Optimal Constructs for Chip Level Modeling

by

Dongil Han

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

APPROVED:

J. R. Armstrong, Chairman

J. G. Tront

D. W. Luse

August 1986

Blacksburg, Virginia

Optimal Constructs for Chip Level Modeling

by

Dongil Han

J. R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

Anaysis and comparison of nine different Hardware Description Languages is presented. Comparison features are discussed and each language is analysed according to the comparison features, which are: sequencing mechanisms, applicability to generic structures, abstraction of data and operation, timing mode, communication mechanisms, and instantiation and interconnection of elements. Based on the analysis of the languages, optimal constructs for chip level modeling are extracted. Example descriptions of a microprocessor system MARK 2 are presented.

ACKNOWLEDGEMENTS

I sincerely wish to thank Dr. James R. Armstrong for his continuous suggestions and patience during the preparation of this thesis. I would also like to thank Dr. J.G. Tront and Dr. D.W. Luse for serving my committee. I would like to express my deepest gratitude to my parents, wife and daughter, whose continuous support and encouragement made the completion of this thesis possible.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

CHAPTER 1. INTRODUCTION

Six major structural levels exist in the representation of digital systems. They are:

PMS (Processor Memory Switch), Chip, Register, Gate, Circuit, Silicon.

Starting at the highest level presently considered, the PMS level, the hierarchy extends downward through chip, register, gate ,circuit, and silicon. It is apparent that the level of detail increases as the description level moves to lower level.

At present time the position of the window depends on the type of design activity being carried out. The designer of VLSI devices may model at levels ranging from the silicon level up through the chip level. The mainframe computer manufacturer, on the other hand, will have a different window, which is the main focus of this thesis. For as the complexity of VLSI devices increases, it will be impractical to include the gate level in the window, as a single chip will contain hundreds of thousands of gates. The next level up is the register level which models a chip as a set of registers, multiplexors, and data paths. While a certainly less complex

than the gate level, this representation level may also give unnecessary detail for those interested in only true input/output response of a VLSI device. It is likely then that those concerned with the design of computer systems, whether they be large mainframes or personal computers will consider the chip level to be lowest level in their design window.

Hardware description languages (HDLs) have been employed in the systematic design, modeling, verification and evaluation of the digital systems. The increasing complexity of digital circuits demands good hardware description languages. Recent efforts of government and industry to standardize hardware description languages represents the growing importance of these languages [23].

For the past eight years, chip level modeling and simulation techniques have been developed at Virginia Tech under Dr.J.R. Armstrong [1,2,3,12]. A chip level modeling and simulation system known as GSP was developed and extensively used. As the language was employed in the various projects, its structure evolved, i.e., as new modeling situation were encountered, the modeling constructs were augmented to handle these situations.

It is the purpose of this thesis to investigate the ideal set of constructs for chip level modeling. To do this, generic VLSI logic structures are modeled to determine the effectiveness of potential modeling language constructs. Modeling constructs from different hardware description languages (HDL) are then compared in terms of modeling effectiveness. The languages studied include: 1)AHPL-developed at the University of Arizona, 2)ISPS-developed at CMU, 3)ISP'-the language used in N.2 simulation system, 4)VHDL-the VHSIC hardware description language, 5)GSP, 6)GSP2-the high level version of GSP, 7)TI's HDL-the language used in the TILAD simulation system developed at TI, 8)SLIDE-I/O description language, 9)HHDL-The input language for HELIX simulation system developed at Silva-Lisco. These languages are chosen because they cover various developmental background (e.g., academia, industry, and government), also these simulators (except HHDL and SLIDE) are currently up and running at Va. Tech. The VHDL simulator is now under development. It will be available in August, 1986 at Virginia Tech.. The VHDL analyser and the design library manager are running at Va.Tech. Other languages[21,22] are referenced whenever it is found that they have effective constructs for chip level modeling.

CHAPTER 2. THE LANGUAGES

The languages studied are these nine languages: GSP, GSP2, ISPS, ISP', AHPL, VHDL, SLIDE, TI's HDL and HHDL. Here we give a short summary of their characteristics.

GSP[13] and GSP2[18,43] were developed under Dr.Armstrong at Va.Tech. GSP is an assembler type language with special constructs for detailed timing description. GSP2 is the improved version of GSP. It is a pascal-like block structured language, and has the same level of detailed timing description capability as GSP.

ISPS[10,24,25] was developed at CMU, and is derived from the well known ISP notation. ISP was developed to be used on instruction set evaluation. ISPS added many constructs such as process and delay constructs. So it may be used to describe general digital systems.

ISP'[5,14,15,16] also has its origin in ISP but it has more constructs for synchronization, timing and communication. ISP' is the modeling language for the N.2 simulation system which is widely used in industry.

AHPL[7] was developed at the University of Arizona. It has been widely used as a pedagogical tool.

VHDL[6,13,31,32,33] is the language developed by the VHSIC program of Department of Defense. VHDL has been developed as a government standard HDL and expected to be a IEEE standard.

SLIDE[11] is the HDL which specializes in I/O description. The motivation of the development of this language is to describe the behavior of I/O hardware at the high level.

TI's HDL[4,40] is the modeling language for the TILADS simulation system developed at Texas Instrument.

Finally, HHDL[19,42] is the hardware description language for the HELIX simulation system. HHDL is a commercial language and widely used in industry.

All of these languages can describe digital circuits at or above register transfer level. Old but popular languages such as CDL and DDL are worth comparing, but are not included in this thesis because they had already been analysed and compared in excellent papers [9,26].

CHAPTER 3. COMPARISON OF THE LANGUAGES

3.1 COMPARISON FEATURES

Six comparison features were identified as the languages were studied.

1. Sequencing Mechanisms

   A sequencing mechanism refers to the order of execution of a description. With the sequencing mechanism, we can classify HDL's into three categories[8]:

   - In a procedural language, textual structure is important as in a typical programming languages. Statements are executed sequentially.

   - In strongly nonprocedural language, every action has a specific condition to be met and the action is executed only when the condition is satisfied. If conditions of several actions are satisfied at the same simulation time, the actions execute in parallel.

- The block[1] oriented nonprocedural description is in the continuum between the previous two categories. Each block has a condition to be satisfied, but once the condition is satisfied, statements in the block execute in sequence. So nonprocedurality exists at the block level.

2. Applicability to Generic Modeling Structures

Generic modeling structures refer to the basic classes of structure required to implement chip level modeling [1]. The generalized delay model is mainly concerned with propagation delay and module suspension constructs. The minimum energy level model is concerned with constructs which are related to the input timing such as set_up, hold time and minimum pulse width. These models are discussed in greater detail in Chapter 3.3.

3. Abstraction of Data and Operation

The abstraction of data and operations is concerned with data typing, operators, functions, and procedures.

---

[1] Here, the word block does not mean the 'block' statement of VHDL.

4. Timing Mode

The timing mode refers to the ability to describe asynchronous or synchronous circuits or systems. Some HDL's have ability to describe both kind of systems. On the other hand, some languages can model only one of them.

5. Communication Mechanisms

Communication mechanisms refer to the communication methods between hardware modules or within a hardware module such as ports, signals and global variables.

6. Instantiation and Interconnection of Elements

Instantiation refers to the creation of elements within the description. Interconnection refers to the connection of elements which are created by the instantiation statements through wires. Instantiation and interconnection represent the structure of an entity.

3.2 SEQUENCING MECHANISMS

This section describes and compares the mechanisms by which one activity can influence another in a digital system and

the mechanisms by which these activities may be grouped. Such activities include data and control operations.

The HDL's can be classified into procedural and nonprocedural languages by sequencing mechanism [9]. In a procedural language, the activities have an explicit order of execution which is conditioned by the completion of the preceding activities and concurrent activities are grouped in 'time blocks'. The description is then a list of these blocks [9]. Figure 1(a) shows two simple examples of procedural description. The first example (a1) has two time blocks each of which is separated by 'next' operator. In this case, the statement X:= Y is executed first then the statement Y := X is executed. Consequently X has the same value as Y after execution of these two statements. On the other hand, the second example (a2) is composed of two statements within a single time block. Therefore two statements are executed concurrently. The result of the execution is that X and Y swap values.

Nonprocedural languages give no meaning to the textual ordering of the activities. Each activity is associated with a 'guard' describing the condition of execution. If the values of guards evaluate to true, then the activities which are associated with guards execute simultaneously. Nonprocedural language may further be classified into strongly nonproce-

dural and block oriented nonprocedural language [8]. In strongly nonprocedural languages, if at any stage of execution of a program a condition is satisfied, then corresponding actions execute concurrently, otherwise they are not invoked. Figure 1(b) represents the general form of a strongly nonprocedural description. In block oriented languages, nonprocedurality exists at the block level as can be seen in figure 1(c). Each block has a guard condition which is a boolean expression. When the guard conditions evaluate to true, the blocks begin execution in a procedural fashion.

ISP',SLIDE and GSP2 are considered as block oriented nonprocedural languages. VHDL has a characteristic of both types of nonprocedural languages. The process mechanism of VHDL has a characteric of block oriented nonprocedurality. Concurrent signal assignment statements in VHDL, on the other hand, have a characteristic of strongly nonprocedural languages. The 'process' construct is used in VHDL,ISP' and SLIDE to represent the block. In GSP2, the 'event block' is used instead of the 'process'. The semantics of 'process' and 'event block' are similar in that these constructs represent independent executing environments, which may be a description of a piece of hardware. But there are a few differences between these nonprocedural languages in describing guard conditions which we now describe:

```
    .                                    .
    .                                    .
X := Y; ---  time block 1        X := Y; ___time block 1
next;                            Y := X;
Y := X; ---  time block 2        next;
    .                                    .
    .                                    .

  (a1)                                (a2)
```

(a) Procedural Descriptions

```
condition 1: concurrent actions;
condition 2: concurrent actions;
      .
      .
condition N: concurrent actions;
```

(b) Strongly Nonprocedural Description

```
condition 1:
 ____________________
| A := B and C;      |
| next;              |
| Q := A;            |
| NQ := not A;       |
| next;              |
|____________________|

condition 2:
 ____________________
| " procedural       |
|     description "  |
|____________________|

condition 3:
        .
        .
```

(c) Block Oriented Nonprocedural Description

Figure 1. Sequencing Mechanisms

- Figure 2 represents the structure of a process of VHDL. A process has a sensitivity list which specifies a set of signals to which it is initially sensitive. When any signal changes value, statements within the process execute sequentially.

  The sensitivity list may be changed inside of the process by disabling or enabling specific signals which are in the current sensitivity signal set.

- In ISP', there are two kinds of processes; 'main' and 'when'. There is no guard condition for a main process. The 'main' process is a section of source code which executes as an endless loop. The 'when' process requires a guard which is composed of a port condition and optional state (global) conditions. The port condition can have a modifier which sensitizes the wait for either a leading edge, trailing edge, or any changes (if there is no modifier). Once the condition occurs, statements of the process begin execution. When the end of the 'when' process source text is reached, control returns to the original wait condition. Figure 3 depicts the structure of ISP' description.

```
process (X,Y)
 begin
  if not X'stable and X='1' then
    ----
  elsif Y='0'  then
    ----
  end if;
 end process;
```

Figure 2. Structure of a Process of VHDL

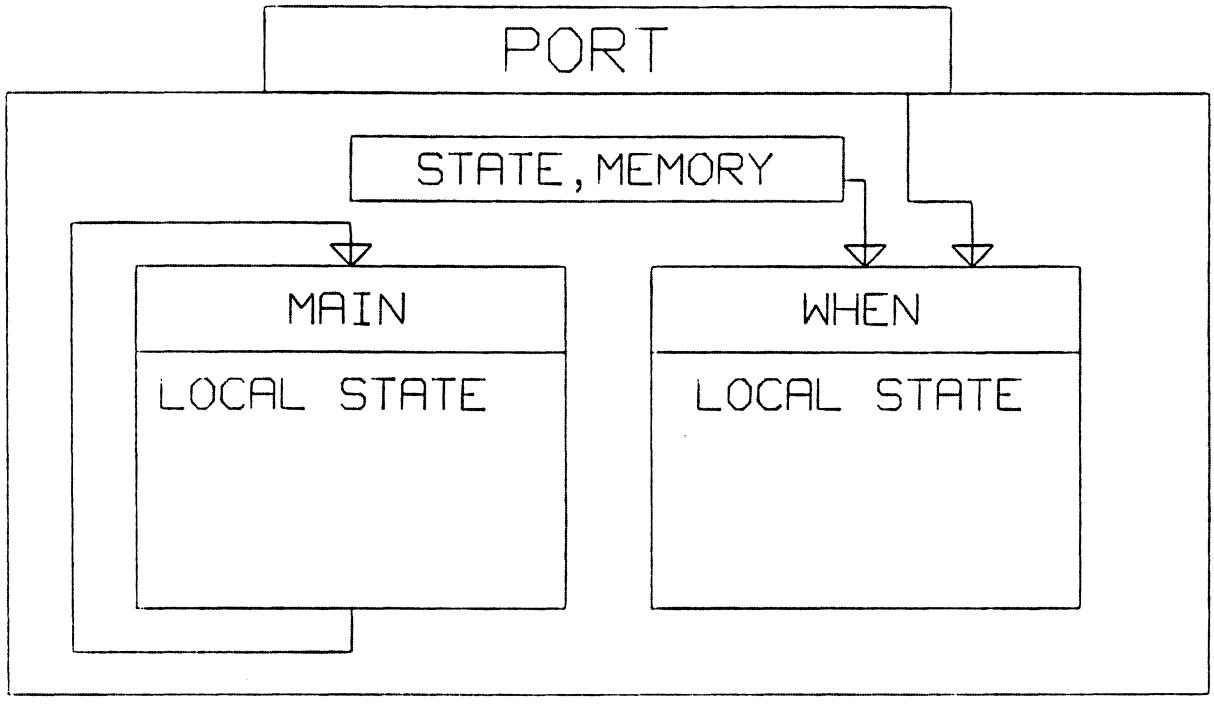
In this example, the process will start execution if the value of the port 'p' changes from 1 to 0 and at the same time variable x is not equal to 0.

- In SLIDE, each process has a guard condition to start the process. A process can have subprocesses. Each process also has a priority and can execute when:

  1. the process is a subprocess of one which is executing, and

  2. the guard condition is true, and

  3. no process at the same subprocess level with a higher priority is executing.




```
state  x<12>;
port   p, bus<20>;

(when   (p:trail(x neq 0)
     ----
     ----)
```

Figure 3. Structure of ISP' Description

An example of a SLIDE description is given below:

```
main process
 init A: 0  when x equ\
 init B: 1  when y equ/
  ----------
  process A; begin ---------- end
  process B; begin ---------- end
```

In the above example, the description has two processes: process A and process B which have priority 0 and 1 respectively. When x makes a negative transition (x equ\), the process A starts execution. While the process A is executing, the process B can not start execution even if y signal makes a positive transition (y equ/) because the priority of the process A is higher than that of the process B. This priority mechanism can be used to time order the execution of processes.

- In GSP2, an event block defines events and the model's reaction to these events. Consider the example below:

```
EVENT event_name  ON  event_condition
  BEGIN
   ------
   ------
  END
```

Here an event condition may include PIN variables, global registers and memories. The event condition expression must evaluate to a one bit so that it is either true (contains '1')or false (contains '0'). There are special functions (RISE,FALL and CHANGE) that may only appear in a event event expression.

- In HHDL, an UPON statement evaluates its boolean expression each time a net in the CHECKLIST is updated and the statement is executed whenever the boolean expression is true. For example consider an UPON subprocess as below:

```
UPON not a CHECK a,b  DO
     ASSIGN not a TO a  DELAY 10;
```

Here, the signal a and b are in a checklist and a boolean expression of the subprocess is 'not a'. The expression is evaluated when either the value of a or b changes. If the boolean expression " not a " evaluates to true, then the value of a is inverted and scheduled to output after 10 unit times. The statements within an UPON subprocess can be arbitrary complex. Unlike the UPON subprocess, a TRANSMIT subprocess may consist of only one assignment statement. An example of TRANSMIT subprocess is given below:

TRANSMIT not a TO a CHECK a,b DELAY 10;

In the above example, singal a is inverted and scheduled to output after 10 unit times when either value of signal a or b changes.

From the characteristics of sequencing mechanism of block oriented nonprocedural languages described above, three comparison points are extracted as follows:

1. Invocation of a block and use of signal change detection functions within a block

   A process of VHDL has a sensitivity list which is a set of signals. That is, the process is invoked whenever any signals in a sensitivity list have been changed. Specific conditions for data transform may be written inside the process with statements such as IF_THEN_ELSE and signal attributes such as x'CHANGE and x'QUIET, where x is a signal name which belongs to a sensitivity list of a corresponding process. Other languages (ISP',SLIDE and GSP2) have event condition expression parts separated from the executable body of the description. An event that invokes a process may be a rising transition, falling transition or any changes of a signal. But these

functions that detect signal transitions can not be used within a executable body.

Another problem is that for some languages, only one signal transition function can be used within an event condition expression under the assumption that the signal transitions never occur simultaneously in the real digital hardware. Let's take a level sensitive D type latch an example, and describe it with GSP2 and VDHL.

In Figure 4(a), a GSP2 description contains two event blocks. An output 'q' follows the value of an input 'd' while a 'clk' is in high state. After the 'clk' makes 1 to 0 transition, the value change of 'd' does not affect the value of 'q'. So the value changes of the two inputs 'clk' and 'd' should be events. Since GSP2 does not permit use of more than one signal transition function within an event expression, an event block must be declared for each event. Therefore at least two event block should be used to describe a D type latch. There is a way in GSP2 to describe the same behavior within a single event block using @CONCAT function as can be seen in Figure 4(b). Concatenated two single bit inputs are treated as a two bit vector and any change of two inputs can trigger the event block. But since transition detection functions can not be used within executable part

```
EVENT latch ON @CHANGE (D);
 IF CLK = #B1  THEN
  BEGIN
   q := D  PROP 10;
   nq := @NOT(D)  PROP 10;
  END

EVENT clk_chng ON @CHANGE (clk);
 IF clk = #B1  THEN
  ARM latch;
 ELSE
  DISARM latch;
 ENDIF
```

(a) Description with Two Event Blocks

```
EVENT sig_chng ON @CHANGE(@CONCAT(d,clk));
 BEGIN
  IF d <> d_old THEN
   IF flag = #B1 THEN
    BEGIN
     q := d PROP 10;
     nq := @NOT(d) PROP 10;
     d_old := d
    END
   ENDIF;
  ELSE
   IF clk <> clk_old THEN
    BEGIN
     IF clk = #B1 THEN
      flag := #B1;
     ELSE
      flag := #B0;
     ENDIF;
     clk_old := clk
    END
   ENDIF
  ENDIF
 END
```

(b) Description with One Event Block

Figure 4. GSP2 Descriptions of a D Type Latch

```
process (D,CLK)
 begin
  if not D'stable  and  CLK = '1'  then
   Q <= D after 10 ns;
   NQ <= not D after 10 ns;
  end if;

  if not CLK'stable  then
   if CLK = '1'  then
    enable D;
   else
    disable D;
   end if;
  end if;
 end process;
```

Figure 5. VHDL Description of a Level Sensitive D type latch

of an event block, a modeler has to declare extra variables to retain the old values of the signals that are used in an event condition expression to identify the source of the event block call. Also event control statements ARM and DISARM can not be used in a single event block description since the disarming of its own event block disables the invocation of that event block permanently. Therefore a variable named flag is used to check whether the event block has to respond to a new input value or not. Also, the single block description is more complex and harder to read than the two event block description in GSP2.

