asm21toc user manual

Adrian Johnstone   Elizabeth Scott

Technical Report

CSD-TR-LA00-A

February 7, 2000

Royal Holloway
University of London

Department of Computer Science
Egham, Surrey TW20 0EX, England

Contents

List of Figures

List of Tables

Chapter 1

Introduction

`asm21toc` is a tool for translating applications written in the assembly language for Analog Device's ADSP-21xx family of fixed point Digital Signal Processors into ANSI standard C. The motivation is twofold: conversion to C aids programme comprehension for programmers who are not experienced with DSP assembler languages; and the high level source can be compiled for new architectures. This last is particularly important given recent trends in the DSP market, which until recently has been one of the last bastions of assembly language programming.

Traditional 16-bit DSP's have small virtual address spaces and highly constrained register/operation relationships that make them difficult compiler targets. A factor 20 difference in performance between hand coded routines and the output of the C compiler is not unusual, and as a result applications for devices like the ADSP-21xx are rarely written in C.

Recent DSP introductions such the TMS320C6x are based on general purpose register architectures that are much more compiler-friendly, but perhaps more importantly these devices feature wide multiple internal pipelined datapaths allowing parallel execution of operations. Correct scheduling of operations across functional units, taking into account register access constraints, is hard to do by hand, and debugging optimised code on architectures without hardware interlocks to preserve the sequential semantics of the unscheduled code can be very frustrating.

High level language compilers for these kinds of architectures incorporate scheduling algorithms which *do* preserve the sequential code semantics. The experience in the general purpose processor world is that hand scheduling of assembler for pipelined, superscalar and VLIW machines is so hard that assembly language programming has been almost completely abandoned. It is not at this stage clear whether assembly programming will be similarly abandoned in the DSP sector: TI for instance has put considerable effort into producing a scheduler that works with a purely sequential assembler input. This tool performs very well and is an attractive entry point for programmers who prefer assembler. Most probably, future applications will be programmed mainly at high level with some hand-coding of core routines.

1.1 Translation styles

Program comprehension means different things to different users. A high level language programmer with little or no assembly experience might prefer a translated style in which the high level code is fluent and clear with no references to quirks of the original implementation or architecture. An assembler programmer, on the other hand, who is migrating a program to high level code might prefer to see a translation which is as close as possible to the original assembly code so that a line-by-line check of the competence of the translation can be performed.

To address each of these extremes, `asm21toc` provides four styles of translation:

1. basic translation in which control flow is represented by `goto` statements and labels and ADSP-21xx machine registers as global variables;

2. control flow reduced translation in which, where possible, `goto`'s are removed and rewritten using conventional high level structures such as `if-then-else` and `while-do`;

3. register-reduced translation in which many references to machine registers are back substituted; and,

4. control-and-register reduced translation which combines styles 2 and 3.

The kind of translation to be performed is specified using command line option switches. All styles are compatible and may be intermixed within a single translated application. Essentially the translation level only affects the code within each translated C function so, for instance, a function in a control-flow reduced translation may be replaced by the same function from a basic translation without modifying any of the surrounding functions.

1.2 Translator inputs and outputs

Most `asm21toc` applications are made up of multiple source files that are separately assembled and then linked together. `asm21toc` must also perform this linking step to reconcile declarations and usages of functions and variables. Rather than using a separate linker, `asm21toc` requires a complete list of primary source files to be presented when the tool is invoked. (Secondary source files, that is, files that are included from within primary source files should not be specified as they will be automatically read in as necessary). For each file in this list, `asm21toc` generates a single C source file with the same name as the DSP source file but with a suffix of `.c`, along with a master file called `a2c_main.c` and a header file `a2c_main.h`. These two files contain declarations for register definitions, some macros and a `main()` function that calls initialisation functions and then the function nominated as the entry point for the application.

```
.module test;

.entry adder;

.const five=5;
.var/dm z;

.init z: 3;

adder: nop;

      ax0 = dm(z);
      ay0 = five;
      ar = ax0+ay0;
      dm(z) = ar;

      rts;

.endmod;
```

Figure 1.1 Example DSP source file

1.2.1 Translated file format

To illustrate the general layout of translated files we use the simple DSP source file `test.dsp` shown in Figure 1.1. The corresponding output file, which was generated with the command `asm21toc -eadder test`, is presented in Figure 1.2.

Each translated file `#include`s the file `a2c_main.h` so as to access global register declarations and support macros. Any `.const` declarations in the ADSP-21xx source are then rendered using `#define` statements followed by declarations for variables appearing within `.var` clauses in the original source.

Variable initialisation is performed with a special initialiser function rather than by using the ANSI-C variable initialisers. These initialiser functions, one *per* translated source file, are called by the `main()` function in `a2c_main.c` before the entry point function for the application is activated.

The rest of the translated file comprises prototypes for the the translated functions, in alphabetical order followed by the functions themselves, again in alphabetical order.

1.2.2 Documentation outputs

In addition to the ANSI-C translations, `asm21toc` collates a variety of descriptive information into two documentation files.

1. A *log file* (called `asm21toc.log`) which contains summary information on functions and their control flow along with a combined symbol map and cross reference.

```
/* Translated from module 'test' in file 'test.dsp'
   by asm21toC V1.10 Feb 20 2000 14:27:05 */

#include "a2c_main.h"

/* Define manifest constants */

#define five 5

/* Declare variables local to this module */

  int z;

/* Declare variable initialisation function */

void asm21_data_initialiser_for_test(void)
{
  z = 3;
}

/* Function prototypes */

void adder(void);

/* Function bodies */

void adder(void)
{
/* Input parameter registers: (none) */
/* Output parameter registers: (none) */
/* Local registers: AZ, AN, AV, AC, AX0, AY0, AR */
/* Unused live global registers: (none) */
adder:
  /* nop here in ASM21xx code */
  ax0 = z;
  ay0 = five;
  ar = ax0 + ay0;
  z = ar; /* dataflow analyser marks this expression as dead  */
  return;
}


/* End of translation from module 'test' */
```

Figure 1.2 Example DSP source file and corresponding translation

2. A *graph file* (called `asm21toc.vcg`) which contains a graphical rendering of the translated application showing the breakdown into functions; the flow of control between and within functions; and the availability of *substitutable* expressions. This file maybe displayed on-screen or printed using the VCG (Visualisation of Compiler Graphs) utility written by Georg Sänder [San95]. VCG is public domain software and versions are available for Unix/X-windows and Windows-3.1 (which also runs under Windows-95, Windows-98 and Windows-NT).

1.3 Manual overview

The rest of this manual provides both a guide to running `asm21toc` and an overview of its internal operation. Chapter 2 describes the command line options and memory requirements for `asm21toc`. The log file and the graph file are described in Chapter 3. A comprehensive list of error messages along with explanatory paragraphs is given in Chapter 4 followed by a description of the tool's internals in Chapter 5 and finally an appendix which includes reprints of three published papers concerning `asm21toc`.

Chapter 2

Running asm21toc

The overall flow for the tool is shown in Figure 2.1. `asm21toc` reads one or more ADSP-21xx assembler source file (of default filetype `.dsp`) and, for each one an ANSI-C (`.c`) translation file along with files `a2c_main.c` and `a2c_main.h`. An entry label must be specified to the translator using the `-e` switch.

2.1 Memory demands

`asm21toc` requires quite large amounts of memory when running since its internal data structures are coded for generality and performance rather than compactness. In addition, the complete text of the original source files must be held in memory during translation. Across a variety of translated applications, observed memory consumption rates tend to be around 1Mbyte of runtime memory space per 300–500 lines of `asm21toc` source.

Memory utilisation varies according to the mix of directives and statements in the source code: processing of many directives requires no dynamic memory. Processing of constant and variable directives consumes memory during the initial parsing stages when the symbol table is being built, but no extra memory during back end phases. Control flow reduction and data flow analysis consume memory proportional to the number of high level control flow constructs detected, and the number of substitutable expressions found, respectively.

These requirements are not unmanagable: due to the small (16K word) program address space of the ADSP-21xx processors programs we would expect the largest of applications to run without swapping on a 64MByte machine. One large application that we have translated requires 20Mbyte of run time store with a further 0.5Mbyte of memory to hold the translator program itself.

2.2 Portability of asm21toc

`asm21toc` is implemented using the `rdp` translator generator [JS98]. `rdp` itself is highly portable and has been used on a wide variety of systems including PC's running DOS, Windows-95, Windows-98 and Windows NT; Macintoshes; Acorn Archimedes and a very wide range of Unix systems using both the GNU `gcc` and vendor's own compilers. The code generated by `rdp`, and the hand

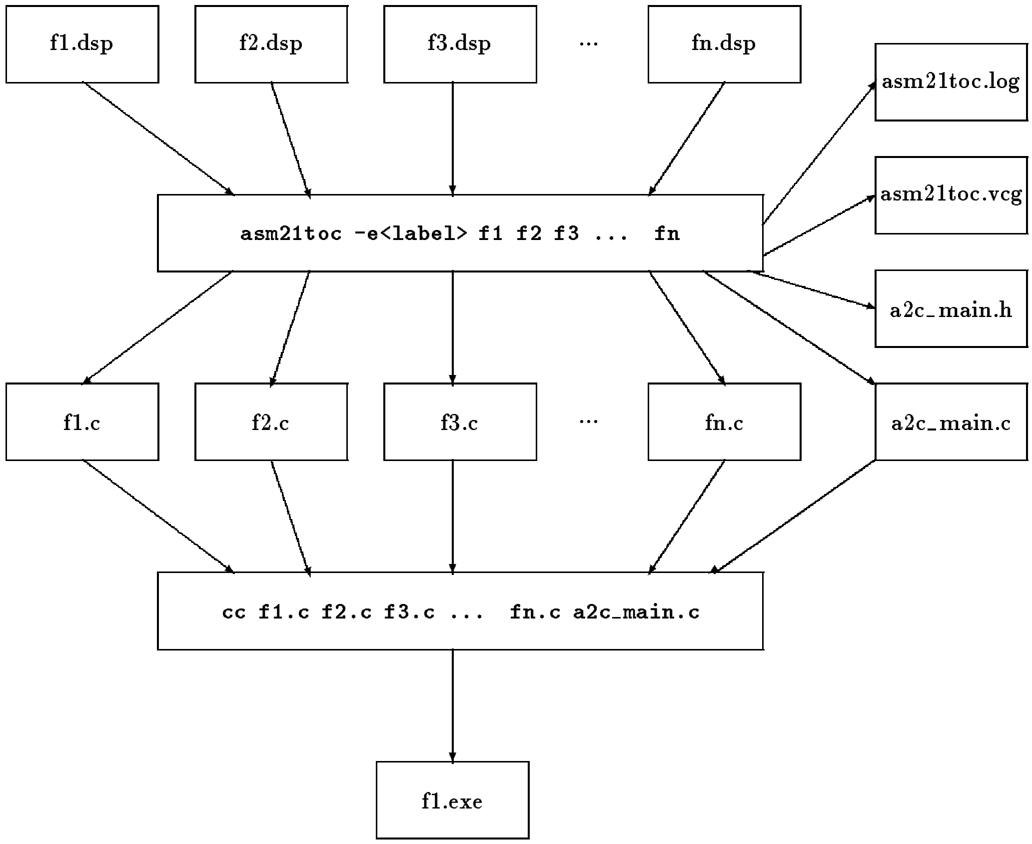

Figure 2.1 asm21toc flow

written parts of `asm21toc` are written in ANSI-C (not C++) and perform only straightforward text I/O. Development has been for the Win-32S windows subsystem, but versions have also been successfully tested in Windows-NT and Compaq Unix for the Alpha processor. In general, porting of `asm21toc` is straightforward for any system that supports an ANSI-C compiler.

2.3 Command line options and parameters

The `asm21toc` command accepts parameters which can either be *option switches* which are denoted by a leading minus sign (`-`) followed by a letter, or *filename arguments* which are anything else. Options and filenames can be intermixed: it is not necessary to place the filename(s) after the options.

