Bound-T time and stack analyzer

Application Note

ADSP-21020

TR-AN-21020-001
Issue 2
2013-11-28
Tidorum Ltd.

Tidorum Ltd
www.tidorum.fi
Tiirasaarentie 32
FI-00200 Helsinki
Finland

This document was written at Space Systems Finland Ltd by Niklas Holsti, Thomas Långbacka and Sami Saarinen within an ESTEC-supported project to develop a worst-case execution-time analyser for the ADSP-21020 processor architecture.

The document is currently maintained by Niklas Holsti at Tidorum Ltd.

Document reference:	TR-AN-21020-001
Document issue:	2
Document issue date:	2013-11-28
Bound-T/21020 version:	4b1
Last change included:	BT-CH-0258
Web location:	*http://www.bound-t.com/app_notes/an-21020.pdf*

Credits:
This document was created with the free OpenOffice.org software, *http://www.openoffice.org/*.

The information in this document is believed to be complete and accurate when the document is issued. However, Tidorum Ltd. reserves the right to make future changes in the technical specifications of the product Bound-T described here. For the most recent version of this document, please refer to the web address *http://www.bound-t.com/*.

If you have comments or questions on this document or the product, they are welcome via electronic mail to the address info@tidorum.fi, or via telephone, fax or ordinary mail to the address given below.

Please note that our office is located in the time-zone GMT + 2 hours (+3 hours in the summer) and office hours are 9:00 -16:00 local time.

Cordially,

Tidorum Ltd.

Telephone: +358 (0) 40 563 9186
Web: *http://www.tidorum.fi/*
http://www.bound-t.com/
Mail: *info@tidorum.fi*

Post: Tiirasaarentie 32
FI-00200 HELSINKI
Finland

Credits

The Bound-T tool was first developed by Space Systems Finland Ltd. (*http://www.ssf.fi/*) with support from the European Space Agency (ESA/ESTEC). Free software has played an important role; we are grateful to Ada Core Technology for the Gnat compiler, to William Pugh and his group at the University of Maryland for the *Omega* system, to Michel Berkelaar for the *lp-solve* program, to Mats Weber and EPFL-DI-LGL for Ada component libraries, and to Ted Dennison for the *OpenToken* package. Call-graphs and flow-graphs from Bound-T are displayed with the *dot* tool from AT&T Bell Laboratories. Some versions of Bound-T emit XML data with the *XML_EZ_Out* package written by Marc Criley at McKae Technologies.

To implement the ADSP-21020 version of Bound-T, we have used both the free technical information and the GCC-based compilers provided by Analog Devices Inc.

Contents

Index of tables

Document change log

Issue	Section	Changes
1	All	First issue, written at and issued by Space Systems Finland Ltd., using the FrameMaker tool.
2	All	Imported into OpenOffice (via plain text). Change log started. Change bars relative to issue 1 are not given. Page numbering is consecutive and starts from 1 for the front cover, for easier PDF handling.
	All	Chapters reordered to put the "user manual" chapters first and the "reference manual" chapters last.
	All	Systematically using the name ADSP-21020 for the target processor. Updated document layout to present Tidorum standard and content to version 4b1 of Bound-T for ADSP-21020.
	6	Added tables of warning and error messages related to COFF files and architecture files.

1 INTRODUCTION

1.1 Purpose and scope

Bound-T is a tool for computing bounds on the worst-case execution time and stack usage of real-time programs by means of a static analysis of the machine code of the program. There are different versions of Bound-T for different target processors. This Application Note supplements the general Bound-T manuals (references [1], [2], and [3]) by giving additional information and advice on using Bound-T for one particular target processor, the Analog Devices Digital Signal Processor architecture known as the ADSP-21020 [5]. This information includes

- the kinds of input files (executable programs) that Bound-T for ADSP-21020 can read,
- the ADSP-21020-specific command-line options for Bound-T,
- the ADSP-21020-specific details of the Bound-T assertion language, and
- the ADSP-21020-specific warning and error messages that Bound-T can emit.

Furthermore, the Application Note details how the analysis in Bound-T handles the features of the ADSP-21020 architecture, with emphasis on features for which the analysis is approximate or even absent.

Some information in this Application Note applies only when the target-program executable is generated with the Analog Devices software-development tools (the g21k C compiler, the assembler and linker [4]). This information could have been the subject of an independent Application Note but was included here because these are the tools usually employed by ADSP-21020 developers.

This Application Note is also applicable to other implementations of the same architecture, such as the radiation-resistant TSC-21020 processor produced by ATMEL. All these processors will be referred to collectively as "the ADSP-21020".

Note that Bound-T does not support the later ADI processors such as the 21060 and other SHARC models.

There may be other Bound-T Application Notes on issues that are not limited to the ADSP-21020, but nevertheless can be relevant when using Bound-T on ADSP-21020 programs. For example, there may be Application Notes dealing with the target-independent properties of certain cross-compilers, or the target-independent aspects of how Bound-T reads and interprets certain executable-program formats. Check the Bound-T web-site *http://www.bound-t.com/* for such information.

1.2 Overview

The reader is assumed to be familiar with the general principles and usage of Bound-T, as described in the Bound-T Reference Manual [1] and the Bound-T User Guide [2]. The User Guide contains a glossary of terms, many of which will be used in this Application Note.

In a nutshell, here is how Bound-T bounds the worst-case execution time (WCET) of a subprogram: Starting from the executable, binary form of the program, Bound-T decodes the machine instructions, constructs the control-flow graph, identifies loops, and (partially) interprets the arithmetic operations to find the "loop-counter" variables that control the loops, such as n in "for (n = 1; n < 20; n++) { ... }".

By comparing the initial value, step and limit value of the loop-counter variables, Bound-T computes an upper bound on the number of times each loop is repeated. Combining the loop-repetition bounds with the execution times of the subprogram's instructions gives an upper bound on the worst-case execution time of the whole subprogram. If the subprogram calls other subprograms, Bound-T constructs the call-graph and bounds the worst-case execution time of the called subprograms in the same way.

When the program under analysis contains complex loops that Bound-T cannot analyse automatically the user must set the repetition bounds for these loops. This is done by writing *assertions* in the Bound-T assertion language [3]. Assertions can also guide and help the analysis in other ways.

This Application Note explains how Bound-T has been adapted to the architecture of the ADSP-21020 processor and how to use Bound-T to analyse programs for this processor. To make full use of this information, the reader should be familiar with the register set and instruction set of this processor, as presented in references [5] and [6].

The remainder of this Application Note is structured as follows:

The remainder of this Application Note is divided into a *user guide* part and *reference* part. The user guide part consists of chapters 2 through 3 and is structured as follows:

- Chapter 2 explains those Bound-T command arguments and options that are wholly specific to the ADSP-21020 or that have a specific interpretation for this processor.
- Chapter 3 addresses the user-defined assertions on target program behaviour and explains the possibilities and limitations in the context of the ADSP-ADSP-21020 and the Analog Devices development tools.

The remainder of the Application Note forms the reference part as follows:

- Chapter 4 describes the main features of the ADSP-21020 architecture and how they relate to the functions of Bound-T.
- Chapter 5 defines in detail the set of ADSP-21020 instructions and registers that is supported by Bound-T.
- Chapter 6 lists and explains all warning and error messages that this version of Bound-T can emit, in addition to the generic messages that apply to any version of Bound-T. The generic messages are described in the Bound-T Reference Manual [1].

1.3 References

[1] Bound-T Reference Manual.
Tidorum Ltd. Doc.ref. TR-RM-001.
http://www.bound-t.com/manuals/ref-manual.pdf

[2] Bound-T User Guide.
Tidorum Ltd., Doc.ref. TR-UG-001.
http://www.bound-t.com/manuals/user-guide.pdf

[3] Bound-T Assertion Language.
Tidorum Ltd. Doc.ref. TR-UM-003.
http://www.bound-t.com/manuals/assertion-lang.pdf

[4] Using Bound-T in HRT Mode.
Tidorum Ltd. Doc.ref. TR-UM-002.
http://www.bound-t.com/manuals/hrt-manual.pdf

[5] ADSP-21020 User's Manual.
Analog Devices Inc. Second Edition, 1995.

[6] ADSP-21000 Family, C Tools Manual.
Analog Devices Inc. Third Edition, 1995.

[7] Bound-T Application Note: Virtuoso.
Space Systems Finland Ltd., Doc.ref. DET-SSF-MA-004.

1.4 Abbreviations and acronyms

See also reference [2] for terms specific to Bound-T and reference [5] for the mnemonic operation codes and register names of the ADSP-21020.

ADI	Analog Devices Inc.
ALU	Arithmetic and Logic Unit
CCP	C Calling Protocol
COFF	Common Object File Format
DAG	Data Address Generator
DAG1	Data Address Generator 1
DAG2	Data Address Generator 2
DM	Data Memory
DSP	Digital Signal Processor
HRT	Hard Real Time [4]
PCSP	PC Stack Protocol
PM	Program Memory
TBA	To Be Added
TBC	To Be Confirmed
TBD	To Be Determined
WCET	Worst-Case Execution Time

1.5 Typographic conventions

We use the following fonts and styles to show the role of pieces of the text:

register	The name of an ADSP-21020 register embedded in prose.
instruction	An ADSP-21020 instruction.
-option	A command-line option for Bound-T or other tools.
symbol	A mathematical symbol or variable.
text	Text quoted from a text / source file or command.
identifier	An identifier from a program.

2 USING BOUND-T FOR ADSP-21020

2.1 Input formats

Executable target-program files

The target program executable file must be supplied in COFF format. Two variants are supported: little-endian with 18-byte symbol records, as produced by the ADI tools on Microsoft Windows, and big-endian with 20-byte symbol records, as produced by the ADI tools on Sun Solaris systems. The COFF headers are not used for this distinction, since we have found them unreliable in this respect. Instead, the option *-coff_unix* must be given to select the big-endian form.

Patch files not supported

Bound-T provides the general option *-patch filename* that names a file that contains patches to be applied to the loaded target-program memory image before analysis starts. The format of the patch file is specific to the target processor. The ADSP-21020 version of Bound-T does not currently support patching and so no patch-file format is defined.

2.2 Command arguments and options

Generic options and arguments

The generic Bound-T command format, options and arguments are explained in the Bound-T Reference Manual [1] and apply without modification to the ADSP-21020 version of Bound-T. The command line usually has the form

boundt_sharc *options target-program-file root-subprogram-names*

(The name of the executable should perhaps have the form *boundt_21020* rather than *boundt_sharc;* the latter form was chosen because support for the later SHARC processors was planned.)

For example, to analyse the execution time of the main subprogram in the target program stored as the COFF file prog.coff under the option *-trace calls*, the command line is

```
boundt_sharc  -trace calls  prog.coff  main
```

Root subprograms can be named by the link identifier, if present in the program symbol-table, or by the entry address in hexadecimal form. Thus, if the entry address of the main subprogram is 12A0 (hex), the above command can also be given as

```
boundt_sharc  -trace calls  prog.coff  12A0
```

All the generic Bound-T options apply. The generic option *-help* provides information about the options Bound-T provides, either singly or in groups. For example, the option *-help sharc* lists and describes all the options specific to the ADSP-21020 version of Bound-T.

ADSP-21020-specific options

The additional ADSP-21020-specific options are explained in Table 1 below.

