Lab:	WCET Analysis Assignment
Course:	Real-Time Systems, GK
Period:	Autumn 2006

Lab introduction

The purpose of this lab assignment is to give students a chance to practice the theories covered by lecturers and course literature. The assignment will be focusing on static Worst-Case Execution Time (WCET) analysis and on the WCET analysis tool called Bound-T. The assignment will also cover concepts like compiler, linker, assembler, object code, program analysis, execution time, control-flow graph, call graph, and more.

The assignment should be solved individually or in groups of two. The assignment must be handed in no later than Monday 2006-10-30. For more information on report procedures please see the Report section at page 13.

Introduction to WCET analysis

The purpose of a *worst-case execution time* (WCET) analysis is to provide an upper bound on all possible execution times of a computer program. Reliable WCET estimates are a key component in providing timing guarantees of real-time systems, and are especially important when it must be proven that the system will behave correctly even in the most stressful situations. WCET estimates are used in real-time systems development to create schedules and to perform schedulability analysis, to determine whether performance goals are met for periodic tasks, and to check that interrupts have sufficiently short reaction times [Gan06].

For example, in the following response time formula,

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

the C_i value should be a safe upper estimate of the WCET of task τ_i for all its possible inputs and hardware configurations.

The problem that needs to be addressed by any WCET analysis is that a computer program typically has no single fixed execution time. *Variations* in the execution time occur due to the characteristics of the work the program performs and the hardware on which it runs.

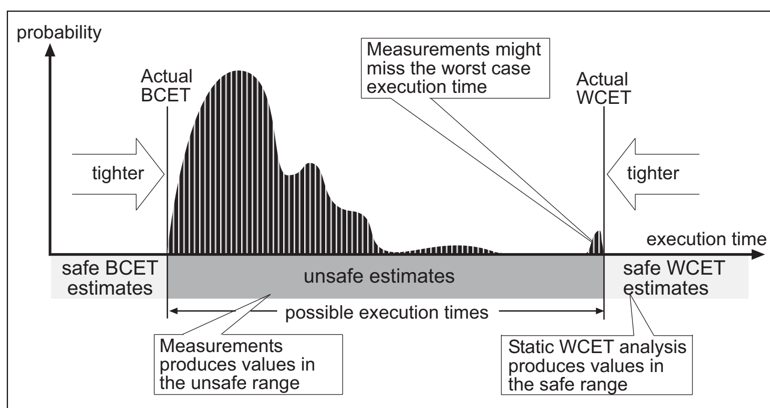
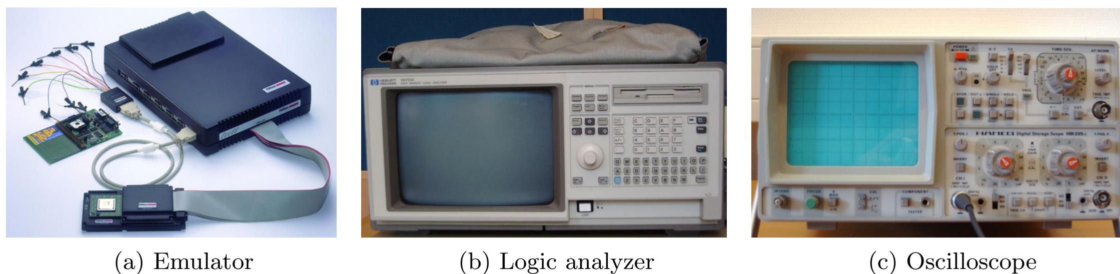


Figure 1: Execution time estimates

The (actual) *worst-case execution time* (WCET) is the longest possible execution time of a program (or task) when the program is run on its target hardware. Similarly, the (actual) *best-case execution time* (BCET) is the shortest possible execution time of a program (or task). The BCET can be of interest in control-applications where the output must be sent to the controlled object neither too soon, nor too late. The *average-case execution time* (ACET) lies somewhere in-between the WCET and the BCET and depends on the execution time distribution of the program.

The goal of a WCET analysis is to produce *bounds* of the WCET (and the BCET). To be valid for use in hard real-time systems, the bounds must be *safe*, i.e. guaranteed not to be less than actual WCET. To be useful, they must also be *tight*, i.e. provide acceptable overestimations compared to the actual WCET. Figure 1 shows how these estimates of WCET and BCET relate to the actual WCET and BCET. The example program has a variable execution time, and the curve shows the probability distribution of its execution time.

Dynamic WCET analysis



(a) Emulator (b) Logic analyzer (c) Oscilloscope

Figure 2: Tools for dynamic timing analysis

Today, the common method (if any) in industry to derive WCET estimates is by measurements, also known as *dynamic timing analysis*. A wide variety of measurement tools are employed, including emulators, logic analyzers, oscilloscopes, simulators, and software profiling tools [Ste04] (see Figure 2 for an illustration). The methodology is basically the same for all approaches: run the same program many times and try different potentially ”really bad” input values to provoke the WCET.

