HardwareC - A Language for Hardware Design Version 2.0

David Ku
Giovanni De Micheli

Technical Report No. CSL-TR-90-419

April 1990

This research was sponsored by NSF, under grant No. MIP-87-19546, by AT&T and DEC jointly with NSF, under a PYI Award program, and by a fellowship provided by Philips/Signetics.

HardwareC – A Language for Hardware Design
Version 2.0

David Ku *and* Giovanni De Micheli

Technical Report: CSL-TR-90-419

April 1990

Computer Systems Laboratory
Departments of Elecrical Engineering and Computer Science
Stanford University
Stanford, CA 94305-4055

Abstract

High-level synthesis is the transformation from a behavioral level specification of hardware, through a series of optimizations and translations, to an implementation in terms of logic gates and registers. The success of a high-level synthesis system is heavily dependent on how effectively the high-level language captures the *ideas* of the designer in a simple and understandable way. Furthermore, as system-level issues such as communication protocols and design partitioning dominate the design process, the ability to specify ***constraints*** on the timing requirements and resource utilization of a design is necessary to ensure that the design can integrate with the rest of the system. In this paper, a hardware description language called *HardwareC* is presented. *HardwareC* supports both declarative and procedural semantics, it has a C-like syntax, and it is extended with notion of concurrent processes, message passing, timing constraints via tagging, resource constraints, explicit instantiation of models, and template models. The language is used as the input to *the Hercules/Hebe* High-level Synthesis System.

Key Words and Phrases: Hardware Description Language, Constraint Specification, High-level synthesis

Contents

List of Figures

HardwareC – A Language for Hardware Design
Version 2.0

1 Introduction

High-level synthesis is the transformation from a behavioral level specification of hardware to a register transfer level description, which may then be mapped to a VLSI implementation. The success of high-level synthesis systems is heavily dependent on how effectively the input language captures the ideas of the designer in a simple and understandable way. This paper describes ***HardwareC,*** a hardware description language that is used as the input to ***the Hercules/Hebe High-level synthesis system*** [2, 3].

Hardware ***behavior*** generally consists of two sets of specifications – a description of the ***functionality*** and a set of ***design constraints.*** The functionality describes the tasks to be performed by the hardware design, such as logic operations or control flow constructs. The design constraints specify the timing and resource requirements that are imposed on the final hardware implementation. As circuits become more complex, the effective integration of a design with the rest of the system often dominate the design process. This increases the importance of modeling hardware partitions and interfacing.

HardwareC is a language that uniformly incorporates ***both*** functionality and design constraints. A HardwareC description is synthesized and optimized by the Hercules and Hebe system, where tradeoffs are made in producing an implementation satisfying the timing and resource constraints that the user has imposed on the design. The resulting implementation is in terms of an interconnection of logic and registers described in a format ***called the Structural Logic Intermediate Format.*** We refer the interested reader to [3] for an overview of the Hercules system.

1.1 Motivations

Many hardware description languages have been proposed and used in both academia and in industry. Most hardware description languages are oriented towards hardware simulation and documentation. Notable examples include ISP, ADLIB/SABLE, ESIM, and VHSIC Hardware Description Language. Conversely, HardwareC has been designed to be a hardware description language for synthesis. The following criteria are very important in the design of a synthesis-oriented HDL, and they have been addressed in HardwareC:

1. ***Supports fill spectrum of design styles.***

   The language should support readily the varying spectrum of design styles of the designer, ranging from a pure behavioral description that is independent of the structural implementation, to a mixture of behavior and structure, to a pure structural description of the interconnection and instantiation of hardware modules.

2. ***Supports external interfacing.***

   An important characteristic of hardware designs is the need to communicate with other modules in the system subject to a particular handshaking protocol. The ability of a design to interface with, and synchronize to, external signals and events is of critical concern in the specification of complex circuit designs, and should be fully supported by the language.

3. ***Supports timing and resource constraints.***

   As the complexity of hardware increases, there is a corresponding increase in the number and strictness of the constraints that are imposed on the hardware design. Timing constraints that specify bounds on the time separation between operations, and constraints that specify the resource utilization and binding of operations to resources, are important elements in the language.

4. ***Simple to learn and use.***

   The language is a tool that the designer uses to capture and transform abstract ideas into complete designs. The tool must therefore be simple to learn and easy to use. Specifically, the language should contain the most basic constructs that are needed to describe a design, without being burdened by specialized constructs that are specific to a particular class of designs.

Some synthesis systems and hardware description languages support only a specific design style, either pure structure or pure behavior. We believe a more effective approach to design is to use a flexible underlying language that captures the *essence* of the design from the designer, whether that essence be behavioral, structural, or a mixture of both. This criterion is crucial in a design environment since very often the designer has a particular structure in mind when designing hardware. This partial structure should be captured by the language, and reflected in the results of synthesis.

1.2 Features of HardwareC

HardwareC attempts to satisfy the requirements stated above. As its name suggests, it is based on the syntax of the C programming language [1]. The language has its own hardware semantics, and differs from the C programming language in many respects. In particular, numerous enhancements are made to increase the expressive power of the language, as well as to facilitate hardware description. The major features of HardwareC are:

- ***Both procedural and declarative semantics*** – Designs can be described in HardwareC either as a sequence of operations and/or as a structural interconnection of components.

- ***Processes and interprocess communication*** – HardwareC models hardware as a set of concurrent ***processes*** that interact with each other through either port passing or message passing mechanisms. This permits the modeling of concurrency at the functional level.

- *Unbound and bound calls* – An *unbound* call invokes the functionality corresponding to a called model. In addition, HardwareC supports *bound* call that allows the designer to constrain the implementation by explicitly specifying the particular *instance* of the called model used to implement the call, i.e. bind the call to an instance.

- *Template models* – HardwareC allows a single description to be used for a group of similar behaviors *through the use* of *template models.* A template model is a model that takes in addition to its formal parameters one or more integer parameters, e.g. an adder template that describes all adders of any given size.

- *Varying degrees of parallelism* – For imperative semantic models, HardwareC offers the designer the ability *to* adjust *the* degree of parallelism *in* a given design *through the* use of *sequential* (*[*]), *data-parallel* ({ }), and *parallel* (< >) groupings of operations.

- *Constraint specification* – Timing constraints are supported through tagging of statements, where lower and upper bounds are imposed on the time separation between the tags. Resource constraints limit the number and binding of operations to resources in the final implementation.

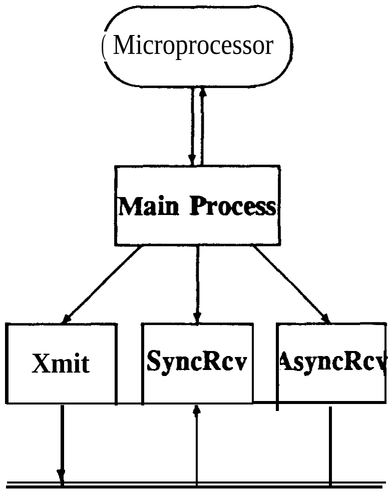
2 Modeling Hardware Behavior

We model hardware behavior as a collection of concurrent modules. Each module represents a functionality that can be described either in terms of a structural interconnection of components (i.e. *declarative* semantic), or as a set of operations sequenced in time that performs a particular *algorithm* (i.e. *imperative* or procedural semantic). The modules communicate and synchronize with each other explicitly through the use of parameters, which can be either *ports* on which values are placed and retrieved, or *channels* on which messages are sent and received. Ports and channels are described in later sections.

The concept of a design consisting of concurrent modules is powerful for both hardware and software systems. In both domains, it allows the designer to *specify the coarse-grain parallelism* between interacting modules at a *high* level, and *isolate the points of communication and synchronization* between the modules in an explicit manner. To illustrate the concept, consider the Intel 8251 UART shown in Figure 1. The UART is modeled as four concurrently executing modules. *The main* module accepts commands from the micro-processor and coordinates the execution of the other modules. The *transmitter* writes data out on the serial interface, and *the* two receiver modules, *synchronous-receiver* and *asynchronous_receiver*, read data from the serial interface.

2.1 Types of Models

There are four fundamental design abstractions in HardwareC, corresponding *to block, process, procedure,* and *function models.* At the topmost level, a design is given in terms of a *block.* A block describes the structural relationship and physical connectivity among the various components of a design. It has a *declarative* semantic, and consists of an interconnection of logic and instances of other blocks and processes. For example, a block model that describes a ripple chain of adders is shown in Figure 2.



Serial Interface

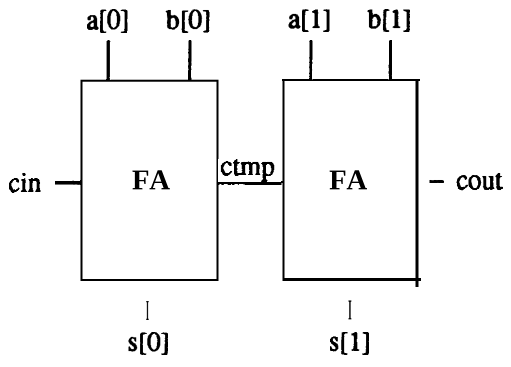
Figure 1: Hardware model for Intel 8251 UART

In contrast, process, procedure, and function models have an imperative semantic. An imperative semantic model describes an algorithm that consists of a set of operations sequenced in time. The algorithm consists of data-flow operations such as logic expressions and assignments to shared variables, and control flow constructs such as sequencing, branching, and iteration. Although process, procedure, and function all describe an encapsulation of operations in the form of an algorithm, a ***process*** model differs from the others in that it executes the algorithm repeatedly. That is, a process automatically restarts execution upon the completion of its last operation. An example of a process that finds the greatest common divisor is shown in Figure 3. On the other hand, the computation described by a ***procedure*** or ***function*** model execute only when the model is ***called.***

The model of hardware behavior as a collection of concurrent and interacting processes is natural for hardware description since hardware modules continuously operate on a time varying set of inputs. Therefore, blocks describe the structural relationships among the processes, which in turn describe algorithms consisting of a hierarchy of procedures and functions. We now describe the syntax of model definitions.

