Assembler for Macintosh With PowerPC

Contents

Figures, Tables, and Listings

About This Book

This book, *Assembler for Macintosh With PowerPC*, describes how to write assembly-language programs that execute under the Macintosh Operating System. It gives an overview of the assembly-level programming issues relevant to the PowerPC microprocessor. It also discusses the use of the assembler and the structure of an assembly program.

This book helps you prepare source files for assembly using Apple's PowerPC assembler, PPCAsm. You will also find the book useful if you are debugging at the assembly level or if you need to know about the PowerPC processor organization.

You are encouraged to write small assembly-language routines that enhance your C source programs. You might want to write

- stand-alone resources that directly access the PowerPC hardware
- assembly-language optimizations that replace selected C routines with instructions that are not emitted by the C/C++ compiler

You should already be familiar with a Macintosh development environment, high-level languages (for example, C or C++), assembly-language programming, debuggers, and the Macintosh Operating System.

This book contains five chapters. You can read it from start to finish, or you can read Chapters 1 and 2 and then use Chapters 3 through 5 as a reference.

You should read Chapter 1, "PowerPC Assembly Programming," to get an understanding of the PowerPC assembly-language calling conventions and the execution issues you may face when writing assembly-language code. You may also find this chapter useful if you are debugging assembly-level object files.

You should read Chapter 2, "Using the Assembler," for information on how you specify an assembler command line and how your source file is assembled. There are also descriptions of the formats for the assembler's diagnostic messages and assembly source listings.

The remainder of the book is a reference to the assembler. Chapter 3, "Coding Conventions," describes the syntax rules and overall form required for source text processed by the assembler. Chapter 4, "Assembler Directives," describes assembler commands that perform such operations as reserving space or defining data. Chapter 5, "Macros," describes the definition of assembler macros and their use. Appendix A, "Command Line Options," describes the assembler command line options. Appendix B, "Extended Mnemonics," describes the extended instruction mnemonics that the assembler supports.

The glossary, which follows the appendix, provides definitions for terms used in this book. At the end of this book is an index, which will help you locate information about specific topics.

Related Documentation

This book is part of a larger suite of books that contain information essential for developing PowerPC applications and other software. For information about building a PPCAsm application, see *Building Programs for Macintosh With PowerPC*. For information about the PowerPC execution environment, see *Inside Macintosh: PowerPC System Software*. For information about the C and C++ compiler that you can use to compile your source code into a PowerPC application, see *C/C++ Compiler for Macintosh With PowerPC*. For information about debugging and measuring the performance of PowerPC applications, see the *Macintosh Debugger Reference*. For information about performing floating-point calculations in PowerPC applications, see *Inside Macintosh: PowerPC Numerics*.

This book gives only a brief summary of the PowerPC programming model defined in the IBM *PowerPC User Instruction Set Architecture*. You should refer to the IBM document for descriptions of the base user-level instruction set, user-level registers, data types, and addressing modes. For more information on a particular PowerPC microprocessor, you should refer to the appropriate document such as the Motorola *PowerPC 601 RISC Microprocessor User's Manual*.

Conventions Used in This Book

This book uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information, such as parameter blocks, appears in special formats so that you can scan it quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary at the end of this book.

Types of Notes

There are several types of notes used in this book.

Note
A note like this contains information that is interesting but possibly not essential to an understanding of the main text. ◆

IMPORTANT
A note like this contains information that is essential for an understanding of the main text. ▲

Code Examples

All code examples follow the assembler convention that only statement labels and comment lines can begin in column 1 of an assembly language statement. Other assembler statements are indented (for example, machine instructions). For more information, see Chapter 3, "Coding Conventions."

Development Environment

This book shows code examples in assembly language using the Macintosh Programmer's Workshop (MPW). They show methods of using various routines and illustrate techniques for accomplishing particular tasks. However, Apple Computer does not intend that you use exactly these code samples in your application. You can find the location of code listings in the list of figures, tables, and listings.

For More Information

APDA is Apple's worldwide source for hundreds of development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	1-800-282-2732 (United States) 1-800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDAorder
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information on registering signatures, file types, Apple events, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M/S 303-2T
Cupertino, CA 95014-6209

C H A P T E R 1

PowerPC Assembly Programming

Contents

PowerPC Assembly Programming

This chapter describes the PowerPC assembly-language programming conventions for the Macintosh Operating System. It covers the following topics:

- calls between functions
- accesses to other functions and global data
- PowerPC pipeline execution

You should read the sections “Assembly-Language Calling Conventions” and “Accessing Imported Data and Code” if your assembly-language program references code or data in other programs. You should read “PowerPC Execution Issues” if you want to optimize your assembly-language program or if you are debugging at the assembly-language level.

To use this chapter, you should already be familiar with the PowerPC run-time environment described in *Inside Macintosh: PowerPC System Software*. You should also be familiar with the PowerPC instruction mnemonics and addressing modes described in the IBM *PowerPC User Instruction Set Architecture*. For more information on a particular PowerPC processor, refer to an appropriate document (such as the Motorola *PowerPC 601 RISC Microprocessor User’s Manual*).

Assembly-Language Calling Conventions

This section describes the assembly-language version of the calling conventions described in *Inside Macintosh: PowerPC System Software*. It describes the conventions used when your assembly-language program makes direct cross-TOC calls and pointer-based calls to another function. It also describes the conventions used to preserve the stack and nonvolatile registers when your assembly-language function is called by another function. If you need to access code and data in another code fragment, you should read “Accessing Imported Data and Code” on page 1-8.

Calling Function Responsibilities

When calling a function within the same code fragment, the calling function (or *caller*) should include a branch instruction that sets the Link Register (for example, the `bl` instruction). The **Link Register** holds the return address of the currently executing function. This means that the effective address of instruction following the branch instruction is placed in the Link Register. When exiting, the called function should branch to the Link Register (for example, the `blr` instruction). This is shown in the following example:

```
CallToMyProg:    bl    MyProg   ; branch to MyProg & save
                                ; return address
                 ...

MyProg:          ...

                 blr            ; branch to saved return address
```

IMPORTANT

Whenever possible, you should group those functions that call each other in the same code fragment. This grouping will take advantage of PC-relative branches. ▲

A **cross-TOC call** is any function call to a code fragment that could be external to the caller's. A cross-TOC call requires that the Table of Contents Register be changed to the called function's TOC value. The **Table of Contents Register (RTOC)** is a register that points to the table of contents of the code fragment containing the code currently being executed.

The linker inserts system glue code (or *global linkage function*) to handle a *direct cross-TOC call*, a call that accesses the called function by address. The linker also inserts system glue code to handle what is known as a *pointer-based call*, a call in which the function is accessed indirectly through a procedure pointer (for example, a call to a library function whose address is not known).

You identify a direct cross-TOC call with an instruction sequence that has a branch-and-link instruction followed by an instruction that performs nothing— a NOP instruction (pronounced NO-op). The linker inserts system glue code to switch the context from the current code fragment to the new code fragment in the place of the NOP instruction you put in your code. For example, the linker transforms the instruction sequence

```
bl    .xyz
nop              ; assembled as ori 0,0,0
```

to

```
bl    .xyz{GL}
lwz   RTOC, 20(SP)
```

The system glue code for the function `.xyz` is shown in Listing 1-1.

Listing 1-1 System glue code for a direct cross-TOC call

```
lwz   r12, xyz(RTOC) ; load transition vector for xyz
stw   RTOC, 20(RTOC) ; save value of caller’s RTOC
lwz   r0, 0(r12)     ; load address from transition vector
lwz   RTOC, 4(r12)   ; set callee’s RTOC address
mtcr  r0             ; move code address to Count Register
bctr                 ; jump to callee
```

The return from a direct cross-TOC call consists of restoring the value of the caller’s RTOC:

```
lwz   RTOC,20(SP)
```

You identify a pointer-based call as you do a direct cross-TOC call with an instruction sequence that has a branch-and-link instruction followed by a NOP instruction. The linker inserts system glue code to switch the context from the current code fragment to the new code fragment. For example, the linker transforms the following instruction sequence:

```
bl    ._pointer_glue12
nop                        ; assembled as ori 0,0,0
```

to

```
bl    ._pointer_glue12{GL} ; call to the pointer glue
lwz   RTOC, 20(SP)         ; reload caller’s RTOC
```

The system glue code for the pointer glue function is shown in Listing 1-2.

Listing 1-2 System glue code for a call through a procedure pointer

```
lwz   r0, 0(r12)     ; get the code address
stw   RTOC, 20(SP)   ; save caller’s RTOC value
mtcr  r0             ; move code address to Count Register
lwz   RTOC, 4(r12)   ; set callee’s RTOC value
bctr                 ; jump to callee
```

During a function call, the caller uses a common parameter area to pass arguments to the called function. The PowerPC calling conventions specify that parameter lists are passed between functions from left to right. Each parameter is aligned on a word (4 byte) boundary, although alignment within data structures is not affected.

Within the same object file, function names for both the caller and called function should be unique. Likewise, function names should be unique when the caller and called function are in separate object files.

Called Function Responsibilities

The called function (or *callee*) must preserve its stack frame during a function call. To preserve its own stack frame, the called function must duplicate the prolog code usually generated by a compiler. The **prolog code** is a sequence of instructions at the beginning of a called function to save the context of the calling function. The code performs the following steps:

1. Save the contents of the Link Register if necessary.
2. Save the nonvolatile contents of the Condition Register to be used.
3. Save the contents of the nonvolatile floating-point registers to be used.
4. Save the contents of the nonvolatile general-purpose registers to be used.
5. Store the current stack pointer (or back chain) and decrement the stack pointer by the size of the stack frame.

An example of assembly-language prolog code is shown in Listing 1-3.

Listing 1-3 Assembly-language prolog code

```
linkageArea:   set   24       ; run-time architecture dependent
params:        set   32       ; callee parameter area
localVars:     set   0        ; callee local variables
numGPRs:       set   0        ; volatile GPRs used by callee
numFPRs:       set   0        ; volatile FPRs used by callee)
spaceToSave:   set   linkageArea + params + localVars
spaceToSave:   set   spaceToSave + 4*numGPRs + 8*numFPRs

; PROLOG (callee responsibilities)
      mflr  r0,                 ; get Link Register
      stw   r0,8(SP)            ; store Link Register on stack
      stwu  SP, -spaceToSave(SP) ; skip over caller save area
```

function body

The register conventions listed in Table 1-1 through Table 1-3 define the callee’s nonvolatile registers that are saved on the stack during a function call. A **nonvolatile**

register should be restored if it is overwritten during a function call. However, the contents of a **volatile register** may be overwritten without saving during a function call.

Table 1-1 Fixed-point register conventions

Register	Contents	Usage
GPR0	Volatile	Function prolog and epilog
GPR1	Nonvolatile	Stack pointer
GPR2	Nonvolatile	TOC pointer (also known as RTOC)
GPR3–GPR10	Volatile	Arguments passed to a function or returned a value or pointer
GPR11–GPR12	Volatile	Function prolog and epilog
GPR13–GPR31	Nonvolatile	Storage for local variables

Table 1-2 Floating-point register conventions

Register	Contents	Usage
FPR0	Volatile	Scratch area for a local function
FPR1–FPR13	Volatile	Parameters passed to a function or returned a value
FPR14–FPR31	Nonvolatile	Storage for local variables

Table 1-3 Condition Register conventions

Register	Contents	Usage
CR0	Volatile	Scratch area or set by integer instruction record bit
CR1	Volatile	Scratch area or set by floating-point instruction record bit
CR2–CR4	Nonvolatile	Local storage
CR5–CR7	Volatile	Scratch area

To restore its own stack frame, the callee must duplicate the epilog code usually generated by a compiler. The **epilog code** is a sequence of instructions at the end of a called function to restore the context of the calling function. The code performs the following steps:

1. Restore the contents of the nonvolatile general-purpose registers which were overwritten.
2. Restore the stack pointer to the value it had on entry.
3. Restore the contents of the Link Register, if necessary.
4. Restore the nonvolatile contents of the Condition Register.
5. Restore the contents of the nonvolatile floating-point registers.
6. Return to the caller.

You need to restore only the contents of the nonvolatile registers that were overwritten during the execution of the called function (as well as any function that the callee may call). An example of assembly-language epilog code is shown in Listing 1-4.

Listing 1-4 Sample assembly-language epilog code

function body

```
; EPILOG (callee responsibilities)
   lwz        r0,spaceToSave(SP)+8    ; get saved Link Register
   addic      SP,SP,spaceToSave       ; reset stack pointer
   mtlrr0                             ; reset Link Register
```

Accessing Imported Data and Code

This section describes the conventions for identifying assembly-language imports. An *import* is a named object (function or global variable) defined in another code fragment.

You must declare every import in your assembly-language program with an `IMPORT` assembler directive. You also must create an entry for each import in the table of contents (TOC) to make it accessible within a function. The linker and Code Fragment Manager use the TOC to resolve symbol references between code fragments.

A table of contents consists of a TOC anchor that marks the beginning of the TOC followed by the TOC entries. Here is an example.

```
IMPORT   val                   ; import the symbol val
...
TOC                            ; TOC anchor
TC       val[tc], val          ; create a TOC entry for fun
...
```

The general syntax of a TOC entry is

```
TC          name[TC], expression[, expression, . . .]
```

where *name* is the name of the TOC entry and *expression* is a symbol or expression for the TOC entry.

If you want to import a global variable, the TOC entry consists of a 4-byte pointer to that variable. Listing 1-5 shows a sample TOC entry for the global variable `gHelloString`.

Listing 1-5 A TOC entry for a global variable

```
tc gHelloString[TC],gHelloString ; TOC entry for gHelloString
```

If you want to import a function, the TOC entry consists of a 4-byte pointer to a transition vector. A **transition vector** is a data structure in the static data area of a code fragment that usually consists of the address of the imported function and the TOC value for the fragment. Listing 1-6 shows a TOC entry and transition vector for the exported symbol `ClickHandler`.

Listing 1-6 A TOC entry for an imported function

```
tc       ClickHandler[TC],ClickHandler[DS]; TOC entry

csect    ClickHandler[DS]                ; transition vector
dc.l     .ClickHandler[PR]               ; function address
dc.l     TOC[tc0]                        ; TOC address
```

See Chapter 4, “Assembler Directives,” for more information on the `IMPORT` and `TC` assembler directives.

PowerPC Execution Issues

In most instances, the PowerPC microprocessor executes your assembly-language program in the order in which it was stored in memory. However, if you notice delays when your program executes, you should be familiar with the PowerPC instruction pipeline so that you can optimize the performance of your assembly-language program or debug your program at the assembly-language level. The rest of this section describes the PowerPC instruction pipeline and its effects on program execution. It also describes techniques that you can use in your assembly-language program to minimize these effects.

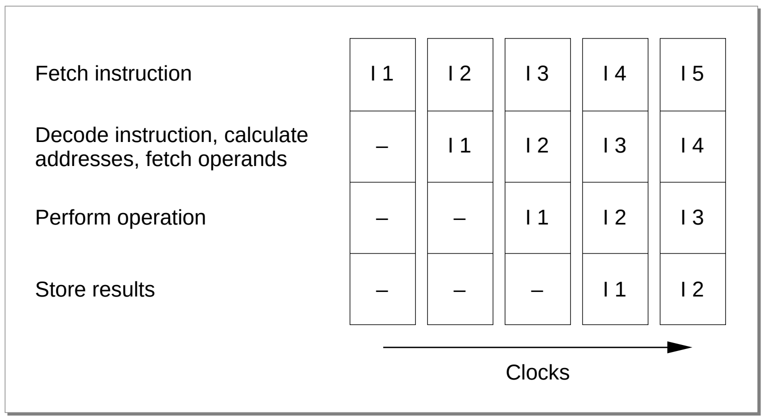
Pipeline Organization

The PowerPC microprocessor does not always wait for an instruction to complete before the next instruction begins. Instead, it uses **pipelining** to divide program execution into a sequence of stages that can process multiple instructions (for example, one branch instruction, one fixed-point instruction, and one floating-point instruction) at the same time. The PowerPC instruction pipeline works in four stages:

1. Fetch instruction
2. Decode instruction (combined with address calculation and operand fetch)
3. Perform operation
4. Store results

A new instruction starts in the pipeline before a previous instruction has completed. This means that several instructions are fetched, decoded, and executed at the same time in a pipeline. Figure 1-1 illustrates the flow of instructions through the PowerPC instruction pipeline.