This is time-consuming and difficult work, and even worse, it is hard to guarantee that the actual WCET has been found. Furthermore, each measurement run only gives the execution time for a single execution path. If the number of possible test-cases are too large it might be impossible to perform exhausive testing. In many cases it might also be problematic to set up a testing environment which will acts like the final resulting system [Zha05].

As illustrated in Figure 1, measurements are inherently *unsafe*, limited to producing timing results which are equal to or less than the actual WCET. When using measurements, a safety margin must be added to the obtained result, in the hope that the real worst case lies below the resulting WCET estimate. However, if too much margin is added, resources will be wasted, and if the added margin is too small, the resulting system will be potentially unsafe.

Assignment 1

Open the file `sensor.c` *in the* `sensor` *directory in an editor of your choice (e.g. emacs). The file consists of two C functions* `main()` *and* `foo()`*. The program can be seen as a typical embedded system program (or task), which reads two values from some sensors, (*`sensor1` *and* `sensor2`*), performs a calculation on the values (by calling* `foo()`*), and returns a result (*`res1 + res2`*).*

Please note that the read values are given as constants, (`READ1` *and* `READ2`*), in the code. In a real embedded system they might be set on a periodic basis using some memory-mapped I/0.*

Assume that you, to be sure to capture the WCET, have decided to run and measure the timing for all possible combinations of values for the two sensors. You perform each measurement by setting

`READ1` and `READ2` to some possible values, generating an executable, downloading the executable to your target embedded system (e.g. the Lego Mindstorms), and finally measuring a time using some dynamic timing analysis tool. After some practise, each measurement takes on average one minute to perform.

1.1 *How long time do you need to spend on measurement if your read sensor values both are 16 bits integers and you want to test all possible value combinations?*

After carefully inspecting the environment in which your system should operate, you determine that you can safely represent the read sensor values as an 8 bits unsigned char (`READ1`) and an 8 bits signed char (`READ2`) respectively.

1.2 *After this inspection, how long time do you now need to spend on measurement if you want to test all combination of input values?*

To further limit the time spent on measurements you decide to do a code inspection, trying to manually determine what input values that might give your WCET. You assume that the run that executes most instructions will also generate the WCET (something which might be true for simple embedded hardware, but not neccessarly for more advanced ones).

1.3 *Assuming that the above assumptions are true, what combination of read sensor values are likely to give the program WCET? Motivate your answer using the code given in `sensor.c`.*

Static WCET Analysis

Static timing analysis is an alternative method to derive WCET estimates. Such analysis does not perform any actual execution of the program. Instead it relies on models and analyses of the characteristics of the software and hardware involved. Given that the models are correct, a static analysis will derive a *safe* WCET estimate, i.e. a value greater than or equal to the actual WCET (see Figure 1).

Both the properties of the software and the hardware affect the execution time of a program. Consequently, static WCET analysis is usually divided into three phases: a *flow analysis* phase, deriving information on the possible execution paths through the program, a *low-level analysis* phase, which determine the timing behaviour of instructions in the program given the architectural features of the target hardware, and a final *calculation* phase, where the costliest execution path of the program under analysis is found using information from the first two phases.

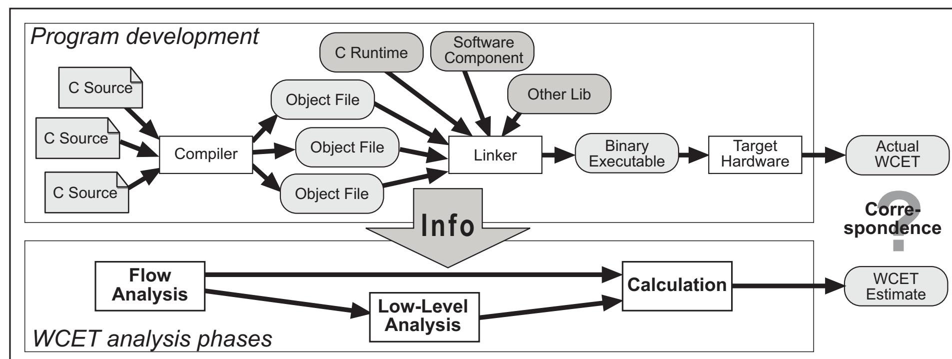


Figure 3: Embedded program development and static WCET analysis

Figure 3 gives an illustration of the normal program phases in embedded system development. Basically, the programmer is writing a number of C source files which are compiled to object files. The object files are linked together with library, runtime, OS and other precompiled object files, to form the final executable program [Lev00]. Information collected during these program development phases all provide valuable input to a static WCET analysis.

The flow analysis can be made on the source code, the intermediate code (inside the compiler) or

the machine code. The low-level analysis is by necessity made on the machine code (the binary, linked executable), since this is the program that will be run on the target hardware.