Table 1: Command Options for ADSP-21020

Option		*Meaning and default value*
-arch X	*Function*	*X* names an architecture file (.ach file) that Bound-T shall read to find the memory segments and the memory banks. See below for the way the architecture file is used.
	Default	A default architecture (defined within Bound-T, not in an .ach file) as explained below.
-coff_endian big *-coff_endian little*	*Function*	Specifies the endianness (octet significance order) to be assumed for the COFF file. The endianness may depend on the cross-compiler that generated the COFF file. See also the option *-coff_unix* .
	Default	Little-endian order: *-coff_endian little* .
-coff_sym_length L	*Function*	Specifies the length *L*, in octets, to be assumed for the symbol records in the COFF file. The length may depend on the cross-compiler that generated the COFF file. See also the option *-coff_unix* .
	Default	A length of 18 octets: *-coff_sym_length 18* .
-coff_trace	*Function*	See the *coff* item in the tracing options in Table 2.
	Default	No tracing.
-coff_unix	*Function*	Indicates that the COFF file comes from the ADI Unix tools and is big-endian with 20-byte symbol records. This is equivalent to the two options *-coff_endian=big -coff_sym_length=20*
	Default	The COFF file is assumed to come from the ADI MS-Windows tools and be little-endian with 18-byte symbol records. This is equivalent to the two options *-coff_endian=little -coff_sym_length=18*
-dag_trace	*Function*	See the *dag* item in the tracing options in Table 2.
	Default	No tracing.
-dm_read_ws X	*Function*	Sets the number *X* of memory wait states assumed for a data-memory read, if the memory location is not certain to be in the stack (see *-stack_read_ws*). This value can be overridden by user assertions.
	Default	Zero wait states: *-dm_read_ws 0*
-dm_write_ws X	*Function*	Sets the number *X* of memory wait states assumed for a data-memory write, if the memory location is not certain to be in the stack (see *-stack_write_ws*). This value can be overridden by user assertions.
	Default	Zero wait states: *-dm_write_ws 0*

Option		Meaning and default value
-fetch_ws X	*Function*	Sets the number *X* of memory wait states assumed for an instruction fetch from program memory. This value can be overridden by user assertions.
	Default	Zero wait states: *-fetch_ws 0*
-pm_read_ws X	*Function*	Sets the number *X* of memory wait states assumed for a read of data from the program memory. This value can be overridden by user assertions.
	Default	Zero wait states: *-pm_read_ws 0*
-pm_write_ws X	*Function*	Sets the number *X* of memory wait states assumed for a write of data to the program memory. This value can be overridden by user assertions.
	Default	Zero wait states: *-pm_write_ws 0*
-root P	*Function*	Specifies the calling protocol *P* to be assumed for all root subprograms. This setting can be overridden by user assertions. The possible values of *P* are *ccp* , *pcsp* , and *isr* , standing respectively for the C calling protocol [6], the native PC-stack protocol [5], and the ISR calling convention [5]. At present Bound-T considers the ISR protocol to be the same as the PCSP protocol.
	Default	Root subprogram are assumed to follow the C calling protocol: *-root ccp*
-stack_read_ws X	*Function*	Sets the number *X* of memory wait states assumed for a read from the stack, that is from a data-memory address which is index register **I6** or **I7** plus an offset. This value can be overridden by user assertions. For a safe WCET bound, the value of this option should not be greater than the value of *-dm_read_ws*. See section 5.11.
	Default	Zero wait states: *-stack_read_ws 0*
-stack_write_ws X	*Function*	Sets the number *X* of memory wait states assumed for a write to the stack, that is to a data-memory address which is index register **I6** or **I7** plus an offset. This value can be overridden by user assertions. For a safe WCET bound, the value of this option should not be greater than the value of *-dm_write_ws*. See section 5.11.
	Default	Zero wait states: *-stack_write_ws 0*

Default architecture file

If no *-arch* option is given, Bound-T uses an internal default architecture that corresponds to the following .ach file:

```
.system main;
.processor ADSP21020

.segment/pm/ram/begin=0x000000/ end=0x0000ff seg_rth;
```

```
.segment/pm/ram/begin=0x000100/ end=0x0003ff seg_init;
.segment/pm/ram/begin=0x000400/ end=0x003fff seg_pmco;
.segment/pm/ram/begin=0x004000/ end=0x007fff seg_pmda;
.segment/dm/ram/begin=0x00000000/end=0x00006fff seg_dmda;
.segment/dm/ram/begin=0x00007000/end=0x00007fff seg_stak;

.bank/pm0/wtstates=0/wtmode=internal/begin=0x000000;
.bank/pm1/wtstates=0/wtmode=internal/begin=0x008000;
.bank/dm0/wtstates=0/wtmode=neither/begin=0x00000000;
.bank/dm1/wtstates=0/wtmode=neither/begin=0x20000000;
.bank/dm2/wtstates=0/wtmode=neither/begin=0x40000000;
.bank/dm3/wtstates=0/wtmode=neither/begin=0x80000000;

.endsys
```

The architecture is currently used only to distinguish COFF sections that contain program instructions from those that contain data. In future versions of Bound-T, the architecture may be used to define the wait states, at least for PM and possibly for DM.

ADSP-21020-specific -trace options

Table 2 below describes the ADSP-21020-specific items for the generic option *-trace*, to ask for certain additional outputs from Bound-T.

Table 2: ADSP-21020-Specific *-trace* Options

***-trace* item**	**Traced information**
coff	COFF elements as they are read from the target program file. This may help to understand problems with the COFF file and its interpretation. If this option is selected together with the general *-dump* option, the COFF data are displayed twice: once while reading them, and once after the whole file has been read, as usual for *-dump*. The option *-coff_trace* is a deprecated form, equivalent to *-trace coff* .
dag	Extra NOP cycles assigned to a flow-graph edge for DAG load/use blocking. By default, only the total number of edges and extra cycles are displayed (as Notes).

ADSP-21020-specific -warn options

Table 3 below describes the ADSP-21020-specific items for the generic option *-warn*, to enable or disable specific warnings from Bound-T. By default all these warnings are disabled.

Table 3: ADSP-21020-Specific *-warn* Options

***-warn* item**	**Warning condition**
coff_sc	COFF symbols with strange "Storage Class" attributes, for which Bound-T cannot assign a known location in memory.
exit	Calls or jumps to the "_exit" subprogram, which terminates the program under analysis, and which are therefore modelled as returns from the subprogram in which they lie.

-warn item	Warning condition
short_loop	DO UNTIL loops that are very short and therefore have loop termination overhead (up to 2 cycles) when the actual number of iterations is small. Such loops may cause some over-estimation in the WCET bound, up to the duration of the loop termination overhead, per termination of the loop.

2.3 HRT analysis

The Hard Real Time (HRT) architecture pattern divides a real-time program in a specific way into concurrent tasks and protected objects. Bound-T has a special analysis mode for HRT programs. The general features and usage of this mode are described in reference [4]; there are no specific considerations for the ADSP-21020.

For HRT programs the ADSP-21020 is sometimes used with the Virtuoso kernel from Eonic Systems. Please refer to the separate Bound-T Application Note discussing Virtuoso [7].

2.4 Choice of calling protocol

The analysis of the computations in a subprogram depends on the calling protocol of the subprogram. The *C Calling Protocol* (CCP, defined in [6]) enforces a register discipline that considerably assists these analyses, as explained in section 5.5. The only alternative to the CCP is the *PC Stack Protocol* (PCSP), the native ADSP-21020 call/return protocol which uses the instructions **CALL** and **RTS**, and has no assumptions on register usage.

Bound-T assumes a protocol for a subprogram as follows:

- If there is a user assertion on the *ccp* property for this subprogram, this assertion defines the calling protocol, as described in section 3.8.
- Otherwise, if the subprogram is a "root" subprogram (that is, a subprogram named on the command line), the option *-root* defines the protocol; the default value of *-root* is CCP.
- Otherwise, the subprogram is analyzed by Bound-T only if it is called (directly or indirectly) from a root subprogram. If the call uses the CCP calling sequence, the callee is assumed to follow the CCP, otherwise the callee is assumed to follow the PCSP.

Therefore, in the absence of other specifications all root subprograms are assumed to use the CCP, and the protocol for any lower-level subprogram is defined by the way it is called from the higher-level subprograms. Bound-T emits a warning message if the same subprogram is called with both CCP and PCSP (in different calls).

With the assertion file a property can be asserted for a subprogram to tell that it does not follow the CCP register-usage rules internally, although it is called with the CCP sequence. See section 3.8.

2.5 Basic output format limitations

Most Bound-T outputs, including warning and error messages, follow a common, basic format that contains the source-file name and source-line number that are related to the message. However, the ADI C tools [6] do not maintain mappings between source-line numbers and

program memory addresses for an optimised program. The source-line number will then be missing from the message, or given in an inexact form, depending on the command-line option *-lines* (see [1]).

3 WRITING ASSERTIONS

3.1 Overview

This chapter explains any specific limitations and possibilities for user-specified assertions — written in the Bound-T assertion language [3] — when Bound-T is used with ADSP-21020 programs. Most of these issues are not caused by the ADSP-21020 as target processor, but by the Analog Devices development tools [6].

The issues concern the naming of subprograms, variables and source lines (via line numbers), in particular for optimised executables.

The special properties that can be asserted for ADSP-21020 programs are listed at the end of this chapter.

3.2 Naming scopes

The COFF file contains much symbolic information for debugging purposes. However, the COFF standard does not directly support a hierarchical (block-structured) name-space. Bound-T uses the source-file information to create scopes for symbols, as shown in Table 4 below. This gives unambiguous names to all subprograms and variables, even when they have the same source-code identifier.

According to this table. the full Bound-T symbol-name for a subprogram contains two levels: the first level is the name of the source file which contains the subprogram, and the second level is the name of the subprogram itself. For example, if the source file "subs.c" contains a function called "Foo", the Bound-T symbol for this subprogram is "subs.c|Foo", where the solidus | is used to separate the two levels.

For a local variable, the symbol has three levels: the name of the source file, the name of the subprogram which contains the local variable, and the name of the local variable itself. The most complex case is a parameter passed in a register, which is like a local variable except that the fixed string "Regparm" is inserted as level 3 and the parameter name is level 4.

Table 4: Naming Scopes

Type of symbol	Scope and name levels			
	Level 1	**Level 2**	**Level 3**	**Level 4**
Subprogram	Source file	Subprogram	N/A	N/A
Global variable	Source file	Variable	N/A	N/A
Parameter or local variable	Source file	Subprogram	Parameter Variable	N/A
Parameter passed in register	Source file	Subprogram	"Regparm"	Parameter
Statement label in C code	Source file	Subprogram	Label	N/A
Label in assembly-language code	Source file	Label	N/A	N/A

Note that the ADI C compiler [6] prefixes each C subprogram, variable name, or statement label with one underscore, "_". Moreover, for each parameter passed in a register, the C compiler usually also allocates a local-variable slot in the call frame on the stack, with the same name, which is why we add the "Regparm" scope to separate register parameters from local variables.

