Hardware Implementation of Programming Languages for Real-Time

M. Ward* and N. C. Audsley
Real Time Systems Group
University of York
York, UK
michael.ward,neil.audsley@cs.york.ac.uk

Abstract

Real-Time Systems place large demands on the languages used to implement them. The current, processor based implementation methods do not allow accurate timing analysis due to the complexity of modern processors. FPGAs provide a means to implement a real-time system in way that allows accurate timing analysis.

Existing implementations of programming languages in hardware do not support the needs of a real-time system. This paper presents a hardware implementation of SPARK Ada that allows accurate timing analysis.

1 Introduction

Real-time systems require two things from their implementation environment: predictable timing and substantial language support [3]. Current implementation methods provide the language support, but the ever increasing complexity of modern processors are eroding the ability to analyse their timing. This is a major issue as a real-time system is as dependent on the timely arrival of information, as the accuracy of the information.

FPGAs provide a means to implement a system with a predictable timing model, but the language support is lacking. Current hardware programming languages are based on subsets of conventional languages, with extra hardware syntax added. This makes the porting of existing software problematic, with support for the needs of real-time systems substantially lacking. Better use of FPGAs could be made in real-time systems with a full implementation of an existing real-time language. This paper describes a method of compiling Ada to hardware, including the use of procedure calls and functions in expressions. The timing analysis of this for use in a real-time system is also discussed.

*This work was supported by EPSRC and BAE SYSTEMS

1.1 Real-Time Background

Real-Time Systems are those where the timing of an output is as important as the accuracy of its value. Often, these types of systems have to be proved correct in both the functional and time domains. Confidence in the timing of the software is achieved through static analysis[3]:

1. *Worst-Case Execution Time (WCET) Analysis* - Each basic block of the program is examined for its execution time. The longest path through the program is then found, and the worst case execution time for each process can be calculated.

2. *Scheduling Analysis* - Since a number of the processes in the system will run on a single processor, it has to be shown that they will all meet their deadlines despite interference from the other processes.

Calculating the WCET of a basic block can be done simply by adding together the execution times of the individual instructions. This gives an accurate WCET only when every single instruction takes the time given in the data book, and there are no other delays. This only occurs with legacy processors (e.g. Z80 or 68000) which have no architectural speed up features. Modern processors include speed-up features (e.g. pipelines, branch prediction units) that make WCET analysis difficult. Despite much work on this (e.g. [7][10]) high levels of pessimism still plague the WCET analysis process.

This paper addresses the issue of pessimism reduction or removal by utilising general purpose hardware rather than CPU's for the target implementation. This change of implementation medium raises issues with the Ada language definition, which are discussed later.

1.2 FPGA Background

Field Programmable Gate Arrays (FPGAs) were first released by Xilinx in 1985[21]. An FPGA provides a grid of logic, memory blocks and connections with which a circuit can be built up (figure 1). Each

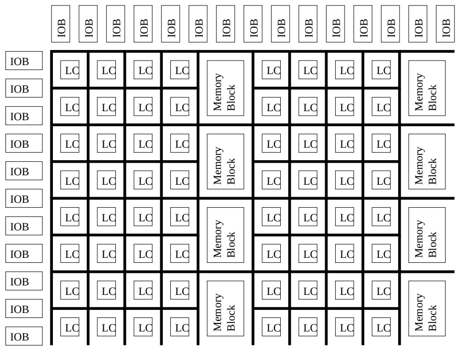

Figure 1: FPGA structure. This shows the top left corner of an FPGA with memory blocks as well as logic cells. The programmable routing is shown as thick lines.

logic cell can perform a number of logic functions, and also store some data in registers. The connections in the FPGA allow any logic cell to be connected to any other. These features allow complicated and large circuits to be programmed into the chip.

FPGAs can be reprogrammed in milliseconds, simply by changing their configuration bitstream. The bitstream is created by vendor tools from a high-level circuit description (e.g. VHDL).

1.3 Hardware Compilation Background

There are two forms of language available for hardware compilation:

- *Hardware Description Language (HDL)* - A typical HDL (e.g. VHDL [16], Pebble [11] and SL [8]) allows a hierarchical description of a circuit. Each block in the circuit can be described in terms of lower level blocks, and the connections between them, or a functional description of the block, possibly with other blocks used in this description.
- *High Level Language (HLL)* - A HLL allows the programmer to describe the functionality of the circuit using normal programming syntax. All translation to and arrangement of the final circuit is done by the compiler.

HDLs are unsuited to the implementation of software-based real-time systems. They have no communication constructs (though VHDL++ [5] proposes these), or timing facilities (e.g. delays and clocks) both of which are important to real-time systems [3]. HDLs provide coarse grained parallelism support through the circuit block structure.

