TMS320C8x Register Allocator and Code Compactor User's Guide

Release 2.00

Literature Number: SPRU217
Manufacturing Part Number: D418022–9741 revision *
February 1997

TEXAS INSTRUMENTS

Preface

Read This First

About This Manual

This manual tells you how to use the TMS320C8x Register Allocator and Code Compactor (ppca) tool.

Notational Conventions

This document uses the following conventions.

- Program listings, program examples, and interactive displays are shown in a `special typeface` similar to a typewriter's. Examples use a **`bold version`** of the special typeface for emphasis; interactive displays use a **`bold version`** of the special typeface to distinguish commands that you enter from items that the system displays (such as prompts, command output, error messages, etc.).

  Here is an example of a command that you might enter:

  ```
  ppca myfile
  ```

- In syntax descriptions, the instruction, command, or directive is in a **bold typeface** font and parameters are in an *italic typeface*. Portions of a syntax that are in **bold** should be entered as shown; portions of a syntax that are in *italics* describe the type of information that should be entered. Here is an example of a directive syntax:

  name **.reg** *type*

  .reg is the directive. This directive requires two parameters, indicated by *name* and *type*.

- Square brackets ([and]) identify an optional parameter. If you use an optional parameter, you specify the information within the brackets; you do not enter the brackets themselves. Here is an example of a directive that has optional parameters:

  .cjump *label1* [, *label2*...]

  .cjump is the directive. This directive requires one or more parameters, indicated by *label1* and [, *label2*...]. Notice that when more than one parameter is specified, the parameters are separated by commas.

Related Documentation From Texas Instruments

The following books describe the TMS320C8x and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number.

TMS320C80 (MVP) C Source Debugger User's Guide (literature number SPRU107) describes the 'C8x master processor and parallel processor C source debuggers. This manual provides information about the features and operation of the debuggers and the parallel debug manager; it also includes basic information about C expressions and a description of progress and error messages.

TMS320C80 (MVP) Code Generation Tools User's Guide (literature number SPRU108) describes the 'C8x code generation tools. This manual provides information about the features and operation of the linker and the master processor (MP) and parallel processor (PP) C compilers and assemblers. It also includes a description of the common object file format (COFF) and shows you how to link MP and PP code.

TMS320C8x Master Processor User's Guide (literature number SPRU109) provides information about the master processor (MP) features, architecture, operation, and assembly language instruction set; it also includes sample applications that illustrate various MP operations.

TMS320C8x Multitasking Executive User's Guide (literature number SPRU112) provides information about the multitasking executive software features, operation, and interprocessor communications; it also includes a list of task error codes.

TMS320C8x Parallel Processor User's Guide (literature number SPRU110) provides information about the parallel processor (PP) features, architecture, operation, and assembly language instruction set; it also includes software applications and optimizations.

TMS320C8x System-Level Synopsis (literature number SPRU113) describes the 'C8x features, development environment, architecture, memory organization, and communication network (the crossbar).

TMS320C80 Transfer Controller User's Guide (literature number SPRU105) provides information about the transfer controller (TC) features, functional blocks, and operation; it also includes examples of block write operations for big- and little-endian modes.

TMS320C80 Video Controller User's Guide (literature number SPRU111) provides information about the video controller (VC) features, architecture, and operation; it also includes procedures and examples for programming the serial register transfer (SRT) controller and the frame timer registers.

TMS320C80 Digital Signal Processor Data Sheet (literature number SPRS023) describes the features of the TMS320C80 and provides pinouts, electrical specifications, and timings for the device.

TMS320C80 to TMS320C82 Software Compatibility User's Guide (literature number SPRU154) describes how to port software developed for one of these devices to the other. It also presents a set of software compatibility guidelines for developing software that will run on either device.

Modified Goertzel Algorithm in DTMF Detection Using the TMS320C80 Application Report (literature number SPRA066) describes the C-callable Goertzel dual-tone multi-frequency (DTMF) detection algorithm implementation on one of the TMS320C80's parallel processors.

Interfacing SDRAM to the TMS320C80 Application Report (literature number SPRA055) describes an interface between the 'C80 and the TMS626802-10 SDRAM(s). It illustrates the operation of the SDRAM interface and provides schematics for a baseline SDRAM interface to the TMS320C80.

Interfacing DRAM to the TMS320C80 Application Report (literature number SPRA056) describes an interface between the 'C80 and the TMS417400 DRAM(s). The report also describes the interface's connection to 4MB and 8MB SIMMs, a timing analysis for the proposed design, and a complete set of schematics.

TMS320C80 H.320 Software Library White Paper (literature number SPRY002) describes the TMS320C80's single-chip implementation of the H.320 videoconferencing standard.

TMS320C80 Digital Signal Processor Data Sheet (literature number SPRS023) describes the features of the TMS320C80 and provides pinouts, electrical specifications, and timings for the device.

TMS320C8x Software Development Board Installation Guide (literature number SPRU150B), included with the TMS320C8x SDB, provides information about how to install and use the SDB.

TMS320C8x Software Development Board Programmer's Guide (literature number SPRU178), included with the TMS320C8x SDB, provides descriptions of hardware functions, complete API references, theory of operation, and example code for the SDB.

TMS320C8x (DSP) — Fundamental Graphic Algorithms Application Book (literature number SPRA069) contains application notes that demonstrate several 'C8x application programs. These notes are as follows:

Transform3 Command is an application presented for the Master Processor which demonstrates how two-dimensional graphic coordinate transformations can use the 'C8x floating-point unit efficiently. Performance estimation also is provided.

Transform4 Command is an application presented for the Master Processor which demonstrates how three-dimensional graphic coordinate transformations can use the 'C8x floating-point unit efficiently. Performance estimation also is provided.