- ***Block model*** – A block contains either logic operations, or calls to other blocks (therefore supporting hierarchy) and processes. The formal syntax of block definition is:

```
block name ( parameter list )
    parameter declarations
    block-body
```



Structural interconnection
(a)

```
block ripple(a,b,s,cin,cout)
  in port a[2],b[2],cin;
  out port s[2],cout;
<
  boolean ctmp;
  FA(a[0],b[0],s[0],cin,ctmp);
  FA(a[1],b[1],s[1],ctmp,cout);
>
```

Block model description
(b)

Figure 2: Example of using blocks to describe a ripple chain of adders.

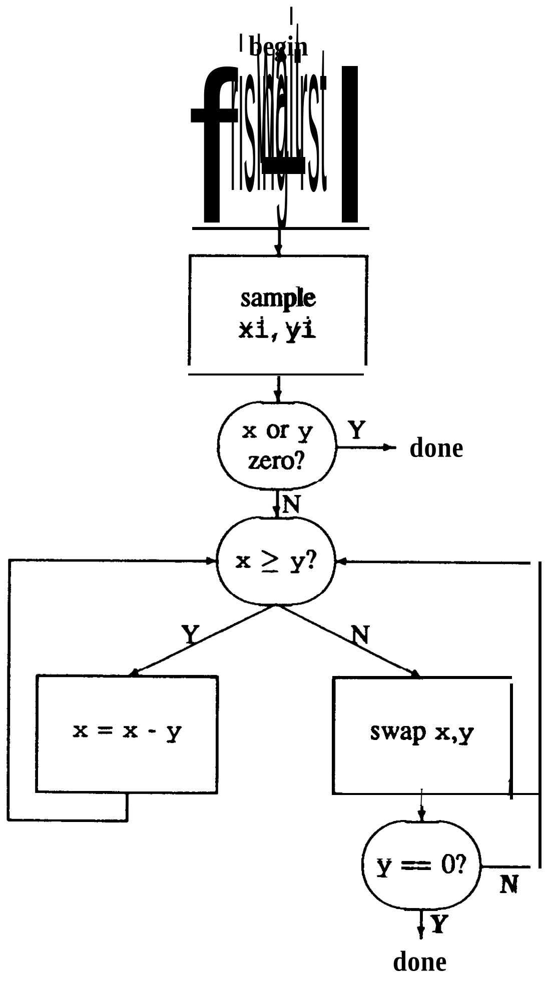
where *name* is an identifier consisting of an alphanumeric string of letters and digits, with the first character being a letter. ***parameter list*** is a list of identifiers separated by commas, each representing a formal parameter of the model. The size and type of the formal parameters are declared ***in parameter declarations,*** with the content of ***the*** model contained in ***block-body.***

- ***Process model*** – A process consists of a hierarchy of procedures or functions, and executes concurrently with processes in the system. A process will restart itself upon completion of its execution, implying that an operation within a process will be activated at most once during each execution of the process. The formal syntax of process definition is:

  process name (parameter list)
      parameter declarations
      process-body

- ***Procedure model*** – A procedure is an encapsulation of operations, and may consists of calls to other procedures or functions. A procedure executes whenever it is called by another process, procedure, or function, whereupon the control flow is temporarily transferred to the called model. Upon completion, the control flow returns to the calling model. No recursion is permitted in the language. The formal syntax of procedure definition is:

  [**procedure**] ***name (parameter list)***
      parameter declarations
      procedure-body



Flow chart of algorithm
(a)

```
process gcd(xi,yi,rst,ou)
   in port xi[8], yi[8];
   in port rst;
   out port ou[8];
[
   static x[8] = 0, y[8] = 0;
   /* set output to 0 during algorithm */
   write ou = 0;
   /* wait until rising edge of rst */
   while ( ! rst )
   ;
   /* sample input */
   {
      x = read(xi);
      y = read(yi);
   }
   if ((x != 0) & (y != 0)) {
      /* using euclid's gcd algorithm */
      repeat {
         while (x >= y)
            x = x - y ;
         /* x should be less than y now */
         /* so exchange x and y */
         < x = y ; y = x ; >
      } until ( y == 0 );
   }
   write ou = x;
1
```

HardwareC process
(b)

Figure 3: Example of process that repeatedly samples the inputs on the rising edge of `r st`, then finds the greatest common divisor of two input values using Euclid's algorithm.

Note that the keyword **procedure** is optional, where [ ] denotes an optional clause.

- *Function model* – A function is semantically equivalent to a procedure, with the difference that a function returns a scalar or vector of Boolean values to the calling model, e.g. an add function returns the sum of two operands. The formal syntax of function definition is:

> **function** ***name (parameter list)***
> **return boolean**[***return-size***]
> ***parameter declarations***
> ***function-body***

where ***return-size*** is the return size of the function. If ***return-size*** is not specified, then the default return size is 1. To return values to the calling model, explicit assignments are made to a keyword **return_value** in the body of the function. The size of **return_value** is identical to the size of the function model. The last assigned value to **return_value** is the value that is returned.

Example 2.1.1. To illustrate the definition of models and parameters, consider the example below.

```
/*  SimpleAdd -
 *      returns the sum of two S-bit operands
 */
function SimpleAdd(opl, op2) return boolean[6]
    in boolean opl[5], op2[5];

    /*  return_value is keyword  */
    return_value = opl + op2;
}
```

SimpleAdd is a function that returns the sum of two input operands. □

2.2 Synchronous Synthesis Paradigm

HardwareC is a hardware description language with specific ***constructs*** for ***the*** design of ***synchronous*** digital circuits controlled by a single-phase system clock, as synthesized by the ***Hercules*** system. In particular, HardwareC supports:

- *Synchronous I/O operations*
- *Synchronous message passing*
- *Synchronous register loading*

These operations are assumed to be synchronized to a clock cycle, and to take an integral number of clock periods, or equivalently, an integral number of ***control states,*** to execute. Similarly, the constructs of data-dependent iteration and process in HardwareC have an underlying synchronous model, i.e. the tasks described by the iterative and process constructs take an integral number (possibly zero) of clock cycles to execute.

2.3 Declare Before Use

Whenever a model is called, the arguments to the invocation are checked for both compatibility in the variable size and type, as well as for compatibility in the direction of the formal parameter (`in`, `out`, or `inout`). For instance, an input parameter cannot be used as the argument to a call that requires an output parameter. Similarly, an output parameter cannot be used as the argument to a call that requires an input parameter. This compile time consistency check improves the security of the language.

In order to provide this information to the parser, it is necessary to ***declare*** a model before it can be called. The declaration of a model involves specifying its parameters without describing its body. Syntactically, a keyword **declare** is prefixed on a model definition, where the body of the model is left out. In particular,

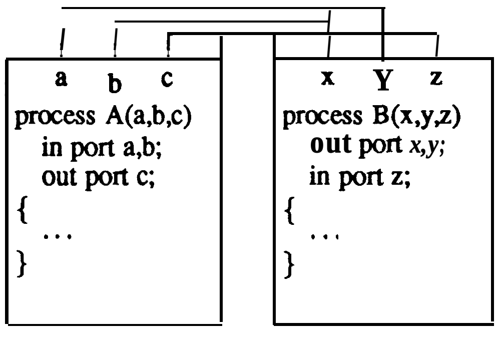
> **declare block** ***name (parameter List)***
>   ***parameter declarations***
>
> **declare process** ***name (parameter list)***
>   ***parameter declarations***
>
> **declare [procedure]** ***name (parameter list)***
>   ***parameter declarations***
>
> **declare function** ***name (parameter list)***
>   **return boolean[** ***return-size*** **]**
>   ***parameter declarations***

The names that are used in the parameter declarations need not be the same as the names of the parameters in the actual definition of the model.

Example 2.3.1. The following example illustrates the syntax of model declaration by declaring the `SimpleAdd` function in the previous section.

```
declare function SimpleAdd( x, y ) return boolean[6]
    in boolean x[5], y[5];
```

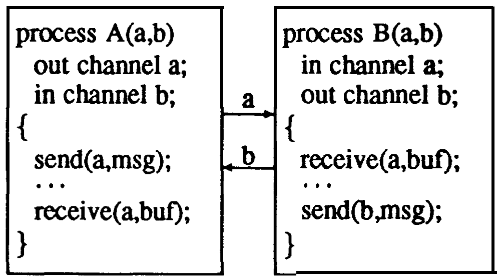
Note that the names of the parameters in the declaration (x and `y`) need not match the **names** in the definition (`opl` and `op2`). □



```
process A(a,b,c)
  in port a,b;
  out port c;
{
  ...
}
```

```
process B(x,y,z)
  out port x,y;
  in port z;
{
  ...
}
```

Port passing paradigm
(a)



```
process A(a,b)
  out channel a;
  in channel b;
{
  send(a,msg);
  ...
  receive(a,buf);
}
```

```
process B(a,b)
  in channel a;
  out channel b;
{
  receive(a,buf);
  ...
  send(b,msg);
}
```

Message passing paradigm
(b)

Figure 4: Port passing versus message passing mechanisms.

2.4 Parameters to Models

The transfer of data to and from the models is accomplished through the use of two mechanisms – ***port passing*** and ***message passing.*** Both mechanisms involve passing information in the form of ***formal parameters*** to a model, and are illustrated in Figure 4.

- **Port passing** – assumes the existence of a shared medium, such as wires or memory, that interconnects the hardware modules implementing the models. The protocol which governs correct handshaking between the modules is provided by the designer, and is described as an integral part of the high-level description. The ports are further categorized into ***global*** and ***local*** ports.
  - *Global port: Any* access to a global port is *immediately* reflected to the other connected models. For blocks, the global ports serve as linkages between the processes and define their structural relationships. For processes, procedures, and functions, global ports allow direct access to external signals, regardless of the nesting depth of the calling hierarchy. In

particular, the value of a global port may change during the execution of the model. The syntax of global port declaration is:

{ **in** | **out** | **inout** } **port** *portname* { , *portname* }*;

where *portname* is the name of the global port, and the notation { }* means zero or more occurrences of the quantity enclosed in the curly braces. A global port can be a scalar or a vector, the size of the port is enclosed in square brackets. For example, `in port rst;` describes a `scalar` global `port rst`, and `out port sum[5];` describes a global port `sum` with 5 elements.