There have been several implementations of HLLs for hardware compilation. Most of these are based on the C language (e.g. Handel-C [2], Transmogrifier C [6] and BRASS [4]), though Occam (Handel [13]) and Java (Xilinx Forge [21]) have been used. These languages are not suitable for real-time compilation as they do not provide the high-level communication and programming-in-the-large facilities required for real-time systems software.

Hardware compilation for real-time systems requires the support of a true real-time language by a hardware compiler. A real-time language has a number of requirements[3] including, syntactic support for the concurrency, communication and timing needs of a real time system. This paper considers the hardware compilation of Ada95, restricted by the Ravenscar (concurrent) and SPARK (sequential) subsets.

2 Ada95 Hardware Compiler

Figure 2 shows the structure of the SPARK Ada 95 hardware compiler being developed at York [17, 19, 18]. The front end parsing of the GNAT compiler[1] is accessed through the Ada Semantic Interface Specification (ASIS) [1]. The ASIS interface allows access to the syntactic and semantic information produced by an existing compiler during the compilation process, obviating the need to implement a parser and analyser specifically for this compiler. The hardware compiler takes four passes:

1. *Parsing and Analysis*
   The source program is compiled by the GNAT Ada compiler, which generates the semantic information used by the ASIS interface.
2. *Variable Analysis*
   The usage of variables is analysed to allow the implementation of the variables to fit with their use in expressions, for example a single ported memory cannot be used twice in a single cycle.
3. *Circuit Graph Generation*
   The circuit is created by template instantiation of the statements in the program. This is an intermediate representation containing all the cells and interconnections in the final circuit.
4. *Output Generation*
   The circuit graph is converted to a readable form, either textual, graphical, or a netlist which is converted to the configuration for the FPGA.

The method of compiling a high level language to hardware used in [13] or [20] produces a simple timing model - assignment takes one clock, and everything else is free. This makes analysing the timing of a program very easy (just count the assignments)

[1] The GNAT Ada compiler is a fully validated Ada95 compiler, the public domain version is available from `www.gnat.com`

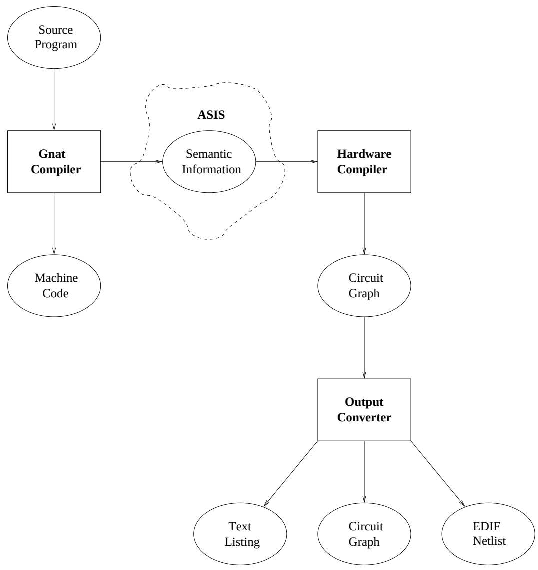

Figure 2: Hardware compiler structure

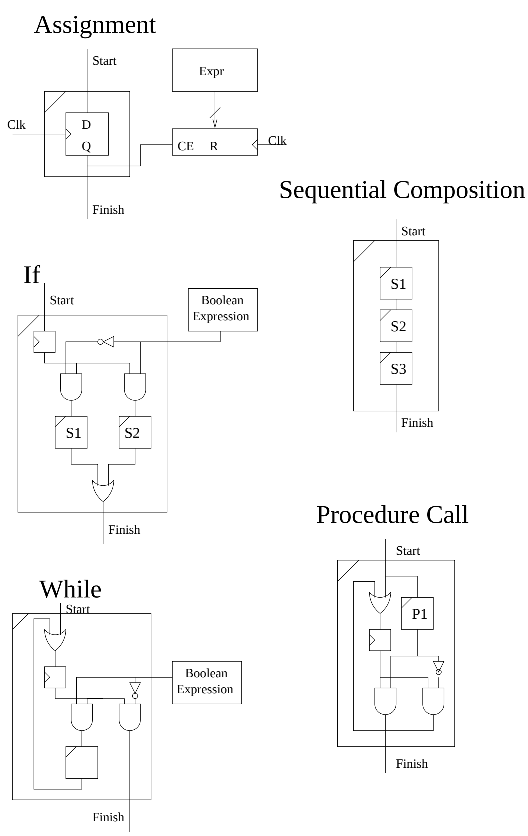

Figure 3: Circuit templates for statement compilation.

but does restrict the language features that can be implemented. Changing this timing model allows more features of the source language to be included. A more relaxed model would allow statements other than assignment to use clock cycles, even allowing one type of statement to have different timings dependent on the content of the statement. The following sections describe the compilation of statements and expressions taking advantage of the more relaxed timing model.

