

Bound-T Application Note ADSP-21020 TSC-21020

Version 1

June 2001



SPACE SYSTEMS
F I N L A N D

Space Systems Finland Ltd

www.ssf.fi

Kappelitie 6

FIN-02200 ESPOO

Finland

This document was written at Space Systems Finland Ltd. by Niklas Holsti, Thomas Långbacka and Sami Saarinen.

The document is currently maintained by the same team.

Copyright 2000, 2001 Space Systems Finland Ltd.

This document can be copied and distributed freely, in any format, provided that it is kept entire, with no deletions, insertions or changes, and that this copyright notice is included, prominently displayed, and made applicable to all copies.

Document reference: DET-SSF-MA-002
Document issue: 1
Document issue date: 2001-06-20
Bound-T version: 1
Web location: <http://www.bound-t.com/app-notes/21020>

Trademarks:
Bound-T is a trademark of Space Systems Finland Ltd.

Preface

The information in this document is believed to be complete and accurate when the document is issued. However, Space Systems Finland 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 boundt@ssf.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, and office hours are 9:00 -16:00 local time.

Cordially,

Space Systems Finland Ltd.

Telephone: +358 9 6132 8600
Fax: +358 9 6132 8699
Web: <http://www.ssf.fi>

Mail: Kappelitie 6
FIN-02200 ESPOO
Finland

Credits

The *Bound-T* tool was first developed 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.

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

This page is blank on purpose

CONTENTS

Chapter 1	Introduction	
1.1	Purpose and Scope	1
1.2	Overview	1
1.3	References	2
1.4	Abbreviations and Acronyms	2
Chapter 2	The 21020 and Timing Analysis	
2.1	The ADSP-21020	3
2.2	Static Execution Time Analysis on the 21020	3
Chapter 3	Supported 21020 Features	
3.1	Overview	4
3.2	Levels of Support	4
3.3	Implications of Limited Support	6
3.4	Reminder of Generic Limitations	6
3.5	Support Synopsis	7
3.6	Data Registers and Memory Accesses	9
3.7	Registers and the C Calling Protocol	10
3.8	Modes, System Registers, Condition Codes	10
3.9	Computational Operations	11
3.10	Instructions	13
3.11	Program Sequencing Registers	14
3.12	Other Registers	14
3.13	Time Accuracy and Approximations	14
Chapter 4	Using Bound-T 21020	
4.1	Input Formats	20
4.2	Command Arguments and Options	20
4.3	HRT Analysis	22
4.4	Choice of Calling Protocol	22
4.5	Basic Output Format Limitations	23
4.6	Warning Messages	23
4.7	Error Messages	26
Chapter 5	Writing Assertions	
5.1	Naming Scopes	29
5.2	Naming Subprograms	30
5.3	Naming C Variables	30

5.4	Naming Assembler Variables	31
5.5	Naming Statement Labels	31
5.6	Naming Items by Address	31
5.7	Properties	32

LIST OF TABLES

Table 1: Definition Analysis vs Arithmetic Analysis 5
Table 2: Generic Limitations of Bound-T 7
Table 3: Synopsis of 21020 Support 8
Table 4: DAGs Loaded and Used by an Instruction 15
Table 5: Effect of Memory Wait States on Execution Time 17
Table 6: Approximations for Instruction Times 18
Table 7: Naming Scopes 29

This page is blank on purpose

1 Introduction

1.1 Purpose and Scope

Bound-T is a tool for computing bounds on the worst-case execution time of real-time programs; see reference [1]. There are different versions of Bound-T for different target processors. This Application Note supplements the Bound-T User Manual [1] 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 [3].

Some information in Chapters 4 and 5 of 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 21020”.

A Bound-T version for the ADSP-2106x series, also known as the SHARC, is not available at the time of writing, but it is likely that one will be developed in the near future. Please contact Space Systems Finland Ltd. for more 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 User Manual [1]. The user manual also 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.

This Application Note explains how Bound-T has been adapted to the architecture of the 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 [3] and [4].