Figure 1-1 PowerPC instruction pipeline

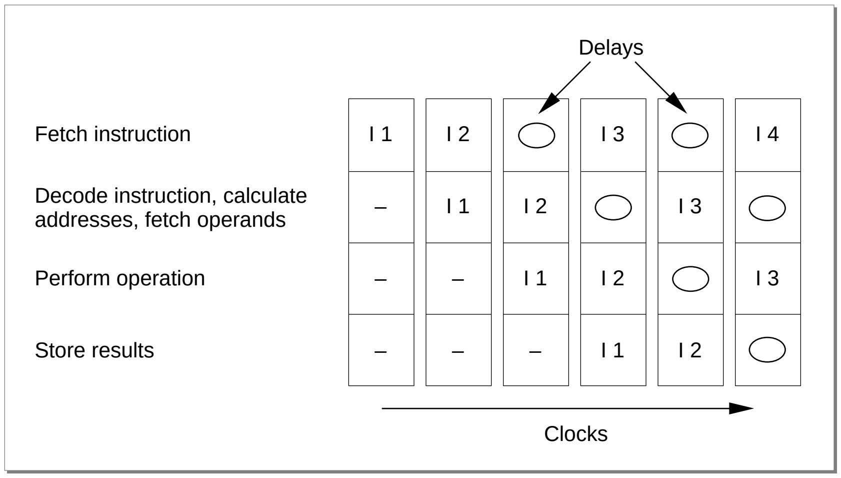


Note that three instructions are being processed concurrently in the third CPU clock cycle: instruction I1 is being executed, instruction I2 is being decoded, and instruction I3 is being fetched.

Pipeline Delays

Pipeline delays occur when the number of instructions that are overlapped is reduced as one or more pipeline stages are idled. Each delay produces a bubble that reduces the number of overlapped instructions in the pipeline. This is because each pipeline stage is idle for one CPU clock cycle as the bubble moves through the instruction pipeline. Figure 1-2 illustrates the effect of bubbles in the PowerPC instruction pipeline.

Figure 1-2 Pipeline delays



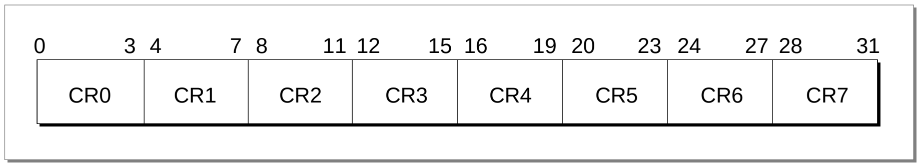
As an example, suppose that instruction I3 depends on operands fetched by instruction I2. This means that instruction I3 cannot be fetched from storage at the same time that instruction I2 fetches its operands. As a consequence, the PowerPC microprocessor idles the instruction fetch stage for one CPU clock cycle while instruction I2 fetches its operands. This bubble moves through the instruction pipeline as the remaining stages are idled for one CPU clock cycle each.

When you write assembly-language programs for the PowerPC microprocessor, you should be aware that *conditional branches* and *shared-resource conflict*s create these bubbles in the instruction pipeline. The PowerPC microprocessor can often detect when bubbles occur, and it attempts to resolve any possible delays in hardware. The rest of this section describes how the PowerPC microprocessor handles branches and shared-resource conflicts to help you debug at the assembly-language level. It also describes techniques you can use to minimize the impact of bubbles in your assembly-language program.

Conditional Branches

A conditional branch is evaluated using the value of any bit in the eight fields of the Condition Register. The PowerPC microprocessor retains the condition codes generated by instructions in the Condition Register. Retaining the code reduces the number of compare instructions in your assembly-language program. Figure 1-3 shows the layout of the Condition Register.

Figure 1-3 Condition Register



You can optimize branch evaluation by setting the Condition Register several instructions ahead. You can set the Condition Register in the following two ways:

- Use the form of a floating-point or fixed-point instruction that sets the record bit. The result code is implicitly stored in one of the first two fields of the Condition Register (field CRO is used for fixed-point results and field CR1 for floating-point results). Here is an example.

```
divw. r4,r5,r6       ; divide word with record bit set
. . .
bc    0x0120,0,2(SP) ; check CRO & branch if bit 2 is false
```

- Create Boolean expressions with any two fields in the Condition Register and have the result placed in a third field for a compare operation. Here is an example.

```
crand cr3,cr5,cr6  ; CR5 && CR6 and result stored in CR3
...
beq   cr3,2(SP)    ; branch if CR3 is equal
```

When the fields of the Condition Register have not been set in time to evaluate the branch, the PowerPC microprocessor uses **branch prediction** to guess the outcome of a conditional branch and fill the instruction pipeline with a sequential stream of instructions. Instructions issued beyond a predicted branch do not complete until the branch is resolved. Any results produced in the instruction pipeline are marked as tentative so that they cannot overwrite any user-accessible registers or memory locations. When a branch prediction is correct, instruction execution continues along the predicted path and the tags on tentative results are removed. When a branch prediction is incorrect, the tentative results are purged and any tentative instructions in progress are canceled.

If you know the likely outcome of a conditional branch, you can add a suffix to any conditional branch instruction mnemonic to set a bit for predicting whether a branch will be taken. When you add a plus sign (+) as a suffix, the branch is predicted to be taken. When you add a minus sign (-) as a suffix, the branch is predicted not to be taken. Here is an example.

```
blt+  2(SP)
```

This instruction branches to the target address 2(SP) if field CRO has the “less than” bit set. The plus sign as a suffix specifies that the branch is predicted to be taken.

For more information on how the PowerPC microprocessor handles conditional branches, see the IBM *PowerPC User Instruction Set Architecture*. For more information on setting the Condition Register for a specific PowerPC microprocessor, see the appropriate document (for example, the Motorola *PowerPC 601 RISC Microprocessor User's Manual*).

Shared-Resource Conflicts

A shared-resource conflict occurs when successive instructions contend for the contents of the same memory location or register. This conflict occurs in the following circumstances:

- An instruction in the pipeline might set a register whose contents must be read by an earlier instruction in the pipeline. The second instruction must not overwrite the contents of the register until the register is free to be modified.
- An instruction in the pipeline might produce a result that is used by another instruction in the pipeline. The latter instruction must be delayed until the result is available.
- Two different instructions in a pipeline might store data to a common memory location, but overlapped execution might reverse the order in which the location is updated.

The PowerPC microprocessor uses **scoreboarding** to detect shared-resource conflicts and to synchronize the instruction pipeline when instructions share data. Hardware tracks the contents of registers to prevent executing subsequent instructions in the instruction pipeline while a register is used as a destination of any instruction. To prevent data from being overwritten inadvertently, an instruction cannot store its results back to the destination register as long as that register is used by a pending instruction. Likewise, to prevent the wrong data from being read inadvertently, an instruction cannot enter the instruction pipeline until its source operands are available from a pending instruction.

However, the performance of your program can degrade during the time that the instruction pipeline cannot overlap multiple instructions. You can avoid shared-resource conflicts by manually inserting delay slots between successive read instructions and store instructions to the same location. These delay slots should be instructions that do not depend on the contents of the shared resource. The following example uses an instruction that resets the Link Register as a delay slot:

```
lwz   r6,2(SP)          ; load the contents of 2(SP) into R6
mtlr  r0                ; delay slot using reset Link Register
addi  r4,r6,1           ; add 1 into R4
```

For information on the instruction timings of a particular PowerPC microprocessor, you should refer to the appropriate document (for example, the Motorola *PowerPC 601 RISC Microprocessor User's Manual*).

C H A P T E R 2

Using the Assembler

Contents

This chapter shows you how to use the PPCAsm assembler. It describes how to construct an assembler command line. It then describes how an executable file is produced. It provides information about the diagnostic messages that the assembler might produce. It also describes the assembly listing options. This chapter is not a complete guide to building a PPCAsm application. See *Building Programs for Macintosh With PowerPC* for more information.

Assembling Files

The PPCAsm assembler is an MPW tool that takes a PowerPC assembly-language source program and produces code for the PowerPC run-time environment. You handle all coding tasks, such as editing source files and entering commands, using standard MPW methods. After you start the MPW Shell application, you can enter a PPCAsm command from any window. A PPCAsm command typically uses the following syntax, which includes the PPCAsm invocation, options, and a source filename:

`PPCAsm [`*option list*`]` *filename*

While there is a file-naming convention to end an assembly source file with a `.s` suffix, the assembler allows any name. After assembly, it creates an object file of the same name but with a `.o` suffix appended. Therefore, a source file `a.s` generates the object file `a.s.o`. Neither filenames nor options are case-sensitive. For example, to assemble a source file, you might enter this command:

```
ppcasm file1.s -o objfile1.o
```

Here, `file1.s` is your source file. The `-o` option specifies that the name of the resulting object file is `objfile1.o`.

Note
If you do not specify a filename on the PPCAsm command line, the assembler accepts standard input as the source file and creates the object file `a.o`. ◆

You can find a complete description of the assembler command line options in Appendix A, “Command Line Options.”

Producing Executable Files

After assembling your source file, the assembler produces an object file in the **Extended Common Object File Format (XCOFF)**. Apple has developed a much more efficient executable format, the **Preferred Executable Format (PEF)**, for the PowerPC run-time environment. Your XCOFF object files are converted to a PEF file during linking. For more information on PEF, see the discussion of the MakePEF tool in *Building Programs for Macintosh With PowerPC*.

Assembler Messages

While assembling your source file, the assembler has several ways of sending progress information. The assembler sets one of the following *status codes* to indicate the result of assembling a source file:

`0`	No errors detected in any of the files assembled
`1`	Parameter or option errors
`2`	Errors detected
`3`	Execution terminated

The assembler returns the status code to the MPW Shell application. The MPW Shell application converts the status code to a string value and assigns it to the MPW Shell variable `{Status}`. You can test this variable and take appropriate action.

The assembler generates *warning messages* to indicate usage in your source file that might require your attention but that does not terminate assembly. The assembler generates *error messages* to indicate error conditions that you must correct to successfully assemble your source file. Warning and error messages are sent to standard error, and they appear in a format similar to this one:

```
### Error 3018 ### Expected ','
    File "fnmacro.s"; Line 22
```

The message format lets you easily read about the error condition and quickly access the troublesome line of code. If an assembly listing is being generated with the `-lo` option, error messages are sent to the listing file. A typical message contains the following information:

- The ID number and description of the error condition.
- The MPW file and the line number where the error condition occurred. By placing your cursor on the end of this line or selecting the line and pressing the Enter key, you can automatically open the specified file as your target window, with the erroneous line of code selected.

Error messages that occur during macro expansion also include the name of the macro that was called and the line number in the macro where the error occurred.

Note
You can use the `-w` option to specify that the assembler suppress warning messages during assembly. ◆

Assembly Listings

You may want an assembly listing to compare your source file with the object code that the assembler produces. You can use the following options when generating an assembly listing:

- To specify a filename for an assembly listing, use the `-lo` option.
- To direct the assembly listing to standard output, use the `-l` option.

The listing format lets you easily read your assembly source file. A typical listing contains the following information:

- The version number and date of the assembler
- The date and time of the current assembly
- The name of the current control section
- The location of the generated object code (seven hexadecimal digits or blank if the corresponding statement does not generate object code)
- The generated object code in hexadecimal format (two hexadecimal words for PowerPC instructions)
- The data defined with `EQU` and `SET` directives (a byte, a word, or two words, depending on the data type)
- The assembly source statement that generated the object code

An assembly listing appears in a format similar to Listing 2-1.

Listing 2-1 An assembly listing

```
PowerPC Assembler - Ver 1.0b (Mar 11 1994)03/14/94 02:48:54 PM
Copyright Apple Computer, Inc. 1992-1993
All rights reserved.
Section  Loc      Object Code Source Statement
                              MACRO
                              MakeCFunction &fnName
                                 export &fnName[DS]
                                 export .&fnName[PR]
                                 toc
                                    tc &fnName[TC], &fnName[DS]
                                 csect &fnName[DS]
                                    dc.l .&fnName[PR]
                                    dc.l TOC[tc0]
                                 csect .&fnName[PR]
                              ENDM
                              MakeCFunction MyFun
                                 export MyFun[DS]
                                 export .MyFun[PR]
                                 toc
MyFun    0000000  0000 0000         tc MyFun[TC], MyFun[DS]
MyFun    0000000                 csect MyFun[DS]
MyFun    0000000  0000 0000         dc.l .MyFun[PR]
MyFun    0000004  0000 0008         dc.l TOC[tc0]
.MyFun   0000000                 csect .MyFun[PR]
```

Whenever the assembly listing is written to standard output using the `-l` option, the assembler places an asterisk (`*`) after the `Object Code` field for instructions that have unresolved forward references. When the listing is directed to a file using the `-lo` option, instructions with forward references are displayed with the correct value in the listing because the forward references are resolved at the end of the assembly.

Coding Conventions

Contents

This chapter describes the syntax rules and overall form required for source text processed by the assembler. You should read this chapter if you need to write an assembly program or read an assembly-level listing. This chapter covers the following topics:

- assembly program structure
- symbols and symbolic expressions
- addresses and address expressions

To use this chapter, you need to be familiar with the PowerPC instruction mnemonics and addressing modes described in the IBM *PowerPC User Instruction Set Architecture*. For more information on a particular PowerPC processor, refer to the appropriate document (such as the Motorola *PowerPC 601 RISC Microprocessor User's Manual*).

Assembly Program Structure

Each significant line of a PowerPC assembly program is either a machine instruction or an assembler directive. A **machine instruction** generates executable code, and an **assembler directive** specifies an operation during assembly. To learn the rules for writing machine instructions, see "Statement Fields" on page 3-4. To learn the rules for writing assembler directives, see Chapter 4, "Assembler Directives."

You can group code or data into a **control section.** Each control section has a storage class assigned by the assembler. The contents of a control section are stored in the object file. Data of a specific type need not be contiguous in the source code, because the assembler groups all data of a common type together when it creates the output file.

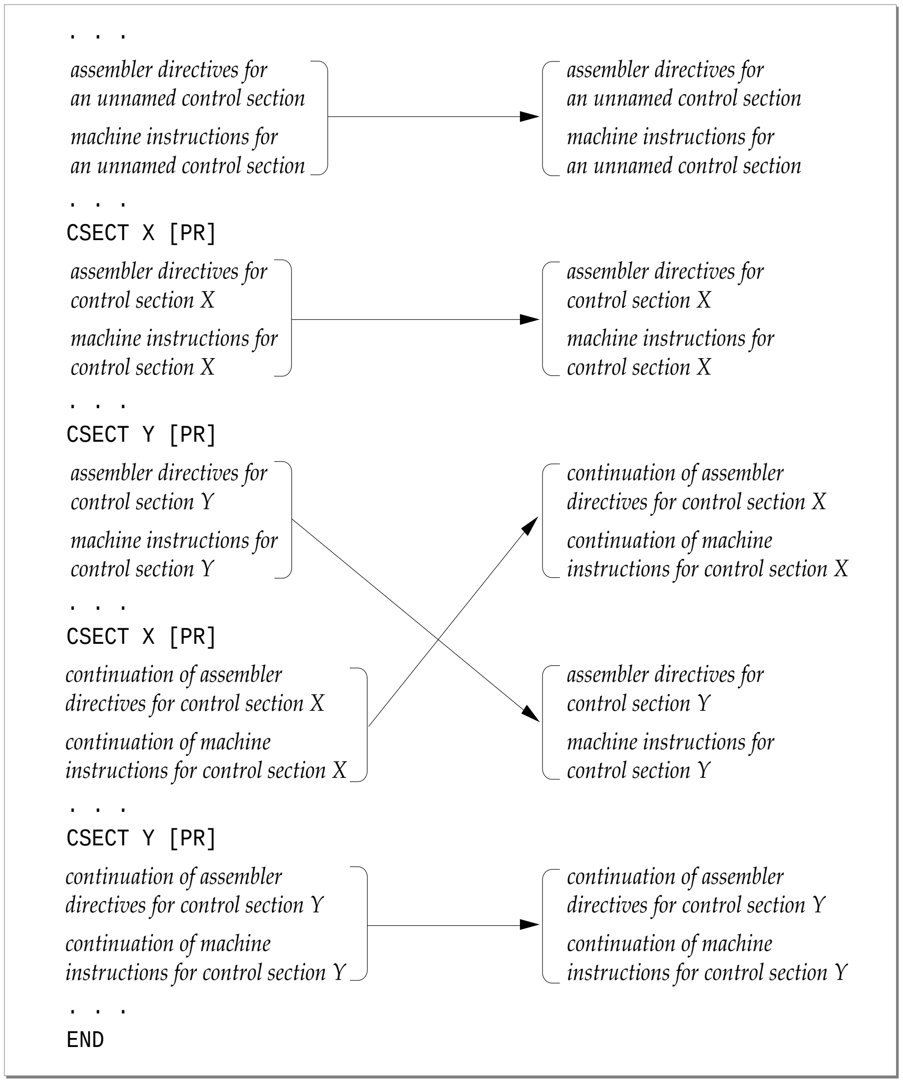
You can use a `CSECT` assembler directive to define a control section to have a given storage class and name. The assembler can also create a control section implicitly when generating instructions or data. By default, the assembler creates an unnamed control section whose storage class is determined by the kind of assembler directive used to create the data.

In addition, you can use a `TC`, `COMM`, or `LCOMM` assembler directive to define a control section of a specific type. The `TC` assembler directive creates table-of-contents entries for references to other code fragments; the `COMM` and `LCOMM` assembler directives create common blocks for storing data. For more information on the specific syntax of these assembler directives, see Chapter 4, "Assembler Directives."