All `asm21toc` options are processed in strict left to right order. This is significant because some options can override the actions of other options: in such cases the rightmost instance of an option will override any earlier ones.

2.3.1 Summary help

As is conventional for Unix and MS-DOS utilities, issuing the command name with no parameters generates a summary help message which lists the tools capabilities. The help message for `asm21toc` is shown in Figure 2.2.

2.3.2 `asm21toc` filename parameters

Any `asm21toc` parameter that does not consist of minus sign (`-`) followed by a non-whitespace character will be taken as a filename.

`asm21toc` attempts to add a default filetype to each filename you specify. It starts at the rightmost character, and looks backwards for a period (`.`). If it encounters one *before* it finds the start of the filename or an instance of either the Unix or MS-DOS directory separators (`/` and `\`) then it assumes that the user has supplied a filetype and leaves the filename untouched. If it does not find a period then it appends `.dsp` to that filename.

2.3.3 `asm21toc` options

Table 2.1 summarises the list of command line switches that may be supplied to `asm21toc`. The third column marks some switches as *developer* options and the rest as *user* options. Developer options are intended to ease debugging of the translator and are unlikely to be useful during normal translation. The rest of this section comprises a description of each option, in alphabetical order.

`-C` enable control flow analysis

Control flow analysis replaces `goto` statements in the ANSI-C translation with the usual high-level control constructs such as `if-then-else` and `while-do`. Note that there are configurations of `goto` statements that cannot be mapped

```
asm21toc - map asm21 to ANSI-C V01.11 (c) Adrian Johnstone 1999, 2000
Generated on Feb 21 2000 11:09:09 and compiled on Feb 21 2000 at 11:10:37

Usage: asm21toc [options] source[.dsp]

-f    Filter mode (read from stdin and write to stdout)
-l    Make a listing
-o<s> Write output to filename
-s    Echo each scanner symbol as it is read
-S    Print summary symbol table statistics
-t<n> Tab expansion width (default 8)
-T<n> Text buffer size in bytes for scanner (default 1Mbyte)
-v    Set verbose mode
-V<s> Write derivation tree to filename in VCG format

-e<s> Set entry label
-r<n> Set VCG rendering level
-c    Render call edges unfolded
-d<n> Set level of debugging information
-k    Kill comments
-m<s> Register type modifier (default none)
-C    Enable control flow analysis
-D    Force do loop counters to use single variable
-E    Enable expression substitution
-L    Enable line number comments
```

Figure 2.2 asm21toc help message

Switch	Function	Type
`-C`	Enable control flow analysis	user
`-c`	Render call edges unfolded	user
`-D`	Force do loop counters to use single variable	user
`-d<n>`	Set level of debugging information	developer
`-E`	Enable expression substitution	user
`-e<s>`	Set entry label	user
`-f`	Filter mode (read from stdin and write to stdout)	developer
`-k`	kill comments	user
`-L`	Enable line number comments	user
`-l`	Make a listing	user
`-m<s>`	Register type modifier (default none)	user
`-o<s>`	Write log file to filename	user
`-r<n>`	Set VCG rendering level	user
`-S`	Print summary symbol table statistics	developer
`-s`	Echo each scanner symbol as it is read	developer
`-T<n>`	Text buffer size in bytes for scanner (default 1Mbyte)	user
`-t<n>`	Tab expansion width (default 8)	user
`-V<s>`	Write derivation tree to filename in VCG format	developer
`-v`	Set verbose mode	user

Table 2.1 Command line switch summary

into high level constructs – these are called *irreducible* schemas and any function that contains an irreducible schema should be retranslated without the `-C` option instead.

`-c` render call edges unfolded

By default, the blue edges that connect `call` nodes to the entry point of the associated functions are suppressed when VCG initially displays a flow graph. The edges may be switched on with the *Folding* | *Expose/Hide Edges* | *2* command. The `-c` option inverts this behaviour so that by default the call edges are displayed at startup. More information on using VCG with `asm21toc` may be found in Chapter 3.

`-D` force do loop counters to use single variable

The ADSP-21xx `do` assembler instruction manipulates an internal four-deep stack of loop start addresses. In general, these manipulations are not statically visible and so the translated code *simulates* the internal stack. This simulation is likely to confuse a C programmer who does not know about the underlying ADSP architecture. The `-D` option suppresses the simulation code and adopts a much simpler model in which each `do` loop is given its own counter that is assumed to be locally initialised. This option should be used with great care since it can generate code that has different semantics to the original assembler.

`-d`*level* set level of debugging information

`asm21toc` can present debugging information concerning the progress of its internal phases. In the distributed version of `asm21toc` the textual debugging messages have been suppressed, but the `-d` switch with a non-zero value may be used to add node numbering information to the graph output file.

`-E` enable expression substitution

The ADSP-21xx architecture contains functional units with *pipeline* registers on their input and outputs. Computations require the transfer of operands from data or program memory to input registers such as `ax0` or `mx0`, followed by the computation itself, followed by a transfer from an output register such as `ar` to program or data memory. A naïve translation would render these translations literally, whereas a high level programmer would probably write an expression directly on the main memory operands. The `-E` option causes `asm21toc` to trace each read reference to a machine register back through the code to find corresponding writes. If there is only one register write for a given register read, then that read operation may be back substituted with the write expression. The effect of this is to simplify the translated code.

`-e`*label* set entry label

`asm21toc` writes out a mainline file called `a2c_main.c` which contains declarations for register variables and a `main()` function that calls each initialiser function and then calls the entry function for the application. `asm21toc` does not attempt to deduce the entry address which must instead be supplied using the `-e` command line switch. If no `-e` option is supplied then a warning message is issued and this preprocessor directive is written into the `a2c_main.c` file.

```
#error no entry label specified: use -e switch on the asm21toc command
```

The effect of this directive is to output an error message during compilation of the `a2c_main.c` file.

`-f` filter mode

In filter mode, input is read from the standard input and written to the standard output. The `-f` option sets the input to `stdin` (either the keyboard, or the output of a previous operation within a pipe) and the output to `stdout`, but subsequent `-o` options or filenames can be used to override this. Similarly, this option overrides any previous `-o` option.

`-k` kill comments

Assembler comments are visually formatted to suit the layout conventions of assembler code, and may be inappropriate to the ANSI-C standard. The `-k` option completely suppresses comment translation.

Level		Output
0	Literal	The raw datastructures are rendered
1	All unfolded	The complete application is unfolded
2	Expressions folded	Only the control flow graph nodes are shown
3	Sequences folded	Only the nodes with multiple entries or exits are shown
4	Functions folded	Only the function header nodes are shown

Table 2.2 Graph rendering levels

`-L` enable line number comments

When activated, this option inserts a comment at the start of each translated expression showing the corresponding line number in the original assembler source.

`-l` make a listing

Usually, only lines containing syntax errors are echoed to the screen whilst `asm21toc` is running. When a `-l` option is issued each line of the source file is echoed to the screen as it is read, along with the line number of the current file. Nested include files are noted with the name of the file at entry and exit from each source file.

`-m`*modifier* set register type modifier

By default, variables and registers in the original DSP application are mapped to `int` variables in the translated application. The `-m` option takes a string with no embedded spaces such as `short` or `long` which will be prepended to all declarations and casts. More complex modifiers such as `long long` can be achieved by specifying a macro name such as `-mLONGLONG` and then defining the macro name on the compiler command as follows.

```
cc -DLONGLONG="long long" f1 ...
```

`-o` set output filename

This option may be used to change the log filename, which by default is called `asm21toc.log`.

`-r`*level* set VCG rendering level

The flow graph rendition produced by `asm21toc` may be examined using the VCG tool. Within VCG, interactive commands may be used to hide or display different classes of edges and to expand or fold away parts of the graph. Completely unfolding a large graph is tedious, so the `-r` option is supplied to modify the initial visibility and folding attributes of the graph. The available rendering levels are shown in Table 2.2. The default rendering level is 2. More information on using VCG with `asm21toc` may be found in Chapter 3.

`-S` print summary symbol table statistics

`asm21toc` uses hash coded symbol tables of conventional design. The `-S` option prints out a histogram of bucket utilisation frequencies and the mean bucket utilisation figure for each table. Note that the scanner uses one table internally to hold the ADSP-21xx keywords and statistics for that table are printed too.

`-s` echo each scanner symbol as it is read

When debugging scanners it can be very helpful to get a diagnostic dump of each lexeme as it is passed to the parser. This option produces one line of output *per* lexeme and is only recommended for detailed debugging since it generates a lot of output.

`-T`*size* text buffer size in bytes for scanner

`asm21toc` allocates a fixed text area of 1Mbyte at initialisation time. If the sum of all the variable names in the translated application plus the sum of all the source file lengths in the application exceed 1Mbyte then this buffer will overflow. The `-T` option may be used to override the default text buffer size for the running parser. Any valid 32-bit unsigned number may be specified. If an out-of-memory message is delivered by the operating system, rather than by `asm21toc` then, of course, a machine with more memory must be used.

`-t`*width* tab expansion width

When listing source lines it is important that tabs are correctly expanded or the user's formatting may be lost. `asm21toc` supports fixed tab stops every n characters where n defaults to 8, but may be set to some other value with a `-t` option.

`-V`*filename* dump derivation tree in VCG format

The first phase of `asm21toc` generates an intermediate description of the application to be translated in the form of a tree. This tree can be output in a textual form that is suitable for reading into the VCG compiler graph visualisation tool [San95]. By default this file is named `rdparser.vcg`, and is similar in form to that produced using a `-r0` option. More information on using VCG with `asm21toc` may be found in Chapter 3.

`-v` set verbose mode

In verbose mode `asm21toc` reports its version number, the number of CPU seconds used during translation and the amount of dynamic memory allocated.

Chapter 3

Documentation outputs: the log and graph files

`asm21toc` is intended to aid programme comprehension as well as simply translating an application. To this end, two documentation files are produced. The `log` file is a text file that contains cross reference and summary control flow information. The graph file is graphical file that may be displayed using the VCG utility. The detailed form of the graph file is specified using command line switches, but its purpose is to display detailed control and data flow information about the application being translated.

3.1 The log file

The log file `asm21toc.log` contains symbol table information and sketch control flow diagrams for each module in the application.

The layout of the file is:

1. file list in alphabetical order,
2. global symbol cross reference table,
3. for each file in the file list:
   (a) local symbol cross reference table, and
   (b) control flow sketch.

3.2 Symbol table handling in `asm21toc`

Immediately after parsing, `asm21toc` *reconciles* symbol declarations and usages in a process that is equivalent to that performed by the `ld21` linker when applications are being built. The contents of the log file show the result of this reconciliation.

`asm21toc` creates a symbol table to hold global identifiers, and a local symbol table for each input source file. The global table is initially loaded with identifiers corresponding to the ADSP-21xx internal registers. During parsing of the source files, whenever a new ADSP-21xx source file is encountered, a

Attribute	Usage
Label	Label for instruction or directive
Macro	Macro name (not supported by `asm21toc`)
Module	Module name
Register	ADSP-21xx internal register
Constant	Defined in `.const` directive
Variable	Defined in `.var` directive
Port	Defined in `.port` directive
Segment	Defined in `.segment` directive
External	Defined in `.external` directive

Table 3.1 Label types

Attribute	Defining context
External	`.extern` directive
Global	`.global` directive
Entry target	`.entry` directive
Call target	Operand of `call` instruction
Jump target	Operands of `jump` instruction
Do target	Label nominated in `do` instruction
Unused	Not used as an operand in any instruction

Table 3.2 Label attributes

separate symbol table is created to hold the labels and other identifiers local to that file.