For this assignment we will generate and analyze code for the Renesas H8/3297 [Hit00], a RISC microcomputer running at 16MHz. It includes a H8/300 CPU core, common for all chips in the H8/300 series. The H8/3297 is the processing unit of the Lego Mindstorms [Leg05], an off-the-shelf kit of Lego bricks for building and controlling small prototype robots.

Assignment 2

To compile and link your example code you will be using the `make` *command. To get* `make` *to work correctly you need to open the file* `Makefile.mk` *in an editor and set the* `ROOTPATH` *variable to the location of your* `h8_env_and_code` *directory. Execute the shell command* `make`*. This should generate an output like:*

```
h8300-hms-gcc -g -O2 -fno-builtin -fomit-frame-pointer -Wall -o sensor.o -c sensor.c
h8300-hms-ld crt0.o sensor.o -fno-builtin -relax -e _start libmint.a -o sensor.coff-h8300
h8300-hms-objdump -D -S sensor.coff-h8300 >sensor.dis
```

The first line tells gcc to compile `sensor.c` *into the object file* `sensor.o`*. The second line links together the object files* `crt.o`*,* `sensor.o` *and the library file* `libmint.a` *into one executable coff file* `sensor.coff-h8300`*. The final line makes a disassembly of the coff file into a h8/300 assembler file* `sensor.dis`*. If you want to remove the generated files you can execute the* `make clean` *command.*

Open the `sensor.dis` *file in an editor. Investigate the assembler code to see how it corresponds to the orginal C code in* `sensor.c`*. You can see that each source code function get a corresponding code piece in the assembler code.*

2.1 *What are the start- and end-addresses for* `_main` *respectively* `_foo` *in the generated assembler code? What assembler code addresses correspond to the two function calls to* `foo()` *in the* `main()` *code?*

The h8/3297 has eight 16 bits registers, (numbered as `r0` *..* `r7`*). Most of the registers are used to hold arguments to instructions.*

2.2 *What register is used for holding a) the* `sensor1` *value? b) the* `sensor2` *value? c) the argument to function* `foo()`*? d) the return value from* `foo()`*?*

Most C code constructs can be directly mapped to some corresponding h8/300 instructions. However, the compiler can sometimes perform optimizations which change the machine code layout compared to the original C code. For example, the original `while(i <= max)` *condition gets replaced with two comparisions located at two different places in the code (at address* `130` *and* `148`*).*

2.3 *Consider the* `if(j < 5)` *condition in the* `foo()` *function. What assembler instructions corresponds to this condition? How has the compiler modified the assembler code comparison compared to the original comparison?*

Some C code instructions has no direct counterpart in the h8/300 instruction set. The `libmint.a` *library file contains some routines for handling such troublesome C code constructs.*

2.4 *Replace the sum* `res1 + res2` *with the product* `res1 * res2`*. Regenerate* `sensor.dis` *and open the file in an editor. Do the same thing using a division (*`/`*) and a subtraction (*`-`*). For what operators (*`-`*,*`*` *and* `/`*) does the resulting code size change significantly? How are these operators handled in the resulting h8/300 assembler code?*

C does not specify the size of an `int`*, but leaves it to the target architecture to define a size which it finds suitable. For example, on a standard PC, an* `int` *is usually defined as a 32 bit integer.*

2.5 *Use the* `typedef short type;` *row in* `sensor.c` *to change all the program variables to be of type* `char`*,* `short`*,* `int`*, and* `long int` *respectively. Rerun* `make` *and consider the generated assembler code for each case. What types will generate the same assembler code? Which type changes the resulting code size significantly? What is the size of an* `int` *for the h8/300 architecture?*

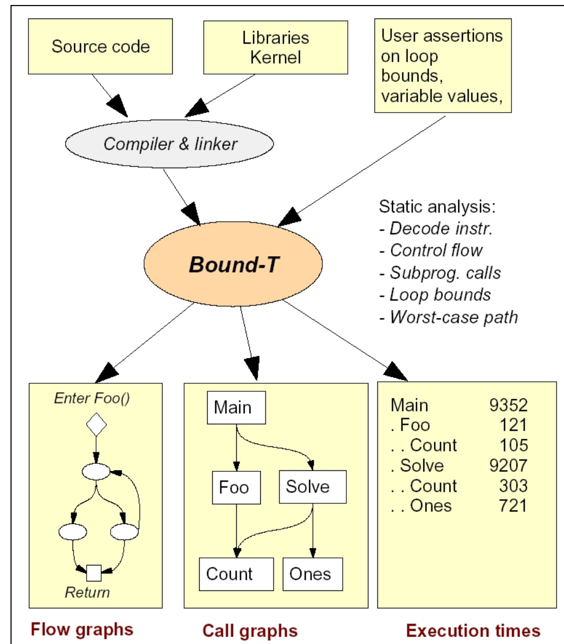


Figure 4: The Bound-T WCET analysis tool

The Bound-T WCET analysis tool

Bound-T is a commercially available WCET analysis tool from Tidorum Ltd [Tid06]. A shown in Figure 4, the tool works directly upon the machine code (i.e. the binary, linked executables) and does not look at source code. Still, Bound-T can usually interact with the user in source-code terms, such as subprogram identifiers, by using the debugging information from the executable file to translate between the source-code level and the machine-code level.