The remainder of this Application Note is structured as follows:

- Chapter 2 describes the main features of the 21020 architecture and how they relate to the functions of Bound-T.
- Chapter 3 defines in detail the set of 21020 instructions and registers that is supported by Bound-T.
- Chapter 4 explains those Bound-T command arguments and options that are wholly specific to the 21020 or that have a specific interpretation for this processor.
- Chapter 5 addresses the user-defined assertions on target program behaviour and explains the possibilities and limitations in the context of the ADSP-21020 and the Analog Devices development tools.

1.3 References

- [1] Bound-T User Manual.
Space Systems Finland Ltd., Doc.ref. DET-SSF-MA-001.
- [2] Bound-T Application Note: Virtuoso.
Space Systems Finland Ltd., Doc.ref. DET-SSF-MA-004.
- [3] ADSP-21020 User's Manual.
Analog Devices Inc. Second Edition, 1995
- [4] ADSP-21000 Family, C Tools Manual.
Analog Devices Inc. Third Edition, 1995.

1.4 Abbreviations and Acronyms

See also reference [1] for abbreviations specific to Bound-T and reference [3] for the mnemonic operation codes and register names of the 21020.

ADI	Analog Devices Inc.
CCP	C Calling Protocol
DAG	Data Address Generator [3]
DSP	Digital Signal Processor
PCSP	PC Stack Protocol
TBA	To Be Added
TBC	To Be Confirmed
TBD	To Be Determined
WCET	Worst-Case Execution Time

2 The 21020 and Timing Analysis

2.1 The ADSP-21020

The 21020 [3] 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 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 [4] which is also often used by assembly-language libraries, at least when designed to interface with C programs.

2.2 Static Execution Time Analysis on the 21020

The 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 UNTIL loops with few iterations.
- Memory wait states that vary in number depending on the address.

See section 3.13 for more information about the approximations.

3 Supported 21020 Features

3.1 Overview

This section specifies which 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 21020, Bound-T also has generic limitations as described in the User Manual [1]. For reference, these are briefly listed in section 3.4.

General support level

In general, when Bound-T is analysing a target program for the 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 very uncommon exceptions. Bound-T supports both the processor's internal call/return protocol, using the CALL, RTS, and RTI instructions, and the C Calling Protocol ([4], 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.

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 21020 target system.

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

3.2 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 Bound-T 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, the following table lists some 21020 instructions in the leftmost column and their definition-analysis and arithmetic analysis in the two other columns. The instructions are assumed to be executed in sequence. The analysis contains just the aspects supported by Bound-T.

Table 1: 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, AC all get the value 0.
4	$R7 = R4 * R5$	R7 gets a new value.	R7 gets an opaque value (Bound-T does not support multiplication in arithmetic analysis).
5	$MRF = R4 * R5$	No effect. The MRF register is not tracked.	No effect.
6	$R5 = RND\ MRF$	R5 gets a new value.	R5 gets an opaque value (any reading of an unsupported register 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 $R5 \geq R4$ 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. 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.

3.3 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 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 1, 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 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 1 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 “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.

3.4 Reminder of Generic Limitations

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

Table 2: Generic Limitations of Bound-T

Generic Limitation	Remarks for 21020 target
Understands only integer operations in loop-counter computations.	All results from floating-point operations can be considered opaque.
Understands only addition, subtraction and multiplication by constants, in loop-counter computations.	The multiplier and shifter can be modelled very skimpily.
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 21020, although the 21020 instructions for zero-overhead loops may guide programmers to use counter-based loops more frequently.
May not resolve aliasing in dynamic memory addressing.	No implications specific to the 21020.

3.5 Support Synopsis

The following table gives a synoptical view of the level of support for 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 3: Synopsis of 21020 Support

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.
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.
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.
Alternate (secondary) registers as controlled by MODE1 bits. Memory access with 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.

Table 3: Synopsis of 21020 Support

21020 registers, instructions, or other features	Decoding	Control flow	Definition	Arithmetic	Remarks
21060 instructions					To be implemented in future version.

3.6 Data Registers and Memory Accesses

The 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 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.

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 (R_i corresponds to F_i). 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 21020 index registers (I0 - I15) is fully supported.