When `asm21toc` encounters a label it checks the local symbol table to see if the label has already been added. If not, then a new symbol is created. Symbols have a *type* and a (possibly empty) set of *attributes*. The available values for the type, and the list of possible attributes is shown in Tables 3.1 and 3.2 respectively.

The type and attributes of a symbol may be changed after it is encountered for the first time. Essentially, the type of a symbol is specified by the context in which it is declared – macro name, external and so on, and the attributes are set by the contexts in which the label is *used*. For instance a symbol may be declared as `.external` but used as the target of a jump instruction, in which case it would have type *external* and attribute *jump target*. Of course, it is possible for a label to be used in several different contexts – as a jump target and as a call target, for instance which is why the attributes form a set.

Each symbol table record also contains a placeholder for a *size* expression which is used to hold the size of arrays declared with directives such as `.var my_var[30];` and a list of initialisation expressions from `.init` or `.const` directives.

3.3 Symbol resolution

After all the files have been parsed, the global symbol table contains only the predeclared labels, and the local symbol tables contain only the symbols encountered within their corresponding source file. The symbol tables are *resolved* in a process analogous to linking.

All symbol tables are traversed, and any labels with the *global* attribute are copied to the global symbol table. *External* labels are matched to the corresponding definition in the global table. If no matching definition is found a suitable global symbol is created. The attributes of the local symbol are then logically OR-ed with the attributes of the global symbol, and any label size and initialisation expressions in the local symbol are also copied up. This phase is referred to as label *promotion*.

After promotion is completed, the symbol tables are traversed again in a *symbol update* phase. For each *external* label, the attributes of the corresponding global symbol are logically OR-ed into the local symbol, and any size and initialisation expressions are copied down.

The net effect of these phases is to ensure that each symbol contains the union of all the attributes displayed by any instance of the symbol immediately after parsing.

3.4 The cross reference table

The log file contains a cross reference table for each `asm21toc` symbol table, that is one global table and as many local tables as there are source files. An extract from a cross reference table is shown in Figure 3.3. The labels are listed in alphabetical order along with their type and any attributes they carry. The cross reference information includes the declaration line (File number/line number) and a list of line numbers on which the label is used.

3.5 The control flow sketch

`asm21toc` includes extensive support for the construction of flow graphs which may be visualised using VCG as described later in this chapter. A complete flow graph for an application can sometimes be a little overwhelming, so the log file includes *control flow sketches* which can, of course, be viewed using a simple text editor.

In essence, the control flow sketch is a list of the instructions in the ADSP-21xx source file that can affect control flow. A complete example for a bootstrap monitor that used the synchronous serial ports to communicate with an RS-232 terminal is shown in Figure 3.1.

The following parts of the ADSP-21xx source file are printed within the control flow sketch: all other detail is suppressed:

1. labels, with their attributes;

2. `do` statements;

```
Control flow sketch for module 'primary_bootstrap'

----------------------------------------
boot21: (Entry)
do polling until forever
polling: (Do target)
call txchar
do progreadloop until ce
call rxchar
call rxchar
call rxchar
progreadloop: (Do target)
call txchar
call dumper
jump 0
----------------------------------------
rxchar: (Call target)
do rxbits until ce
if eq jump rxl1
rxl1: (Jump target)
rxbits: (Do target)
return from subroutine
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
----------------------------------------
txchar: (Call target)
do txbits until ce
if eq jump txl1
txl1: (Jump target)
txbits: (Do target)
return from subroutine
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
----------------------------------------
txhex4: (Call target)
do txhex4loop until ce
call txchar
txhex4loop: (Do target)
return from subroutine
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
----------------------------------------
dumper: (Global) (Call target)
do dumploop until ce
call txhex4
call txchar
call txhex4
call txhex4
call txchar
call txchar
dumploop: (Do target)
return from subroutine
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

Figure 3.1 A control flow sketch

```
Local symbols for module 'primary_bootstrap' (file #1 '..\asm21toc\vaptest.dsp')

            Name   Type   Defined _____Used on lines_____  Attributes
       baud19200 constant  1/12                            (Unused) Value 19
        baud9600 constant  1/13     14                     Value 39
        baudrate constant  1/14     41                     Value baud9600
          boot21    label  1/33     30                     (Entry target)
          dumper    label  1/112    28   55                (Global) (Call target)
        dumploop    label  1/122   114                     (Do target)
         polling    label  1/37     35                     (Do target)
primary_bootstrap   module  1/1                             (Unused)
     progreadloop    label  1/51     47                     (Do target)
          rfsdiv constant  1/24                            (Unused) Value rfsdiv0
         rfsdiv0 constant  1/16     24                     Value 0X3FF4
         rfsdiv1 constant  1/19                            (Unused) Value 0X3FF0
          rxbits    label  1/81     74                     (Do target)
          rxchar    label  1/66     48   49   50           (Call target)
            rxl1    label  1/79     77                     (Jump target)
         sclkdiv variable  1/24     41                     Size 1
        sclkdiv0 constant  1/16     24                     Value 0X3FF5
        sclkdiv1 constant  1/19                            (Unused) Value 0X3FF1
          scntrl variable  1/24     40   59   67           Size 1
         scntrl0 constant  1/16     24                     Value 0X3FF6
```

Table 3.3 A cross reference table

Annotation	Usage
`----------------------------------------`	Function header
`^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^`	Return
if <cond> return from subroutine ----^	Conditional return

Table 3.4 Control flow sketch annotations

3. **jump** statements;
4. **call** statements;
5. **return from subroutine** statements;
6. **return from interrupt** statements; and
7. conditional statements.

Some annotations are inserted to make the resulting sketch more readable. The idea is to break the code up into separate functions, insofar as that is possible. A line of underscores is inserted before any label that is a call target. These correspond to function entry points. A line of uparrows is added after **rts** and **rti** statements to indicate a function exit point. Conditional returns are followed by an inline annotation, as shown in Table 3.4.

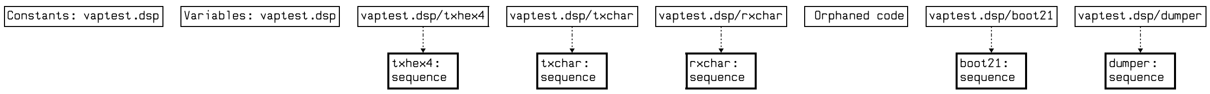

Figure 3.2 A completely folded, hierarchical control flow graph

3.6 The graph file and VCG

A *control flow graph* is closely related to the flow diagrams that were the basis of program design before structured programming languages were popularised. Flowgraphs are a powerful analysis tool which can aid program comprehension, especially in jump-intensive code. `asm21toc` renders flowgraphs into a textual format that can be visualised using the VCG tool. This chapter provides an overview of using VCG with `asm21toc`. More detail on VCG can be found in the accompanying manual.

`asm21toc` creates *primitive* flow graphs immediately after the symbol table has been reconciled. A primitive control flow graph is made up of nodes and edges: graphically, nodes are represented as boxes, and edges as lines between them. There are several different kinds of nodes and edges which are drawn using different colours when VCG displays the graph.

If control flow analysis has been enabled with the `-C` command line option, `asm21toc` attempts to *reduce* the primitive control flow graph by detecting patterns that correspond to high level constructs such as `if-then-else` and `while-do`. Of course, these structures can be nested, so in general the result of control flow analysis is a *tree* of sub-graphs each representing a high-level control construct. If the reduction process is completely successful, then each function should be reduced to a single hierarchy control flow node.

Figure 3.2 shows an example hierarchical control flow graph. The functions are listed along the top line along with boxes which hold the declarations of program variables and constants. Beneath each function is a hierarchy node which represents the code within the function.

There is also a box for *orphaned code*. This is code that is unreachable from the entry point declared using the `-e` command line option. Typically, orphaned code comprises either disabled debugging code, or interrupt service routines.

VCG can be used to progressively display more and more detail of the control flow graph. This is important because navigating a large graph is hard, and it is a good idea to only keep on-screen the necessary level of detail. Figure 3.3 shows the effect of completely unfolding the `txhex4` function. Here we see hierarchy nodes with their associated internal graphs attached. The hierarchy nodes have labels such as `sequence` and `do-while` which describe the kind of control flow structure they represent. Primitive control flow nodes are labeled with the line of ADSP-21xx code that they correspond to. When viewed in colour, distinguishing between the classes of node is straightforward because the hierarchy nodes are cyan and the primitive nodes are green.

When `asm21toc` is run without the `-C` command line option, the resulting control flow graphs are *flat*. Flat graphs probably give the best overall view of

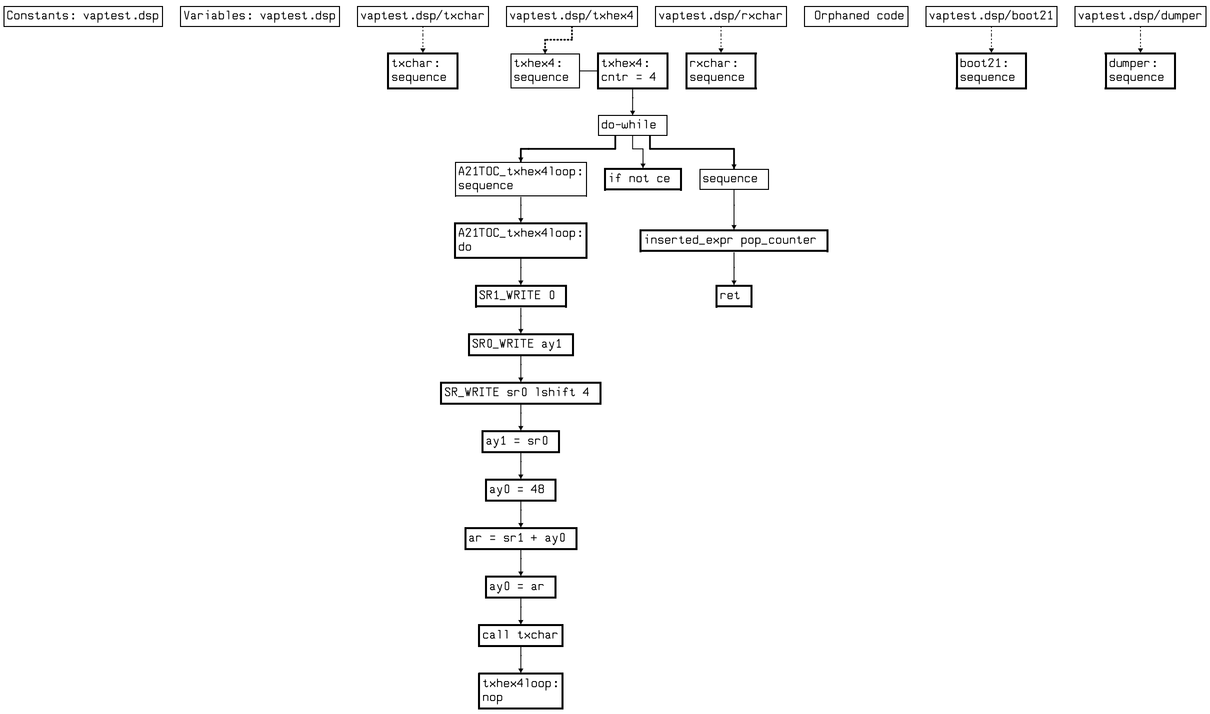

Figure 3.3 A partially unfolded hierarchical control flow graph

an application, but can be very large. The complete flat graph for a small boot ROM is shown in Figure 3.4.

3.6.1 Control flow nodes and expression nodes

There are four categories of nodes:

1. header nodes (black or magenta);
2. primitive control flow nodes (green);
3. hierarchical control flow nodes (cyan);
4. expression nodes (black)

Each function is given a header node which is always drawn at the top of the graph. Below the header node, a chain of primitive or hierarchical control flow nodes represents the sequence of actions performed by that function. Each control flow node has, hidden within it, an *expression tree* which shows the detail of the action performed at that point. Normally this expression tree remains hidden since the label on the control flow node contains the same information.

The primitive control flow graph is constructed by converting each line of the ADSP-21xx source into a tree of black expression nodes which then has a green control node attached to its root. The control nodes are then connected in the same sequence as the corresponding lines of code except that when a

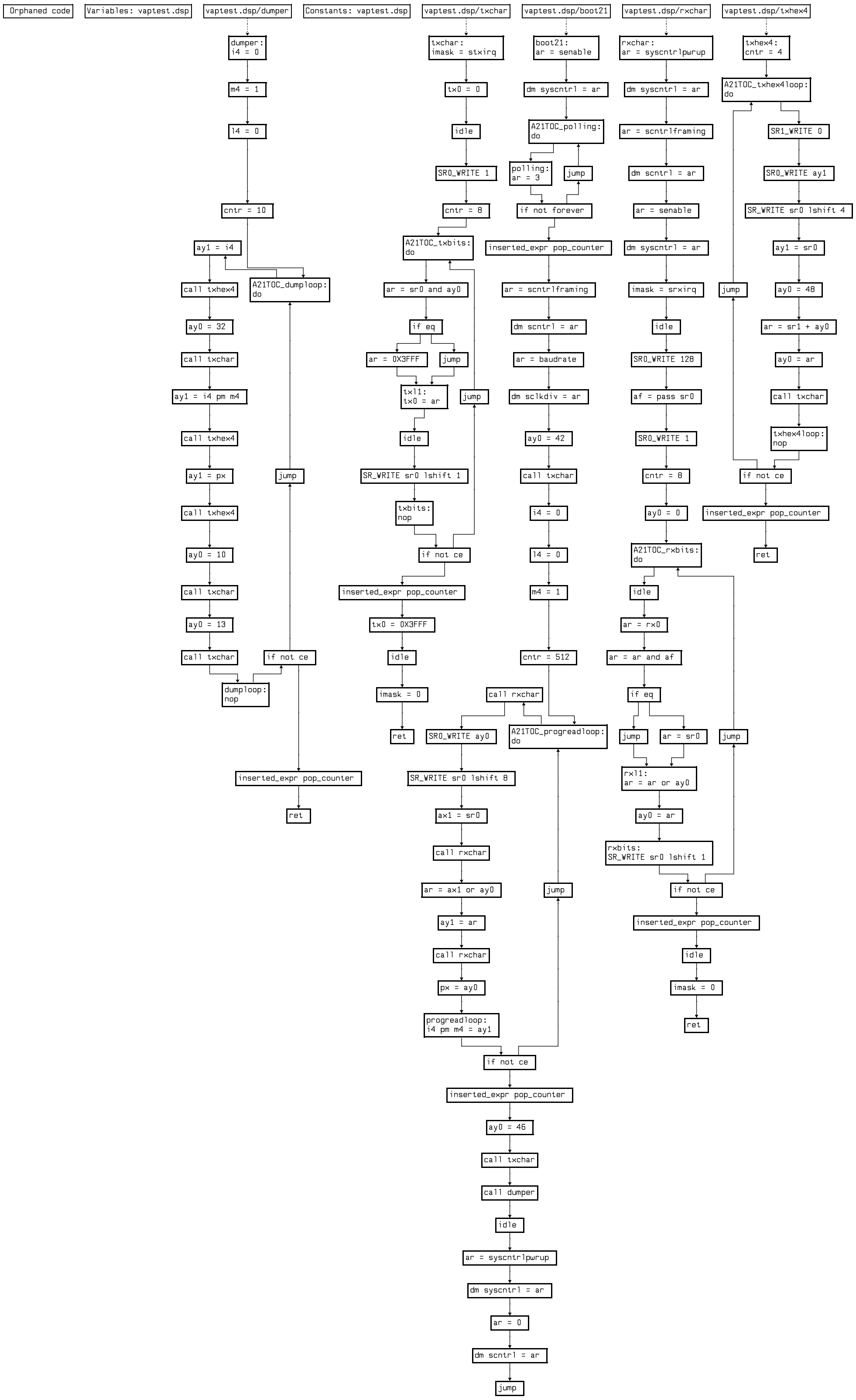

Figure 3.4 A flat control flow graph

Level		Output
0	Literal	The raw datastructures are rendered
1	All unfolded	The complete application is unfolded
2	Expressions folded	Only the control flow graph nodes are shown
3	Sequences folded	Only the nodes with multiple entries or exits are shown
4	Functions folded	Only the function header nodes are shown

Table 3.5 Graph rendering levels

jump is encountered, the edge leaving the jump node is redirected to the target of the jump.

A conditional instruction (an `if` statement) has two edges leaving the corresponding control node. The green edge is taken if the condition associated with the `if` statement is true and the red one if it is false.

3.6.2 Hierarchical control flow nodes

If the control flow analyser has been enabled with a `-C` command line switch then a *hierarchical* control flow graph is constructed by identifying high level control flow constructs such as `if-then-else` and `while-do`. The parts of the primitive graph corresponding to the high level graph are rewritten into a new hierarchy control flow node. The original primitive nodes in the control structure are not deleted: instead they are kept 'inside' the hierarchy node and can be displayed and re-hidden using the VCG *unfold* and *fold* commands, as described below.

3.6.3 Classes of edges

Normal edges are the same colour as the node they point to with the single exception of the red edge that leaves a primitive `if` node. There are two other classes of edges which are normally hidden but may be displayed using VCG's Expose/Hide edges command.

Call edges are coloured blue and connect `call` instructions to the labeled instruction that marks the start of a function. *Substitution* edges show opportunities for variable substitution that will be exploited if the `-E` command line switch is used.

3.7 Rendering levels

Completely unfolding a large graph is tedious, so the `-r` option is supplied to modify the initial visibility and folding attributes of the graph. The available rendering levels are shown in Table 3.5. The default rendering level is 2.

3.8 Displaying a function call graph

A function call graph shows which functions are called by other functions. It can give a useful overview of the organisation of an application. The **asm21toc** control flow graph can be folded into a function call graph by folding all function blocks away and then switching on the call edges. This can either be done manually within VCG, or the following command line options can be used to generate such a graph directly from **asm21toc**:

`asm21toc -r4 -c` *file*

3.9 Using VCG

Full details on the VCG tool may be found in the accompanying manual. We present a summary of the more important commands here.

Start VCG by issuing the command