Figure 3-1 illustrates a portion of an assembly program that consists of three control sections: an anonymously named control section and two control sections named X and Y.

The first control section in the left column of Figure 3-1 is unnamed because it does not have an explicit `CSECT` assembler directive. Note that in the left column the contents of control sections X and Y continue where the previous control section with the same name left off. The assembler sorts each named section in the order encountered. The right column of Figure 3-1 shows the reordering for the control sections in the left column.

Figure 3-1 A portion of an assembly program showing control sections

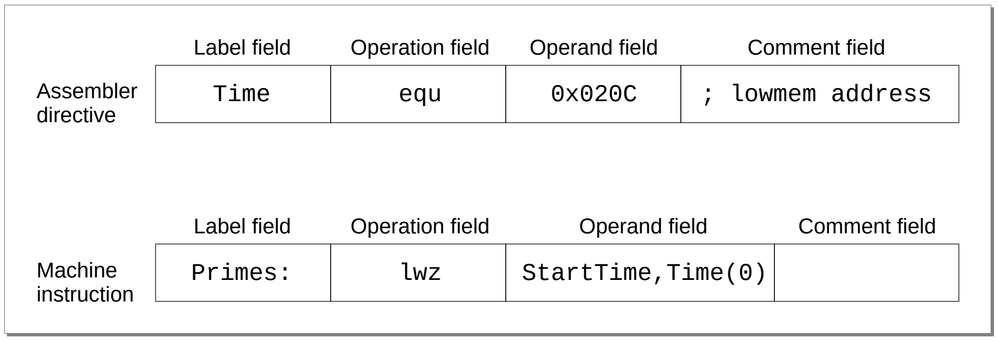

. . .
assembler directives for an unnamed control section
machine instructions for an unnamed control section
. . .
CSECT X [PR]
assembler directives for control section X
machine instructions for control section X
. . .
CSECT Y [PR]
assembler directives for control section Y
machine instructions for control section Y
. . .
CSECT X [PR]
continuation of assembler directives for control section X
continuation of machine instructions for control section X
. . .
CSECT Y [PR]
continuation of assembler directives for control section Y
continuation of machine instructions for control section Y
. . .
END
assembler directives for an unnamed control section
machine instructions for an unnamed control section
assembler directives for control section X
machine instructions for control section X
continuation of assembler directives for control section X
continuation of machine instructions for control section X
assembler directives for control section Y
machine instructions for control section Y
continuation of assembler directives for control section Y
continuation of machine instructions for control section Y

Statement Fields

An assembler directive or machine instruction consists of from one to four fields: *label*, *operation*, *operand*, and *comment*. These fields must be separated by one or more spaces and must be written in the order stated. An assembler directive or machine instruction may be up to 1024 characters long and can consist of one or more lines.

Figure 3-2 has examples of a machine instruction and an assembly directive.

Figure 3-2 An assembly statement



Label Field

The label field is the first field in a machine instruction or assembler directive. It is an optional field that contains an assembler identifier. The label field follows these rules:

- The label field must end with a space or a colon.
- The label identifier must not include a colon.

You do not need a space between a label identifier and a colon (`:`). In addition, you do not need a space before a comment starting with a semicolon (`;`).

In general, the label identifier for machine instructions and assembler directives is assigned the value of the location counter of the first byte of the instruction or data object that is defined by the next operation.

IMPORTANT
The `CSECT` and `TC` assembler directives are exceptions to this description of the value of the label identifier. For more information on these directives, see Chapter 4, "Assembler Directives." ▲

Label identifiers that begin with an at symbol (`@`) are called `@`-labels (pronounced at-labels). All `@`-labels are considered to be *temporary* labels that have a limited scope. The scope of an `@`-label extends through the assembly program, in both directions, to the nearest label that does not begin with `@`. You can redefine an `@`-label, but not in the scope of another instance of the same `@`-label. All `@`-labels defined or used within macros follow the same rules for the label field, but their scope is limited to the body of the macro. In addition, `@`-labels cannot be exported.

Operation Field

The operation field specifies the mnemonic for the desired machine operation, macro call, or assembler directive. The operation field must be preceded by at least one space. Valid operations include

- mnemonics for the POWER instructions, described in the IBM *AIX Version 3.2 for RISC System/6000 Assembler Language Reference*

- mnemonics for the PowerPC instructions, described in the IBM *PowerPC User Instruction Set Architecture* or the appropriate PowerPC processor document (such as the Motorola *PowerPC 601 RISC Microprocessor User's Manual*)
- extended mnemonics for the PowerPC instructions listed in Appendix B, "Extended Mnemonics"
- implementation-dependent mnemonics, described in an appropriate PowerPC processor document (such as the Motorola *PowerPC 601 RISC Microprocessor User's Manual*)
- assembler directives, described in Chapter 4, "Assembler Directives"
- macros, described in Chapter 5, "Macros"

Operand Field

The operand field follows the operation field for machine instructions, assembler directives, and macros. However, not all operations require operands. The operand field is separated from the operation field by at least one space. The operand field can contain subfields separated by commas. The number of subfields is determined by the type of operation.

You can place spaces anywhere in the operand field except within a single symbol. A semicolon (`;`) or number sign (#) is required to separate the operand field from the comment field if a comment is present.

IMPORTANT

If you want to continue a statement on the next line, place a backslash (\) in the operand field before any semicolon or number sign that precedes the comment field. ▲

Comment Field

You can use the comment field to document your program. Comments are ignored by the assembler and may contain any character, including a space.

A comment field is the last field on an assembler statement. Either a semicolon (;) or a number sign (`#`) precedes a comment field.

Comment lines do not have to begin in column 1. It is a comment line if the first significant character of a line is a semicolon or a number sign. Blank lines are ignored by the assembler. You can use them to improve the readability of your assembly program.

Location Counter

The **location counter** assigns control section addresses to machine instructions or data. Because all control sections are considered relocatable, all control sections are assembled with their addresses relative to zero. You can treat each control section as having its own location counter. Therefore, the location counter is a relative offset from the start of the current control section.

You can use two special symbols to refer to the current value of the location counter in a control section: the asterisk (`*`) and dollar sign (`$`). These symbols, which can only appear in the operand field of a machine instruction or assembler directive, represent the location of the first byte of currently available storage after any required boundary alignment. Using a current location symbol in an assembler statement has the effect of placing a label identifier in the label field of the statement and then using that label identifier in the operand field where the current location symbol appears. An exception occurs when the `DC` directive defines more than one constant. In this case, the location counter symbol stands for the location where the current constant is being defined.

Referencing Named Objects

The **scope** of a named object is the area in an assembly program in which the name can be referenced. All named objects in an assembly program are global in scope, except for `@`-labels, macro parameters, and record fields. The scope of a **global name** is anywhere in the source file. A global name can be accessed outside of the named control section. A global name must be unique within the assembly program.

The scoping rules can be overridden with the `EXPORT` and `IMPORT` assembler directives. The general form of these assembler directives is

```
EXPORT    Name_k,Name_l, . . .      ; export declarations
IMPORT    Name_m,Name_n, . . .      ; import declarations
```

You can reference names within an assembly program from another code fragment using exports and imports. An **export** is a named object that can be made accessible to another code fragment (or *exported*). An **import** is a named object that is defined in another code fragment (or *imported*).

You can use the `EXPORT` assembler directive within an assembly program to declare any specified named objects as exported. The exports *Name*$_k$,*Name*$_l$, . . . can be accessed from other files that import those named objects. You can export only code or data label identifiers, control section names, global data definition, storage allocation names, and local common block names.

Once a named object is declared as exported, other files can import the same names by using the `IMPORT` assembler directive. The imports have global scope in the file.

IMPORTANT
The names contained in an `EXPORT` assembler directive are global to the file in which they are declared. Any references to these names from external assembly programs are considered imports. Note that you must explicitly declare imported names with an `IMPORT` assembler directive. ▲

Here is a summary of the program scope rules.

- Named objects with global scope are accessed from the point of first reference or definition to the end of the file. A global name can be
  - any named object in a named or unnamed control section

   - □ any named object imported from another control section
   - □ any qualified field name
- ■ You can use the `EXPORT` assembler directive to reference global names across assembly programs. Exported names can be referenced within a file that imports them.

For more information on the `EXPORT` and `IMPORT` assembler directives, see Chapter 4, "Assembler Directives."

Using Symbols

Except for comments, all fields of a machine instruction or assembler directive consist of symbols. A **symbol** is a character or combination of characters that you use to represent identifiers, numeric constants, character strings, or operators. Furthermore, you can use most symbols as terms in a symbolic expression. The rest of this section describes how you use symbols and symbolic expressions.

Identifiers

An **identifier** (or *name*) is a symbol used to reference a named object. The first character of an identifier must be an uppercase or lowercase letter (`A-Z`, `a-z`), an underscore (_), or an at symbol (`@`). Periods (`.`) can only appear as the first character of an identifier. Subsequent identifier characters can be letters, digits (`0-9`), and underscores (_).

An identifier can be up to 1024 characters long. All characters are significant, and case is not significant for machine instructions or assembler directives. The assembler assigns to exported and imported identifiers the case used in the `EXPORT` or `IMPORT` assembler directive used to define the symbol.

Examples of valid identifiers are

```
BYTE            ApplZone        X
A_1             Start           inverseBit
Next_Char       .funk           NextChar
_trap           a2
```

Numeric Constants

A **numeric constant** is a symbol used to reference a constant value. The assembler expresses numeric constants as decimal, hexadecimal, binary, octal, and floating-point values. Table 3-1 defines the syntax for numeric constants.

Table 3-1 Syntax of numeric constants

Constant type	**Examples**

Decimal numbers formed as a string of decimal digits (`0-9`).	`23`
	`5`
	`32`

Hexadecimal numbers: specified by a dollar sign (`$`) or the character pair "0x" followed by a sequence of hexadecimal digits (`0-9`, `A-F` or `a-f`). "0x" is the digit zero followed by an uppercase or lowercase "x".	`$123`
	`$1A3C`
	`$FFFF`
	`0x342`
	`0xDEADBEEF`
	`$a1C2`
	`0xFFFFFFFF`
	`0x0`

Binary numbers: specified by a percent sign (`%`) followed by a sequence of binary digits (`0-1`).	`%1010`
	`%101`
	`%1011101`

Octal numbers: specified using a leading 0 digit followed by string of octal digits (`0-7`).	`0123`
	`0377`
	`0100`

Floating-point numbers: specified by enclosing a decimal or hexadecimal string with double quotes. Decimal strings are any valid representation as described in the *Inside Macintosh: PowerPC Numerics*. Hexadecimal strings are hexadecimal numbers with a leading dollar sign as described above. In the latter case, it is assumed the hexadecimal value represents the format of a floating-point data type.	`"123"`
	`"-123."`
	`".456"`
	`"-0"`

The assembler treats all decimal, hexadecimal, and binary constants as 32-bit values. For example, the value of $FFFF is 65535 and not –1. If –1 is the desired value, it must be written in decimal notation as –1 or in hexadecimal notation as $FFFFFFFF. The assembler allows floating-point numbers only in the context of the `DC` assembler directives (single, double, or extended).

Strings

A **string** is a symbol that consists of a sequence of one or more ASCII characters (including spaces) enclosed in single quotation marks ('). Within a string, two single quotation marks in succession (`''`) represent one single quotation mark. Some examples of strings are

`'Hello'`	
`'don''t'`	
`''''`	(one single quotation mark)
`'&&x'`	(in macros only)

Strings follow these rules on their length and treatment:

- A string constant used in an arithmetic expression as a numeric value represents only the first four characters in the string (a right-aligned 32-bit value padded on the left with zeros).
- The `STRING` assembler directives let you specify the format of string constants when they appear in data definition assembler directives. For more information on the `STRING` assembler directives, see Chapter 4, "Assembler Directives."

Symbolic Expressions

A **symbolic expression** consists of either a single term or an arithmetic combination of terms. A **term** can consist of the following symbols: an identifier, a constant, or the value of the location counter (* or $). You can use a symbolic expression in the operand field of a machine instruction or assembler directive.

Arithmetic terms are treated as 32-bit signed values and are combined by arithmetic, logical, shift, and relational operators. Table 3-2 lists the operators from lowest precedence to highest; operators of the same precedence are grouped together.

Table 3-2 Order of precedence of operators, lowest to highest

Symbols	Operation
`++` `||`	Logical OR
`--`	Logical exclusive OR
`**` `&&`	logical AND
`|`	bit OR
`^`	bit exclusive OR
`&`	bit AND
`==` `=`	Equal to
`<>` `≠` `!=`	Not equal to
`<`	Less than
`>`	Greater than

continued

Table 3-2 Order of precedence of operators, lowest to highest (continued)

Symbols	Operation
`<=` ≤	Less than or equal to
`>=` ≥	Greater than or equal to
`>>`	Shift right
`<<`	Shift left
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/` ÷	Division
`//` `%`	Modulus division
`~`	One's complement
¬ `!`	Logical NOT
`-`	Unary negation[†]
`( )`	Grouping by parentheses

[†] Unary negation of hexadecimal constants yields the two's-complement negative value of that constant.

The rules for coding symbolic expressions are as follows:

- A symbolic expression may consist of any number of terms.
- Subexpressions may be grouped in parentheses to determine the order of operations when it differs from the order dictated by operator precedence.
- Parentheses may be nested to a maximum depth of 20.
- Only the `+`, `–`, `~`, and `!` operators are allowed at the start of a symbolic expression.
- Only unary negation can appear after an operator if it is followed by a left parenthesis, any digit, or an identifier. Otherwise, a symbolic expression cannot contain two terms or operators in succession.
- Floating-point constants cannot be combined with any operators, because the assembler does not evaluate floating-point expressions.

Here are some examples of properly coded symbolic expressions.

```
*                              * + -*
* + 100                        'a' - 32
Rec.Field + 10                 Alpha + (i > j) * 10
```

```
(a && b) || (c && d)          -64
Rec.Field == 8                2 - -2
(a - b)/(20 + (c - d))        (a-b)+(c-d)
~(x + 10)                     a >> b
A != 10                       ~0x20
(a && b)                      ! (a or b)
'ab' + $8000                  Alpha - Beta
-0x10 + -0x20                 (a ** b) ++ (c ** d)
-10 + x.y                     A * 10
```

Evaluation of Symbolic Expressions

A single symbol represents a single-term expression and takes on the value represented by the symbol (the value associated with the named object, the constant, or the value of the location counter). The assembler reduces a multiterm expression to a single value following these rules:

- Each numeric term is given a 32-bit value (overflows are ignored).
- Every symbolic expression is computed as a 32-bit signed value (limits are ignored).
- Any subexpression enclosed in parentheses is reduced to a single value. The resulting value is then used in computing the final value of the symbolic expression.
- When subexpressions are grouped in nested parentheses, the innermost subexpression is evaluated first.
- Operations are performed from left to right, following the precedence shown in Table 3-2.
- Division always yields an integer result; any fractional part of the result is dropped.
- Division by zero yields 0 as the result.
- The relational operators assign the absolute value 1 when the relation is true and the absolute value 0 when the relation is false. The comparison is algebraic.
- The << and >> operators shift the left operand by the number of bits specified in the right operand. Zeros are shifted into vacated bit positions.

Absolute and Relocatable Expressions

The assembler assigns statement label identifiers an absolute value or a relocatable value. An absolute value can be determined at assembly time, whereas a relocatable value cannot be determined until run time.

An **absolute expression** is a single term or an arithmetic combination of terms that evaluate to an absolute value (such as an `EQU` or `SET` label). All operators are allowed in absolute expressions, subject to the rules just defined in the previous section. Here are some examples.

```
front:   equ   0xFACE
count:   set   4
scratch: equ   r4
fun:     equ   (37-23)/4
funstr:  set   4+3
```

A **relocatable expression** is a term or an arithmetic combination of terms that evaluate to a relocatable value determined by the linker. A relocatable expression can be evaluated only for relocatable values in the same control section. For instance, any statement label identifier within a control section has a relocatable value because the linker is free to change the starting address of a control section. The location counter also has a relocatable value when it is used in a control section.

If you add or subtract an absolute value to a relocatable term, the result is a relocatable value. In addition, you can take the difference of two relocatable values in the same control section, and the result is a fixed displacement that is an absolute value. All other operations on relocatable expressions are not permitted. Here are examples of valid operations on relocatable expressions.

```
datastart:                               ;relocatable expression
            dc.l  val1
            dc.w  val2
dataend:                                 ;relocatable exppression
sizeofdata: equ   $-datastart            ;fixed displacement
            cmpli 1,r13,dataend-datastart ;fixed displacement
```

Forming Addresses

You should already be familiar with the effective addressing modes discussed in the IBM *PowerPC User Instruction Set Architecture*. This section describes the syntax for the following address expressions:

- 32-bit signed absolute
- label-relative
- TOC-relative
- record template

32-Bit Signed Absolute Expressions

You can use 32-bit signed absolute expressions to represent registers, absolute values, and address masks. Where an unsigned 16-bit quantity is required by a machine operation, the expression is still treated as a signed 32-bit quantity and then truncated to the appropriate number of low-order bits.