A global port **can be in** or **out** or `inout`, depending on whether the port is read-only, write-only, or read-write. `inout` global ports are bidirectional lines that can either be referenced or modified. The access protocol of this bidirectional line is left to the designer, and specified as part of the design description.

– *Local ports:* In contrast to global ports, a local port is used to transfer data across the calling hierarchy, e.g. calling a function with a set of operands, and obtaining the results after the function completes execution. Local ports are not used to interconnect concurrently executing processes, and hence are not defined for blocks and processes. The value of a local port remains unchanged during the execution of the corresponding model (either procedure or function). The syntax of local port declaration is:

{ **in** | **out** } **boolean** *portname* { , *portname* }*;

where *portname* is the name of the local port. A local port can be either in or `out`, depending on whether the local port is read-only or write-only in the corresponding model. An input local port is assumed to not change in value during the execution of the model, and an output local port will reflect its value only when the model completes execution.

- Message passing – via explicit *send* and ***receive*** operations can be used for both synchronization and data-transfer. The message passing paradigm is synchronous and blocking, meaning that the sending process will wait until the corresponding receiving process has acknowledge the message. Likewise, a receiving process will wait until the corresponding process sends the message.

  The transfer of information takes place on ***channels*** that interconnect the models. The corresponding hardware for communication, as well as its handshaking protocol, are automatically synthesized by **the** synthesis tools. Channels **can be scalars** or **vectors; the** size of a channel is the **size** of the **data** that is transferred on the channel. The syntax of channel declaration is:

  { **in** | **out** } **channel** *channelname* { , ***channelname*** }*;

  where ***channelname*** is the name of the channel. An `in` channel can receive incoming messages, whereas messages can be sent on an **out** channel to other models.

Due to its declarative semantics, a block can only have global ports and channels. A process can also only have global ports and channels because of its repetitive execution. For procedure and function models, both local and global ports are allowed in addition to channels.

Example 2.4.1. **An Extended Example.** We will illustrate the concept and syntax of models through a simple design. The design contains two counters, which are activated depending on the value of the `select` signal. Each counter uses a two-stage ripple-carry adder. We begin the description in a top-down fashion, starting with the topmost model `main`.

```
block main(select, data, resultl, result2)
    in port select;
    in port data[8];
    out port result1[8], result2[8];
<
    counter(select, data, resultl);
    counter(!select, data, result2);
>
```

Note that `main` is a block, consisting of an interconnection of `two` identical processes `counter`. The idea is to increment the appropriate counter with ***data,*** based on the Boolean value of ***select. Note*** that the declarative semantic of blocks implies that all operations within a block execute concurrently without control flow dependencies. This is indicated by the use of parallel compound statements (< >) to encapsulate the operations. Compound statements will be described further in Section 8.1. Now we describe the counterprocess.

```
process counter(select, data, result)
    in port select;
    in port data[8];
    out port result[8];

    static value[8];
    boolean carry;

    if ( select )

        /* increment */
        adder(value, data, value, carry);

    result = value;
```

The counter process **first** checks whether it is selected, then increments the value of an internal register ***value*** by the appropriate amount by calling a procedure `adder`. To make the example more interesting, we describe the adder procedure as consisting of two 4-bit addition functions. Note that ***value*** appears `twice in the call to adder`. The first refers to the value of ***value,*** and the second indicates that ***value*** is to be overwritten by the new result. Therefore, the "reference" to a variable is decoupled **from** the **"modification"** to the variable. We describe the adder procedure as follows.

```
procedure adder(op1, op2, result, carryout)
    in boolean op1[8], op2[8];
    out boolean result[8];
    out boolean carryout;
{
    boolean carry;

    result[0:3] = add4bit(op1[0:3], op2[0:3], 0, carry);
    result[4:7] = add4bit(op1[4:7], op2[4:7], carry, carryout);
}
```

The adder **procedure** makes two calls to **a 4-bit** addition function. Since **the** value of **the** result is **valid** only after the **procedure** completes execution, ***result*** and ***carryout*** are output local ports. Finally, we describe **the** addition function.

```
function add4bit(op1, op2, cin, cout) return boolean[4]
    in boolean op1[4], op2[4];
    in boolean cin;
    out boolean cout;
{
    int i;
    boolean carry;

    carry = cin;
    for i = 0 to 3 do
    {
        return_value[i] = op1[i] ^ op2[i] ^ carry;
        carry = op1[i] & op2 [i] I carry & (op1[i] | op2[i]);
    }
    cout = carry;
}
```

Recall that **return-value** is **the** return value of **the** function model, and its size is equal to **the** size of the function model (4 in this case). □

As a final point, a procedure or function **cannot** he defined within the **body** of another model. This restriction follows the **C** language, which disallows nested procedural definitions. The resulting flattening of the procedural definition is appropriate since for hardware description, it is more appropriate to identify explicitly all inputs **and outputs to a given model. Otherwise, a procedure defined within the scope** of **another allows access to all variables that are defined within the scope of its definition, and hence its boundary is not well defined if nested procedural definitions are allowed.**

block	**boolean**	**break**	**case**
channel	**constraint**	**cycles**	**declare**
default	**delay**	**do**	**downto**
else	**for**	**free**	**from**
function	**if**	**in**	**inout**
instance	**int**	**load**	**maxtime**
mintime	**msgwait**	**of**	**out**
port	**procedure**	**process**	**read**
receive	**register**	**repeat**	**reset**
return	**return_value**	**rl**	**rr**
send	**static**	**step**	**switch**
tag	**template**	**to**	**until**
while	**with**	**write**	**xor**

Figure 5: Keywords in HardwareC.

3 Constants and Variables

There are two types of ***data*** entities in the language – constants and variables. They comprise the basic data objects that are manipulated in a model. Constants can be decimal, hexadecimal, or binary, depending on their prefixes. Variables are named via ***identifiers.*** An identifier is a sequence of letters and digits; the first character must be a letter. The underscore "_" counts as a letter. Upper and lower cases are differentiated. The identifiers in Figure 5 are reserved for use as keywords, and may not be used as variable names.

Note also the user is adviced against naming variables as **T***n*, **M***n*, or **X***n*, where ***n*** is an integer value, e.g. T3, M2, or X1. The reason is because the Hercules synthesis system may automatically create temporary variables whose names may coincide with these user-defined names. Although the system automatically resolves all naming conflicts, the user-defined names may be changed.

3.1 Constants

There are several types of constants in the language – ***decimal constants, hexadecimal constants,*** and ***binary constants.*** A sequence of digits is taken to be hexadecimal if it is prefixed by 0x or 0X. The hexadecimal digits include a or A through f or F with values 10 through 15. A sequence of binary digits 0 or 1, prefixed by 0b or 0B, is a binary constant. A sequence of digits without prefixes ranging from 0 through 9 is a decimal constant. We now state the convention of interpreting the order of the digits.

Convention: The default convention for constants is from the most significant digit (MSD) to the least significant digit (LSD).

The convention is consistent with the C programming language. For example, in the default convention, 0x03 is 3 and 0x0F is 15 [1]. The designer needs to ensure that the selected convention is used uniformly throughout the specification. Note that for decimal constants, the first appearing digit is ***always*** the most significant digit.

Negative constants are represented using two's complement convention. In particular, if the most significant bit is 1, then the constant is negative. Otherwise, the constant is positive.

Example 3.1.1. Consider the assignment of -5 to a four-bit quantity `signal[4]`, e.g. `signal = -5`. The assignment is equivalent to `signal = 0b1011` assuming the default convention of MSB to LSB. The assignment is equivalent to the following **bitwise** assignments:

```
signal[3:3] = 1;     /* MSB */
signal[2:2] = 0;
signal[1:1] = 1;
signal[0:0] = 1;     /* LSB */
```

The -5 is interpreted in terms of two's complement representation for 4 bits, or 0b1011. □

3.2 Variables

A ***variable*** in the language is used to access the results of computation in a given program. When a program references a particular variable at different locations in the code, it may reference different ***values,*** depending on whether the variable has been re-assigned between the references. The ***value*** of a variable is defined to be the data most recently assigned to it, where data is defined to be the results of procedure call, binary and unary operators, I/O command, or message passing.

There are three major ***variable types*** in the language – **int**, **boolean** and **static**, corresponding to integer, Boolean, and register variables. In the mapping to final hardware implementation, `boolean` variables are synthesized either as wires or registers, and `static` variables are always synthesized as registers. In contrast, `int` variables are provided for the convenience of the description, and will be resolved at compile time during the synthesis tasks. The integer variables ***are*** often called ***meta-variables*** to emphasize the fact that they are not synthesized in the final hardware implementation.

All variables must be declared before use, and can be declared within any compound statements in the description. A declaration specifies the type and size of a given variable. Following the semantics of structured languages, a variable is visible only within the scope of its definition. A variable with the same name at a deeper nesting block level will override any current definition of the variable.

Example 3.2.1. For example, all nested declarations in the following code segment are valid.

```
{
    int     i;
    boolean x;
```

The Hercules/Hebe synthesis system has the option of adopting the alternate convention from LSD to MSD.

```
          int     i;      /* new i */
          boolean x;      /* new x */
          boolean y;
          ...

      /* y is not defined here */
```

Note that the Boolean variable **y** is not defined in the **outer** compound statement. □

No global variables are allowed in *HardwareC*. This restriction is due to the fact that global variables allow side effects that are not explicitly identified. This is undesirable from the standpoint of security, verifiability, and program readability. If some data must be shared between two models, then the data should be explicitly specified as parameters to the two models that are connected. We now describe each of the three variable types in detail.