2.1 Statements

Statements are compiled into a set form, with the expression graphs from the statement being added to specialise the form. Each statements' circuit representation has the same interface, to allow for easy appending of statements. This interface consists of two signals, a 'start' signal, and a 'finish' signal. At present only assignments, while and for loops, if decisions and procedure calls have been implemented, though other statement forms can be easily included. Figure 3 shows the templates used for circuit generation, these are described below.

- *Assignment*
  A single flip-flop is required to keep the one-hot encoding of the state. The output of this flip-flop is connected to the data select for the appropriate value into the result register, and the write enable for that register. The output of the expression is connected to the same port of the selector. For multiple cycle expressions, additional flip-flops can be inserted.

- *If decision*
  The template introduces only one more flip-flop to the circuit, to allow for the evaluation of the expression. The true and else paths of the statement are compiled separately. An AND pair selects, according to the result of the expression, which path is to be followed. The final OR gate allows a finish signal from either path to propagate to the next statement.

- *Case decision*
  A case statement picks one of several alternatives depending on the value of an expression.

- *While Loop*
  Like the if statement, a single state flip-flop is added to allow calculation of the expression. An AND pair is then used to select whether the loop body or the following statement is started. In order to allow for the loop to be repeated, the finish signal of the loop body is ORed with the statement start signal before the expression calculation flip-flop.

- *For Loop*
  The for loop template is largely based on the

while loop template, since a for loop is a restriction of a while loop. An assignment is included before the loop to place the lower loop bound in the loop variable register. Another assignment, between the end of the loop body and the OR gate allows the loop variable to be incremented. The loop expression is a comparison of the variable against the upper loop bound.

2.2 Expressions

Expressions are compiled in a simple, recursive fashion. The result of an expression compilation is the output port of the last operation. Since the expression tree is easily retrieved from the ASIS interface, the conversion of an expression to a circuit is done by case analysis on the type of the root node of the expression. There are four types of node that the compiler can deal with:

1. *Constants* - A constant is constructed by tying the appropriate signals high or low.
2. *Simple Variables* - A simple variable is introduced into an expression by making a connection from the register that represents it to the expression tree.
3. *Array Variables* - Array variables are connected in similar way, but they also need the index expressions to be supplied to the memory. This means that the index expression needs to be compiled, then connected to the appropriate address port.
4. *Operators* - All operators are compiled to an operation cell which is connected to the output of the expression trees that form the arguments to the operator.

2.3 Sub-program Calls

One of the requirements dependable systems place on languages is the ability to cope with large and complex programs. Part of the support for this is the provision of sub-programs (procedures and functions) to allow the system to be broken down into manageable components. The SPARK subset does not allow recursion, simplifying the implementation of subprograms.

There are three main parts to the compilation of sub-programs:

1. *Compilation of Sub-program Bodies*
   A sub-program contains a (possibly empty) list of declarations, and a set of statements. How these are compiled needs to be considered.
2. *Treatment of Parameters*
   The parameters for a sub-program need to be passed in some way. The method used has to be decided upon.
3. *Compilation of Calls*
   The transfer of data between the calling and the called sub-program has to be controlled, as does the transfer of control between the two.

These are dealt with in the following sections.

2.3.1 Sub-program Bodies

A sub-program has a structure very similar to that of a main program; a list of declarations, and a set of statements. Since the language being implemented is non-recursive, only one call to a sub-program has to be allowed for. The needs of non-recursive sub-programs are the same as for the main program, so the same method can be used to generate the circuitry for the sub-programs as for the main program.

There is one additional statement, `return`, that needs to be compiled as a result of including sub-programs. This specifies the value to be returned from a function. By the syntax of SPARK, a function can have only one return statement, and this must be the last statement in the function. The result of a function needs to be stored for use in the expression which contains the function. This cannot be stored in the function so the storage must be done outside the function circuit. The compilation of the return statement therefore only requires the evaluation of the return expression.

2.3.2 Parameters

The method by which parameters are passed to the subprogram needs to be considered. Standard implementations use pass-by-value and pass-by-reference techniques.

- *Pass-by-value*
  This produces a conceptually simpler circuit, but for complex variables (e.g. arrays) this technique is expensive in either connections (to pass all the values simultaneously), or time (to pass values sequentially).
- *Pass-by-reference*
  This technique requires a link from each use of the parameter to each local variable that is used to provide that parameter. This saves significant amounts of logic over pass-by-value as no local storage is needed for the parameter. However, the static links back to calling code can be expensive, as the correct link needs to be selected.

The trade-offs between these two approaches suggests that single value parameters should be passed as values, and large complex structures (e.g. arrays) should be passed using a reference. The difficulty with this split is what to do with medium sized data structures (such as records). A pass by value could be very large (several fields of a record), compared to the use within the procedure (possibly only a single field), pass by reference would also require the ability to access any of the fields in the record. A possible solution to this is to pass only those fields that are required by the procedure (in effect, treating each field as a separate simple variable).

