TRACIS: Transformations on Ada for Circuit Synthesis

A Report on the methodology for a Silicon Compiler[1]

Sanjay V. Rajopadhye

P. A. Subrahmanyam

Department of Computer Science
University of Utah
Salt Lake City

UTEC-83-076

I Introduction

This report describes in detail, the ongoing design and implementation of a transformation system, for *compiling* specifications of integrated circuits into silicon. There are many levels in this process, and the area that we focus on produces target specifications of asynchronous and synchronous control units and the associated data paths. This target is compatible with the ASSASSIN system [1] which generates layouts from specifications of control units. The input to our system is an Ada program (restricted to a single Procedure Body) which specifies a certain computation. The Procedure Body is itself assumed to contain no package or task declarations or inatantiations and no Entry call statements. The result of the transformations performed by the system is a program consisting of the original specifications, with the target description appended to it.

Currently the system is in an experimental stage, and many of the intermediate decisions are specified interactively by the user. In spite of these limitations we feel that it is a valuable tool that we can use to study the exact mechanisms of the transformations as well as to understand how various syntactic and/or semantic analyses (e.g. data flow analysis) of the input can affect them.

---

[1]This research was sponsored by Defense Advanced Research Projects Agency, US Department of Defense under contract MDA903-81-C-0414

The rest of this report is organized as follows. In the following section we describe some of the motivation and background for the work. Section III describes the implementation of the system, which is further elaborated by means of a walkthrough of a detailed example (a part of the internet protocol circuit) in Section IV. Lastly, in Section V we describe some of our limitations, both in terms of equipment and computing resources that have so far acted as hurdles to our implementation efforts, and also indicate directions for future research.

II Motivation and Background

The overall transformation strategy underlying this effort of mapping high level specifications into silicon can be partitioned into three major phases:

1. Programs in a stylized subset of Ada are transformed into (the description of) a hardware configuration comprised of one or more interacting state machines. This description is in an appropriate hardware description language.

2. The description of a hardware configuration as obtained above is transformed into a symbolic, 2-dimensional circuit representation using Path Programmable Logic (PPLs) [6, 5]. The ASSASSIN system developed by Carter performs this task for control unit specification.

3. The PPL description of the circuit is then automatically translated into files detailing the fabrication masks required for processing the chip. Both Caltech Intermediate Form (CIF), CALMA, and ComputerVision (CV) format can be produced for such masks. These low level tools have been available for quite some time, and are now in a comparatively stable state.

As mentioned earlier, our effort concentrates on the first phase of the above transformation process. The declarative part of an Ada program may be viewed as defining an "environment" in which the program statement sequences define a (set of interacting) state machine(s). Our system transforms programs in an Ada subset to <state machine, environment> pairs described in a hardware description language. We have chosen to use (a stylized version of) Ada itself as a hardware description language. Obviously, this implies that not every Ada program can serve as a hardware description of a chip. But since the transformation system is designed to generate only specific Ada constructs in the target, the latter can be guarranteed to conform to the required Ada style.

TRACIS is built up in an Interlisp environment, using a language development tool,

POPART [9] (A Producer Of Parsers And Related Tools). POPART provides the user the ability to generate and manipulate parse trees for specified grammars. The primary function of POPART is as a parser generator (a la MINI or YACC). Its input is hence the BNF specification of a language, and using the production rules it generates a recursive descent parser, with limited backtracking capability (limited only to options on the RHS of a single production rule). In addition, it also produces a set of grammar dependent tools for the language. These tools include a structure editor, a pattern-matcher, and a rudimentary transformation system (actually built up on top of the editor). The primary data structure that all these tools operate on is the parse tree representation. Also, since TRACIS is written in Interlisp, the whole programming environment of the latter (viz history lists, programmer's assistant, dwimify, etc.) are available to the user.

The POPART transformation system provides commands such as **Find Nonterminal**, **Insert After**, **Replace**, **ReplaceAll** etc. In each of these, the commands are applied to the *current expression*, which is a subtree of the whole parse tree. Thus the editor commands (which are used to change the current expression) and the transformation commands, provide the basis for the transformations performed by TRACIS.

There are basically two levels of users of the POPART system. The *system builder* defines a grammar using POPART and with the associated tools builds a system on top of it, exactly as we are doing. The end user will have access to this system, and also to the POPART editor and transformation commands, as well as to commands defined by the overall system. However s/he will not be able to modify the grammar, and alter the other tools that have been defined.

III System Implementation: An Overview

TRACIS is similar to the code generation phase of a conventional compiler in the sense that it makes a pass over the parse tree and generates target code (which is in this case Ada itself, and is **appended** to the source rather than being a totally separate entity). A point of contrast however, is that in TRACIS the input is an unadorned parse tree. There is no symbol table information available. Hence, the first pass in TRACIS is perforce a combination of an information gathering and usage analysis pass. Strictly speaking, this phase is not performed in a single *pass* over the source; rather, for every declaration in the declarative **part** of the subprogram, usage analysis is performed *as needed*. The focus of

attention of the system therefore changes during this phase. The block diagram of the system is given in Fig III-0 and Fig III-1 indicates the functional modules invoked in the top level of system execution. In the latter POPART commands are in boldface, and comments (introduced for this document) are in parantheses.