Draw Colored Lines Command is an application presented for the 'C8x parallel processor which demonstrates how to generate colored lines efficiently. Performance estimation also is provided.

Draw Colored Trapezoids Command is an application presented for the 'C8x parallel processor which demonstrates how to generate colored trapezoids efficiently. Performance estimation also is provided.

Parallel Processor Integer and Floating-Point Math describes several C-callable 32-bit IEEE 754 standard floating-point math subroutines. Performance estimation also is provided.

Implementation of the Vector Maximum Search Benchmark on the TMS320C8x Parallel Processor Application Report (literature number SPRA087) uses the *Vector Maximum Search* benchmark to demonstrate the efficient performance of the TMS320C8x parallel processors. This manual describes a software implementation that uses the parallel processor's advanced assembly language features to implement this benchmark.

Acoustic Echo Cancellation — Algorithms and Implementation on the TMS320C8x Application Report (literature number SPRA063) describes the implementation of an integral N-tap digital acoustic echo canceller on the TMS320C8x parallel processor. The report presents a brief discussion of generic echo cancellation algorithms. The implementation considerations for a 512-tap (64-ms span) echo canceller on the TMS320C8x are described in detail, as well as the software logic and flow for each program module.

Viewing TMS320C8x Register Bit Fields and Memory-Mapped Registers in the HLL Debugger (TMS320 DSP Designer's Notebook, DNP# 69) describes a method for viewing 'C8x register bit fields and memory-mapped registers in the 'C8x HLL debugger.

Writing TMS320C8x PP Code Under the Multitasking Executive (TMS320 DSP Designer's Notebook, DNP# 73) provides useful guidelines for writing 'C8x Parallel Processor (PP) assembly language or C code that can run under the Multitasking Executive.

TMS320C82 Digital Signal Processor Data Sheet (literature number – Preliminary) describes the features of the TMS320C82 and provides pinouts, electrical specifications, and timings for the device.

Trademarks

320 Hotline On-line is a trademark of Texas Instruments Incorporated.

If You Need Assistance . . .

❑ World-Wide Web Sites

TI Online	http://www.ti.com
Semiconductor Product Information Center (PIC)	http://www.ti.com/sc/docs/pic/home.htm
DSP Solutions	http://www.ti.com/dsps
320 Hotline On-line™	http://www.ti.com/sc/docs/dsps/support.htm

❑ North America, South America, Central America

Product Information Center (PIC)	(972) 644-5580		
TI Literature Response Center U.S.A.	(800) 477-8924		
Software Registration/Upgrades	(214) 638-0333	Fax: (214) 638-7742	
U.S.A. Factory Repair/Hardware Upgrades	(281) 274-2285		
U.S. Technical Training Organization	(972) 644-5580		
DSP Hotline	(281) 274-2320	Fax: (281) 274-2324	Email: dsph@ti.com
DSP Modem BBS	(281) 274-2323		

DSP Internet BBS via anonymous ftp to ftp://ftp.ti.com/mirrors/tms320bbs

❑ Europe, Middle East, Africa

European Product Information Center (EPIC) Hotlines:

Multi-Language Support	+33 1 30 70 11 69	Fax: +33 1 30 70 10 32	Email: epic@ti.com
Deutsch	+49 8161 80 33 11 or +33 1 30 70 11 68		
English	+33 1 30 70 11 65		
Francais	+33 1 30 70 11 64		
Italiano	+33 1 30 70 11 67		
EPIC Modem BBS	+33 1 30 70 11 99		
European Factory Repair	+33 4 93 22 25 40		
Europe Customer Training Helpline		Fax: +49 81 61 80 40 10	

❑ Asia-Pacific

Literature Response Center	+852 2 956 7288	Fax: +852 2 956 2200
Hong Kong DSP Hotline	+852 2 956 7268	Fax: +852 2 956 1002
Korea DSP Hotline	+82 2 551 2804	Fax: +82 2 551 2828
Korea DSP Modem BBS	+82 2 551 2914	
Singapore DSP Hotline		Fax: +65 390 7179
Taiwan DSP Hotline	+886 2 377 1450	Fax: +886 2 377 2718
Taiwan DSP Modem BBS	+886 2 376 2592	

Taiwan DSP Internet BBS via anonymous ftp to ftp://dsp.ee.tit.edu.tw/pub/TI/

❑ Japan

Product Information Center	+0120-81-0026 (in Japan)	Fax: +0120-81-0036 (in Japan)
	+03-3457-0972 or (INTL) 813-3457-0972	Fax: +03-3457-1259 or (INTL) 813-3457-1259
DSP Hotline	+03-3769-8735 or (INTL) 813-3769-8735	Fax: +03-3457-7071 or (INTL) 813-3457-7071
DSP BBS via Nifty-Serve	Type “Go TIASP”	

❑ Documentation

When making suggestions or reporting errors in documentation, please include the following information that is on the title page: the full title of the book, the publication date, and the literature number.

Mail: Texas Instruments Incorporated
Technical Documentation Services, MS 702
P.O. Box 1443
Houston, Texas 77251-1443

Email: comments@books.sc.ti.com

Note: When calling a Literature Response Center to order documentation, please specify the literature number of the book.

Contents

Tables

Examples

Chapter 1

Getting Started With ppca

The register allocator and code compactor (ppca) allocates symbolic variables to physical registers and (optionally) compacts assembly language instructions for the TMS320C8x's parallel processors. This chapter explains the functionality of the tool and describes how to invoke it. The ppca-specific directives are listed and a brief description of each is provided.

Topic **Page**

1.1 About ppca

ppca performs two independent, although related, actions that make it easier to use the TMS320C8x's parallel processors (PP). The tool

❑ Automatically allocates symbolic variables to physical PP registers

❑ Compacts assembly language instructions to reduce code size and improve execution speed

ppca allows the source and destination operands for each instruction to be referred to symbolically, as in a high-level language. These symbolic variables can then be allocated to physical PP registers. ppca automates the process of allocating the symbolic variables to physical registers.