Predefined Register Names

The assembler supports the names of these general-purpose registers as absolute expressions:

GPR0–GPR31	General-purpose registers
R0–R31	General-purpose registers (same as GPR*n*)
FP0–FP31	Floating-point registers
F0–F31	Floating-point registers (same as FP*n*)

The assembler also supports the specific uses for these general-purpose registers as absolute expressions:

SP	Stack pointer (GPR0)
RTOC	Table of Contents register (GPR2)

These names are case insensitive. Also, any of the above register names may be equated to other name symbols by using the `EQU` or `SET` assembler directives.

The assembler recognizes the names of these special-purpose registers as absolute expressions:

ASR	Address space register
BAT0U–BAT5U	Block address translation register (upper)
BAT0L–BAT5L	Block address translation register (lower)
CTR	Count register
DAR	Data access register
DBATL	Block address translation decrement register (lower)
DBATU	Block address translation decrement register (upper)
DEC	Decrement counter
DSISR	Data storage interrupt status register
EAR	External access register
IBATL	Block address translation increment register (lower)
IBATU	Block address translation increment register (upper)
LR	Link register
RTC	Real-time clock (upper + lower)
RTCD	Real-time clock divisor
RTCI	Real-time clock increment
RTCL	Real-time clock (lower)
RTCU	Real-time clock (upper)
SDR1	Storage description register 1

SPRG0–SPRG3	Software-use special-purpose registers
SRR0–SRR1	Machine status save/restore registers
XER	Fixed-point exception register

Refer to the IBM *PowerPC User Instruction Set Architecture* for more information on these registers.

Label-Relative Address Expressions

You can use label-relative address expressions for operands that specify target label identifiers and displacements from such labels. A label-relative expression makes an implicit reference to the control section that contains the target label definition.

Here are three examples.

```
loopTop
subr+10
funct+(a - b)
```

TOC-Relative Address Expressions

You can use TOC-relative address expressions to specify expressions that are relative to a single TOC (table of contents) entry or to the value of the RTOC (GPR2). A valid expression has the label identifier of a TOC entry created with the `TC` assembler directive. Here is an example that uses a TOC-relative address expression for a *cross-TOC call* between an assembled code fragment and a compiled code fragment.

```
            import   .printf[PR]    ; C print function

            toc
T_strtbl:   tc       StringTable[TC], StringTable[RW]

            csect    StringTable[RW]
            dc.b     'Hello, World'
            csect    PrintStrings[PR]
            lwz      r3, T_strtbl(RTOC)
            bl       .printf[PR]    ; call to printf routine
            nop
            ...
```

See Chapter 1, “PowerPC Assembly Programming,” for more information on cross-TOC calls. For more information on the `TC` assembler directive, see Chapter 4, “Assembler Directives.”

Record Template Address Expressions

Record templates provide a natural way to define data structures. In higher-level languages, data structure definitions can be used as type specifications, because they have a type name associated with a storage layout. For example, the name of a Pascal record type or C data structure can be used to define a data item as having the layout and size defined by the type of the named record. Record names can also be used to define fields of other record structures to create even more complex types.

A record template begins with a `RECORD` directive and ends with an `ENDR` directive. Assembler directives that follow the `RECORD` directive describe the layout of the template, and `DS`, `ORG`, and `ALIGN` directives define the template fields.

`RECORD`

. . .

DS, ORG, ALIGN, EQU, and SET assembler directives . . .

. . .

`ENDR`

Record templates have their own location counter that corresponds to the next available data location in the template. As each data field is defined, the location counter is incremented by the size of that data field. A record template definition does not allocate space, but a reference to a record template in a `DS` assembler directive allocates space for an instance of the record. The field definitions in a record template definition have the effect of `EQU` assembler directives that define the offsets of the fields from the origin of the record.

IMPORTANT

The field definitions in a record template are not retained when the assembler creates the object file. The assembler uses this information only at assembly time. ▲

For more information on assembler directives, see Chapter 4, “Assembler Directives.”

Record Template Field Reference Expressions

You can use a record template field reference expression to reference a field of a record template. Template fields are referenced as qualified names with the form *name.fieldname*, where *name* is a required template name and *fieldname* is the name of the field within the named template. Record template names and field names are considered absolute expressions. The following example defines a number of distinct fields (but no storage is reserved):

```
ControlBlock:  RECORD
code_base:     ds.l          ; addr of base of program code
string_table:  ds.l          ; addr of string table
heap:          ds.l          ; addr of heap
line_count:    ds.w          ; num lines that have been executed
               ENDR
```

However, when an instance of a record is allocated, such as

```
myControlBlock:ds.b     ControlBlock
```

any reference to the name of the instance, or to a field of the instance, is a relocatable expression.

Record Template Offset Expressions

You can use a record template offset expression in the same way as a record template field reference expression, except that no data storage label identifier is specified. Thus the resulting value is always an absolute value representing some offset from the start of the record template.

Here are examples of record template offset expressions.

```
ControlBlock
ControlBlock.code_base
ControlBlock.string_table
ControlBlock.heap+2
```

Record Instance Offset Expressions

You can use a record instance offset expression in the same way as a record template field reference expression, except that no data storage label identifier is specified. Thus the resulting value is always a relocatable value representing some offset from the start of the record template.

Here are examples of record instance offset expressions.

```
MyControlBlock
MyControlBlock.code_base
MyControlBlock.string_table
MyControlBlock.heap+2
```

C H A P T E R 4

Assembler Directives

Contents

This chapter describes the assembler directives. You should read this chapter to understand specific commands to the assembler to perform such operations as reserving space or defining data.

You should have already read Chapter 3, "Coding Conventions," to understand the format of assembly statements. This chapter begins by describing the syntax for assembler directives. The rest of the chapter describes the assembler directives so that you can

- define the structure of an assembly program
- describe the associations of named objects across object files
- provide the assembler information about the assembly process
- control the current location counter
- switch input to another file
- describe the layout of data
- assign a value to a symbol
- define constants
- reserve storage for data
- control assembler listings
- control conditional assembly

Directive Syntax

The description of the assembler directives follows a number of conventions intended to help you understand how to write assembly statements. The syntax for an assembler directive statement follows the general format for assembler statements. Here are some examples.

```
label               operation    operand    comment
MaxDenom:           equ          12         ; maximum denominator
MaxDenomSquared     set          144        ; MaxDenom*MaxDenom
```

The identifier in the *label* field may be shown as required. If the *label* field is not present, then no label identifier may appear on the directive.

The *operation* field specifies which directive to process. It is always required.

The *operand* field can contain operands for the assembler directive. The operand field can contain subfields separated by commas. The number of subfields is determined by the type of directive.

The *comment* field lets you document your directive statement. For assembler directives with all optional parameters, comments may still be specified—even when none of the parameters are given—by using the standard convention of placing a semicolon or number sign in the operand field and then following it by the comment field.

Capitalization

There is no distinction between uppercase and lowercase letters in assembler directives. However, this book uses uppercase letters to indicate symbols that are required. Here are some examples.

`UPPER`	The word "upper" is required.
`W`	The letter "w" is required.
`ON`	The word "on" is required.

Symbolic Parameters

Lowercase letters and words in italics represent symbolic parameters whose values you supply. Symbolic parameters may contain hyphens for readability. The following terms may be used:

abs-expr	An absolute expression
rel-expr	A relocatable expression
expr	An absolute or relocatable expression
string	A string constant
floating	A floating-point constant
name	An assembler name symbol
label	A label identifier
filename	A string constant representing a filename

Items within braces (and separated by vertical bars) represent required items. You can choose only one item. Here are some examples.

`{B | W | L}`
`{`*abs-expr* `|` *rel-expr*`}`

Items within brackets are optional. Here are some examples.

`[`*addr-reg*`]`
`[YES | NO]`
`[.B | .W | .L]`

An ellipsis (. . .) indicates zero or more parameters. An ellipsis that is preceded by a comma (, . . .), indicates an optional list of parameters with each item separated by commas. Here are some examples.

Unlimited number of parameters:	*$param_1$,$param_2$, . . .*
Specific number of parameters:	*$param_1$[,$param_2$, . . . [,$param_5$]]*

An underlined item indicates a default value when you omit an optional parameter. You must refer to the description of each assembler directive to determine how a specific default is applied. Here are some examples.

`[B | `<u>`W`</u>` | L]`
[*abs-expr* **|** <u>*rel-expr*</u>]

You must write the following delimiter characters where specified in a parameter:

`,`	Comma
`( )`	Parentheses
`[ ]`	Brackets that do not refer to optional parameters
`.`	Period
`*`	Asterisk
`=`	Equal sign
`{ }`	Braces

Any occurrence of a bracket, parenthesis, or vertical bar (**|**) as required syntax is shown in bold.

Defining Program Structure

The assembler provides the following directives that you can use to define the structure of an assembly program:

`CSECT`	Start or continue a control section.
`DSECT`	Start or continue a dummy control section.
`TOC`	Define the table of contents.
`TC`	Define a table of contents entry.
`COMM`	Define a named common block.
`LCOMM`	Define a local common block.
`END`	End assembly.

CSECT

You can use the `CSECT` directive to start or continue a control section.

[*label*]   `CSECT` [*name*][**[***class***]**]

label	A label identifier.
name	The name of the control section.
class	The storage class of the control section (see Table 4-1).

DESCRIPTION

A `CSECT` directive groups code or data into named control sections of the specified storage class listed in Table 4-1.

Table 4-1 Storage classes for control sections

Class	Description
`PR`	Read/write control section
`RO`	Read only control section
`DB`	Debug table
`GL`	Cross-TOC glue code
`XO`	Extended opcode
`SV`	SVC
`TB`	Traceback table
`TI`	Traceback index
`TC0`	TOC anchor
`TC`	TOC entry
`RW`	Read-write data
`DS`	Transition vector
`BS`	Zero-initialized data
`UC`	Unnamed Fortran common

A `CSECT` directive is the most general form for defining control sections. The assembler creates a control section with the specified name and storage class if the control section does not exist when the assembler encounters a `CSECT` directive. The value of the location counter for a new control section is zero. Otherwise, the assembler makes the existing control section the current control section when the assembler encounters a `CSECT` directive, and the location counter continues at its previous value.

The label identifier in a `CSECT` directive is assigned the value of the location counter for the first instruction or data item within its scope (rather than the location counter value for the `CSECT` directive).

The assembler treats a control section as unnamed if *name* is omitted from a `CSECT` directive. The assembler treats a control section as read/write (`PR`) if *class-expr* is omitted from a `CSECT` directive.

DSECT

You can use the `DSECT` directive to start or continue a dummy control section.

`DSECT [`*name*`]`

name	The name of the dummy control section.

DESCRIPTION

You can use the `DSECT` directive to store frequently accessed data within the current control section. This means that the linker does not need to resolve symbol references from the current control section to data items in the dummy control section. Instructions and assembler directives that store data in a dummy control section affect its location counter.

IMPORTANT
You cannot use the `DSECT` directive to create a separate control section, and you cannot write to a dummy control section if the current control section is read only. Use the `CSECT` directive with `RW` storage class instead. ▲

TOC

You use the `TOC` assembler directive to define the beginning of the table of contents.

```
TOC
```

DESCRIPTION

The `TOC` directive performs the following actions:

- Determines if it is the first `TOC` directive in the current assembly.
- Creates a control section named TOC of type `TC0` as the TOC anchor if it does not exist.

The linker creates the TOC anchor (`TC0`) as the first entry of the table of contents. You declare table-of-contents entries with the `TC` assembler directive within the scope of the `TOC` assembler directive. This means that the TOC can be continued whenever a `TOC` assembler directive appears.

EXAMPLE

```
toc
tc    fussY[TC], fussY[DS]    ; create a TOC entry for fussY
csect fussY[DS]               ; transition vector of fussY
dc.l  fussY[PR]               ; address of fussY
dc.l  TOC[tc0]                ; TOC address of fussY
```

SEE ALSO

See Chapter 3, "Coding Conventions" for a description of imported and exported symbols, and TOC-relative address expressions. See *Inside Macintosh: PowerPC System Software* for more information on the table of contents and transition vectors.

TC

You can use the `TC` directive to assemble values into a table of contents entry.

[*label*]   `TC`   [*name*]**`[TC],`** *rel-expr*[, . . .]

label	A label identifier.
name	The name of the TOC entry. The storage class for TOC entries is `TC`. You can use *name***`[TC]`** to refer to the TOC entry.
rel-expr	A symbol or expression for the TOC entry.

DESCRIPTION

The `TC` directive collects one or more values defined by *expr* into a TOC entry. The `TC` directive performs the following actions:

- Creates a new control section of type `TC` as *name*. An existing control section of that name and type becomes the current control section. If *name* is omitted, an unnamed TOC control section is created.
- Inserts *expr* as the contents of the TOC entry. If *expr* is imported, the assembler creates a reference between the object or expression and the TOC entry.

The value of a label identifier in a `TC` directive is an offset relative to the TOC anchor (rather than the location counter value of the `TC` directive). You should reference addresses within a TOC entry using a local label identifier and the Table of Contents Register (RTOC).

IMPORTANT

The linker combines TOC entries that contain only one expression. This can occur when more than one TOC entry have the same name and reference the same control section. Because the linker can relocate a TOC entry, you should assign a unique name to each TOC entry that references a different offset within a control section. ▲

EXAMPLE

```
toc                            ; create a table of contents
tc    fussY[TC], fussY[DS]     ; create a TOC entry for fussY
csect fussY[DS]                ; transition vector of fussY
dc.l  fussY[PR]                ; address of fussY
dc.l  TOC[tc0]                 ; TOC address of module for fussY
```

SEE ALSO

See Chapter 3, "Coding Conventions" for a description of imported and exported symbols, and TOC-relative address expressions. See *Inside Macintosh: PowerPC System Software* for more information on the table of contents and transition vectors.

COMM

You use the `COMM` directive to define a block of uninitialized storage called a *common block*.

`COMM` *name*`,`*abs-expr*

name	The name of the common block.
abs-expr	The length of the common block *name* in bytes.

DESCRIPTION

You can use the `COMM` directive to initialize a common block called *name* when the size is known but the block may be initialized in another file. Several files can share a common block. A common block has a storage class of `BS` (uninitialized data). The space for common blocks is created at load time.

EXAMPLE

```
comm proc,5120
```

LCOMM

You can use the `LCOMM` directive to define a local block of uninitialized storage called a *local common block*.

`LCOMM` *abs-expr*`[`*,name*`]`

abs-expr	The length of the local block in bytes.
name	The name of the local block.

DESCRIPTION

You can use the `LCOMM` directive to define a local common block for data that will not be accessed outside of the current file. The space for a local common block is created at run time.

EXAMPLE

```
lcomm 4120,buffer
```

END

You use the `END` directive to terminate the assembly of a file.

```
END
```

DESCRIPTION

Before terminating the assembly of a file, the assembler attempts to resolve all references to `@`-labels. If there are any unresolved references to `@`-labels, the assembler generates an error message. The assembler also makes an attempt to resolve all forward references and any local label identifiers defined within the unnamed sections.

SEE ALSO

If you need more information on assembly program structure or `@`-labels, see Chapter 3, "Coding Conventions."

Symbol Scope Control

The assembler provides directives to associate names across object files. These directives describe the scope of the objects named in the assembler directives.

`EXPORT`	Identify a locally defined name as accessible to other files.
`IMPORT`	Identify an externally defined name as accessible within a file.

EXPORT

You use the `EXPORT` directive to declare a list of locally defined names as accessible to other files.

`EXPORT` *name*`[=>`*string*`], ...`

name	The name of a locally defined symbol made accessible to other files.
string	The external name of a locally defined symbol made accessible to other files.

DESCRIPTION

Any name listed in an `EXPORT` directive is said to be *exported*. You use the `IMPORT` directive to reference a name in an external file that has been exported.

You can use a locally defined name in more than one `EXPORT` directive, but the assembler uses only the last instance. You cannot use an `@`-label in an `EXPORT` directive. You can substitute an external name *string* for a locally defined name when the assembler naming conventions prevent you from using a particular name (usually with special characters not allowed in assembler names). The assembler exports *string* to an external file while *name* is still used within the current file. The substitution occurs as the object file is created.

EXAMPLE

```
EXPORT fun,sun,run
EXPORT fun=>'%fun',sun=>'%sun',run=>'%run'
```

SEE ALSO

For information on assigning values to symbols, see “Symbol Definition” on page 4-18. For more information on symbols and `@`-labels, see Chapter 3, “Coding Conventions.”

IMPORT

You can use the `IMPORT` directive to define a list of externally defined names as accessible within the current file.

`IMPORT` *name*`[<=`*string*`], ...`

name	The name of an externally defined symbol made accessible in the current file.
string	The local name for an externally defined symbol.