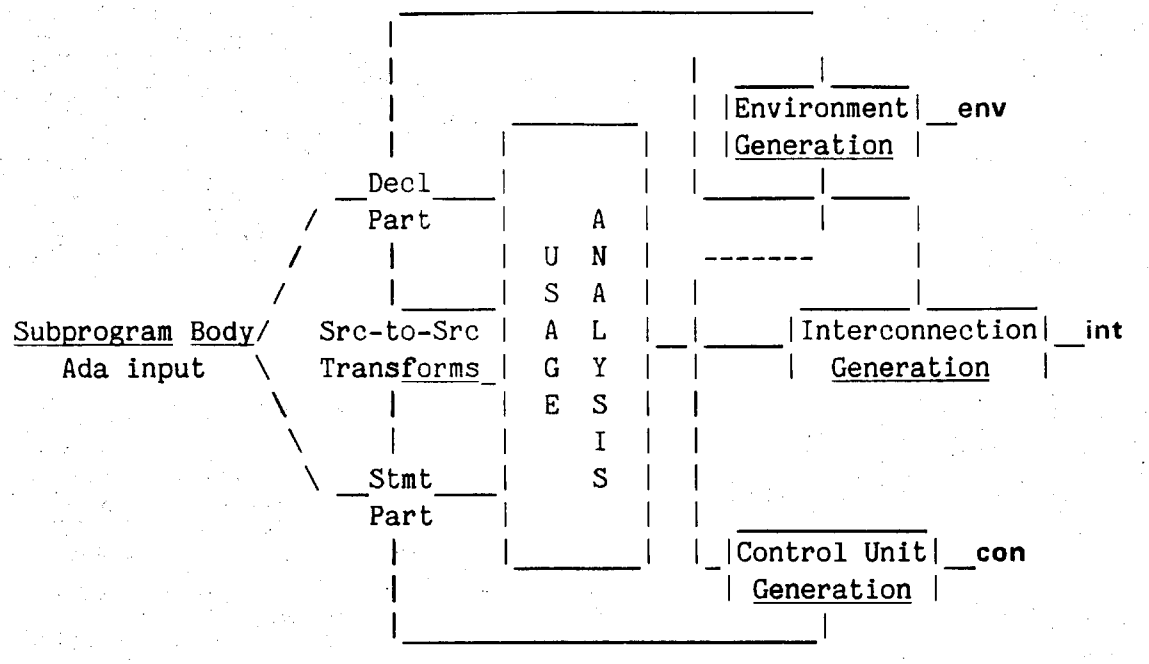

Figure III-1: Block Diagram of TRACIS

TRACIS assumes that the input program is a single Procedure Body. This does restrict the scope of the system, in that we do not handle packages and tasks, and the associated interface issues, but we felt that these extensions can be worked upon once we have the skeleton of the system automated. For procedures that involve more than one Subprogram Body, this is not a very serious restriction, and can be easily bypassed by surrounding the desired input by a Procedure Body. The system then generates the target description for the environment of the target state machine by a step-by-step traversal through the Declarative Part of the Subprogram Body. For every Object Declaration it encounters, TRACIS decides on an implementation in the target machine, based on certain system

```
Top;                    (Focus attention on the entire parse tree)
In SubprogramBody;      (Go in to Subprogram Body, followed by ...)
In DeclarativePart;     (... Declarative Part, and then onto  ...)
In DeclarativeItem;     (... Declarative Item)
ApplyToCurrent SelDeclType;
                        (SelDeclType analyzes the Declaration and
                         generates an implementation record for it)
while Next do ApplyToCurrent SelDeclType
                        (Do it for all Declarations)
GenAdditionals;         (Generate implementation records for any
                         additional elements as needed)
First;                  (Go back to the first declaration and ...)
GenerateEnv;            (... make a pass over all declarations,
                         select Object Declarations and create ...)
while Next do GenerateEnv
                        (... a list of elements in the environment)
ProduceTarget;          (Go to the top and insert a new Subprogram
                         Body contaning generic package package
                         instantiations for each of these elements
                         and also interconnections)
Top;
In SubprogramBody;
In SequenceOfStatements;
In Statement;
ApplyToCurrent SelectStatementType;
                        (Perform code generation for it)
while Next do SelectStatementType
                        (and the rest of the statement part)
```

Figure III-2: Top Level overview of TRACIS using POPART commands

defaults. Other Declarations are either Type Declarations or Subprogram Declarations[1]. Type Declarations cause the creation of a default implementation for objects that are declared to be of that type. Subprogram declarations cause the system to be invoked recursively on that particular subprogram. As an example the following is a part of TRACIS's execution transcript, and indicates how the object declaration for the identifier TOSTable is handled. Boldfaced font indicates user input. The numbers on the left are interlisp history-list numbers, and explanatory comments are in brackets