The PP assembly language allows multiple complex instructions to occur in a single cycle. In addition to register allocation, ppca can reorder instructions so that the instructions can be performed in parallel. The reordering process ensures that the data dependencies in the original code are maintained. The resulting code should be smaller in size and execute faster. The tool can accept input in unpacked (where only one of the complex instructions is used in a cycle), partially packed, or fully packed form (where all of the instruction slots are filled in the cycle). The tool does not break apart packed operations in an instruction but may add to any unused slots available, unless prohibited by a directive.

1.2 Program Development

The manner in which ppca is intended to be used is as follows. An algorithm is developed for the TMS320C8x's parallel processors and prepared in a ppca source file (for example, myfile.p). In this source file, symbolic variables are freely used, as they would be in any high-level language program. To help ppca perform its register allocation and code compaction tasks, special directives are added to the source file by the user. ppca is then run on the source file to allocate the variables to registers and (optionally) reorder the source file instructions into a more compact, parallel format. The tool produces an assembly code file (in this case, myfile.ss) which can be assembled by the PP assembler (ppasm) to generate object code.

1.3 Invoking ppca

To invoke the tool, enter:

ppca [–p | –p2] [–k] [–n | –n1 | –n2 | –n3] [*ppasm_options*]
[*input_file* [*output_file*]]

ppca	Invokes the register allocator and code compactor.
–p	Performs *both* register allocation and code compaction on the input file. By default, only register allocation is performed.
–p2	Performs code compaction, but *not* register allocation.
–k	Keeps the pre-process file.
–n or **–n1**	Prevents the commenting of directives.
–n2	Prevents the output of line numbers, block headings, and the opcode-usage table.
–n3	Prevents the commenting of directives, line numbers, block headings, and the opcode-usage table.
ppasm_options	Assembler options. Any command-line option that can be specified for ppasm also can be specified for ppca.
input_file	The input source file. By default, the input source file is assumed to have an extension of .p.
output_file	An alternate name for the output from the tool. By default, the output is written to a file with the same name as the source file and the extension .ss. For example, if the input file is named myfile.p, the output file will be named myfile.ss.

Most often the tool is used for register allocation only. This operation can be performed by simply specifying

```
ppca myfile
```

If code compaction is also required, use

```
ppca –p myfile
```

By default, register allocation is performed first, followed by code compaction. Performing the code compaction before the register allocation may produce a more compact output. For more information, see Section 2.9 Compacting Before Register Allocation. This section explains the use of the –p2 and –n command line options. For a description of the –k option see Section 2.8.

1.4 Summary of Directives

The input format is a PP assembly language file. ppca uses the same parser as the PP assembler (ppasm), and the same constructs used for ppasm can be used with ppca. Additional directives may be required to assist the allocation and compaction process. These directives are summarized in Table 1–1 and are more fully described in Chapter 2.

Table 1–1. Summary of Allocation and Compaction Directives

Directive	Syntax	Description	Page
.cexit	**.cexit** [*variable* | *register* [, *variable* | *register*...]]	specifies a conditional jump to a point outside the current file and the registers and variables that are live at that point	2-8
.cjump	**.cjump** *label1* [, *label2...*]	specifies a conditional jump to one or more labels in the current file	2-6
.copy	**.copy** ["]*filename*["]	copies the contents of an include file into the output file	2-15
.dummy	*name* **.dummy**	declares a dummy variable to which unneeded instruction results are assigned	2-2
.entry	**.entry** [*variable* | *register* [, *variable* | *register*...]]	specifies the instruction that is the entry point for the file and the registers and variables that are live on entry to the routine	2-4
.free	**.free** *register* [, *register...*]	specifies which registers can be allocated	2-11
.live	**.live** *variable* [, *variable...*]	specifies a new instance of a variable	2-12
.lock	**.lock** *register* [, *register...*]	specifies which registers cannot be allocated	2-11
.nodelay	**.nodelay**	placed before a branch or call instruction to reserve its delay slots	2-14
.nopack	**.nopack**	temporarily disables compaction	2-13
.pack	**.pack**	re-enables compaction following a .nopack directive	2-13
.reg	*name* **.reg** *type*	declares each variable and its type	2-2
.uexit	**.uexit** [*variable* | *register* [, *variable* | *register*...]]	specifies an unconditional jump to a point outside the current file and the registers and variables that are live at that point	2-8
.ujump	**.ujump** *label1* [, *label2...*]	specifies an unconditional jump to one or more labels in the current file	2-5

Chapter 2

Using ppca

This chapter explains how to use ppca. In describing the tasks that the tool performs, the ppca-specific directives are discussed, including their syntax, and examples are provided.

Topic **Page**

2.1 Declaring Variables

The register allocation performed by ppca takes place on variables in the source file. Just as with a high-level language, each variable used in the assembly file must be explicitly declared along with its type. Use the .reg directive to declare all variables. The syntax for this directive is:

name **.reg** *type*

where *name* is a variable name consistent with those used in the ppasm .set directive, and *type* is one of the types described in Table 2–1.

Table 2–1. Variable Types

Type	Description
d	specifies data unit registers
ga	specifies global address registers
gx	specifies global index registers
la	specifies local address registers
lx	specifies local index registers

No other PP registers (for example, mf, loop registers) can be allocated. Two .reg directives cannot have the same variable name.

Dummy Variables

In some circumstances, the result of an instruction is not required. For example, if the instruction sets status information and it is the status that is required, not the write to the destination. In these circumstances, the result can be specified as a dummy variable. Dummy variables must first be declared using the directive:

name **.dummy**

Example 2–1 illustrates how the dummy variable is used to store the result of an instruction.