```
vcg asm21toc.vcg
```

With large files there may be a delay whilst the graph layout is calculated.

Within VCG, the display can be reloaded from the disk file by pressing `G` at any time. The cursor keys may be used to move around the displayed graph. The *Scale* | *maximal aspect* command rescales the display so that the whole graph fits on screen. *Scale* | *normal* returns the display to its default scaling.

The *Folding* | *Unfold subgraph* and *Folding* | *Fold subgraph* commands are used to reveal and hide hierarchical nodes. In each case, after selecting the command, point to the node to be processed and then click with the left button. Multiple nodes may be selected in this way. After selection is complete, click with the right button to perform the action.

An uncomfortable feature of VCG is that it always attempts to draw the graph in a way that is attractive to the eye regardless of the previous positioning of the nodes. As a result, the display can shift around significantly as nodes are folded or unfolded. There is no way, unfortunately, to change this behaviour.

The *Folding* | *Expose/hide edges* command opens a dialogue box with three tick boxes for the three classes of edge which are:

1. normally visible edges,
2. substitution edges, and
3. call edges.

Clicking on the tick boxes may be used to enable and disable the various classes of edge.

3.10 Printing a graph

The *File* | *Print/export graph* command opens a complicated dialogue box that may be used to send rendered versions of the graph to a variety of output devices in a variety of formats. For straightforward printing to the default Windows printer, select **Printer (5)** from the middle column of radio buttons near the top of the box and then **Maxspect** from the column of radio buttons near the bottom of the box. This scales the graphic to fit the paper. Then click **OK**. A printer preview display opens (which is not always accurate). Click **Print** and go through the normal printer dialogue as appropriate for your Windows printer, and then click **Close** to remove the print preview display.

Within the VCG print/export dialogue box, sliders may be used to change the scaling of the image, and large graphs can be split onto multiple printer pages. Output can also be sent to a file for inclusion in other documents using a variety of graphics file formats.

Chapter 4

Run time messages

This chapter gives a comprehensive list of asm21toc's error messages listed in alphabetical order by severity level. asm21toc messages are printed in the form:

```
******: <severity> - <message>
```

where <severity> can be one of

Fatal	Fatal error leading to immediate termination of the run
Error	Error in input format or internal structures
Warning	Warning that output should be treated with care
(empty)	Informational message

and <message> is one of the following messages.

4.1 Fatal error messages

during substitution, CFG node <n> has no expression out-edges

An internal error has occurred. Please submit a bug report.

during substitution, CFG operator node <n> has no right hand side

An internal error has occurred. Please submit a bug report.

errors detected in source file

This message is issued at the end of the parsing phase if syntax errors have been reported. Later stages of translation are aborted, and no output files are produced.

internal error: expecting = node under const node

An internal error has occurred. Please submit a bug report.

internal programming error: label not found in labels table

An internal error has occurred. Please submit a bug report.

`unable to open file '`*filename*`' for write`

`asm21toc` was unable to open a file for output. This may be because a write-protected file of the same name already exists, or because the disk is full or because the path to the file does not exist.

`unable to open file '`*filename*`' for read`

`asm21toc` was unable to open a file for input. This may be because the file is protected against reading or because the file does not exist.

4.2 Error messages

`CFG out edge with count of` *count* `detected`

An internal error has occurred. Please submit a bug report.

`constant '`*label*`' not found in symbol table`

An internal error has occurred. Please submit a bug report.

`count nesting level out of range 1..4: value` *count*

More than four nested levels of `asm21toc do` statement were found.

`during CFG construction, call of non-label` *string*

An `asm21toc call` instruction with an expression, rather than a simple label name was detected. Typically this error is caused by calls to absolute addresses. Use an `asm21 .const` expression to define a label with the expression as its righthand side.

`during CFG construction, call of undefined label '`*label*`'`

The target of a `call` instruction was not defined.

`during CFG construction, jump to non-label`

An `asm21toc jump` instruction with an expression, rather than a simple label name was detected. Typically this error is caused by calls to absolute addresses. Use an `asm21 .const` expression to define a label with the expression as its righthand side.

`during CFG construction, jump to undefined label '`*label*`'`

The target of a `call` instruction was not defined.

`during substitution, found unexpected operator '`*label*`' at node <n>`

An internal error has occurred. Please submit a bug report.

`Functions '`*label*`' and '`*label*`' share node (`*number*`:<n>)`

ANSI-C functions may only have a single entry point, but it is, of course, possible to define multiple entry functions in ADSP-21xx assembly language. These will be translated as separate functions, but this message will be issued for each shared expression.

`hierarchy node <n> (`*label*`/<n>) has expression out edge with count of <n>`

An internal error has occurred. Please submit a bug report.

`hierarchy node <n> (`*label*`/<n>) has NOT HIERARCHY kind`

An internal error has occurred. Please submit a bug report.

`hierarchy node <n> (`*label*`/<n>) has unknown kind`

An internal error has occurred. Please submit a bug report.

`in file `*label*`, entry or call target '`*label*`' has no control tree: skipping render`

A `call` or `jump` instruction references a label that has not been defined. The usual cause of this error is the omission of one or more source files in the application from the `asm21toc` command line.

`in function `*func*`, `*label*` is used without a prior definition at node `*number*

The dataflow analyser has detected a variable which is used on the right hand side of an assignment before it appears on the left hand side, that is an uninitialised variable.

`last statement in DO loop does not have exactly one successor: skipping loop tail insertion`

If the last statement in an ADSP-21xx `do` loop is an `if` statement then correct translation is not guaranteed: this message is issued as a warning.

`local label '`*label*`' has been promoted but global symbol is missing`

label is used in an `.external` context but has no corresponding `.global` declaration.

`multifunction instruction clause <n> is not a simple data transfer:`

An invalid multifunction instruction has been detected with more than one ALU operation in it.

`multifunction instruction includes more than three clauses:`

An invalid multifunction instruction has been detected with more than three comma-delimited clauses.

`promoted symbol '`*label*`' is absent from global scope`

Required global symbol *label* has not been declared.

`include file '`*label*`' not found`

`asm21toc` was unable to open a file referenced in an `.include` directive.

`module contains macros - expand using asm21pp`

`asm21toc` does not perform macro expansion. The `asm21 -p` option may be used to perform macro expansion on files before passing them to `asm21toc`.

`redeclaration of label '`*label*`'`

Symbol *label* has been declared more than once.

4.3 Warning messages

`constant '`*label*`' is used as absolute DAG address: converting to variable`

Normally, labels appearing in ADSP-21xx assembly `.const` directives are rendered as manifest constants using `#define` directives in the ANSI-C output file. Some constants are used to specify memory mapped registers and other memory locations. Where `asm21toc` encounters a `.const` label being used this way it suppresses the `#define` directive and generates a variable declaration instead.

`external symbol '`*label*`' has multiply defined constant value`

An external `.const` label has been declared more than once.

`external symbol '`*label*`' has multiply defined initialisation`

An external `.var` label has been initialised more than once.

`external symbol '`*label*`' has multiply defined size`

An external `.var` label has been declared as an array more than once.

`Found if in non-matching schema at node <n> (`*label*`:<n>)`

An irreducible control flow construct has been detected.

function *filename/label* contains non-reducible structure

An irreducible control flow construct has been detected.

function '*label*' has no control tree: skipping control flow analysis

A jump instruction references a label that has not been defined. The usual cause of this error is the omission of one or more source files in the application from the asm21toc command line.

function '*label*' has no control tree: skipping normalisation

A jump instruction references a label that has not been defined. The usual cause of this error is the omission of one or more source files in the application from the asm21toc command line.

global symbol '*label*' has multiply defined constant value

A global .const label has been declared more than once.

global symbol '*label*' has multiply defined initialisation

A global .var label has been initialised more than once.

global symbol '*label*' has multiply defined size

A global .var label has been declared as an array more than once.

if node <n> (*label*:<n>) has non-disjoint root/then/else nodes

An irreducible control flow construct has been detected.

if node <n> (*label*:<n>) has non-disjoint then and else branches

An irreducible control flow construct has been detected.

non-label '*label*' used as address - cannot convert to variable

An expression has been used inside a dm() or pm() memory reference rather than a simple variable. Typically this error is caused by references to absolute addresses. Use an asm21 .const expression to define a label with the expression as its righthand side.

orphaned code at *label*:<n>

The dataflow analyser has detected code that does not belong to any function. Typically this is either unused debugging code or functions that are called asynchronously as interrupt handlers.

`symbol '`*label*`' is declared as .external but does not exist in global scope: skipping attribute promotion`

A label has been declared in a module and also marked as `.external`. The ADSP-21xx assembler accepts this syntax, although it is strictly speaking a programmer error. `asm21toc` issues a warning message for completeness – no corrective action is required.

4.4 Informational messages

`<n.m> CPU seconds used`

In verbose mode (the `-v` command line switch) `asm21toc` reports the amount of CPU time used during translation.

`Heap manager calloc'ed` *count1* `bytes, malloc'ed` *count2* `bytes and realloc'ed` *count3* `bytes`

In verbose mode (the `-v` command line switch) `asm21toc` reports the amount dynamic memory activity.

Chapter 5

asm21toc internals

This chapter describes how `asm21toc` works and some restrictions on the kind of code that can be translated.

5.1 Parsing

Parsing is the process of scanning a text and identifying structural relationships within phrases. In the context of computer languages, parsing usually means the parsing of *Context Free Languages* for which a large body of theory is available. Since well-understood mathematical models of these kinds of languages have been constructed, it is possible to automate some phases of the translation process. Tools that help in this process are called *translator generators* or, less accurately in our case, *parser generators*.

`asm21toc` is implemented in `rdp`, a translator generator written by the authors and now in its seventh version. `rdp` comes with extensive tutorial documentation which provides a good introduction to the field as a whole as well as detailed information on using `rdp` [JS97d, JS97a, JS97b, JS97c]. Postscript versions of these documents may be fetched *via* the `rdp` web page at

```
http://www.cs.rhbnc.ac.uk/languages/rdp
```

5.1.1 The acceptable language

The `asm21toc` parser recognises a language that is, strictly speaking, larger than the set of valid ADSP-21xx assembler statements. `asm21toc`, for instance will accept an assembler statement such as

```
ar = i0 + ar;
```

even though `i0` and `ar` are not valid input operands to the adder in the ADSP architecture. During parsing, `asm21toc` treats all registers as equivalent and does not attempt to match pipeline registers to functional units. It appears that the `asm21` assembler works similarly: error messages concerning operand-operator pairing come from `asm21`'s second phase, not from the parser. `asm21toc` does not even need to check in after parsing: it simply goes ahead and performs the translation since, of course, at high level there is no restriction on the names of ANSI-C variables that may be used with ANSI-C operators.

The language is also widened in other ways. The statement

```
ar = i1 OR i0 + ar
```

is quietly translated to `ar = i1 | i0 + ar`; even though no such operation could be directly executed by the ADSP-21xx data path.

The reason for broadening the language is to simplify both the implementation and the testing of the translator. The justification for not strictly following the ADSP-21xx assembler language is that all applications can be checked for ADSP-21xx validity by passing the source through the ADSP-21xx development tools, and it is a precondition of translation that the source has been so checked. Nevertheless, the tool is consistent in that it will either translate or give error messages: the back end stages can handle anything that the parser accepts regardless of whether it is valid ADSP-21xx code.

5.1.2 Preprocessing

The ADSP-21xx assembler allows macros to be defined, with two different syntaxes: ANSI-C style and its own more functional style. `asm21toc` does not process macros directly. Instead, macros must be expanded by issuing the command `asm21 -p` for each file in the application, including files included using the `.include` directive. The output files (which have default filetype `.app`) may then be supplied to `asm21toc` for translation. The language of the `.app` files is substantially the same as the ADSP-21xx assembly language but includes some annotations which `asm21toc` parses and discards. This constitutes another widening of the `asm21toc` language over that accepted by the ADSP-21xx assembler.

5.2 Parser output

If the parser finds syntax errors, it reports them and terminates the translation process after all files have been read but before translation proper begins.

Assuming that no syntax errors were found, the output of the parser takes the form of three structures:

1. a *Reduced Derivation Tree* (RDT) which represents the grammatical structure of the source files;

2. two or more *symbol tables* which are directories holding the names of labels, constants, variables and so on in a form that allows them to be rapidly looked up and modified; and

3. a *comment buffer* which contains a linked list of all the comments in the source files, tagged with their file number and line number within that file.

```
.module test;

.entry main;

.const five=5;
.var/dm z[five];

.init z: 3,7,9,10;

main:

      i0 = ^z;
      m1 = 1; m0 = 0;
      cntr = five;
      ax0 = 0;

      do flip until ce;
      ay0 = dm(i0,m0);
      if gt ar = ax0 - ay0;
flip:
      dm(i0,m1) = ar;

      rts;