DESCRIPTION

Any name listed in an `IMPORT` directive is said to be *imported*. You can substitute a local name for an externally defined *string* that is not a valid assembler identifier. The substitution occurs as the object file is created.

EXAMPLE

```
IMPORT fun,sun,run
IMPORT fun<='%fun',sun<='%sun',run<='%run'
```

Controlling Assembly

The assembler provides three directives to control the assembly process:

`DIALECT`	Specifies a target architecture.
`STRING`	Specifies a string format.
`ALIGNING`	Controls default alignment

DIALECT

You can use the `DIALECT` directive to specify the target architecture.

`DIALECT {POWER | `<u>`POWERPC`</u>`}`

`POWER`	Use the POWER instruction set.
`POWERPC`	Use the PowerPC instruction set.

DESCRIPTION

The assembler accepts only machine instructions of the target architecture named in the `DIALECT` directive. A `DIALECT` directive applies until the end of the file or until another `DIALECT` directive is encountered. The initial setting for the `DIALECT` directive is `POWERPC`.

STRING

You can use the `STRING` directive to define the format for string constants.

```
STRING   {P[String] | C[String] | ASIS[String] | PC[String]}
```

`P[String]`	An MPW Pascal string (the string characters preceded with a length byte that specifies the number of characters that follow).
`C[String]`	A C string (the string characters followed by a null byte).
`ASIS[String]`	A string encoded as the characters between single quotes.
`PC[String]`	A null-terminated string preceded by a length byte.

DESCRIPTION

A `STRING` directive applies until the end of the file or until another `STRING` directive is encountered. The initial setting for the `STRING` directive is `CString`.

Your choice of string format affects the `DC` and `DCB` directives. There may be additional zero bytes after last character to end the string on a word (`DC.W`) or long word (`DC.L`) boundary.

ALIGNING

You can use the `ALIGNING` directive to control default alignment.

```
ALIGNING {on | off}
```

`on`	Default alignment is active.
`off`	Default alignment is not active.

DESCRIPTION

When default alignment is active, the assembler aligns the following instructions and data on boundaries determined by the size of data (for example, double word data is aligned on 8-byte boundaries). The initial setting for the `ALIGNING` directive is `on`.

Setting the Location Counter

The assembler provides directives that control the current location counter.

`ORG`	Set the location counter.
`ALIGN`	Align the location counter.

ORG

You can use the `ORG` directive to change the location counter of a control section or record template to *expr*.

`ORG` *expr*

expr	The new value of the location counter.

DESCRIPTION

The assembler changes the addresses of assembly statements that follow an `ORG` directive to start at the new value of the location counter. The expression *expr* cannot contain any forward, undefined, or imported references. The value of *expr* must be nonnegative. If the size of *expr* is larger than the current size of the control section, the area between the current end of the control section and the new origin is undefined.

ALIGN

You can use the `ALIGN` directive to align the location counter of the current control section or record template to the next multiple of the expression *expr*.

`ALIGN` *abs-expr*

abs-expr	The alignment control value.

DESCRIPTION

The expression *expr* must be zero or positive (interpreted as 2^{expr}, such as 3 is 2^3). It cannot contain any forward, undefined, or imported references. The value of *expr* can be 0 to 12. Common values include

`0`	Byte alignment
`1`	Word alignment
`2`	Long word alignment
`3`	Double word alignment

Controlling Assembler Input

The assembly file control directive `INCLUDE` lets you switch input to another file.

`INCLUDE`	Switch input to another file.

INCLUDE

You can use the `INCLUDE` directive to cause the assembler to switch input to *pathname*.

`INCLUDE` *pathname*

pathname	The directory path for the input file.

DESCRIPTION

The assembler takes input from the specified file until the end-of-file mark is reached. At the end-of-file mark, the assembler continues to take input from the assembly statement line that follows the `INCLUDE` directive. The assembler will not switch back if it reaches an `END` directive that indicates the end of the assembly.

The file specified by *pathname* can have an `INCLUDE` directive for another file. You can nest `INCLUDE` directives up to the maximum number of open files allowed by the development environment.

You can also use the `-i` command line option to search for include files in a pathname you specify. If the include files are not found in the pathname, the following directories are searched if you specified a partial pathname (no colons in the name or a leading colon): the directory containing the current input file, other directories specified in the `-i` option as listed, and the directories specified in the MPW Shell variable `{PPCAsmIncludes}`.

Note
You cannot use an `INCLUDE` directive in a macro definition. ◆

Defining Record Templates

The assembler provides directives that define record templates.

`RECORD`	Defines a record template.
`ENDR`	Ends a record template definition.
`WITH`	Creates a new name scope in which record fields can be referenced without record qualification.
`ENDWITH`	Ends the scope opened by a WITH directive.

RECORD

You can use the `RECORD` directive to start a record template definition.

name `RECORD [{` **{***origin-fieldname***}** `|` *offset-expr*`} [,{ INCR | DECR }]]`

name	The name of the record template.
origin-fieldname	The name of the record template origin.
offset-expr	The initial offset of the template.
`INCR`	Allocates fields at ascending locations.
`DECR`	Allocates fields at descending locations.

DESCRIPTION

The origin of a record template is the field that has a zero offset. You can specify the origin if the first field is not the origin with *origin-fieldname*. Using *origin-fieldname* shifts the origin of the record template from the beginning of the template to the field specified by *origin-fieldname*. The assembler reads in the template definition as if the initial offset was zero and subtracts the relative offset of the origin field from each field offset.

You can specify *offset-expr* as an initial offset into the template. The *offset-expr* is an absolute expression that cannot contain any forward, undefined, or imported references. An offset other than zero has the effect of starting the first field of the record at an offset other than zero.

ENDR

You use the `ENDR` directive to terminate a record template definition.

```
ENDR
```

DESCRIPTION

The `ENDR` directive changes the mode of the assembler to process other assembler directives or machine instructions.

WITH

You use the `WITH` directive to create a new name scope in which record fields can be referenced without record qualification.

`WITH` *name*, . . .

name	The name of a record template.

DESCRIPTION

Within the scope of the `WITH` and `ENDWITH` directives, record fields can be referenced without record qualification. You use the specified list of record names as implied record qualification for record field references. The names may be record template names, or field names of record templates if they are records themselves. The implicit qualification remains in effect until a matching `ENDWITH` is reached. Here is an example.

```
point    record
h        ds.w     1
v        ds.w     1
         endr
         with     point
         .                    ; here h and v can be referenced
         .                    ; without "point" qualification
         .
         endwith
```

You can nest `WITH` directives. The assembler searches for the qualification starting with the innermost `WITH` specification. You can also specify more than one record template or data module name with a single `WITH` directive. When you specify multiple names with a single `WITH` directive, the effect is the same as nesting `WITH` directives, with the last `WITH` parameter being considered the deepest nested `WITH`. For example:

```
WITH     alpha,beta,gamma
...
ENDWITH
```

is equivalent to

```
WITH  alpha
WITH  beta
WITH  gamma
...
ENDWITH
ENDWITH
ENDWITH
```

SEE ALSO

For more information on qualified names, see Chapter 3, "Coding Conventions."

ENDWITH

You use the `ENDWITH` directive to end a `WITH` record qualification scope.

```
ENDWITH
```

DESCRIPTION

The `ENDWITH` directive terminates the innermost (most recent) `WITH` directive.

Symbol Definition

The assembler provides the `EQU` and `SET` directives to define name symbols with a value.

`EQU`	Assign a permanent value to a name symbol.
`SET`	Assign a temporary value to a name symbol.

EQU

You can use the `EQU` directive to permanently assign a value to *name*.

label `EQU` {*expr* | *reg*}

label	A label identifier.
expr	An absolute or relocatable expression.
reg	The name of a register.

DESCRIPTION

You must define *label* without name duplications in the same scope. The operand field may be an arithmetic expression or a register name.

The expression *expr* can be absolute or relocatable, but it must not contain any forward, undefined, or imported references. You can use only integer values.

SET

You can use the `SET` directive to temporarily assign a value to *name*.

label `SET` {*expr* | *reg*}

label	A label identifier.
expr	An absolute or relocatable expression.
reg	The name of a register.

DESCRIPTION

The `SET` directive performs the same function as the `EQU` directive, but *label* may have been previously defined by another `SET` directive. An expression *expr* can be absolute or relocatable, but it must not contain any forward, undefined, or imported references.

Defining Data and Allocating Storage

The assembler provides directives to define constants, initialized data, and to reserve storage for data. You can use the directives for data control sections and record templates.

`DC`	Define a constant.
`DCB`	Define a constant block.
`DS`	Define uninitialized storage.

DC

You can use the `DC` directive to define a constant.

[*label*] `DC[.`*size*`]` {*expr* `|` *flloating* `|` *string*}`,` . . .

label	A label identifier.
size	The size of the data aligned to the next byte (`B`), word (`W`), or long word (`L`) boundary (default is word boundary). For floating-point constants, you can use either long word (`L`) or double word (`D`) boundaries.
expr	An absolute or relocatable expression.
floating	A floating-point constant (enclosed in double quotation marks).
string	A string constant.

DESCRIPTION

The label identifier references the first byte of data after alignment. The *size* parameter determines the size of the data and its alignment.

The operand field contains either an expression *expr*, a floating-point constant *floating*, or a string constant *string* to be defined as constant data. All the values represent one block of ascending bytes.

An expression *expr* can contain forward references and must fit in the specified size. For example, a value of 1000 cannot be declared with a `DC.B` directive. The assembler cannot validate the value of imported references.

A string constant *string* is formatted according to the setting of the current `STRING` directive. The assembler pads space after adding any size bytes or zero termination.

A `DC` directive can be padded after the last value you specify. The assembler pads space before storing data values so that alignment is based on size.

SEE ALSO

For more information on the `STRING` directive, see "Controlling Assembly" on page 4-12. For more information on control sections, see Chapter 3, "Coding Conventions."

DCB

You can use the `DCB` directive to define a constant block.

`[`*label*`]`   `DCB[.`*size*`]`   *length*`,{`*expr* `|` *flloating* `|` *string*`}`

label	A label identifier.
size	The size of the data aligned to the next byte (`B`), word (`W`), or long word (`L`) boundary. For floating-point constants, you can use either long word (`L`) or double word (`D`) boundaries.
length	An absolute expression specifying the storage allocated.
expr	An absolute or relocatable expression.
floating	A floating-point constant (enclosed in double quotation marks).
string	A string constant.

DESCRIPTION

The assembler allocates the number of bytes (`.B`), words (`.W`), long words (`.L`), or double words (`.D`) specified by *length*. The *length* parameter must be an absolute expression without any forward, undefined, or imported references. The value of *length* must be greater than zero.

An expression *expr* can contain forward references and must fit in the specified size. For example, a value of 1000 cannot be declared with a `DCB.B` directive. The assembler cannot validate the value of imported references.

A string constant *string* is formatted according to the setting of the current `STRING` directive. The assembler pads space after adding any size bytes or zero termination.

A `DCB` directive can be padded after the last value you specify. The assembler pads space before storing data values so that alignment is based on size.

SEE ALSO

For more information on the `STRING` directive, see "Controlling Assembly" on page 4-12. For more information on control sections, see Chapter 3, "Coding Conventions."

DS

You can use the `DS` directive to define uninitialized storage.

`[`*label*`]` `DS[.`*size*`]` `{`*length* `|` *recordName*`}`

label	A label identifier.
size	The data alignment to the next byte (`B`), word (`W`), or long word (`L`) boundary.
length	An absolute expression specifying the storage allocated.
recordName	The name of the record that represents the length.

DESCRIPTION

The assembler allocates the number of bytes (`.B`), words (`.W`), or long words (`.L`) specified by *length*. The *length* parameter must be an absolute expression without any forward, undefined, or imported references. The value of *length* is greater than zero.

The *size* parameter determines only alignment. The label identifier references the first byte of the block after alignment.

The storage area allocated by a DS directive can also be specified by a template identifier *recordName* that has been previously defined. The size of the storage area is determined by the size of the template.

SEE ALSO

For more information on allocating space using template definitions, see “Defining Record Templates” on page 4-16.

Controlling Listings

The assembler provides one directive to control assembler listings.

`PRINT`	Turn printing to listing on or off.

PRINT

You can use the `PRINT` directive to control the printing of a listing.

```
PRINT {ON | OFF}
```

DESCRIPTION

If a listing is being generated, then you can use the `PRINT` directive to stop or start printing. The assembler takes no action if a listing is not being generated.

Controlling Conditional Assembly

The `IF`, `ELSE`, and `ENDIF` directives provide the main construct for conditional assembly. These keyword macros must be used in conjunction with one another to form a complete conditional construct.

`IF`	Begin conditional assembly.
`ENDIF`	End conditional assembly.
`ELSE`	Mark optional conditional statements.

IF

You use the `IF` directive to identify the beginning of conditional assembly.

`IF` *bool-expr*
    assembler directives or machine instructions

bool-expr	The Boolean expression to be evaluated.

DESCRIPTION

Each `IF` directive must have a matching `ENDIF` directive. The assembler evaluates the Boolean expression *bool-expr* when an `IF` statement is processed. When the Boolean expression is true, the assembler processes the statements associated with the `IF` directive. When the Boolean expression is false, the assembler skips the statements associated with the `IF` directive. There can be nested `IF` directives. If the outer `IF` directive is false, the inner `IF` directives are ignored.

Boolean expressions in the assembler are defined as a special case of arithmetic expressions. A Boolean expression that yields a zero result is interpreted as false, and a nonzero result is interpreted as true.

SEE ALSO

For more information on expressions, see Chapter 3, "Coding Conventions."

IFDEF

You use the `IFDEF` directive to identify the beginning of conditional assembly when *name* is defined.

`IFDEF` *name*
    assembler directives or machine instructions

name	A previously defined name to be evaluated.

DESCRIPTION

Each `IFDEF` directive must have a matching `ENDIF` directive. The assembler determines whether *name* has not been previously defined. When *name* is previously defined, the assembler processes the statements associated with the `IFDEF` directive. When *name* is not previously defined, the assembler skips the statements associated with the `IFDEF` directive. There can be nested `IFNEF` directives.

IFNDEF

You use the `IFNDEF` directive to identify the beginning of conditional assembly when *name* is not defined.

`IFNDEF` *name*
    assembler directives or machine instructions

name	A previously defined name to be evaluated.

DESCRIPTION

Each `IFNDEF` directive must have a matching `ENDIF` directive. The assembler determines whether *name* has not been previously defined. When *name* is not previously defined, the assembler processes the statements associated with the `IFNDEF` directive. When *name* is previously defined, the assembler skips the statements associated with the `IFNDEF` directive. There can be nested `IFNDEF` directives.

ENDIF

You use the `ENDIF` directive to mark the end of conditional assembly.

```
ENDIF
```

DESCRIPTION

Each type of `IF` directive must have a matching `ENDIF` directive. Any number of `IF` directives can be nested arbitrarily within an `IF`-`ENDIF` construct.

ELSE

You use the `ELSE` directive to mark the beginning of conditional assembly statements to be processed if the Boolean expression for an `IF` directive is false. An `IF` directive does not need an `ELSE` directive.

`ELSE`

assembler directives or machine instructions

DESCRIPTION

The `ELSE` directive is optional in an `IF`-`ENDIF` construct. When the Boolean expression for an `IF` directive is true, the assembler processes the statements associated with the `IF` directive and skips any statements associated with an `ELSE` directive. When the Boolean expression for an `IF` directive is false, the assembler skips the statements associated with the `IF` directive and processes the statements associated with the corresponding `ELSE` directive.

C H A P T E R 5

Macros

Contents

This chapter describes how to define and use macros. A macro is a previously defined sequence of assembly statements that the assembler processes when it encounters a corresponding macro call. You can control which assembly statements are processed by using conditional directives.

You should have already read Chapter 3, "Coding Conventions," to understand the format of assembly statements. This chapter describes how you can

- create a macro definition
- add comments to a macro definition
- use macro parameters in a macro definition
- call macros
- use keyword parameters

About Macros

A **macro definition**, or *macro,* is a sequence of assembly statements that defines the name of a macro, the format of its call, and the assembly statements that are to be processed when the macro is invoked. A macro definition consists of four parts:

- the header directive (`MACRO`)
- the macro prototype statement
- the macro body
- the trailer directive (`ENDM`)

A diagram of macro definition syntax follows.

```
MACRO
name[.]    [parameter-list]
machine instructions or assembly directives
ENDM
```

Macro definitions follow these rules:

- A macro definition must precede any calls to that macro.
- A macro definition can appear within other macro definitions.
- A macro definition that appears within a conditional section of source text (for example, a section controlled by an `IF` directive) is not defined if the assembler skips over the definition when evaluating the condition.

The `MACRO` and `ENDM` directives begin and end a macro definition. These directives take no labels or operands.

The Macro Prototype Statement

The **macro prototype statement** defines the name of the macro and the format of calls to the macro. The prototype statement also defines the names of the macro parameters. The form of the macro prototype statement is

name [*parameter-list*]

The only required field in the prototype statement is the *name* field, which identifies the macro for later calls. The macro name must follow the rules for assembler name symbols (see Chapter 3, “Coding Conventions”). The macro name must not have the same name as a machine instruction, assembler directive, or other macro.