Example 2–1. Using dummy variables to store instruction results

```
    dummy     .dummy

    dummy = var1 – var2   ; not interested in the result,
                          ; only in the status
    var3  = [n] var1      ; move based on the status
```

In every instance that the dummy variable is used, ppca attempts to allocate a free register using the following criteria:

1) If there is a global transfer to a D register in parallel with the write to the dummy variable, that D register is used.
2) If there is a local transfer to a D register in parallel with the write to the dummy variable (and the write to the dummy variable is not a global transfer) that D register is used.
3) If the first two criteria are not met, any D register that is not live is used (for example, any D register that does not contain the value of any variable that is read later in the program).

If none of the criteria is met, an allocation error message is displayed.

Reads of dummy registers are not allowed. However, they can be achieved using a regular variable declared of d type which is preceded by a .live directive as shown in Example 2–2.

Example 2–2. Using .live to achieve a read of a dummy variable

```
var1   .reg   d
var2   .reg   d
dum1   .reg   d

       .live dum1   ; this forces a new instance of dum1
       d6 = var1 + var2
       || d6 = dum1 * dum1
       ...
       .live dum1   ; this forces a new instance of dum1
       d6 = var1 – var2
       || d6 = dum1 * dum1
```

The .live directive is discussed further in Section 2.5 on page 2-12.

2.2 Defining Entry Points

ppca assumes the first instruction in the code is the entry point for the file. If the first instruction is not the entry point or the code contains multiple entry points, you must specify the entry point by using the .entry directive. To specify an entry point, a .entry directive must be placed *before* the instruction which is the entry point. The syntax for .entry is:

.entry [*variable* **|***register* [,*variable* **|***register*...]]

The list of variables and registers separated by commas specify the variables and registers that are live on entry to the routine. Live variables and registers contain values that have been loaded prior to the entry point. These values will be used in the routine. If they are not defined explicitly in the .entry directive, ppca may try to use these registers, overwriting the values in them before they have been read. Thus, the code would not function as intended. Because most routines have some registers live on entry, it is recommended that you specify an entry point with the .entry directive in every source file.

2.3 Specifying Program Flow

To perform correct register allocation and code compaction, ppca must comprehend the program flow through an assembly language program. The tool cannot predict program branches, and assumes that program execution is sequential unless otherwise directed.

The directives .ujump, .cjump, .uexit, and .cexit provide program flow information. They are used to indicate that there is a different flow from the instruction preceding the directive, instead of (or in addition to) the instruction following the directive. Use these directives to indicate branches, returns, and loops.

An important point to remember when these directives are used to indicate a branch or call, is that they must be placed after the instruction in the second delay slot, *not* after the instruction containing the branch.

Use .ujump and .cjump to indicate branches and returns *inside* the current file. Use .uexit and .cexit to indicate branches and returns *outside* the current file.

Specifying a branch to a label inside the current file

The .ujump and .cjump directives allow you to specify one of three types of branches:

- ❑ A branch to a label in the current file
- ❑ A return from the current subroutine to the calling function or functions that reside in the current file
- ❑ A loop

The .ujump directive

The .ujump directive is used to specify an unconditional branch to a label in the current file or to specify a return from the current subroutine to the calling function or functions that reside in the current file.

The syntax of the .ujump directive is:

.ujump *label1* [,*label2*...]

This directive is placed after the instruction at which the branch takes place. It informs ppca that there is no path from the instruction before the directive to the instruction following the directive. One or more labels separated by commas should be placed after the .ujump directive to specify that there is a path from this point to the label.

An example of when more than one label should be specified occurs when a subroutine is called from several different places in a file. The subroutine will have several possible return points. Place the .ujump directive at the end of the subroutine and specify the labels for all possible returns.

The labels used with .ujump must be located within the current file. Labels which are .global (for example, external references) are not allowed, and .uexit should be used instead.

The code fragment in Example 2–3 shows an unconditional branch.

Example 2–3. Using .ujump to specify an unconditional branch

```
LAB1:
   ... = var1
   br = LAB3    ; unconditional branch
   nop
   nop
   .ujump LAB3  ; branch occurs here because of the
                ; 2-cycle delay
LAB2:
   ... = var1   ; flow never reaches this instruction
LAB3:           ; flow skips to this point
```

In Example 2–3, .ujump is used because there is an unconditional branch to LAB3. Because this branch is unconditional, there is no flow from the nop before the .ujump to the instruction at LAB2. This causes the read of var1 at LAB1 to be the last read of var1, which extends the lifetime of var1 to LAB1. Because the lifetime of var1 ends at LAB1, var1's register can be reused by ppca after LAB1. This would not be the case if the .ujump directive were missing, since the lifetime of var1 would extend to LAB2. Thus, it is vital to the correct operation of the register allocation, that the program flow directives be added, wherever necessary, to the source file.

The .cjump directive

The .cjump directive is used to specify a conditional branch to a label in the current file or to specify a return from the current subroutine to the calling function or functions that reside in the current file.

The syntax of the .cjump directive is:

.cjump *label1* [, *label2*...]

Example 2–4 shows how .cjump can be used for a conditional branch.

Example 2–4. Using .cjump to specify a conditional branch

```
   var1 = ...
   ... = var2
LAB1:
   ... = var1
   br = [n] LAB3   ; conditional branch
   nop
   nop
   .cjump LAB3
LAB2:
   ... = var1      ; flow continues if n bit is clear
LAB3:              ; flow skips to here if n bit is set
```

Because the jump to LAB3 is conditional, there is a flow between the nop before LAB2 and the instruction at LAB2. Unlike the previous Example 2–3 for .ujump directives, this flow extends the lifetime of var1.