Modify registers (M0 - M15) are supported fully.

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 in the definition level. The use of an index register with post-modification when the corresponding length register is non-zero leads to storing an opaque value to the index register.

Universal Register addressing

The 21020 has a mechanism of “universal register addressing” using an 8-bit address to define the source and destination for some data-moving instructions ([3], 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 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.

3.7 Registers and the C Calling Protocol

The C Calling Protocol (CCP) ([4], 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 4.4). The CCP rules are the following.

Firstly, CCP divides the 21020 registers into two subsets: *compiler registers* that are preserved across any call, and *scratch registers* that need not be preserved. Bound-T supports 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 will check 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 as background knowledge in the arithmetic analysis.

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

3.8 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 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 as opaque in ordinary conditional instructions, as is LCE in that context.

Direct assignment to the ASTAT register via universal register addressing or system-register bit-manipulation is understood as storing opaque values in the status flags.

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

3.9 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 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:

$R_n = R_x + R_y$
 $R_n = R_x - R_y$
 $R_n = R_x + R_y + CI$
 $R_n = R_x - R_y + CI - 1$
 COMP(R_x , R_y)
 $R_n = R_x + CI$
 $R_n = R_x + CI - 1$
 $R_n = R_x + 1$
 $R_n = R_x - 1$
 $R_n = -R_x$
 $R_n = PASS R_x$
 $R_n = ABS R_x$

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 ($R_n = [R_x + R_y] / 2$). 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 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 3.8).

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.

3.10 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 [3].

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 3.9.

All the “immediate shift” instructions are supported on the definition level, and some are supported on the arithmetic level; see section 3.9.

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.

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 3.6, 3.11 and 3.12.

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 (without making it incorrect).

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 (well it’s not very hard is it!).

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.

3.11 Program Sequencing 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.

3.12 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 a command-line option.

3.13 Time Accuracy and Approximations

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

DAG and Memory Control Register Writes

According to section 7.2.1.5 of [3], the 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 [3] does not say exactly which instructions have these properties. Bound-T follows the behaviour of the ADI simulator for the 21020. This behaviour is shown

in Table 4 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 4: DAGs Loaded and Used by an Instruction

Instruction type and page in [3]	Loads DAGs	Uses DAGs
<i>compute / dreg ↔ DM / dreg ↔ PM</i> A-12	none	1 and 2
<i>compute / ureg ↔ DM PM, register modify</i> A-14	1 if target <i>ureg</i> is in DAG1 2 if target <i>ureg</i> is in DAG2 else none	1 if <i>DM</i> 2 if <i>PM</i>
<i>compute / dreg ↔ DM PM, immediate modify</i> A-16	none	1 if <i>DM</i> 2 if <i>PM</i>
<i>compute / ureg ↔ ureg</i> A-18	1 if target <i>ureg</i> is in DAG1 2 if target <i>ureg</i> is in DAG2 else none	none (even if the source <i>ureg</i> is in a DAG)
<i>immediate shift / dreg ↔ DM PM</i> A-20	none	1 if <i>DM</i> 2 if <i>PM</i>
<i>compute / modify</i> A-22	none	1 if DAG1 register is modified 2 if DAG2 register is modified
<i>indirect jump/call / compute</i> A-26	none	2 if indirect (via DAG2) else none (if PC-relative)
<i>do until counter expired</i> A-30	none	none (even if <i>LCNTR</i> is initialized from a DAG register)
<i>ureg ↔ DM PM (direct addressing)</i> A-34	1 if target <i>ureg</i> is in DAG1 2 if target <i>ureg</i> is in DAG2 else none	none
<i>ureg ↔ DM PM (indirect addressing)</i> A-35	1 if target <i>ureg</i> is in DAG1 2 if target <i>ureg</i> is in DAG2 else none	1 if <i>DM</i> 2 if <i>PM</i>
<i>immediate data → DM PM</i> A-36	none	1 if <i>DM</i> 2 if <i>PM</i>
<i>immediate data → ureg</i> A-37	1 if target <i>ureg</i> is in DAG1 2 if target <i>ureg</i> is in DAG2 else none	none
<i>l register modify / bit-reverse</i> 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 extreme 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 (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 21020 simulator adds this NOP cycle to the callee’s last instruction.