In a VHDL description in the figure 5, signals D and CLK are included in a check list of a process. Initially any signal changes of D or CLK trigger the execution of the process. After that, the sensitivity list of the process is controlled by the transition of a signal CLK. If the CLK signal makes a 1 to 0 transition, a statement 'disable D' is executed, so the process becomes insensitive to the transition of a signal D. If the CLK signal makes a 0 to 1 transition, a statement 'enable D' is executed and the process can be invoked whenever the signal D changes value. In this description the whole behavior of a D type latch is described within a single process. The approach of VHDL gives more flexibility to a programmer

in writing the description of an entity than those of other languages.

2. Components of guard condition

The process of VHDL and the event block of GSP2 are sensitive not only to the value change of the port (pin in GSP2) but also to the value changes of the internal states which are global to the module. This means that one process can explicitly invoke the other processes which are contained in the same description module. This feature is very useful to describe one of three generic models, Delay with Decision Point mentioned in Chapter 3.3.1.2. In ISP', on the other hand, the detection of a change of values is limited to the port which is to communicate with other modules. So it is impossible that one process can wake up other processes directly.

3. Guard condition control capability

VHDL has 'enable' and 'disable' constructs to control the sensitivity list of a process. The process is initially sensitive to all the signals in a sensitivity list. But the process that executes the disable statement will become insensitive to each of the signals named in the statement. The enable statement reverses the action of

the disable statement. GSP2 has 'ARM' and 'DISARM' constructs to control the sensitivity of an event block. DISARM is to ignore all occurances of the event specified by event name. ARM reverses the action of DISARM. The INHIBIT and PERMIT statements of HHDL have he same semantics of the DISARM and ARM statements of GSP2.

These event control constructs simplify the description of hardware. If a language does not have these constructs, a modeler has to declare flag variables to check whether a particular signal or event block is enabled or not. Further, these constructs increase the simulation efficiency by reducing the frequency of the event evaluation.

Table 1 compares the overall sequencing mechanism of each language and the characteristics of block oriented nonprocedural languages.

Table 1. Comparison of the Sequencing Mechanisms

	sequencing mechanism	invocation mechanism	guard condition	guard control
VHDL	process nonprocedural	sensitivity list	multi	enable, disable
GSP2	event block nonprocedural	expression	one	arm, disarm
GSP	procedural	port value change	n/a	n/a
HHDL	process nonprocedural	expression	multi	permit, inhibit
ISP'	process nonprocedural	expression	one	no
TI's HDL	procedural	port value change	n/a	n/a
SLIDE	process nonprocedural	expression	one	priority
ISPS	procedural	n/a	n/a	n/a
AHPL	procedural	n/a	n/a	n/a

3.3 APPLICABILITY TO THE GENERIC CHIP LEVEL MODELING STRUCTURES

As Armstrong reported [1], chip level modeling can be accomplished by implementing two basic classes of structures. These structures are called Generalized Delay Models (GDMs) and the Minimum Energy Models (MEMs). Using these basic models in various forms, one can implement any chip level model.

This section discusses the necessary constructs to implement these generalized structures.

3.3.1 Generalized Delay Models

As shown in Figure 5. Generalized Delay Models are classified into the following three categories.

3.3.1.1 Simple Delay

Simple propagation delay from an input to an output is easily modeled at the chip level. The mechanism involved is the delayed assignment of one variable to another variable. This is accomplished by scheduling a subsequent queue event to cause this to happen [1]. VHDL,HHDL,GSP2,GSP and TI's HDL have delayed assignment constructs so that one can implement the simple delay. For example, in GSP2 one would write:

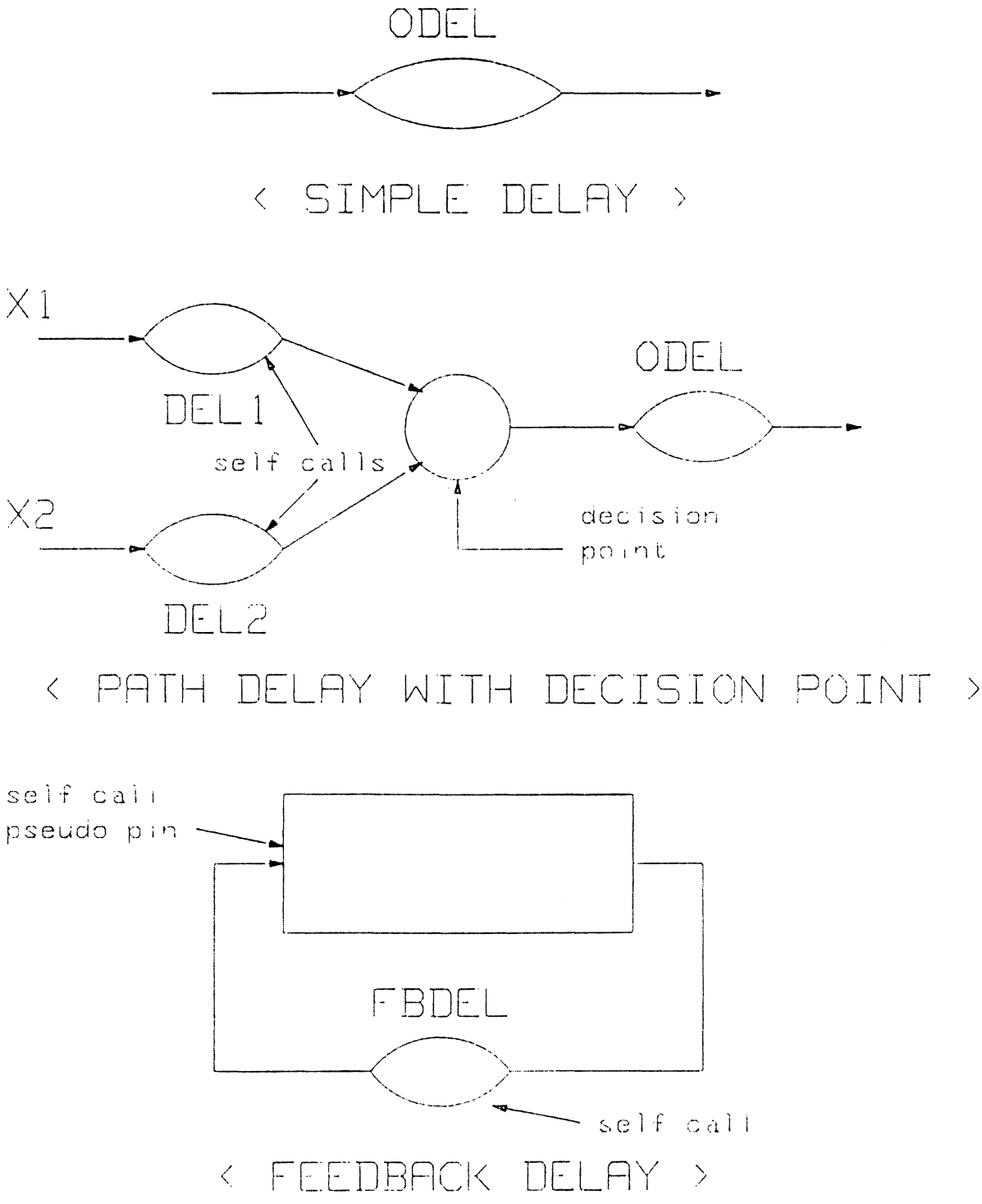


Figure 6. Generalized Delay Model

```
A := B  PROP 10;
```

which will cause variable A takes on the value of variable after 10 time units.

3.3.1.2 Path Delay with Decision Point

This model is also shown in Figure 5. Here inputs X1 and X2 activate separate delay paths (DEL1 and DEL2) which converge at a decision point. At this point a logical decision is made to determine whether to propagate a new value forward through the output delay path ODEL. This structure and variations on it are very commonly encountered in chip level models. The basic requirements for the hardware description languages to implement this structure are:

1. Simple propagation delay construct discussed in 3.3.3.1., and

2. A module self-invocation capability, and

3. Constructs to identify the source of the module invocation.

Basically the languages that have event driven mechanisms can implement the delay with decision point structure with varying degree of difficulties.

In GSP, a module description starts execution when any input pins or special pins (selfcall pins) change value. The special pins are used to self-invoke the module. Identification of the source of the module call is performed by comparing the previous value with the current value of the pin. That is, every selfcall pin that a modeler uses has an associated internal storage location to store the previous value. Whenever the module is invoked, coded comparisons are made to see which pin changes value. Then the pin which receives new value is identified as the source of module call. After that the program jumps to the decision point to perform the appropriate actions. The implementation of this structure with TI's HDL is similar to that of GSP. INOUT pins are used as selfcall pins and a function @VCHANGE is provided to identify the source of the module call. The reason that GSP and TI's HDL use the selfcall mechanism to implement the delay with decision point is that only one process[2] can be described within a module description. Other languages such as HHDL, GSP2 and VHDL, on the other hand, can contain mul-

---

[2] a process here means an independantly executing environment

tiple processes within a module and one process can directly invoke another process. The minimum requirements to implement this model with multiple processes are that there must be a construct for delayed internal signal propagation, also a process must be able to respond to the value change of the internal signals.

In Figure 7 is shown the VHDL implementation of the delay with decision model. This program describes the behavior of a chip that performs the logical 'and' or 'exclusive-or' operation between two four bit vectors A and B, according to the value of the SELECT input. If SELECT is true then the 'and' operation is selected, otherwise the exclusive-or operation is selected. First an entity named LOGIC_MUX is declared, which describes the interface. Next is given the architecture declaration that describes the behavior of an entity. The block statement inside the body defines the region where new signals can be declared. In this description, two internal four bit signals ANDVAL and XORVAL are declared. The behavior of the chip is defined in terms of two processes: PATH_DEL and DECISION. In PATH_DEL process, whenever the input signal A or B change value, the result of 'and' operation is propagated to ANDVAL after 10 ns and the result of 'exclusive-or' operation is propagated to XORVAL after 20 ns. The value changes of ANDVAL and XORVAL or a value change of the control input SELECT trigger the process DECISION and

a decision is made as to whether ANDVAL or XORVAL value should be propagated to the output C with a delay of 10 ns.

3.3.1.3 Feedback Delay

A third common delay model is the feedback delay model [1]. This model is known as the Huffman model (Figure 8) and has great general application in that it can be used to model any sequential circuit. The forward path consists solely of combinational logic while the feedback delay incorporates the memory device. For sequential circuits the delay is the form of clocked flip-flops, while in asynchronous circuits the feedback is pure propagation delay. As an example, consider a counter module. The module operates such that when the input CON=1, the counter counts up at a 2MHz rate using an internal oscillator. When CON=0, the state of the counter does not change. The combinational logic section consists of an incrementer which is enabled by the CON input. The feedback delay would be implemented with clocked flip-flops which would store the state of the counter.

The feedback delay method can be modeled by using the selfcall mechanism discussed above. It also could be modeled using the VHDL process or UPON subprocess of HHDL in which the output of the process is fedback to the internal signals and is part of the sensitivity list or checklist of the

```
entity LOGIC_MUX
 (A: in  bit_vector;
  B: in  bit_vector;
  SELECT: in bit;
  C: out bit_vector)  is
end LOGIC_MUX;

architecture EXAMPLE of LOGIC_MUX is
 A: block
     signal ANDVAL, XORVAL : bit_vector(3 downto 0);

  PATH_DEL: process ( A,B )
             begin
              ANDVAL <= A and B after 10 ns;
              XORVAL <= A xor B after 20 ns;
            end process PATH_DEL;

  DECISION: process ( ANDVAL,XORVAL,SELECT )
             begin
              if SELECT = '1'
               then
                C <= ANDVAL after 10 ns;
               else
                C <= XORVAL after 10 ns;
              end if;
            end process DECISION;
 end block A;
end EXAMPLE;
```

Figure 7. VHDL Implementation of the Delay with Decision Point Model

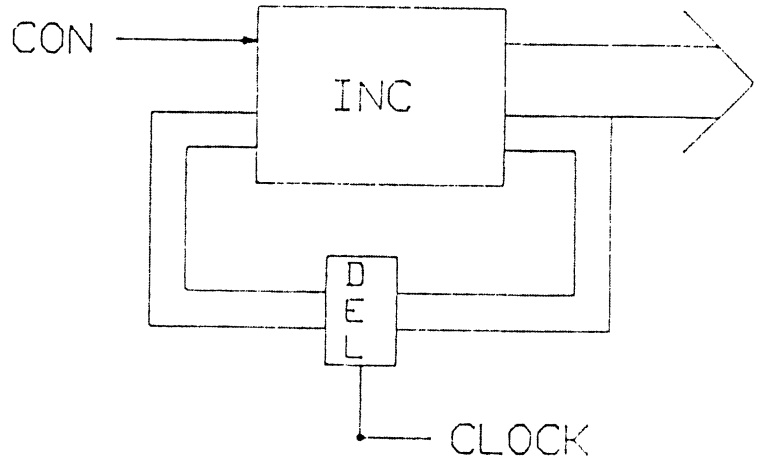


Figure 8. Huffman Model

process. Here we will illustrate the selfcall approach. Shown in Figure 9 is a GSP2 implementation of the module. In this example note that the counter description is enclosed within the module body. In the module description, first literals, pins, and internal registers are declared. The behavioral description of the counter is contained within the 'event' block. Here the event block has the name CLK_RISE and is triggered when the @rise(CLK) function evaluates to a logic 1. Within the event block, the counter is incremented, the new value of the count scheduled for outputing and the clock signal(CLK) transitions are scheduled. Since CLK is declared as bi-directional, changes on it can be scheduled and when those changes occur the module will be activated by the rise on CLK. Thus the selfcall mechanism is achieved.

```
module  COUNTER
 literals
  CLOCK_WIDTH = '500'
  OUT_DEL = '20'
 pins
  CON: input; Z[0|4]: output; CLK: bidirect
 declare
  COUNT[0|4]: register

event CLK_RISE On @and(@rise(CLK), CON=#B1);
 begin
  COUNT:= @add(COUNT,#b1);
  Z:= COUNT prop OUT_DEL;
  CLK:= #b0 prop (CLOCK_WIDTH/2);
  CLK:= #b1 prop CLOCK_WIDTH
 end
endmod
```

Figure 9. GSP2 Representation of the Counter Using Selfcall

Similar counting processes can also be implemented with 'delay' or 'wait' construct which suspends the module execution for a specific amount of time. In the example in Figure 10, the event condition becomes true when the CON signal goes up to logic 1. After that counting continues with the interval of CLOCK_WIDTH time units.

There are two kinds of constructs for suspending the module execution. One is the construct that suspends the module execution for a specific amount of time ('delay' of ISP', 'wait' of TI's HDL and 'wait' of GSP2) and the other is the construct that suspends the module execution until a specified condition occurs ('wait' of ISP'). We shall term it a 'conditional wait' construct. HHDL's wait construct has mixed characteristics of the above two kinds of constructs. Consider the two HHDL examples shown below:

```
WAITFOR true 30;

WAITFOR a=1 CHECK a;
```

The first example specifies that the execution of subprocess is suspended for 30 time units. The second example shows that the subprocess is waiting for the rising signal change of a. In this example a statement WAITFOR is followed by a wait termination condition (a=1) and a sensitivity list (CHECK a)

```
event CON_RISE  on  @rise (CON);
 begin
  while CON=#b1   do
   begin
   COUNT:= @add(COUNT,#b1);
   Z:= COUNT   prop OUT_DEL;
   wait CLOCK_WIDTH
   end
 end
```

Figure 10. Description of Counter with 'wait' Construct

So if signal a changes value and the changed value of a is 1, then the module suspension is terminated. The conditional wait construct eases the specification of state transitions by allowing a description to wait for an signal transition and then continue from the wait point.

The module suspension mechanisms are implemented in essentially two different ways:

1. ISP' 'delay' and TI's HDL 'wait' are used to suspend the module execution. The statement 'DELAY 20' implies the process is suspended for 20 time units and then resumed at the point of suspension. During the suspension period, the process is unable to respond to internal and external stimuli, but these value changes are saved for processing after the suspension period has elapsed. So this construct can be used to allow a recently activated block to gather input changes before preceding with sim-

ulation. This helps eliminate process or block "thrashing". We shall term this construct a nonretriggerable wait.

2. GSP2 also has a 'wait' construct which has the same form as the TI's HDL construct. However, the construct is designed so that external signal changes that occur during the wait period do activate the event block. We shall term this construct a retriggerable wait.

The WAIT construct was originally designed to model at the computer architecture level. In this application, processes can in many cases be considered to be atomic in that once activated they are isolated and need not respond to further inputs. In modeling logic at the chip level, one can not make this assumption. The retriggerable wait feature might be useful to describe the behavior of overiding clear or reset line of digital hardware[20] and it might be encountered during fault simulation [43]. Thus from the point view of chip level modeling, the retriggerable wait construct is preferred. Suppose we have retriggerable wait construct but the nonretriggerable wait is desirable in certain cases. We may combine the sensitivity control statements such as ARM and DISARM of GSP2 with 'wait' to accomplish the effect of the nontriggerable wait. Therefore, the retriggerable wait

construct is more versatile than the nonretriggerable wait construct.

VHDL does not have the 'wait' construct but this mechanism can be implemented with the process construct and internal signal declaration. As we can see in the example shown in Figure 11, we can implement the 'wait' by separating the description into two processes and by using a signal assignment statement with propagation delay. In the process P1, a signal WAIT is inverted and scheduled with WAIT_TIME delay. This WAIT signal value change triggers the process P2 after WAIT_TIME. So this implementation effectively simulates the 'wait' mechanism.

One of the restrictions of this implementation is that we can only implement the retriggerable wait. That is, during the wait period the process responds to the input stimuli. This is because one can disable signal X and Y just before executing "WAIT <= not WAIT after WAIT_TIME" but we have no means to re-enable the signal X and Y since an 'enable' statement within second process can not affect the sensitivity list of first process.

The second restriction is that the implementation of the wait mechanism with VHDL may force the modeler to create extra signals and processes, which may complicate the description

```
B: block
    signal WAIT: bit;
    P1: process (X,Y)
        begin
         --
         --
         WAIT <= not WAIT after WAIT_TIME;
        end process P1;

    P2: process (WAIT)
        begin
         --
         --
       end process P2;
end block B;
```

Figure 11. VHDL Implementation of 'wait'

and decrease the readability of the description. An example in Figure 12 represents this situation.