---

[1]Limitations of storage space on the TOPS20 Interlisp have forced us to use a subset of Ada. We feel that the features that we have deleted viz. Generics, Renamings, Representation Specifications and some similar esoteric features, do not adversely affect the scope of our research. The exact grammar that we use is available in Appendix II

```
69_Pretty...              [Pretty-print the current expression]
TOSTable : array (0..1023)of octet;...
70_ApplyToCurrent SelDeclType...
Should this array be represented as a RAM or a set of Registers RAM
Please Evaluate the value of the array range 1024
```

After this first pass, the system (ideally) performs a *usage analysis* on the input, and determines information required for the final implementation. This includes introduction of additional data elements, deletion of certain data elements, and/or changes in certain data elements in the target machine environment. Any of these changes may also involve some changes in the statement part of the subprogram body; e.g. if usage analysis yields the possibility of saving a loop counter by merging two loops, the source code must actually represent this merge, in addition to the removal of the additional loop counter. We call this analysis usage analysis rather than flow analysis, because it is a superset of the more conventional flow analysis. For example usage analysis could detect that the only arithmetic operations performed on a particular variable are multiplications and divisions by powers of two. This would result in an optimization that caused its implementation to be just a register without any arithmetic capability. Currently, this analysis is being performed through user interaction. This is an important part of the system, and we feel that by setting up an environment where the results of usage analysis can be utilized to change the source provides us with a skeletal system, where we can later experiment with specific usage analysis schemes and strategies.

Another important class of results of usage analysis is the actual data flow in the program. This can be mapped fairly directly into the physical interconnections between the various data objects in the target machine environment. This is also implemented as an interactive query of the user for the various connections among the elements in the environment.

TRACIS now focuses on the Sequence Of Statements part of the Subprogram Body. It makes a pass over these statements, generating *code* for the state machine that constitutes the control unit of the target. This is done in a manner similar to the code generation phase in conventional compilers. The system builds up a list of states and associated transitions, and also the input and output signals of the control unit. The scheme we use is described in detail in [7] and is similar to that proposed by Drenan [2].

At this point a large part of the information is in data structures similar to those found in conventional symbol tables. The state machine details are also in a separate data structure and the source code is unaffected (except for some source-to-source transformations during the usage analysis phase, as indicated earlier).

Now the system focuses on the whole input (i.e. a Subprogram Body) and, using the POPART editor commands creates another Subprogram Body (called Target) and appends it to the first. It then enters this and generates the Ada code for the entire target. This consists mainly of three parts -- the data elements in the environment part correspond to instantiations of generic packages from a predetermined library of standardly available target primitives, their interconnections are specified in the Sequence of Statements part as function calls on a dummy function *CONNECT* and the control unit is another package declaration. This package, called ControlUnit contains first a declaration of all the input and output signals that the Control Unit receives/sends. This is followed by individual task specifications for each state in the machine. The Entry Declarations in each of these task specifications define the names of the transitions from the state. Later in the ControlUnit package TRACIS generates the task bodies for each of these tasks, where it specifies the source, destination, input conditions, output conditions, and nature (i.e. fork, join or regular) of each of the transitions declared earlier. In the Sequence Of Statements part of the Target Subprogram the system generates statements which specify the interconnections among the data elements in the environment implemented as calls on the function **CONNECT** with the port names of various environment elements as arguments. This mechanism, although it is not equivalent to specifying a connection if Ada semantics are used, has been found to be an acceptable target, since it is compatible with the ASSASSIN system.

A. Implementation Status

The target currently generated by the system includes just the declarations that comprise the environment part, and the CONNECT statements in the procedure body. The implementation for generating the package for the control unit is currently under way. The target that is presented in Appendix I has been hand generated and is being used to guide the design of the state machine generation phase.

IV A Detailed Example: An early version of the RIP Chip

We now present an example that provides a clearer idea of our transformation methodology. The example is drawn from our familiarity with the RIP chip experiment [4]. Given below is a small procedure, equivalent to a part of the module that handles outbound datagrams in the Internet Protocol. In this procedure certain parameter values required by the module that fragments the datagrams are initialized.

```
procedure ReadInitParameters (NoOfTOS : octet)is
type octet is
range 0..255; ParamsReg : array (1..8)of octet;
TOSTable : array (0..1023)of octet;
TOSTableIndex : integer := 0;
procedure MemoryOutReq (ByteOfAddress : ByteOfAddressType
                        ; DoRead : boolean; Data : out OctetType)
begin
  for index in 1..8
  loop MemoryOutReq (DoRead => false,Data => ParamsReg (index))
       ;
    end loop
  ;
  for index in 1..ParamsReg (8)
  loop
    for index2 in 1..ParamsReg (7)
    loop MemoryOutReq (ByteOfAddress => DontCare, DoRead => false,
                       Data => TOSTable (TOSTableIndex))
         ;
         TOSTableIndex := TOSTableIndex+1;
      end loop
    ;
    end loop
  ;
end ReadInitParameters;
```

The (hand generated) target corresponding to the above input specification is available in Appendix I.