.endmod;
```

Figure 5.1 Test source file

These structures are all cross-linked: the RDT nodes contain links to the symbol table and the comment buffer to enable information to be quickly looked up concerning individual lines of code. Similarly, identifiers in the symbol table are linked to that part of the RDT that represents their declaration.

Some parts of the translation process work by traversing the symbol table and then looking at sub-parts of the RDT as necessary, conversely some translation tasks work by traversing the RDT which is necessary if, for instance, the ordering of individual expressions is important.

The `rdp` manuals referenced earlier contain extensive documentation on the structure of RDT's, the way symbol tables are implemented and the library support for traversing RDT's and symbol tables. Here, we simply give an example of the output for illustrative purposes. Figure 5.1 shows a small test program that converts values in an array to their negative absolute form. In Figure 5.2 we see the RDT output by the parser for that program. The RDT for any `asm21toc` translation can be examined by adding the -V option to the command line and then examining the RDT file (called `rdparser.vcg` by default) with the VCG tool as described in the previous chapter.

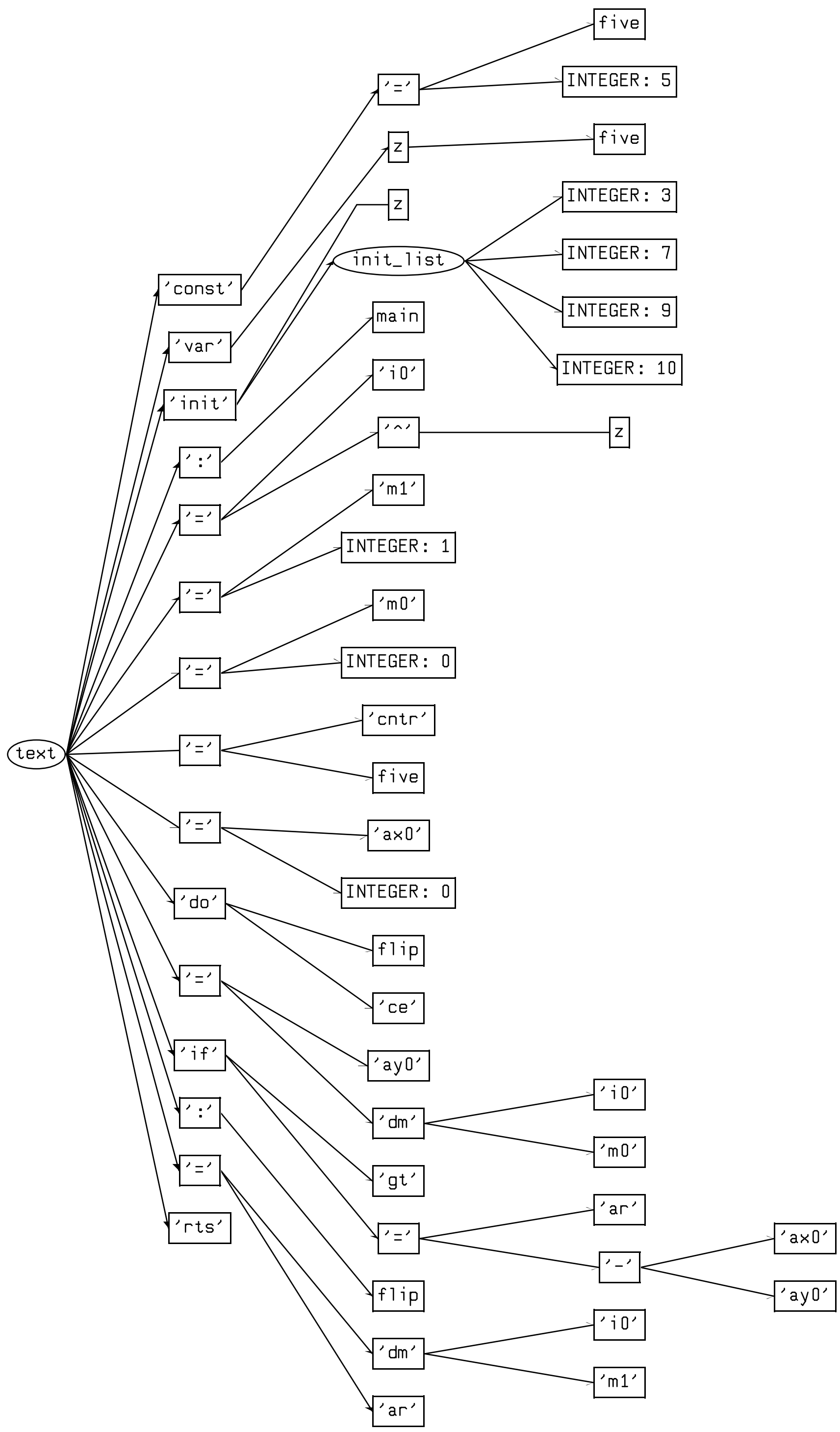

Figure 5.2 Test file RDT

The RDT is an almost complete rendition of the source program except that comments have been stripped out, some punctuation (especially instances of the statement terminator `;`) is discarded and lines of ADSP-21xx source which only carry information that can be held in the symbol table are also suppressed. Hence, the `.module`, `.endmod` and `.entry` directives have been suppressed from the RDT.

5.3 Symbol table processing

[1]Immediately after parsing, `asm21toc` *reconciles* symbol declarations and usages in a process that is equivalent to that performed by the `ld21` linker when applications are being built. The contents of the log file show the result of this reconciliation.

`asm21toc` creates a symbol table to hold global labels, and a local symbol table for each input source file. The global table is initially loaded with labels corresponding to the ADSP-21xx internal registers. During parsing of the source files, whenever a new ADSP-21xx source file is encountered, a separate symbol table is created to hold the labels and other identifiers local to that file.

When `asm21toc` encounters a label it checks the local symbol table to see if the label has already been added. If not, then a new symbol is created. Symbols have a *type* and a (possibly empty) set of *attributes*. The available values for the type, and the list of possible attributes is shown in Tables 3.1 and 3.2 respectively.

The type and attributes of a symbol may be changed after it is encountered for the first time. Essentially, the type of a symbol is specified by the context in which it is declared – macro name, external and so on, and the attributes are set by the contexts in which the label is *used*. For instance a symbol may be declared as `.external` but used as the target of a jump instruction, in which case it would have type *external* and attribute *jump target*. Of course, it is possible for a label to be used in several different contexts – as a jump target and as a call target, for instance which is why the attributes form a set.

The symbols also includes a placeholder for a *size* expression which is used to hold the size of arrays declared with directives such as `.var my_var[30];` and a list of initialisation expressions from `.init` or `.const` directives.

5.3.1 Symbol resolution

After all the files have been parsed, the global symbol table contains only the predeclared labels, and the local symbol tables contain only the symbols encountered within their corresponding source file. The symbol tables are *resolved* in a process analogous to linking.

All symbol tables are traversed, and any labels with the *global* attribute are copied to the global symbol table. *External* labels are matched to the corresponding definition in the global table. If no matching definition is found a suitable global symbol is created. The attributes of the local symbol are then

[1]This section is repeated from Chapter 3.

logically OR-ed with the attributes of the global symbol, and any label size and initialisation expressions in the local symbol are also copied up. This phase is referred to as label *promotion.*

After promotion is completed, the symbol tables are traversed again in a *symbol update* phase. For each *external* label, the attributes of the corresponding global symbol are logically OR-ed into the local symbol, and any size and initialisation expressions are copied down.

The net effect of these phases is to ensure that each symbol contains the union of all the attributes displayed by any instance of the symbol immediately after parsing.

5.4 Log file output

The log file (which contains symbol table, cross reference and control flow sketch information as described in Chapter 3) is output immediately after symbol table reconciliation but before modifications are made to the tree structures output by the parser.

5.5 Tree fixup steps

After symbol table resolution, a series of local modifications are made to the RDT. The shape of the RDT is ultimately governed by the syntax of the ADSP-21xx assembler, and there are a few situations in which re-ordering of the tree into a form that is more like that required by an ANSI-C compiler is desirable. Most of these transformations involve only changes to the attributes of a single node or the re-ordering a small sub-tree: the overall structure does not change significantly. Here is a list of the transformation passes, in the order in which they are performed.

1. *Set entry label as call target* – the entry label nominated with a `-e` label on the command line is located and marked in both the symbol table and the RDT as the initial function that should be called from `main()` in `a2c_main.c`.

2. *Remap promoted symbols in RDT* – links within the tree to labels which have been promoted to the global symbol table are remade to point at the new entry in the global table.

3. *Merge trees* – the parser outputs a single RDT for each source file in the application which are combined into a single tree in order of their appearance on the command line.

4. *Mark edge numbers and count nesting* – some later stages of processing do not preserve the left-to-right ordering of nodes in the tree, so each node carries an integer *child number* which is initialised with its position, so the leftmost node has child number 1, its immediate sibling 2 and so on. In the same step, nestings of `do` statements are checked.

5. *Rewrite non C monadic operators* – some monadic operators in the ADSP-21xx assembler such as `ashift` and `lshift` are remapped to diadic expressions.

6. *Fix up multiplier instructions* – assignments to the multiplier registers are tricky in that *clearing* a multiplier register sets the multiplier overflow bit but other operations do not. The parser cannot distinguish these cases. `asm21toc` maintains a set of potential register modifications in each node of the RDT which is modified on this pass to correctly reflect the semantics of the ADSP-21xx multiplier instructions.

7. *Fix up shifter instructions* – similarly, shifter instructions affect the arithmetic status bits in irregular ways that the parser cannot easily distinguish. On this pass the register modification sets are updated to correctly reflect the semantics of shifter instructions.

8. *Fix up monadic subtractions* – the diadic and monadic subtract operators also have different effects on the status bits which the parser cannot easily distinguish. On this pass the register modification sets are updated to correctly reflect the semantics of shifter instructions.

9. *Rewrite sub-register aliases* – the multiplier and shifter result registers are wider than 16-bits, and can be accessed in two different ways in both the ADSP-21xx instruction set and in the assembly language. In some contexts the entire register is referred to as `MR` or `SR`, and in others 8- or 16-bit slices are individually referred to as sub-registers `MR0, MR1, MR2, SR0` and `SR1`. A naïve translation of the assembly code would use different register names for all these but, of course, in practice `MR` and `SR` are not independent of their sub-registers. This is known as *aliasing*–`SR0` and `SR1` constitute partial aliases of `SR`. On this pass the tree is rewritten to remove these ambiguities.

5.6 Control Flow Graph construction

The RDT constructed by the parser is, as its name suggests, a tree, and thus can represent the order in which assembler statements are presented, but cannot fully represent the order in which statements may be executed. The execution of a loop, for instance, is cyclic and trees are acyclic structures. Nevertheless, the tree is a very useful representation for individual expressions because the *priority* and *associativity* of operators can be represented by the tree's shape.

If the translated high level code were to directly represent jumps with `goto` statements then simply traversing the tree would be sufficient, and indeed the first prototype of `asm21toc` worked that way. However, identification of high-level control structures, and the consequent elimination of `goto` statements from the output requires that control flow be directly represented in the tool's intermediate form.

After the RDT has been rewritten using the transformations described in the previous sections, a *primitive control flow graph* is constructed. It is made up of a network of control flow nodes, each of which contains an expression. In most cases, the expression corresponds to a single ADSP-21xx assembly statement. Conditional statements are represented by a node whose expression is the condition of the corresponding `if` statement with two edges leaving it: one points to another control flow node that has the body of the `if` statement within it, and the other points to the node for the line of assembly that comes after the conditional statement.

The primitive control flow graph is constructed by visiting each child of the root node in the RDT and creating a control flow node for it. These are connected in left-to-right sequence, except where a jump occurs in which case the out-edge of the jump node is connected to the jump target.