Figure 12 is a partial description of a simple processor MARK2 [25]. The description has three processes: A, B and C. In the process A, the contents of a program counter PC is loaded into a memory address register MA by executing a statement " MA <= PC; ", then the processor issues a memory read signal (RW <= 1) and waits for 150 ns to get valid data from a data bus. This waiting mechanism is implemented by executing a statement " WAIT <= not WAIT after 150 ns; ". After 150 ns, the process B is invoked since the value of a WAIT signal has been changed. The first statement of the process B is to fetch the instruction into an instruction register IR (IR <= DATA;). The second statement " PC <= ADD (PC,"00001") " is to increment the program counter. Notice at this point that the program counter PC must be declared as a signal global to all processes since PC is used within the process A and the process B. PC can not be declared as a variable since a variable in VHDL is local to the process where it is declared. The fact that the PC must be a signal affects the rest of the description. The process C is to decode the instruction and to perform the appropriate actions according to the instruction. The instruction 1 " when 1 => PC <= ADD (SUB (PC, "000001") , IR (7 downto 3)) " is to jump to the address relative to the program counter. Notice that the content of the program counter indexed is the value

```
A: process ( CLK )
    begin
     CLK <= not CLK after 450 ns;        -- internal clock
     MA <= PC;
     RW <= '1';
      .
      .
     WAIT <= not WAIT after 150 ns;
end process A;

B: process ( WAIT )
    begin
     IR <= DATA;
     PC <= ADD ( PC, "00001" );
      .
      .
     NEXT <= not NEXT after 10 ns;
end process B;

C: process ( NEXT )
   begin
    case INTVAL (IR(2 downto 0 )) is
     when 0 => PC <= IR (7 downto 3);
     when 1 => PC <= ADD(SUB(PC,"00001"),IR(7 downto 3));
       .
       .
       .
    end case;
end process C;
```

Figure 12. An Example of VHDL Description with Delay Mechanism

of the program counter before it is incremented by the statement " PC <= ADD (PC, "00001") " in the process B. To offset the result of incrementing the program counter, the program counter is decremented by 1 and added to the operand portion of the instruction (IR(7 downto 3)), then the resulting value is loaded to the program counter. Here the value of the program counter used as an argument of the 'SUB' function must be the incremented value resulting from the execution of the statement " PC <= ADD (PC,"00001") " in process B. The program counter PC has already been declared as a signal to implement the delay mechanism . The problem here is that in VHDL, a result of the execution of a signal assignment statement is not immediately available. That is, at least infinitesimal time must be passed in order to get the result of a signal assignment statement. Suppose that one attempts to describe process B and process C within a single process, then the value of the program counter used as an argument of the 'SUB' function is not the correct value since the incremented valued of the program counter is not available at this point in time. This is the reason why the instruction fetch (process B) and the instruction decode (process C) cycle are described in separate processes and another delay mechanism (NEXT <= not NEXT after 10 ns) is implemented. The result of the separation of the processes is that the instruction register IR must be declared as a signal since IR is used both in process A and process B. This

kind of reasoning goes on and on through the entire description of an entity. Also, the wait mechanism can not be used within a procedure of VHDL since it must be implemented with at least two processes. The procedure of VHDL does not accept the multiple processes. The complete description of the MARK2 processor is given in Appendix A.

3.3.2 Minimum Energy Model

In addition to the modeling of propagation delay, one must also be concerned with minimum energy problems on signal inputs. In terms of chip specification parameters this concern is expressed as set-up time, hold time and minimum pulse width requirements [1]. At the electrical circuit level, the minimum energy requirement is expressed in terms of the amount of charge required to turn on a transistor. When modeling at a high level, this requirement translates into specifying that a signal should be stable for a certain time relative to a certain event. In older HDLs the modeler wrote code to implement this signal stability check, in newer languages the capability is built into the construct.

Let's take one example that has set-up, hold time and minimum pulse width requirements on the input pins. Figure 13 shows input timing specifications.

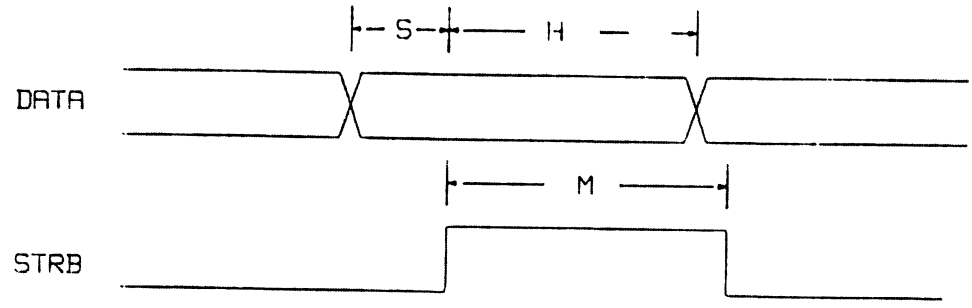


Figure 13. Input Timing Diagram

In this example, the DATA input should be stable at least S time units before the STB iinput makes a 0 to 1 transition and H time units afterward for correct sampling of DATA input. Also STRB should be stable at least M time units.

VHDL

VHDL has built-in signal attributes STABLE(t) and DELAYED(t). STRB'STABLE(t) evaluates to TRUE when the signal STRB has been stable for the t time units. STRB'STABLE(0) or STRB'STABLE evaluates to FALSE only when the signal STRB just makes a transition. DELAYED(t) delays the value of a signal t time units. Therefore STRB'DELAYED(t)'STABLE will evaluate to FALSE t time units after signal STRB has changed. VHDL has an assertion mechanism which specifies the condition that is expected to be true all the time. When the condition evaluates to FALSE, a corresponding user defined message can be reported.

Combining the assertion mechanism and the built-in signal attributes, we can describe the set-up, hold time and minimum pulse width specifications as follows:

```
assert STRB'STABLE or not STRB or DATA'STABLE(S)
report "DATA SET-UP TIME VIOLATION."

assert STRB'DELAYED(H)'STABLE or not STRB'DELAYED(H) or
       DATA'STABLE(H)
report "DATA HOLD TIME VIOLATION."

assert STRB'STABLE or STRB or STRB'STABLE(H)
report "STRB MINIMUM PULSE WIDTH VIOLATION."
```

Note that the assertion expression for the set-up time checking is equivalent to "not(not STRB'STABLE and STRB and not DATA'STABLE(S))" using De Morgan's theorem. Similar statements can be made for the other two checks.

GSP2

GSP2 has a special function @SETUP(X,t). This function is used to monitor the past history of a vector. @SETUP(X,t) evaluates to '1' when a vector has been unchanged for t time units. This function is similar to the signal attribute

STABLE(t) OF VHDL. But @SETUP(X,0) which corresponds to X'STABLE is not defined in GSP2. Instead GSP2 provides the functions @RISE(X) and @FALL(X) that detect the direction of the transition of a vector. The setup requirement of the previous example may be written with GSP2 as below:

```
EVENT setup_chk ON @AND(@SETUP(DATA,S),@RISE(STRB));
  BEGIN
  --------
  --------
  END
```

In this example event condition is given by ANDing the values of the two functions @SETUP(DATA,S) and @RISE(STRB). So when the STRB input just makes a 0 to 1 transition and DATA input has been stable for S time units, the statements section begin execution.

The hold time requirement checking is somewhat different from that of VHDL. GSP2 does not have a function to delay the value of a signal. But GSP2 has a WAIT statement that suspends the execution of an event block for the specified amount of time. Using this WAIT construct together with SETUP function, we can implement the hold time requirement with GSP2. In fact, the hold time test checks the stability of the future value of one signal relative to the particular transition of an-

other signal. But suspending the execution of the module for the hold time units as soon as the module sees the particular transition of one signal and then checking the stability of another signal after that delay time has the same effect as to check the stability of the future value.

```
EVENT setup_hold_chk ON @AND(@SETUP(DATA,S),@RISE(STRB));
 BEGIN
  WAIT H;
  IF @SETUP(DATA,H)    THEN      ! hold time test !
    -----------
    -----------
  ELSE
   WRITE LOG 'DATA HOLD TIME VIOLATION.', CURRENT$TIME
  ENDIF
 END
```

In this example, as soon as the signal STRB makes 0 to 1 transition, the set-up checking is performed. If the set-up test is successfully passed, then the event block is delayed H time units. After that, the check is made to see whether the DATA signal has been stable or not during the delay time unit (H time units). If it has not been stable, an error message is reported.

Minimum pulse width can be described similarly as follows:

```
EVENT min_pulse_width  ON  @RISE(STRB);
 BEGIN
  WAIT M;
  IF @NOT(@SETUP(STRB,M))  THEN
   WRITE LOG 'STRB PULSE WIDTH IS TOO SHORT',CURRENT$TIME
  ENDIF
 END
```

TI's HDL

In TI's HDL, the timing relationship between arbitrary signals can be described in the TIMING CONSTRAINTS sub-section. These constraints are monitored during simulation. The previous example can be written in TI's HDL as below:

```
TIMING CONSTRAINTS
DATA TXX MIN S  BEFORE STRB TLH;
    (* SETUP CHECK *)
DATA TXX MIN H  AFTER  STRB TLH;
    (* HOLD  CHECK *)
STRB TXX MIN M  AFTER  STRB TLH;
 (* MINIMUM PULSE WIDTH CHECK *)
```

The entry TXX means any transition and TLH represents rising transition. In the above example, the set-up and hold time constraints state that the DATA signal should not change

during S time units before the leading edge of the STRB signal, also should be stable H time units afterward.

ISP'

The current version of ISP' does not have a construct for the minimum energy modeling but they have a plan to add the setup and hold test statements.

The proposed setup statement tests that all the ports listed have been stable for at least expression time units. If one of the listed ports has been changed, a flag is raised which can be detected by the simulator. The hold statement tests that all the listed ports remain stable for at least expression time units. If one of the listed ports changes in the next expression time units a flag is raised. In other words, the setup statement checks past value changes while the hold statement checks future value changes of signals. The following example illustrates the ISP' description of setup and hold time requirements:

```
when (STRB:lead):= (
   setup DATA for S ;  /setup check/
   hold  DATA for H ;  /hold check /
   hold  STRB for M ;  /minimum pulse width check/
       ---------
```

```
---------
----------     )
```

In this example, as soon as the input port STRB sees the rising edge, the setup and hold statements begin execution simultaneously.

By comparing various HDL's we can identify the three basic constructs that are necessary to describe set-up, hold time and minimum pulse width.

1. a construct that can monitor the past history of signals.

2. constructs that can detect the change as well as the direction of the change of signals.

3. a construct that can delay the value of signals such as 'delayed' signal attribute of VHDL.

In addition to these constructs, reporting facility that can give users specific error massages should be contained in the HDL's.

Table 2 compares the applicability of HDLs to the generic chip level modeling structures. VHDL, GSP2 and the forth coming version of ISP' can implement the generic modeling

structures easily. These languages provide high level constructs to implement the generalized delay models and the minimum energy models. HHDL can implement the generalized delay models easily with high level constructs but the minimum energy model must be implemented with rather primitive constructs. That is, HHDL does not have a construct that checks the stability of a signal during certain amount of time. Instead, it provides a function that tells a model the current simulation time. By comparing the previously activated time and the currently activated time of a signal, a model can determine whether the signal has been stable or not for a certain amount of time. ISPS, AHPL and SLIDE can not implement any of the four generic modeling structures. The GSP implementations of the generalized modeling structures except simple delay model are classified as 'possible but difficult to describe'. The reason is that the language itself is a low level language. Therefore a modeler must code the mechanism with low level constructs such as selfcalls and flag variables.

Table 2. Comparative Applicability to the Generic Chip Level Modeling Structures

	simple delay	delay with decision point	feedback delay	minimum energy model
GSP	E	D	D	D
GSP2	E	E	E	E
VHDL	E	E	E	E
HHDL	E	E	E	D
TI's HDL	E	D	D	E
ISP'	E	E	E	E
ISPS	N	N	N	N
AHPL	N	N	N	N
SLIDE	N	N	N	N

- legend -

D: possible but difficult to implement
E: easy to implement
N: impossible to implement

3.4 ABSTRACTION OF DATA AND OPERATION

3.4.1 Abstraction of Data

In order to model hardware, it is necessary to describe the electrical signals that are used to convey information within the hardware in terms of abstractions that are more easily understood [44]. One such abstraction is the 'type'. Hardware description languages typically include a data type that represents the bit values '0' and '1' with the standard operations defined on bits and bit_vectors. Also, HDL's provide the multi-dimensional array type to model bulk storage. The HDL's included in this comparison provide such predefined data types. VHDL and HHDL provide user defined typing mechanism to support the higher level of abstraction.

In this chapter, the terminology defined in VHDL will be used as a standard in describing and comparing other languages.

VHDL

The basic unit of data in VHDL is an object which is the container of a value within a system. Each object has a set of properties that control what value the object may have and what operations can be applied to the object. There are three classes of objects in VHDL: constants, variables, and

signals. A constant is an object whose value can not be changed. Its value is determined when it is created by a constant declaration. A variable, on the other hand, is an object whose value may be changed. A variable declaration creates a container that may hold any value of the type of the corresponding variable assignment statement. When this occurs, any old value in the container is immediately replaced by the new one. Signals are also objects whose values may be changed. Signals and variables differ, however, in the number of containers involved. A variable represents a single container of the values, and the value of the variable is the value in that container. In contrast, a signal represents a collection of one or more containers, and the value of the signal is a function of the values of all of the containers in the collection. Such containers are called 'drivers' of the signal. Another characteristic of signals that differentiates them from variables is that they have a time dimension. Values assigned to the drivers of the signal are always scheduled to 'occur'(i.e.,to become the value of the driver) at future points in time. Consequently, no signal assignment statement ever affects the current value of a signal. So the signal assignment statements below describe the data swapping between a signal A and signal B:

```
A <= B;
B <= A;
```

Every VHDL object has an associated type. An object of a given type may need to be restricted so that it may only contain a subset of values of the type. VHDL is a strongly typed language in the sense that in any context where type matching is meaningful, such as logical expression, assignment and procedure parameter passing, provision must be made for verifying type correspondence. VHDL includes several predefined types and subtypes. Predefined types are BIT, BIT_VECTOR, INTEGER, REAL, BOOLEAN, CHARACTER, and STRING. Predefined subtypes include POSITIVE and NATURAL. The sets of values of the predefined types are shown below [6].

PREDEFINED TYPES	SET OF VALUES
BIT	'0' and '1'
BIT_VECTOR	All arrays of bits
INTEGER	All integers supported by implementation
REAL	All floating point numbers supported by the implementation
BOOLEAN	The values 'false' and 'true'
CHARACTER	All characters in the ASCII character set
STRING	All arrays of characters

Besides these predefined types and subtypes, VHDL provides user defined typing mechanisms to be used to create a wide variety of data types. These types are classified into scalar

types and composite types. Scalar types include enumeration type, numeric type, and physical type. Physical type allows the expression of quantities that carry a unit of measurement. A physical type declaration specifies a set of such units, all defined in terms of some base unit. Below is the definition for the physical type TIME:

```
type TIME is range 0 to 1E18
units
 fs;              --femtosecond (base)
 ps= 1000fs       --picosecond
 ns= 1000ps       --nanosecond
 us= 1000ns       --microsecond
 ms= 1000us       --milisecond
 s=  1000ms       --second
 min= 60s         --minute
 hour= 60min      --hour
```

The base unit is femtoseconds(fs) and all the other units are defined as multiples (directly or indirectly) of that base unit. Composite types are types that consist of separate elements of scalar types or of other composite types. The two classes of composite types are arrays and records: the elements of an array type are all the same type, whereas the elements of a record may be different.

VHDL provides another abstraction mechanism called 'PACKAGE' [6]. A package a mechanism in VHDL for grouping and storing declarations. One of the advantages of placing declarations inside packages is that the design data contained inside packages may be shared among several designers. Once a package is defined, it may be referenced by any other description. A context clause (with PACKAGE_NAME, use PACKAGE_NAME) at the beginning of a description specifies which package contents will be visible to the description. Usually the declarations stored in a package are related in some way. For instance, a package might contain declarations relating to one's complement arithmetic, or might contain declaration relating to multi_valued logic.

Since VHDL is a strongly typed language, there is no implicit type conversion in VHDL. To do a type conversion, a user must write a specific type conversion function he or she want to perform.

GSP2

There are three classes of objects: signals, variables and constants.

Each object is associated with four kinds of data types: bit, bit_vector, two dimensional array of bits and integer. The

relation between objects and data types are given in Figure 14. Integers are used in program control constructs and as timing delay parameters and may not represent actual hardware. The basic operations of add, subtract, multiply, divide, and unary minus are provided to manipulate them.

There are two type conversion functions: @VECT and @INT. The @VECT function is to convert integers to bit_vectors and the @INT function is to convert bit_vectors to integers. Signed magnitude representation of an integer form is assumed in these type conversions. They are the only type conversion functions GSP2 provides. Variables defined to be type bit and bit_vectors are used to model registers and flip-flops found in the digital device to be simulated. GSP2 is a strongly typed language, meaning that operations (including assignment) can not be performed on data that is not the same type. Accessing of subfields of registers are allowed in GSP2. Arrays of bit_vectors are limited to be two dimensional and can be used to model RAMs, ROMs, and other bulk storage devices. Ports(PINS) are just a special type of signals in that each port has a direction associated with it. The input and output behavior of a modeled device is handled in GSP2 by using these ports. There are three port modes in port signals: in, out, inout. Each port must specify one of these three port modes. A variable of GSP2 has a combined characteristic of a signal and variable of VHDL. A variable as-

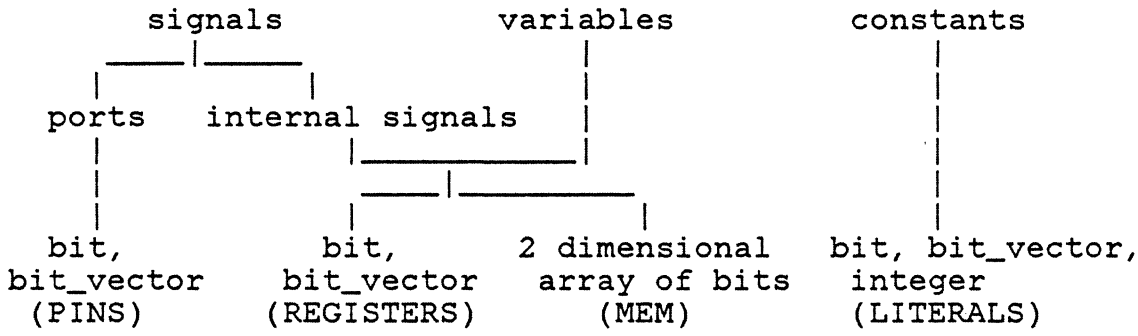


Figure 14. Objects and Data Types in GSP2

---

signment statement without propagation delay(PROP) of GSP2 acts like a variable assignment statement of VHDL in that a variable which is on the left side of an assignment statement takes on the value of a right side expression immediately. A variable assignment statement with PROP clause of GSP2, on the other hand, behaves like a signal assignment statement which has a time dimension.