V Limitations and Further Work

The target for the immediate future is to complete the implementation of the state machine generation phase. The strategy for this part has been developed, and many of the algorithms are similar to code-generation algorithms used in conventional compilers. More ambitious goals are to extend the system to handle a larger part of the Ada language and

incorporate the flow/usage analysis function into the system. Another possible avenue of research is to explore the possibility of using different (i.e. non Ada) languages, and if necessary develop the required languages, based more on an understanding of the exact needs.

The format in which the production rules of the grammar are specified to POPART is such that there is exactly one rule per nonterminal. Theoretically this does not lead to any loss in power, and is an inconvenience at best since the right hand side of a production may contain any number of alternatives. Because of this restriction, any given grammar usually has to be *massaged* so that a correct parser can be produced. One of the results of the massaging is that an excessive number of dummy nonterminals are introduced in the grammar. Since the POPART editor also utilizes the same grammar, the actual editing becomes slightly cumbersome, as do the editing functions of TRACIS.

Because of the above problem, one of the principal hurdles faced in the implementation of TRACIS is an address space limitation on the Dec-20. The data structures used by POPART are so space-consuming that for any reasonable size program the underlying Interlisp system runs out of storage. One of the primary objectives during the initial design of TRACIS has been to make the target that it generates compatible with the ASSASSIN system. Since the input language to ASSASSIN is not Ada we decided to add *syntactic sugaring* to it to couch it in terms of Ada syntax. However, since the current version of TRACIS operates on a single grammar (a recent development in POPART now permits the handling of multiple grammars, at the expense of even more storage) these syntactic constructs must be retained in the grammar. The grammar used is thus perforce much larger than that needed for the source specification. An area of future work is to explore the possibility of paring it down to a manageable size.

In addition to the storage limitations there are certain deficiencies with the whole POPART environment. Although the parse tree is a useful representation of the input (and is indespensable for certain manipulations such as pretty printing, etc.) we now believe that an Abstract Syntax Tree (AST) is more appropriate. Although the POPART reference manual states that the parse tree can be compacted and can be thought of as an AST, many of the transformation and editor functions operate strictly on the parse tree, and the advantage gained by the compaction is minimal. As has been already mentioned, POPART

also has no provision for symbol tables. The reason for this is that a particular symbol table (and associated symbol table managing scheme) implies a specific set of scoping rules, which makes the system language dependent. However, a provision for associating external (i.e. non-POPART) data structures with specific nodes in the parse tree (or the AST) would be a valuable asset. Such an arrangement would also leave the issues of scoping and variable bindings independent of the grammar. The *parser* generator part of such a system (i.e. one based on AST's and providing hooks for external data structures) is currently being developed by Tinker [8].

A means of specifying performance constraints is also required, and an exploration and development of a theoretical foundation for these aspects is also needed. Some of the theoretical areas, (eg CCS [3]) are being explored currently and could yield valuable results.