The *parameter-list* field is optional. This field defines the macro parameters for operands passed to a macro and referenced in the macro body. A **macro parameter** consists of an ampersand followed by a valid assembler identifier. Refer to “Macro Parameters” on page 5-5 for more information.

The Macro Body

The **macro body** consists of assembly statements between the macro prototype statement and the macro trailer directive (`ENDM`). The macro body consists of two types of statements:

- model statements that generate machine instructions or assembler directives
- calls to other macros that may also generate assembly statements

The statements in the macro body consist of the label, operation, operand, and comment fields.

- The label field can either be blank or contain a symbol or symbolic parameter. A label in this field gets the value of the location counter at the time of macro expansion rather than macro definition.
- The operation field can contain a machine instruction, assembler directive, or symbolic parameter.
- The operand field follows the rules for operands of the specified operation.
- The comment field can either be blank or contain any characters following the comment character (#). You should note that parameter references in comments are not expanded.

Refer to Chapter 3, “Coding Conventions,” for more information on assembly statement format.

Macro Comments

Comment lines in macros start with either the number sign (#) or the semicolon (`;`). These types of comments appear in the listing of the macro definition and macro expansion. In addition, comment lines that begin with a period and number sign (`.#`) or a period and a semicolon (`.;`) can appear in macro definitions. These comments appear in the macro definition but do not appear when the macro is expanded. The following example demonstrates the two types of macro comment statements:

```
MACRO
MacroExample
.; This comment line will not appear when the macro is expanded.
# This comment appears in the macro definition and expansion.
li r3,1
ENDM
```

Macro Parameters

Macro parameters are a special type of variable that is assigned a value when a macro is invoked. Macro parameters are used to pass information to a macro from the point at which the macro is called.

You define macro parameters in the prototype statement of a macro definition. Any macro parameter referenced in a macro body must be defined in the prototype statement. The assembler assigns a value to a macro parameter when a macro is called. The value of a macro parameter can be modified only by another call to the macro.

Macros have one predefined macro parameter, `&dot`. You can use this parameter to explicitly pass a period (.) in a macro the same way as in a machine instruction. The macro parameter `&dot` has a value of a period or a space, depending on whether the macro name was followed by a period in the macro call.

Listing 5-1 shows the use of macro parameters.

Listing 5-1 A macro parameter before a macro call

```
MACRO
Incr     src,dest          ; get src value, incr, move to dest
addic    &src,&dest,1      ; &src=&dest+1
ENDM
```

The assembler replaces any reference to a macro parameter in the body of a macro with the value from the macro call that corresponds to the specified parameter. The only exception is a macro parameter that appears in a comment field or line. Thus, if the `Incr` macro statement is called with `Alpha` as its `&src` parameter and `Beta` as its `&dest` parameter, the assembler will generate the assembler source statement in Listing 5-2.

Listing 5-2 A generated source statement after a macro call

```
addic Alpha,Beta,1   ; &src=&dest+1
```

The `&src` parameter defined in the `Incr` macro statement in Listing 5-1 appears twice in the `addic` macro statement. When the macro is expanded, the first reference to `&src` in the `addic` macro statement is replaced with the parameter value `Alpha`. The second instance of `&src` is not replaced, because it appears in the comment field.

You can give a macro parameter a default value by following its name in the prototype statement with an equal sign (=) and the default value. The assembler assigns the default value to the macro parameter if the corresponding actual macro operand is omitted in a macro call. Here is an example.

```
MACRO
BumpSz  &Sz=4, &reg=r3
. . .
ENDM
```

You can concatenate macro parameters to other characters or symbols without any intervening spaces. However, characters or numbers concatenated to the right of a macro parameter reference can be interpreted as part of the parameter name. You should therefore terminate the parameter reference with a period (`.`) to distinguish the end of the parameter name from the characters concatenated to its right.

If a macro parameter is followed by a period, the parameter and the period are replaced with the value of the parameter when the macro is expanded. The period does not appear in the generated statement.

Table 5-1 shows what happens when the macro parameter `&Param` is concatenated with various combinations of text, other parameters, and special characters.

Table 5-1 An example of macro parameter concatenation

Concatenation	Macro expansion
`&Param.B`	AB
`&Param..B`	`A.B`
`&Param.(B)`	`A(B)`
`B&Param`	BA

Table 5-1 An example of macro parameter concatenation (continued)

Concatenation	**Macro expansion**
`B,&Param`	`B,A`
`B2&Param`	`B2A`
`&Param.2B`	`A2B`
`&Param,.2B`	`A,.2B`
`&Param&Param`	`AA`
`&Param.&Param`	`AA`
`&Param..&Param`	`A.A`

NOTE The column on the right show shows the result from macro expansion if the value of `&Param` is A.

Macro Calls

A **macro call** is an assembly statement that invokes a macro definition. The assembler expands the macro definition to generate assembly statements that replace the macro call statement.

The format of a macro call is

[*label*] *macro-name*[.] [*parameter-list*] [*comment*]

The *label* field of the macro call can contain a symbolic name. Any label appearing on a macro call will be defined on the first instruction or data item generated.

The operation field contains the name of the macro to be invoked. The macro must be defined before it is called.

The operand field can contain a parameter list separated by commas. The order of the macro parameters in the prototype statement determines the order of the operands in a macro call statement.

The macro parameters defined in a macro definition are called *formal parameters*, and the corresponding operands specified in a macro call are called *actual parameters*. The actual parameters represent the values that are assigned to the formal parameters when a macro is called.

A macro call can have a comment field. Either a semicolon (`;`) or number sign (#) begins a comment field.

Refer to Chapter 3, "Coding Conventions," for more information on the format of assembly statements.

Operand Syntax

A macro operand can contain an arbitrary string of characters, as long as the following rules are followed:

- A comma ends one macro operand and begins the next macro operand.
- An `@`-label cannot be passed as a parameter to a macro.

Omitted Operands

If you do not specify a macro operand for the corresponding formal parameter, the assembler still expects the comma that separated the omitted operand from any operand that follows. This comma preserves the relative positions of each operand. If you omit one or more of the last operands from a macro call, you can omit the commas that separated the omitted operands. Any macro operand that you omit from a macro call statement has the null string as a value. Here is an example.

```
ExMacro  &parm1,&parm2,&parm3,&parm4,&parm5  ; prototype statement
ExMacro  table,*+6,,'syntax error'           ; macro call
```

In the macro call, the third parameter has been omitted (the null string between the second and third commas), as has been the fifth parameter (no final comma required).

The assembler replaces parameter references with a null string any time it encounters a reference to an omitted operand. However, you can give a parameter a default value in the prototype statement. The assembler assigns this default value to the parameter if the operand is omitted in a macro call.

Nested Macro Calls

A **nested macro call** is a macro call from the body of another macro definition. A macro call that is not made from the body of a macro definition is called an **outer macro call.**

The assembler suspends processing of the current macro when it encounters a nested macro call. The assembler expands the nested macro call before it resumes processing the suspended macro. The **macro call chain** represents the macros that are suspended while the assembler expands the current macro definition.

The macro definition invoked by an outer macro call can contain many levels of nested macro calls. A particular nested macro can be called at different times at different macro call depths. The number of levels of nested macros is limited and can be controlled by the `-depth` command line option (see Appendix A, “Command Line Options,” for more information).

Listing 5-3 shows how macros are called from within other macros. In it, the macro `Incr2` calls the macro `Incr` macro defined in Listing 5-2 on page 5-6.

Listing 5-3 A nested macro

```
MACRO
Incr2    &a,&b,&c
.; increment &a and &b, put sum in &c
Incr     &a,&a
Incr     &b,&b
add      &c,&a,&b
ENDM
```

If the macro `Incr2` is called with `X`, `Y`, and `Z` as its values, the assembler generates the following assembler statements from expanding the macro statement `Incr2 X,Y,Z`:

```
addic    X,X,1
addic    Y,Y,1
add      Z,X,Y
```

The assembler generates `addic X,X,1` with the first inner macro call to `Incr`. The assembler generates `addic Y,Y,1` with the second inner macro call. The assembler generates `add Z,X,Y` with the outer macro call to `Incr2`.

Macro Call Termination

You can specify the `EXITM` directive within a macro body to stop expanding the current macro definition before reaching the end. After it encounters an `EXITM` directive, the assembler processes the statement that follows the macro call that invoked the terminated macro definition. If the terminated macro call was made from the body of another macro definition, the assembler resumes processing the next macro call in the macro call chain.

IMPORTANT
The `EXITM` assembler directive terminates only the current macro definition. Do not confuse the `EXITM` directive with the `ENDM` directive that is required as the last statement of every macro definition. ▲

Keyword Parameters

Keyword parameters provide an alternative to the positional parameter mechanism. A **keyword parameter** is a macro parameter that is identified in a macro call statement by its name rather than its position. With keyword parameters, you can write macros that have parameters that need not appear in any particular order, with default values when the parameter is not specified. When calling a macro with keyword parameters, you usually specify only the parameters that require values other than the defaults.

Keyword Parameter Definition

The prototype statement for a macro with keyword parameters defines the name of the macro, the format of calls to the macro, and the names of the keyword parameters. The form of the macro prototype statement with keyword parameters is

name `[`*keyword*`==[`*value*`]],...`

A keyword parameter consists of an ampersand (&) followed by a valid assembler identifier. Each keyword parameter is followed two equal signs (==). If present, the default value is a character string that is substituted for the parameter in the body of the macro. Here are some examples.

```
&file==fun.text
&recsize==1024
```

Here is an example of a keyword prototype definition with keyword parameters.

```
CopyBuf     &src==InBuf,&dest==,&count==512
```

Keyword Macro Calls

The format of a keyword macro call is

[label] *name*`[.]` `[`*keyword*`=[`*value*`]],...`

The actual keyword operands do not have an ampersand (&) prefix on the keyword parameter name; they are followed by a single equal sign and the actual parameter value. Here are some examples of actual keyword specifications.

```
RecSize=1024
dest=printBufr
Count=
```

You should follow these rules when writing keyword macro calls:

- Actual keyword operands can be written in any order.
- Specify only keyword operands whose values differ from the default values in the macro prototype statement.
- Do not specify extra commas for omitted operands.

Appendixes

Command Line Options

This appendix describes the command line options you can specify to guide the operation of the assembler. Options begin with a minus sign to distinguish them from filenames and other options. For letter or keyword options, the case is ignored. Options may appear in any order. For options that take a parameter, there must be least one space or tab between the option letter and the parameter.

```
ppcasm [ -auxheader modtype ][ -blocksize size ] [ -c[heck] ]
[ -d[efine] name[=value] ] [ -depth[=value] ]
[ -dialect Power | PowerPC ][ -i pathname  ]
[ -l ] [ -lo listingName ] [ -o objName ] [ -p ] [ -t ] [ -w ] file
```

Table A-1 describes each command line option and any default values.

Table A-1 Assembler command line options

Option	**Meaning**
`-auxheader` *modtype*	Generate an auxiliary XCOFF header with a module of type *modtype.* Values for *modtype* are `1R` (single use) `RE` (reusable) `RO` (read only) Default: Do *not* generate an auxiliary header
`-blocksize` *size*	Specifies the input/output buffer size in bytes to *size.* Default: 512
`-check`	Check for syntax errors and do not generate an object file.
`-d[efine]` *name*[=*value*]	Assigns a decimal integer value to *name.* This option is equivalent to placing the directive *name* `SET` *value* at the beginning of the source file. You can define more than one name by specifying either multiple `-d` options or multiple *name* assignments (separated by commas). Default value: 1
`-depth` *value*	Changes the maximum permitted depth of the macro call stack to *value.* Default: 64
`-dialect` *type*	Accepts the POWER or PowerPC instruction mnemonics. Values for *type* are `Power` (POWER instructions) `PowerPC` (PowerPC instructions) Default: `PowerPC`

continued

Table A-1 Assembler command line options (continued)

Option	**Meaning**
`-i` *pathname*	Search for include files in *pathname.* If the include files are not found in *pathname*, the following directories are searched if you specified a partial pathname (no colons in the name or a leading colon): the directory containing the current input file; other directories specified in the `-i` option as listed; the directories specified in the MPW Shell variable `{PPCAsmIncludes}`.
`-l`	Write the assembly listing to standard output (unless `-lo` is specified).
`-lo` *listingName*	Write the assembly listing to *listingName.*
`-o` *objName*	Write the object file to *objName.*
`-p`	Write version, progress, and summary information to the diagnostic output file.
`-t`	Display only the assembly time and number of lines to the diagnostic output file even if progress information (`-p` option) is not being displayed.
`-w`	Suppress warning messages.

Extended Mnemonics

PPCAsm supports the extended mnemonics described in the IBM *PowerPC User Instruction Set Architecture*. This appendix lists the extended mnemonics that are supported in the following categories:

- simplified branches
- branches that incorporate conditions
- access to and from special-purpose registers
- traps
- other extended mnemonics

For more information on extended mnemonics, see the IBM *PowerPC User Instruction Set Architecture* or the appropriate PowerPC processor document (such as the Motorola *PowerPC 601 RISC Microprocessor User's Manual*).

Simplified Branches

This section lists the branch extended mnemonics that already specify a branch condition. If you know the likely outcome of a branch condition, you can add a suffix to a branch extended mnemonic to set a bit for predicting whether a branch will be taken. When you add a plus sign (+) as a suffix, the branch is predicted to be taken. When you add a minus sign (-) as a suffix, the branch is predicted not to be taken.

Extended mnemonic	**Operation**	**Base mnemonic equivalent**
bctr	Branch Unconditionally to CTR	bcctr 20,0
bctrl	Branch Unconditionally to CTR, set LR	bcctrl 20,0
bdnz addr	Decrement CTR, Branch if CTR Non-zero to Relative addr	bc 16,0,addr
bdnza addr	Decrement CTR, Branch if CTR Non-zero to Absolute addr	bca 16,0,addr
bdnzf BI,addr	Decrement CTR, Branch if CTR Non-zero and Condition False to Relative addr	bc 0,BI,addr
bdnzfa BI,addr	Decrement CTR, Branch if CTR Non-zero and Condition False to Absolute addr	bca 0,BI,addr
bdnzfl BI,addr	Decrement CTR, Branch if CTR Non-zero and Condition False to Relative addr, set LR	bcl 0,BI,addr
bdnzfla BI,addr	Decrement CTR, Branch if CTR Non-zero and Condition False to Absolute addr, set LR	bcla 0,BI,addr

Extended mnemonic	Operation	Base mnemonic equivalent
bdnzflr BI	Decrement CTR, Branch if CTR Non-zero and Condition False to LR	bclrl 2,BI
bdnzflrl BI	Decrement CTR, Branch if CTR Non-zero and Condition False to LR, set LR	bclrl 0,BI
bdnzl addr	Decrement CTR, Branch if CTR Non-zero to Relative addr, set LR	bcl 16,0,addr
bdnzla addr	Decrement CTR, Branch if CTR Non-zero to Absolute addr, set LR	bcla 16,0,addr
bdnzlr	Decrement CTR, Branch if CTR Non-zero to LR	bclr 16,0
bdnzlrl	Decrement CTR, Branch if CTR Non-zero to LR, set LR	bclrl 16,0
bdnzt BI,addr	Decrement CTR, Branch if CTR Non-zero and Condition True to Relative addr	bc 8,BI,addr
bdnzta BI,addr	Decrement CTR, Branch if CTR Non-zero and Condition True to Absolute addr	bca 8,BI,addr
bdnztl BI,addr	Decrement CTR, Branch if CTR Non-zero and Condition True to Relative addr, set LR	bcl 8,BI,addr
bdnztla BI,addr	Decrement CTR, Branch if CTR Non-zero and Condition True to Absolute addr, set LR	bcla 8,BI,addr
bdnztlr BI	Decrement CTR, Branch if CTR Non-zero and Condition True to LR	bclr 8,BI
bdnztlrl BI	Decrement CTR, Branch if CTR Non-zero and Condition True to LR, set LR	bclrl 8,BI
bdz addr	Decrement CTR, Branch if CTR Zero to Relative addr	bc 18,0,addr
bdza addr	Decrement CTR, Branch if CTR Zero to Absolute addr	bca 18,0,addr
bdzf BI,addr	Decrement CTR, Branch if CTR Zero and Condition False to Relative addr	bc 2,BI,addr
bdzfa BI,addr	Decrement CTR, Branch if CTR Zero and Condition False to Absolute addr	bca 2,BI,addr
bdzfl BI,addr	Decrement CTR, Branch if CTR Zero and Condition False to Relative addr, set LR	bcl 2,BI,addr
bdzfla BI,addr	Decrement CTR, Branch if CTR Zero and Condition False to Absolute addr, set LR	bcla 2,BI,addr
bdzflr BI	Decrement CTR, Branch if CTR Zero and Condition False to LR	bclr 2,BI
bdzflrl BI	Decrement CTR, Branch if CTR Zero and Condition False to LR, set LR	bclrl 2,BI
bdzl addr	Decrement CTR, Branch if CTR Zero to Relative addr, set LR	bcl 18,0,addr
bdzla addr	Decrement CTR, Branch if CTR Zero to Absolute addr, set LR	bcla 18,0,addr
bdzlr	Decrement CTR, Branch if CTR Zero to LR	bclr 18,0
bdzlrl	Decrement CTR, Branch if CTR Zero to LR, set LR	bclrl 18,0