This directive is also used to indicate hardware loops that have been set up in the code. The .cjump directive should be placed after the loop end-address, and the label should be the loop start-address value. You can specify multiple labels to indicate that nested loops share a common end-address, as shown in Example 2–5.

Example 2–5. Using .cjump with multiple labels

```
loop_setup:
   le1 = inner_end ; inner loop ends at label inner:
   lrs1 = 7        ; inner loop occurs 7 times
   ls1 = inner     ; inner loop starts at label inner:
   le0 = outer_end ; outer loop ends at outer_end:
   lrs0 = 5        ; outer loop occurs 5 times
   nop
   nop             ; by default, outer loop begins at
                   ; label outer:, 2 cycles after lrs0=5
outer:
   var1 = var2     ; inner loop starts here
inner:
   var2 = var2 + 1 ; inner loop starts here
inner_end:         ; both outer and inner loops end here
outer_end:
                   ; indicates that flow can go to the
                   ; inner or outer labels or to the
                   ; following instruction
   var1 = var2
   .cjump outer, inner
```

Specifying a branch or return to a point outside the current file

The .uexit and .cexit directives allow you to specify one of two types of exits:

- ❑ A branch to a label outside the current file
- ❑ A return from a subroutine to the calling function or functions that reside outside the current file

The .uexit directive is used to specify an unconditional exit from the current file. Its syntax is:

.uexit [*variable* **|** *register* [, *variable* **|** *register*...]]

Specifying .uexit informs ppca that there is no path from the instruction before the directive to the instruction following the directive.

The .cexit directive is used to specify a conditional exit from the current file. Its syntax is:

.cexit [*variable* **|** *register* [, *variable* **|** *register*...]]

.cexit is placed after the instruction takes place. This directive informs ppca that there is a path from the instruction to the next instruction and an additional branch outside the current file.

After .uexit or .cexit, you can specify a list of variables or registers separated by commas. This list specifies which variables and registers must be kept live to this point in the file. This prevents the registers from being reused and preserves their values for a destination point outside the current file.

You do not have to specify a label with .uexit or .cexit because ppca only analyzes one file. When using .cjump or .ujump, the tool automatically follows the program flow to determine which variables need to be kept live. However, because you are branching or returning to a point outside the current file, you must use .cexit and .uexit to explicitly specify the variables used in the destination function so that the lifetimes of the variables are extended to the exit point.

The difference between .cexit and .uexit is that .cexit is conditional and .uexit is unconditional. This does not affect the lifetime of variables that are explicitly mentioned in .cexit or .uexit, but this does affect the lifetime of other variables that are in the code. Example 2–6 shows an unconditional branch.

Example 2–6. Using .uexit to specify an unconditional branch

```
LAB1:
   ... = var1
   br = iprs    ; unconditional return
   nop
   nop
   .uexit var2  ; return occurs here because of the
                ; 2-cycle branch delay
LAB2:
   ... = var1   ; flow never reaches here
```

In Example 2–6, .uexit is used because there is an unconditional return from this subroutine. The calling function in this example is assumed to be outside the current file. It is also assumed that the calling function will receive var2 as a return value. Specifying var2 in .uexit prevents the register used by var2 from being reallocated by the ppca prior to the exit, ensuring that the calling function can read the register contents. The lifetime of var2 is extended to just before LAB2. Because this exit is unconditional, there is no flow from the nop before .uexit and the instruction at LAB2. This causes the read of var1 at LAB1 to be the last read of var1, which causes the lifetime of var1 to end at LAB1. Because the lifetime of var1 ends at LAB1, var1’s register can be reused by ppca.

If the branch is conditional, the register var1 cannot be reused after LAB1 because execution might continue to LAB2, as shown in Example 2–7.

Example 2–7. Using .cexit to specify a conditional branch

```
   var1 = ...
   ... = var2
LAB1:
   ... = var1
   br = [n] iprs   ; conditional return
   nop
   nop
   .cexit var2     ; if n is set, branch occurs here
                   ; because of the 2-cycle branch delay
LAB2:
   ... = var1      ; flow continues if n bit is clear
```

Calling external subroutines

If the subroutine being called is within the current source file, then no special consideration needs to be made. To indicate a subroutine call, the .ujump or .cjump directives should be used depending on whether or not the call was conditional, as shown in Example 2–8.

Example 2–8. Using .cjump and .ujump when indicating subroutine calls

```
      var1 = ...
            :
        call = sub1         ; unconditional call
        nop
        nop
        .ujump sub1         ; unconditional jump
ret1:       :
        ... = var1
            :
        call =[n] sub1      ; conditional call
        nop
        nop
        .cjump sub1         ; conditional jump
ret2:       :
            :
sub1:       :
            :
        br = iprs
        nop
        nop
        .ujump ret1, ret2   ; two possible return points
```

Because the subroutine is within the source file, ppca can evaluate execution flow. It can detect the fact that there is a dependency path from the write of var1 through the subroutine, back to ret1 and to the read of var1.

When the subroutine is an external subroutine (for example, it resides in a separate file), then it is important that the call to the subroutine is always followed by .cexit, regardless of whether the call is a conditional branch to the subroutine or not. This informs ppca that there is a flow through the subroutine call back to the return point. You may also need to specify the live registers in the parameters to the .cexit which are inputs into the subroutine. Specify any outputs from the subroutine in a .entry directive, as shown in Example 2–9.

Example 2–9. Using .entry to specify outputs from a subroutine

```
      var1 = ...
            :
      d0   = ...      ; parameter to extsub1
      d1   = ...      ; parameter to extsub1
      call = extsub1  ; unconditional call
      nop
      nop
      .cexit d0,d1    ; conditional exit (subroutine)
      .entry d2       ; output register from extsub1
            :
      ... = var1
            :
```

2.4 Reserving Registers

ppca analyzes the code to determine where registers are already used, so that it does not allocate those registers to variables. However, you might want to tell the tool explicitly not to use certain registers, even though these registers are not apparently used in the supplied code. This capability is provided by the directives:

.lock *register* [,*register*...]