Appendix I Target for the RIP example

```
procedure Target (MemAck, Req, MstClr, Clock    : in boolean;
                  MemReq, MemDoRead, Ack        : out boolean;
                  MemOctet                : in integer range 0..225) is

package ParamReg0 is new Register(Size => 8);
package ParamReg1 is new Register(Size => 8);
package ParamReg2 is new Register(Size => 8);
package ParamReg3 is new Register(Size => 8);
package ParamReg4 is new Register(Size => 8);
package ParamReg5 is new Register(Size => 8);
package ParamReg6 is new Register(Size => 8);
package ParamReg7 is new Register(Size => 8);

package EntryCtr is new IncClrRegister(Size => 8);
package ByteCtr is new IncClrRegister(Size => 8);
package Counter3 is new IncClrRegister(Size => 3);
package TOSAddrReg is new IncClrRegister(Size => 8);

package TOSRAM is new RAM(DataSize => 8, AddrSize => 16);

package LoadControls is new Decoder(InSize => 3);

package EdoneComp is new EqComp;
package BdoneComp is new EqComp;
package SevenCheck is new ConstEqComp;

package ControlUnit is

  MstClr         :insignal       := unknown;
  MemAck         :insignal       := unknown;
  Req            :insignal       := unknown;
  Four           :insignal       := unknown;
  Seven          :insignal       := unknown;
  EntryDone      :insignal       := unknown;
  TOSDone        :insignal       := unknown;

  MemReq         :latched        := ff;
  MemDoRead      :latched        := ff;
  Ack            :latched        := ff;
  TOSRdWr        :latched        := ff;
  ClrTOSAddrReg  :gated          := ff;
  IncTOSAddrReg  :gated          := unknown;
  TOSRAMSel      :gated          := ff;
  LdDecode       :gated          := ff;
```

```
  ClrCtr3       :gated          := ff;
  IncCtr3       :gated          := ff;
  ClrEntryCtr   :gated          := ff;
  IncEntryCtr   :gated          := ff;
  ClrByteCtr    :gated          := ff;
  IncByteCtr    :gated          := ff;


task StateSTRT is
  entry MoveS01;
end StateSTRT;

task StateS01 is
  entry MoveS02;
  entry MoveSTRT;
end StateS01;

task StateS02 is
  entry MoveS03;
  entry MoveS04;
  entry MoveSTRT;
end StateS02;

task StateS03 is
  entry MoveS01;
  entry MoveSTRT;
end StateS03;

task StateS04 is
  entry MoveS05;
  entry MoveSTRT;
end StateS04;

task StateS05 is
  entry MoveS06;
  entry MoveSTRT;
end StateS05;

task StateS06 is
  entry MoveS07;
  entry MoveSTRT;
end StateS06;

task StateS07 is
  entry MoveS08;
  entry MoveS05;
```

```
  entry MoveSTRT;
end StateS07;

task StateS08 is
  entry MoveS09;
  entry MoveSTRT;
end StateS08;

task StateS09 is
  entry MoveS10;
  entry MoveSTRT;
end StateS09;

task StateS10 is
  entry MoveS11;
  entry MoveSTRT;
end StateS10;

task StateS11 is
  entry MoveS12;
  entry MoveS09;
  entry MoveSTRT;
end StateS11;

task StateS12 is
  entry MoveS09;
  entry MoveSTRT;
end StateS12;

end ControlUnit;

package body Controlunit is

  task body StateSTRT is
    begin
      accept MoveS01() do
        move(on(clk and Req), to(StateS01));
      end MoveS01;
      hold(ClrCtr3);
    end StateSTRT;

  task body StateS01 is
    begin
      select
        accept MoveS02() do
          move(on(clk and MemAck), to(StateS02));
```

```
      end MoveS02;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    set(MemReq);
    set(MemDoRead);
    hold(IncCtr3);
  end StateS01;

task body StateS02 is
  begin
    select
      accept MoveS03() do
        move(on(clk and not(Req) and not(Four) and not(MemAck)),
               to(StateS03));
      end MoveS03;
      or
      accept MoveS04() do
        move(on(clk and not(Req) and Four and not(MemAck)),
               to(StateS04));
      end MoveS04;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    reset(MemReq);
    set(Ack);
  end StateS02;

task body StateS03 is
  begin
    select
      accept MoveS01() do
        move(on(clk and Req), to(StateS01));
      end MoveS01;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
  end StateS03;

task body StateS04 is
```

```
  begin
    select
      accept MoveS05() do
        move(on(clk), to(StateS05));
      end MoveS05;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    hold(ClrCtr3);
  end StateS04;

task body StateS05 is
  begin
    select
      accept MoveS06() do
        move(on(clk and MemAck), to(StateS06));
      end MoveS06;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    set(MemReq);
  end StateS05;

task body StateS06 is
  begin
    select
      accept MoveS07() do
        move(on(clk), to(StateS07));
      end MoveS07;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    hold(LdDecode);
    reset(MemReq);
  end StateS06;

task body StateS07 is
  begin
    select
      accept MoveS08() do
```

```
        move(on(clk and not(MemAck) and Seven), to(StateS08));
      end MoveS08;
      or
      accept MoveS05() do
        move(on(clk and not(MemAck) and not(Seven)), to(StateS05));
      end MoveS05;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    hold(IncCtr3);
  end StateS07;

task body StateS08 is
  begin
    select
      accept MoveS09() do
        move(on(clk), to(StateS09));
      end MoveS09;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    hold(ClrEntryCtr);
    hold(ClrByteCtr);
    hold(ClrTOSAddrReg);
  end StateS08;

task body StateS09 is
  begin
    select
      accept MoveS10() do
        move(on(clk and MemAck), to(StateS10));
      end MoveS10;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    set(MemReq);
  end StateS09;

task body StateS10 is
  begin
```

```
    select
      accept MoveS11() do
        move(on(clk), to(StateS11));
      end MoveS11;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    hold(TOSRAMSel);
    hold(not(TOSRdWr));
  end StateS10;

task body StateS11 is
  begin
    select
      accept MoveS12() do
        move(on(clk and not(MemAck) and EntryDone), to(StateS12));
      end MoveS12;
      or
      accept MoveS09() do
        move(on(clk and not(MemAck) and not(EntryDone)), to(StateS09));
      end MoveS09;
      or
      accept MoveSTRT() do
        move(on(clk and MstClr), to(StateSTRT));
      end MoveSTRT;
    end select;
    hold(IncByteCtr);
    hold(IncTOSAddrReg);
  end StateS11;

task body StateS12 is
  begin
    select
      accept MoveS09() do
        move(on(clk and not(TOSDone)), to(StateS09));
      end MoveS09;
      or
      accept MoveSTRT() do
        move(on(clk and (TOSDone or MstClr)), to(StateSTRT));
      end MoveSTRT;
    end select;
    hold(ClrByteCtr);
    hold(IncEntryCtr);
  end StateS12;
```

```
end ControlUnit;

begin

Connect(ParamReg6, EdoneComp.In1);
Connect(EntryCtr, EdoneComp.In2);
Connect(EdoneComp.EqOut, ControlUnit.EntryDone);

Connect(ParamReg7, BdoneComp.In1);
Connect(ByteCtr, BdoneComp.In2);
Connect(BdoneComp.EqOut, ControlUnit.TOSDone);

Connect(Counter3, SevenCheck.VarInp);
Connect(7, SevenCheck.ConstInp);
Connect(SevenCheck.EqOut, ControlUnit.Seven);

Connect(Req, ControlUnit.Req);
Connect(MemAck, ControlUnit.MemAck);
Connect(MstClr, ControlUnit.MstClr);
Connect(Clock, ControlUnit.clk);

Connect(ControlUnit.MemReq, MemReq);
Connect(ControlUnit.MemDoRead, MemDoRead);
Connect(ControlUnit.Ack, Ack);
Connect(ControlUnit.TOSRdWr, TOSRAM.RdWrt);
Connect(ControlUnit.TOSRAMSel, TOSRAM.Select);
Connect(ControlUnit.ClrTODAddrReg, TOSAddrReg.Clr);
Connect(ControlUnit.IncTODAddrReg, TOSAddrReg.Inc);
Connect(ControlUnit.ClrCtr3, Counter3.Clr);
Connect(ControlUnit.IncCtr3, Counter3.Inc);
Connect(ControlUnit.ClrEntryCtr, EntryCtr.Clr);
Connect(ControlUnit.IncEntryCtr, EntryCtr.Inc);
Connect(ControlUnit.ClrByteCtr, ByteCtr.Clr);
Connect(ControlUnit.IncByteCtr, ByteCtr.Inc);
Connect(ControlUnit.LdDecode, LoadControls.Enable);

Connect(Counter3, LoadControls.Inp);
Connect(LoadControls.Out0, ParamReg0.Load);
Connect(LoadControls.Out1, ParamReg1.Load);
Connect(LoadControls.Out2, ParamReg2.Load);
Connect(LoadControls.Out3, ParamReg3.Load);
Connect(LoadControls.Out4, ParamReg4.Load);
Connect(LoadControls.Out5, ParamReg5.Load);
Connect(LoadControls.Out6, ParamReg6.Load);
Connect(LoadControls.Out7, ParamReg7.Load);
```

```
Connect(MemOctet, ParamReg0, ParamReg1, ParamReg2, ParamReg3, ParamReg4,
        ParamReg5, ParamReg6, ParamReg7, TOSRAM.Data);
Connect(TOSAddrReg, TOSRAM.Addr);


end Target;
...
```