3.2.1 Integer Variables

Integer variables may be used in any expression, including arithmetic, logic, and relational expressions, and can only be scalar quantities. They are mainly used as indices to constant iteration loops (`for` loops), and as indices for accessing components of Boolean vectors.

> **Warning:** Integer variables (also called meta-variables) are valid **only** when their values can be resolved at compile time.

Integer variables are used only for the convenience of the description, and are **not** synthesized into hardware. This restriction disallows the referencing of integer values that are updated either in a data-dependent loop or conditional construct. The examples below illustrate the restriction.

Example 3.2.2. The following examples illustrate invalid references of an integer variable inside first a conditional, then a data-dependent loop. The reason is because it is not possible to determine the value of the integer variables at compile time.

```
int i;
boolean a[4];

if ( a > 5 )
    i = 0;
else
    i = 1;

/* referencing i is invalid here */
```

Specifically, depending on the value of the **Boolean** variable a, `i` is set to either 0 or 1. Therefore, `i` cannot be resolved statically, and hence the reference is invalid. The code segment below illustrates **the invalid** use of integer variables in data-dependent loops.

```
int i;
boolean a[4];

while ( a < 5 )
{
    /* referencing i is invalid here */
    i = i + 1;
}
/* referencing i is invalid here also */
```

The referencing of `i` will be invalid since the number of times the loop iterates is not known at compile time. In order to describe the iteration, it is necessary to use a Boolean variable, as shown below.

```
boolean new_i[8];         /* 8-bit number */
boolean a[4];

while ( a < 5 )
{
    new_i = new_i + 1;
}
/* new_i contains the number of loop iterations */
```

□

The following examples demonstrate the use of integer variables **and** expressions in accessing components of a Boolean vector, and in control structures.

Example 3.2.3. The `swap` procedure swaps two four-bit **halfs** of the input vector.

```
/*
 *    swaps the two nibbles in "a",put result in "b"
 *        b[0:3] <= a[4:7]
 *        b[4:7] <= a[0:3]
 */

swap (a, b)
    in boolean a[8];
    out boolean b[8];
{
    int i, j;
```

```
        /* copies LSB nibble to b bit-by-bit */
        for i = 0 to 3 do
            b[ i+4:i+4 ] = a[i:i ];

        /* copies MSB nibble to b */
        b[ 0:3 ] = a[ 4:7 ];
    }
```

The exact syntax on accessing components of a Boolean vector is discussed in the next section. □

Example 3.2.4. The example below produces an output Boolean vector of a given pattern, where the least significant 3 bits are set to all one's.

```
    int i;
    boolean vec[24 ];

    for i = 0 to 7 do {
        switch (i) {
        case 0:
            vec[ 3*i:3*i+2 ] = 0x7;   /* binary 111  */
            break;
        default:
            vec[ 3*i:3*i+2 ] = i;
            break;
        }
    }
```

Upon execution, the variable vec should have the following values, starting from the MSB (bit 23) to the LSB (bit 0): `111 110 101 100 011 010 001 111`. □

3.2.2 Boolean Variables

In contrast to integer variables, boolean variables have a corresponding mapping to a hardware implementation. A boolean variable may be implemented as wires or registers, depending on the decisions of the synthesis system. A boolean variable represents one or more signals, where each bit of the variable corresponds to a *signal* that can be either 0 or 1. The total number of bits is the *size* of the variable. If the size is one, then the boolean variable is a *scalar*; otherwise, it is a vector.

A number is represented in boolean variables using the 2's complement convention, and the range of values a boolean variable can take depends on its size. For example, a scalar boolean variable can assume the values of 0 and 1, and a vector of size n can assume the values ranging from $-2^{n/2} \ldots (2^{n/2} - 1)$. The indices of a vector start from 0 to $n-1$, where index 0 corresponds to the least significant bit (LSB), and index $n-1$ corresponds to the most significant bit (MSB).

A boolean variable is initialized to zero, and its value is not saved across procedure invocations. That is, the value assumed by a boolean variable defined in a given model will not be retained the next time the model is invoked. The formal syntax of boolean variable declaration is:

boolean *var* { , *var* }*

where *var* is the name of the variable, and the notation { }* means zero or more occurrences of the quantity contained within the curly braces. The size of the variable can also be specified as part of the declaration, with the default size being one (scalar). The following declarations are all valid boolean variables. In particular, a is a scalar, and b is a vector of five elements starting from index 0 through 4.

```
boolean a;          /* scalar */
boolean b[5];       /* vector */
```

Both boolean variables and static variables (described in the next section) represent Boolean values, and are referred to as ***Boolean variables.*** In Boolean vectors, specifying the variable name without brackets, or with empty brackets, represent the ***entire*** vector. For example, b and b [ ] are equivalent to b[0:4]. In addition, it is also possible to access a ***subrange*** of values in a vector. This is specified by the colon (:) notation. For example, b[2:3] represents a vector of two values that corresponds to the third and fourth element of b. The most significant bit (MSB) is always the higher index, with the least significant bit (LSB) being the smaller index. In specifying subranges, the user can specify the lower and upper indices in any order, either *var*[*lower*:*upper*] or *var*[*upper*:*lower*], *e.g.* b[2:3] = b[3:2].

Integer variables and expressions can be used in variable declarations to specify the dimensions of the variable, or they can be used to access components and subranges of Boolean variables. In fact, integer expressions can be used whenever constants are required, e.g. in port/channel declarations or variable declarations.

Example **3.2.5.** The integer variable i is used for both accessing subranges of a Boolean vector, as well as in the declaration of a variable q.

```
int i;

i = 3;
b[i:i+1] = b[i-2:i-1]; /* b[3:4] = b[1:2] */
{
    boolean q[i+1];     /* q has 4 elements */
}
```

The integer variable i can be used in variable declarations since it is always resolved statically. □

3.2.3 Static Variables

Static variables are similar to boolean variables, with the semantic difference that their values are retained across procedural invocations. Since static variables have ***state information,*** they are implemented as registers in the resulting hardware. The formal syntax for `static` variable declaration is:

> ***static var*** [= ***init_value***] { , ***var*** [= ***init_value***] }*

where ***var*** is the name of the variable. Static variables may be optionally initialized to a given value, which corresponds to initializing the register implementing the variable ***to an initial*** value. ***hit-value can*** be any constant or integer expression. If ***init_value*** is not specified, then the initial value is assumed to be 0.

Example 3.2.6. A typical use of static variables is in storing state information for finite automata machines. An oscillator model is described below.

```
process oscillate(clear, clk)
    in port clear;
    out port clk;

    static state = 1;

    state = !clear & !state;
    clock = state:
}
```

The value of ***state*** depends on the value of ***clear*** and the value of ***state*** in the previous execution. □

4 Expressions

The constants and variables are combined using binary and unary ***operators*** to form ***expressions. There are*** four major types of expressions in the language – ***arithmetic, logical, relational,*** and auto. The operators can be unary or binary, and take both integer (i.e. `int`) and Boolean (i.e. `boolean` and `static`) variables as operands. Integer expressions or variables can be used as constants in an expression because they are always resolved at compile time. Figure 6 summarizes the operators.

- **Arithmetic Operators - The bii arithmetic operators are +, −, *, and /. There is an unary −, but no unary +. The + and − operators have the same precedence, which is lower than the (identical) precedence of * and /, which are in turn lower than unary minus. Arithmetic** operators, **as in [illegible] are grouped left to right The order of evaluation is not specified for associative** and **commutative operators like * and +, and the synthesii system is free to interpretate the order of** evaluation, **either $(a+b)+c$ or $a+(b+c)$. The** arithmetic **expressions** assume the two's complement convention for **the** operands.

Type	*Operator*	*Description*
Arithmetic	+	binary addition
	−	binary subtraction or unary minus
	*	binary multiplication
	/	binary division
Logical	&	bitwise AND
	|	bitwise OR
	xor or ^	bitwise XOR
	!	unary bitwise NOT
	@	binary concatenate
	rr	rotate right
	rl	rotate left
	>>	shift right
	>>	shift right
Relational	>	greater than
	<	less than
	>=	greater equal
	<=	less equal
	! =	not equal
	==	equal
Auto	++	auto-increment
	−−	auto-decrement

Figure 6: Valid operators in HardwareC.

- **Logical Operators** - The binary logical operators are &, |, and *xor* (or equivalently ^), corresponding to the bitwise AND, bitwise OR, and bitwise XOR operations. The unary ! complementation operator takes the bitwise complement of a given value or variable.

  A value or variable may be shifted by the shift and rotate operators. In particular, < < shifts to the left, > > shifts to the right, *rl* rotates to the left, and *rr* rotates to the right. Furthermore, the binary concatenation operator @ concatenates the left operand as the most significant portion with the right operand as the least significant portion. For example, `b[0:7] = x[0:3] @ y[0:3];` will assign `x[0:3]` to the least significant bits `b[0:3]`, and `y[0:3]` to the most significant bits `b[4:7]`.

- **Relational Operators** – The relational operators consist of >, > =, <, and < =, all with the same precedence. Just below them in precedence are the equality operators: == and ! =, which have the same precedence. Relationals have lower precedence than arithmetic operators, so expressions like $x < num + 3$ *are* taken *as* $x < (num + 3)$. The relational expressions are assumed to use two's complement convention for the operands in the comparisons.

- **Auto Increment and Decrement** – Similar to C, HardwareC provides auto-increment (+ +) and auto-decrement (− −) operators. For example, $a++$ and $++a$ are equivalent to $a = a + 1$, whereas $a--$ and $--a$ are equivalent to $a = a - 1$. However, there is one major difference: an auto-incremented or decremented expression ***cannot be referenced.*** That is, $b = a++$ is illegal. This difference arises from the convention that only assignment statements can affect the value of a given variable, thereby disallowing the side-effects of referring to ++ and −− expressions.

5 Assignments

An ***assignment*** to a variable modifies the ***value*** of the variable, and affects subsequent references to the variable. Both variables (either **int, boolean,** and **static**) and output ports can be assigned. We describe the assignment to variables in this section. Assignments to ports are described in Section 6.1.