Some rewriting of the control flow graph then occurs. In particular, do loops are rewritten using jumps, and counter based loops have counter manipulation expressions inserted.

The raw, internal form of the primitive control flow graph for the example of Figure 5.1 is shown in Figure 5.3. The more normal graphical rendering of the primitive control flow graph is shown in Figure 5.4. Here the code is arranged by function with associated header blocks and, where possible, control is shown as flowing from the top to the bottom of the figure. The aim is to emulate a flow-chart like display.

5.6.1 Function normalisation

In ANSI-C, as in most high level languages, functions must have a unique entry point[2]. However, at machine level it is straightforward to write functions with multiple entry points: that it a section of code that may be entered through more than one `call` instruction. In `asm21toc` we identify functions by taking each call target in turn and then exploring all paths through the control flow graph from that entry point. The exploration of a path terminates when we either visit a node that we have already seen whilst exploring from the current entry point (which copes with loops) or when we encounter a return from subroutine instruction. Any node that appears on a path from more than one entry point is called a *shared node*, and its existence is reported *via* a warning message.

Some normalisation of shared nodes is performed. Conceptually we *clone* shared sections of code, so that a new function is created for each entry point. In practice, we use a slightly more compact representation, but the effect in the translated code is that separate, single-entry functions are written out.

5.7 Data flow analysis

Dataflow analysis is the tracing of relationships between the values a variable may take. A fundamental dataflow analysis is the establishment of *use-define* relationships. A variable is *used* if it appears on the right hand side of an

[2]FORTRAN 90 is perhaps the only example of a well known language to support multiple entry functions.

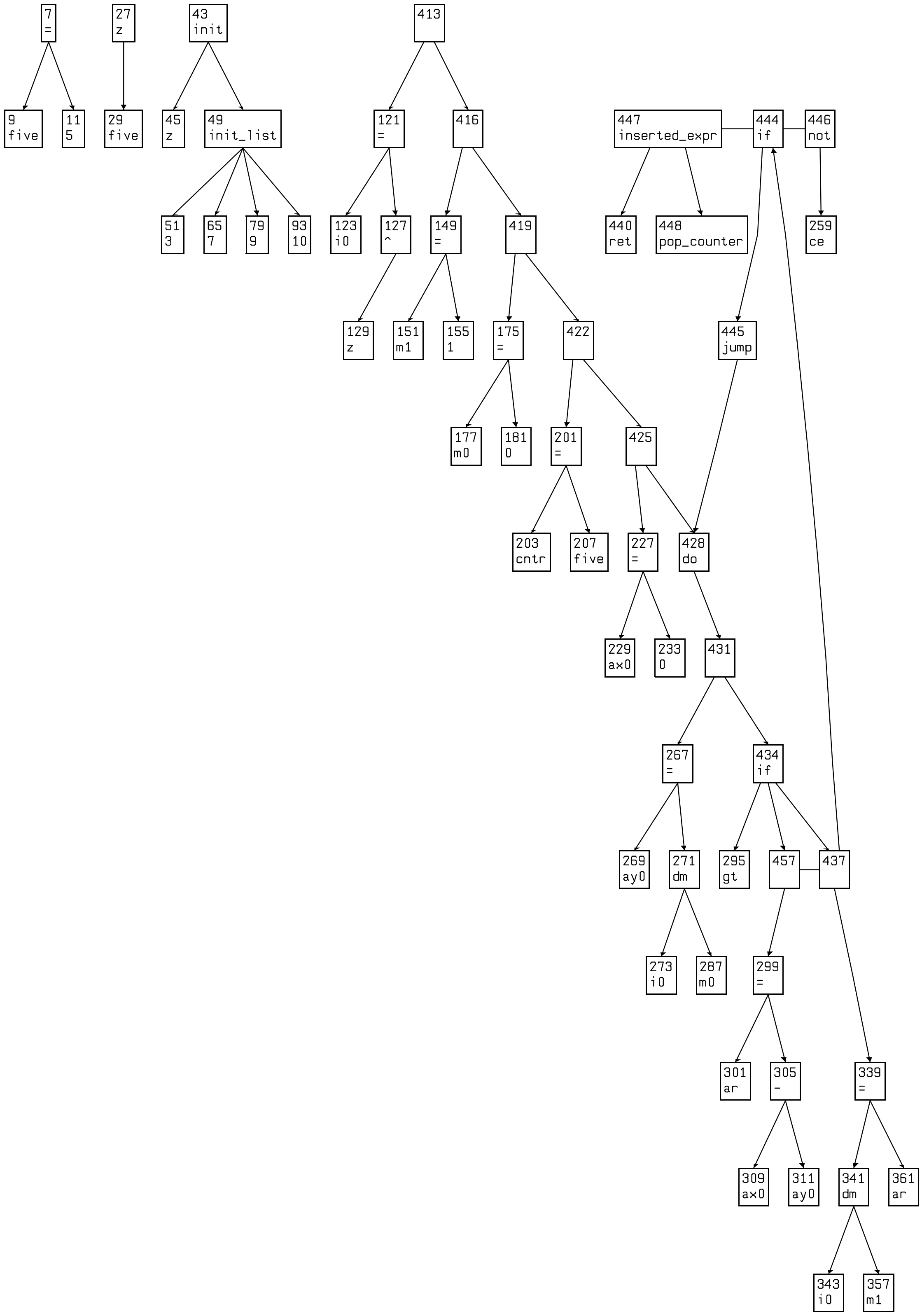

Figure 5.3 The raw form of the primitive Control Flow Graph

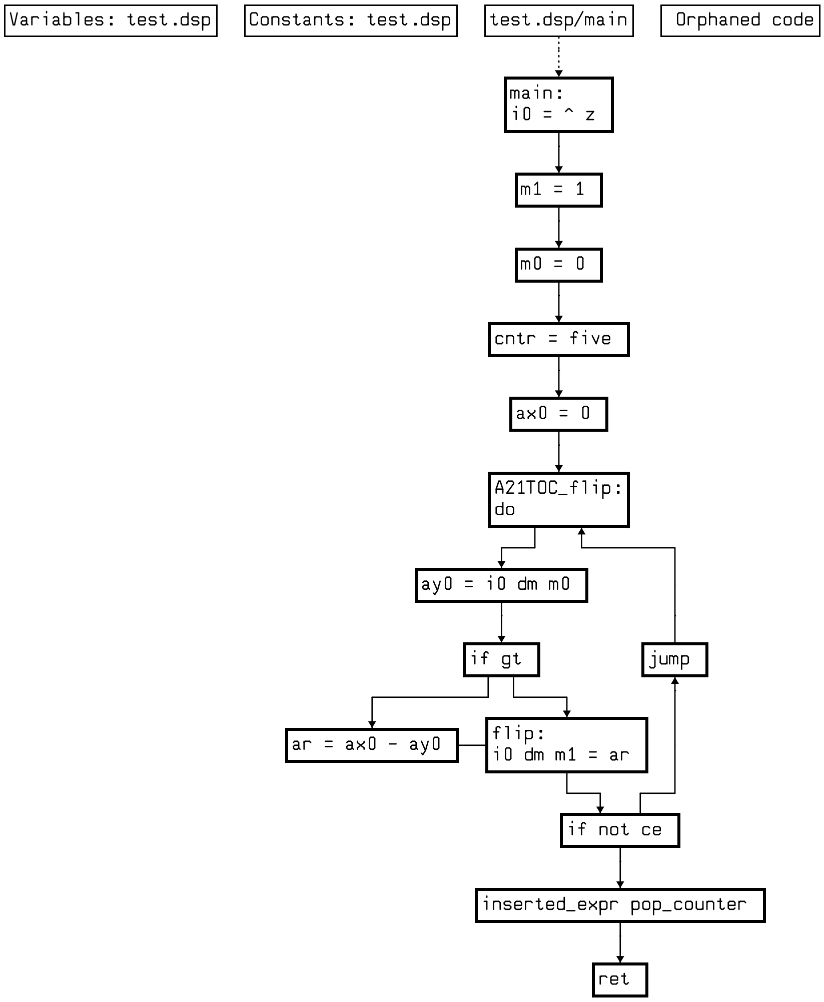

Figure 5.4 The user view of the primitive Control Flow Graph

assignment or any similar context in which the variable is *read*. A variable is *defined* when it appears on the left hand side of an assignment or is used in any similar context where it is written.

`asm21toc` performs an exhaustive use-define analysis which follows control between functions. This is called an *inter-procedural* dataflow analysis in the literature, and is rarely performed by compilers because the benefits compared to int*ra*-procedural analyses are small and the computation cost is high. Our situation is a little different: firstly we can afford to expend minutes or even hours of processor time on the translation since it is essentially a one-off exercise, and secondly it turns out that the existence of status bits and pipeline registers in the ADSP-21xx architecture cause unwelcome effects in the translated code if the dataflow analysis is incomplete.

The status bits in a conventional processor architecture are set as part of the execution of every arithmetic and logical operation. Typically there are between four and eight such bits, depending on the processor. The ADSP-21xx status bits occupy the `ASTAT` register, and a variety of instructions set them and use them. The problem for reverse compilation is that a naïve translator must assume that *all* status bit updates are important, whereas in reality the vast majority of them are soon overwritten as a side-effect of a later instruction and never actually used.

If all status bit updates are to be translated then the output high level programme will be primarily composed of status bit updates, since typically there are five updates for each arithmetic operation. Such code might be semantically correct but would be impenetrable to human readers since the underlying algorithm would be obscured. To be practical, the translator must identify the status bit updates that are used (for instance in a subsequent `of` statement) and suppress the rest.

5.7.1 Use, define, live and substitutable sets

Information on register usage and definition is represented directly within the control flow graph. The individual status bits are represented as separate variables rather than *via* the `ASTAT` register. Whilst the RDT is being built, `asm21toc` initialises two sets within each RDT node which represent register usage and definition. The elements of each set are the ADSP-21xx machine registers, and the sets are initially empty. The parser has encoded within it information on the usage and definition semantics of each operator and looks at the operands of the operator to decide which elements to add to the USE and DEFINE sets. There are a few cases that the parser cannot readily distinguish: these are fixed up during the tree rewriting phases as described above.

The dataflow analyser proper examines these USE and DEFINE sets to establish (i) whether a variable definition is ever subsequently used and (ii) whether a variable usage has exactly one corresponding definition point. The first allows `asm21toc` to suppress 'dead' expressions which include the vast majority of status bit updates, and the second enables expression substitution which may be optionally used to remove many references to machine registers from the translated code.

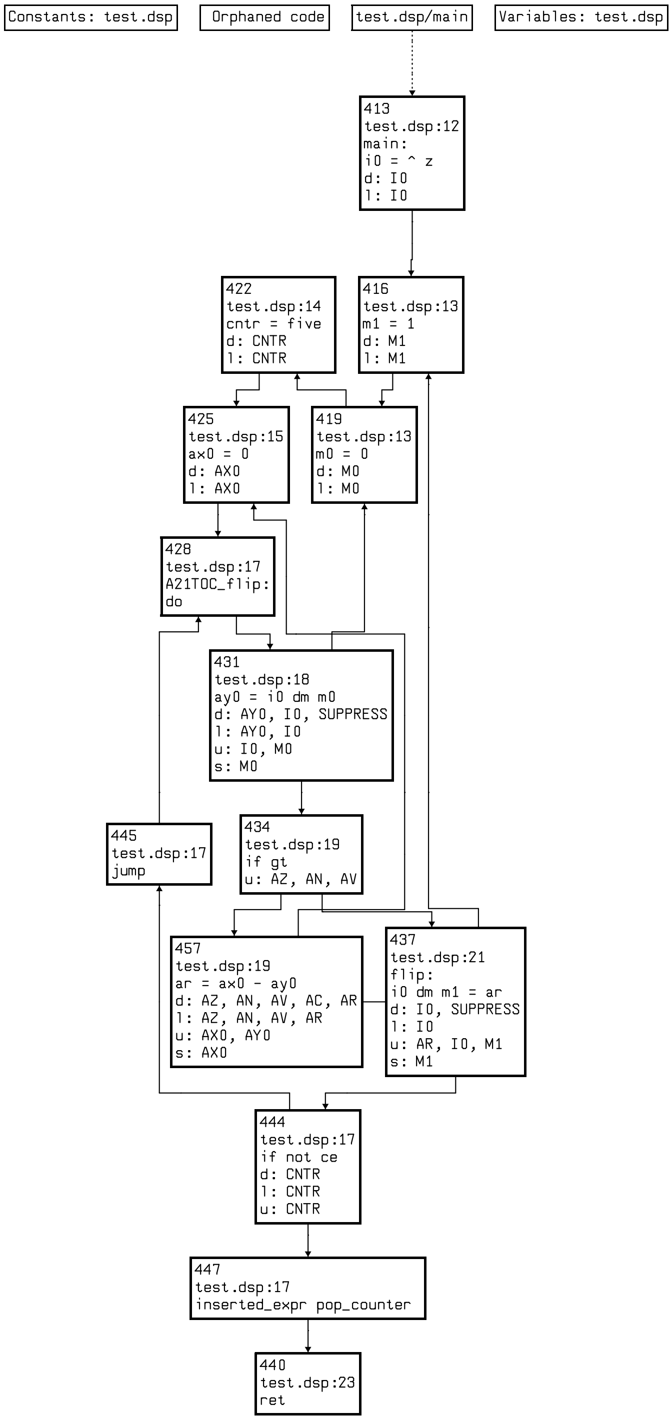

Figure 5.5 Primitive Control Flow Graph with dataflow information

The output of the dataflow analyser is in the form of a further two sets of registers that are attached to each node (the LIVE and SUBSTITUTABLE sets) and extra control flow graph edges connecting substitutable register usages to their corresponding definitions. The results of dataflow analysis for the test example of Figure 5.1 are shown in Figure 5.5 which was produced using the `-d1` command line option of `asm21toc`. Here we see that each node has a node number, a reference to the file and line number of its expression and listings of up to four sets of registers (`d:`, `l:`, `u:` and `s:` being the DEFINE, LIVE, USE and SUBSTITUABLE sets respectively). If a set is empty, it is not displayed. In addition, for each substitutable variable, a (yellow) back-edge is drawn from the use node to the corresponding display node: see for instance nodes 437 and 416 which use and define the `M1` register.

The basic dataflow information is always used by the translator to suppress dead expressions. Substitution is only performed when the `-E` command line option is invoked. The full analysis is always performed even if substitution is not requested since the liveness analysis requires a full analysis. This is important because deeply nested `if` structures and certain other kinds of nesting require analysis times that are exponential in the depth of the nesting, and there is no way to turn the analysis off. This is not a problem in practice because deep nestings are rare. We discuss the real-world occurrence of nested structures in one of the papers reproduced in Appendix A.

An important by product of the dataflow analysis is an analysis of register usage within functions. `asm21toc` distinguishes six cases.

1. A variable which is used before it is defined constitutes a read of an uninitialised variable.

2. A variable which is used within a function with matching definitions outside the function is an input parameter to that function.

3. A variable that is defined inside a function and then used outside it is an output parameter.

4. A variable that is defined inside a function and not subsequently used is dead.

5. Variables which are defined and used purely within the function are local variables.

6. Variables which are live at entry to a function but not used within the function are variables belonging to the calling context.

`asm21toc` places comments in the translated code containing this information for register definitions. At the start of each function, the input and output parameters are listed along with local variables and live global variables. Dead expressions are suppressed if they are assigned to status bits or marked with a comment warning that the result is never used. Uninitialised variables generate warning messages during dataflow analysis.

5.8 Control flow analysis

Control flow analysis is optional, and will be completely skipped if the `-C` command line option is not invoked. The purpose of control flow analysis is to remove `goto`'s arising from jumps in the primitive control flow graph, and replace with high level control flow structures such as `if-then-else` and `while-do`. The control flow analyser does this by looking for patterns in the control flow graph that match *schemas*.

A schema is a template for a control flow structure: for instance a node with two children each of which has a single out-edge pointing to the same successor can be written as an `if-then-else` statement. Similarly, a node with two children, one of which has a single out-edge that points back to the parent can be rewritten as a `while-do` loop. This rewriting process can continue recursively since high-level control flow constructs can be nested. Figure 5.6 shows the result of applying such an analysis to the program in Figure 5.1. Nodes with names such as `sequence` and `if-then` are hierarchy nodes. They appear in cyan when viewed in colour

In detail, great care must be taken over the design of schemas to ensure that rewritings (or *reductions* as they are more formally known) are applied unambiguously and in a convergent manner so that control flow graphs for functions can be reduced to a single hierarchy node. Functions which can not be completely subsumed into a single node are called *irreducible*. `asm21toc` makes no guarantees as to the correct hierarchical translation of irreducible functions: in such cases a warning message is output and the user must re-translate the function without the `-C` command line option and then substitute the primitive translation for the hierarchical one. This need only be done on a function-by-function basis: the other functions in the source file that *are* reducible will interwork correctly with the non-reduced one. The penalty, of course, is that the irreducible function will have `goto` statements in it. In practice, irreducible functions are rare. We give statistics on the occurrence of such functions in the papers reproduced in Appendix A.