GSP

GSP has three classes of objects: signals, variables and integers. Also three kinds of data types are associated with objects. Data types in GSP are bit, bit_vector and integer. Unlike VHDL, a signal and a variable are not differentiated in the declaration section. They are declared as REGs which mean registers. If an object is declared as a REG and is used

in an assignment instruction with propagation delay, it is treated as a signal which has a time dimension. Otherwise it is treated as a variable which means the destination of an assignment instruction takes on a value of the source immediately. A variable and a signal declared as a REG can only have a bit or a bit_vector type. A constant of GSP is declared in EVW part of a declaration (see below example) and can only have an integer type. The value of the variable and the signal are not available to the other modules unless their values are transfered to the ports (PINs) which are logical interfaces to the outside of the module. Ports (PINs) work like signals and variables except that lines from other module can connect to these ports. Only port value changes cause module calls. That is, when a source module port is connected to a destination module port, the source port state change causes the destination port state change and this state change invokes the destination module. Figure 15 represents the relationship between object and data types.

An example of a GSP declaration part is given below:

```
REG(1) ENTEM,RSTEM
REG(3) UPTEM
PIN    DATA(1,8), ENABLE(9), OSC(151)
EVW    CP(100), RS(25)
```

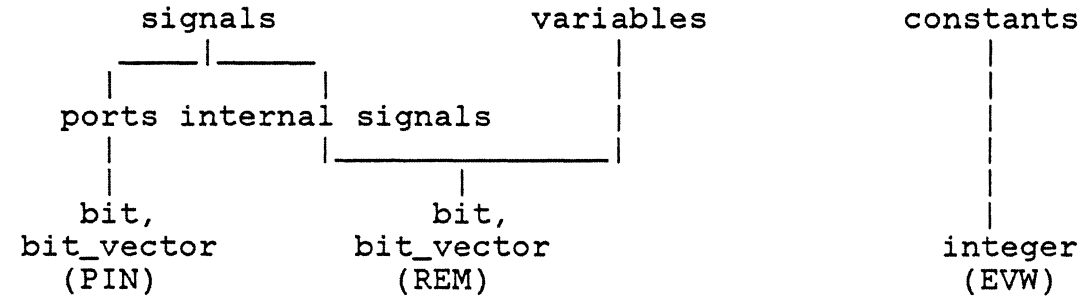


Figure 15. Objects and Data Types in GSP

The above declaration defines two single bit registers ENTEM and RSTEM, a three bit register UPTEM. PIN variable DATA has pin number 1 through 8, ENABLE is a single bit pin which is a pin number 9. Pins that have pin numbers above 150 are used as pseudo pins to implement the self call mechanism. Writing to these pins resuls in self-invocation of the module. The EVW (Event Write) section defines integer type constants which are used to represent simulation time units. In the above example CP and RS define 100 and 25 simulation time units respectively. Two dimensional arrays (memories) can not be declared in GSP, but there is a way to represent and to access them using IDX ,MOV instructions and BYT as below:

```
( assume that IR contains '01010110' )
        IDX   IR(0),4,1
        MOV   MEM@1,CON
 MEM:   BYT   #00,#01,#02,#03,#04,#05,#06
```

```
BYT  #07,#08,#09,#10
```

The above example illustrates the method of reading data from the integer array labeled MEM. First, the lowest 4 bits of the IR register are converted to an integer, then this integer is put into the index register1 (IDX IR(0),4,1). After that, the content of the memory which is indexed by the value of the index register1 is transfered to the register CON. Consequently, CON has a value '#06' in a binary form. This memory access process actually involves two internal type conversions (binary -> integer, integer -> binary). Writing to this address can be performed by reversing the source and destination of the MOV instruction (e.g., MOV CON,MEM@1). These two implicit type conversion mechanisms are the only type conversion methods available in GSP.

TI's HDL

There are two kinds of objects in the behavioral program of the TI's HDL: variable and signal. A variable which has a type bit or bit_vector may be declared within one of two variable declaration sections: INTEGER and BOOLEAN. A variable declared within an INTEGER section is interpreted as a two's complement number of 32 bits within the behavioral program, while a variable declared within a BOOLEAN section is not.

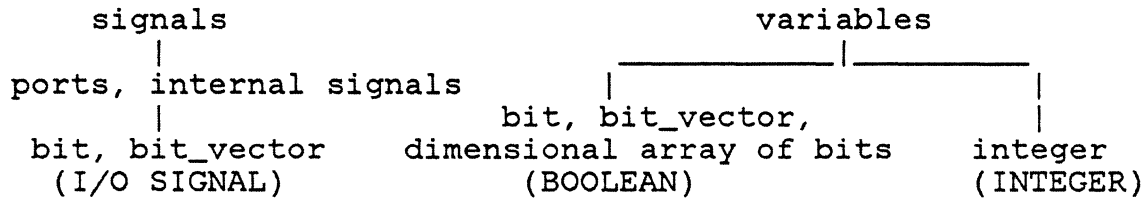


Figure 16. Objects and Data Types in TI's HDL

Figure 16 represents the relationship between objects and data types.

An example of a variable declaration is shown below:

```
INTEGER I(0 TO 31),K,L;
BOOLEAN X(0 TO 15), Y(7 TO 0, 15 TO 0);
```

In the above example, Variables K and L are treated as 32 bit two's complement numbers. A variable declared in the BOOLEAN section can be a two dimensional array of bits.

An I/O signal list which corresponds to the ports in VHDL immediately follows the block statement. A port signal must have one of the five port modes: INPUT, OUTPUT, INOUT, GLOBAL, and LOCAL. The GLOBAL port mode is used to identify signal names that are the same in every block. For example, power and ground must be declared with the GLOBAL mode. The

LOCAL port mode characterize signals which are entirely within a block. That is, the signals do not cross block boundaries.

ISP'

There are two kinds of objects in ISP': variables and constants. A variable can be declared in one of three sections: states, memories and ports. A variable declared in states and ports sections may be a bit type or a bit_vector type. But a variable declared in a memories section must be a two dimensional array of bits. Figure 17 represents the relationship between objects and data types.

A port declaration in ISP' consists of a port name followed by optional bit width, initial value, attribute specifications. There are three classes of port attribute as follows;

- input/output/bidirectional
- or/and interconnect
- connect/disconnect

The input/output/bidirectional attribute class (port mode) defines the direction of the data transfer of that port. The

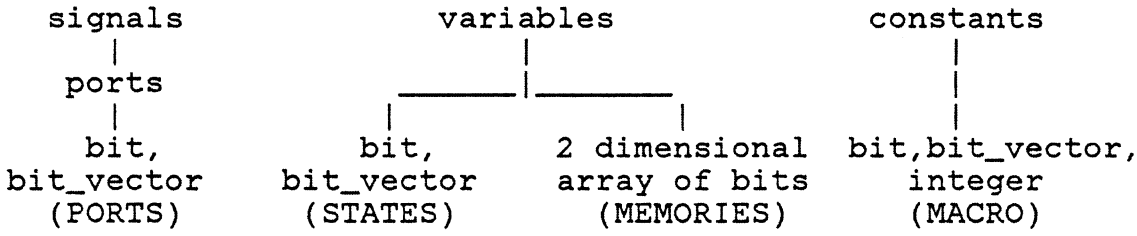


Figure 17. Objects and Data Types in ISP'

---

default value is bidirectional. The or/and attribute defines the logic function which is used when evaluating signals connected to the port. The value of a signal is defined to be the result of combining the values of all the connected ports on the signal. This attribute may be used to describe the 'wired or' or 'wired and' bus which has multiple sources. The default is 'or'. The connect/disconnect attribute defines the initial port connection status. Ports which are declared as input ports can not be declared as connected ports. Consider an example below:

```
ports address<16>(22)'and: output
```

Above declaration defines a 16 bit, output only, initially connected, initial value of 22, 'wired and' logic port named address.

ISP' is not a strongly typed language in that a variable of type bit or bit_vector is implicitly treated as a two's complement integer. Also logical operations between two different bit lengths can be performed. In this case, the bit length adjustments are determined automatically according to the context. There is no explicit type conversion mechanism in ISP'.

HHDL

HHDL is a language that has its basis the PASCAL language, so it provides the most of the PASCAL supported data types. There are three classes of object: signals, variables and constants. HHDL has a user defined typing mechanism as VHDL does. Explicit type conversion within a module is not permited in HHDL. However, a type conversion between modules can be implemented by creating a translation module which corresponds to a type conversion function. This translation module in HHDL is used to interface the more abstract signal types to the less abstract signal types. For example, an instruction set of a microprocessor can be declared as an enmeration type using character strings such as 'add' or 'jump'. But in the lower level of description these instructions must be used as a bit_vector type. In this case a translation module should be created just like other hardware

modules. Then the interconnections are made to this translation module from each module.

As can be seen in Table 3, every language supports at least bit and bit_vector types and two dimensional array of bits which are the most basic structures of digital systems. Only VHDL and HHDL provide the user defined typing mechanisms. Integers are also supported by every language. However, in some languages their usage is restricted. In GSP an integer can only be used as a constant and to represent the propagation delay. Integers in GSP2 can only be used as an index variable and a propagation delay time. Strongly typed languages such as VHDL, HHDL and GSP2 provide explicit type conversion mechanisms while implicit type conversions are performed in nonstrongly typed languages such as ISP' and TI's HDL. Concatenation of bits and bit_vectors are allowed in every language except GSP. Subfield declaration (alias in VHDL terminology) which gives different name to the subrange of previously declared object are allowed in VHDL, HHDL and ISP'.

Table 3. Comparison of Data Abstraction

	VHDL	GSP2	GSP	TI'HDL	ISP'	HHDL
user defined type	y	n	n	n	n	y
bit	y	y	y	y	y	y
bit_vector	y	y	y	y	y	y
integer	y	y	y	y	y	y
real	y	n	n	n	n	y
array dimension	multi	two	two	two	two	multi
subfield declaration	y	n	n	n	y	y
bit_vector concatenation	y	y	n	y	y	y
strongly typed	y	y	n	n	n	y
explicit type conversion	y	y	n	n	n	y

3.4.2 Operators

Operators are classified into logical, arithmetic, relational and vector. The vector operator includes shift and rotate operators. Expressions are formulas that define how a value is to be computed. Expressions consist of operands and operators. Rules of operator precedence are usually as in Algol.

3.4.2.1 Arithmetic Operators

Every language supports addition (+), subtraction (-), multiplication (*), and division(/) on integers. GSP has instructions for addition (ADD) and subtraction (SUB, NEG) on bit_vectors. VHDL and HHDL provide the real type and the above operators can be applied to the real type data. Table 4 is the list of operators that each language provides.

3.4.2.2 Logical Operators

Every language supports basic logical operators AND,OR,XOR and NOT. Logical operators that each language provides are shown in Table 5. ISPS and TI's HDL provide equivalence operator (EQV). If the arguments of the EQV operator are unmatched in length, a shorter one is expended in ISP'. TILADS provides the parity operator (#/). The result of this opera-

tion is '1' if an argument has odd number of one's, otherwise its result is '0'.

3.4.2.3 Relational Operators

Every language except GSP has the same relational operators even though the symbols are different. Relational operators that each language provides are shown in Table 5. The meanings of the symbols are self explanatory.

3.4.2.4 Shift and Rotate operators

GSP, ISPS, ISP', GSP2, and TI's HDL provide shift and rotate operators or functions. VHDL and AHPL have no such primitives, but those operations can be accomplished by doing index manipulation. For example, bit_vector REG(3 downto 0) of VHDL can be shifted one bit left with zero filling by signal assignment statements as follow:

```
for I=3 downto 0  loop
 REG(I) <= REG(I-1);
end loop;
 REG(0) <= '0'
```

Table 7 compares the shift and rotate operators that each language provides. ISP' and ISPS provide extensive sets of

shift and rotate operators. In ISPS, all shift operators have a name of the form Sxy where x is either L(left) or R(right) to indicate the direction of shifting, and y is either 0,1,R,D, or I to indicated the source of shift-in bits. The first two (0,1) indicate a continuous stream of 0 or 1 bits, respectively. R indicate a Rotation. D indicates a duplication, and the shift-in bits are simply a replication of the bit contained in the shift-in position. I indicates Immediate and the shift-in bit is the rightmost bit of the second operand.

In ISP', there are three types of shift operations: logical, arithmetic and rotates. The key words 'logical', 'arith', and 'rotate' are preceded by either a '*:' to indicate a left shift or rotate, or a '/:' to indicate a right shift or rotate.

Table 4. Arithmetic Operators

GSP	ISPS	AHPL	ISP'	VHDL	GSP2	TI's HDL	SLIDE
	+	+	+	+	+	+	+
	-	-	-	-	-	-	-
	*	*	*	*	*	*	*
	/	/	/	/	/	/	/
ADD					ADD		
SUB					SUB		
	MOD		MOD	MOD			MOD
			ABS	ABS			
		**		**			
NEG							
			SXT				
			EXT				
				REM			

- legend -

ADD: bit_vector addition
SUB: bit_vector subtraction
MOD: modulus
ABS: absolute value
** : exponentiation
SXT: sign extention
EXT: zero extention
REM: remainder

Table 5. Logocal Operators

GSP	ISPS	AHPL	ISP'	VHDL	GSP2	TI's HDL	SLIDE
AND	AND	AND	AND	AND	AND	AND	AND
OR	OR	OR	OR	OR	OR	OR	OR
XOR	XOR	XOR	XOR	XOR	XOR	XOR	XOR
NOT	NOT	NOT	NOT	NOT	NOT	NOT	NOT
			NAND	NAND			
			NOR	NOR			
	EQV						EQV
						#/	

Table 6. Relational Operators

GSP	ISPS	AHPL	ISP'	VHDL	GSP2	TI's HDL	SLIDE
BEQ	EQL	EQL	=	=	=	=	EQL
BNE	NEQ	NEQ	/=	<>	<>	<>	NEQ
	LSS	LSS	<	<	<	<	LSS
	LEQ	LEQ	<=	<=	<=	<=	LSS
	GTR	GTR	>	>	>	>	GTR
	GEQ	GEQ	>=	>=	>=	>=	GEQ

Table 7. Shift and Rotate Operators

GSP	ISPS	ISP'	GSP2	TI's HDL
SHR	Sxy	*:logical	SHIFT	SLL
ROR		/:logical	ROTATE	SRL
		*:arith		SCL
		/:arith		SRA

3.4.3 Control Statement

Table 8 compares the control statements in the languages. The control statements are classified into three categories: selector, loop and terminator. Almost every HDL's provides 'if_then_else' and 'case' statements to specify the selection of alternative actions and also provides conventional repetition statements 'while_do' and 'until_do'. VHDL 'EXIT WHEN' provides a mechanism to skip the iteration on certain condition. Terminators are the statements to exit from the executing module. VHDL 'RETURN', GSP2 and TI's HDL 'EXIT' statements terminate the current executing procedure or body. When the procedure or body is called next time, executions start from the beginning of the procedure or body.

There are two approaches to exit model procedures in GSP. With the Exit and Restart (EXR), the next time the procedure is called the execution will start at the beginning of the procedure. When an Exit and Continue (EXC) instruction is executed, execution resumes at the GSP instruction following the EXC when the procedure is called next time. EXR instructions are used where one is modeling the sampling of asynchronous inputs while the EXC is used in modeling synchronous behavior.

Table 8. Comparison of Control Statements

	SELECTOR	LOOP	TERMINATOR
GSP	BEQ BNE		EXR EXC
ISPS	IF => DECODE =>	REPEAT	LEAVE RESTART RESUME TERMINATE STOP( )
AHPL			ENDSEQUENCE
ISP'	IF CASE	WHILE DO UNTIL	
VHDL	IF_THEN_ELSE CASE	FOR LOOP WHILE LOOP NEXT WHEN EXIT WHEN	RETURN
GSP2	IF_THEN_ELSE CASE	LOOP WHILE	EXIT
TI's HDL	IF_THEN_ELSE CASE	FOR DO UNTIL DO	EXIT

3.5 TIMING MODE

This section discusses the necessary constructs for chip level HDLs to describe synchronous and asynchronous digital systems.

Generally, sequential networks can be classified into synchronous and asynchronous network depending on the timing point of view one adapts. In a synchronous system, a master clock controls the state changes of the entire network and therefore the state of the system is known at a specific point in time. In an asynchronous system, there is no master clock. A change in an input can take affect and propagate through the system at any point in time. Therefore the order in which input signals change is important in the determination of the behavior of the system [44].

AHPL is a hardware description language that describes a synchronous digital system. Data operations in AHPL are synchronized with the trailing edge (or leading edge) of the control pulse and control flip-flops are triggered by the trailing edge (or leading edge) of a clock signal which is implicit and drives the entire system. Other languages in a comparison set such as VHDL, GSP2, TI's HDL, HHDL, SLIDE, IPS' and GSP do not assume any specific kind of digital sys-

tem. They provide a general mechanism to describe synchronous and asynchronous systems. Generally, HDLs that have an event driven mechanism, which responds to stimulus of the system and can describe both kinds of digital system. In a description of a synchronous system with a language that has an event driven mechanism, a stimulus might be a master clock of the system. But in a description of an asynchronous system, stimuli to the system might be any signal changes that affect the output values.

VHDL and HHDL provide constructs that ease the description of synchronous behavior of a digital system. A guard expression together with a 'memoried' signal assignment statement is used to describe synchronous behavior. Consider the following VHDL example:

```
B1: block (CLK = '1' and not CLK'stable)
        begin
        SYNC :   Q <= memoried D;
        ASYNC:   P <= A and B;
       end block B1;
```

An expression that follows a block statement is a guard expression of this block. Whenever the CLK signal makes a 0 to 1 transition, this expression evaluates to true. There are two signal assignment statements labeled SYNC and ASYNC re-

spectively. The statement labeled SYNC has a reserved word 'memoried' while the statement labeled ASYNC has not. An assignment statement that has a reserved word 'memoried' is called a memoried assignment statement. This statement is only executed when a value of a guard expression becomes true and/or when the guard expression has been true and the value of a signal which is in the right side of arrow symbol (<=) is changed. So the memoried signal assignment statement in the above example is executed whenever the CLK signal makes a 0 to 1 transition. That is, execution of this statement is synchronized with the rising edge of the CLK signal. On the other hand, the statement labeled ASYNC in the above example is executed whenever the value of A and/or B changes.