Timing of Conditional Instructions

Many 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 [3] is not entirely clear on how the value of the condition affects the execution time of a conditional instruction. Based on experiments with the ADI simulator for the 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 4) 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 4) does so always, whether its condition (if any) is true or false.

Memory Wait States

The User’s Manual [3] 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 var-

ious types is shown in Table 5 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 5: Effect of Memory Wait States on Execution Time

Instruction type in terms of the memory 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 <i>false</i>	$2 + f$
Access PM only, condition <i>true</i>	$2 + f + p$
Access both DM and PM	$2 + f + \max(p, d)$

Table 5 assumes that the instruction is not involved in a DAG load/use conflict (Table 4). To handle a conflict, Bound-T adds one cycle to the execution time of the instruction pair (as computed from Table 5) whatever the number of wait states (since the delay is internal to the processor).

Table 5 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.

Summary of Approximations

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

Table 6: 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 ([3], section 3.2.2). Bound-T assumes conservatively that the cache 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 UNTIL loops with few iterations	Short UNTIL loops with few iterations may require some delay cycles to terminate and fetch the next instruction ([3], 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 if the upper bound on the number of iterations is small enough.	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 ([3], 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 [3], 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 4.)	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 (Table 4) 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.	1 cycle per call
Memory wait states that vary between memory banks	In the 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. 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 5.7. Since Bound-T reads the target-program's architecture file (.ach file), it could use the memory specifications it contains, but this is not yet implemented in the current version.	Depends on user-given values. See Table 5.
Additional memory wait states at page boundaries	The 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, as above.	Depends on user-given values. See Table 5.

Table 6: Approximations for Instruction Times

Case	Description	Maximum Error
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 ([3], 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 5.

4 Using Bound-T 21020

4.1 Input Formats

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 MS Windows, and big-endian with 20-byte symbol records, as produced (TBC) 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.

4.2 Command Arguments and Options

The generic Bound-T command format, options and arguments apply without modification to the 21020 version of Bound-T.

There are additional 21020-specific options as explained in the table below. Note that a target-specific option must be written as one string with no embedded blanks, so the option-name and its numeric parameter, if any, are contiguous. For example, the form “`-dm_read_ws3`” is correct, “`-dm_read_ws 3`” is not.

Option		Meaning and default value
-archX	<i>Function</i>	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.
	<i>Default</i>	A default architecture (defined within Bound-T, not in an .ach file) as explained below.
-coff_trace	<i>Function</i>	COFF elements are displayed on standard output as they are read from the target program file. May help to understand COFF problems. If this option is selected when no root subprograms are named on the command line, the COFF data are displayed twice: once while reading them, and once after the whole file has been read, as usual when no root subprograms are given.
	<i>Default</i>	No trace.
-coff_unix	<i>Function</i>	Indicates that the COFF file comes from the ADI Unix tools and is big-endian with 20-byte symbol records.
	<i>Default</i>	The COFF file is assumed to come from the ADI Windows tools and be little-endian with 18-byte symbol records.