5.9 Translated file output

After the analyses have completed, the actual output of the translated `.c` files is relatively straightforward. Output is performed in a the following series of passes over the local symbol table for each input source file.

1. Output constants using `#define` preprocessor directives.

2. Output variable declarations using the *size* expression in the symbol table to specify arrays.

3. Output variable initialiser function using the *init* expression lists.

4. Output function prototypes in alphabetical order.

5. Output functions in alphabetical order.

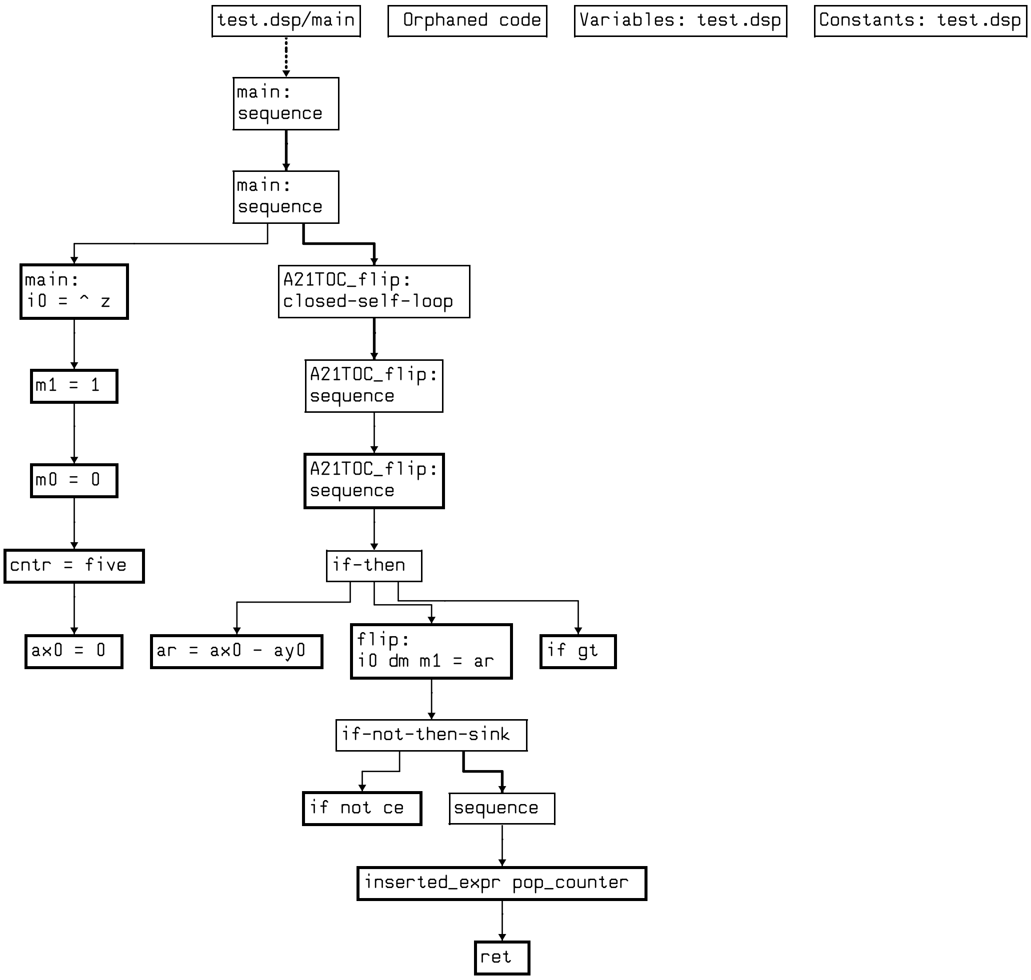

Figure 5.6 The unfolded hierarchical Control Flow Graph

During the final phase, the sub-part of the (possibly hierarchical) control flow graph referenced by the symbol table entry for the function name is traversed and ADSP-21xx syntax written out in ANSI-C syntax.

5.10 Restrictions on input files

Some types of ADSP-21xx program do not have straightforward translations into ANSI-C. The general strategy in `asm21toc` is to translate everything that may be handled safely and without significant run-time overhead and to issue warnings concerning untranslated structures. There is a grey area, and every user will have their own idea as to what constitutes safe and efficient code. The following subsections highlight difficulties associated with non-statically visible constructs; variable aliasing; processing modes; and coding idioms in each case describing the behaviour of `asm21toc` when such problems are encountered. The papers included in the appendix give more detail, in particular on the probability of encountering these problems.

5.10.1 Non-statically visible constructs

An aspect of a program is not statically visible if the dataflow and control flow analysers cannot make unequivocal predictions of runtime behaviour. All real programs contain many variables that are non-statically visible since, if the value of every variable were statically visible then there would be no point in running the program: the compiler could simply evaluate the result directly. A completely statically visible program is really a calculation on constant values. Any true variables whose values depend on runtime-supplied data are necessarily non-statically visible. This in itself is not a problem: `asm21toc` simply translates the calculations into calculations in ANSI-C. The difficulties arise when there are non-statically visible calculations in the ADSP-21xx application that do not have straightforward semantic equivalents in ANSI-C.

Consider, for instance, indirect jumps and calls that is transfers of control to a location whose address is calculated at runtime and held in a register. ANSI-C has only rudimentary facilities for describing run-time calculated transfers of control (function pointers and the `switch statement`) and in any case the dataflow analyser is essentially stopped in its tracks by such instructions since, if the analyser cannot know where control flows to it cannot follow all paths back from a register usage. There exist algorithms based on a technique called *program slicing* that can help narrow the set of candidate target locations, but even these do not help in the worst case.

An even more extreme case of non-statically visible coding is presented by self-modifying code, in which not just jump targets but entire blocks of instructions are calculated at run time. In this case the situation is a little easier in practice. The modified Harvard architecture of the ADSP-21xx means that only write accesses to program memory could possibly be candidates for self-modifying code, and in practice we find that these are extremely rare.

`asm21toc` warns when it finds non-statically visible constructs. It is possible in some cases to imagine that part of the processor architecture be *simulated* in the translated code, thus handling dynamically that which cannot be handled statically. Of course, at one extreme the entire architecture could be simulated as is done n the `sim21` simulator, but that would be uninteresting given that we want to provide an efficient runtime environment.

Counted loops are one area where the optimal simulation *vs* translation decision is unclear. By default, `asm21toc` simulates in detail the semantics of the four-deep counter stack, thus capturing the common programming idiom in which count values are passed a parameters to functions in the `count` register. The resulting code, however, is hardly transparent and so the `-D` command line option is available to allow a simpler (although not always correct) translation to be made based on local counter variables.

5.10.2 Aliasing of variables

If two or more `.var` declarations are made with the same absolute address then we have two identifiers for the same location. These constitute *aliases* in the same way that the `MR` register aliases with the `MR0`, `MR1` and `MR2` registers. The unaliasing mechanism in the present version of `asm21toc` can only handle the `MR` and `SR` registers. In the limit, aliasing arising from dynamic memory allocation is not statically visible anyway, although this is unlikely to be a practical problem in the kinds of applications deploying the ADSP-21xx processor.

5.10.3 Processing modes

The ADSP-21xx supports a series of processing modes which are enabled under programme control. These modes change the semantics of instructions: for instance *saturation* mode forces arithmetic results that have overflowed to yield a full scale deflection value rather than wrapping round as in conventional modulo arithmetic. It is possible to make the mode of the processor non-statically visible by making a mode change conditional on runtime data. This is unusual – some statistics on mode changes and their visibility are given in the papers reproduced in Appendix A. During parsing `asm21toc` lists all instructions that change the mode of the processor, and a manual check is recommended. Since many of the modes represent operations that are expensive to implement in a high level language (such as saturated arithmetic), `asm21toc` generates translations which use arithmetic modes that most closely match the semantics of ANSI-C.

5.10.4 Coding idioms

`asm21toc` works at the level of individual expressions, and presently has no capability to remap groups of statements into higher level constructs beyond the control flow reduction algorithm described above. Two areas of the processor's typical use would benefit from an ability to recognise coding *idioms*: division and generalised function manipulation.

Division on the ADSP-21xx is accomplished with multiple cycles of the division primitive instructions. Unfortunately for the reverse compiler writer, programmers sometimes use these primitives in devious ways to accomplish bit packing and other arcane operations, so the prescence of a division primitive does not of itself indicate that division is being attempted. Even when a division is being programmed, there are different ways of interleaving the division operations with other statements and this makes translation to the ANSI-C `/` operator challenging. `asm21toc` presently use macros to represent the division primitives. More research is required into typical usage of these primitives before a satisfactory general algorithm can be designed, although recognition of specific templates would be straightforward.

Generalised functions are implemented by some programmers to overcome limitations of the very shallow hardware program counter stack on the ADSP-21xx. This style of programming maintains a stack in data memory which works in conjunction with the hardware stack (and the call and return instructions) to support arbitrary nesting of functions. Finding function boundaries in such an environment is challenging, especially given the multitude of ways in which such a scheme may be implemented. `asm21toc` presently makes no attempt to locate functions beyond those defined by the hardware call instruction.

Bibliography

[JS97a] Adrian Johnstone and Elizabeth Scott. Designing and implementing language translators with `rdp` – a case study. Technical Report TR-97-27, Royal Holloway, University of London, Computer Science Department, December 1997.

[JS97b] Adrian Johnstone and Elizabeth Scott. `rdp` – a recursive descent compiler compiler. User manual for version 1.5. Technical Report TR-97-25, Royal Holloway, University of London, Computer Science Department, December 1997.

[JS97c] Adrian Johnstone and Elizabeth Scott. `rdp_supp` – support routines for the `rdp` compiler compiler version 1.5. Technical Report TR-97-26, Royal Holloway, University of London, Computer Science Department, December 1997.

[JS97d] Adrian Johnstone and Elizabeth Scott. A tutorial guide to `rdp` for new users. Technical Report TR-97-24, Royal Holloway, University of London, Computer Science Department, December 1997.

[JS98] Adrian Johnstone and Elizabeth Scott. `rdp` – an iterator based recursive descent parser generator with tree promotion operators. *SIGPLAN notices*, 33(9), September 1998.

[San95] Georg Sander. *VCG Visualisation of Compiler Graphs.* Universität des Saarlandes, 66041 Saarbrücken, Germany, February 1995.

Appendix A

Published papers concerning `asm21toc`

At the time of writing, three conference papers have been published concerning **`asm21toc`**, reprints of which are included here.

The first entitled *Reverse compilation of Digital Signal Processor assembler source to ANSI-C* was presented to the International Conference on Software Maintenance at Oxford in the Summer of 1999. Some aspects of the early version of the tool described there differ from the version documented in this manual.

The next (*Reverse compilation for Digital Signal Processors*) was delivered in January 2000 at the 33rd Hawaii International Conference on System Sciences where it received a best paper nomination. It presents some results from actual translations and highlights the cyclical nature of computer architecture design in which we are seeing a return to the visible timing constraints programming model common in the early 1950's.

Finally, *What assembler programmers get up to: control flow challenges in reverse compilation* (which was presented to the Conference on Software Maintenance and Renovation in Zurich during February 2000) focuses on the frequency of non-reducible control flow and multiple entry functions in real DSP assembly programs.