For some examples, consider the following source-code snippet, assumed to be located in the source file "jtables.c":

```
static int LastDCVal;    /* predictor for DC coding */
   ...
void
ScaleQTable (UINT8 QFactor)
{
   unsigned int Quality;
   ...
   abort_here:
   ...
}
```

The C-level identifiers in this code will be accessible as the following symbols, where the solidus '|' is the default scope delimiter:

- "jtables.c|_LastDCVal" (global variable)
- "jtables.c|_ScaleQTable" (subprogram)
- "jtables.c|_ScaleQTable|Regparm|_QFactor" (parameter in register)
- "jtables.c|_ScaleQTable|_QFactor" (parameter as local variable)
- "jtables.c|_ScaleQTable|_Quality" (local variable)
- jtables.c|_ScaleQTable|_abort_here" (statement label)

For assembly-language files, the ADI tools seems to put in the COFF symbol-table a source-file name that is not the real source-file name, but the name of some temporary file, which is often the same as the real source-file name but with the suffix changed to ".is".

3.3 Naming subprograms

The ADI C toolchain [6] generates the COFF Symbol Table information even for an optimised executable. The Symbol Table contains the names of all subprograms (and also the statement labels) and gives the corresponding Program Memory address for each of them. Thus, the naming of subprograms poses no problems whether the code is optimised or not.

Subprograms are named using their linkage names, which for C functions is the C name prefixed with an underscore. For example, "Foo" becomes "_Foo".

In addition to the linkage name there is one level of scope, which contains the source-file name.

3.4 Naming variables

Global C variables

The ADI C toolchain [6] generates COFF Symbol Table information listing the names and addresses of all global variables, even for an optimised executable. Thus, global variables can be named and tracked without problems, although the C compiler again prefixes all variable names with an underscore.

The "static" i.e. file-scope qualifier has no effect on naming. All global variables have one level of scope containing the source-file name, whether "static" or not.

Local C variables and C parameters

In a non-optimised compilation, symbolic information on local variables and parameters is available, with two levels of scope: the source-file name and the subprogram name. A third, synthetic "Regparm" level is added for parameters passed in registers.

Symbolic information on local variables is not provided in an optimised executable, and it seems likely that optimisation can have drastic effects on the set of local variables, such as deleting them in favour of using registers.

Assembler variables

Assembler variables are named as C variables, but the assembler respects the source-code name and does not add any underscores or mangle the name in other ways.

The source-file name is provided as one level of scope. However, it seems that this is usually not the real source-file name, such as "foo.asm", but the name of the pre-processed file where the suffix is ."is", becoming "foo.is" for example.

Since Bound-T cannot distinguish assembly-language PM symbols meant to represent subprograms from those that are meant to represent PM data, many such symbols will be defined both as subprograms and as data cells. This should do no harm, unless you try to analyse some data as it if were a subprogram.

3.5 Naming statement labels

Bound-T for the ADSP-21020 tries to make statement labels available as symbols, for use in assertions that identify loops based on the label of some statement within the loop. This applies both to statement labels in C code and in assembly-language code.

The ADI C compiler prefixes statement-label identifiers with an underscore, just as for subprogram identifiers and variable identifiers. Therefore, a C label like "start" appears to Bound-T as the symbol "_start".

Statement labels in assembly-language programs are entirely equivalent to subprogram names, and Bound-T identifies them as both subprogram symbols and label . Typically, the symbol scope for a statement label in assembly language consists only of the source-file name and excludes the name of the subprogram which contains the label (because assembly-language source-code seldom brackets subprograms with the "begin function" and "end function" symbols).

Note that for eternal C-code loops (of the form "loop: ... goto loop;") the ADI C compiler can place the label ("loop") on an instruction that is not within the loop body, in which case the label cannot be used to identify this loop in an assertion.

3.6 Naming statements by source-line numbers

Bound-T for the ADSP-21020 supports the use of source-code line-numbers for identifying statements in assertions. However, the ADI C compiler seems not to provide a mapping fmor line-numbers to object-addresses, when the code is optimized, which of course prevents the use of source line numbers in assertions.

Moreover, the ADI assembler seems never to provide such a mapping. Therefore, source-line numbers cannot be used to identify program locations in assembly-language modules.

3.7 Naming items by address

The registers are named in assertions with the **address** keyword, followed by a quoted string. The same keyword is used to name variables by their memory address. The value syntax for registers is a one-letter register-set identifier followed by a decimal register number within the valid range 0 - 15.

The valid register set identifiers are R for fixed point registers, I for index registers, M for modifier registers, B for base registers, and L for length registers. Lower-case variants (r, i, m, b, l) are also acceptable.

The memory addresses are given by a two-letter address-space identifier (DM for data memory or PM for program memory, with case-insensitive matching) followed by the hexadecimal address of the variable. The hexadecimal address is given by using decimal numbers 0 -9, and letters a, b, c, d, e and f (case-insensitive). Note that the address must not be preceded by "0x" nor surrounded by "16# .. #" nor followed by an "H" suffix.

Some examples of assertions naming registers or data addresses:

```
variable address "R3" 0 .. 100;   -- Register R3 bounded.
variable address "r3" 0 .. 100;   -- Same thing.

variable address "dm3fa7" = 20;
-- DM word at address hex 3FA7 (decimal 16295) holds
-- the value (decimal) 20.
```

3.8 Properties

The assertable properties for the ADSP-21020 are listed and explained in the following table.

Table 5: Assertable Properties for the ADSP-21020

Property name		Meaning, values and default value
ccp	*Function*	Changes the assumed property of a subprogram to follow or not to follow CCP register usage internally. Note that this property has a meaning only in the subprogram scope. Assertion scopes are explained in [3].

Property name		Meaning, values and default value
	Values	1 - The subprogram follows the CCP-protocol both as called and internally. The subprogram may use some other calling protocol in its own calls to lower-level subprograms. 0 - The subprogram does not follow the CCP protocol.
	Default	The default is determined for each subprogram separately as explained in section 2.4.
dm_read_ws	*Function*	Changes the number of data-memory read wait states in the current context.
	Values	Number of wait state cycles.
	Default	Zero wait states or the value given in a command-line option *-dm_read_ws*.
dm_write_ws	*Function*	Changes the number of data-memory write wait states in the current context.
	Values	Number of wait state cycles.
	Default	Zero wait states or the value given in a command-line option *-dm_write_ws*.
fetch_ws	*Function*	Changes the number of program-memory instruction fetch wait states in the current context.
	Values	Number of wait state cycles.
	Default	Zero wait states or the value given in a command-line option *-fetch_ws*.
pm_read_ws	*Function*	Changes the number of program-memory data read wait states in the current context.
	Values	Number of wait state cycles.
	Default	Zero wait states or the value given in a command-line option *-pm_read_ws*.
pm_write_ws	*Function*	Changes the number of program-memory data write wait states in the current context.
	Values	Number of wait state cycles.
	Default	Zero wait states or the value given in a command-line option *-pm_write_ws*.
stack_read_ws	*Function*	Changes the number of stack read wait states in the current context.
	Values	Number of wait state cycles.
	Default	Zero wait states or the value given in a command-line option *-stack_read_ws*.
stack_write_ws	*Function*	Changes the number of stack write wait states in the current context.
	Values	Number of wait state cycles.
	Default	Zero wait states or the value given in a command-line option *-stack_write_ws*.

4 THE ADSP-21020 AND TIMING ANALYSIS

4.1 The ADSP-21020 processor

The ADSP-21020 [5] is a 32-bit floating-point Digital Signal Processor (DSP). It has a Harvard architecture (separated program and data memories) and pipelined fetch, decode and execute cycles. Each instruction is 48 bits wide. Data can also be accessed in the program memory. A 2-way, set-associative instruction cache for 32 instructions reduces contention between fetches and data accesses on the program-memory bus.

Integer addition, subtraction and multiplication are supported in hardware but division is not. All floating point operations are supported in hardware. Special addressing hardware units support access to vectors, arrays and circular buffers with little overhead from index manipulations.

The ADSP-21020 supports zero-overhead loops, which are both nestable (six levels in hardware) and interruptable. Both delayed and non-delayed branches are supported.

An on-chip subroutine call stack handles up to 19 nested calls (less one for each active hardware loop). Programs written in C use a memory-resident stack and a specific calling sequence [6] which is also often used by assembly-language libraries, at least when designed to interface with C programs.

4.2 Static execution time analysis on the ADSP-21020

The ADSP-21020 architecture is very regular and quite fitting for static analysis by Bound-T. Instruction timing in no case depends on the data being processed, but only on the control flow.

The following architectural features can lead to approximate (over-estimated) execution times for the concerned instructions:

- Instruction cache effects.
- Short **DO UNTIL** loops with few iterations.
- Memory wait states that vary in number depending on the address.

See section 5.11 for more information about the approximations.

5 SUPPORTED ADSP-21020 FEATURES

5.1 Overview

This section specifies which ADSP-21020 instructions, registers and status flags are supported by Bound-T. We will first describe the extent of support in general terms, with exceptions listed later. Note that in addition to the specific limitations concerning the ADSP-21020, Bound-T also has generic limitations as described in the general manuals [1][2].

For reference, the generic limitations are briefly listed in section 5.2.

General support level

In general, when Bound-T is analysing a target program for the ADSP-21020, it can decode and correctly time all instructions, with minor approximations.

Bound-T can construct the control-flow graphs and call-graphs for all instructions, with a few exceptions. Bound-T supports both the processor's internal call/return protocol, using the **CALL**, **RTS**, and **RTI** instructions, and the C Calling Protocol ([6], section 3.2.7).

When analysing loops to find the loop-counter variables, Bound-T is able to track all the integer (fixed point) additions and subtractions. Bound-T correctly detects when this integer computation is overridden by other computations, such as multiplications or floating-point operations in the same registers. However, aliasing in the DM or PM may not be detected.

In summary, for a program written in a compiled language such as Ada or C, it is unlikely that the Bound-T user will meet with any constraints or limitations that are specific to the ADSP-21020 target system.

Before detailing the exceptions to the general support, some terminology needs to be defined concerning the levels of support.

Levels of support

Four levels of support can be distinguished, corresponding to the four levels of analysis used by Bound-T:

1. *Instruction decoding*: are all instructions correctly recognised and decoded? Is the execution time of each instruction correctly and exactly included in the WCET, or only approximately?
2. *Control-flow analysis*: are all jump, call and loop instructions correctly traced to their possible destinations? Are there other instructions that could affect control flow, and are they correctly decoded and entered in the control-flow graph?
3. *Definition analysis*: does the analysis correctly trace the effect of each instruction on the data flow, in terms of which "cells" (registers, memory locations) are defined (written, modified) by the instruction?
4. *Arithmetic analysis:* to what extent are the arithmetical operations of instructions mastered, so that the range of the results can be bounded?

These levels are hierarchical in the sense that a feature is considered to be supported at one level only if it is also supported at all the lower levels, with arithmetic analysis as the highest level.

Opaque values

When an operation is supported at the definition level, but not at the arithmetic level, then Bound-T's arithmetic analysis considers the operation's results to be "unknown" or *opaque*.

When an opaque value is stored in a register or memory location, the store is understood to destroy the earlier (possibly non-opaque) value and replace it with the opaque value. For arithmetic analysis, an opaque value represents an unconstrained value from the set of possible values of the storage cell (32 bits for a general register, 1 bit for a flag).

The difference between definition analysis and arithmetic analysis is crucial to Bound-T's ability to bound the worst-case times of loops. To illustrate this difference, Table 6 below lists some ADSP-21020 instructions in the second column and their definition-analysis and arithmetic analysis in the third and fourth columns. The instructions are assumed to be executed in sequence. The analysis contains just the aspects supported by Bound-T.