Extended mnemonic	Operation	Base mnemonic equivalent
bdzt BI,addr	Decrement CTR, Branch if CTR Zero and Condition True to Relative addr	bc 10,BI,addr
bdzta BI,addr	Decrement CTR, Branch if CTR Zero and Condition True to Absolute addr	bca 10,BI,addr
bdztl BI,addr	Decrement CTR, Branch if CTR Zero and Condition True to Relative addr, set LR	bcl 10,BI,addr
bdztla BI,addr	Decrement CTR, Branch if CTR Zero and Condition True to Absolute addr, set LR	bcla 10,BI,addr
bdztlr BI	Decrement CTR, Branch if CTR Zero and Condition True to LR	bclr 10,BI
bdztlrl BI	Decrement CTR, Branch if CTR Zero and Condition True to LR, set LR	bclrl 10,BI
bf BI,addr	Branch if Condition False to Relative addr	bc 4,BI,addr
bfa BI,addr	Branch if Condition False to Absolute addr	bca 4,BI,addr
bfctr BI	Branch if Condition False to CTR	bcctr 4,BI
bfctrl BI	Branch if Condition False to CTR, set LR	bcctrl 4,BI
bfl BI,addr	Branch if Condition False to Relative addr, set LR	bcl 4,BI,addr
bfla BI,addr	Branch if Condition False to Absolute addr, set LR	bcla 4,BI,addr
bflr BI	Branch if Condition False to LR	bclr 4,BI
bflrl BI	Branch if Condition False to LR, set LR	bclrl 4,BI
blr	Branch Unconditionally to LR	bclr 20,0
blrl	Branch Unconditionally to LR, set LR	bclrl 20,0
bt BI,addr	Branch if Condition True to Relative addr	bc 12,BI,addr
bta BI,addr	Branch if Condition True to Absolute addr	bca 12,BI,addr
btctr BI	Branch if Condition True to CTR	bcctr 12,BI
btctrl BI	Branch if Condition True to CTR, set LR	bcctrl 12,BI
btl BI,addr	Branch if Condition True to Relative addr, set LR	bcl 12,BI,addr
btla BI,addr	Branch if Condition True to Absolute addr, set LR	bcla 12,BI,addr
btlr BI	Branch if Condition True to LR	bclr 12,BI
btlrl BI	Branch if Condition True to LR, set LR	bclrl 12,BI

Branches That Incorporate Conditions

This section lists the branch extended mnemonics that let you specify a branch condition. If you know the likely outcome of a branch condition, you can add a suffix to a branch extended mnemonic to set a bit for predicting whether a branch will be taken. When you add a plus sign (+) as a suffix, the branch is predicted to be taken. When you add a minus sign (-) as a suffix, the branch is predicted not to be taken.

Extended mnemonic	Operation	Base mnemonic equivalent
beq [CRn,]addr	Branch if Equal to Relative addr	bc 12,[CRn+]2,addr
beqa [CRn,]addr	Branch if Equal to Absolute addr	bca 12,[CRn+]2,addr
beqctr [CRn]	Branch if Equal to CTR	bctr 12,[CRn+]2
beqctrl [CRn]	Branch if Equal to CTR, set LR	bcctrl 12,[CRn+]2
beql [CRn,]addr	Branch if Equal to Relative addr, set LR	bcl 12,[CRn+]2,addr
beqla [CRn,]addr	Branch if Equal to Absolute addr, set LR	bcla 12,[CRn+]2,addr
beqlr [CRn]	Branch if Equal to LR	bclr 12,[CRn+]2
beqlrl [CRn]	Branch if Equal to LR, set LR	bclrl 12,[CRn+]2
bge [CRn,]addr	Branch if Greater Than or Equal to Relative addr	bc 4,[CRn+]0,addr
bgea [CRn,]addr	Branch if Greater Than or Equal to Absolute addr	bca 4,[CRn+]0,addr
bgectr [CRn]	Branch if Greater Than or Equal to CTR	bctr 4,[CRn+]0
bgectrl [CRn]	Branch if Greater Than or Equal to CTR, set LR	bctrl 4,[CRn+]0
bgel [CRn,]addr	Branch if Greater Than or Equal to Relative addr, set LR	bcl 4,[CRn+]0,addr
bgela [CRn,]addr	Branch if Greater Than or Equal to Absolute addr, set LR	bcla 4,[CRn+]0,addr
bgelr [CRn]	Branch if Greater Than or Equal to LR	bclr 4,[CRn+]0
bgelrl [CRn]	Branch if Greater Than or Equal to LR, set LR	bclrl 4,[CRn+]0
bgt [CRn,]addr	Branch if Greater Than to Relative addr	bc 12,[CRn+]1,addr
bgta [CRn,]addr	Branch if Greater Than to Absolute addr	bca 12,[CRn+]1,addr
bgtctr [CRn]	Branch if Greater Than to CTR	bctr 12,[CRn+]1
bgtctrl [CRn]	Branch if Greater Than to CTR, set LR	bctrl 12,[CRn+]1
bgtl [CRn,]addr	Branch if Greater Than to Relative addr, set LR	bcl 12,[CRn+]1,addr
bgtla [CRn,]addr	Branch if Greater Than to Absolute addr, set LR	bcla 12,[CRn+]1,addr
bgtlr [CRn]	Branch if Greater Than to LR	bclr 12,[CRn+]1
bgtlrl [CRn]	Branch if Greater Than to LR, set LR	bclrl 12,[CRn+]1
ble [CRn,]addr	Branch if Less Than or Equal to Relative addr	bc 4,[CRn+]1,addr
blea [CRn,]addr	Branch if Less Than or Equal to Absolute addr	bca 4,[CRn+]1,addr
blectr [CRn]	Branch if Less Than or Equal to CTR	bctr 4,[CRn+]1
blectrl [CRn]	Branch if Less Than or Equal to CTR, set LR	bcctrl 4,[CRn+]1
blel [CRn,]addr	Branch if Less Than or Equal to Relative addr, set LR	bcl 4,[CRn+]1,addr

Extended mnemonic	Operation	Base mnemonic equivalent
blela [CRn,]addr	Branch if Less Than or Equal to Absolute addr, set LR	bcla 4,[CRn+]1,addr
blelr [CRn]	Branch if Less Than or Equal to LR	bclr 4,[CRn+]1
blelrl [CRn]	Branch if Less Than or Equal to LR, set LR	bclrl 4,[CRn+]1
blt [CRn,]addr	Branch if Less Than to Relative addr	bc 12,[CRn+]0,addr
blta [CRn,]addr	Branch if Less Than to Absolute addr	bca 12,[CRn+]0,addr
bltctr [CRn]	Branch if Less Than to CTR	bctr 12,[CRn+]0
bltctrl [CRn]	Branch if Less Than to CTR, set LR	bctrl 12,[CRn+]0
bltl [CRn,]addr	Branch if Less Than to Relative addr, set LR	bcl 12,[CRn+]0,addr
bltla [CRn,]addr	Branch if Less Than to Absolute addr, set LR	bcla 12,[CRn+]0,addr
bltlr [CRn]	Branch if Less Than to LR	bclr 12,[CRn+]0
bltlrl [CRn]	Branch if Less Than to LR, set LR	bclrl 12,[CRn+]0
bne [CRn,]addr	Branch if Not Equal to Relative addr	bc 4,[CRn+]2,addr
bnea [CRn,]addr	Branch if Not Equal to Absolute addr	bca 4,[CRn+]2,addr
bnectr [CRn]	Branch if Not Equal to CTR	bctr 4,[CRn+]2
bnectrl [CRn]	Branch if Not Equal to CTR, set LR	bctrl 4,[CRn+]2
bnel [CRn,]addr	Branch if Not Equal to Relative addr, set LR	bcl 4,[CRn+]2,addr
bnela [CRn,]addr	Branch if Not Equal to Absolute addr, set LR	bcla 4,[CRn+]2,addr
bnelr [CRn]	Branch if Not Equal to LR	bclr 4,[CRn+]2
bnelrl [CRn]	Branch if Not Equal to LR, set LR	bclrl 4,[CRn+]2
bng [CRn,]addr	Branch if Not Greater Than to Relative addr	bc 4,[CRn+]1,addr
bnga [CRn,]addr	Branch if Not Greater Than to Absolute addr	bca 4,[CRn+]1,addr
bngctr [CRn]	Branch if Not Greater Than to CTR	bctr 4,[CRn+]1
bngctrl [CRn]	Branch if Not Greater Than to CTR, set LR	bcctrl 4,[CRn+]1
bngl [CRn,]addr	Branch if Not Greater Than to Relative addr, set LR	bcl 4,[CRn+]1,addr
bngla [CRn,]addr	Branch if Not Greater Than to Absolute addr, set LR	bcla 4,[CRn+]1,addr
bnglr [CRn]	Branch if Not Greater Than to LR	bclr 4,[CRn+]1
bnglrl [CRn]	Branch if Not Greater Than to LR, set LR	bclrl 4,[CRn+]1
bnl [CRn,]addr	Branch if Not Less Than to Relative addr	bc 4,[CRn+]0,addr
bnla [CRn,]addr	Branch if Not Less Than to Absolute addr	bca 4,[CRn+]0,addr
bnlctr [CRn]	Branch if Not Less Than to CTR	bctr 4,[CRn+]0
bnlctrl [CRn]	Branch if Not Less Than to CTR, set LR	bctrl 4,[CRn+]0
bnll [CRn,]addr	Branch if Not Less Than to Relative addr, set LR	bcl 4,[CRn+]0,addr
bnlla [CRn,]addr	Branch if Not Less Than to Absolute addr, set LR	bcla 4,[CRn+]0,addr

Extended mnemonic	Operation	Base mnemonic equivalent
bnllrl [CRn]	Branch if Not Less Than to LR, set LR	bclrl 4,[CRn+]0
bns [CRn,]addr	Branch if Not Summary overflow to Relative addr	bc 4,[CRn+]3,addr
bnsa [CRn,]addr	Branch if Not Summary overflow to Absolute addr	bca 4,[CRn+]3,addr
bnsctr [CRn]	Branch if Not Summary overflow to CTR	bctr 4,[CRn+]3
bnsctrl [CRn]	Branch if Not Summary overflow to CTR, set LR	bctrl 4,[CRn+]3
bnsl [CRn,]addr	Branch if Not Summary overflow to Relative addr, set LR	bcl 4,[CRn+]3,addr
bnsla [CRn,]addr	Branch if Not Summary overflow to Absolute addr, set LR	bcla 4,[CRn+]3,addr
bnslr [CRn]	Branch if Not Summary overflow to LR	bclr 4,[CRn+]3
bnslrl [CRn]	Branch if Not Summary overflow to LR, set LR	bclrl 4,[CRn+]3
bnu [CRn,]addr	Branch if Not Unordered to Relative addr	bc 4,[CRn+]3,addr
bnua [CRn,]addr	Branch if Not Unordered to Absolute addr	bca 4,[CRn+]3,addr
bnuctr [CRn]	Branch if Not Unordered to CTR	bctr 4,[CRn+]3
bnuctrl [CRn]	Branch if Not Unordered to CTR, set LR	bctrl 4,[CRn+]3
bnul [CRn,]addr	Branch if Not Unordered to Relative addr, set LR	bcl 4,[CRn+]3,addr
bnula [CRn,]addr	Branch if Not Unordered to Absolute addr, set LR	bcla 4,[CRn+]3,addr
bnulr [CRn]	Branch if Not Unordered to LR	bclr 4,[CRn+]3
bnulrl [CRn]	Branch if Not Unordered to LR, set LR	bclrl 4,[CRn+]3
bso [CRn,]addr	Branch if Summary Overflow to Relative addr	bc 12,[CRn+]3,addr
bsoa [CRn,]addr	Branch if Summary Overflow to Absolute addr	bca 12,[CRn+]3,addr
bsoctr [CRn]	Branch if Summary Overflow to CTR	bctr 12,[CRn+]3
bsoctrl [CRn]	Branch if Summary Overflow to CTR, set LR	bctrl 12,[CRn+]3
bsol [CRn,]addr	Branch if Summary Overflow to Relative addr, set LR	bcl 12,[CRn+]3,addr
bsola [CRn,]addr	Branch if Summary Overflow to Absolute addr, set LR	bcla 12,[CRn+]3,addr
bsolr [CRn]	Branch if Summary Overflow to LR	bclr 12,[CRn+]3
bsolrl [CRn]	Branch if Summary Overflow to LR, set LR	bclrl 12,[CRn+]3
bun [CRn,]addr	Branch if Unordered to Relative addr	bc 12,[CRn+]3,addr
buna [CRn,]addr	Branch if Unordered to Absolute addr	bca 12,[CRn+]3,addr
bunctr [CRn]	Branch if Unordered to CTR	bctr 12,[CRn+]3
bunctrl [CRn]	Branch if Unordered to CTR, set LR	bctrl 12,[CRn+]3
bunl [CRn,]addr	Branch if Unordered to Relative addr, set LR	bcl 12,[CRn+]3,addr

Extended mnemonic	Operation	Base mnemonic equivalent
bunla [CRn,]addr	Branch if Unordered to Absolute addr, set LR	bcla 12,[CRn+]3,addr
bunlr [CRn]	Branch if Unordered to LR	bclr 12,[CRn+]3
bunlrl [CRn]	Branch if Unordered to LR, set LR	bclrl 12,[CRn+]3

Move From/To a Special-Purpose Register

This section describes the extended mnemonics that access a special-purpose register. When accessing a special-purpose register, note that

- the Time Base registers (TB) are PowerPC registers that are not implemented on the 601 processor
- the PowerPC mnemonic `mttb` is not implemented on the 601 processor
- registers MQ, RTCL, and RTCU are not part of the PowerPC architecture (they are included in the 601 processor for POWER compatibility)
- access to registers RTCL and RTCU is read-only when the 601 processor is in user mode
- register DEC is accessible only in supervisor mode on PowerPC processors (read-only in user mode on the 601 processor)
- the DBAT registers will be available only when PPCAsm supports 64-bit instructions

Extended mnemonic	Operation	Base mnemonic equivalent
mfasr Rx	Move From Address Space Register	mfspr Rx,280
mfctr Rx	Move From Count Register (CTR)	mfspr Rx,9
mfdar Rx	Move From Data Address Register (DAR)	mfspr Rx,19
mfdbatl Rx,n	Move From DBAT Lower Registers DBAT0L through DBAT3L	mfspr Rx,537+2*n
mfdbatu Rx,n	Move From DBAT Upper Registers DBAT0U through DBAT3U	mfspr Rx,536+2*n
mfdec Rx	Move From Decrementer Register (DEC)	mfspr Rx,6
mfdsisr Rx	Move From Data Storage Interrupt Status Register (DSISR)	mfspr Rx,18
mfear Rx	Move From External Access Register (EAR)	mfspr Rx,282
mfibatl Rx,n	Move From IBAT Lower Registers IBAT0L through IBAT3L	mfspr Rx,529+2*n
mfibatu Rx,n	Move From IBAT Upper Registers IBAT0U through IBAT3U	mfspr Rx,528+2*n

Extended mnemonic	Operation	Base mnemonic equivalent
mflr Rx	Move From Link Register (LR)	mfspr Rx,8
mfmq Rx	Move From MQ Register	mfspr Rx,0
mfpvr Rx	Move From Processor Version Register	mfspr Rx,287
mfrtcl Rx	Move From Real Time Clock Lower Register (RTCL)	mfspr Rx,5
mfrtcu Rx	Move From Real Time Clock Upper Register (RTCU)	mfspr Rx,4
mfsdr1 Rx	Move From Storage Description Register 1 (SDR1)	mfspr Rx,25
mfsprg Rx,n	Move From General Special Purpose Registers SPRG0 through SPRG3	mfspr Rx,272+n
mfsrr0 Rx	Move From Save/Restore Register 0 (SRR0)	mfspr Rx,26
mfsrr1 Rx	Move From Save/Restore Register 1 (SRR1)	mfspr Rx,27
mftb Rx	Move From Time Base Lower Register	mftb Rx,268
mftbu Rx	Move From Time Base Upper Register	mftb Rx,269
mfxer Rx	Move From Fixed Point Exception Register (XER)	mfspr Rx,1
mtasr Rx	Move To Address Space Register	mtspr 280,Rx
mtctr Rx	Move To Count Register (CTR)	mtspr 9,Rx
mtdar Rx	Move To Data Address Register (DAR)	mtspr 19,Rx
mtdbatl n,Rx	Move To DBAT Lower Registers DBAT0L through DBAT3L	mtspr 537+2*n,Rx
mtdbatu n,Rx	Move To DBAT Upper Registers DBAT0U through DBAT3U	mtspr 536+2*n,Rx
mtdec Rx	Move To Decrementer Register (DEC)	mtspr 22,Rx
mtdsisr Rx	Move To Data Storage Interrupt Status Register (DSISR)	mtspr 18,Rx
mtear Rx	Move To External Access Register (EAR)	mtspr 282,Rx
mtibatl n,Rx	Move To IBAT Lower Registers IBAT0L through IBAT3L	mtspr 529+2*n,Rx
mtibatu n,Rx	Move To IBAT Upper Registers IBAT0U through IBAT3U	mtspr 528+2*n,Rx
mtlr Rx	Move To Link Register (LR)	mtspr 8,Rx
mtmq Rx	Move To MQ Register	mtspr 0,Rx
mtrtcl Rx	Move To Real Time Clock Lower Register (RTCL)	mtspr 21,Rx
mtrtcu Rx	Move To Real Time Clock Upper Register (RTCU)	mtspr 20,Rx
mtsdr1 Rx	Move To Storage Description Register 1 (SDR1)	mtspr 25,Rx
mtsprg n,Rx	Move To General Special Purpose Registers SPRG0 through SPRG3	mtspr 272+n,Rx
mtsrr0 Rx	Move To Save/Restore Register 0 (SRR0)	mtspr 26,Rx
mtsrr1 Rx	Move To Save/Restore Register 1 (SRR1)	mtspr 27,Rx

Extended mnemonic	Operation	Base mnemonic equivalent
mttbl Rx	Move To Time Base Lower Register	mtspr 284,Rx
mttbu Rx	Move To Time Base Upper Register	mtspr 285,Rx
mtxer Rx	Move To Fixed Point Exception Register (XER)	mtspr 1,Rx

Traps

This section describes the trap extended mnemonics. Although 64-bit double comparison traps are listed, they will be available only when PPCAsm supports 64-bit instructions.