Bound-T follows the analysis principles depicted in Figure 3. The user specifies the executable file and the subprograms (e.g. C functions) to be analysed. Bound-T fetches the instructions from the executable file, builds the control-flow graphs and call-graphs, analyses the flow of execution (mainly by finding loop iteration bounds) and calculates an upper bound on the WCET. The call-graphs and control-flow graphs can be emitted in a graphical format.

Bound-T supports a variety of target platforms, including Intel 8051, ATMEL AVR, Renesas H8/300, SPARC V7 (ERC32), and ADSP-21020. A port to ARM7 is under way [Tid06]. Bound-T runs on normal PC host platforms under Linux or MS Windows.

For this assignment we will be using the H8/300 Bound-T version to analyse code generated for the Renesas H8/3297. You run Bound-T by typing `boundt_h8_300` in a shell with suitable command-line parameters as will be explained below. By typing `boundt_h8_300 -help` you get an informative listing of the arguments and options you could use when running the program. There is a large number of options; the main ones will be explained below in more detail. You can also refer to the Bound-T User Manual and to the Bound-T Application Note for the H8/300, both of which are available on the Bound-T web-site [Tid06].

The following options must be used for all analyses in this assigment, unless we give contrary instructions: `-lego -stack=external`. The first option (`-lego`) tells Bound-T that the program under analysis is assumed to run on the Lego Mindstorms processor (H8/3297). The second option (`-stack=external`) tells Bound-T that the stack memory, which is used for return addresses, parameters and local variables, will be located in the separate RAM chip. This off-chip or "external" RAM has a longer access time than the "internal" RAM on the processor chip itself.

The basic Bound-T tool is not interactive; you have to supply all the necessary inputs on the command line. If you want to change some argument or option, you must change and rerun the whole Bound-T analysis.

Assignment 3

Check that you can run Bound-T by executing the shell command

```
boundt_h8_300 -version
```

This should produce the output

```
Bound-T 2b5 for Renesas H8/300
```

Let Bound-T analyse the WCET of the original `sensor.coff-h8300` *program by executing the following shell command:*

```
boundt_h8_300 -lego -stack=external sensor.coff-h8300 _main
```

The resulting output should be some different printouts, ending with the following row:

```
Wcet:sensor.coff-h8300:sensor.c:_main:24-37:8044
```

This means that Bound-T has determined that 8044 processor cycles is an upper bound on the WCET of `_main`*. The other fields identify the executable file (*`sensor.coff-h8300`*), the source-code file in which this function lies (*`sensor.c`*) and the line-numbers in this source file (*`24-37`*).*

3.1 *What is this WCET bound in microseconds, assuming that the processor runs at 16 million cycles per second (16 MHz)?*

3.2 *The function* `foo()` *is called twice from* `main()`*. Use the generated Bound-T printout to calculate a WCET for the* total *time (in microseconds) spent in* `foo()`*.*

Basic blocks, control-flow graphs and call graphs

Bound-T performs a decoding of h8/300 assembler instructions from the binary coff file. By adding the `-trace decode` option to Bound-T you can see the decoded program. The decoded instructions are used to form *basic blocks*, *control-flow graphs* and *call graphs*.

A basic block is a sequence of instructions that can be entered only at the first instruction in the sequence and exited only at the last instruction in the sequence [Muc97]. A control-flow graph (cfg) is a representation of the code where nodes are basic blocks and edges between the nodes represent the possible flow of the program. Normally, each function in the code gets its own cfg.

A call-graph (cg) is a representation of how different functions in the program are calling each other. Each node corresponds to a function and the edges between nodes represent function calls.

Assignment 4

Once again, let Bound-T analyse the WCET of the original `sensor.coff-h8300` *program. Make sure that there is a directory named* `dotgraphs` *in the* `sensor` *directory (otherwise you should create it). Execute the following command (all in one row):*

```
boundt_h8_300 -lego -stack=external -dot_dir dotgraphs -draw address -draw all
sensor.coff-h8300 _main
```

The `-dot_dir dotgraphs` *option tells Bound-T to emit the generated cg and cfgs in a format called* `dot` *in the* `dotgraphs` *directory. The* `-draw address` *option tells Bound-T to include the start and end address of each basic block. The* `-draw all` *option specifies that each function call should get a cfg of its own.*