Table 6: Definition Analysis vs Arithmetic Analysis

No.	Instruction	Definition analysis	Arithmetic analysis
1	**R4 = 33**	**R4** gets a new value.	**R4** gets the value 33.
2	**R5 = R4 + 1**	**R5** gets a new value. **AZ**, **AN**, **AC** get new values.	**R5** gets the value **R4** + 1, which is 34. **AZ**, **AN**, **AC** all get the value 0.
3	**COMP (R4, R5)**	**AZ**, **AN**, **AC** get new values.	**AN** gets the value 1, since **R4** < **R5**. **AZ** and **AC** both get the value 0.
4	**R7 = R4 * R5**	**R7** gets a new value.	**R7** gets an opaque value, because Bound-T does not support multiplication in the general arithmetic analysis. However, in this case, where both operands have known values, the constant-propagation analysis deduces that **R7** gets the value 33 · 34 = 1122.
5	**MRF = R4 * R5**	No effect, because the analysis does not track the **MRF** register.	No effect, because the analysis does not track the **MRF** register.
6	**R5 = RND MRF**	**R5** gets a new value.	**R5** gets an opaque value, because any reading of an unsupported register such as **MRF** is opaque.
7	**R1 = R5 - R4**	**R1** gets a new value. **AZ**, **AN**, **AC** get new values.	**R1** gets the value **R5** - **R4**. **AZ** gets the value 1 if **R5** = **R4**, otherwise 0. **AN** gets the value 1 if **R5** < **R4**, otherwise 0. **AC** gets the value 1 if **R5** >= **R4** or if **R5** < 0, otherwise 0 (assuming **R4** =33)

Note that in the last row, arithmetic analysis tracks the fact that **R1** is now the difference between **R5** and **R4**, even though **R5** has an opaque value. Moreover, the analysis tracks the dependencies of the condition flags on the relative values of **R4** and **R4**. This tracking is important, for example when Bound-T examines the way a loop-body modifies a variable, to see if the variable is the loop-counter.

In fact, the same holds for all the table rows: arithmetic analysis tracks the formulae, not the values; the values (or value ranges) are then calculated from the formulae when needed.

Implications of limited support

Looking at the support levels from the Bound-T user's point of view, the following implications arise when the target program uses some ADSP-21020 feature which is not supported at some level.

- *Arithmetic analysis*: If a feature is supported at all levels except arithmetic analysis, then using this feature in any loop-counter computation will keep Bound-T from identifying the loop-counters (due to opaque values) so these loops cannot be bounded automatically. However, the other results from Bound-T stay valid.

For example, if the initial value of a loop-counter is read from the **MRF** register, as for **R5** in Table 6, then Bound-T cannot compute bounds for the initial value and thus cannot bound the loop.

- *Definition analysis*: If a feature is not supported in definition analysis, then in addition to the preceding impact, using this feature implies a risk of invalidating the arithmetic analysis, and thus a risk of incorrect results from Bound-T. Few ADSP-21020 features are at this level of non-support, and Bound-T will warn if they are used.

For example, if the instruction **R5 = RND MRF** in Table 6 were not supported in the definition analysis, it would not be seen to store a new value in **R5**, and the next instruction **R1 = R5 - R4** would seem to get **R5**'s value from the instruction **R5 = R4 + 1**, which would be quite wrong.

- *Control-flow analysis*: If a feature is not supported in control-flow analysis, then Bound-T can produce arbitrary (correct or incorrect) results when this feature is used in the target program, because the correct control-flow graphs cannot be determined. Again, Bound-T will warn of such usage.
- *Instruction decoding*: If a feature is not supported even for decoding, then it is useless to run Bound-T on a target program that uses this feature, since the only reliable result will be error messages.

5.2 Reminder of generic limitations

To help the reader understand which limitations are specific to the ADSP-21020 architecture, the following compact list of the generic limitations of Bound-T is presented.

Table 7: Generic Limitations of Bound-T

Generic limitation	Remarks for ADSP-21020 target
Understands only integer operations in loop-counter computations.	All results from floating-point operations are considered opaque.
Understands only addition, subtraction and multiplication by constants, in loop-counter computations.	The multiplier and shifter also often produce opaque values in the analysis model.
Assumes that loop-counter computations never suffer overflow.	Leads to non-support of the saturation-mode arithmetic (for counter computations), since it makes a difference only for overflows. Thus, the **ALUSAT** bit is ignored in condition codes.
Can bound only counter-based loops.	No implications specific to the ADSP-21020, although the ADSP-21020 instructions for zero-overhead loops may guide programmers to use counter-based loops more frequently.

Generic limitation	Remarks for ADSP-21020 target
May not resolve aliasing in dynamic memory addressing.	No implications specific to the ADSP-21020.

5.3 Support synopsis

The following table gives a synoptical view of the level of support for ADSP-21020 features. A plus "+" in a cell means that the feature corresponding to the table row is supported on the level corresponding to the table column. A shaded cell indicates lack of support.

The features are ordered from the fully supported at the top, to the unsupported at the bottom. More detail on the support level is given in the following sections.

Table 8: Synopsis of ADSP-21020 Support

ADSP-21020 registers, instructions, or other features	Decoding	Control flow	Definition	Arithmetic	Remarks
R0 .. R15: fixed point, addition, subtraction	+	+	+	+	
R0 .. R15: XOR when both operands are the same register	+	+	+	+	Equivalent to zero.
Index registers **I0 .. I15**, except bit-reverse	+	+	+	+	
M0 .. M15 **L0 .. L15** **B0 .. B15**	+	+	+	+	The C Calling Protocol requires some **M** and **L** registers to hold constants.
ASTAT flags **AZ**, **AN**, **AC**	+	+	+	+	
Condition codes: **EQ**, **LT**, **LE**, **AC**, **NE**, **GE**, **GT**, **NOT AC**	+	+	+	+	
NOP and **IDLE** instructions	+	+	+	+	The idling time is not considered.
R0 .. R15, fixed point: multiplication, shift, average **AND**, **OR**, **NOT**, **CLIP** **XOR** for two different registers	+	+	+		If the operation sets a flag to a constant (usually zero), then the flag is supported for arithmetic, too.
Index register **I0 .. I7** bit-reverse operations	+	+	+		
F0 .. F15, floating point, all operations	+	+	+		
Condition codes other than the above	+	+	+		
The use of circular buffers	+	+	+		
Reading (via Universal Register address): **PC**, **PCSTK**, **PCSTKP** **FADDR**, **DADDR**, **LADDR** **CURLCNTR**, **LCNTR**	+	+	+		
System registers read or written via Universal Register address, or modified by system-register bit-manipulation: **MODE1** (not alternate register bits), **MODE2** **ASTAT** (see above for flags) **STKY**, **IRPTL**, **IMASK**, **PMWAIT**, **DMWAIT**	+	+	+		Instructions that store values in **PMWAIT** and **DMWAIT** have no effect on the number of memory wait states assumed for instruction timing; that assumption is set by the command-line options *-xxx_ws* .
Alternate (secondary) registers as controlled by **MODE1** bits. Memory access with index register **I0** in bit-reversed mode as controlled by **MODE1** bit **BR0.**	+	+			An assignment to **MODE1** will generate a warning message.
Assignment (via Universal Register address) to: **PCSTK**, **PCSTKP**, **LADDR**, **CURLCNTR**, **LCNTR.** Push or pop of loop stack or status stack.	+				The possible effect on control-flow is not modelled. A warning message is generated.
Instructions for later SHARC processors					Only ADSP-21020 instructions are supported and decoded. Other instructions lead to error messages.

5.4 Data registers and memory accesses

The ADSP-21020 contains several sets of registers with different roles. This section explains how Bound-T supports these registers. The next section describes the additional support when the C Calling Protocol is in use.

Fixed-point register file R0 - R15

All ADSP-21020 register file locations (**R0 - R15**) are supported fully by Bound-T in fixed-point use. Each register is modelled as a separate data cell. However, there are general limitations on the modelling of overflow and signedness.

Floating-point register file F0 - F15

Floating-point operations (**F0 -F15**) are not supported in arithmetic analysis, but only as storing an opaque value in the underlying fixed-point register-file location (**Ri** corresponds to **Fi**). The registers **F0 - F15** are not even modelled as data cells, just as opaque views of **R0 - R15**.

Index and Modify Registers

The use of ADSP-21020 index registers (**I0 - I15**) is fully supported. Bit-reversed indexing is not supported in the arithmetic analysis.

Modify registers (**M0 - M15**) are fully supported.

Length and Base registers

The use of length registers (**L0 - L15**) and base registers (**B0 - B15**) is supported fully. However the use of circular data buffers with length and base registers is supported only on the definition level, and is not modelled in the arithmetic analysis.

Universal Register addressing

The ADSP-21020 has a mechanism of "universal register addressing" using an 8-bit address to define the source and destination for some data-moving instructions ([5], section A-5).

The addressable registers include the general register file **R0 - R15**, all the index, modifier, length and base registers, the Program Sequencing registers, the system registers including **ASTAT**, and several registers related to memory banks, busses and timers.

Fortunately, the universal register address is always statically known from the ADSP-21020 instruction (it is an immediate field). Thus, Bound-T supports universal register addressing of **R0 - R15**, **I0 - I15**, **M0 - M15**, **L0 - L15** and **B0 - B15** for arithmetic analysis.

The other universally-addressable registers are discussed later in this chapter. Most of them are supported only on the definition level.

5.5 Registers and the C Calling Protocol

The C Calling Protocol (CCP) ([6], section 4.2.1) is a set of rules on register usage that influences Bound-T's analysis. For each subprogram it analyses, Bound-T chooses whether or not to assume the CCP rules (as explained in section 2.4). The CCP rules are the following.

Firstly, CCP divides the ADSP-21020 registers into two subsets: *compiler* registers that are preserved across any call, and *scratch* registers that need not be preserved. Bound-T supports and uses this distinction in its arithmetic analysis.

Secondly, in CCP the index registers **I6** and **I7** are used as frame and stack pointers, respectively. Bound-T relies on **I6** to trace the use of subprogram parameters in arithmetic operations (when the subprogram is analysed separately for each call). **I6** should be modified only in the calling and returning sequences. Bound-T checks this and generate a warning if the rule is broken.

Thirdly, CCP specifies which registers are used for passing parameters and function values. The first three 32-bit parameters are passed in **R4**, **R8**, **R12**, and remaining parameters on the stack; a 32-bit return value is in **R0** and a 64-bit value in **R0** and **R1**. Bound-T uses these rules in arithmetic analysis to bring numerical values from the actual parameters at the call site, to a call-specific analysis of the callee.

Finally, CCP specifies that some **M** registers have fixed values: **M5** = **M13** = 0, **M6** = **M14** = 1, and **M7** = **M15** = -1. Also all **L** registers are specified to be zero. Bound-T uses these rules as background knowledge in the arithmetic analysis.

The C Calling Protocol also defines specific instruction sequences (code idioms) for calling a subprogram and for returning from one. Bound-T detects these sequences by look-ahead in the instruction decoder.

5.6 Modes, system registers, condition codes

The **ASTAT** status flags that are supported in arithmetic analysis are **AZ** (ALU result zero), **AN** (ALU result negative) and **AC** (ALU fixed-point carry).