Extended mnemonic	Operation	Base mnemonic equivalent
trap	Trap Unconditionally	tw 31,0,0
tdeq Rx,Ry	Trap if Double is Equal	td 4,Rx,Ry
tdeqi Rx,SI	Trap if Double is Equal Immediate	tdi 4,Rx,SI
tdge Rx,Ry	Trap if Double is Greater Than or Equal	td 12,Rx,Ry
tdgei Rx,SI	Trap if Double is Greater Than or Equal Immediate	tdi 12,Rx,SI
tdgt Rx,Ry	Trap if Double is Greater Than	td 8,Rx,Ry
tdgti Rx,SI	Trap if Double is Greater Than Immediate	tdi 8,Rx,SI
tdle Rx,Ry	Trap if Double is Less Than or Equal	td 20,Rx,Ry
tdlei Rx,SI	Trap if Double is Less Than or Equal Immediate	tdi 20,Rx,SI
tdlge Rx,Ry	Trap if Double is Logically Greater Than or Equal	td 5,Rx,Ry
tdlgei Rx,SI	Trap if Double is Logically Greater Than or Equal Immediate	tdi 5,Rx,SI
tdlgt Rx,Ry	Trap if Double is Logically Greater Than	td 1,Rx,Ry
tdlgti Rx,SI	Trap if Double is Logically Greater Than Immediate	tdi 1,Rx,SI
tdlle Rx,Ry	Trap if Double is Logically Less Than or Equal	td 6,Rx,Ry
tdllei Rx,SI	Trap if Double is Logically Less Than or Equal Immediate	tdi 6,Rx,SI
tdllt Rx,Ry	Trap if Double is Logically Less Than	td 2,Rx,Ry
tdllti Rx,SI	Trap if Double is Logically Less Than Immediate	tdi 2,Rx,SI
tdlng Rx,Ry	Trap if Double is Logically Not Greater Than	td 6,Rx,Ry
tdlngi Rx,SI	Trap if Double is Logically Not Greater Than Immediate	tdi 6,Rx,SI
tdlnl Rx,Ry	Trap if Double is Logically Not Less Than	td 5,Rx,Ry
tdlnli Rx,SI	Trap if Double is Logically Not Less Than Immediate	tdi 5,Rx,SI
tdlt Rx,Ry	Trap if Double is Less Than	td 16,Rx,Ry
tdlti Rx,SI	Trap if Double is Less Than Immediate	tdi 16,Rx,SI

Extended mnemonic	Operation	Base mnemonic equivalent
tdne Rx,Ry	Trap if Double is Not Equal	td 24,Rx,Ry
tdnei Rx,SI	Trap if Double is Not Equal Immediate	tdi 24,Rx,SI
tdng Rx,Ry	Trap if Double is Not Greater Than	td 20,Rx,Ry
tdngi Rx,SI	Trap if Double is Not Greater Than Immediate	tdi 20,Rx,SI
tdnl Rx,Ry	Trap if Double is Not Less Than	td 12,Rx,Ry
tdnli Rx,SI	Trap if Double is Not Less Than Immediate	tdi 12,Rx,SI
tweq Rx,Ry	Trap if Word is Equal	tw 4,Rx,Ry
tweqi Rx,SI	Trap if Word is Equal Immediate	twi 4,Rx,SI
twge Rx,Ry	Trap if Word is Greater Than or Equal	tw 12,Rx,Ry
twgei Rx,SI	Trap if Word is Greater Than or Equal Immediate	twi 12,Rx,SI
twgt Rx,Ry	Trap if Word is Greater Than	tw 8,Rx,Ry
twgti Rx,SI	Trap if Word is Greater Than Immediate	twi 8,Rx,SI
twle Rx,Ry	Trap if Word is Less Than or Equal	tw 20,Rx,Ry
twlei Rx,SI	Trap if Word is Less Than or Equal Immediate	twi 20,Rx,SI
twlge Rx,Ry	Trap if Word is Logically Greater Than or Equal	tw 5,Rx,Ry
twlgei Rx,SI	Trap if Word is Logically Greater Than or Equal Immediate	twi 5,Rx,SI
twlt Rx,Ry	Trap if Word is Logically Greater Than	tw 1,Rx,Ry
twlgti Rx,SI	Trap if Word is Logically Greater Than Immediate	twi 1,Rx,SI
twlle Rx,Ry	Trap if Word is Logically Less Than or Equal	tw 6,Rx,Ry
twllei Rx,SI	Trap if Word is Logically Less Than or Equal Immediate	twi 6,Rx,SI
twllt Rx,Ry	Trap if Word is Logically Less Than	tw 2,Rx,Ry
twllti Rx,SI	Trap if Word is Logically Less Than Immediate	twi 2,Rx,SI
twlng Rx,Ry	Trap if Word is Logically Not Greater Than	tw 6,Rx,Ry
twlngi Rx,SI	Trap if Word is Logically Not Greater Than Immediate	twi 6,Rx,SI
twlnl Rx,Ry	Trap if Word is Logically Not Less Than	tw 5,Rx,Ry
twlnli Rx,SI	Trap if Word is Logically Not Less Than Immediate	twi 5,Rx,SI
twlt Rx,Ry	Trap if Word is Less Than	tw 16,Rx,Ry
twlti Rx,SI	Trap if Word is Less Than Immediate	twi 16,Rx,SI
twne Rx,Ry	Trap if Word is Not Equal	tw 24,Rx,Ry
twnei Rx,SI	Trap if Word is Not Equal Immediate	twi 24,Rx,SI
twng Rx,Ry	Trap if Word is Not Greater Than	tw 20,Rx,Ry
twngi Rx,SI	Trap if Word is Not Greater Than Immediate	twi 20,Rx,SI
twnl Rx,Ry	Trap if Word is Not Less Than	tw 12,Rx,Ry
twnli Rx,SI	Trap if Word is Not Less Than Immediate	twi 12,Rx,SI

Other Extended Mnemonics

This section describes the other extended mnemonics that PPCAsm supports. Although double-word-comparison extended mnemonics are listed, they will be available only when PPCAsm supports 64-bit instructions.

Extended mnemonic	Operation	Base mnemonic equivalent
clrldi Rx,Ry,n	Clear Left Double Immediate	rldicl Rx,Ry,0,n
clrlsldi Rx,Ry,b,n	Clear Left and Shift Left Double Immediate	rldic Rx,Ry,n,b-n
clrlwi Rx,Ry,n	Clear Left Word Immediate	rlwicl Rx,Ry,0,n
clrlslwi Rx,Ry,b,n	Clear Left and Shift Left Word Immediate	rlwic Rx,Ry,n,b-n
clrrdi Rx,Ry,n	Clear Right Double Immediate	rldicr Rx,Ry,0,63-n
clrrwi Rx,Ry,n	Clear Right Word Immediate	rlwicr Rx,Ry,0,63-n
cmpd crfD,Rx,Ry	Compare Doubleword	cmp crfD,1,Rx,Ry
cmpdi crfD,Rx,SI	Compare Doubleword Immediate	cmpi crfD,1,Rx,SI
cmpld crfD,Rx,Ry	Compare Logical Doubleword	cmpl crfD,1,Rx,Ry
cmpldi crfD,Rx,UI	Compare Logical Doubleword Immediate	cmpli crfD,1,Rx,UI
cmpw crfD,Rx,Ry	Compare Word	cmp crfD,0,Rx,Ry
cmpwi crfD,Rx,SI	Compare Word Immediate	cmpi crfD,0,Rx,SI
cmplw crfD,Rx,Ry	Compare Logical Word	cmpl crfD,0,Rx,Ry
cmplwi crfD,Rx,UI	Compare Logical Word Immediate	cmpli crfD,0,Rx,UI
crclr Bx	Condition Register Clear	crxor Bx,Bx,Bx
crmove Bx,By	Condition Register Move	cror Bx,By,By
crnot Bx,By	Condition Register Not	crnor Bx,By,By
crset Bx	Condition Register Set	creqv Bx,Bx,Bx
extldi Rx,Ry,n,b	Extract and Left Justify Double Immediate	rldicr Rx,Ry,b,n-1
extrdi Rx,Ry,n,b	Extract and Right Justify Double Immediate	rldicl Rx,Ry,b+n,64-n
extlwi Rx,Ry,n,b	Extract and Left Justify Word Immediate	rlwicr Rx,Ry,b,n-1
extrwi Rx,Ry,n,b	Extract and Right Justify Word Immediate	rlwicl Rx,Ry,b+n,64-n
insrdi Rx,Ry,n,b	Insert from Right Double Immediate	rldimi Rx,Ry,64-(b+n),b
insrwi Rx,Ry,n,b	Insert from Right Word Immediate	rlwimi Rx,Ry,64-(b+n),b
la Rx,D(Ry)	Load Address	addi Rx,Ry,D
la Rx,v	Load Address	addi Rx,Rv,Dv
li Rx,SI	Load 16-bit Signed Immediate	addi Rx,0,SI
lis Rx,SI	Load 16-bit Signed Immediate, Shifted	addis Rx,0,SI
mr Rx,Ry	Move Register	or Rx,Ry,Ry

Extended mnemonic	Operation	Base mnemonic equivalent
nop	No Operation	ori 0,0,0
not Rx,Ry	Complement Register (Logical Not)	nor Rx,Ry,Ry
rotldi Rx,Ry,n	Rotate Left Double Immediate	rldicl Rx,Ry,n,0
rotrdi Rx,Ry,n	Rotate Right Double Immediate	rldicl Rx,Ry,64-n,0
rotld Rx,Ry,Rz	Rotate Left Double	rldcl Rx,Ry,Rz,0
rotlwi Rx,Ry,n	Rotate Left Word Immediate	rlwicl Rx,Ry,n,0
rotrwi Rx,Ry,n	Rotate Right Word Immediate	rlwicl Rx,Ry,64-n,0
rotlw Rx,Ry,Rz	Rotate Left Word	rlwcl Rx,Ry,Rz,0
sldi Rx,Ry,n	Shift Left Double Immediate	rldicr Rx,Ry,n,63-n
slwi Rx,Ry,n	Shift Left Word Immediate	rlwicr Rx,Ry,n,63-n
srdi Rx,Ry,n	Shift Right Double Immediate	rldicl Rx,Ry,64-n,n
srwi Rx,Ry,n	Shift Right Word Immediate	rlwicl Rx,Ry,64-n,n
sub Rx,Ry,Rz	Subtract	subf Rx,Rz,Ry
subc Rx,Ry,Rz	Subtract Carrying	subfc Rx,Rz,Ry
subi Rx,Ry,val	Subtract Immediate	addi Rx,Ry,-val
subic Rx,Ry,val	Subtract Immediate Carrying	addic Rx,Ry,-val
subic. Rx,Ry,val	Subtract Immediate Carrying and Record	addic. Rx,Ry,-val
subis Rx,Ry,val	Subtract Immediate Shifted	addis Rx,Ry,-val

Glossary

absolute expression A single term or an arithmetic combination of terms that evaluate to an absolute value. Compare **relocatable expression**.

assembler directive An assembly statement that directs the assembler to perform a certain operation.

branch prediction A technique that attempts to guess the outcome of a conditional branch and fetch instructions along the predicted path.

Condition Register The register that holds the condition codes for instruction results and branch comparisons.

control section A grouping of instructions and/or data whose contents are stored in an object file.

cross-TOC call A call to a function that could be external to the caller's. A cross-TOC call requires that the Table of Contents Register be changed to the called function's TOC value. See also **Table of Contents Register**.

epilog code Instructions at the end of a called function to restore the context of the calling function.

export A named object that can be made accessible to another code fragment.

Extended Common Object File Format (XCOFF) The object file format produced by the assembler.

global name A named object that can be referenced anywhere in the source file.

identifier A symbol used to reference an object.

import A named object that is defined in another code fragment.

inner macro call See **nested macro call**.

keyword parameter A macro parameter that is identified in a macro call statement by its name rather than its position.

Link Register The register that holds the return address of the currently executing function.

location counter A counter that assigns control section addresses to machine instructions or data.

machine instruction An assembly statement that generates executable code.

macro See **macro definition.**

macro body The assembly statements between the macro prototype statement and the macro trailer directive (`ENDM`).

macro call An assembly statement that invokes a macro definition. The assembler expands the macro definition to generate assembly statements that replace the macro call statement.

macro call chain The macros that are suspended while the assembler expands the current macro definition.

macro definition A sequence of assembly statements that defines the name of a macro, the format of its call, and the assembly statements that are to be generated when the macro is invoked.

macro parameter A name representing an operand passed to a macro and referenced in the macro body. It consists of an ampersand followed by a valid assembler identifier.

macro prototype statement A macro definition statement that specifies the name of a macro and the format of calls to the macro.

name See **identifier.**

nested macro call A macro call from the body of another macro definition.

nonvolatile register A register whose contents should be preserved during a function call. Compare **volatile register**.

numeric constant A symbol used to reference a constant value.

outer macro call A macro call that is not made from the body of a macro definition.

PEF See **Preferred Executable Format**.

pipelining A technique that allows a processor to process more than one instruction at the same time.

Preferred Executable Format (PEF) The executable format for the PowerPC execution environment.

prolog code Instructions at the beginning of a called function to save the context of the calling function.

relocatable expression A term or an arithmetic combination of terms that evaluate to a relocatable value that is determined by the linker. Compare **absolute expression**.

RTOC See **Table of Contents Register**.

scope In an assembly file, the area in which a named object can be referenced.

scoreboarding A technique that a processor uses to detect shared-resource conflicts and synchronize the pipeline when instructions share data.

string A sequence of one or more ASCII characters (including spaces) enclosed in single quotation marks.

symbol A character or combination of characters used to represent an identifier, a numeric constant, or a string. See also **identifier**, **numeric constant**, and **string**.

symbolic expression A single term or an arithmetic combination of terms. See also **term**.

Table of Contents Register (RTOC) A register that points to the table of contents of the code fragment containing the code currently being executed. See also **cross-TOC call**.

term The designation for an identifier, or numeric constant, or the value of the location counter (*) that can be used in a symbolic expression.

transition vector A data structure in the static data area of a code fragment that usually consists of the address of the imported function and the TOC value for the fragment.

volatile register A register whose contents may be overwritten during a function call. Compare **nonvolatile register**.

XCOFF See **Extended Common Object File Format**.

Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Line art was created using Adobe Illustrator™ and Adobe Photoshop™.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITER
Joseph Aseo

LEAD WRITER
Tim Monroe

DEVELOPMENTAL EDITOR
Jeanne Woodward

ART DIRECTOR
Betty Gee

ILLUSTRATOR
Shawn Morningstar

PRODUCTION EDITOR
Rex Wolf

ONLINE PRODUCTION EDITOR
Alexandra Solinski

PROJECT MANAGER
Patricia Eastman

Special thanks to Fred Forsman, Liz Ghini, Alan Lillich, Pete Richardson, and Ira Ruben.

Acknowledgments to Gabriel Acosta–Lopez, Scott Boyd, Richard Clark, Ron Hochsprung, Bruce Jones, Michael Kline, Phil Koch, Jean-Charles Mourey, and Sean Parent.