When investigating the `dotgraphs` *directory you should find four dot files:* `cg__main_001.dot`*,* `fg__main_002.dot`*,* `fg__foo_003.dot` *and* `fg__foo_004.dot`*. The first file contains the cg for* `_main`*, the second file is the cfg for* `_main`*, while the third and fourth files are the cfgs for* `_foo` *(the are two cfgs since* `_foo` *is called twice).*

To view the generated graphs we need to convert the dot files to postscript files using the `dot` *program. To convert the* `fg__foo_003.dot` *file to a postscript file execute the following command:*

```
dot -Tps fg__foo_003.dot -o fg__foo_003.ps
```

The generated file, `fg__foo_003.ps`*, can be opened using a postscript viewer, e.g.* `gsview32`*.*

4.1 *Consider the cfg in* `fg__foo_003.ps`*. a) How many basic blocks are included in the cfg? b) How is the WCET path represented graphically? c) Due to what condition, the* `while(i <= max)` *or the* `if(j > max)`*, will the loop be exited? d) For how many iterations will the loop be executed according to the WCET?*

If you do not want separate cfgs for each call of a function you should use `-draw total` *instead of the* `-draw all` *option.*

4.2 *Use the* `-draw total` *option to generate only one single cfg for* `_foo`*, (note that the* `fg__foo_003.dot` *file gets updated). How many iterations will the loop in* `_foo` *be executed in total (for both calls to* `_foo` *together) according to the WCET?*

To have a graphical illustration of the code could also be useful when you want to see what code optimizations the compiler has made.

4.3 *As an illustrating example, in* `sensor.c`*, replace the* `res1 + res2` *with* `res1 + res2 * 32`*, and regenerate the executable. Consider the cfg for* `_main`*. How has the cfg layout changed compared to the original cfg?*

Analysing the program flow

The purpose of a flow analysis is to derive bounds on the dynamic execution behavior of the program. This includes information on which functions get called, loop bounds, dependencies between conditionals, etc. Since the flow analysis does not know the execution path which corresponds to the longest execution time, the information must be a safe (over)approximation including *all* possible program executions.

Automatic flow analysis methods calculate this information with little or no manual intervention. For complex programs, it is hard (and in the general case impossible, due to the halting problem) to derive this information. To be feasible, automatic flow analysis methods therefore calculate approximations of the flow information, and allow additional information as *manual annotations*.

Bound-T has an automatic flow analysis for determining loop bounds. The analysis builds upon a technique called Presburger arithmetic [Pug91] and is able to calculate loop bounds for many type of loops, but not all. A general restriction is that the loop must have a counter variable for which Bound-T can find the initial value, the increment or decrement and the final value. For the H8/300 the counter variable must furthermore be a signed 8-bit or 16-bit integer.

Bound-T supports a large number of annotations for controlling the WCET analysis, although Bound-T calls them *assertions*. Assertions can be used to support or replace the automatic loop analysis. Assertions can directly specify the number of iterations of loops. Assertions can also specify the values a variable can have, at specific points in the program, from which Bound-T may be able to compute loop-bounds that depend on this variable. Other kinds of assertions let the user remove parts of the code which should not be included in the analysis, e.g. error-handling code.

Assertions are written as separate text files and not placed in the source-code files (this is why Bound-T calls them assertions rather than annotations). By modifying the assertions, various execution scenarios can be analysed, e.g. to find the program's response times under various input loads. You specify the assertion file for Bound-T by using the `-assert` option.

The Bound-T *assertion language* has its own syntax that is defined and explained in the Bound-T User Manual. However, we will give examples of the most common forms of assertion below.

Assignment 5

Enter the `sum` *directory and execute the* `make` *command. Study the files* `main.c`*,* `sum.c`*, and* `sum2.c`*, and note that the loop in the function* `sum()` *depends on the parameter* `len`*. Try to analyse the* `sum()` *function on its own using the command*

```
boundt_h8_300 -lego -stack=external main.coff-h8300 _sum
```

5.1 *You should get an error message and some other output. Can you explain what they mean, and why no WCET bound is found?*

The WCET of the `sum()` *function depends on how many times the loop repeats. You can tell Bound-T to assume a certain number of repeats by writing the following assertion into a text file, calling the file* `sum10.txt` *for example:*

```
subprogram "_sum"
   loop repeats 10 times; end loop;
end "_sum";
```

Don't worry about the indentation or the division into lines; the assertion language is "free format". Take care to get the keywords, semicolons, quotes (`"`*) and function names correct.*

Repeat the analysis of `sum()` *but add the option* `-assert sum10.txt` *to the command before the parameter* `main.coff-h8300`*.*

5.2 *What is the WCET of* `sum()` *under the assumption that the loop body is executed exactly 10 times?*

5.3 *Use different assertions to find out how long one repetition of the loop takes. Develop a formula for the WCET of* `sum()` *when* `len`> 0 *of the form* $WCET = A + B*$`len`*, where* A *and* B *are some constants. Find the values of* A *and* B*. (This formula does not give the right result when* `len`$= 0$ *because the compiler generates a check specifially for this case.)*