.free *register* [,*register*...]

where *register* is a valid data unit or index register. The .lock directive must be placed *before* the block of instructions the directive refers to. The .free directive must be placed *after* the block of instructions the directive refers to. Multiple .locks and .frees may be placed throughout the code. Although the same register may be locked multiple times without resulting in an error, the first .lock or .free determines the locking as shown in Example 2–10.

Example 2–10. Using .lock and .free to reserve and free registers

```
    .lock d0,a2,x1  ; d0, a2, and x1 cannot be allocated
    <code>
    .lock a2,x2     ; x2 cannot be allocated, a2 already
                    ; locked
    <code>
    .free d0,x1     ; d0 and x1 are free to be allocated
    <code>
    .free a2,d0     ; a2 is free, d0 is already free
```

The .lock and .free directives only affect the contiguous sequence of instructions that they surround. They are not affected by the flow-control and entry-point directives.

2.5 Controlling Register and Variable Lifetime

The register-allocation process determines the lifetime of registers and variables by checking every read of a register or variable and searching backward to find the first write of that register or variable. The lifetime of a register extends from the time it is first allocated to the time it becomes free.

Loop optimizations that perform a dummy read of a register or variable can hamper register allocation.

Example 2–11. Loop containing an uninitialized register

```
       reg1 = d3
loop1:
       reg1 = d0 + d1
       || *a0++  = reg1
       .cjump loop1
```

In Example 2–11, successive values of reg1 are computed and stored in a loop. The first value of reg1 is not initialized. Because there is a read of reg1 at this instruction, ppca looks backward through the code to find a corresponding write of reg1. If a corresponding write is found, you may not want the register to which reg1 is allocated to be considered live in instructions prior to loop1 because this prevents this register from being used by other instructions. To prevent this, a .live directive can be placed at the point where reg1 becomes live, as shown in Example 2–12.

Example 2–12. Using .live to indicate the lifetime of register

```
       .live reg1
loop1:
       reg1 = d0 + d1
       || *a0++ = reg1
       .cjump loop1
```

A list of variables separated by commas can be used with .live. For example,

```
.live reg1, reg2, reg3
```

2.6 Controlling Compaction

When using the compaction process, some instructions can be represented by more than one type of operation. For instance, an operation that uses the stack pointer such as `d0 = *sp++` could be represented as either a local or a global transfer. Normally this is not a problem as ppca selects the appropriate form that provides the maximum compaction.

However, the one instruction type that does require careful consideration is the register move instruction.

The register move instruction

The register move instruction can be treated as either an ALU or global transfer. However, the two forms are not identical, because the ALU version sets status but the global transfer does not. If the more general form of register move is used (`d0 = d1`), then ppca assumes that either form can be used. This means that the instruction could be packed with either another ALU or global transfer as appropriate. If the form `d0 =g d1`, then the global transfer form is used. `nop || d0 = d1` does not force the instruction to use a global transfer.

The pack directives

The following directives are only applicable if compaction is being performed. It might be necessary to control compaction by temporarily disabling and re-enabling the packing operation. The following directives are for this purpose.

.nopack

.pack

The .nopack directive disables compaction temporarily until the next .pack directive. The .pack directive re-enables compaction after a .nopack directive has disabled compaction.

2.7 Controlling Scheduling

ppca automatically ensures that position-dependent instructions are retained in the same position in the compacted output as they appeared in the input file. Other instructions may be moved into the delay slots of a branch or lrs instruction when it is valid to do so, but the branch or lrs itself is not moved within the block of instructions being compacted.

Although the compaction process attempts to use all instruction slots, occasionally you might want to reserve delay slots. If you want to reserve the two delay slots of a branch instruction, you can use the .nodelay directive. The syntax for this directive is:

.nodelay

When this directive is placed *before* a branch or call instruction, ppca leaves the branch or call instruction's delay slots unused, as nops. You might want to reserve delay slots if your code branches into the delay slots of other branches.

2.8 Copying the Contents of an Include File

The ppca .copy directive is a modification of the existing PP assembler (ppasm) .copy directive. The ppasm .copy directive is used to copy the contents of an include file into the output file. Because ppca-specific directives such as .ujump, and .entry are not part of the input format for ppasm, they must be commented out of the source file. ppca comments out all of its directives in the source file, but does not modify the contents of include files. If include files contain ppca-specific directives, they must also be commented out, otherwise they will result in syntax errors when the output file is assembled.

The .copy directive has been modified so that it causes the code to be copied rather than included in the output file. The .copy directive itself is commented out. You can nest .copy directives up to 32 levels deep. The syntax for this directive is:

.copy [**”**]*filename*[**”**]

Copying takes place only when the .copy directive occurs in the source code or in another include file that is a .copy itself. Files included with a .include directive are not copied, nor is any file specified with a .copy within that include file.

An intermediate file, containing any copied include files is stored in a file with the same base name as the source file but with a .pca extension. This file is normally deleted after use, but can be retained if the –k command line option is used. In Example 2–13, a code fragment is input to ppca to produce an intermediate file.

Example 2–13. Copying the contents of an include file

(a) Code input to ppca

```
    .copy     ”copy.i”
    .include  ”include.i”
    var1  =   var2 + var3
```

(b) ppca output

```
;;;       .copy     ”copy.i”
;;;var1   .reg      d        ; contents of ”copy.i”
          .include  ”include.i”
          var1  =   var2 + var3
```

2.9 Compacting Before Register Allocation

If the input file is to be compacted as well as register allocated, ppca first performs the allocation and then performs the compaction. More compaction could be achieved if the compaction was performed before the register allocation. However, the resultant code could be compacted so that it required more registers than are available, causing the allocation process to fail.