For arithmetic analysis of condition codes, the **ASTAT** status flags are only used when they are defined by a fixed-point operation (**AF** = 0). In addition, since the saturation arithmetic is not supported, we can assume **ALUSAT** = 0 and thus the following simplified definitions of the supported ADSP-21020 condition codes can be used:

LT = **AN** *and* **-AZ**
LE = **AN** *or* **AZ**
GE = **-AN** *or* **AZ**
GT = **-AN** *and* **-AZ**

The condition code **TRUE** is of course supported fully. The **LCE** condition code is fully supported in **DO UNTIL LCE** loops. The remaining condition codes are considered opaque in ordinary conditional instructions, as is **LCE** in that context.

Direct assignment to the **ASTAT** register via universal register addressing is understood as storing opaque values in the status flags. However, if the assigned value is an immediate constant, the supported status flags are set to the correct immediate values. If the **ASTAT** register is updated with a system-register bit-manipulation instruction, the new values of the supported status flags are correctly modelled on all levels.

The use of system registers other than **ASTAT** is supported only on the definition level; all their values are considered opaque.

5.7 Computational operations

Whether or not a computational operation is supported on the arithmetic analysis level depends exclusively on the generic abilities of Bound-T; the only concern here is to map these abilities onto the ADSP-21020 instruction set.

Fixed-point operations

All fixed-point ALU operations are supported for definition analysis. The following addition, subtraction and comparison operations are supported for arithmetic analysis:

```
Rn = Rx + Ry
Rn = Rx - Ry
Rn = Rx + Ry + CI
Rn = Rx - Ry + CI - 1
COMP(Rx, Ry)
Rn = Rx + CI
Rn = Rx + CI - 1
Rn = Rx + 1
Rn = Rx - 1
Rn = -Rx
Rn = PASS Rx
Rn = ABS Rx
```

For these operations, the arithmetic effect is supported for the ALU status flags **AZ**, **AN** and **AC**.

When programming in assembly language, it is advisable to limit all loop-counter arithmetic to use only the above operations and move-operations (and other features supported on the arithmetic level). This will maximise Bound-T's automatic loop-bounding ability.

The fixed-point ALU operations that are not supported in arithmetic analysis are **AND**, **OR**, **XOR**, **NOT**, **CLIP** and **Average** (**Rn = [Rx + Ry] / 2**). In arithmetic analysis these operations are understood to store opaque values in the target register and the status flags.

As a special case, **XOR** is supported for arithmetic when its left and right operands are the same register, since the result is always zero. Moreover, if an operation yields a constant flag value (usually zero), then this flag is supported arithmetically for this operation.

Shifter and multiplier operations

Shift operations with literal (immediate) shift-counts could be supported on the arithmetic level, when they are equivalent to multiplication of a register by a constant, but the present version of Bound-T for the ADSP-21020 does not yet support this; the result of any shift operation is considered opaque.

Other shifter and multiplier operations are supported in definition analysis, but not for arithmetic analysis, where they are understood as storing an opaque value in the target register.

However, for the operations that do not modify a general register, such as multiplier operations that use the multiplier result register as target, or the **BTST** (bit test) shifter operation, the target-register value is left untouched and is not made opaque.

Floating-point operations

Floating-point ALU operations are supported on the definition level but not on the arithmetic level, where they are seen as storing opaque values in the target register and the ALU status flags (see section 5.6).

One floating-point ALU operation, **COMP**, does not redefine the target register value and for this operation the target-register value is left untouched and is not made opaque.

Multifunction operations

The definition-analysis and arithmetic analysis of multifunction operations is done in the same way as for the similar single operations.

The level of support is determined independently for each part of the multifunction operation.

5.8 Instructions

How an instruction is supported is determined mainly by the computational operations it contains. Below, the instruction groups are discussed in the same order as in appendix A of reference [5].

All the "compute" instructions are supported on the definition level, including the "move" and "register modify" or "immediate modify" sub-operations. Arithmetic support depends on the data type and operation as explained in section 5.7.

All the "immediate shift" instructions are supported on the definition level, and some are supported on the arithmetic level; see section 5.7.

Branch instructions

All jump and call instructions are supported on all levels. However, there are generic limitations on the control-flow analysis of indirect jumps and calls, where the target address is not static but is dynamically computed at run-time.

All return instructions are supported on all levels.

Loops

All **DO UNTIL** instructions are supported on all levels. Recall that there are generic limitations on the bounding of loops, depending on the complexity of the loop's termination conditions.

All **DO UNTIL LCE** loops are arithmetically analysed, and can be bounded if the initial value of **LCNTR** is arithmetically bounded.

Moves and miscellanea

All "move" instructions are supported on the arithmetic level when the source and target are fixed-point registers or fixed-point variables in static memory locations. When the source or target are floating-point registers or floating-point variables in static memory locations, support is reduced to the definition level. For universal register moves, see sections 5.4, 5.9 and 5.10.

System-register bit-manipulation is supported on the definition level except for changes to **MODE1** that could activate the bit-reversal mode of index register **I0**, or activate alternate (secondary) register sets.

Bit-reverse operations for index registers **I0** -**I7** are supported on the definition level but are not supported for arithmetic analysis. They will also hamper the identification of aliases in indirectly addressed memory locations and thus weaken the definition analysis.

Push and pop of the loop stack or status stack are not supported on the control-flow level, even, because they alter the program sequencing state in a complex way.

The **NOP** operation is supported on all levels, of course.

The **IDLE** instruction is supported on all levels, but Bound-T issues a message to warn that the idling time is not included in the WCET results.

5.9 Program Sequencer registers

The direct reading (via universal register addressing) of registers in the Program Sequencer is supported in Bound-T on the definition level (but all values read are considered opaque). These registers are the following, where an asterisk (*) indicates a read-only register:

PC*	program counter
PCSTK	top of PC stack
PCSTKP	PC stack pointer
FADDR*	fetch address
DADDR*	decode address
LADDR	loop termination address
CURLCNTR	current loop counter
LCNTR	loop counter

Writing values into these registers may alter the control flow in a way that Bound-T does not model, and so such writes are supported only on the instruction decoding level. If they occur in the target program, it is the user's responsibility to judge if Bound-T's results are still valid for WCET analysis.

5.10 Other registers

Any register not discussed above is supported at the definition level, but lumped together. Reading such a register yields an opaque value and writing into the register has no effect on control-flow or other modelled data values.

This includes registers such as **DMWAIT** and **PMWAIT** that define the wait-states for memory accesses. In other words, Bound-T does not track the values assigned to **DMWAIT** and **PMWAIT** to adjust the time it assigns to instructions. The number of memory wait states is set by command-line options (the *-xxx_ws* options).

5.11 Time accuracy and approximations

Bound-T reports WCET values that take into account most of the timing features of the ADSP-21020. This section explains these features, how Bound-T models them, and where Bound-T must make assumptions or approximations.

Writes to DAG registers or Memory Control Registers

According to section 7.2.1.5 of [5], the ADSP-21020 inserts an extra **NOP** cycle between two consecutively executed instructions if the first instruction "loads a DAG register" and the second instruction uses the same DAG "for data addressing". The explanation in [5] does not say exactly which instructions have these properties. Bound-T follows the behaviour of the ADI simulator for the ADSP-21020. This behaviour is shown in Table 9 in terms of the DAGs "loaded" and "used" by each type of instruction that accesses memory or a DAG. (DAG1 is for the Data Memory and DAG2 for the Program Memory.)

Table 9: DAGs Loaded and Used by an Instruction

Instruction type and page in [5]	Loads DAGs	Uses DAGs
compute / dreg ↔ DM / dreg ↔ PM A-12	none	1 and 2
compute / ureg ↔ DM|PM, register modify A-14	1 if target *ureg* is in DAG1 2 if target *ureg* is in DAG2 else none	1 if DM 2 if PM
compute / dreg ↔ DM|PM, immediate modify A-16	none	1 if DM 2 if PM
compute / ureg ↔ ureg A-18	1 if target *ureg* is in DAG1 2 if target *ureg* is in DAG2 else none	none (even if the source *ureg* is in a DAG)
immediate shift / dreg ↔ DM|PM A-20	none	1 if DM 2 if PM
compute / modify A-22	none	1 if DAG1 register is modified 2 if DAG2 register is modified
indirect jump|call / compute A-26	none	2 if indirect (via DAG2) else none (if **PC**-relative)
do until counter expired A-30	none	none (even if **LCNTR** is initialized from a DAG register)
ureg ↔ DM|PM (direct addressing) A-34	1 if target ureg is in DAG1 2 if target ureg is in DAG2 else none	none
ureg ↔ DM|PM (indirect addressing) A-35	1 if target ureg is in DAG1 2 if target ureg is in DAG2 else none	1 if DM 2 if PM
immediate data → DM|PM A-36	none	1 if DM 2 if PM
immediate data → ureg A-37	1 if target ureg is in DAG1 2 if target ureg is in DAG2 else none	none
I register modify / bit-reverse A-42	none	1 if DAG1 register is modified or bit-reversed 2 if DAG2 register is modified (bit-reverse is N/A for DAG2)

As an example, consider an instruction of type "A-14" that loads an index register in DAG1 from PM. An example of such an instruction is:

```
r4=r0+r2, i4=pm(i12,m13)
```

This instruction takes one cycle to execute, plus one cycle more since it accesses PM data (and if we assume that the next instruction is not in the cache). If the next instruction uses DAG1, one more extra NOP cycle is created, increasing the duration of the example instruction to 3 cycles (or more if there are PM wait states).

Whether an instruction "loads" or "uses" a DAG can also depend on the condition of the instruction; see below.

Call and return sequences

For any kind of subprogram call, whether it uses the C Calling Protocol (CCP) or the PC Stack Protocol (PCSP), Bound-T includes the call-sequence in the *caller's* execution time and the return-sequence in the *callee's* time. Thus, the WCET reported for a given subprogram corresponds to an execution from the first instruction at the subprogram's entry point, up to and including the last instruction of the subprogram.

DAG load/use blocking, as discussed above, may occur at a CCP return. The last instruction of the CCP return-sequence pops the old frame pointer (index register **I6**) from the DM stack and thus "loads" DAG1. If the instruction after the call "uses" DAG1, one **NOP** cycle results, which Bound-T includes in the *caller's* execution time. Note that the ADI simulator for the ADSP-21020 adds this **NOP** cycle to the *callee's* last instruction.

Timing of conditional instructions

Many ADSP-21020 instructions can be conditional in the sense that the instruction is executed only if a status flag is true (or false). The User's Manual [5] is not entirely clear on how the value of the conditition affects the execution time of a conditional instruction. Based on experiments with the ADI simulator for the ADSP-21020, Bound-T uses the following rules:

- A conditional instruction that accesses the DM incurs DM wait-state cycles only when the condition is true.
- For a conditional instruction that accesses the PM for data, an instruction-cache miss always causes one extra cycle, whether the condition is true or false, but the instruction incurs PM wait-state cycles for the data access only when the condition is true.
- A conditional instruction that "loads a DAG register" (as defined in Table 9) does so only when the condition is true. If the condition is false, the DAG register is not loaded (well, of course) and the next instruction can use this DAG without extra delay.
- An instruction that "uses a DAG register" (as defined in Table 9) does so always, whether its condition (if any) is true or false. Therefore, if the preceding instruction "loaded" a register in this DAG, one extra cycle is inserted even if the condition is false.

Memory wait-states

The ADSP-21020 User's Manual [5] states in section 7.2.1.6 that internally programmed memory wait states cause one cycle of delay for each wait state. The behaviour of the ADI simulator is a little more complex. The number of cycles taken by instructions of various types is shown in Table 10 as a function of the number d of DM wait states, the number f of PM wait states for instruction fetch, and the number p of PM wait states for data access. Bound-T follows this table.