5.4 *Study the function* `sum2()` *in* `sum2.c`*. Note that there is a conditional statement in the loop. Which variables influence the execution time of* `sum2()`*? For a given value of the parameter* `len`*, which value of* `data[]` *gives the longest execution time?*

Study the function `main()` *in* `main.c`*. Note that it calls* `sum()` *twice and* `sum2()` *once. Compute an upper bound on the WCET of the* `main()` *function with the command*

```
boundt_h8_300 -lego -stack=external main.coff-h8300 _main
```

Bound-T computes WCET bounds for each call of `min()`*,* `sum()` *and* `sum2()` *and finally a WCET bound for the whole* `main()`*. Note that this analysis did not need any assertions because Bound-T was able to use the actual parameter values for the* `len` *parameter in the calls of* `sum()` *and* `sum2()` *to bound the loops in these functions.*

5.5 *Note that Bound-T analyses the* `sum()` *function twice, once for each call from* `main()`*, and computes different loop-bounds and WCET bounds for the two calls. Explain, by studying the code in* `main()` *and* `sum()`*, why the WCET bounds for the two calls to* `sum()` *are different.*

5.6 *The* `main()` *function calls* `sum2()` *and the loop in* `sum2()` *conditionally calls* `min()`*. How many executions of* `min()` *are included in the WCET bound for* `main()` *and why? Hint: in the call-graph, check the label on the edge to* `min()`*. How many times will* `min()` *actually be called when* `main()` *is executed?*

Tell Bound-T to assume only three calls of `min()` *by writing the following assertion into a text file, calling the file* `min3.txt` *for example:*

```
subprogram "_sum2"
   call to "_min" repeats 3 times; end call;
end "_sum2";
```

5.7 *Reanalyse* `main()` *with this assertion by adding the option* `-assert min3.txt` *to the Bound-T command. What has changed in the output?*

Analysing the hardware timing

The purpose of low-level analysis is to determine the timing behaviour of instructions given the architectural features of the target system. For modern processors it is especially important to

study the effects of various performance enhancing features, like caches and pipelines. If the hardware is too complex, it might be hard to exactly model the hardware behaviour, and safe (over)approximations might then be made. For example, if the analysis cannot determine if an instruction will be in the cache or not when referenced, it might be safe overapproximation to assume that the instruction will cause a cache miss. Fortunately, many embedded system processors, like the H8/3292, are rather simple, making them suitable for static WCET analysis.

Most instructions in the H8/3292 have a fixed execution time. However, the H8/3292 has several different ROM and RAM memory areas with different access time, making the execution time of an instruction depend on where its code and data are located.

The low-level analysis in Bound-T for the H8/300 assigns to each instruction an execution time (number of cycles) that has three components. Firstly, the fetch time depends on the address of the instruction (on-chip or off-chip); this is statically known. Secondly, there is a constant time that represents the internal processor cycles that the instruction needs. Thirdly, the operand-access time depends on the addresses of the operands in memory. Some instructions work on registers only, so no time is needed for memory accesses. Some instructions use statically known addresses for which Bound-T can use the right access time for that part of memory. However, many instructions read or write using dynamically computed memory addresses. The current version of Bound-T for the H8/300 divides such memory accesses into two kinds: *stack accesses* and *non-stack accesses*.

A stack access takes its address explicitly from the stack-pointer (register `r7`), perhaps with a constant offset. The gcc compiler uses such accesses for subprogram parameters and local variables. Since Bound-T usually analyses subprograms separately (instead of analysing the whole program at once), it does not try to compute the actual value of the stack pointer (`r7`). Instead, Bound-T assumes that the *whole* stack area is located either in the fast, on-chip, internal RAM, or in the slower, off-chip, external RAM. The user should tell Bound-T where the stack lies, with the command-line options `-stack=internal` or `-stack=external`. The default is `-stack=internal`. The instructions `push`, `pop`, `bsr`, `jsr` and `rts` also access the stack (but implicitly).

The actual location of the stack is controlled in the linker script (by placing the "stack segment" at some address) or at run-time by the program itself (by changing the value of the stack pointer).

A non-stack access is any other memory access with a dynamically computed address, that is, an address taken from some register other than the stack pointer `r7`. At present, Bound-T for the H8/300 does not try to find bounds on such memory accesses, but simply makes the worst-case assumption that they may point to the slowest kind of memory.

An access to the internal, on-chip memory takes a constant time that is set by the H8/300 processor design. In contrast, the processor design only sets a *lower bound* on the access time for an external, off-chip memory. This lower bound comes from the design of the processor's external memory bus. Some external memories may be too slow to work at this speed and an access may then need extra *wait-states*. Each wait-state adds one cycle to the access time for each accessed octet, during which the processor is idle. The number of wait-states may be different for reading and writing.