In HHDL, a modeler can declare the clock which defines an infinite sequence of possibly multi-phase signals. These signals can be used to control synchronous circuitry. The SYNC operator is used for indicating synchronization of operations with the leading edges of declared clocks. An example of declaration of a clock and the usage of a synchronized signal assignment statement of HHDL is given below:

```
CLOCK: main_clk(1,2)    -- clock declaration
    ASSIGN x TO y  SYNC main_clk PHASE 1
```

The above example declares a clock named main_clk which has 2 phases of 1 unit time interval. When signal x changes value, signal y takes on a value of x on leading edge of phase 1 of main_clk.

3.6 COMMUNICATION MECHANISMS

In recent HDL's, port constructs are used to communicate with the other modules (intermodule communication). The ports are the signals through which the description module communicates with the outside world. In VHDL, the signal construct is also used to communicate within an entity. The difference between the port signal and internal signal is that the port signal has directions (i.e.,in, out, inout), while the internal signal has no direction and no restrictions. But the change of the internal signal can not be felt outside of the entity. Variables in VHDL are declared within a process and are local to the process. They are used only for algorithmic calculation. So there is no analogy with real hardware. The characteristic of signals that differentiates them from variables is that they have a time dimension. Values assigned to the signal are always scheduled to occur at future points in time. Consequently, no signal assignment statement ever affects the current value of a signal. HHDL also has a internal signal and port constructs (nets) which have the same semantics. But global variables can be declared within

a module(component) so there are two means to communicate between subprocesses.

Variables of GSP2 have the combined characteristics of signals and variables of VHDL. Variable assignment statements of GSP2 that have no propagation delay(PROP) act like a variable assignment of VHDL in that a variable which is in the left side of the assignment statement takes on the value of a right side expression immediately. A variable assignment statement that has PROP clause, on the other hand, behaves like a signal assignment of VHDL which has a time dimension. Variables are global to a module and no local variables can be declared in GSP2.

Table 9. Comparison of Communication Mechanisms

	COMMUNICATION MECHANISMS	
	INTRA-	INTER-
GSP2	global variables (no local variables)	port(PIN) (in, out, inout)
VHDL	signals	port (in, out, inout, buffer, linkage)
HHDL	global variables, internal signals	port (inward, outward, bothways)
ISP'	global variables (states, memories)	port (in, out, inout)
TI's HDL	global variables	port (I/O signal) (in, out, inout)

3.7 INSTANTIATION AND INTERCONNECTION

In chip level modeling, a chip is described as single entity using a behavioral description. So it seems unnecessary for a chip level language to contain explicit structural information inside the description of a chip. However, from a system designer's point of view, a chip is regard as a primitive element. Thus, a chip level hardware description language should be able to connect these primitive elements together. In this section, instantiation refers to the creation of instance which is previously declared or described and interconnection refers to the connection of instantiated elements.

3.7.1 Organization of Interconnection Description

There are two different approaches in HDLs to specify the interconnection of the elements. One approach is that a language has interconnection constructs within itself. VHDL and TI's HDL follow this approach. Another approach is that an HDL is used only to describe the behavior of an entity, and a separate language or a connection description file is used to specify the interconnection of previously described entities. HHDL, ISP', GSP2 and GSP are included in this category. Usually, simulation oriented languages employ the latter approach to ease the implementation of a simulator. But a

hardware description language is used not only for simulation, but for documentation purpose. In this regard, it is desirable that a language contain structural information within itself.

3.7.2 Description of Regular Structures

Hardware components often exhibit some degree of regularity in their structure, i.e., they may consist of multiple replications of a single component or connection pattern. For example, a 16 bit shift register may be composed of 16 flip-flops connected in sequence. It is tedious and error prone to use 16 component instantiations to describe such a structure. To support this kind of situation, VHDL provides an iterative 'generate' statement to more easily create a regular structure and specify its interconnection. Often, the boundaries of a regular structure exhibit slightly different structural characteristics than the rest of the structure. The first and last flip-flops of a register, for instance, may be connected differently than the inner flip-flops. To describe this situation, VHDL has a conditional generate statement. This form of the generate statement performs a macro expansion in the same way as the iterative generate, but in this case the enclosed statements are either expanded or not, based on the value of a condition.

3.7.3 Parameterized Instantiation of a Component

Hardware designers use multiple instances of the same component in their designs. Instances may differ in the names assigned to their I/O ports or their parametric information such as inherent timing delays. This parameterized instantiation construct enables one to represent classes of functionally and structurally identical components differing in timing characteristics. Also, this improves readability and reduces the size and complexity of a description.

VHDL provides generic declaration constructs that allow referencing design entities to specify values to be used as constants within a design entity. These parameters may be used to describe the size, environmental characteristics, timing characteristics, etc. of the design entity. Such information may be strictly documentation, or may be used in the calculation of propagation delays or other characteristics of the outputs of a design entity.

ISP' supports parameterization constructs simlar to VHDL but in a different way. Instantiation and interconnection of modules in ISP' are specified in a topology file. A composite, which is a collection of ISP' modules grouped together for the purpose of replacing a single module, can be declared within a topology file. The 'composite' is similar to a de-

sign entity of VHDL, which contains only component declarations and instantiations. Parameterization of a description in ISP' is through composite parameters. When a composite is instantiated, composite parameters provide text substitution within the declaration of the composite body. The declaration of the composite body may contain time delay. This time delay operates as a timing scaling factor which is applied to the time delay of all elementary modules internal to the composite. Consequently, parameterization of a description in ISP' is applicable to the composite level not to the elementary module level. A collection of ISP' modules grouped together for the purposes of replacing a single module is called a composite.

3.7.4 Bus Description

In a hardware, data paths that are driven by multiple sources are handled differently than single source data paths. In some cases, such as buses in which only one driver at a time is operating, tristate drivers are typically used in order to avoid confusion of signals. In other cases, such as 'wired and' and 'wired or' connections in which any number of drivers may be operating at a time, the behavior of such connections is identical to well understood logic functions, and they are used accordingly. Therefore, it is important that HDLs describe behavior of buses in a well defined manner.

VHDL provides generalized bus description capability [6]. That is, the language itself does not have pre-defined bus functions. It only provides a mechanism to describe the behavior of particular type of buses. For a signal to be a bus, it must be declared within an associated 'bus resolution function'. Such a function takes on arbitrary number of inputs of a given type and returns a single output value of the same type. This function is invoked whenever the bus with which it is associated receives new values.

ISP' and GSP2, on the other hand, have built in bus description constructs. In ISP', 'out' and 'bidirect' ports which are connected to a a bus can be declared as 'and' or 'or' using port signal attributes. If output ports which have 'and' attributes are connected together within a topology file, it acts like a wired and bus in real hardware. A wired or bus can be described similarly. To make the bus description easy, ISP' provides 'connect', 'disconnect' and 'vconnect' statements. If the statement 'disconnect (port_name)' is executed, the corresponding port is logically disconnected from a bus and no longer contributes to the evaluation of a signal, and a value of the bus signal is re-evaluated at that point in time. A 'connect' statement reverses the action of 'disconnect' statement. Another statement 'vconnect' is similar to the 'connect (port_name, expression)' statement adding an expression which is assigned

to the named port. The assigned value is used to represent the port in the connected signal evaluation.

GSP2 also has the similar bus description constructs with ISP'. 'Wired and' and 'wired or' bus are explicitly defined in the connection file which specifies the interconnection between modules.

Even though 'wired and', 'wired or' and 'tristate' are the most frequently used bus implementation techniques, a chip level hardware description language should be able to adapt to a new technology. In this sense, VHDL's approach that the user can define his own bus function according to the specific busing technology is preferable. Table 10 compares the instantiation and interconnection features of the languages.

Table 10. Comparison of Interconnection and Instantiation

	organi-zation	regular-structure	parameteri-zation	bus description
VHDL	language	y	y	y
GSP2	connection file	n	n	y
GSP	connection file	n	n	n
HHDL	separate language	y	y	y
ISP'	topology file	y	y	y
TI's HDL	language	y	y	y

CHAPTER 4. COMPARATIVE SYSTEM MODELING

This chapter describes the example modeling of the processor system MARK2 with VHDL, GSP2, ISP, and HHDL. The first part of the chapter describes the function of each component of the Mark2 system. The second part describes the major comparison points of the descriptions written with different languages. Full descriptions of the Mark 2 system with four different languages are presented in Appendix A.

4.1 MARK2 SYSTEM

The model used are those of a simplified processor unit, a RAM unit, two 8212's for parallel I/O, a UART for serial I/O. The block diagram of the system is given in figure 18. There are five address lines (A0 - A4) that control chip selection for either memory access or I/O.

The chip select logic is incorporated into the model of the processor and is a function of the three high address lines (A2 - A4). Address line A4 is used to differentiate between I/O and memory access. When logic level of A4 is 0, a RAM is selected. Otherwise, one of I/O ports is selected. The amount of memory that addressable by the processor is limited to the low four address lines (A0 - A3), i.e, 16 bytes. Once

I/O access mode is selected by A4, A3 is used to select the type of I/O port. Logic level 0 on A3 indicates the selection of a parallel I/O and logic level 1 indicates the selection of a serial I/O port. The schematic diagram of Mark 2 system is shown in Figure 19 and the logic diagram used to implement the chip select function is given in Figure 19.

4.1.1 Processor Module (MARK 2)

The processor unit, designated the MARK 2, is similar to the late 1940's processor MARK 1[25] which had the capability of executing only seven instructions. The main difference between the two processor models is that the memory area on MARK 1 is modified to provide an external memory area (the RAM module) for the MARK 2. This requires that the MARK 2 provide for read/write signals and chip select functions.

The word length of the MARK 2 is 8 bit long which allows for the address length of 5 bits and instruction opcodes of 3 bits. The descriptions of the seven instructions are as follows:

INSTRUCTION	OPCODE	DESCRIPTION
JMP	000	absolute jump
JRP	001	jump relative to program counter
LDN	010	load 2's complement of operand into accumulator

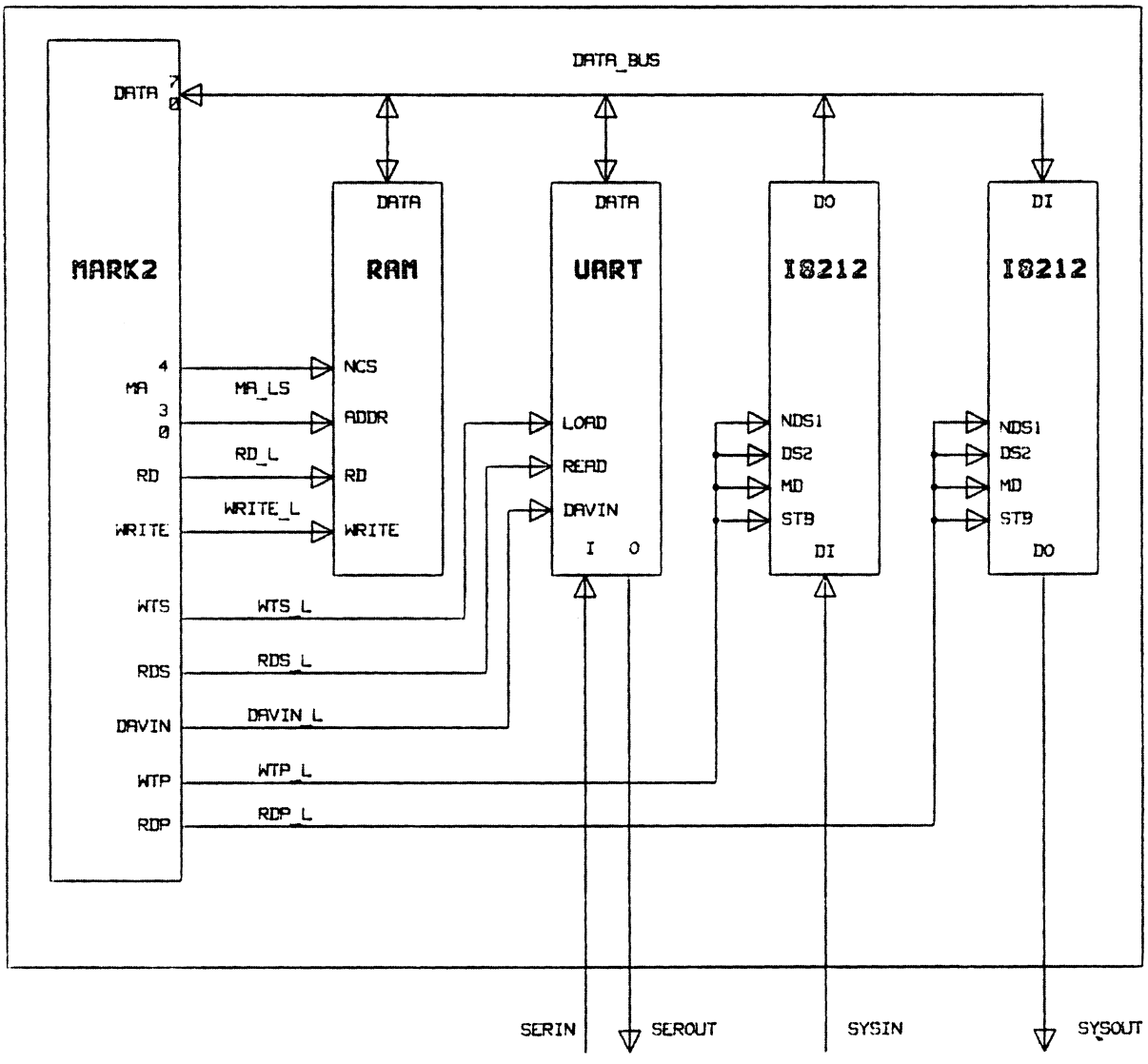
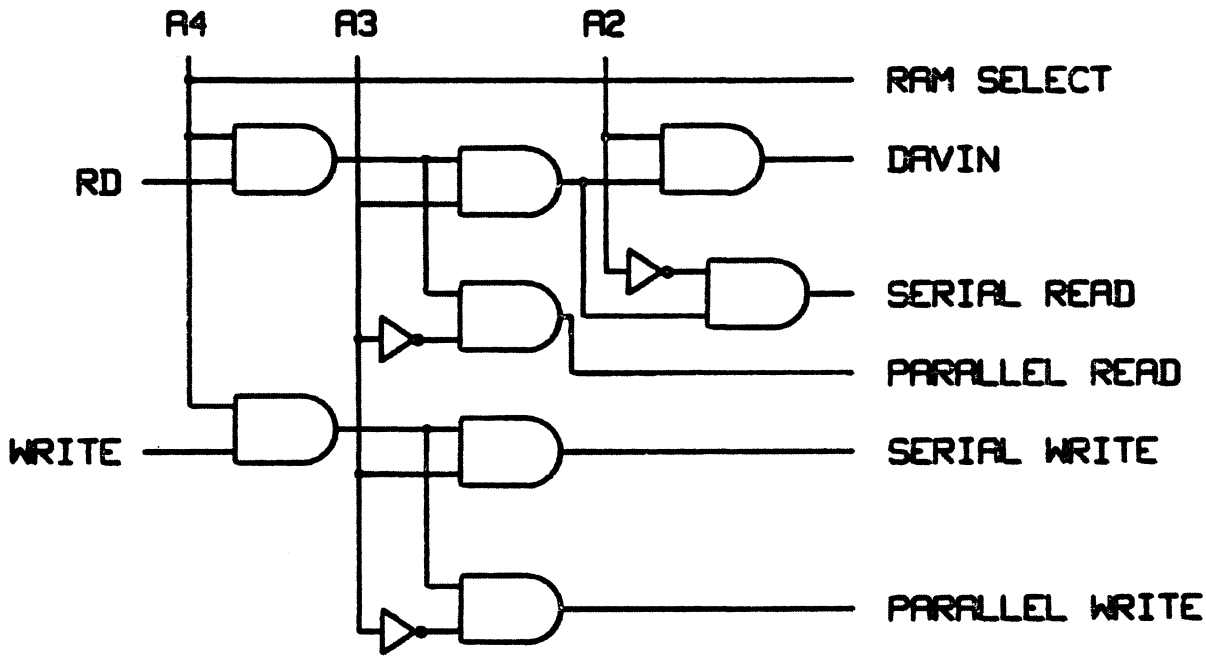


Figure 18. Mark 2 System



A4	A3	A2	A1	A0	
0	X	X	X	X	MEMORY REFERENCE
1	0	X	X	X	PARALLEL I/O'
1	1	0	X	X	SERIAL I/O
1	1	1	X	X	DAVIN

Figure 19. Decoding Logic and Address Map

STO	011	store the contents of the accumulator
SUB	100 or 101	subtract the operand from the accumu_ lator and place it in the accumulator
CMP	110	compare accumulator against zero; if less than zero, skip next instruction, else execute the next instruction
STP	111	stop_now

4.1.2 Serial I/O Module (UART)

An UART is used as a serial input, output port of the system. DATA input values are loaded into the output register OREG at the rising edge of the control input LOAD. After that, contents of OREG is transmited to the output (O) serially starting from the highest bit. Duration of the serial bit is controlled by the internal clock (CLK1).

Receiving of the serial input (I) is initiated by the first negative transition of I (starting bit), then the value of the input I are sampled every 100 ns until 8 samples are received. These values are stored in the input register IREG. Upon transition of the control input READ from 0 to 1, the contents of IREG are transmitted to the DATA output.

DAVIN input is used to check the status of the UART. Upon completion of receiving serial inputs, internal flag VDAV is

set to '00000001'. When DAVIN makes a 0 to 1 transition, The contents of VDAV are transmitted to the DATA output.

4.1.3 Parallel I/O (Intel 8212)

Two Intel 8212 are used as parallel I/O ports in the system. The chip is partitioned into two parts: the control and latch part. Internal signals S1, S2 and S3 are used to connect these two parts.

The I8212 has control inputs DS1, DS2, MD and STB. These inputs are used to control device selection, data latching, output buffer state and service request flip-flop. When DS1 is low and DS2 is high the device is selected. When MD is high the output buffers are enabled and the source of clock to the data latch is from the device selection logic. STB input is used as the clock to the data latch for the mode MD=0 and to synchronously reset the service request flip-flop (SRQ). SRQ is negative edge triggered. The SRQ flip-flop is used to generate and control interruptss in microcomputer systems. It is asynchronously set by the CLR input (active low).

4.1.4 RAM Module

The RAM module is activated only when the chip select pin (NCS) is low and either the read (RD) or write (WRITE) pin

goes high. When the chip has been selected and the the read line goes high, the data in the location referenced by the decoding of the address on the address lines is moved onto the output data lines (DATA) after delay of 150ns. To store data in the RAM memory, the WRITE line must be high while the NCS pin is low.