Table 10: Effect of Memory Wait States on Execution Time

Instruction type in terms of the memory data accesses	Execution time in cycles for d, f and p as explained in the text
No DM or PM data access	$1 + f$
Access DM only	$1 + max\,(f, d)$
Access PM only, condition false	$2 + f$
Access PM only, condition true	$2 + f + p$

Instruction type in terms of the memory data accesses	Execution time in cycles for d, f and p as explained in the text
Access both DM and PM	$2 + f + max(p, d)$

Table 10 assumes that the instruction is not involved in a DAG load/use conflict (per Table 9). To handle a conflict, Bound-T adds one cycle to the execution time of the instruction pair (as computed from Table 10) whatever the number of wait states (since the delay is internal to the processor).

Table 10 can be understood as a consequence of the following rules:

- Firstly, each instruction takes one cycle to execute (in the processor). A new instruction is fetched concurrently (for one cycle), but the fetch wait-states consume an additional f cycles, giving $1 + f$ cycles in total.
- If the instruction accesses the DM, this is concurrent with the fetch, giving $max(f, d)$ wait cycles, for a total time of $1 + max(f, d)$ cycles.
- If the instruction accesses PM data, firstly there is always one extra cycle (at least if the cache misses), changing the constant term from 1 to 2. If the actual access is prevented by a false condition, the total time is thus $2 + f$ cycles. If the condition is true, PM data are accessed using p cycles, giving $f + p$ wait cycles and a total time of $2 + f + p$ cycles.
- If the instruction accesses both PM data and DM data, it seems that only the PM data access is concurrent with the DM access. The PM fetch access is done as a distinct sequential step. This leads to $f + max(p, d)$ wait cycles and a total time of $2 + f + max(p, d)$ cycles.

If all PM accesses were concurrent with the DM access, the last case would be expected to have $max(f + p, d)$ wait cycles.

Wait states for stack data

When a system has both fast and slow DM areas, it makes sense to place the CCP stack in fast memory. To support this, Bound-T lets you set the number of wait states to be assumed for data in the CCP stack, separately from the number of wait-states for other DM accesses, using respectively the command-line options *-stack_read/write_ws* and *-dm_read/write_ws* .

However, note that Bound-T cannot always find out if a given memory access lies in the stack, or in some other DM area. This happens for example if one subprogram passes a pointer to a stack-located variable as a parameter to another subprogram. In such cases, Bound-T considers that the access is not a stack access, which is safe (conservative) *only* if the stack memory is faster, or at least *not slower* than the other memory.

Therefore, you should ensure that the number of wait-states set for stack accesses (*-stack...ws*) is less or equal to the corresponding number of wait-states for general memory (*-dm...ws*). Otherwise, the WCET bounds may be underestimated.

Summary of approximations

The following table lists the cases where Bound-T uses an approximate model of the timing of ADSP-21020 instructions.

Table 11: Approximations for Instruction Times

Case	Description	Maximum error
Instruction cache effects	If an instruction accesses the program memory for operand data, a bus conflict on the program-memory bus causes an instruction-fetch delay, unless the instruction to be fetched is in cache ([5], section 3.2.2). Bound-T assumes conservatively that the cache always misses, thus the delay for the additional instruction fetch is always included for these instructions.	1 PM access (including wait states) per such instruction
Short **DO UNTIL** loops	Short **DO UNTIL** loops with few iterations may require some delay cycles to terminate and fetch the next instruction ([5], section 3.5.1.2). As Bound-T computes only an upper bound on the number of iterations, it cannot know if the delay occurs or not, and so it assumes conservatively that the delay always occurs.	2 cycles per loop termination
Writing Memory Control registers	An instruction that writes a memory control register (**DMWAIT**, **DMBANK1-3** or **DMADR** for DAG1, and **PMWAIT**, **PMBANK1** or **PMADR** for DAG2) suffers an extra **NOP** cycle if the following instruction uses the corresponding DAG ([5], section 7.2.1.5). Bound-T assumes that this extra **NOP** cycle always occurs, without inspecting the following instruction. (For the other case discussed in this section of [5], where the first instruction loads a DAG index, modifier, base or length register, Bound-T adds the extra cycle only if required by the following instruction; see Table 9.)	1 cycle per such write instruction
DAG load/use blocking on return	If a callee subprogram has several return points, some of which load a DAG register (per Table 9) in the last instruction, Bound-T assumes that any execution of the subprogram may end with loading any of these DAGs, and can thus cause an extra cycle if the instruction after the call uses any of these DAGs, even if some of the possible return points do not load any DAG.	1 cycle per call
Memory wait states that vary between memory banks	In the ADSP-21020, different memory banks can be configured to have different numbers of wait states. By default, Bound-T assumes the same number of wait states for any memory access (of a particular kind: fetch, stack data, other DM data, PM data). This number is set by the user, so a safe choice would be the largest number of wait states in any memory bank. If a constant number of wait-states is not satisfactory, the number can be specified for individual loops or subprograms by means of property assertions as explained in section 3.8. Bound-T reads the target-program's architecture file (.ach file) and could therefore use the memory specifications it contains. This is a possibly useful extension, not yet implemented.	Depends on user-given values. See Table 10.
Additional memory wait states at page boundaries	The ADSP-21020 can be configured to insert additional wait states when the addressed memory page changes. Bound-T however assumes the same number of wait states for any memory access without considering access sequences or page boundaries.	Depends on user-given values. See Table 10.
Bit-reversed addressing mode	Addresses output in bit-reverse mode always activate the lowest bank of data memory space, including the number of wait states associated with it ([5], section 7.2.9). The number of wait states assumed by Bound-T does not depend on whether the addressing mode is bit-reversed.	Depends on user-given values. See Table 10.

6 WARNINGS AND ERRORS FOR THE ADSP-21020

6.1 Warning messages

The following lists the Bound-T warning messages that are specific to the ADSP-21020 or that have a specific interpretation for this processor. The messages are listed in alphabetical order. The Bound-T Reference Manual [1] explains the generic warning messages, all of which may appear also when the ADSP-21020 is the target.

As Bound-T evolves, the set and form of these messages may change, so this list may be out of date to some extent. However, we have tried to make the messages clear enough to be understood even without explanation. Feel free to ask us for an explanation of any Bound-T output that seems obscure.

There are also some possible warnings arising from problems in the COFF input file. These are listed and described in Table 12.

Table 12: Warning Messages Specific to the ADSP-21020 Target

Warning message		Meaning and remedy
Alternate registers not supported	*Reasons*	A system-register bit-manipulation instruction sets or toggles bit 1 .. 7 or 10 of the **MODE1** register, which may enable the use of alternate processor registers. The analysis in Bound-T does not support switching between alternate registers.
	Action	If the code under analysis changes the primary/alternate register-set selection, note that the analysis results may be wrong.
Arithmetic effect of Push/Pop is not modelled	*Reasons*	The instruction manipulates loop counter or status stacks and its effect is not modelled.
	Action	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion.
Assertion on property "CCP" overrides default	*Reasons*	The calling protocol of the present subprogram is defined by an assertion on the "CCP" property (see secion 3.8), rather than by the default or automatic choice in Bound-T.
	Action	None needed, assuming that this overriding is intentional.
Calling protocol unknown; entry bounds unknown	*Reasons*	The calling protocol of the present subprogram is not known to Bound-T. Therefore, Bound-T cannot deduce bounds on the registers on entry to the subprogram, because the bounds depend on the protocol.
	Action	Consider asserting the "CCP" property for the subprogram, which will tell Bound-T what calling protocol to assume.
Call to _exit is converted to return	*Reasons*	The instruction calls subprogram *_exit* and is therefore treated as a return instruction.

Warning message		Meaning and remedy
	Action	Note that the subprogram *_exi*t is not analyzed by Bound-T, and its execution time is not included in the reported WCET results. Note also that if this call is in a subprogram which is not the root subprogram, treating the call as a return means that the analysis assumes that execution does not stop at this call, but returns to the caller of this subprogram and continues there. This can lead to surprising (unrealistic) analysis results.
CCP call to non-CCP subprogram: *S*	*Reasons*	The present subprogram contains a call to the subprogram *S,* which is known (or believed) not to follow the C Calling Protocol, but the call is in the CCP form.
	Action	If this is an intentional mix of calling protocols, check that the analysis of parameter passing and stack management is correct. If the mixing of protocols is not intentional, correct the program to use protocols consistently.
CCP return within DO UNTIL loop at *A*	*Reasons*	The instruction at address *A* starts a CCP return sequence, but there are some live **DO UNTIL** loops in the stack. This is abnormal, since the CCP protocol assumes that the loop stack is invariant over a call, and here the call would push something on the loop stack.
	Action	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion.
Condition NOT LCE not in loop, considered opaque	*Reasons*	The instruction uses the **LCE** (loop counter not expired) condition with no containing loop.
	Action	Note that Bound-T is unable to analyse the condition on this instruction. If it forms part of a loop-counting mechanism, the loop cannot be bounded automatically.
Delayed jump to _exit is converted to return	*Reasons*	The instruction jumps (with delay) to subprogram *_exit* and is therefore treated as a return instruction.
	Action	Note that the subprogram *_exi*t is not analyzed by Bound-T, and its execution time is not included in the reported WCET results. Note also that if this jump is in a subprogram which is not the root subprogram, treating the jump as a return means that the analysis assumes that execution does not stop at this jump, but returns to the caller of this subprogram and continues there. This can lead to surprising (unrealistic) analysis results.
DO UNTIL LCE without counter init ends at *E*, considered opaque	*Reasons*	This **DO UNTIL** loop, which ends at address *E*, uses the loop counter as the end condition (**LCE**) without initialising the counter.
	Action	Note that Bound-T is unable to analyse the condition on this instruction. This **DO UNTIL** loop cannot be automatically bounded.
DO UNTIL loop is short	Reasons	A short **DO UNTIL** loop was identified. Loops that are shorter than the processor pipeline terminate in special ways [5].

Warning message		Meaning and remedy
	Action	Note that the estimated WCET for this loop may be too large by 2 cycles, if the number of loop iterations is small.
DO UNTIL within loop-end delay at *A*	*Reasons*	There is a **DO UNTIL** instruction at address *A*, too close to the end of another **DO UNTIL** loop. The program is illegal [5].
	Action	Correct the program.
I0 bit-reverse mode not supported	*Reasons*	A system-register bit-manipulation instruction sets or toggles bit 1 of the **MODE1** register, which may enable bit-reverse mode for index register **I0**. Bound-T does not support bit-reversed indexing in its analysis.
	Action	If bit-reverse mode is enabled, note that the analysis results may be wrong.
Idling time not included in the WCET	*Reasons*	The subprogram contains an **IDLE** instruction, for which the idling time cannot be known.
	Action	Note that the computed WCET for this subprogram contains no contribution from the idling time.
Illegal instruction taken as NOP	*Reasons*	The program contains an illegal instruction (for which an error message was already emitted). Bound-T attempts to continue the analysis by assuming that this instruction does nothing.
	Action	Note that if this assumption is wrong, the analysis results may be wrong. Remove the illegal instruction from the program.
Immediate modifier *M* too large for Program Memory	*Reasons*	The modifier value *M* that is used with a Program Memory index register is too large. Only the 24 least significant bits of the value are used.
	Action	The user is responsible for evaluating the impact.
Improper change to stack pointer	*Reasons*	The instruction has modified the stack pointer in a way that cannot be analysed. Bound-T loses its knowledge of the stack-pointer value at this point in the control flow.
	Action	Note that the use of stack is unanalysable after this instruction.
Jump to _exit is converted to return	*Reasons*	The instruction jumps (without delay) to subprogram *_exit* and is treated as a return instruction.
	Action	Note that the subprogram *_exit* is not analysed by Bound-T, and its execution time is not included in the reported WCET. Note also that if this jump is in a subprogram which is not the root subprogram, treating the jump as a return means that the analysis assumes that execution does not stop at this jump, but returns to the caller of this subprogram and continues there. This can lead to surprising (unrealistic) analysis results.
Jump to _exit with non-empty loop stack	*Reasons*	The instruction jumps to subprogram *_exit*, but still has live **DO UNTIL** loops in the stack.
	Action	The user is responsible for evaluating the impact on the results.