The formal syntax of an assignment is:

variable = expression;

where ***expression*** is either an arithmetic, logical, or relational expression, an I/O read statement, an IPC receive or msgwait statement, or a function call. Only constants or integer expressions can be assigned to integer variables. There is no restriction on the values that can be assigned to Boolean variables.

While a `static` variable is always implemented by a register, an assignment to a static variable may not require the update of the corresponding register. This is because it is possible to group together multiple assignments to a static variable, so that a single register update is necessary to effect the multiple assignments. However, the user may explicitly load the implementing register with a new value via the **load** statement. Load statements are applicable to only static variables. The formal syntax of the load statement is:

load ***static-variable = expression;***

The value of *static_variable* will be updated one control state after the execution of the assignment.

6 Input/Output

Data is transferred to and from a particular model using either port passing or message passing mechanisms. In this section, we present the port passing paradigm of ***port assignments*** and ***explicit I/O commands***. We describe the message passing paradigm in Section 7.

6.1 Port Assignments

Assignment to an output port will update the *value* of the corresponding port, and hence *do not* consume a control state during execution of the corresponding hardware. The way in which this value is reflected to the other models depends on whether the port is local or global.

For local ports, The value that is returned is the last assigned value in the model. For global ports, on the other hand, explicit I/O commands are used to reflect a value on the corresponding global port. If there are no explicit I/O commands for a given global port, then the value that is returned is also the last assigned value in the model (similar to local ports). Otherwise, assignments to global ports are ignored if there are explicit I/O commands made to the ports. Assignments `are` illegal for `inout` global ports, for which all accesses must be made with explicit I/O commands.

Example 6.1.1. To illustrate the return value of a local port as the *last* assigned value in the model, consider the model `lastone`.

```
procedure example( result )
    out boolean result[4];

    result = 1;
    result = 2;
    result = 3;
    result = 4;
    result = 5;   /* last assigned value */
```

The output local `port result` will return 5 to the calling model. Each assignment will not take a control state during hardware execution time. □

The return value of a function is treated as an output local port since it is not valid until the completion of the function. Assignments are made to the reserved keyword **return_value**, which has size equal to the function model's return size. The last assigned value to **return_value** is the value that is returned to the calling model.

6.2 Explicit I/O Commands

I/O commands explicitly access the ***global ports*** for writing or reading (sampling). An explicit I/O command immediately reads or modifies the value of a given global port across the hierarchy. All I/O commands consume one control state during hardware execution time (i.e. it is synchronous). The three I/O commands that operate on global ports are described below.

6.2.1 Write Statement

A ***write statement*** writes a given value to the corresponding **out** or `inout` **global** port. **Each write consumes** a control state, and reflects the assigned value to the given port in the next cycle. Any **value** written to a global port will be retained until either the next **write** or **free** statement. The formal **syntax** of a write statement is:

write ***port = expression;***

where ***port can be*** either the entire global port, or specific subranges of the port. In the example below, **the out port** c will generate a pulse:

```
write c = 0;    /* port has 0       */
write c = 1;    /* port has 1       */
write c = 0;    /* port has 0 again */
```

6.2.2 Free Statement

A ***free statement*** **sets the corresponding** `inout` global port to high impedance float value. For both **free** and **write** commands, the effect of the change on the global port will take place exactly one cycle after the statement begins execution. Any **write** to a global port that has been set to float state will overwrite it with the new value. The formal syntax for the free statement is:

free ***port*** { **,** ***port*** }*;

where ***port*** can be either the entire port, or specific subranges of the port. For example, if d is an inout global port, then the following code segment will force d to be 1, then high-Z, then 1 again.

```
write d = 1;    /* d has 1       */
free d;         /* d has high-Z */
write d = 1;    /* d has 1 again */
```

6.2.3 Read Statement

A ***read statement*** samples the corresponding **in** or `inout` global port into a register, and returns the output of the sampling register. Execution of a **read** statement will take one cycle to complete. The formal syntax for read statement is:

***var* = read (*port*);**

For example,

```
y = read( x );
/* y is sampled version of x */
```

Example 6.2.1. Explicit I/O commands can be used to implement a given handshaking protocol. In the function `memory-read` below, the model first waits until the request line (`rq`) goes low, upon which it asserts the acknowledge line (`ak`) and writes the address onto the address lines (`addr`). The `data` lines of the external memory (`data`) are then read in.

```
function memory-read(addr, data, ak, rq, val) return boolean[8]
    out port addr[16];      /* address line */
    inout port data[8];     /* data line */
    out port ak;            /* request line */
    in port rq;             /* request line */
    in boolean val[16];     /* addr to read */
[
    while ( rq )        /* wait */
        ;
    <
        write ak = 1;           /* take line */
        [
            write addr = val;    /* put address */
            return_value = read(data);
        3
    >
    write ak = 0;
1
```

Notice that the statements are executed in serial order ([ ] compound statement) to ensure that the reading of the data lines takes place after the synchronization with `rq`. □

7 Message Passing

HardwareC supports synchronous blocking *send-receive* message passing scheme. The medium of transfer is called a *channel. The* size of a channel represents the number of bits that are communicated between two models, and may be specified by the designer in the channel declaration. Synchronous message passing suspends the execution of a model until the message is received or acknowledged. It provides a simple yet powerful approach to interprocess synchronization and limited data transfer without incurring the cost of message buffering.

There are three primitive *operations in* message passing: *send, receive,* and *msgwait.* We describe each operation in turn.

7.1 Send

The *send* statement transmits a message on a given `out` channel. The current model will wait and synchronize until the receiving model issues a ***receive*** on the given channel, whereupon the transfer of information will take place. For example, let ***out-channel*** be an output channel of size ***n***, then

send (***out-channel, message***);

will send a message of size ***n*** on the channel ***our-channel. out-channel*** must refer to the entire channel, i.e. sending on a portion of a channel is not allowed.

7.2 Receive

A ***receive*** statement accepts a message from an `in` channel, and will wait and synchronize until a message is sent on the given channel. For example, let ***in-channel*** be an input channel of size n, then

buffer = **receive** (***in-channel***);

will receive a message of ***size n*** on ***the*** channel ***in-channel,*** and assign it to the variable ***buffer. in-channel*** must refer to the entire channel, since receiving from a portion of the channel is not allowed. Note that a register is automatically created to sample the incoming message.

7.3 Msgwait

The msgwait command is a query that returns a scalar Boolean flag indicating whether the specified input channel has pending messages. Let *in-channel* be an input channel, then

flag = **msgwait** ***(in-channel***);

will return a scalar flag depending on whether there are messages pending from the given channel.

Example 7.1. An example illustrating the message passing constructs is given below as a producer-consumer process interaction.

```
process Producer (A, B)
    out channel A [8];
    in channel B[4];

    boolean    data[8], buffer[4];

    data = . . . .
    send( A, data );        /* send    */
    buffer = receive(B);    /* receive */
1
```

```
process Consumer(X, Y)
    in channel X[8];
    out channel Y[4];

    boolean     data[4], item[8];

    if ( msgwait(X) )

        item = receive(X);
        data = . . .;
        send(Y, data);
    ]
    else
        /* producer not ready */
```

The producer generates a data value, then sends the information out to the consumer on the 8-bit channel A. It then waits and receives data from an input 4-bit channel B. In the consumer, data is received from the input channel X, and information is sent on the output channel Y. How the channels are connected among the processes determines the interaction among the processes. □

7.4 Channel Variables in Blocks

The flexibility of the message passing mechanism is largely due to the freedom of the designer to determine how the channels are interconnected among the models. Such structural interconnection is specified at the block level. To make possible interconnection among channels, *channel* variables are defined in a block model to describe the linkages among the processes.

The formal syntax of channel variable declaration is:

channel ***channel*** { , ***channel*** }*;

where ***channel*** is the name of the channel variable. The size of the channel variable can be optionally specified, the default being scalar (size of 1). Note that channel variables are not defined in processes, procedures, or functions.

Example 7.2. We illustrate the use of channel variables by extending the previous producer-consumer example to show how the channels are connected together.

```
block main( . . . )
<
    channel wide[8];
    channel narrow[4];

    Producer (wide, narrow);
    Consumer (wide, narrow);
>
```

Note that it is important to ensure an input channel of one process is connected to an output channel of another process. □

8 Control Flow

The control flow statements of a language specify the order the computations are carried out. HardwareC supports a single-in, single-out control flow, similar to the Pascal programming language. This implies that no ***gotos***, ***breaks*** from loops, and ***returns*** are allowed in the language. Such restriction is appropriate since by supporting a single-in, single-out control flow, the hardware semantics of the language is made simpler, which greatly aids in the correctness verification of programs.

An expression such as $a = 3$ or $b++$ becomes a ***statement*** when it is followed by a semicolon, as in

```
a = 3;
b++;
```

As in C, the semicolon is used as a statement terminator, rather than a separator as in Algol-like languages. A statement can either be a variable assignment, an if-then-else statement, a switch statement, a for statement, a while statement, an input/output statement, a message passing primitive, or a compound statement. A semicolon by itself represents a null statement.

8.1 Compound Statement

A ***compound statement*** is used to group variable declarations and statements together so that they are syntactically equivalent to a single statement. ***HardwareC*** supports three types of compound statements – ***data-parallel, serial,*** and ***parallel*** compound statements. Data-parallel compound statements are encapsulated using curly braces ({ and }), serial compound statements are encapsulated using square brackets ([and]), and parallel compound statements are encapsulated using angle brackets (< and >). The differences between the different types are:

- Data-parallel Compound Statement { } – The statements within a data-parallel compound statement can all execute in parallel, subject to the data-dependencies that exist between the statements. For example,

  ```
  {
          variable_declarations;

          statement1;
          statement2;
  }
  ```

  means that ***statement1*** can be executed concurrently with ***statement2*** if the statements are data-independent. The degree of parallelism is determined by the synthesis system. A data-parallel compound statement completes execution when the last statement in the compound statement completes execution.