2.3.3 Calls

There are a number of problems that need to be solved in order to enable the compilation of sub-program calls:

1. Passing data to the sub-program.
2. Passing control to and from the sub-program.
3. Handling references to non-local variables.
4. Retrieving data from the sub-program.

Parameter Passing

There are two kinds of data that need to be passed to a sub-program; simple data values, and information needed for references (e.g. the range of an array slice), but these are also simple data values. These can be passed in the same manner as an assignment, evaluating the expression and clocking the value into the parameter register.

The expressions used in sub-program parameters are analysed differently according to the type of the subprogram. A procedure call is not an expression, so all the parameter expressions have not been analysed together. Consequently, the assignment of arguments to parameter's have to be done sequentially. A function call is part of an expression so all its parameter expressions will have been analysed together, allowing the parameter assignments to be done simultaneously.

Sub-program Control

The statements of a sub-program compile to produce a circuit with start and end signals, similar to the interface for the statement templates described above. The simplest method of inserting a sub-program call is to treat it as a statement and directly connect it as with any other statement. This does however have two problems when a sub-program is called from more than one place:

1. The start signal will be driven by more than one output possibly causing damage to the FPGA. This could be solved by inserting an OR gate between the calling and the sub-program circuits.
2. When the finish signal becomes active, execution will continue in all parts of the circuit that call the sub-program, causing multiple execution instances in a single, supposedly sequential circuit. Some method of restricting the dispersal of the finish signal needs to be employed.

The finish signal needs to be masked by so only the calling circuit is restarted. This can be done at the calling end by storing the fact that the sub-program call is in process, and using this to mask off the finish signal. As all sub-programs are non-recursive in SPARK, there can only be one place that a call to a sub-program was made, and so only one place that the execution will continue.

Handling References

As described above, a parameter that is passed by reference is represented in the sub-program as any other variable is, the only difference being that no storage exists in the sub-program. The output of these placeholders is a single set of connections that will carry all data accesses to the actual parameters, and a pair of control signals to indicate data read and write. These need to be connected back to the actual parameters in such a way that only those of the calling procedure are activated. This problem is similar to the return of control from sub-programs, so a similar solution can be taken.

At first it would seem that all the connections need to be masked by the output of the calling flip-flop described above. However, since the address and write data are only presented to the storage elements when either of the control signals (data read and data write) are active (the control signals mask the data in the selectors for each storage element), only these signals need to be masked by the calling state. Since only one of the calling states relating to a sub-program call can be active at once, one storage element cannot be accessed by two separate references at the same time.

Parameter Retrieval

Parameters passed by reference do not have to be returned to the calling process as the value changes are done to the variables directly. Simple return parameters of procedures can be copied back to the actuals as an assignment. The return parameter of an expression needs to be temporarily stored before it is used in the expression in case the same function is used elsewhere in the calculation of the expression.

3 Timing Analysis

This section describes the timing analysis of the produced circuits for real-time systems. Real-time systems depend on the timely arrival of results as much as the correctness of the result. In order to prove that a system meets its timing requirements the worst case timing of all operations needs to be found.

This is done through analysis of the code, or in this case, the circuit. For this, a timing model that allows the worst case timings to be found needs to be constructed. This section describes the timing model for these sequential part of the language. A comparison of the worst case timing characteristics of this implementation technique and a normal, processor based implementation has been done[17] which showed that the pessimism involved in the FPGA implementation was almost nil.

The compilation process described in the previous sections requires a timing model much more complex than the "one clock per assignment" model used by Handel or Wirth. The timing analysis of a the circuit can be automated by the compiler as it processes the original program.

The timing analysis depends on three main parts:

1. Expressions - clock cycles required to evaluate (including array access, operations and function calls).
2. Statements - combination of expression timings.
3. Sub-program calls - execution time of sub-program statements and parameter passing.

3.1 Expressions

The complexity of analysis of an expression depends on the complexity of the components of the expression. A long expression involving only simple components is easy to analyse, where as a short expression with complex components is much harder to analyse. A simple expression is one that doesn't need a clock cycle to evaluate, for example, a variable reference or a simple operator such as addition. Array accesses can count as a simple component, if only one or two access are made to the same array in a single cycle.

A complex expression component takes one or more clock-cycles to evaluate. Complex components include array accesses over the two that can be handled in a single clock-cycle, multiple cycle operators (e.g. shift-add multipliers), or function calls. When more than two array accesses are required for an expression, due to the limitations of the array implementation, some of the accesses have to be done prior to the evaluation of the expression and stored in a register for for use when the expression is evaluated. This adds one clock-cycle for each additional pair of accesses.