Warning message		Meaning and remedy
Large literal *L* = hex *H,* used as signed = *V*	*Reasons*	A large value (*L* as unsigned decimal, *H* as unsigned hexadecimal) has been interpreted as a signed value (*V*, decimal) by Bound-T, but it might be wrong. For example, the 32-bit datum that in its unsigned form has the value 4294967294 can also be interpreted as -2. On the other hand, it might be intended as a bit mask.
	Action	The user is responsible for evaluating the impact.
Large literal *L* = hex *H,* used as unsigned	*Reasons*	A large value (*L* as unsigned decimal, *H* as unsigned hexadecimal) has been interpreted by Bound-T as unsigned, but it might be wrong.
	Action	The user is responsible for evaluating the impact.
MODE1 system register modified	*Reasons*	The instruction changes the value of the MODE1 register, which may enable alternate register sets and/or bit-reverse mode for index register I0. Bound-T does not support alternate registers or bit-reversed indexing.
	Action	If either alternate register sets of bit-reversal are activated, note that the analysis results may be wrong.
Non-CCP callee assigns CCP cell: *C*	*Reasons*	In this call, the caller is assumed to follow the C Calling Protocol (CCP) but not the callee. However, the callee seems to change the value of some register or other storage cell which should be invariant over the call, if the caller follows the CCP.
	Action	Check that the assumptions (or assertions on the ccp property) are correct for this program.
Non-CCP call to CCP subprogram: *S*	*Reasons*	The present subprogram contains a call to the subprogram *S,* which is known (or believed) to follow the C Calling Protocol, but the call is not in the CCP form.
	Action	If this is an intentional mix of calling protocols, check that the analysis of parameter passing and stack management is correct. If the mixing of protocols is not intentional, correct the program to use protocols consistently.
Parameter *P* mapped beyond caller's stack frame	*Reasons*	The present subprogram accesses a parameter *P* from the stack, but this parameter seems not to be in the stack frame of the immediate caller, but in some even earlier stack frame (along the present call path). This is unusual, because a C subprogram usually cannot see such local variables in higher-level subprograms and so should not be able to access them (except through a pointer, which is not the case here).
	Action	Note that the analysis of parameters passed to this subprogram may not be correct. Inform Tidorum of the problem, if you can give your program to Tidorum for deeper analysis.
Program Memory cell address is too large.	*Reasons*	A memory access to Program Memory has an address that is out of bounds of the memory space.
	Action	The user is responsible for evaluating the impact.

Warning message		Meaning and remedy
Program Sequencer register modified: *R*	*Reasons*	The instruction modifies the Program Sequencer register *R* (via Universal Register addressing). Changing the values of Program Sequencer registers can change the control-flow in ways that Bound-T does not master.
	Action	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion, because even the control-flow analysis may be invalidated.
Return within DO UNTIL loop at *A*	*Reasons*	The instruction at address *A* is a PCSP return (**RTS**) with some live **DO UNTIL** loops in the stack.
	Action	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion.

Table 13 below describes the warnings that may be issued for problems in the COFF input file.

Table 13: Warnings for COFF Problems

Warning message		Meaning and remedy
Assumed to be DM section: *S*	*Reasons*	According to the COFF symbol-table, an external or static symbol which is not a function symbol lies in section *S*, but the architecture file does not say if section *S* is Data Memory or Program Memory. Bound-T assumes that this symbol is a variable in Data Memory.
	Action	Note that this assumption may be wrong. If necessary, add the information to the architecture file.
COFF Aux_BF out of context	*Reasons*	The COFF file contains an auxiliary entry of type "Begin Function", but not at the start of the symbols for a subprogram. Bound-T ignores this auxiliary entry.
	Action	Probably no action is needed.
COFF Aux_BF within block	*Reasons*	The COFF file contains an auxiliary entry of type "Begin Function" nested within a block scope. This is unexpected. Bound-T ignores this auxiliary entry.
	Action	Probably no action is needed.
COFF Aux_EF out of context	*Reasons*	The COFF file contains an auxiliary entry of type "End Function", but not at the end of the symbols for a subprogram. Bound-T ignores this auxiliary entry.
	Action	Probably no action is needed.
COFF Aux_EF within block	*Reasons*	The COFF file contains an auxiliary entry of type "End Function" nested within a block scope. This is unexpected. Bound-T ignores this auxiliary entry.
	Action	Probably no action is needed.
Format of COFF line number table is incorrect	*Reasons*	The table mapping source-code line-number to machine code addresses, which is part of the COFF debugging data, seems to be internally inconsistent. (An entry in the table has a line number of zero, which means that the entry is a reference to a subprogram, but the referenced symbol-table entry seems not to be a subprogram entry.)

Warning message		Meaning and remedy
	Action	Note that the mapping between source-code line-numbers and machine-code address may be wrong or incomplete, as Bound-T uses it.
Invalid register for COFF symbol: *S*	*Reasons*	The COFF symbol table says that symbol *S* lies in a register, but gives a register number that is too large. Bound-T skips this symbol-table entry.
	Action	Note that this symbol will not be available for use in assertions.
Memory space unknown for segment *S*	*Reasons*	The memory address space (PM or DM) is unknown for this segment, probably because it is not specified in the architecture file.
	Action	Add the information to the architecture file.
Section *S* has relocation entries, perhaps file is not linked	*Reasons*	The section called *S* in the binary has relocation entries and might not be linked.
	Action	Ensure that the COFF file is fully and statically linked, with no remaining relocations needed.
Segment *S* assumed to be *N**8 bits *Space Kind*	*Reasons*	Segment *S* is not listed in the architecture file. Bound-T assumes that this segment has *N*-octet words (addressing units), is in the indicated memory *Space* (PM or DM), and is of the indicated *Kind* (RAM, ROM, or PORT).
	Action	If the assumption is wrong, add this segment to the architecture file with the right properties.
Skipped misplaced COFF auxiliary symbol, *kind*	*Reasons*	The COFF file contains an "auxiliary" symbol of the given *kind* but it is out of place in the symbol stream. Bound-T therefore ignores and skips this symbol entry.
	Action	Probably no action is needed.
Skipping COFF symbol, kind = *K*, Nature = *N*	*Reasons*	The COFF file contains a symbol entry of the given kind and nature, but it seems to be out of place in the symbol stream and is therefore ignored.
	Action	Probably no action is needed. Possibly some symbols will not be available to Bound-T for use in assertions or other inputs.
Unexpected length *L* of optional file header (skipped)	*Reasons*	The COFF file has an "optional header" with a non-zero length *L* octets. Bound-T cannot use such headers and therefore skips them.
	Action	Note that the information in the optional header is not available to Bound-T, whatever it is.
Unsure about COFF symbol with Storage Class *C* : *symbol*	*Reasons*	The COFF "storage class" attribute of this *symbol* has the strange value *C*. Bound-T cannot classify the symbol.
	Action	Note that the symbol may not be usable to identify subprograms or variables, and you may have to use machine addresses instead.
Width and space unknown for segment *S*	*Reasons*	The memory address space (PM or DM) and word width are unknown for this segment *S*, probably because they are not specified in the architecture file.
	Action	Add the information to the architecture file.

Table 14 below describes warnings that may be issued for problems in the architecture file.

Table 14: Warnings for Architecture File Problems

Warning message		Meaning and remedy
Bank *B* not specified; using default attributes	*Reasons*	The properties of the memory bank *B* are not specified in the architecture file, so Bound-T will use default properties, which are wait states = 0, wait-mode = "neither", and page size = 0.
	Action	If the defaults are not correct, add the correct information to the architecture file.
Width and space unknown for segment *S*	*Reasons*	The architecture file does not specify the word-width nor the address space (PM or DM) for the segment named *S*. If this segment is used, Bound-T will have to guess some default values (which are reported as other warnings).
	Action	Add this information to the architecture file.

6.2 Error messages

This section lists and describes the Bound-T error messages that are specific to the ADSP-21020 or that have a specific interpretation for this processor. The messages are divided into three tables as follows:

- Table 15 shows the error messages related to the analysis in general, including errors related to command-line options specific to the ADSP-21020.
- Table 16 shows the error messages related to the COFF file.
- Table 17 describes the error messages related to the architecture file.

In each table the messages are listed in alphabetical order. The Reference Manual [1] explains the generic error messages, all of which may appear also when the ADSP-21020 is the target.

As Bound-T evolves, the set and form of these messages may change, so this list may be out of date to some extent. However, we have tried to make the messages clear enough to be understood even without explanation. Feel free to ask us for an explanation of any Bound-T output that seems obscure.

Table 15: Error Messages Specific to the ADSP-21020 Target

Error Message		Meaning and Remedy
Architecture file could not be opened	*Problem*	The architecture file (named with the *-arch* option) seems to exist, but for some reason could not be opened for reading. The name of the architecture file is shown in field 3 of this error message.
	Reasons	Perhaps the user does not have access rights to read the file.
	Solution	Ensure that the user has access rights to read the architecture file.
Architecture file was not found	*Problem*	The architecture file (named with the *-arch* option) seems not to exist. The name of the architecture file is shown in field 3 of this error message.

Error Message		Meaning and Remedy
	Reasons	Error in the command-line option
	Solution	Correct the file-name in the *-arch* option.
Branch without loop abort within loop delay at *A*	*Problem*	There is a branch instruction, without Loop Abort, at address *A* too near to the loop's end. The program is illegal.
	Reasons	A mistake in the program.
	Solution	Correct the program.
Call too near end of DO UNTIL loop	*Problem*	The program contains a **DO UNTIL** loop which contains a **CALL** instruction, but the program is illegal because this **CALL** is to close to the end of the loop.
	Reasons	The program is illegal.
	Solution	Correct the program.
Constant cell modified in a CCP subprogram at *A*	*Problem*	A subprogram that is assumed to follow the CCP protocol contains an instruction, at address *A*, which violates the protocol by changing the value of a register that should have a constant value.
	Reasons	Programming error, or there is a CCP-call to a subprogram that intentionally does not follow the CCP -protocol internally. The call makes Bound-T assume that the subprogram follows the CCP fully, both in its caller/callee interactions and in its internal logic.
	Solution	Add a "CCP" property for this subprogram into the assertion file.
Data Memory address *A* is in no DM section	*Problem*	The program seems to access DM data at address *A*, but this address is not in any of the DM sections defined in the architecture file.
	Reasons	The program and the architecture file are not coherent, or the analysis is exploring an execution path that is infeasible in reality.
	Solution	Check that the given architecture file is valid for the given program.
DO UNTIL nested too deeply at *A*	*Problem*	**DO UNTIL** loop stack has overflowed at address *A*. The present subprogram has more than 6 levels of nested **DO UNTIL** loops. Note that Bound-T at present does not keep track of how the loop stack grows from subprogram to subprogram in the call chain. For example, if subprogram Foo has a 4-deep loop nest, where the innermost loop calls subprogram Bar which has a 3-deep loop nest, then the loop stack overflows (4 + 3 = 7 > 6), but Bound-T does not detect the overflow.
	Reasons	A mistake in the program.
	Solution	Obtain a correct COFF file.