However, you can compact the code without performing register allocation by specifying the –p2 option on the command line. This option enables compaction to be performed before allocation by running ppca twice. For example

```
ppca –p2 –n infile.p infile.int    ; compact-only to an
                                   ; intermediate file
ppca infile.int infile.ss          ; allocate-only
```

This two-pass approach gives you the ability to edit the compacted code in the intermediate file, should the allocation process fail. You must specify the –n option so that the tool will not comment out its own directives in the first invocation of ppca.

Because you are running ppca twice, the tool generates two sets of instruction numbers, block comments and opcode-usage tables. To prevent the tool from generating this information twice, use the –n2 command line option which inhibits the output of this information.

Chapter 3

ppca Outputs

This chapter describes the types of output produced by ppca. These types include usage tables as well as progress and error messages.

Topic **Page**

3.1 Output Format

The output of ppca is a file containing PP assembly language instructions ready for input to the PP assembler (ppasm). This file has the same base name as the source file but with an extension of .ss. All the ppca-specific directives are changed to comments by inserting `;;;` (three semicolons).

Each instruction is numbered with a comment. For example,

```
;;; 27
  d0 = d1 + d2
;;; 28
  d4 = d2>>d3
```

In addition, the output file can contain up to three tables:

- *set table* is a list of .set statements which map the variables to the physical registers. This table is added to the output file if the input file contains variables.
- *variable-usage table* maps variables to the instructions in which they are used
- *opcode-usage table* shows whether the ALU, local, and global portions of each instruction are being used

Variable Naming Conventions

Each variable declared by the .reg directive is translated to a .set directive and placed in a set table. The register to which the variable has been allocated is filled in by ppca. If different registers are used for the same variable in different places in the code, the variable name has _*xxx* appended to it, where *xxx* is a sequence number starting at 001. The tool also appends the _*xxx* to the variable names within the code.

If the exact sequence `_xxx` is the last four characters of a variable name, ppca fills in the sequence number over the top of the `xxx` characters. This preserves the text formatting of the source file so that comments and code remain lined up in the ppca output file. For example, the statement

```
reg = d0 + d1; comment
```

will appear in the output file as

```
reg_001 = d0 + d1; comment
```

Notice that the code and comment are shifted to the right.

To avoid this problem, append `_xxx` to the name of the original variable in the input file:

```
reg_xxx = d0 + d1; comment
```

This statement will appear in the output file as:

```
reg_001 = d0 + d1; comment
```

Now the code and comment are aligned exactly as in the input file.

If an allocation runs out of physical registers, false registers are allocated to variables. These names have the following syntax:

_*<register-type><index-number>*

For instance, if all the d registers are used, then _d0, _d1, etc. are used. Similarly, if all the global index registers are used, then _gx0, _gx1 are used.

Variable-Usage Table

The variable-usage table helps you to view the lifetime of variables. This table is useful when the register allocation process runs out of registers and the variables are assigned to false registers. This table helps you determine where code can be rearranged or modified to use fewer physical registers.

In the variable-usage table, variables that use the same physical register are grouped together. The group of variables is ordered so that the first instruction to use the register is listed first and so on.

The variable name is placed next to the physical register name, followed by a series of columns that represent instructions. These instructions are indicated by a header in the table which numbers the instructions. If the variable is not used in that instruction, the column contains a . (period). If the variable is used, the column contains an X. Because it is possible for more than one variable to share the same register in the same instruction (if it is the end of one variable's lifetime and the start of another), variables using the same register can have more than one X in the same column. To make this distinction clearly, the first X in a sequence is substituted with a < and the last X is substituted with a > as shown in Example 3–1.

Example 3–1. Variable-Usage Table

```
;;; Variable-Usage Table
;;;                           1111111111222222222233333333334444444
;;;                 0123456789012345678901234567890123456789012343
;;;xyPosition   d4 ......................<XXXXX>.................
;;;yDelta_001   d4 ............................<>................
;;;DrawableBase d4 ................................<X>...........
;;;ExtAddress   d4 ..................................<XX>........
;;;dLa_PT_A     d5 <XXXXXXXXXXXXXXXXXXXX>........................
;;;dWidth_001   d5 ........................<XXXXXXXXXXX>.........
;;;NextLpCount  d5 ....................................<XXXXX>...
;;;yDelta_002   d5 ..........................................<XXX
```

Opcode-Usage Table

The opcode-usage table shows the compactness of the code and which unused instruction slots are available for further compaction. The table consists of four lines representing the multiply, ALU, global-transfer, and local-transfer portions of an instruction. Each column of the table represents an instruction. A column can contain one of three characters:

- `X` which means the instruction uses that portion of the instruction
- `.` which means the instruction does not use that portion of the instruction
- `–` which means that portion of the instruction cannot be used because of restrictions in the format of the opcode

The format for the opcode-usage table is shown in Example 3–2.

Example 3–2. Opcode-Usage Table

```
;;;Opcode-Usage Table:
;;;                 11
;;;       012345678901
;;;   MPY --...---...-
;;;   ALU XX...XXX...X
;;;GLOBAL ---XXXXXXXX-
;;; LOCAL .-X--XXXXXX.
```

3.2 Progress and Error Messages

ppca outputs various progress, warning, and error messages categorized as parse-stage messages, register allocation messages, and compaction messages. The following subsections describe the three types of messages and explain what you should do if you receive these messages.