If your H8/300 system has such a slow external memory, you must tell Bound-T the number of wait-states with the options `-read_ws` and/or `-write_ws`. For example, to declare three wait-states for memory reads use `-read_ws=3`. The default is no wait-states for read or write.

Bound-T for the H8/300 has a number of other options that relate to the low-level analysis and the specific type of H8/300 processor and compiler that are used, but they are not important for this assignment.

Assignment 6

6.1 *Analyse the WCET of* `_main` *in the* `sum` *directory twice, first assuming that the stack is in external memory and then that it is in internal memory. Assume that the external memory needs no wait-states. Do not assert the number of calls of* `min()`*. What is the difference in the WCET bound?*

6.2 *Compare the WCET bound for the* `sum` *subprogram under three conditions: a) no wait-states (you did this analysis before), b) one wait-state for reading but none for writing, and c) one wait-state*

for writing but none for reading. Assume 10 iterations of the loop and internal stack. Explain the result (it should not be necessary to look at the machine code for this).

Advanced Bound-T WCET analysis features

The last assignment will let you investigate some of the more advanced features of Bound-T.

Enter the `insert` directory and execute the `make` command. Study the program implemented in the files `main.c`, `clist.c`, `clist2.c` and `clist3.c`. Read the descriptions in the header files (`.h` files). This program has three different implementations of a simple data structure: an ordered list of at most 100 characters.

The `clist` module implements a list that allows duplicated characters (more than one occurrence of a given character). The subprogram that inserts a new character in the list, `clist_insert()`, does not check for list overflow and allows duplicates.

The `clist2` module implements a similar list but without allowing duplicates. The insertion subprogram, `clist2_insert()`, checks for list overflow and for an attempt to insert a duplicated character. In both cases, the subprogram calls `clist2_error()` to handle the error. In this example, `clist2_error()` only copies the error message to the global variable `last_message`.

The `clist3` module implements a list with the same properties as `clist2`. The difference compared to `clist2` is in the design of the insertion subprogram, `clist3_insert()`. Here this subprogram has two loops, one for finding the proper place in the list and the other for shifting the rest of the list to make room for the new character. In `clist` and `clist2` the same loop handles both jobs.

Assignment 7

7.1 *Try to find a WCET bound for* `clist_insert()` *and* `clist2_insert()` *using Bound-T without any assertions. Why does it fail for* `clist_insert()`*? Why does it work for* `clist2_insert()`*?*

7.2 *Describe the scenario (the values of the variables and parameters) that leads to the WCET for* `clist2_insert()`*. You may find it helpful to look at the call-graph and the control-flow graph.*

For the analysis of `clist_insert()` *we can tell Bound-T about the limit on* `length` *by an assertion on the value of* `length`*. Write the following line into a text file:*

```
variable "clist.c|_length" <= 100;
```

This tells Bound-T that the value of `length` *is at most 100 at any time and at any point in the program.*

The prefix `_clist.c|` *in the variable name shows that we are talking about the* `length` *in the* `clist` *module, as distinct from the* `length` *variables in* `clist2` *and* `clist3` *which have the prefixed names* `clist2.c|_length` *and* `clist3.c|_length` *respectively.*

7.3 *Analyse* `clist_insert()` *with this assertion. Looking at the whole program, can you be sure that the assertion is valid, and why?*

7.4 *In fact, for any correct call of* `clist_insert()` *we can be sure that the* initial *value of* `length` *cannot be larger than 99. Explain why.*

The maximum value 99 for the initial value of `length` *when* `clist_insert()` *is called can be asserted as follows:*

```
subprogram "_clist_insert" (variable "clist.c|_length" <= 99)
end "_clist_insert";
```

When a variable-value assertion is written in this way, in parentheses after the subprogram name, it means that the assertion holds on entry to the subprogram, but may not hold later, after the subprogram changes the value of the variable.

7.5 *Analyse* `clist_insert()` *with the above assertion. You should get a smaller WCET bound than for the earlier assertion claiming that* `length <= 100` *always. Why? Which case is now excluded from the analysis? What is the maximum value that* `length` *has in the excluded case?*

For real programs one is often interested only in execution scenarios that are "nominal" or free from errors. For `clist2_insert()` *this means those cases where* `clist2_error()` *is* not *called, in other words where the list does not overflow and the new character is not a duplicate of some character already in the list. You can tell Bound-T to examine only cases where* `clist2_error()` *is not called by an assertion like this:*

```
all calls to "_clist2_error" repeat 0 times; end calls;
```

7.6 *Analyse* `clist2_insert()` *with this assertion and compare the new, smaller WCET bound to the bound without any assertions. Why is the difference not the same as the WCET bound for* `clist2_error()` *itself? Hint: compare the worst-case paths (bold paths) in the cfgs.*

7.7 *Is the above assertion* valid *for the whole program, considering what the main subprogram does?*