Multiple cycle operators need to have their processing registers loaded with the appropriate values, and then be left to execute. This means that they take one clock cycle for loading the registers, plus that needed to setup for the calculation of the argument expressions, and the appropriate number of cycles for the operator to execute. Function calls are timed as for sub-program calls as described below, but with two exceptions. Once all the argument expressions have been set up, loading the parameter registers occurs in a single clock cycle. After the function has executed, there are no parameters to be copied out, but the return value does need to be stored in a local register.

3.2 Statements

The timing of a statement is more dependent on the timing of its expressions than the timing of the basic template. Each statements worst case timing is a combination of the component timings as follows (where T is time in clock cycles, S is setup time in clock-cycles and X is number of loop iterations):

- *Assignment*
  `var := exp1`
  $T = S_{var} + S_{exp1} + 1$
- *Sequence*
  `st1; st2; ...  stn;`
  $T = T_{st1} + T_{st2} + ... + T_{stn}$
- *If*
  `IF exp1 THEN seq1 ELSE seq2 END IF;`
  $T = S_{exp1} + 1 + max(T_{seq1}, T_{seq2})$
- *Case*
  `CASE exp1 IS WHEN const1 => seq1; ...`
  `WHEN constn => seqn; END CASE`
  $T = S_{exp1} + 1 + max(T_{seq1}, ...T_{seq2})$
- *While*
  `WHILE exp1 LOOP seq1 END LOOP`
  $T = S_{exp1} + 1 + X * (S_{exp1} + T_{seq1})$
- *For*
  `FOR var IN exp1 ..  exp2 LOOP seq1 END LOOP`
  $T = S_{exp1} + S_{exp2} + 2 + X * (T_{seq1} + 2)$

3.3 Sub-program Calls

The timing of a sub-program call depends on three things:

1. *Parameter passing*
   Each in parameter of the sub-program that is passed by value needs to have its value calculated and passed. This requires the setup time of the expression, plus a clock cycle to load it into the parameter register. Any parameter passed by reference (in or out) needs to have the parameter information passed, which again is setup time plus one. This gives a total parameter passing time of the sum of expression setup times plus the number of parameters to be passed.
2. *Execution time*
   The execution time of the subprogram is the timing of the sequence of statements it contains.

3. *Parameter retrieval*
   Each out parameter that is passed by value (pass by reference values will have been put in the original variables automatically) needs to be copied back to the appropriate local variable. This will normally take one clock cycle per parameter, but it is possible that some expression evaluation will be needed (copying back to an array with a function derived index), in which case the setup for each expression will need to be added.

It should be noted that a function call passes all its parameters simultaneously, so only one clock cycle instead of one per parameter is required in addition to the setup time for the argument expressions. As a function cannot have any out parameters, there is no need for parameter retrieval, but there is a single cycle needed to retrieve the return value.

4 Examples

This section looks at a number of examples, each concentrating on a different result of the compilation process. Firstly, the output circuit is considered, showing how it can be traced back to the original code. Then the timing analysis of a procedure is examined, followed by a comparison of WCET between FPGA and normal processor based implementations. Finally, the space usage of the FPGA is considered.

4.1 Compiler Output

The use of a template instantiation compilation method allows for the output circuit to be traced back to the source program. Consider the example program given in figure 4, the compiled circuit graph this produces is shown in figure 5. Each node of the graph represents a cell in the final circuit. The cells are named with a unique identifier (e.g. I28), the type of the cell (e.g. COR - an OR gate), and a description of its purpose (e.g. Proc_start).

The four variables are replicated as registers (diamonds in figure 5). The variable a is cell I1, its OR cell being I2, and select I3. The other variables have been converted in a similar way (b as I4, I5 and I6, c as I7, I8 and I9 and d as I10).

The circuit starts with cell I29, which provides a single pulse to the start signal of the main procedure of the program. OR gate I28 is the first cell of the procedure, providing a means to combine start signals from all points that call the procedure. The first statement of the procedure is converted in 3 cells. The expression is calculated by adder I15, the constant 3 being generated by cell I14, c is retrieved from its register cell. Cell I13 holds the state for the state-machine, and drives the select and or for register a (I1).

```
procedure example is
    a, b, c, d : Integer;
begin
   a := 3 + c;
   if a < 5 then
      b := a;
   else
      b := c;
   end if;
   c := 5 + d;
end example ;
```