Option		Meaning and default value
-dm_read_wsX	<i>Function</i>	Sets the number <i>X</i> of memory wait states assumed for a data-memory read, if the memory location is not certain to be in the stack. This value is overridden by user assertions.
	<i>Default</i>	Zero wait states (-dm_read_ws0).
-dm_write_wsX	<i>Function</i>	Sets the number <i>X</i> of memory wait states assumed for a data-memory write, if the memory location is not certain to be in the stack. This value is overridden by user assertions.
	<i>Default</i>	Zero wait states (-dm_write_ws0).
-stack_read_wsX	<i>Function</i>	Sets the number <i>X</i> of memory wait states assumed for a read from the stack, that is from a data-memory address which is <i>i6</i> or <i>i7</i> plus an offset. This value is overridden by user assertions.
	<i>Default</i>	Zero wait states (-stack_read_ws0).
-stack_write_wsX	<i>Function</i>	Sets the number <i>X</i> of memory wait states assumed for a write to the stack, that is to a data-memory address which is <i>i6</i> or <i>i7</i> plus an offset. This value is overridden by user assertions.
	<i>Default</i>	Zero wait states (-stack_write_ws0).
-pm_read_wsX	<i>Function</i>	Sets the number <i>X</i> of memory wait states assumed for a read of data from the program memory. This value is overridden by user assertions.
	<i>Default</i>	Zero wait states (-pm_read_ws0).
-pm_write_wsX	<i>Function</i>	Sets the number <i>X</i> of memory wait states assumed for a write of data to the program memory. This value is overridden by user assertions.
	<i>Default</i>	Zero wait states (-pm_write_ws0).
-fetch_wsX	<i>Function</i>	Sets the number <i>X</i> of memory wait states assumed for an instruction fetch from program memory. This value is overridden by user assertions.
	<i>Default</i>	Zero wait states (-fetch_ws0).

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.

4.3 HRT Analysis

For HRT analysis, the 21020 is usually run with the Virtuoso kernel from Eonic Systems. Please refer to the separate Bound-T Application Note discussing Virtuoso [2]; there are no specific considerations for the 21020.

4.4 Choice of Calling Protocol

The definition-analysis and (especially) arithmetic analysis of a subprogram depend on the calling protocol of the subprogram. The C Calling Protocol (CCP) enforces a register discipline that considerably assists these analyses, as explained in section 3.7.

Bound-T chooses the protocol automatically as follows:

- All “root” subprograms (that is, the subprograms named on the command line) are assumed to follow the CCP.
- Any other subprogram is analysed 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 not.

The only alternative to the CCP is the PC Stack Protocol (PCSP), the native 21020 call/return protocol which uses the instructions CALL and RTS, and has no assumptions on register usage. Bound-T emits an error 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 5.7).

4.5 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 [4] 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.

4.6 Warning Messages

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

The specific warning messages refer mainly to unsupported or approximated features of the 21020.

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.

Warning Message		Meaning and Remedy
Alternate registers not supported	<i>Reasons</i>	The instruction assigns to MODE1, which might switch between alternate and primary registers.
	<i>Action</i>	The user is responsible for evaluating the impact of alternate register set usage. Since it amounts to an unanalysed change to all the affected register cells, it can invalidate the loop-bounding analysis.
Arithmetic effect of Push/Pop is not modelled.	<i>Reasons</i>	The instruction manipulates loop counter or status stacks and its effect is not modelled.
	<i>Action</i>	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion.
Bit-reversal mode for I0 not supported	<i>Reasons</i>	The instruction assigns to MODE1, which might activate bit-reversal mode for I0.
	<i>Action</i>	The user is responsible for evaluating the impact of bit-reversal mode for I0. It can invalidate any analysis that Bound-T applies to I0 or indirectly through I0.
Call to <code>_exit</code> is converted to return	<i>Reasons</i>	The instruction calls subprogram <code>_exit</code> and is treated as a return instruction.
	<i>Action</i>	Note that the subprogram <code>_exit</code> is not analysed by Bound-T, and its execution time is not included in the reported WCET results.