Try to analyse the WCET of `clist3_insert()` *without any assertions. This will fail because Bound-T cannot bound the second loop, the one that "shifts the rest of the list to make room". The reason for this failure is that the lower bound of this loop depends on the value of the local variable* `place`*, which is modified in the first loop. Although Bound-T knows that* `place` *must be less than 100, currently it does not understand that* `place` *must be* ≥ 0*. Tell Bound-T this with the assertion*

```
subprogram "_clist3_insert"
   variable "_clist3_insert|_place" >= 0;
end "_clist3_insert";
```

Note how the required prefix of the local variable `place` *differs from the prefix of the global variable* `length`*: for a local variable use the subprogram name; for a global variable use the source-file name. (If the variable-name is unique in the whole program, no prefix is needed.)*

7.8 *Use the above assertion to find a WCET bound for* `clist3_insert()`*. Look at the number of loop iterations in the worst-case path in the cfg. Are they overestimated, that is, larger than necesary? Hint: think about how the two loops in* `clist3_insert()` *are connected.*

You can get a better WCET bound for `clist3_insert()` *by determining the time per iteration of the two loops and then using assertions to define the scenario where the heavier loop iterates for the maximum number of times, while the lighter loop iterates for the corresponding smaller number of times. Always assume the nominal case, that is, no calls to* `clist3_error()`*.*

You will need to assert various numbers of repetitions for the first loop and the second loop. To specify which loop an assertion applies to you can use various properties of the loop, such as whether the loop uses (reads) or defines (writes) a specific variable, or whether the loop calls a specific subprogram. In `clist3_insert()`*, the variable* `place` *is defined only in the first loop, so the following assertion will set the number of repetitions of the first loop:*

```
subprogram "_clist3_insert"
   loop that defines "_clist3_insert|_place"
      repeats 10 times;
   end loop;
end "_clist3_insert";
```

To identify the second loop, say "loop that not defines ...".

You can either assert both loops at the same time, or assert one loop at a time and let Bound-T determine the loop-bound for the other loop (independently of your assertion). If you want Bound-T to determine the loop-bound for the second loop you must also include the assertion that `place` *is* ≥ 0*. All assertions for one run must be placed in the same* `subprogram .. end` *block with the* `loop` *assertion.*

7.9 *Find a better WCET bound for* `clist3_insert()` *as suggested above.*

7.10 *Find a WCET bound for the main program by asserting the best possible loop-bounds for each "insert" subprogram and by other suitable (and valid) assertions. Explain and motivate your assertions. Do you think that the WCET bound is still larger than the actual execution time? Why, or why not?*

Report

A proper report is to be handed in for the assignment. It should include at least the following:

- The cover page provided at the end of this document.
- Answers to the given assignments.
- When asked for, illustrate your answers by printing the (modified) C or assembler program.
- When asked for, provide informative listings of test runs in Bound-T.

The assignment must be handed in no later than Monday 2006-10-30.

Some general guidelines of how to make a report:

- Use a word processor, text editor or typewriter to type your solutions. Pictures and diagrams may be drawn by hand.
- Provide answers to given questions. Answers should be extracted from Bound-T runs.
- Staple your report thoroughly and with the pages in correct order.

References

[Gan06] Jack Ganssle. Really real-time systems. In *Proc. of the Embedded Systems Conference, Silicon Valley 2006 (ESCSV 2006)*, April 2006.

[Hit00] Hitachi. Hitachi Single-Chip Microcomputer H8/3297 series. *Hardware manual, 3rd edition*, 2000.

[Leg05] Lego Mindstorms homepage, 2005. `www.legomindstorms.com`.

[Lev00] J. Levine. *Linkers and Loaders*. Morgan Kaufmann, 2000. ISBN 1-55860-496-0.

[Muc97] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997. ISBN: 1-55860-320-4.

[Pug91] William Pugh. The Omega test: a Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Supercomputing*, pages 4–13, 1991.

[Ste04] David B. Stewart. Measuring execution time and real-time performance. In *Proc. of the Embedded Systems Conference, San Francisco 2004 (ESCSF 2004)*, March 2004.

[Tid06] Tidorum. Bound-T tool homepage, 2006. `www.tidorum.fi/bound-t/`.

[Zha05] Yina Zhang. Evaluation of methods for dynamic time analysis for CC systems AB. Master's thesis, Mälardalen University, Västerås, Sweden, August 2005.

Lab:	WCET Analysis Assignment
Course:	Real-Time Systems, GK
Period:	Autumn 2006

Lab Assistant

name:	Anders Pettersson
address:	Dept. of Computer Science and Electronics Mälardalen University SE-721 23, Västerås, Sweden
email:	anders.pettersson@mdh.se
phone:	+46 (0) 21 107011

I/We have solved these assignments independently, and each of us has actively participated in the development of all of the assignment solutions.

Student 1	Student 2
.... Signature	.... Signature
.... Name	.... Name
.... Personnummer	.... Personnummer
.... User name	.... User name