- Serial Compound Statement [ ] – The statements within a serial compound statement are guaranteed to execute in serial order, starting from the first statement in the compound statement. For example, ***statement1 will*** always ***execute*** before ***statement2,*** regardless of their data-dependencies.

```
[
        variable_declarations;

        statement1 ;
        statement2;
]
```

Serial compound statement allows the designer the ability to specify control dependencies between otherwise data-independent statements. A serial compound statement completes execution when the last statement in the compound statement completes execution.

- Parallel Compound Statement < > – The statements within a parallel compound statement are ***guaranteed to*** execute ***in*** parallel. For example, ***statement1*** will always execute concurrently with ***statement2.*** A parallel compound statement is completed when all statements in the compound statement have completed execution.

```
<
        variable_declarations;

        statement1;
        statement2;
>
```

Since all statements are executed in parallel, a variable can be assigned to at most once in the compound statement. The values of the variables that are referenced inside the compound statement are variable values just before entering the compound statement.

Example 8.1.1. We illustrate the differences between the data-parallel { } and parallel < > compound statements. Consider the swapping of two boolean variables x and y. Let us make a first pass on by using data-parallel compound statements.

```
{
    x = y;
    y = x;
```

Unfortunately, the code segment above will ***not*** perform the swap. The first assignment transfer the value of y to x, so by the time the second assignment is executed, x already assumes the value of y, and hence the second assignment is equivalent to assigning **y** onto itself. The reason lies in the data-dependency that exists between the two assignments that forces them to be executed in series.

Let us try again, this time using `an` intermediate variable `temp` to temporarily hold the swapped value.

```
{
    boolean temp;

    temp = x;
    x = y;
    Y = temp;          /* x swap y */
```

This time the code segment will perform the swap, which is analogous to the procedural semantics of software programming languages. Let us now use the parallel compound statements to describe the swap.

```
<
    x = y;
    Y = x;
>
```

The parallel compound statement guarantees that the two assignments will be carried out in parallel, and that the right-hand side of the assignments refer to the values of the variables ***before*** entering the compound statement. Therefore, the assignment x = y is semantically equivalent to x = ***previous value of*** y, similarly, the assignment y = x is equivalent to y = ***previous value of* x.** □

8.2 If-Else Statement

`Theif-else` statement is used to make branching decisions based on the value of a particular expression. Formally, the syntax is

if *(expression)*
    statement-l
[***else***
    statement-2]

where the ***else*** part is optional. ***If the expression*** is evaluated ***to be*** nonzero (or "true"), then ***statement-1*** is executed. Otherwise, if ***the*** else part is specified, ***statement-2*** is ***executed*** instead. ***The expression*** must evaluate to a one-bit value, and can be either a variable, or any arithmetic, logical, and relational expression.

Example 8.2.1. An example of if-else statement is given below, where a is assigned either 0 or 1 depending on the value of the conditional expression.

```
if ( ! b & chipselect )
    a = 0;    /* any statement */
else
    a = 1;    /* any statement */
```

□

8.3 Switch Statement

The `switch` statement is a multi-way conditional that tests whether a given expression matches one of a number of constant cases, and branches accordingly. The individual cases in the switch statement may be cascaded, and are delimited through the use of `break` statements. The formal syntax of `switch` statement is given below, where the `default case` is optional:

switch ***(expression)*** {
case case-l:
    statement-l
case case-21
    statement-2
. . .
default:
    statement-D

The case value ***case-i*** must be a decimal, hexadecimal, or binary constant. `Break` statements are used to identify the end of a particular case. They are illegal in any other contexts, such as for premature exits from while loops. Cascading and fall through of the different cases are also supported.

Example 8.3.1. An example of the use of switch statement in selecting among a set of operations based on a given `codeopcode`.

```
switch ( opcode ) {
case 1:
    result = a + b ;
    break;
case 2:
    result = a - b;
    break;
case 3:
case 4:
```

```
            result = a & b;
            break;
        default:
            result = 0;
            break;
```

Note that for `opcode` equal to 3 and 4, the value of `result` is set to the logical-AND of the operands. cl

8.4 Looping Constructs

There are two types of iterative loop constructs in HardwareC – `for` loops and `while` loops. `For` loops have iterations whose bounds are known at compile time. A `while` loop has iterations whose exit condition can be data-dependent, and hence is in general unknown at compile time.

8.4.1 For Loop

A `for` loop is a constant bound iteration on a given integer variable. The formal syntax is:

for ***indexvar*** = ***value1*** **to** ***value2*** *[* **step** ***value3***] **do**
    statement

for ***indexvar*** = ***value1*** **downto** ***value2*** [**step** ***value3***] **do**
    statement

where *value1*, *value2*, and *value3* can be any constant or integer expression. The `step` clause is optional, and has a default of one. The variable ***indexvar*** must be an integer variable.

8.4.2 While Loop

The `while` loop is a data-dependent iteration on a given Boolean expression. The formal syntax of the ***while*** loop is as follows.

while (***expression***
    statement

where ***expression*** can be any variable or expression that evaluates to a scalar quantity. The loop body executes ***until expression*** evaluates to 0.

In addition, a variant of the while loop executes the body first prior to evaluating the loop exit condition. The formal syntax of the ***do-while*** loop is as follows.

do
    statement
while (***expression***);

where ***expression*** can be any variable or expression that evaluates to a scalar quantity. The loop body executes until ***expression*** evaluates to 0.

Finally, a third variant of the data-dependent loop is the ***repeat-until*** loop, where the loop body executes repeatedly until the expression evaluates to non-zero. ***The*** formal ***syntax*** of ***repeat-until*** loops is:

repeat
    statement
until (***expression***);

9 Template Models

Very often two descriptions differ in only very restricted ways. For example, they are the same with the sole exception that the variable sizes are different, as illustrated below for a four-bit and a five-bit adder.

```
/*
 *  Four-Bit adder
 */
procedure adder4(a, b, c, cin, cout)
    in boolean a[4], b[4], cin;
    out boolean c[4], cout;
{
    /* 4 bits add */


/*
 *  Five-Bit adder
 */
procedure adder5(a, b, c, cin, cout)
    in boolean a[5], b[5], cin;
    out boolean c[5], cout;

    /* same as above, but for 5 bits */
}
```

It is much simpler and expressive if only ***one*** description is given for the adder function which takes an argument specifying the size of the operation. This approach offers the advantages of (1) consistency of descriptions, (2) economy of code, which shortens design time, and (3) reusability of code (polymorphism).

In HardwareC, the mechanism which supports parameterized descriptions is a ***template.*** A template can be applied to any of the four types of models (block, process, procedure, or function). Templates are similar to generic packages in ADA, or generic classes in several object-oriented languages. A template takes one or more integer arguments as formal parameters. Given a particular mapping of integer values to the integer parameters, a corresponding ***instance*** of the template can be obtained. A good analogy can be made between templates and module generation, e.g. a template is a form of high-level module generation. The formal syntax of templates is given below for each model type.

- Block Template definition:

> **template block** ***name (parameters)***
>   **with** ***(int-parameters)***
>   ***parameter declarations***
> *body*

- Process Template definition:

> **template process** ***name (parameters)***
>   **with** ***(int-parameters)***
>   ***parameter declarations***
> *body*

- Procedure Template definition:

> **template [procedure]** ***name (parameters)***
>   **with** ***(int-parameters)***
>   ***parameter declarations***
> *body*

- Function Template definition:

> **template function** ***name (parameters)***
>   **with** ***(int-parameters)***
>   **return boolean**[*size*]
>   ***parameter declarations***
> *body*

The keyword `template` prefixes the name of the template, and the keyword `with` separates the Boolean-value parameters from the integer parameters. ***The int-parameters are the*** names of the integer parameters, and are separated by commas (,) if more than one is present. These integer parameters are scalar quantities, and can be used in both parameter declaration and the body of the template as integer constants. Specifically, assignment to an integer parameter is not allowed.

Example 9.1. Let us consider the description of a template for the ripple-carry adder procedure.

```
/*
 *     ripple carry adder
 */
template procedure adder(a, b, c, cin, cout) with (size)
    in boolean a[size], b[size], cin;
    in boolean c[size], cout;
{
    int i;
    boolean temp;

    temp = cin;
    for i = 0 to size-1 do {
        c[i:i] = a[i:i] ^ b[i:i] ^ temp;
        temp = a[i:i] & b[i:i] |
               temp & (a[i:i] | b[i:i]);
    }
    cout = temp;
}
```

The template model `adder` takes an integer parameter `size` corresponding to the size of the operands. The integer variable `i` is used as the loop index in the description of the ripple-addition logic. The sum is returned via `c` and the carry-out in `cout`. □

Templates can be used to describe library components such as adders and multipliers. In addition, a model can *call* a template, or explicitly declare an *instance* of the template. Calls to templates are described in Section 10, and explicit instantiation is described in Section 10.2. Templates must be declared or defined before it can be called. Declaration of template is analogous to declaration of models by prefixing the declare keyword before the header.

Example 9.2. We illustrate the declaration of templates by declaring the `adder` procedure template in the previous example.

```
declare template procedure adder(x, y, z, cin, cout) with (size)
    in boolean x[size], y[size], cin;
    in boolean z[size], cout;
```

Again, the actual names that are used as formal parameters in the declaration need not match the names used as formal declarations in the template definition. □

10 Calls and Instances of Models

A model may be ***called*** by another model, and indicates a request to execute the functionality of the called model. The call is implemented by a particular ***instance*** of the hardware corresponding to the invoked model. Therefore, from the standpoint of synthesis, a model can be treated as a *resource* that is allocated and shared among the model calls. In particular, a model that is called is referred to as a *resource.*

Example **10.1.** To illustrate the analogy between models and resources, consider the description of a model cascade that calls an ***udder*** model twice.

```
cascade(a, b, r, cout)
    in boolean a[8], b[8];
    out boolean r[8], cout;

    boolean  temp;

    adder(a[0:3], b[0:3], r[0:3], 0, temp);
    adder(a[4:7], b[4:7], r[4:7], temp, cout);
```

In the final hardware implementation of cascade, the two calls can be implemented by either (1) a single adder, where the two calls are bound to the single adder, or (2) two adders, where each of the two calls are bound to separate adders. Therefore, the adder can be treated as a ***resource*** that can be shared among the multiple model calls. □

There are two types of model calls - *unbound* or ***bound*** calls, as described next. A block model describes an interconnection of models, and hence does not differentiate between unbound and bound model calls. In particular, a call in a block model indicates the structural instantiation of the hardware corresponding to the called model (either a process or another block model), and may not be shared with the other calls in the model.