Warning Message	Meaning and Remedy	
Call to <code>_exit</code> with non-empty loop stack	<i>Reasons</i>	The instruction calls subprogram <code>_exit</code> , but still has live DO UNTIL loops in stack.
	<i>Action</i>	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion.
CCP -call to <code>_exit</code> is converted to return	<i>Reasons</i>	The instruction calls subprogram <code>_exit</code> and is treated as a return instruction.
	<i>Action</i>	Note that the subprogram <code>_exit</code> is not analysed by Bound-T, and its execution time is not included in the reported WCET results.
Condition NOT LCE with no containing loop considered opaque	<i>Reasons</i>	The instruction uses the LCE (loop counter not expired) condition with no containing loop.
	<i>Action</i>	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.
Data access in Program Memory	<i>Reasons</i>	The instruction accesses data in Program Memory.
	<i>Action</i>	Note that the estimated WCET includes one cache-miss cycle for each execution of this instruction, although the cache may sometimes hit.
DO UNTIL LCE without counter init considered opaque	<i>Reasons</i>	DO UNTIL loop uses loop counter as the end condition without initialising the counter.
	<i>Action</i>	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	<i>Reasons</i>	A short DO UNTIL loop was identified.
	<i>Action</i>	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	<i>Reasons</i>	There is a DO UNTIL instruction too close to the end of another DO UNTIL loop.
	<i>Action</i>	Check the binary file.
Dynamic control flow unbounded	<i>Reasons</i>	Bound-T was unable to resolve a dynamic branch instruction.
	<i>Action</i>	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion.
I6 modified outside call/return	<i>Reasons</i>	The instruction modifies I6 but is not contained in a call or return sequence.
	<i>Action</i>	The user is responsible for evaluating the impact.
Idling time ignored	<i>Reasons</i>	The subprogram contains an IDLE instruction, for which the idling time cannot be known.
	<i>Action</i>	Note that the computed WCET for this subprogram contains no contribution from the idling time.

Warning Message	Meaning and Remedy	
Immediate modifier too large for Program Memory.	<i>Reasons</i>	The modifier value, that is used with a Program Memory index register, is too large. Only the 24 least significant bits of the value are used.
	<i>Action</i>	The user is responsible for evaluating the impact.
Improper change to stack pointer	<i>Reasons</i>	The instruction has modified stack pointer in a way that cannot be analysed and the value is lost.
	<i>Action</i>	Note that the use of stack is unanalysable after this instruction.
Jump to <code>_exit</code> is converted to return	<i>Reasons</i>	The instruction jumps to (or calls) subprogram <code>_exit</code> and is treated as a return instruction.
	<i>Action</i>	Note that the subprogram <code>_exit</code> is not analysed by Bound-T, and its execution time is not included in the reported WCET.
Jump to <code>_exit</code> with non-empty loop stack	<i>Reasons</i>	The instruction jumps to subprogram <code>_exit</code> , but still has live DO UNTIL loops in stack.
	<i>Action</i>	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion.
Large immediate data interpreted as signed (negative)	<i>Reasons</i>	The sign of a large value has been interpreted 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, or as the address of a memory location near the end of memory.
	<i>Action</i>	The user is responsible for evaluating the impact.
Large immediate data interpreted as unsigned (positive)	<i>Reasons</i>	The sign of a large value has been interpreted by Bound-T, but it might be wrong.
	<i>Action</i>	The user is responsible for evaluating the impact.
No real function symbols found, scanning all symbols. Some variable symbols may appear as subprograms.	<i>Reasons</i>	The binary file does not contain subprogram symbols and all symbols are considered as a possible subprogram symbol.
	<i>Action</i>	Note that some variable symbols may appear as subprograms.
Program Memory cell address is too large.	<i>Reasons</i>	A memory access to Program Memory has an address that is out of bounds of the memory space.
	<i>Action</i>	The user is responsible for evaluating the impact.
Program Sequencing register modified	<i>Reasons</i>	The instruction modifies some Program Sequencing register via Universal Register addressing, which can change the control-flow in an unknown way.
	<i>Action</i>	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion, since even the control-flow analysis may be invalidated.

Warning Message		Meaning and Remedy
Return within DO UNTIL loop	<i>Reasons</i>	The instruction is a return with some live DO UNTIL loops in the stack.
	<i>Action</i>	The user is responsible for evaluating the impact on the results, and should treat the results with extreme suspicion.
Section has relocation entries, perhaps file is not linked	<i>Reasons</i>	A section in the binary has relocation entries and might not be linked.
	<i>Action</i>	The user is responsible for evaluating the impact.
The section is absent. Constraint_Error will occur if this section is used.	<i>Reasons</i>	The binary file contains a header to a section, but no body for it.
	<i>Action</i>	If the Constraint_Error occurs, check the binary file.
Unbounded dynamic memory access	<i>Reasons</i>	Bound-T was not able to resolve a dynamic data access.
	<i>Action</i>	The user is responsible for evaluating the impact.
Unbounded stack push	<i>Reasons</i>	Bound-T was not able to resolve a stack usage.
	<i>Action</i>	The user is responsible for evaluating the impact.