Appendix II Grammar for the System

```
NameBody := IndexedComponentBody | SelectedComponentBody |
      FunctionCallBody || ;

IndexedComponentBody :=  '( Expression ^ ', ') ;


SelectedComponentBody := '. AIdentifierAllOperator ;

  AIdentifierAllOperator := Identifier | AllConstant | OperatorSymbol ||;

  AllConstant := 'all ;

  ALogicalRelation := IAndRelation |
                      IOrRelation ;

  IAndRelation := 'and Relation ^ 'and ;
  IOrRelation := 'or Relation ^ 'or ;

  AInfixOperation := ARelationalOperator | ARangeOperator |
      ASubtypeOperator || ;
  ARelationalOperator := RelationalOperator SimpleExpression ^
      RelationalOperator# ;
  ARangeOperator := { NotConstant } 'in Range ;
  ASubtypeOperator := { NotConstant } 'in SubtypeIndication ;
  NotConstant := 'not ;

  NestedExpression := '( Expression ') ;

RelationalOperator := '= | '/= | '< | '<= | '> | '>= ;
AddingOperator := '+ | '- | '& ;
UnaryOperator := '+ | '- | 'not ;
MultiplyingOperator := '* | '/ | 'mod | 'rem ;
ExponentiatingOperator := '** ;

InValue := 'in Name ;

OutValue := 'out Name ;

ConstantConstant := 'constant ;

TypeDeclaration := 'type Identifier 'is TypeDefinition '; ;

TypeDefinition := EnumerationTypeDefinition | IntegerTypeDefinition |
            ArrayTypeDefinition | RecordTypeDefinition || ;
```

```
Constraint := RangeConstraint |
              IndexConstraint || ;

RangeConstraint := 'range Range ;

EnumerationTypeDefinition := '( EnumerationLiteral ^ ', ') ;

IntegerTypeDefinition := RangeConstraint;

ArrayTypeDefinition := 'array (Index ^ ', | IndexConstraint)
                          'of SubtypeIndication ;

IndexConstraint := '( DiscreteRange ^ ', ') ;

RecordTypeDefinition := 'record ComponentList RecordEnd ;

  RecordEnd := 'end 'record ;

  NullConstant := 'null ;

CompoundStatement := IfStatement |
                     CaseStatement |
                     LoopStatement |
                     Block |
                     AcceptStatement |
                     SelectStatement || ;

Label := '<< Identifier '>> ;

NullStatement := 'null '; ;

IfStatement := 'if Condition 'then SequenceOfStatements
               { IElseIfStatement + } { 'else SequenceOfStatements# }
                  FinishIf ;

  IElseIfStatement := 'elsif Condition 'then SequenceOfStatements ;

  FinishIf := 'end 'if '; ;

CaseStatement := 'case Expression 'is { IChoicesOfStatements + }
                                      CaseEnd '; ;

  CaseEnd := 'end 'case ;

  OthersConstant := 'others;
```

```
  IChoicesOfStatements := 'when Choice ^ '| '=> SequenceOfStatements;

BasicLoop := 'loop SequenceOfStatements LoopEnd ;

 LoopEnd := 'end 'loop;

IterationClause := 'for LoopParameter 'in { ReverseConstant }
                                        DiscreteRange |
                   'while Condition;

  ReverseConstant := 'reverse;

ExitStatement := 'exit { Name } { 'when Condition } '; ;

ReturnStatement := 'return { Expression } '; ;

GotoStatement := 'goto Name'; ;

SubprogramDeclaration := 'procedure Identifier { FormalPart } |
      'function Designator { FormalPart } 'return SubtypeIndication ;

FormalPart := '( ParameterDeclaration ^ '; ') ;

Mode :=  OutConstant | InOutConstant | InConstant || ;
  InConstant := { 'in } ;
  OutConstant := 'out ;
  InOutConstant := 'in 'out ;

SubprogramBody := SubprogramDeclaration
      'is DeclarativePart
      'begin SequenceOfStatements
      'end { Designator } '; ;

FunctionCallBody := ActualParameterPart | '( ') ;
ActualParameterPart := '( ParameterAssociation ^ ', ') ;

  PackageDeclaration := PackageSpecification '; |
                        GenericPackageInstantiation || ;

PackageSpecification := 'package Identifier 'is { DeclarativeItem + }
                        'end { Identifier };
GenericPackageInstantiation := 'package Identifier 'is 'new Name
                     { '( ParameterAssociation ^ ', ') } '; ;
```

```
PackageBody := 'package 'body Identifier 'is DeclarativePart
                    { 'begin SequenceOfStatements }
                    'end { Identifier } '; ;

TaskDeclaration := 'task { TypeConstant } Identifier
                    { TaskPart } '; ;

     TaskPart := 'is { EntryDeclaration + }
                   'end { Identifier } ;

  TypeConstant := 'type;

TaskBody := 'task 'body Identifier 'is { DeclarativePart }
             'begin SequenceOfStatements
             'end { Identifier# } '; ;

EntryDeclaration := 'entry Identifier { '( DiscreteRange ') }
                    { FormalPart } '; ;

AcceptStatement := 'accept Name { FormalPart }
                    { 'do SequenceOfStatements
                      'end { Identifier } } '; ;

DelayStatement := 'delay SimpleExpression '; ;

SelectStatement := SelectiveWait | ConditionalEntryCall |
                    TimedEntryCall || ;