10.1 Unbound Model Calls

An *unbound call* is a call made to a given model, where the particular ***instance*** of the model used to implement the call is not specified, i.e. the call is not bound to an instance. The synthesis system has the freedom to implement multiple unbound calls by one or more hardware resources.

An unbound call involves specifying the name of the model to be called, along with the arguments to the model separated by commas and enclosed in parentheses. The invoked model must be ***declared*** or ***defined*** before it can be called, otherwise the call will result in an error. Template models can also be called by supplying in addition the values of the integer parameters in the template definition. The formal syntax of an unbound model and template call is:

modelname (***arguments***);
templatename (***arguments***) with (***int-arguments***);

where ***arguments*** is a list of arguments matching one-to-one to the formal parameters of the called model, separated by ***commas, and int-arguments*** can be a list of any constant or integer expressions, also separated by commas. *Block* models can call other block and process models.

Example 10.2. The following code segment illustrates the use of unbound calls. The first call is made to a procedure model `procA`, and the second and third calls are made to an adder template `adder` that takes an integer parameter corresponding to the size of the addition.

```
procA( a, b, c );                /*  model  call  */

adder( x, y,  z ) with (5);      /* template call */
adder( i,  j, k ) with (3);      /* template call */
```

The first template call is a 5-bit addition, and the second template call is a 3-bit addition. □

10.2 Bound Model Calls

In some cases, however, the designer may wish to invoke a specific instance of the called model in the final implementation. This is accomplished by the use of ***bound calls.*** To specify the particular instance of the model to invoke, an **instance** of the model is explicitly declared within the description, in a similar manner as declaring a variable. The bound call then invokes the declared ***instance*** of the model, i.e. binds the call to the instance.

There are several important advantages in explicit instantiation of models. First, the designer can access not only behavior through model calls, but ***also internal state information as well. This*** is analogous to the capabilities of abstract data type languages in software. Second, because a model instantiation is similar to instantiating a hardware module, ***resource sharing*** is supported at the description level. For example, two calls to the same instance will share the instance in the final implementation. Finally, the designer can completely specify the behavior that is intended without relying on hidden assumptions. We describe first the explicit instantiation of models, followed by the use of instances in model calls.

Instance Declaration An **instance** of a model (or template) represents an ***object*** that encapsulates both behavior and state. In a similar manner, a Boolean variable is also an ***object*** whose behavior is specified by the language in terms of the semantics of accessing and modifying the variable. Instances can therefore be treated as ***instance variables*** that are declared and used in the scope of its definition. The syntax of model (or template) instantiation is described below.

instance *modelname* ***instvar*** { , ***instvar*** }∗;
instance ***templatename*** **with** ***(int-args)*** ***instvar*** { , ***instvar*** }∗;

The keyword **instance** prefixes the name of the model or template, followed by the names of the instances to be created, separated by commas. The arguments to the integer parameters in the template instantiation can be any constant or integer expression, and are separated by commas if there are more than one. The instance variables ***instvar*** can be scalar or vector, where a vector of instances denotes a set of instantiations of a given model.

Instance of models or templates can be declared in any compound statement, and the scoping rules for variable visibility also apply to instances. The model or template that is used in the instance definition must be previously declared or defined.

Example 10.2.1. Consider the example below, where `adder` is an addition template, and `counter` is a procedure that increments an internal variable each time it is called.

```
{
    instance  counter           a;      /* 4 bit counter */
    instance adder with (4)     04;     /* 4 bit adder   */

    {
        instance counter a, b;

        a  (...);     /* new counter */
        04 (...);
    }
    a (...);          /* old counter */

    /* b is undefined here */
```

The instance `a` of model `counter` is different for each different nesting of the compound statements.
□

Calling an Instance With the instances explicitly declared, we can invoke a specific instance of a given model by a ***bound model cuff. The*** formal syntax of a bound model call is:

instancename (***arguments***);

where ***arguments*** is a list of arguments matching one-to-one to the formal parameters of the called model, separated by commas, and ***instancename*** is the name of the declared instance. If the instance is a vector quantity, then ***instance-name*** should specify a single element of the vector that is invoked. For example, consider the example below with `add_array` declared as an instance of a given model `adder`.

```
/*  instance  declaration  */
instance  adder         add_array[6];

add_array[3]( . . . );        /* call instance index [3] */
add_array[0]( . . . );        /* call instance index [0] */
add_array[5]( . . . );        /* call instance index [5] */
```

Example 10.3. We illustrate the use of bound calls in the following code segment. Let `procA` and `adder` be the model and template that is used in the previous example. We now declare an instance for each of the models, and invoke the instances.

```
instance  procA                inst_of_A;
instance  adder with (3)       inst_of_B_size3;
instance  adder with (5)       inst_of_B_size5;

inst_of_A( a, b, c );             /* bound call */

inst_of_B_size3( x, y, z );       /* bound call */
inst_of_B_size5( i, j, k );       /* bound call */
```

Note that the name of the instance to be invoked is used in the bound model call. □

Example 10.4. Resource Sharing. We illustrate now the use of bound calls and explicit instantiation to perform resource sharing at the description level. The following counter example increments or decrements the value a according to the value of upload, and contains three generic calls to two S-bit adder templates. Initially, a is assigned to the sum of two variables x and y.

```
/*  initially load */
a = adder( x, y ) with (5);

/*  loop and count */
while ( ! reset )
{
    if ( upload )
        a = adder( a, 1 ) with (5);
    else
        a = adder( a, -1 ) with (5);
}
```

In the most simplistic case, each of the three calls can be implemented by separated hardware resources. However, suppose we would like to use only ***a single*** adder resource in implementing the description. To do so, we explicitly declare **an** instance of the adder template, call it inst_adder, and make all the calls to that instance. In particular,

```
instance adder with (5)   inst_adder;     /* declare */

/*  initially load */
a = inst_adder( x, y );

/*  loop and count */
while ( ! reset )
{
    if ( upload )
        a = inst_adder( a, 1 );
    else
        a = inst_adder( a, -1 );
```

All calls are instantiated, and refer to the same instance. Therefore, we have achieved resource sharing at the description level. ❑

Motivation for Instances A major drawback with many languages is the inability to specify exactly which instance of a given model is invoked in a model call. This restriction is reasonable for models that describe only the functionality without internal state information. However, if a model has internal state associated with it (through the use of static variables), such restriction severly handicaps the usability and expressiveness of the language. In fact, the such deficiency can result in either inefficient or even incorrect implementation, depending on whether the assumptions made by the synthesis system matches those made by the designer.

Example 10.2.2. Consider the description of a counter below.

```
/*
 *    each call to it increments by 1
 */
counter(value)
    out boolean value[8];

    static state[8];

    state = state + 1;
    value = state;
```

Every call to the counter module will increment the corresponding internal state variable by one. If a call is made to `counter` without specifying the particular instance that is to be invoked, then one of two situations will arise.

1. ***Single instance assumption*** – If the synthesis system assumes that one and only one instance is associated with a procedure, then a call to `counter` will always increment the same internal state (corresponding to the single instance).

   However, this approach is overly restrictive since one of the powers of synthesis systems is to explore the spectrum of design tradeoffs between parallel and serial implementations, and by always assuming one instance per procedure this exploration is not possible.

2. ***No assumption on the invoked instance*** – On the other hand, if no assumptions are made on which instance a given call will invoke, the synthesis system will then have the flexibility to either dedicate an instance to the call, or share several procedure calls onto the same instance. However, if the procedure has internal state information, then the description can be incorrect depending on the particular mapping of procedure calls to procedure instances.

The assumptions that are made by the synthesis system may not be what the designer had in mind when writing the description. For instance, in the code segment below, `counter` is called twice.

```
        counter( sum1 );

        counter( Sum;! );
```

The designer can either view the two calls as incrementing the same value twice. Alternatively, the designer can view the two calls to be distinct, each incrementing a value independent of the other. Through instantiation of procedures, the designer can explicitly specify the exact semantics of a procedure call. For example, if the designer wishes to increment a single value twice, then the corresponding code is given below.

```
        {
            instance counter     value;

            value(...);     /* increment */
            value(...);     /* increment again */
```

On the other hand, if the designer wishes to increment two different values, then the code is as follows.

```
        {
            instance counter value1, value2;

            value1(...)     /* increment value1 */
            value2(...)     /* increment value2 */
            value1(...)     /* increment value1 again */
```

□

10.3 Arguments to Model Call

We describe now the arguments to an unbound or bound model call. Valid call arguments depend on the particular class of the corresponding parameter in the model definition. Specifically, if a parameter of the called model is of type:

- *Local port:* For `in` local ports, valid argument includes constant, integer expression, `in port` (`local` or global), `boolean` or *static* variable and expression. For `out` local ports, valid argument includes `out` port (local or global) `boolean` and `static` variable. No expression is allowed.
- *Global port:* There must be one-to-one type correspondence between the parameter type and the supplied argument in the case of global ports. For example, if the type is `in` global port, then only an `in global port` can be used as argument. Likewise for `out` and `inout global ports`.
- *Channels:* Similar to global ports, there must be an one-to-one type correspondence between the argument and channel parameters. In particular, only `in` channels can be arguments to an `in` channel parameter. Likewise for `out` channels.

