, as shown in Figure 5-14. By using various combina-

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

Figure 5-14 Conditional assignment of an MMX byte vector.

tions of comparison operations and masks, a full range of conditional operations can be implemented.

Vector permutation operations

The AltiVec ISA also includes a useful instruction that allows the contents of one vector to be permuted, or rearranged, in an arbitrary fashion, and the permuted result stored in another vector register.

5.4.6 SUMMARY

The SIMD extensions to the Pentium and PowerPC processors provide powerful operations that can be used for block data processing. At the present time there are no common compiler extensions for these instructions. As a result, programmers that want to use these extensions must be willing to program in assembly language.

An additional problem is that not all Pentium or PowerPC processors contain the extensions, only specialized versions. While the programmer can test for the presence of the extensions, in their absence the programmer must write a “manual” version of the algorithm. This means providing two sets of code, one that utilizes the extensions, and one that utilizes the base ISA. Whether these extensions will become popular with users has yet to be determined.

■ SUMMARY

A high level programming language like C or Pascal can be used to write programs while treating the low-level architecture of a computer as an abstraction. An assembly language program, on the other hand, takes a form that is very dependent on the underlying architecture. The instruction set architecture (ISA) is made visible to the programmer, who is responsible for handling register usage and subroutine linkage. Some of the complexity of assembly language programming is managed through the use of macros, which differ from subroutines or functions, in that macros generate in-line code at assembly time, whereas subroutines are executed at run time.

A linker combines separately assembled modules into a single load module, which typically involves relocating code. A loader places the load module in memory and

starts the execution of the program. The loader may also need to perform relocation if two or more load modules overlap in memory.

Before compilers were developed, programs were written directly in assembly language. Nowadays, assembly language is not normally used directly since compilers for high-level languages are so prevalent and also produce efficient code, but assembly language is still important for understanding aspects of computer architecture, such as how to link programs that are compiled for different calling conventions, and for exploiting extensions to architectures such as MMX and AltiVec.

■ FURTHER READING

There are a great many references on assembly language programming. (Donovan, 1972) is a classic reference on assemblers, linkers, and loaders. (Gill *et al.*, 1987) covers the 68000. (Goodman and Miller, 1993) serves as a good instructional text, with examples taken from the MIPS architecture. The appendix in (Patterson and Hennessy, 1998) also covers the MIPS architecture. (SPARC, 1992) deals specifically with the definition of the SPARC, and SPARC assembly language.

Donovan, J. J., *Systems Programming*, McGraw-Hill, (1972).

Gill, A., E. Corwin, and A. Logar, *Assembly Language Programming for the 68000*, Prentice-Hall, Englewood Cliffs, New Jersey, (1987).

Goodman, J. and K. Miller, *A Programmer's View of Computer Architecture*, Sounders College Publishing, (1993).

Patterson, D. A. and J. L. Hennessy, *Computer Organization and Design: The Hardware / Software Interface*, 2/e, Morgan Kaufmann Publishers, San Mateo, California, (1998).

SPARC International, Inc., *The SPARC Architecture Manual: Version 8*, Prentice Hall, Englewood Cliffs, New Jersey, (1992).

■ PROBLEMS

5.1 Create a symbol table for the ARC segment shown below using a form

similar to Figure 5-7. Use “U” for any symbols that are undefined.

```

x          .equ  4000
          .org  2048
          ba    main
          .org  2072
main:      sethi x, %r2
          srl   %r2, 10, %r2
lab_4:     st    %r2, [k]
          addcc %r1, -1, %r1
foo:       st    %r1, [k]
          andcc %r1, %r1, %r0
          beq   lab_5
          jmp1  %r15 + 4, %r0
cons:      .dwb  3

```

5.2 Translate the following ARC code into object code. Assume that `x` is at location $(4096)_{10}$.

```

k          .equ  1024
          .
          .
          .
          addcc %r4 + k, %r4
          ld    %r14, %r5
          addcc %r14, -1, %r14
          st    %r5, [x]
          .
          .
          .

```

5.3 Translate subroutine `add_64` shown in Figure 5-8, including variables `A`, `B`, and `C`, into object code.

5.4 A **disassembler** is a software program that reads an object module and recreates the source assembly language module. Given the object code shown below, disassemble the code into ARC assembly language statements. Since there is not enough information in the object code to determine symbol names, choose symbols as you need them from the alphabet, consecutively, from ‘a’ to ‘z.’

```

10000010 10000000 01100000 00000001
10000000 10010001 01000000 00000110
00000010 10000000 00000000 00000011
10001101 00110001 10100000 00001010
00010000 10111111 11111111 11111100
10000001 11000011 11100000 00000100

```

5.5 Given two macros `push` and `pop` as defined below, unnecessary instructions can be inserted into a program if a `push` immediately follows a `pop`. Expand the macro definitions shown below and identify the unnecessary instructions.

```

.begin
.macro push arg1
addcc    %r14, -4, %r14
st      arg1, %r14
.endmacro
.macro pop arg1
ld      %r14, arg1
addcc %r14, 4, %r14
.endmacro
! Start of program
.org 2048
pop %r1
push %r2
.
.
.
.end

```

5.6 Write a macro called `return` that performs the function of the `jmp1` statement as it is used in Figure 5-5.

5.7 In Figure 4-16, the operand `x` for `sethi` is filled in by the assembler, but the statement will not work as intended if $x \geq 2^{22}$ because there are only 22 bits in the `imm22` field of the `sethi` format. In order to place an arbitrary 32-bit address into `%r5` at run time, we can use `sethi` for the upper 22 bits, and then use `addcc` for the lower 10 bits. For this we add two new pseudo-ops: `.high22` and `.low10`, which construct the bit patterns for the high 22 bits and the low 10 bits of the address, respectively. The construct:

```
sethi .high22(0xFFFFFFFF), %r1
```

expands to:

```
sethi #3FFFFFF, %r1
```

and the construct:

```
addcc %r1, .low10(0xFFFFFFFF), %r1
```

expands to:

```
addcc %r1, #3FF, %r1.
```

Rewrite the calling routine in Figure 4-16 using `.high22` and `.low10` so that it works correctly regardless of where `x` is placed in memory.

- 5.8** Assume that you have the subroutine `add_64` shown in Figure 5-8 available to you. Write an ARC routine called `add_128` that adds two 64-bit numbers, making use of `add_64`. The two 128-bit operands are stored in memory locations that begin at `x` and `y`, and the result is stored in the memory location that begins at `z`.
- 5.9** Write a macro called `subcc` that has a usage similar to `andcc`, except that it subtracts its second source operand from the first.
- 5.10** Does ordinary, nonrecursive macro expansion happen at assembly time or at execution time? Does recursive macro expansion happen at assembly time or at execution time?
- 5.11** An assembly language programmer proposes to increase the capability of the `push` macro defined in Figure 5-9 by providing a second argument, `arg2`. The second argument would replace the `addcc %r14, -4, %r14` with `addcc arg2, -4, arg2`. Explain what the programmer is trying to accomplish, and what dangers lurk in this approach.