4.7 Error Messages

The following lists the Bound-T error messages that are specific to the 21020 or that have a specific interpretation for this processor. The messages are listed in alphabetical order. The User Manual explains the generic error messages, all of which may appear also when the 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.

Error Message		Meaning and Remedy
Branch without loop abort within loop delay	<i>Problem</i>	There is a branch instruction without Loop Abort too near to the loop's end.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
Calling protocol ambiguous	<i>Problem</i>	For this subprogram, there are both CCP-style calls and PCSP-style calls, so it is unclear which calling protocol the subprogram follows.

Error Message	Meaning and Remedy	
	<i>Reasons</i>	Probably this is a mistake in program.
	<i>Solution</i>	Correct the program to use one or the other protocol, but never both for the same subprogram.
Cannot read file	<i>Problem</i>	The binary file cannot be read.
	<i>Reasons</i>	Perhaps the access rights of the file does not allow you to read the file.
	<i>Solution</i>	Check that you are allowed to read the file.
Constant cell modified in a CCP - subprogram	<i>Problem</i>	A subprogram that is assumed to follow CCP -protocol does not follow it.
	<i>Reasons</i>	There is a CCP -call to a subprogram that does not follow the CCP -protocol internally.
	<i>Solution</i>	Add a "CCP 0" property for this subprogram into the assertion file.
Data Memory address exceeds DM section bounds	<i>Problem</i>	A data object is accessed that is out of range of the section that is supposed to contain it.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
DO UNTIL nested too deeply	<i>Problem</i>	DO UNTIL loop stack has overflowed.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
DO UNTIL with zero offset	<i>Problem</i>	DO UNTIL instruction tries to create a zero length loop.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
Dynamically determined call is converted to _exit call	<i>Problem</i>	Target subprogram of this call is not going to be analysed.
	<i>Reasons</i>	Bound-T is unable to analyse dynamic calls.
	<i>Solution</i>	Analyse the called subprogram separately and insert an assertion of the WCET of the call into the assertion file.
Illegal Dual Add/Subtract instruction	<i>Problem</i>	The instruction has an illegal compute part.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
Illegal instruction	<i>Problem</i>	The instruction is not a valid 21020 instruction.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
Instruction address exceeds PM section bounds.	<i>Problem</i>	An instruction is tried to be fetched from outside the code area.

Error Message	Meaning and Remedy	
	<i>Reasons</i>	The control flow of a subprogram goes out of range of the code segment provided by the binary file.
	<i>Solution</i>	Obtain a correct COFF file.
Invalid CU field in a single compute operation	<i>Problem</i>	The instruction tries to use an illegal computation unit.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
Invalid Parallel Multiplier & Dual Add/Subtract instruction	<i>Problem</i>	The instruction has an illegal compute part.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
Loop Abort outside loop.	<i>Problem</i>	There is a branch instruction with Loop Abort outside any DO UNTIL loop.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
Overflow or underflow in 6-bit (or 24-bit) PC-relative address.	<i>Problem</i>	A Program Counter relative branching instruction causes the target address to overflow (address is above the maximum size of Program Memory) or to underflow (address is below zero).
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
Program Memory datum address exceeds PM section bounds	<i>Problem</i>	A data object is accessed that is out of range of the section that is supposed to contain it.
	<i>Reasons</i>	A mistake in the program.
	<i>Solution</i>	Obtain a correct COFF file.
Unexpected end of file	<i>Problem</i>	The binary file is not complete.
	<i>Reasons</i>	The COFF format is inconsistent.
	<i>Solution</i>	Obtain a correct COFF file.

5 Writing Assertions

This chapter explains any specific limitations and possibilities for user-specified assertions when Bound-T is used with 21020 programs. Most of these issues are not caused by the 21020 as target processor, but by the Analog Devices development tools [4].

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 21020 programs are also listed at the end of this chapter.