Parse-stage messages

Before any register allocation or compaction is performed, the source file is parsed to determine if it contains valid PP assembly language syntax. Also, the format of the directives is checked to ensure that it is valid. If any errors occur at this stage, ppca does not perform register allocation or compaction. The following message is output, and the program terminates.

```
PPCA: No allocation or compaction performed due to errors
in parsing
```

Any errors are also written to stderr and to the list output file along with the message:

```
PARSE STAGE: XXX errors, XXX warnings
```

During the parse stage, ppca may print warnings when it detects situations which may affect the register allocation process, such as the following:

```
an instruction forces variables var1 and var2 to be
identical
```

```
an instruction forces variables var1 and var2 to be
companions
```

```
an instruction forces variable var1 to be register var2
```

```
an instruction forces variables var1 and var2 to be pairs
```

Register-allocation messages

If there are no parse errors and –p2 has not been specified on the command line, register allocation proceeds. If the allocation process completes successfully, the following message is output:

```
ALLOCATE STAGE: n variables successfully allocated
```

If the file does not contain .reg directives, this message is output:

```
ALLOCATE STAGE: No register allocation required
```

Errors that you might encounter during the register-allocation process are:

```
Failure to allocate a register for variable var1 due to
companion register requirements cause the allocation
process to fail
```

```
Failure to allocate a register for variable var1 due to
pairing with register var2 cause the allocation process to
fail
```

```
Failure to allocate a register for the dummy var1 when
there are no free registers for use as a dummy write
```

```
Read of a dummy variable var1 at instruction n a variable
declared as a .dummy was read from
```

If the allocation process runs out of registers, variables are allocated false registers. If this occurs, the following message is output and you must rework the input source to use fewer physical registers.

```
n extra register(s) of type var1 has been added to perform
the allocation
```

Another error message which can occur during the allocation process is

```
Companion variables var1 and var2 tied to the same
physical register have interfering writes/reads at
instruction n
```

where ppca has detected that there may be overlapping lifetimes of companion registers in the input source. If you receive this error message, check that the instructions in the source file are correct.

Compaction messages

If –p is specified on the command line, then code compaction is performed. If the compaction process is successful, the following message is output and the program terminates.

```
COMPACTION STAGE: Compacted the original n instructions
into m instructions
```

The compaction stage can fail if no valid schedule for the instructions is provided. In such cases, the following message is output for each block that fails to schedule

```
SCHEDULING FAILED FOR BLOCK n
```

and the program terminates with the message

```
COMPACTION STAGE: Failed to obtain a valid schedule
```

During the scheduling process, ppca detects potential problems related to position-dependent instructions. These problems output the following warning:

```
Position-dependent instruction n for Block m was put into
slot s instead of slot t, which could produce incorrect
schedule
```

Appendix A

Known Deficiencies

The known deficiencies in ppca are the following.

Deficiency

A variable can only be paired with one other variable. Multiple pairings are not allowed.

For example, if you input the following code

```
x      .reg d
var1   .reg d
var2   .reg d

   d0 = x[n]var1
   d1 = x[n]var2
```

you will receive this error

```
*** ERROR! line9: E0800: Cannot pair variable x with var2
as it is already paired with var1
   d1 = x [n] var2
```

Deficiency

When ppca performs compaction, labels that occur within a block are placed together at the beginning of the block. This ensures that any jumps to the code at a particular label reach all code within the block. This capability, however, causes the identification of particular instructions to be lost (for example, the use of .global labels).

If you input the following file and you invoke ppca with the –p option

```
lab1:
   d0 = d1 + d2
lab2:
   *a0 = d3
lab3:
   *a9++ = d4
```

ppca outputs the following code

```
lab1:
lab2:
lab3:
   d0 = d1 + d2
||     *a9++ = d4
||     *a0 = d3
```

Deficiency

ppca allows for an extra delay slot between the loading and reading of the comm register to allow for the latency required to set the Q bit. At this time, the tool does not take into account the three-instruction delay required when using the PTQ bit, as described in the *Interrupt* subsection of the *Waiting for Packet Transfer Completion* section in the *PP User's Guide*. To prevent this problem, code compaction must be turned off using the .nopack/.pack directives.

For example, if you are using PTQ and ppca produced the following code

```
comm = 1 \\ 28  ; Set the PTQ bit
d0 = *a0        ; Read the memory brought in by the
                ; packet transfer
```

the code will not execute correctly because the PTQ interrupt would not occur until three instructions later.

Work-around

Use the .nopack/.pack directives to turn off compaction and to pad the code by three instructions to account for the three-instruction delay.

```
.nopack
comm = 1 \\ 28     ; Set the PTQ bit
nop
nop
nop
.pack
d0 = *a0
```

Although nops are used for this example, you can use any instruction to pad the unpacked region.

Deficiency

Compaction does not work correctly with conditional assembly directives (.if/.else/.endif). For example, consider the following code

```
   d1 = *a0++
   d2 = *a0++
.if 1
   d5 = d1 + d2
.else
   d5 = d1 – d2
.endif
   *a1++ = d5
```

Running ppca on this code with compaction turned on will produce the following output

```
.if 1
.else
.endif
d1 = *a0++
d2 = *a0++
d5 = d1 + d2
d5 = d1 – d2
*a1++ = d5
```

Work-around Instead of using conditional assembly directives, use the conditional compilation directives (#if/#else/#endif). For example, modify the code from the previous example as follows:

```
      d1 = *a0++
      d2 = *a0++
#if 1
      d5 = d1 + d2
#else
      d5 = d1 – d2
#endif
      *a1++ = d5
```

Then, before running ppca, run the file through the C compiler's pre-processor using the following command. Notice that this command assumes that the code is located in the file myfile.p.

```
ppcl –pon myfile.p
```

The pre-processor produces the following output in myfile.pp.

```
d1 = *a0++
d2 = *a0++
d5 = d1 + d2
a1++ = d5
```

This file can then be run through ppca in the normal way.