4.2 COMPARISON OF DESCRIPTIONS

4.2.1 Bus Descriptions

As can be seen in Figure 18, a bi-directional tri-state bus is used in the Mark 2 system. To describe this bus, a hardware description language should support at least a three valued logic system (1,0,Z). Only VHDL and HHDL provide the way to describe the multi-valued logic. In VHDL, a programmer defines his own value system and bus resolution functions which calculate the value of buses. The TSL package in VHDL description contains type declarations, type conversion functions and bus resolutions for the tri-state logic. By using the context statement like 'with package TSL; use TSL' at the heading of each description of a component, the TSL package is automatically referenced within the description.

HHDL, like VHDL provides a general mechanism to describe the multi- valued logic system. In addition, HHDL provides a predefined four valued logic system (unknown,high,low,z). Logical operations for the four valued system and type conversion functions such as a boolean to four valued logic (LOGTOBOOL) and a boolean to four valued logic (LOGFROMBOOL) conversion functions are contained in a modeling package named logpack.

GSP2 and ISP' do not support multi-valued logic systems. GSP2 provides wired-and or wired-or bus description capability. Therefore the data bus of the Mark 2 system is described as a wired-and bus instead of tri-state bus. When the outputs of components which are connected to data bus do not contribute to the value of the bus, the value of corresponding pin should be set to logic 1 by the model. Bus description with ISP' is done by using disconnect, connect, and vconnect statements. When the disconnect statement is executed within a module, a corresponding output pin does not contribute to the value of the bus any more. Execution of the vconnect statement outputs the value of a specified pin to the bus. Therefore new value of the bus is calculated after the vconnect statement is executed. the connect statement is used to receive the value from the bus.

4.2.2 Wait Mechanism

GSP2, ISP', and HHDL provide the wait construct which suspend the module execution for a specified amount of time. This wait construct is used to describe synchronous behavior of the processor Mark 2 to access memory. That is, after the processor issues a memory read signal, it waits for 150 time units before loading the value of the data bus into the data register (MD), since the memory needs a certain amount of time to output the contents of the storage location addressed

by the processor. Writing to the memory also takes time for the proper operation of the memory. Therefore, after sending a write memory signal to the memory, the processor holds the data bus value until the memory is supposed to finish writing. On the other hand, VHDL does not provide the wait construct. The analysis on the effect of VHDL's not including the wait construct is described in Chapter 3.3.1.3.

4.2.3 Description of I8212

The parallel I/O port I8212 is composed of 4 bit latches and the combinational logic that control the mode of operations. To model the combinational logic part, a process should respond to the change of every input signal to the combinational logic. In ISP', a process can respond to only one signal change. Therefore, there are many duplicated parts in the ISP' description of I8212. This duplicated part of descriptions may be written as a procedure but it is still cumbersome. Suppose that we are going to describe a two input AND gate of which A1, A2 are inputs and z is output. Then we should create two processes each of which responds to the value change of a singal as follows:

```
when AND1 (A1: change) :=
 ( Z := A1 and A2 )

when AND2 (A2: change) :=
 ( Z := A1 and A2 )
```

4.2.4 Operation on Bit Vectors

In HHDL, no operation is defined for bit vectors. Operations on bit vectors apply only to the elements not to bit vector itself. To supplement this kind of deficiency in manipulating the bit vectors which is inherited from PASCAL language, HHDL provides the REGISTER PACKAGE to ease the bit vector manipulation. This package includes various functions such as REGEQUAL which compares the contents of two bit vectors, arithmetic functions on bit vectors, shift, rotate functions, etc.. But there is no function that operates between bit vectors and immediate bit vector values. That is, if we are going to assign an immediate eight bit vector value '11111111' to the bit vector type variable A[0..7], we have to write a iterative loop as below:

```
FOR i=0 TO 7  DO
 A[i] := TRUE;
```

Otherwise, we have to convert the immediate bit vector value to an integer value by hand and again convert this integer to bit vector within the function as below:

```
A := REGFROMINT(-1,8)
```

Here, REGFROMINT is a function that convert an integer value to bit vector. The first argument of this function -1 is in-

teger value which corresponds to the 2's complement representation '1111111' and the second argument 8 is the bit length to be converted.

CHAPTER 5. OPTIMAL CONSTRUCTS

This chapter summarizes the analysis of various hardware description languages discussed in Chapter 3 and suggests the following optimum constructs for chip level hardware description languages.

1. Sequencing Mechanism

Nonprocedural languages describe the control circuit with varying degrees of explicity depending upon whether the description languages are block oriented nonprocedural languages or not [8]. Nonprocedural languages that are not block oriented are characterized as strongly nonprocedural. Strongly nonprocedural languages provide better hardware descriptions than procedural languages because the parallelism in hardware is modeled more accurately by these languages. Also, it is easier to describe a data flow model with strongly nonprocedural languages than with procedural languages. However, it is hard to describe new designs because at the initial phase of a chip design, control structures are not clearly defined. Procedural languages, on the other hand, are suitable for specification of new designs because they often originate as algorithms. Block oriented nonprocedural languages are

more flexible than strongly nonprocedural and procedural languages. A modeler can choose the style of a description. If the modeler wants to describe his model with procedural fashion, he may create a single block and describe his whole model within that block. If he wants to describe his model with strongly nonprocedural fashion, he may create a block for each activity so that every activity has a guard condition. Therefore block oriented nonprocedural languages are preferable over strongly nonprocedural and procedural languages.

2. Constructs to Express Guard Conditions

Powerful guard condition expression constructs are very important features in block oriented nonprocedural languages because they simplify the model description and clarify the intention of a modeler. Signal transition detection constructs such as CHANGE(X), FALL(X) and RISE(X) functions of GSP2 or corresponding VHDL signal attributes such as X'STABLE are essential to chip level hardware description languages. In addition to these constructs, higher level constructs which are found in Conlan might be convenient in higher level descriptions. Following is a list of the guard condition expressions of the Conlan language [21].

- change(#,s), fall(#,s), rise(#,s)

  The #th occurrence of each event triggers the block.

- count(event)

  This construct counts the occurrences of an event.

- every_from(#,simulation_time)

  This construct generate an event at every # time units starting from a specific simulation time.

3. Timing Constraints Description Constructs

In addition to the setup, hold time and minimum pulse width checking constructs, better timing checking constructs may be necessary in chip level modeling. Following is a list of the timing checking constructs found in FTL language [22].

- PSTABLE(CLK,DATA): This is a construct that checks for stable data during the duration of a pulse. If the second argument DATA signal changes value while CLK remains high, the timing violation is checked.

Figure 20(a) shows the timing relation between DATA and CLK.

- EDGE(CLK,B,C,A,+): This is a construct that checks for proper period and the duration of stable levels before and after the transition of the signal. When the first argument CLK signal makes a rising transition (+) the check is made to see whether CLK has been stable for B time units and C time units afterward. Also, the period between the current rising transition and the previous rising transition is checked. If the period is less than A time units, a flag is raised. Figure 20(b) shows the timing relations.

- RACE(A,B,R): This is a construct that checks for changes in two signals within a specified time. If signal A and B make transition within R time units, a flag is raised. Figure 20(c) shows the timing relation between A and B.

- ESTABLE2(X1,+,X2,-,DATA): This is a construct that checks for changes in two signals within a specified time. If the DATA signal make a transition between the rising transition (+) of X1 and falling transi-

tion (-) of X2, a flag is raised. Figure 20(d) shows the timing relations between X1, X2, and DATA.

4. Transport Delay Construct

- minimum, typical, and maximum propagation delays
- inertial delay

Transport delay is defined as the time required for the output to switch states, given an input [40]. In chip level modeling, minimal, typical, and maximal transport delay should be described for output signals from the description. At the electrical circuit level, at least certain amount of energy is required to turn on a transistor. This energy may be translated into specifying that a signal should be stable for a certain time after its transition for proper operation of a device. If a signal is changed again during this time, the effect of a transition of the signal can not be felt at the output of the device. That is, the device does not respond to input stimuli. Hardware description languages must be able to describe this phenomenon.

5. Retriggerable Module Suspension Constructs

- unconditional wait (ex., wait 100;)

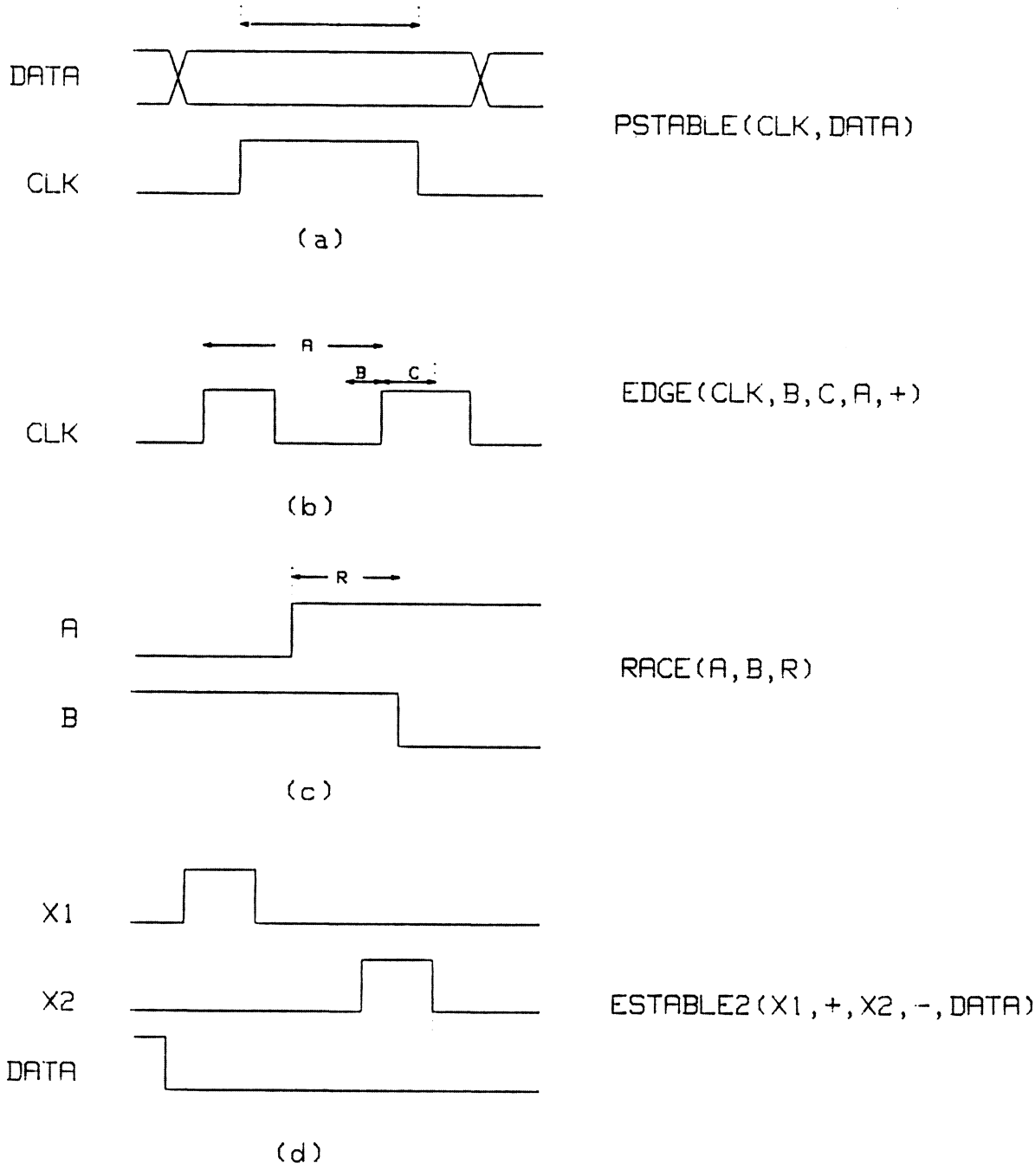


Figure 20. Timing Relations in Timing Constraint Constructs

- conditional wait (ex., waitfor x=1;)

As mentioned in chapter 3.3.1.3, the wait constructs are used to suspend the module execution. Depending on the semantics of the wait construct, a module that is being suspended by executing an wait statement may or may not be reactivated by another stimuli. If the module can be reactivated by another stimuli, we call it retriggerable wait. The retriggerable wait construct is more versatile than the nonretriggerable wait construct in chip level modeling as being analysed in chapter 3. Two forms of the wait construct may exist. One is the wait construct that delays the module execution for the specified amount of time (unconditional wait) and the other is the wait construct that delays the module execution until the condition is satisfied (conditional wait). These constructs are useful in describing the feedback delay model which is one of the generalized delay models discussed in chapter 3.

6. Data Abstraction and Operation

Data abstraction capability in chip level modeling should be provided at the level of high level programming languages. Following is the list of the VHDL data abstraction and operation constructs. While other HDLs may

partially cover the list, VHDL has the best organized data abstraction and operation facilities.

- strongly typed
- user defined types
- explicit type conversion
- package
- function
- procedure
- subfield declaration
- concatenation
- logical operators: and, or, nand, nor, xor, not
- relational operators; =, /=, <, <=, >, >=
- arithmetic operators; +, -, *, /, **, mod, rem, abs
- arithmetic operations for bit_vectors representing integers expressed in various forms (one's comp, two's comp, signed magnitude, unsigned magnitude) should be supported.

In addition to above constructs, shift and rotated operators for bit_vectors should be supported (VHDL does not have these operators).

7. Control Statements

- if_then_else
- case
- looping constructs

The above lists are the essential control constructs. The if_then_else construct provides for two way selection based on the value of a boolean expression. The case construct is for selecting one of multiple alternatives. A chip level HDL should also provide the iterative control constructs that are found in almost every high level programming languages.

8. Timing Modes

A chip level HDL should be able to describe both synchronous and asynchronous systems. Basically nonprocedural languages can implement both synchronous and asynchronous systems. Description of synchronous systems with nonprocedural languages may be done by using one of the input signals as an implicit clock. In this case, a description may be very difficult or unreadable. The guard construct in VHDL is ideal for describing synchronous systems. It might be convenient to be able to declare a explicit clock signal which has multiple phases. Therefore an event driven mechanism with the special constructs for the description of synchronous system such as the guard construct in VHDL and the clock signal declaration in HHDL can conveniently describe both synchronous and asynchronous systems.

9. Communication Constructs

The port construct found in VHDL, HHDL, ISP', GSP2, and GSP is ideal for chip level modeling in describing communication channels between describing entities. Since each port signal has a direction (in, out, inout) associated with it, a mistake in connection can be easily found before simulating the model.

10. Instantiation and interconnection constructs

Instantiation and interconnection constructs are used for the description of a model's structure. Following are the essential constructs to describe the complex structures one might encounter in chip level modeling.

- Constructs for the parameterized instantiation of a component
- Constructs for the description of regular structures
- Constructs for the bus description

11. Multi-valued Logic Description

In chip level modeling, multi-valued logic must be handled. At least four valued logic (0:logic 0, 1:logic 1, X:unknown, and Z:high impedance) must be supported by the language. That is, every logical operators must be de-

fined for the four valued logic mentioned above. In some languages, users can define his own logic system by using the user definable type facility. In this case, each user must write his own logical functions appropriate to his logic system.

CHAPTER 6. CONCLUSION

Nine different Hardware Description Languages were analysed and compared. Optimal constructs that are essential to chip level modeling were suggested. Block oriented nonprocedural languages were identified to be suitable for chip level modeling. Powerful guard expression constructs, timing constraint checking constructs, detailed propagation delay constructs and inertial delay specification construct were found to be essential to describe true input and output behavior of digital chips and systems. The level of data abstraction and operation that high level programming languages provide are also necessary to chip level modeling. The port construct as a communication mechanism between entities was found to be ideal to chip level modeling. Parameterized instantiation construct are necessary to manage the complexity of description. Finally, multi-level logic must be supported in chip level modeling.

APPENDIX A. EXAMPLE DESCRIPTIONS OF MARK 2 SYSTEM

A.1 VHDL DESCRIPTION

A.1.1 Package (Tri-State Logic)

```
package  TSL    is        -- from VHDL benchmark (VHDL 5.0)

--Define three valued logic
 type TRISTATE  is
  (
    'Z',    --high impedance
    '0',    --low level
    '1'     --high level
  );

------------------------------------------------------------
-- Declare an array type for functional behavior vectors:

  type TSL_FUNCTIONAL_BEHAVIOR_VECTOR  is
    array (bit) of TRISTATE;

 ------------------------------------------------------------
 -- Define the default Tie_off value for high impedance
      ('Z') signal
    constant TIE_OFF : BIT :='1';

  ------------------------------------------------------------
  --Define functions for converting between types of BIT
     and TRISTATE

 function TRISTATE_TO_BIT (INPUT: TRISTATE)  return BIT is
   begin
    case INPUT  is
     when 'Z' => return TIE_OFF;
     when '0' => return '0';
     when '1' => return '1';
    end case;
 end TRISTATE_TO_BIT;

 function BIT_TO_TRISTATE (INPUT: BIT)  return TRISTATE  is
  constant  TSL_VALUE: TSL_FUNCTIONAL_BEHAVIOR_VECTOR
     :=( '0'=>'0' , '1'=>'1' );
  begin
```

```
    return TSL_VALUE(INPUT);
 end BIT_TO_TRISTATE;

 --define a TRISTATE "bit_vector"

 type TRISTATE_VECTOR  is array (INTEGER range<>) of atomic
      TRISTATE_RESOLUTION  TRISTATE ;

 ------------------------------------------------------------
 -- Define the resolution mechanism needed for TRISTATE
 -- signals that are multiply driven

 function  TRISTATE_RESOLUTION (INPUT: TRISTATE_VECTOR)
   return TRISTATE   is
   variable RESOLVED_VALUE: TRISTATE:='Z';
    begin
     for I:=INPUT'LOW  to  INPUT'HIGH  loop
      if INPUT(I) /= 'Z'  then
       RESOLVED_VALUE:= INPUT(I);
       exit;
      end if;
     end loop;
    return RESOLVED_VALUE;
  end TRISTATE_RESOLUTION;
 ------------------------------------------------------------
 -- Define function that converts TRISTATE_VECTOR to
     BIT_VECTOR

function BITVEC_TO_TRIVEC (INPUT: BIT_VECTOR)
  return TRISTATE_VECTOR  is
  variable CONVERTED_VALUE: TRISTATE_VECTOR(INPUT'LEFT downto
         INPUT'RIGHT);
   begin
    for I:= INPUT'LOW to INPUT'HIGH  loop
     CONVERTED_VALUE(I):= TRISTATE_TO_BIT(INPUT(I));
    end loop;
   return CONVERTED_VALUE;
end BITVEC_TO_TRIVEC;

 ------------------------------------------------------------
 -- Define a function that converts TRISTATE_VECTOR to
     BIT_VECTOR

 function TRIVEC_TO_BITVEC(INPUT:TRISTATE_VECTOR)
    return BIT_VECTOR   is
    variable CONVERTED_VALUE : BIT_VECTOR(INPUT'LEFT downto
          INPUT'RIGHT);
    begin
     for I:=INPUT'LOW to INPUT'HIGH  loop
      CONVERTED_VALUE(I) := BIT_TO_TRISTATE(INPUT(I));
     end loop;
```

```
    return CONVERTED_VALUE;
  end TRIVEC_TO_BITVEC;
end TSL;
```