Figure 4: Example Ada Procedure

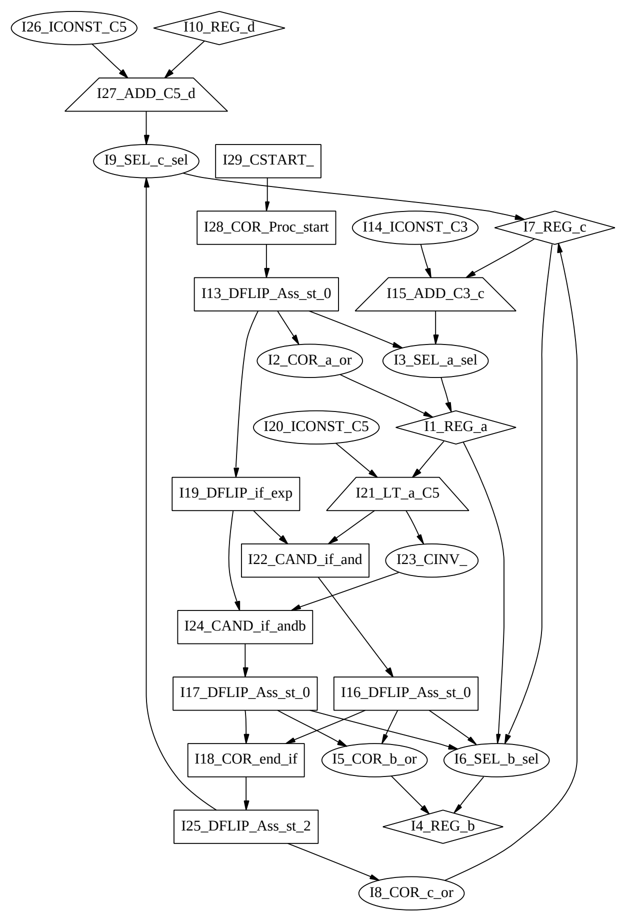

Figure 5: Circuit Graph. Each shape represents a cell of the final circuit. The control path is shown as rectangles, the diamonds are variable registers, and the trapeziums are operators. The cells are named using the following convention: a unique number, the type of the cell, and some information on the purpose of the cell.

```
procedure trisum (n : in integer;
                  sum : out integer) is
      tri : integer;
   begin
      sum := 0;
      for i in 1..n loop
         tri := 0;
         for j in 1..i loop
            tri := tri + j;
         end loop;
         sum := sum + tri;
      end loop;
   end trisum;
```

Figure 6: Example Ada procedure. This calculates the sum of the first n triangular numbers.

The IF statement is implemented in two parts, the expression evaluation and resultant decision, and joining the alternative paths back together. The expression is evaluated by cells I20 (constant 5) and I21 (less than) whilst I19 holds the state. The path is selected through I22 (AND), I23 (NOT) and I24 (AND). The finish signals of the two paths are combined to produce a single signal through OR gate I18. Both paths of the IF statement are a single assignment statement, neither assignment needing expression evaluation means that they are both represented by a single register (I16 for true path, I17 for false), which holds state as the appropriate values are assigned.

The final assignment is done in 3 cells. The state is held in cell I25, the expression being calculated by the add cell I27, using the register for d and the constant 5 (I26). The lack of optimisation in the compiler can be seen here as the constant 5 is generated in two separate cells, not just generated once and used twice.

4.2 Timing Analysis

Considering the code given in figure 6, the timing of the circuit can be analysed using the schema given earlier.

Starting with the body of the inner for loop, the schema is applied as follows. The body of the loop is a sequence of one statement. Therefor the loop body timing is that of the assignment statement. This takes a single clock cycle as the expression $tri + j$ is simple (as is the assigned variable tri), so no setup time is required. The inner loop statement takes $2 + N * 3$ cycles (where N is the number of iterations), as both loop bounds are simple and need no setup time.

The body of the outer for loop is a sequence of three statements. The first is a single cycle assignment, containing no operators or complex expressions in the assigned variable. The second statement is the inner for loop, with a cycle time of $2 + N * 3$. The final statement is another single cycle assignment. This gives a timing of $4+N*3$ for the body of the outer loop. The full timing for this loop is then $2 + M * (6 + N * 3)$, where M is the number of iterations of the outer loop.

The body of the procedure is a sequence of two statements. The first is a simple assignment taking one cycle, and the other is the outer for loop. This gives a total for the procedure of $3 + M * (6 + N * 3)$ cycles.

4.3 WCET Analysis

This section evaluates hardware compilation of the sequential subset of Ada in terms of the off-line timing analysis that has to be performed for real-time systems in order to prove their timing behaviour meets the specifications. This is done by comparing the WCET analysis of the FPGA and conventional implementations. For this evaluation we consider the compilation of the Ada procedure given in figure 6. It is assumed, for the purpose of this evaluation, that the value of `n` is never more than 20, enabling the maximum number of loop iterations to be known.

The comparison will be done across a number of timing analyses:

- *Naive*
  A naive worst case analysis assumes that the worst possible timings will always happen (e.g. every memory access is a cache miss, every loop executes the maximum number of times).

- *Model*
  A modelled worst case applies knowledge about the architecture to the timing analysis. This results in the effects of caches and pipelines being included in the final result.

- *Path*
  Semantic analysis of the program will result in improved knowledge of loop bounds and paths through the program. Together with modelling the architecture, semantic analysis produces a great improvement in the analysed WCET.