SelectiveWait := 'select IWhenSelect ^ 'or { 'else SequenceOfStatements }
                    SelectEnd '; ;

  SelectEnd := 'end 'select ;

  IWhenSelect := { 'when Condition '=> } SelectAlternative;

SelectAlternative := TerminateConstant | (AcceptStatement |
                              DelayStatement)
                                    { SequenceOfStatements } ;
  TerminateConstant := 'terminate '; ;

ConditionalEntryCall := 'select EntryCall { SequenceOfStatements }
                         'else SequenceOfStatements#
                        SelectEnd '; ;

TimedEntryCall := 'select EntryCall { SequenceOfStatements }
```

```
                    'or  DelayStatement { SequenceOfStatements# }
                  SelectEnd '; ;

AbortStatement := 'abort Name ^ ', '; ;

Expression := Relation { ALogicalRelation } ||;

Relation := SimpleExpression { AInfixOperation } ||;

SimpleExpression := IUnaryTerm { OAddingOperatorTerm } ||;
  IUnaryTerm := { UnaryOperator } Term || ;
  OAddingOperatorTerm := IAddingOperatorTerm + ;
  IAddingOperatorTerm := AddingOperator Term;

Term := Factor { OMultiplyingOperatorFactor } ||;
  OMultiplyingOperatorFactor := IMultiplyingOperatorFactor + ;
  IMultiplyingOperatorFactor := MultiplyingOperator Factor;

Factor := Primary { OExponentialPrimary } || ;
  OExponentialPrimary := '** Primary;

Primary := NestedExpression | InValue | OutValue | Name | Literal ||;

Range := SimpleExpression '.. SimpleExpression# ;

DeclarativeItem := ObjectDeclaration | TypeDeclaration |
      SubprogramDeclaration | PackageDeclaration | TaskDeclaration ;

ObjectDeclaration := IdentifierList ': { ConstantConstant }
      ASubtypeOrArrayType { ':= Expression } '; ;

  ASubtypeOrArrayType := SubtypeIndication | ArrayTypeDefinition || ;

NumberDeclaration := IdentifierList ': ConstantConstant ':=
      LiteralExpression '; ;

IdentifierList := Identifier ^ ', ;

Index := TypeMark 'range '<> ;

ComponentDeclaration := IdentifierList ': ASubtypeOrArrayType
                        { ':= Expression } '; ;

ProgramComponent := SubprogramBody | PackageBody | TaskBody |
                    PackageDeclaration | TaskDeclaration || ;
```

```
SequenceOfStatements := Statement +  ;
Statement := { Label } ASimpleOrCompound  ||;
  ASimpleOrCompound := SimpleStatement | CompoundStatement || ;

SimpleStatement := NullStatement |
                   AssignmentStatement |
                   ExitStatement |
                   ReturnStatement |
                   GotoStatement |
                   ProcedureCall |
                   EntryCall |
                   DelayStatement |
                   AbortStatement || ;

AssignmentStatement := Name ':= Expression '; ;

Condition := BooleanExpression ||;

LoopStatement := { LoopIdentifier ': } { IterationClause } BasicLoop
                 { LoopIdentifier# } '; ;

LoopParameter := Identifier;

Block := { BlockIdentifier ': }
             { 'declare DeclarativePart }
             'begin SequenceOfStatements
         'end { BlockIdentifier# } '; ;

Designator := Identifier | OperatorSymbol || ;

ParameterDeclaration := IdentifierList ': { Mode } SubtypeIndication
                            { ':= Expression } ;


EntryCall := Name { ActualParameterPart } '; ;


Name := (Identifier | OperatorSymbol) { NameBody + };


Literal := NumericLiteral | EnumerationLiteral
             | CharacterString | NullConstant || ;

EnumerationLiteral := Identifier | CharacterLiteral || ;

DiscreteRange := Range | TypeMark { RangeConstraint } ;
```

```
  Choice := OthersConstant | DiscreteRange | SimpleExpression ||;

  OperatorSymbol := CharacterString ;

  ProcedureCall := Name { ActualParameterPart } '; ;

  ParameterAssociation := { FormalParameter '=> } ActualParameter;

  FormalParameter := Identifier || ;

  ActualParameter := Expression || ;

  SubtypeIndication := TypeMark { Constraint } ;

  TypeMark := Name ;

  StaticSimpleExpression := SimpleExpression ||;
  LiteralExpression := Expression ||;
  BooleanExpression := Expression ||;
  ComponentSubtypeIndication := SubtypeIndication ||;
  LoopIdentifier := Identifier ||;
  BlockIdentifier := Identifier ||;


CharacterLiteral := LEXEME |> CharLitFilter;

CharacterString := LEXEME |> CharStringFilter;

NumericLiteral      := RealLit | IntegerLit ||;
 RealLit            := Real { 'E Exponent };
 IntegerLit         := Integer { 'E PositiveExponent };

   NegativeExponent := '- Integer ;
   PositiveExponent := { '+ } Integer || ;
   Exponent         := PositiveExponent | NegativeExponent || ;

   Real             := Integer '. Integer#;
   Integer          := LEXEME |> IntegerFilter ;

Identifier := LEXEME |> IDFilter;

  ComponentList := NullConstant '; | { ComponentDeclaration + } ;

  DeclarativePart := { DeclarativeItem + }
                     { ProgramComponent + } ;
```

```
  Compilation := { SubprogramBody + }
...
```