A.1.2 System Interconnection

```
with package TSL; use TSL;
entity MARK2_SYS
  (SYSOUT: out TRISTATE_VECTOR;
   SYSIN : in  TRISTATE_VECTOR;
   SEROUT: out BIT;
   SERIN : in  BIT )    is
end MARK2_SYS;

architecture  CHIP_LEVEL  of MARK2_SYS  is
  block

   --LOCAL COMPONENT DECLARATIONS

   component  MARK2  port (DATA: inout TRISTATE_VECTOR;
                           MA: out BIT_VECTOR ;
                           RD,WRITE,RDS,RDP,WTS,WTR: out BIT;
                           DAVIN: out BIT);

   component  RAM    port (DATA: inout TRISTATE_VECTOR;
                           ADDR: in BIT_VECTOR ;
                           RD,WRITE,NCS: in BIT);

   component  UART   port (DATA: inout TRISTATE_VECTOR;
                           I,LOAD,READ: in BIT ;
                           O: out BIT;
                           DAVIN: in BIT);

   component  I8212  port (DI: in  TRISTATE_VECTOR;
                           DO: out TRISTATE_VECTOR;
                           NDS1,DS2,MD,STB,CLR: in BIT;
                           INT: out BIT);

 signal DATA_BUS : TRISTATE_VECTOR(7 downto 0);
 signal MA_LS    : BIT_VECTOR(4 downto 0);
 signal RD_L, WRITE_L, RDS_L, RDP_L: BIT;
 signal WTS_L, WTP_L, DAVIN_L: BIT;

 begin

   ----Component instantiation statements

  CPU:    MARK2  port (DATA_BUS, MA_LS, RD_L, WRITE_L,
                       RDS_L, RDP_L, WTS_L, WTP_L, DAVIN_L);

  MEM:    RAM    port (DATA_BUS, MA_LS(3 downto 0), RD_L,
                       WRITE_L, MA_LS(4) );

  SER:    UART   PORT (DATA_BUS, WRITE_L, RD_L, SEROUT,
                       DAVIN_L);
```

```
  PARIN:  I8212  port (SYSIN, DATA_BUS, RDP_L, RDP_L, RDP_L,
                       RDP_L, open, open);

  PAROUT: I8212  port (DATA_BUS, SYSOUT, WTP_L, WTP_L, WTP_L,
                       WTP_L, open, open);

  end block;
end MARK2_SYS;
```

A.1.3 Processor (MARK 2)

```
with package TSL; use TSL;
entity  MARK2
  (DATA: inout TRISTATE_VECTOR;
   MA: out BIT_VECTOR;
   RD,WRITE,RDS,RDP,WTS,WTP,DAVIN: out BIT)      is

end MARK2;

architecture  BEHAVIOR of  MARK2    is
 block
  signal  CLK,RD_MEM1,RD_MEM2,WR_MEM,stop_now: BIT;
  signal  IR,ACC :BIT_VECTOR(7 downto 0);
  signal  PC :BIT_VECTOR(4 downto 0);
  signal  COMB :BIT_VECTOR(4 downto 0);
  signal  NEXT1, NEXT2: BIT;
  begin
   process begin
    CLK <= not CLK after 500ns;       --simulation starts
   end process;

   CLK <= not CLK  after 500ns;       --internal clock

   process (CLK, stop_now)
    begin
     if  not stop_now'STABLE then
      disable CLK;                     -- stop_now
     else
      MA <= PC;
      COMB(4 downto 2) <= PC(4 downto 2);
      RD <= '1';  WRITE <= '0';
      COMB(1) <= '1'; COMB(0) <='0';
      RD_MEM1 <= not RD_MEM1 after 150ns;  -- wait for data
     end if;
   end process;

   process (RD_MEM1)
    begin                                  -- instruction
     IR <= TRIVEC_TO_BITVEC(DATA);         --    fetch
     RD <= '0';
     COMB(1) <= '0';
     PC <= ADD(PC,"00001");                -- increment PC
     NEXT1 <= not NEXT1 after 10ns;
   END PROCESS;

   process (NEXT1)
    begin
     case Intval(IR(2 downto 0))      is      -- decoding
      when 0 => PC <= IR(7 downto 3);         -- jmp
      when 1 => PC <= ADD(SUB(PC,"00001"),IR(7 downto 3));
```

```
      when 2 => RD_MEM2 <= not RD_MEM2;        --ldn
      when 3 => WR_MEM  <= not WR_MEM;         --sto
      when 4 | 5 => RD_MEM2 <= not RD_MEM2;    --sub
      when 6 => if ACC(7)='1'  then            --cmp
                  PC <= ADD(PC,"00001");
                end if;
      when 7 => stop_now <= not stop_now;      --stop_now
     end case;
   end process;

   process (RD_MEM2)
    begin
     MA <= IR(7 downto 3);
     COMB(4 downto 2) <= IR(7 downto 5);
     RD <= '1'; WRITE <= '0';
     COMB(1) <= '1'; COMB(0) <= '0';
     NEXT2 <= not NEXT2 after 150ns;       --wait for data
   end process;

   process (NEXT2)
    variable MD: BIT_VECTOR(7 downto 0);
    begin
     MD:=TRI_VEC_TO_BITVEC(DATA);
     RD <= '0';
     COMB(1) <= '0';
     if IR(2 downto 0)="010"   then
      ACC <= ADD(not MD,"00000001");
     else
      ACC <= ADD(ACC,ADD(not MD,"00000001"));
     end if;
   end process;

   process (WR_MEM)
     begin
      MA <= IR(7 downto 3);
      COMB(4 downto 2) <= IR(7 downto 5);
      DATA <= BITVEC_TO_TRIVEC(ACC);
      WRITE <= '1','0' after 220ns;
      COMB(0) <= '1','0' after 220ns;
   end process;
--
--   Decoding logic description
--
   RDS <= COMB(3) and COMB(1) and COMB(4) and not COM(2);
   RDP <= not COMB(2) and COMB(3) and COMB(1);
   WTS <= COMB(0) and COMB(2) and COMB(3);
   WTP <= not COMB(2) and COMB(3) and COMB(0);
   DAVIN <= COMB(4) and COMB(3) and COMB(2) and COMB(1);
 end block;
end BEHAVIOR;
```

A.1.4 Serial I/O (UART)

```
with package TSL; use TSL;
entity UART
--ports
 (DATA: inout TRISTATE_VECTOR;
  I: in BIT;
  LOAD,READ: in BIT;
  O: out BIT;
  DAVIN: in BIT  )          is
end uart;

architecture    BEHAVIOR of UART      is
  block
    signal  CLK1,CLK2: BIT;
    begin
      process (LOAD,CLK1)
        variable OCNTR: integer;
        variable OREG : BIT_VECTOR(8 downto 1);
        begin
         if not LOAD'stable  and LOAD='1'  then
           OREG:=TRIVEC_TO_BITVEC(DATA);
           OCNTR:=8;
           O <= '0';
           CLK1 <= not CLK1 after 100ns;
         end if;
         if not CLK1'stable   then
          if OCNTR /= '0'  then
           O <= OREG(OCNTR);
           OCNTR:= OCNTR-1;
           CLK1 <= not CLK1 after 100ns;
          else
           OCNTR:=8
           O <= '1';
          end if;
         end if;
      end process;

      process (I,READ,CLK2, DAVIN)
        variable IREG: BIT_VECTOR(8 downto 1);
        variable ICNTR: integer;
        variable VDAV: BIT_VECTOR(8 downto 1) :="00000000";
        begin
         if not I'stable and I='0'  then
          disable I;
          ICNTR:=8;
          CLK2 <= not CLK2 after 150ns;
         end if;
         if not CLK2'stable  then
          if ICNTR /= 0  then
           IREG(ICNTR) := I;
```

```
            ICNTR:= ICNTR-1;
            CLK2 <= not CLK2 after 100ns;
           else
            ICNTR:=8;
            VDAV:="00000001";
            enable I;
           end if;
          end if;
          if not READ'stable   then
           if READ='1'  then
            DATA <= BITVEC_TO_TRIVEC(IREG) after 20ns;
            VDAV:="00000000";
           else
            DATA <= "ZZZZZZZZ" after 20ns;
           end if;
          end if;
          if not DAVIN'stable   then
           if DAVIN='1' then
            DATA <= BITVEC_TO_TRIVEC(VDAV);
           else
            DATA <= "ZZZZZZZZ" after 20ns;
           end if;
          end if;
      end process;
   end block;
end BEHAVIOR;
```

A.1.5 Parallel I/O (Intel 8212)

```
with package TSL; use TSL;
entity I8212
  (DI: in  TRISTATE_VECTOR;
   DO: out TRISTATE_VECTOR;
   NDS1,DS2,MD,STB,CLR: in BIT;
   INT: out BIT)                    is

end I8212;

architecture BEHAVIOR  of  I8212    is

 block
    signal  S1,S2,S3: BIT;
  begin
  block( S1='1' )
     signal Q: BIT_VECTOR (7 downto 0);
   begin
    Q <= memoried TRIVEC_TO_BITVEC (DI);
    DO<= BITVEC_TO_TRIVEC (Q)  when  S3  else
         "ZZZZZZZZ";
    Q <= "00000000"  when  not S1 and not CLR else
         Q;
    S2 <= S1 nor CLR;
   end block;
  block
     signal SRQ:BIT;
   begin
    S1 <= not DS1 and DS2  when  MD='1' else
          STB;
    S3 <= (not DS1 and DS2) or MD;
    SRQ<= '1'  when (not S2  or (not DS1 and DS2)) else
          '0'  when STB='0' else
          SRQ;
    INT <= not SRQ nor (not DS1 and DS2);
  end block;
 end block;
end BEHAVIOR;
```

A.1.6 RAM

```
with package TSL; use TSL;
entity  RAM
  (DATA: inout TRISTATE_VECTOR;
   ADDR: in BIT_VECTOR;
   RD,WRITE,NCS: in BIT)              is
end RAM;

architecture BEHAVIOR of RAM   is
 block
  begin
   process (NCS)
    subtype BYTE is BIT_VECTOR(7 downto 0);
    type MEMORY is array(0 to 15) of BYTE;
    variable MEM: static MEMORY;
     begin
      if NCS='0'  then
       if RD='1'  then
         DATA<=BITVEC_TO_TRIVEC( MEM(Intval(ADDR)) )
            after 150ns;
       elsif WRITE='1'  then
         MEM (Intval(ADDR)) :=TRIVEC_TO_BITVEC (DATA);
       end if;
      else
       DO <= "ZZZZZZZZ"  after 20ns;
      end if;
  end process;
 end block;
end BEHAVIOR;
```

A.2 GSP2 DESCRIPTION

A.2.1 Processor (MARK 2)

```
MODULE mark2

LITERALS
  ! instruction register fields !
  opcode = '0|3'; address = '3|5'; sign_bit = '7|1';
  ! instruction codes !
  jmp = '#B000'; jrp = '#B001'; ldn = '#B010';
  sto = '#B011'; sub1 = '#B100'; sub2 = '#B101';
  cmp = '#B110'; stp = '#B111';
  ! time parameter !
  read_period = '150';
  write_period = '200';
  clock_width = '500'

PINS
  start[0|1] : INPUT;
  data_lines[0|8] : BIDIRECT;
  addr_lines[0|5],rd[0|1],wt[0|1],rds[0|1],
  rdp[0|1],wts[0|1],wtp[0|1],davin[0|1] : OUTPUT
DECLARE
  ir[0|8],acc[0|8],pc[0|5],md[0|8],mar[0|5],
  rd_signal[0|1],wt_signal[0|1],clock[0|1] :REGISTER

PROCEDURE @select(COPY addr[0|5] : REGISTER)
  BEGIN
    rd := rd_signal PROP 10;
    rds := @AND(@NOT(addr[2|1],@AND(addr[3|1],
             @AND(addr[4|1],rd_signal)))  PROP 10;
    rdp := @AND(@NOT(addr[3|1]),
                @AND(addr[4|1],rd_signal)) PROP 10;
    wt := wt_signal PROP 10;
    wts := @AND(addr[3|1],@AND(addr[4|1],wt_signal))
           PROP 10;
    wtp := @AND(@NOT(addr[3|1]),
                @AND(addr[4|1],wt_signal)) PROP 10;
    davin := @AND(addr[2|1],@AND(addr[3|1],@AND(rd_signal,
               addr[4|1])))   PROP 20
  END

PROCEDURE @read_mem(COPY addr[0|5]: REGISTER;
                    MOD data[0|8]: REGISTER)
  BEGIN
    addr_lines := addr;
```

```
    rd_signal := #B1;
    @select(addr);
    rd_signal := #B0 PROP read_period;
    WAIT read_period;
    @select(addr);
    data := data_lines
  END

PROCEDURE @write_mem(COPY addr[0|5],data[0|8]: REGISTER)
  BEGIN
    addr_lines := addr;
    data_lines := data;
    wt_signal := #B1;
    @select(addr);
    wt_signal := #B0 PROP write_period;
    WAIT write_period;
    @select(addr);
    data_lines := #HFF PROP 10
  END

EVENT clock ON @RISE(start);
 WHILE start  DO
  BEGIN
   clock := #B1 PROP clock_width / 2;
   clock := #B0 PROP clock_width;
   WAIT clock_width
  END
 ENDWHILE

EVENT cycle ON @RISE(clock);
  BEGIN
    @read_mem(pc,ir);
    mar := ir[address];
    pc := @ADD(pc,#B00001);
    WAIT 50;
    CASE ir[opcode] OF
      jmp: pc := mar;
      jrp: pc := @ADD(@SUB(pc,#B00001),mar);
      ldn: BEGIN
             @read_mem(mar,md);
             acc := @NEG(md)
           END;
      sto: @write_mem(mar,acc);
      sub1,sub2:
           BEGIN
             @read_mem(mar,md);
             acc := @ADD(acc,@NEG(md))
           END;
      cmp: IF acc[sign_bit] = #B1
             THEN
               pc := @ADD(pc,#B00001)
```

```
            ENDIF;
        stp: BEGIN
               WRITE TERMINAL 'processor halted';
               WRITE LOG 'processor halted';
               PAUSE
             END
      ENDCASE
    END

ENDMOD
```

A.2.2 RAM

```
MODULE ram

PINS
  rd[0|1],wt[0|1],ram_addr[0|5] : INPUT;
  ram_data[0|8] : BIDIRECT

DECLARE
  mem[0|5,0|8] : MEMORY

EVENT read_mem ON @AND(@NOT(ram_addr[4|1]),@RISE(rd));
  ram_data := mem[ram_addr,0|8] PROP 100

EVENT write_mem ON @AND(@NOT(ram_addr[4|1]),@RISE(wt));
  mem[ram_addr] := ram_data PROP 100

EVENT float_ram ON @AND(@NOT(ram_addr[4|1]),@FALL(rd));
  ram_data := #HFF PROP 10

ENDMOD
```

A.3.3 Serial I/O (UART)

```
MODULE uart

PINS
  si[0|1],rds[0|1],wts[0|1],davin[0|1] : INPUT;
  uart_data[0|8] : BIDIRECT;
  so[0|1] : OUTPUT VALUE #B1

DECLARE
  preg[0|8],vdav[0|8] : REGISTER;
  icntr: INTEGER

EVENT load_reg ON @RISE(wts);
  BEGIN
   preg := uart_data;
   so := #B0 PROP 90
  END

EVENT ser_out ON @FALL(so);
  BEGIN
    LOOP icntr := 1 TO 8 BY 1 DO
      BEGIN
        WAIT 100;
        preg := @SHIFT(preg,1);
        so := CARRY
      END
    ENDLOOP;
    so := #B1 PROP 100
  END
EVENT read_bit ON @FALL(si);
  BEGIN
    WAIT 50;
    LOOP icntr := 1 TO 8 BY 1 DO
      BEGIN
        WAIT 100;
        preg[0|1] := si;
        IF icntr < 8
          THEN
            preg := @SHIFT(preg,-1)
          ELSE
            vdav := #B00000001;
        ENDIF
      END
   ENDLOOP
  END

EVENT read_data ON @RISE(rds);
  uart_data := preg PROP 40

EVENT uart_float ON @FALL(rds);
```

```
  uart_data := #HFF PROP 5

EVENT davin_read ON @RISE(davin);
  uart_data := vdav PROP 20

ENDMOD
```

A.2.4 Parallel I/O (Intel 8212)

```
MODULE i8212

PINS
  nds1[0|1],ds2[0|1],md[0|1],stb[0|1],
   di[0|8],clr[0|1] : INPUT;
  do[0|8],nint[0|1] : OUTPUT

DECLARE
  s1[0|1] : REGISTER VALUE 0;
  q[0|4] : REGISTER

PROCEDURE  device_select
 BEGIN
  s1 := @OR(@AND(@NOT(nds1),ds2,md), @AND(@NOT(md),STB));
  IF s1
   THEN
    q := di
  ENDIF;
  IF @AND(@NOT(nds1),ds2)
   THEN
    do := q  PROP 20
  ENDIF;
  IF @AND(@AND(@NOT(nds1),ds2),@NOT(md)
   THEN
    do := #HFF PROP 20
  ENDIF;
 END

PROCEDURE intr_cont
 BEGIN
  IF @AND(@NOT(nds2),ds2)
   THEN
    int := #B0  PROP 20
  ENDIF
  IF @AND(@NOT(@AND(@NOT(nds1),ds2),@NOT(s1),clr)
   THEN
    int := #B1  PROP 20
  ENDIF
 END

EVENT nds1_chng ON @CHANGE(nds1);
 BEGIN
  device_select;
  intr_cont
 END

EVENT ds2_chng ON @CHANGE(ds2);
 BEGIN
  device_select;
```

```
  intr_cont
 END

EVENT md_chng ON @CHANGE(md);
  device_select

EVENT stb_chng ON @CHANGE(stb);
 BEGIN
  s1 := @OR(@AND(@NOT(nds1),ds2,md),@AND(@NOT(md),stb));
  IF s1
   THEN
    q := di
  ENDIF;
  IF @AND(@NOT(nds2),ds2)
   THEN
    int := #B0
  ENDIF;
  IF @AND(@NOT(@AND(@NOT(nds1),ds2)),@NOT(s1),clr)
   THEN
    int := #B1 PROP 20
  ENDIF;
  IF @AND(stb,@NOT(@OR(@AND(@NOT(nds1),ds2)),
          @AND(@NOT(clr),s1)))
   THEN
    int := #B1 PROP 20
  ENDIF;
 END

EVENT clr_chng ON @CHANGE(clr);
 IF @AND(@NOT(s1),@NOT(clr))
  THEN
   q := #B00000000
 ENDIF

EVENT data_chng ON @CHANGE(di);
 IF s1
  THEN
   q := di;
 ENDIF

EVENT q_chng ON @CHANGE(q);
 IF @OR(md,@AND(@NOT(nds1),ds2))
  THEN
   do := q  PROP 20
 ENDIF

ENDMOD
```