- *Expected*
  This is the average expected execution time of the longest path. This figure is produced by analysis taking a less pessimistic view of caching and pipeline effects.

The results of these analysis for both implementations are shown in table 1.

4.3.1 FPGA WCET Analysis

The state-machine style of implementation allows easy analysis of the WCET of the final circuit. This is due to the fact that the state-machine will always move onto the next state with each clock tick (unlike a processor that will take a variable amount of time to move

Analysis	Clock Cycles		Time (ms)	
	FPGA	CPU	FPGA	CPU
Naive	1323	341,666	0.066	0.683
Model	1323	77,766	0.066	0.155
Path	753	44,602	0.038	0.089
Expected	753	13,122	0.038	0.026

Table 1: The WCET timings of FPGA and conventional implementations of the example procedure given in figure 6. The time figures use a 20MHz clock for the FPGA and a 500MHz clock for the processor.

onto the next instruction). In contrast to a normal processor the WCET of the compiled circuit is almost exactly the same as its average case (the difference comes from certain paths in if statements taking longer than others). This ignores the pessimism introduced from loop bounds not being known.

The previous section derived the formula for the timing performance of the example procedure. Given that the value of n will never exceed 20, the naive WCET of the procedure is 1323. This is also the same as the modelled figure as there are no parts of the architecture that needs modelling. The semantics of the loop (the inner loop doesn't iterate 20 times each time, instead it iterates once, then twice...) can be, through path analysis [14][15], used to reduce the execution time to 753 cycles.

4.3.2 Conventional WCET Analysis

This can be compared to analysing a conventional implementation where the clock cycle count reached by adding up the times for each instruction will result in a figure that then has to be modified to take account of the architectural speed-up features of the target processor. Using the Intel 80486 architecture as an example, the inner loop takes 58 cycles, the outer loop 45 cycles, and the code outside the loop 46 cycles. A naive assumption to make is that no instructions are pre-fetched, and that every memory access is a cache miss. Assuming a cache miss cost of 30 cycles, a WCET of 341,666 cycles is reached.

A more realistic WCET can be arrived at through modelling the effects of the pipeline and cache. The size of the code means that there would be very few instruction cache misses, but data accesses cannot be modelled as accurately. Recent work at Princeton [9] and by Lundqvist[12] has shown that data cache accesses can be modelled; Lundqvist reported a hit rate of approximately 60% for most software. The number of jumps in the machine code negates the effects of the pipeline (as every jump requires the pipeline to be refilled) producing the result of 77,766 cycles. Including path analysis produces a further reduction in the WCET to 44,602. The measured WCET was calculated with the assumption that all the data values were in the cache, as were all the instructions, and the pipeline was kept full by (mostly) accurate branch prediction.

4.3.3 Discussion

Comparing the clock cycle figures in table 1 for the two different implementation methods we can see that the CPU implementation uses about 60 times more clock cycles for the tightest worst case bound. Most of this apparent speed up is lost in the differences in clock speed between the two implementation methods.

Modern processors run at speeds of several hundred megahertz, the compiled FPGA circuits run at 20 - 30 MHz, depending on expression size and overall circuit size, approximately 25 times slower than the processors. The right-hand two columns of table 1 show the timings of the two implementations when clock speed is taken into account. It can be seen that the speed up of the FPGA over a CPU implementation (for the path analysis) is around 2.4 times. This improvement is due to the much lower pessimism in the analysis of the FPGA implementation. The fact that the measured timings are in favour of the processor implementation shows that the worst case analysis of the FPGA implementation is much easier to do accurately than a conventional implementation.

Comparing the measured execution times it can be seen that the conventional implementation uses 18 times as many clock cycles as the FPGA implementation, but executes approximately a third faster. This may suggest that the FPGA implementation is less efficient, but when concurrency is considered, this inefficiency can be masked by the abilities of FPGA. Consider two concurrent copies of the procedure in figure 6, on a conventional processor, these would have to be executed one after the other, taking 0.052ms, whereas on the FPGA, they can be constructed to execute in parallel, still taking only 0.037ms for both processes.

It should be noted that the calculation of WCET for the circuit is done in terms of clock cycles, but the minimum size of the clock is dependent on the circuit. Large expressions cause the clock period to have to increase. Allowing the compiler to split up large expressions, making them multi-cycle will reduce the clock period, at the cost of making large expressions take two (or more) clock cycles to execute. The extra cycles these expressions will take to execute should be offset by the reduction in cycle time speeding up other parts of the circuit.

4.4 Size

The example code given in figure 6 when compiled to FPGA uses 100 logic blocks. Using a Xilinx Virtex-XCV100 [22] device gives a device utilisation of 8.3% (this device has 1200 logic blocks).