11 Constraints

There are two categories of design constraints that are supported by HardwareC – ***timing*** and ***resource*** constraints. Timing constraints associate ***tags*** with statements, and define upper and lower bounds on the time separation between the tags. Recall that we refer to models that are called as resources that can be allocated and shared among the calls. Resource constraints specify the number of resource components available, and partially bind model calls to a specific instances of the resource pool.

11.1 Timing Constraints

There are three forms of timing constraints, which define the ***minimum*** or ***maximum*** time between two statements, and the ***delay*** of a particular statement. To identify which statements the constraint refers to, ***fags*** are associated with statements.

Tags can be associated with any statement in the language by prefixing the statement with the tag name, followed by a colon (:) as delimiter. Tags can be scalar or vector, and must be explicitly declared in a totally analogous fashion as Boolean variables. Tags are valid only in the scope of their definition. The formal syntax of tag definition is:

tag *name* { , *name* }*;

where the keyword **tag** indicates tag definition, and *name* is the name of the tag. Tags can be scalars or vectors.

With the tags defined, the timing constraints are defined as:

constraint mintime from *tag-src* **to** *tag-dst* = *num* {**cycles**|**units**};

constraint maxtime from *tag-src* **to** *tag-dst* = *nwn* {**cycles**|**units**};

constraint delay of *tag-src* = *num* {**cycles**|**units**};

Note that ***tag-src*** and ***tag-dst*** are single-bit tag quantities, and ***num*** is a positive constant or integer expression. The unit of the timing constraint can be in the number of cycles (**cycles**), or arbitrary units whose interpretation is up to the designer (**units**). Constraint statements can appear anywhere in the description, as long as the tags that are referenced are defined.

Example 11.1.1. An example of tag declaration and use is given below.

```
tag         labell, label2, labelvec[3];

labelvec[0]:        send( channelA, msg );
labelvec[1]:        write port = value;
labell: label2:     y = read(x);
```

There are three tags in the example: `label1` and `label2` are scalar tags, and `labelvec` is a 3-element tag vector. Note that multiple tags can be associated to a given statement by putting multiple sets of tags followed by colon before the statement, as in the case of "`label1 : label2 :`". □

11.2 Resource Constraints

Resource constraints specify the *number of* available resource in the final implementation. The ***resource utilization constraint*** describes the number of instances of a given model. The formal syntax of resource utilization constraints is:

> **constraint resource_usage** *modelname num*;
> **constraint resource_usage** *templatename* ***with*** *(int-arguments) num*;

where *num* is the number of instances of the corresponding model or template. The constraint can appear anywhere in the description. The partial binding of operations to resource components is already made possible through bound calls to instances of a given model.

Example 11.2.1. We illustrate the use of explicit instantiation as resource constraints. Let `modelA` be an arbitrary model that needs to be called 5 times. If the calls are all unbound, then the synthesis system is free to implement the 5 calls with up to five resources corresponding to the hardware implementing model `modelA`. Suppose for high-level considerations only one instance of the model is feasible in the final implementation. This constraint on the resource utilization is achieved by explicit instantiation of the model, followed by bound calls to the instance. Specifically,

```
instance modelA    inst_modelA;     /* declare instance */

inst_modelA(...);        /* same instance */
inst_modelA(...);        /* same instance */
inst_modelA(...);        /* same instance */
inst_modelA(...);        /* same instance */
inst_modelA(...);        /* same instance */
```

Bound model calls allows resource sharing at the description level. The designer can specify the particular instance to which one or more operations are to be bound • I

Another form of resource constraints is the binding of calls to specific instances of the called model. This is supported through bound model calls, as described in Section 10.2.

12 Attributes

The designer can also embed arbitrary attributes by encapsulating the information in double quotes. The information is described as arbitrary strings. The formal syntax of an attribute is:

> **attribute** *"arbitrary-string"*;

This capability allows the high-level design description to capture information that are not immediately used by the behavioral synthesis tools, but pass the detailed constraint information to later synthesis steps that can fully utilize them.

Example 12.0.2. The following are examples of attributes:

```
attribute "comment = this is a comment";
attribute "constraint power = 25";
attribute "this is very interesting";
```

□

13 Miscellaneous

HardwareC relies on the C preprocessor during parsing to handle macro definition (#define) and file inclusion (#include) facilities. The designer is free to use any C preprocessor commands in the description.

14 Acknowledgments

The authors would like to thank Larry Augustin and Thomas Truong for many helpful comments. This research was sponsored by NSF-ARPA, under grant No. MIP-8719546, by AT&T and DEC jointly with NSF, under a PYI Award program, and by a fellowship provided by Philips/Signetics.

References

[1] B. Kernighan, D. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ, 1978.

[2] G. De Micheli, David C. Ku, *HERCULES - A System for High-Level Synthesis*, Proceedings of the 25th Design Automation Conference, Anaheim, CA, June 1988.

[3] David C. Ku, G. De Micheli, *High-level Synthesis and Optimization Strategies in Hercules/Hebe*, Proceedings of the EuroASIC Conference, Paris, France, May 1990.

Appendix

Four detailed examples of hardware description using HardwareC are described below. The first is a four bit carry look-ahead adder. The second is a counter process that uses the four bit adder. The third is the traffic light controller described in the Mead-Conway book. The fourth is the greatest common divisor example.

The examples illustrate the various modeling aspects of HardwareC, and does not necessarily represent complete designs. For example, the adder model needs to be invoked by other models as part of a larger design.

Four-Bit Adder

```
procedure add4bit( a, b, carryin, result, carryout )
    in boolean  a[4];
    in boolean b[4];
    in boolean  carryin;
    out boolean result[4];
    out boolean carryout;
{
    int     i;
    boolean P[4], G[4], new;

    /*
     *   calculate propagate and generate
     */
    for i = 0 to 3 do
        P[i:i]  = a[i:i] xor b[i:i];

    for i = 0 to 3 do
        G[i:i]  = a[i:i] & b[i:i];

    /*
     *    calculate carryout
     */
    carryout = G[3:3] | (P[3:3] & G[2:2])
        | (P[3:3] & P[2:2] & G[1:1])
        | (P[3:3] & P[2:2] & P[1:1] & G[0:0])
        | (P[3:3] & P[2:2] & P[1:1] & P[0:0] & carryin);

    /*
     *    calculate sum
     */
    new = carryin;

    for i = 0 to 3 do {
        result[i:i] = P[i:i] xor new;
        new = G[i:i] | ( P[i:i] & new );
```

Counter

```
process counter( run, loadflag, updown, data, sum )
    in port  run, loadflag, updown, data[4];
    out port sum[4];
{
    boolean temp[5];
    while ( run ) {
        if ( loadflag )
            temp = data;
        else {
            if ( updown )
                add4bit(temp, 1, 0, temp[0:3], temp[4:4]);
            else
                add4bit(temp, 0xf, 0, temp[0:3], temp[4:4]);
        }
        sum = temp[0:3];
```

Traffic controller

```
/*
 *    Mead/Conway Traffic Light Controller
 */

# define HIWAY_GREEN 0
# define HIWAY_YELLOW 1
# define FAR-GREEN 2
# define FAR&-YELLOW 3

# define GREEN 1
# define YELLOW 2
# define RED 3

# define TRUE 1
# define FALSE 0

process traffic ( run, Cars, TimeoutL, Timeouts,
               HiWayL, FarmL, StartTimer )
    in port run;
    in port Cars, TimeoutL, Timeouts;
    out port HiWayL[2], FarmL[2], StartTimer;
{
    static state[2];
    boolean newstate[2];

    while ( run )

      /* combinational logic to determine nextstate */
        switch (state) {
        case HIWAY_GREEN:
          HiWayL = GREEN;
          FarmL = RED;

          if (Cars & TimeoutL) {
             newstate = HIWAY_YELLOW;
             StartTimer = TRUE;
          } else {
             newstate = HIWAY_GREEN;
             StartTimer = FALSE;
          }
          break;

     case HIWAY_YELLOW:
          HiWayL = YELLOW;
          FarmL = RED;

          if ( Timeouts ) {
             newstate = FARM-GREEN;
             StartTimer = TRUE;
          } else {
             newstate   ≠ARM_YELLOW;
             StartTimer = FALSE;
          }
          break;
     case FARM-GREEN:
          HiWayL = RED;
          FarmL = GREEN;

          if ( ! Cars | TimeoutL ) {
             newstate = FARM_YELLOW;
             StartTimer = TRUE;
          } else {
             newstate = FARM-GREEN;
             StartTimer = FALSE;
          }
          break;
     case FARM_YELLOW:
          HiWayL = RED;
          FarmL = YELLOW;

          if ( Timeouts ) {
             newstate = HIWAY_GREEN;
             StartTimer = TRUE;
          } else {
             newstate = FARM_YELLOW;
             StartTimer = FALSE;
```

```
            }
            break;
        }
        state = newstate;
```

GCD The GCD (Greatest Common Divisor) example is derived from an VHDL description in the High-level Synthesis Workshop benchmark suite. The routine calculates the greatest common divisor of two positive numbers.

```
#define SIZE 8        /* size of number */
process gcd (xi, yi, rst, ou)
    in  port xi[SIZE], yi[SIZE];      /* input numbers */
    in  port rst;                     /* restart input */
    out port ou[SIZE];                /* result output */
{
    /* sampled input values and temporary variable */
    boolean x[SIZE], y[SIZE];

    /* set output to zero during computation */
    write ou = 0;

    /* sample input numbers */
    if ( rst )
    <
        x = read(xi);
        Y = read(yi);
    >

    /* gcd just for positive numbers defined */
    if ((x != 0) & (y != 0))

        /* using euclid's gcd algorithm */
        repeat
        {
            /* let x become the remainder of x divided by y */
            while (x >= y)
                x = x - y;

            /* x should be less then y now, so exchange x and y */
            <
                x = y;
                Y = x;
            >
        }
        /* end if y (= former remainder x) is zero */
        until (y == 0);

/* if any number is zero, output shall become zero too */
else
        x = 0;

/* write result (gcd or zero) to output */
write ou = x;
```