Error Message		Meaning and Remedy
DO UNTIL with zero offset at *A*	*Problem*	**DO UNTIL** instruction at address *A* tries to create a zero length loop.
	Reasons	A mistake in the program.
	Solution	Obtain a correct COFF file.
Dynamic call is taken as return	*Problem*	The program contains a call, using the PCSP protocol, in which the address of the callee is computed dynamically and not statically known. Bound-T is unable to model the call correctly and instead models it as a return from the caller. The WCET of the caller may be underestimated.
	Reasons	The program is written in this way.
	Solution	Change the program to avoid dynamic calls.
Dynamic calls not supported	*Problem*	The program contains a call in which the address of the callee is computed dynamically and not statically known. Such calls are not supported in this version of Bound-T.
	Reasons	The program is written in this way. Perhaps, if this is a C program, it uses function pointers.
	Solution	Change the program to avoid dynamic calls, or ask Tidorum Ltd to implement support for such calls.
Illegal instruction at *A*	*Problem*	The instruction at address *A* is not a valid ADSP-21020 instruction.
	Reasons	A mistake in the program, or the program may be compiled for some later SHARC model with an extended instruction set.
	Solution	Obtain a correct COFF file for the ADSP-21020.
Instruction address *A* is in no PM section	*Problem*	The program seems to contain the instruction at address *A*, but this address is not in any of the PM sections defined in the architecture file.
	Reasons	The program and the architecture file are not coherent, or the analysis is exploring an execution path that is infeasible in reality.
	Solution	Check that the given architecture file is valid for the given program.
Invalid CU field in a Single operation	*Problem*	The instruction tries to use an illegal computation unit.
	Reasons	A mistake in the program.
	Solution	Obtain a correct COFF file.
Invalid Dual Add/Subtract instruction	*Problem*	The instruction has an illegal compute part.
	Reasons	A mistake in the program.
	Solution	Obtain a correct COFF file.
Invalid Parallel Multiplier & Dual Add/Subtract instruction	*Problem*	The instruction has an illegal compute part.
	Reasons	A mistake in the program.

Error Message		Meaning and Remedy
	Solution	Obtain a correct COFF file.
Jump too near end of DO UNTIL loop	*Problem*	The program contains a **DO UNTIL** loop which contains a **JUMP** instruction, but the program is illegal because this **JUMP** is to close to the end of the loop.
	Reasons	The program is illegal.
	Solution	Correct the program.
Overflow or underflow in 6-bit (or 24-bit) PC-relative address at *A*	*Problem*	A Program-Counter-relative branching instruction at address *A* causes the target address to overflow (address is above the maximum size of Program Memory) or to underflow (address is below zero).
	Reasons	A mistake in the program, or the analysis is exploring an infeasible execution path.
	Solution	Obtain a correct COFF file.
Patching is not implemented for SHARC	*Problem*	The command-line option *-patch* is used, but this version of Bound-T does not support patching.
	Reasons	The command-line is written that way.
	Solution	Manage without patching, or ask Tidorum to implement patching for this version of Bound-T.
Program Memory datum address *A* is in no PM section	*Problem*	The program seems to access PM data at address *A*, but this address is not in any of the PM sections defined in the architecture file.
	Reasons	The program and the architecture file are not coherent, or the analysis is exploring an execution path that is infeasible in reality.
	Solution	Check that the given architecture file is valid for the given program.
Property *P* has no valid upper bound. Using zero.	*Problem*	The user's assertion file asserts bounds on property *P*, but does not set an upper bound on the property value. An upper bound is required, however.
	Reasons	Error in the assertion file.
	Solution	Correct the assertion file.
Property CPP should be asserted as 0 or 1, not *N*	*Problem*	The user's assertion file gives a value N for the property "ccp" which is illegal. This property takes values 0 or 1 only.
	Reasons	Error in the assertion file.
	Solution	Correct the assertion file.
Return by offset *B* from strange state *S*	*Problem*	The program contains a return sequence which specifies an offset *B* to the normal return address, but the context (state) *S* of this return sequence is unexpected.
	Reasons	Not known.
	Solution	Inform Tidorum.

Error Message		Meaning and Remedy
Return too near end of DO UNTIL loop	*Problem*	The program contains a **DO UNTIL** loop which contains an **RTS** instruction, but the program is illegal because this **RTS** is to close to the end of the loop.
	Reasons	The program is illegal.
	Solution	Correct the program.
Segment index out of range	*Problem*	Some entity in the COFF file refers to an ADSP-21020 segment which is not described in the architecture file.
	Reasons	Inconsistency between COFF file and architecture file.
	Solution	Most likely, the architecture file needs to be extended with information for this segment.

Table 16 below describes the error messages for problems detected in the COFF input file. For all these error messags, the possible reasons are either that the COFF file is corrupted, or that the program (the compiler, assembler, or linker) that generated the COFF file follows different rules. The solution is either to live with the problem or to obtain a COFF file which follows the normal structure and syntax of symbol-table entries.

Table 16: Error Messages for COFF Problems

Error Message		Meaning
Cannot read COFF file	*Problem*	The COFF file exists but could not be read, perhaps because read permission is not granted to the current user.
COFF Aux_EOB out of context	*Problem*	The COFF symbol table contains an "auxiliary entry" in an unexpected position.
COFF *N* symbol *I* has *A* unexpected auxiliary symbols; skipped	*Problem*	The COFF symbol table has, at position *I*, a symbol of the type *N* which is not expected to have "auxiliary"entries, but which in fact has *A* such entries. Bound-T cannot assign meanings to these auxiliary entries and therefore skips and ignores them.
COFF file not found	*Problem*	The COFF file could not be opened because a file of the given name does not exist.
Skipping symbol *I* : *H*	*Problem*	The COFF symbol table contains, at position *I*, an entry of an unexpected form. Bound-T cannot assign a meaning to this entry and therefore skips and ignores it. The octet sequence of the entry is displayed in hexadecimal as *H*.
Unexpected end of COFF file	*Problem*	The COFF file ends unexpectedly (at some unspecified position).
Unexpected end of COFF file while reading section "*S*" from IO index *I*	*Problem*	The COFF file ends unexpectedly at octet position *I*, leaving the section named *S* incomplete.

Error Message		Meaning
Unknown COFF C_Block symbol : *S*	*Problem*	The COFF symbol table contains a "block bracketing" symbol *S*, which is not one of the standard bracket symbols ".bb" or ".eb". Bound-T cannot assign a meaning to this symbol, and therefore may misunderstand the symbol table.
Unknown COFF C_Fcn symbol : *S*	*Problem*	The COFF symbol table contains a "function bracketing" symbol *S*, which is not one of the standard bracket symbols ".bf" or ".ef". Bound-T cannot assign a meaning to this symbol, and therefore may misunderstand the symbol table.

Table 17 below describes the error messages related to the architecture file, when the command-line option *-arch* is used to include such a file in the analysis. Note that the "code location" field (field 5) of these error messages shows the name of the architecture file and the number of the line in the architecture file at which the error was detected. Moreover, the error message text (in field 6) begins with the column number at which the error was detected, in parentheses, and ends with the lexical token from the architecture file at which the error was detected, also in parentheses (sometimes this is the token immediately following the erroneous token). The table shows only the core of the error message and omits the parts giving the column number and the lexical token.

For all these error messages, the reason is an error in the architecture file, and the solution is to correct the architecture file.

Table 17: Error Messages for Architecture File Problems

Error Message		Meaning
Bank qualifier expected	*Problem*	One of the qualifiers of a ".bank" directive (the text following a "/") begins with something that is not a recognized bank qualifier name such as "wtmode".
Bank selector missing	*Problem*	In a ".bank" directive, there is no bank selector, that is, the directive does not specify PM0, PM1, DM0, DM1, DM2, or DM3. Every ".bank" directive must select one of these memory banks.
Delimiter ('/') expected	*Problem*	1. In a ".segment" directive, where a qualifier starting with '/' is expected, something else was found. 2. In a ".bank" directive, where a qualifier starting with '/' is expected, something else was found.
Duplicate bank selection	*Problem*	A ".bank" directive contains more than one bank selector, for example .../PM0/.../PM1/... This is a non-fatal error; Bound-T continues parsing the architecture file. If there are no fatal errors, the analysis is executed using the last specified (rightmost) bank selector.

Error Message		Meaning
Invalid literal (*L*)	*Problem*	A literal number is expected at this point in the architecture file. The string *L* was found, but it is not a valid decimal or hexadecimal number. Perhaps it is meant to be a hexadecimal literal, but the prefix "0x" was omitted.
Invalid width (*W*)	*Problem*	A "width" qualifier in a ".segment" directive specifies a width *W* (in bits) that is not one of the acceptable widths (16, 32, 40, or 48 bits).
Literal number expected	*Problem*	A literal number is expected at this point in the architecture file, but something else was found.
Page size out of range (*S*)	*Problem*	A "pgsize" qualifier in a ".bank" directive specifies a page size *S* that is negative or too large (over 2^{15} = 32768).
Processor name expected	*Problem*	The ".processor" directive keyword is followed by something that is not a "name", lexically speaking. Note that kewords such as "PM0" are not considered "names", and should therefore not be used as the processor name in a ".processor" directive.
Processor, segment, bank or endsys directive expected	*Problem*	At this point in the architecture file, some directive is expected, but the text actually found is not recognized as a directive.
Segment name not defined	*Problem*	In a ".segment" directive, the name of the segment is not given. Each ".segment" directive must define a segment name.
Segment qualifier expected	*Problem*	One of the qualifiers of a ".segment" directive (the text following a "/") begins with something that is not a recognized segment qualifier name such as "begin".
Semicolon expected	*Problem*	A semicolon was expected at this point in the (syntax of) the architecture file, but something else (the token displayed) was found.
System directive expected	*Problem*	The architecture file should start with the ".system" directive, but this file starts with something else (or is entirely empty).
System name expected	*Problem*	The ".system" directive keyword is followed by something that is not a "name", lexically speaking. Note that kewords such as "PM0" are not considered "names", and should therefore not be used as the system name in a ".system" directive.
Unexpected end of architecture file	*Problem*	The architecture file ends in the middle of a syntactic construct, but seems correct up to that point. This error message should never appear, because the end of the text is always detected as the absence of some expected lexical token, reported as such, and the parsing of the file is then aborted.

Error Message		Meaning
Wait mode expected	*Problem*	The value specified in a "wtmode" qualifier of a ".bank" directive is not recognized as one of the known modes ("internal", "external", "neither", "either", "both").
Wait states out of range (*W*)	*Problem*	The value *W* specified in a "wtstates" qualifier of a ".bank" directive is negative or greater than 7.

Tidorum Ltd

Tiirasaarentie 32
FI-00200 Helsinki
Finland

www.tidorum.fi
info@tidorum.fi
Tel. +358 (0) 40 563 9186
Fax +358 (0) 42 563 9186
VAT FI 18688130