References

1. T. M. Carter. ASSASSIN: An Assembly, Specification and Analysis System for Speed-Independent Control-Unit Design in Integrated Circuits Using PPL. Master Th., Department of Computer Science, University of Utah,June 1982.

2. Drenan, L.A. On Transforming Ada to Silicon. Master Th., University of Utah, Department of Computer Science,August 1982.

3. Milner, Robin. *A Calculus of Communicating Systems.* Springer-Verlag, Berlin, Heidelberg, 1980.

4. Organick, E.I. et al. Transformation of Ada Programs into Silicon. Fourth SemiAnnual Technical Report. UTEC-83-075, University of Utah, November, 1983.

5. Patil, S. S. and Welch, T. "A Programmable Logic Approach for VLSI". *IEEE Trans. on Computers C-28* (Sept 1979), 594-601.

6. K. F. Smith; T. M. Carter; and C. E. Hunt. "Structured Logic Design of Integrated Circuits Using the Stored Logic Array". *IEEE Transactions on Electron Devices ED-29*, 4 (April 1982), 765-776.

7. Subrahmanyam, P.A. and Rajopadhye, S. Automated Design of VLSI Architectures: Some Preliminary Explorations. UTEC # 82-067, University of Utah, October (Revised), 1982.

8. Tinker, Pete. AGGAST - A Generator Generator for Abstract Syntax Trees. Class Report.

9. Wile, Dave. POPART: A Producer of Parsers and Related Tools, System Builder's Manual. Unpublished, USC/ISI.