Whilst the compilation process is itself scalable in that any arbitrary conformant Ada program can be compiled to a circuit description (ie. logic gates), the ability to map that to an actual FPGA is clearly dependent upon the size of the FPGA and the tools performing the mapping. Whilst this involves the "place-and-route" tools provided by the FPGA manufacturer, clearly as the circuit increases in size, it becomes too big to fit onto the device. Normally, 100% use of the device is not possible. In practice, the size of Ada program that can be successfully mapped to the Virtex-XCV100 contains (typically) 100 lines of dense looping code, or around 500 lines of code of similar complexity to that in Figure 4. This device is one of the smallest devices in the Xilinx product range. Note that currently, optimisation of the generated circuit is not undertaken by the Ada hardware compiler. This would increase the number of lines of code that could be placed on the device.

5 Conclusions

This paper has shown that the current set of hardware languages are unsuitable for real-time-systems due to their lack of support for the needs of these systems. Instead, a hardware implementation of an existing language (SPARK Ada) has been developed, showing that it is feasible to fully implement an existing language. Methods of implementing expressions, statements and sub-programs have been developed that allow timing analysis to be done accurately, without restricting the use of the full language.

The performance of the compiler has been examined. It has been shown that the output of the compiler is fully traceable back to the source program. The timing analysis of the output of the compiler has been analysed from the source program, and has been shown to give large reductions in pessimism for WCET analysis over processor implementations. The resource usage of the output circuit has also been examined.

The main future extension to this work is the compilation of concurrency and communication in the Ada language. This will be able to be done in true parallel, giving much better utilisation of the available resources than is possible with processor based implementations of real-time systems.

References

[1] Ada Semantic Interface Specification (ASIS). Technical Report ISO/IEC 15291, IEC, 1999.

[2] M. Bowen. *Handel-C Language Reference Manual.* Embedded Solutions Limited, 2.1 edition, 1998.

[3] A. Burns and A. Wellings. *Real Time Systems and Programming Languages.* Addison Wesley, 2nd edition, 1996.

[4] T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The garp architecture and c compiler. *IEEE Computer*, 33(4):62–69, 2000.

[5] S. Franklid. Manual for VHDL++. Technical report, Dept. Information Technology, DTU Lyngby, 1999.

[6] D. Galloway. The Transmogrifier C Hardware Description Language and Compiler for FPGAs. In *FPGAs for Custom Computing Machines*, 1995.

[7] C. A. Healy, D. B. Whalley, and M. G. Harmon. Integrating the Timing Analysis of Pipelining and Instruction Caching. In *Proceedings RTSS'95*, pages 288–297. IEEE, 1995.

[8] S. Holmstrom. SL - A Structural Hardware Design Language. In *Proceedings of FPL 1999*, volume 1673, pages 371–376. LNCS, Springer-Verlag, 1999.

[9] Y.-T. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software.* Klewer Academic, 1999.

[10] S.-S. Lim, J. H. Han, J. Kim, and S. L. Min. A Worst Case Timing Analysis Technique for Multiple-Issue Machines. In *Proceedings of the 19th IEEE Symposium on Real-Time Systems*, pages 334–345, 1998.

[11] W. Luk and S. McKeever. Pebble: A Language for Parametrised and Reconfigurable Hardware Design. In *Proceedings of FPL '98*, volume 1482, pages 9–18. LNCS, Springer-Verlag, 1998.

[12] T. Lundqvist and P. Stenstrom. A method to improve the estimated worst-case performance of data caching. In *Proc. of RTCSA'99*, pages 255–262, December 1999.

[13] I. Page and W. Luk. Compiling Occam into FPGAs. In *FPGAs*, pages 271–283. Abingdon EE&CS Books, Abingdon, UK, 1991.

[14] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Paths. *Real-Time Systems*, 5(1):31–62, 1993.

[15] P. Puschner and C. Koza. Calculating the Maximum Execution Time of Real-Time Programs. *Real-Time Systems*, 1(2):159–176, 1989.

[16] VHDL Language Reference Manual. Technical Report 1076-1993, ANSI/IEEE, 1993.

[17] M. Ward and N. C. Audsley. Hardware Compilation of Sequential Ada. In *Proceedings of CASES 2001*, pages 99–107, 2001.

[18] M. Ward and N. C. Audsley. Hardware Ada - What needs to change? In *Proceedings of IRTAW 2002 - to appear*, 2002.

[19] M. Ward and N. C. Audsley. Language Issues of Compiling Ada to Hardware. In *Proceedings of Ada Europe 2002 - to appear*, 2002.

[20] N. Wirth. Hardware Compilation: Translating Programs into Circuits. *IEEE Computer*, 31(6):25–31, 1998.

[21] Xilinx product information : http://www.xilinx.com/products, 2001.

[22] Xilinx Corporation. *Xilinx Programmable Logic Data Book*, 2000.