5.1 Naming Scopes

The COFF file contains much symbolic information for debugging purposes. The COFF standard does not directly support a hierarchical (block-structured) namespace. Bound-T uses the source-file information to create scopes for symbols, as shown in the following table.

Table 7: 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 Local variable	Source file	Subprogram	Parameter Variable	N/A
Parameter passed in register	Source file	Subprogram	“Regparm”	Parameter

Note that the C compiler will prefix each subprogram or variable name 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 keep them apart.

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;
    ...
    return ...
}
```

The C-level identifiers in this code will be accessible as the following symbols, where ‘|’ 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)

5.2 Naming Subprograms

The ADI C toolchain 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”.

There is one level of scope, which contains the source-file name. However, for assembler subprograms it seems that this is usually not the real source-file name, but the name of some temporary file such as */var/tmp/cca00*.

5.3 Naming C Variables

The ADI C toolchain 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, since all global variables have one level of scope containing the source-file name.

For formal and actual parameters, we assume that the optimised code uses the same passing mechanism (CCP) as the unoptimised code, and thus the symbolic information from the unoptimised executable applies. A future version of the tool may support using an unoptimised executable to analyse an optimised executable.

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 (with a third, synthetic “Regparm” level 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.

5.4 Naming 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 it 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 if it were a subprogram.

5.5 Naming Statement Labels

The current version of Bound-T for the 21020 does not extract statement labels that have been defined in C code. This will be corrected in the near future.

Statement labels in assembly-language programs are entirely equivalent to subprogram names, and Bound-T uses them as such. In a future version, all such symbols will be defined both as subprogram symbols and as label symbols.

Thus, currently Bound-T for the 21020 provides no statement label symbols.

5.6 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 are 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 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:

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

variable address "dm3fa7" 20;
-- DM at address hex 3FA7 = decimal 16295.
```

5.7 Properties

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

Property name		Meaning, values and default value
<i>dm_read_ws</i>	<i>Function</i>	Changes the number of data-memory read wait states in the current context.
	<i>Values</i>	Number of wait state cycles.
	<i>Default</i>	Zero wait states or the value given in a command-line option <i>-dm_read_ws</i> .
<i>dm_write_ws</i>	<i>Function</i>	Changes the number of data-memory write wait states in the current context.
	<i>Values</i>	Number of wait state cycles.
	<i>Default</i>	Zero wait states or the value given in a command-line option <i>-dm_write_ws</i> .
<i>stack_read_ws</i>	<i>Function</i>	Changes the number of stack read wait states in the current context.
	<i>Values</i>	Number of wait state cycles.
	<i>Default</i>	Zero wait states or the value given in a command-line option <i>-stack_read_ws</i> .
<i>stack_write_ws</i>	<i>Function</i>	Changes the number of stack write wait states in the current context.
	<i>Values</i>	Number of wait state cycles.
	<i>Default</i>	Zero wait states or the value given in a command-line option <i>-stack_write_ws</i> .
<i>pm_read_ws</i>	<i>Function</i>	Changes the number of program-memory data read wait states in the current context.
	<i>Values</i>	Number of wait state cycles.
	<i>Default</i>	Zero wait states or the value given in a command-line option <i>-pm_read_ws</i> .
<i>pm_write_ws</i>	<i>Function</i>	Changes the number of program-memory data write wait states in the current context.
	<i>Values</i>	Number of wait state cycles.
	<i>Default</i>	Zero wait states or the value given in a command-line option <i>-pm_write_ws</i> .

Property name		Meaning, values and default value
<i>fetch_ws</i>	<i>Function</i>	Changes the number of program-memory instruction fetch wait states in the current context.
	<i>Values</i>	Number of wait state cycles.
	<i>Default</i>	Zero wait states or the value given in a command-line option <i>-fetch_ws</i> .
<i>ccp</i>	<i>Function</i>	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.
	<i>Values</i>	1 - The subprogram follows the CCP-protocol. 0 - The subprogram doesn't follow the protocol.
	<i>Default</i>	The default is determined for each subprogram separately. See section 4.4.