A.3 ISP' DESCRIPTION

A.3.1 Processor (MARK 2)

```
port

DATA<7:0>'and :bidirectional,
MA<4:0> :output,
RD :output,
WRITE :output,
RDS :output,
RDP :output,
WTS :output,
WTP :output,
DAVIN :output,
CLK: input;

macro

OPCODE:= 2:0&,
ADDRESS:= 7:3&,
SIGN_BIT:= 7&,

JMP:=0&, JRP:=1&, LDN:=2&, STO:=3&, SUB1:=4&, SUB2:=5&,
CMP:=6&, STP:=7&,

READ_PERIOD:=150&, WRITE_PERIOD:=200;

state

IR<7:0>, ACC<7:0>, PC<4:0>, MD<7:0>, MAR<4:0>,
RD_SIGNAL, WT_SIGNAL(0b0);


procedure SELECT ( ADDR<4:0> ) :=
 (
  RD = RD_SIGNAL;
  delay 10;
  RDS = ADDR<4> and ADDR<3> and RD_SIGNAL and not ADDR<2>;
  RDP = not ADDR<3> and ADDR<2> and RD_SIGNAL;
  WT = WT_SIGNAL;
  WTS = ADDR<4> and ADDR<3> and WT_SIGNAL;
  WTP = not ADDR<3> and ADDR<4> and WT_SIGNAL
 )

procedure READ_MEM ( ADDR<4:0> ) :=
 (
```

```
  ADDR_LINES = ADDR;
  RD_SIGNAL = Ob1; next;
  SELECT (ADDR);
  delay READ_PERIOD;
  RD_SIGNAL = Ob0;
  SELECT (ADDR)
 )

procedure WRITE_MEM ( ADDR<4:0>, DATA<7:0> ) :=
 (
  ADDR_LINES = ADDR;
  DATA_LINES = DATA;
  WT_SIGNAL = Ob1; next;
  SELECT (ADDR);
  delay WRITE_PERIOD;
  WT_SIGNAL = Ob0; next;
  SELECT (ADDR)
 )


when CYCLE (CLK:lead) :=
 (
  READ_MEM(PC,IR);
  MAR = IR<ADDRESS>;
  PC = PC + 1 ext 5;
  next;
  case IR<OPCODE>
   JMP: PC = MAR
   JRP: PC = (PC - 1 ext 4 + MAR)
   LDN: ( READ_MEM(MAR);
          MD = DATA_LINES;
          ACC = -MD )
   STO: WRITE_MEM(MAR,ACC)
   SUB1,SUB2: ( READ_MEM(MAR);
                MD = DATA_LINES;
                ACC = -MD + ACC )
   CMP: ( if ACC<SIGN_BIT> = Ob0
           PC = PC + 1 )
   STP:
  esac;
 )
```

A.3.2 RAM

```
port

RD: input,
WT: input,
ADDR<4:0>: input,
DATA<7:0>'and: disconnect;

state

MEM[0:31]<4:0>


when READ_MEM (RD:lead(not ADDR<4>) :=
 (
  delay 100;
  DATA = MEM[ADDR ext 6]<7:0>;
  next;
  connect(DATA);
 )

when WRITE_MEM (WT:lead(not ADDR<4>) :=
 (
  delay 100;
  MEM[ADDR ext 6]<7:0> = DATA;
 )

when FLOAT_RAM (RD:trail(not ADDR<4>) :=
 (
   delay 10;
   disconnect DATA
 )
```

A.3.3 Serial I/O (UART)

```
port

DATA<7:0>'and: bidirect,
I: input,
LOAD: input,
READ: input,
DAVIN: input,
O: output;

state

PREG<7:0>, ICNTR<3:0>, VDAV<7:0>;


when LOAD_REG (LOAD: lead) :=
 (
  PREG = DATA;
  delay 90;
  SO = Ob0;
 )

when SER_OUT (O: trail) :=
 (
  while inctr<>0
   ( delay 100;
     PREG *: logical 1;
     ICNTR = ICNTR-1; )
  ICNTR = 8 ext 4;
  O = Ob1;
 )

when READ_BIT (I: trail) :=
 (
  while icntr <> 0
   ( delay 100;
     preg<0> = I;
     next;
     preg *: logical 1;
     ICNTR = ICNTR-1; )
  ICNTR = 8;
  vconnect (VDAV,1);
 )

when READ_DATA (READ: lead) :=
 (
  vconnect (DATA,PREG);
  vdav = 0;
 )
```

```
when READ_FALL (READ: trail) :=
 (
  disconnect (DATA);
 )

when DAVIN_RISE (DAIN:lead) :=
 (
  vconnect (DATA,VDAV);
 )

when DAVIN_FALL (DAVIN: trail) :=
 (
  disconnect (DATA);
 )
```

A.3.4 Parallel I/O (Intel 8212)

```
port

NDS1: input,
DS2: input,
MD: input,
STB: input,
CLR: input
DI<7:0>: input,
NINT: output;
DO<7:0>: output;

state

S1,Q<7:0>;

when NDS1_CHGN (NDS1) :=
 (
  S1 = (not NDS1 and DS2 and MD) or (not MD and STB);
  next;
  if S1 = 1
   Q = DI;
  if not NDS1 and DS2
   vconnect (DO,Q);
  if not(not NDS1 and DS2) and not MD
   disconnect (DO);
  if not NDS2 and DS2
   INT = 0;
  if not(not NDS1 and DS2) and not S1 and CLR
   INT = 1;
 )

when DS2_CHGN (DS2) :=
 (
  S1 = (not NDS1 and DS2 and MD) or (not MD and STB);
  next;
  if S1 = 1
   Q = DI;
  if not NDS1 and DS2
   vconnect (DO,Q);
  if not(not NDS1 and DS2) and not MD
   disconnect (DO);
  if not NDS2 and DS2
   INT = 0;
  if not(not NDS1 and DS2) and not S1 and CLR
   INT = 1
 )

when MD_CHGN (MD) :=
 (
```

```
  S1 = (not NDS1 and DS2 and MD) or (not MD and STB);
  next;
  if S1 = 1
   Q = DI;
  if not NDS1 and DS2
   vconnect (DO,Q);
  if not(not NDS1 and DS2) and not MD
   disconnect (DO);
 )


when STB_CHGN (STB) :=
 (
  S1 = (not NDS1 and DS2 and MD) or (not MD and STB);
  next;
  if S1 = 1
   Q = DI;
  if not NDS2 and DS2
   INT = 0;
  if not(notNDS1 and DS2) and not S1 and CLR
   INT = 1;
  if STB and not((not NDS1 and DS2) or (not CLR and S1))
   INT = 1;
 )

 when CLR_CHNG (CLR) :=
 (
  if not (not S1 and not CLR)
   Q = 0 ext 8;
 )

 when DATA_CHNG (DI) :=
 (
  if S1=1
   Q = DI;
 )

 when Q_CHNG (Q) :=
 (
  if not NDS1 and DS2 or MD
   DO = Q;
```

A.4 HHDL DESCRIPTION

A.4.1 Processor (MARK 2)

```
MODULE mark2_system

  USE logpack;
      regpack;

  TYPE
      netlog = (unk, lo, hi, z);

  NETTYPE
      reg1bit  = REGISTER[0];
      reg4bit  = REGISTER[3..0];
      reg8bit  = REGISTER[7..0];
      four_state_net = ARRAY[7..0] OF netlogic;

COMPTYPE  mark2;

 OUTWARD  addr : reg8bit;
          rd,write,rds,rdp,wts,wtp,davin: reg1bit;

 BOTHWAYS data : four_state_net;

 INTERNAL clk : reg1bit;

  VAR
   ir,acc,md : REGISTER[7..0];
   pc,mar : REGISTER[4..0]
   rd_signal,wt_signal : reg1bit;

 PROCEDURE select (faddr: REGISER[4..0] )
  BEGIN
   ASSIGN rd_sigal TO rd  DELAY 10;
   ASSIGN NOT faddr[2] NOT faddr[3] NOT faddr[4] NOT rd_signal
          TO rds  DELAY 10;
   ASSIGN NOT faddr[3] AND faddr[4] AND rd_signal
          TO rdp DELAY 10;
   ASSIGN wt_signal TO wt  DELAY 10;
   ASSIGN faddr[3] AND faddr[4] AND wt_signal TO wts  DELAY 10;
   ASSIGN NOT faddr[3] AND faddr[4] AND wt_signal
          TO wtp  DELAY 10;
   ASSIGN faddr[2] AND faddr[3] AND faddr[4] AND rd_signal
          TO davin  DELAY 10;
  END
```

```
PROCEDURE read_mem (faddr: REGISTER[4..0]; VAR fdata[7..0])
 BEGIN
  ASSIGN faddr TO addr;
  ASSIGN TRUE TO rd_signal;
  select (faddr);
  ASSIGN FALSE TO rd_signal;
  WAITFOR TRUE DELAY 150;
  select (faddr);
  data := LOGTOBOOL (fdata);
 END

PROCEDURE write_mem (faddr: REGISTER[7..0],
                     fdata: REGISTER[7..0)
 VAR i: INTEGER
 BEGIN
  ASSIGN faddr TO addres;
  ASSIGN LOGFROMBOOL(fdata) TO data;
  wt_signal := TRUE;
  select(faddr);
  wt_signal := FALSE;
  WAITFOR TRUE DELAY 150;
  select(faddr);
  FOR i=7 DOWNTO 0
   ASSIGN z TO data[i]  DELAY 10;
 END

SUBPROCESS
 cycle:
  UPON clk CHECK clk  DO
   BEGIN
    read_mem(pc,ir);
    mar := ir[4:0]
    pc := REGADD(pc,REGFROMINT(1,5),REG2C);
    WAITFOR TRUE DELAY 50;
    CASE REGTOINT(ir[2:0]) OF
     0: pc := mar;
     1: pc := REGADD(REGSUB(pc,REGFROMINT(1,5),REG2C),
                     mar,REG2C);
     2: BEGIN
         read_mem(mar,md);
         acc := REGNEGATE(md,REG2C);
        END
     3: write_mem(mar,acc);
     4,5: BEGIN
           read_mem(mar,md);
           acc := REGADD(acc,REGNEGATE(md,REG2C),REG2C)
          END
     6: IF acc THEN
         pc := REGADD(pc,REGFROMINT(1,5),REG2C);
     7: BEGIN
```

```
          INHIBIT cycle;
          WRILELN (OUTPUT, 'processor halted');
         END

BEGIN
 ASSIGN TRUE TO clk  DELAY 250;
 REPEAT
  ASSIGN NOT clk TO clk  DELAY 250;
  UNTIL FALSE;
END
```

A.4.2 RAM

```
COMPTYPE  ram;

 INWARD  rd,write,ncs: reg1bit;
         addr: reg4bit;
 BOTHWAYS data: reg8bit;
 VAR
   mem: ARRAY[7..0] OF reg4bit;

 SUBPROCESS
  read_write:
   UPON NOT ncs CHECK ncs DO
    BEGIN
     IF rd THEN
      ASSIGN LOGFROMBOLL(mem[REGTOINT(addr)])
             TO data DELAY 150;
     IF write THEN
      mem(REGFROMINT(addr)) := LOGTOBOOL(data);
    END
  float:
   UPON ncs CHECK ncs DO
    FOR i=7 DOWNTO 0 DO
     ASSIGN z TO data(i) DELAY 20;

BEGIN

END
```

A.4.3 Serial I/O (UART)

```
COMPTYPE  UART;

 INWARD     i,load,read,davin: reglbit;
 OUTWARD    o: reglbit;
 BOTHWAYS   data: four_state_net;
 VAR
   preg: REGISTER[7..0];
   vdav: BOOLEAN;
   icnttr: INTEGER;

 SUBPROCESS
  load_reg:
   UPON load CHECK load DO
    BEGIN
     preg := REGOTBOOL(data);
     ASSIGN FALSE TO o DELAY 90;
    END
  serial_out:
   UPON not o CHECK o DO
    BEGIN
     FOR icntr=1 TO 8  DO
      BEGIN
       WAITFOR TRUE DELAY 100;
       preg := REGLROTATE(preg,1);
       ASSIGN preg[0] TO o;
      END;
     ASSIGN TRUE TO o DELAY 100;
    END
  read_bit:
   UPON NOT i CHECK i DO
    BEGIN
     WAIFOR TRUE DELAY 50;
      FOR intr=1 TO 8 DO
       BEGIN
        WAITFOR TRUE DELAY 100;
        preg[0] := i;
        IF icntr < 8  THEN
         preg := REGLSHIFT(preg,1,FALSE)
        ELSE
         vdav[0] := TRUE;
       END
     END
  read_data:
   UPON read CHECK read DO
    FOR i=7 DOWNTO 0 DO
     ASSIGN z TO data[i] DELAY 5;
  uart_float:
   UPON davin CHECK davin DO
    ASSIGN vdav TO data[0] DELAY 20;
```

```
BEGIN
 vdav := TRUE;
END
```

A.4.4 Parallel I/O (Intel 8212)

```
COMPTYPE  i8212;

 INWARD    di: four_state_net;
           ndsl,ds2,md,stb,clr: reglbit;
 OUTWARD   do: four_state-net;
           int: reglbit;
 INTERNAL  s1,s2,s3,srq: reglbit;
 VAR
   q: REGISTER[7..0];

 SUBPROCESS
  UPON TRUE CHECK di,q,s1,clr  DO
   BEGIN
    IF s1 THEN
     q := LOGTOBOOL(di);
    IF s3 THEN
     ASSIGN LOGFROMBOOL(q) TO do
    ELSE
     FOR i=7 DOWNTO 0 DO
      ASSIGN z TO do;
    ASSIGN s1 NOR clk TO s2;
   END

  UPON TRUE CHECK ds1,ds2,md,stb,srq  DO
   BEGIN
    IF md THEN
     ASSIGN NOT ds1 AND ds2 TO s1;
    ELSE
     ASSIGN stb TO s1;
    ASSIGN (NOT ds1 AND ds2 OR md)  TO s3;
    IF (NOT s2 OR not DS1 and DS2) THEN
     ASSIGN TRUE TO srq
    ELSE
     ASSIGN FALSE TO srq;
    ASSIGN (NOT srq NOR NOT ds1 AND ds2) TO int;
   END

BEGIN

END
```

REFERENCES

1. J.R.Armstrong, "Chip level modeling with Hardware Description Languages", paper in preparation.

2. J.R.Armstrong, "Chip Level Modeling and Simulation", Simulation, October 1983, pp.141-148.

3. J.R.Armstrong, "Chip Level Modeling of LSI Devices", IEEE Trans. on Computer Aided Design of Integrated Circuit and systems, October 1984, pp.288-297.

4. Hardware Description Language Guide - Texas Instruments Logic Array Design System, Texas Instruments, 1982.

5. N.2 ISP' Users Manual, Endot Inc., August 1985.

6. VHDL User's Manual, Document No. IR-MD-065-1, August 1985.

7. J.Hill, G.Peterson, "Digital Systems: Hardware Organization and Design", John Willy & Sons, 1978.

8. A.Singh, "Development of Comparison Features for Computer Hardware Description Languages", Computer Hardware De-

scription Languages and their Applications, North-Holland Publishing Company, 1981, pp.247-263.

9. M.Barbacci, "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems", IEEE Trans. on Computers, February 1975

10. ISPS Reference Manual, Departments of Computer Science and Electrical Engineering, Carnegie-Mellon Univ., 1979.

11. A.Parker, "SLIDE: An I/O Hardware Descriptive Language", IEEE Trans. on Computers, vol. C-30, No.6, June 1981, pp.423-439.

12. J.R.Armstrong, "Chip Level modeling Techniques", Technical Report 8124 (for NASA Langley, E.E. Dept., Virginia Polytechnic Institute and State University, May 1982.

13. GSP User's Guide, E.E. Dept., Virginia Polytechnic Institute and State University, September 1982.

14. G.M.Ordy, "THE N.2 SYSTEM", Proceedings, 20th Design Automation Conference, 1983. pp.520-526.

15. C.W.Rose, "N.mpc: A study in University-Industry Technology Transfer", IEEE Design & Test, February 1984, pp.44-56.

16. R.Cheng, "Functional Simulation Shortens the Development Cycle of a New Computer", Proceedings, 20th Design Automation Conference, 1983, pp.515-519.

17. R.L.Druian, "Functional Models for VLSI Design", Proceedings, 20th Design Automation Conference, 1983, pp.506-514

18. J.M.Kerr, "Compiled Code, Chip Level Simulation Using a High Level Hardware Description Language", M.S Thesis, E.E. Dept., Virginia Polytechnic Institute and State university., 1984.

19. HELIX USERS DOCUMENTATION HELIX 2.0 HHDL (Hierarchical Hardware Description Language) Reference Manual, September 1985. Documentation no. M-026-3.

20. S.Klein, S.Sastry, "Parameterized Modules and Interconnections in Unified Hardware Descriptions", Computer Hardware Description Languages and their Applications, 1981 pp.185-195.

21. R.Riloty, D.Borrione, "The Conlan Project: Concepts, implementations, and Application" Computer vol.18 no.2 Febrary 1985 pp.81-92.

22. D.Schuler, "A Language for Modeling the Functional and Timing Characteristics of Complex Digital Components for Logic Simulation", Proceedings, 16th Design Automation Conference, 1978. pp.54-59.

23. S.Dasgupta, "Computer Design and Description Languages", Advances in Computers, vol.21, 1982, pp.91-154.

24. M.R.Barbacci, J.Northcutt, "Applications of ISPS, an architecture description language", Journal of Digital System, 4(3), pp.221-239.

25. C.Bell, J.Newell, "Computer Structures: Readings and Examples." MacGrow-Hill, New York, 1981.

26. S.G.Shiva, "Computer Hardware Description Languages - A Tutorial", Proceedings, IEEE, Vol.67, No.12, December 1979, pp.1605-1615.

27. A.Pawlac, "Microprocessor Systems Modeling with MODLAN", Proceedings, 20th Design Automation Conference, 1983, pp.804-811.

28. M.R. Barbacci, "Syntax and Semantics of CHDLs", Computer Hardware Description Languages and their Applications, 1981, pp.305-311.

29. S.Dasgupta, "Hardware Description Languages in Microprogramming Systems", IEEE Computer, February 1985, pp.67-76.

30. W.M.VanCleemput, "Computer Hardware Description Languages and their Applications",

31. J.H.Aylor, R.Waxman, C.Scarratt, "VHDL - Feature Description and Analysis", IEEE Design & Test, April 1986, pp.17-27.

32. J.D.Nash, L.F.Saunders, "VHDL Critique", IEEE Design & Test, April 1986, pp.54-65.

33. R.Lipsett, E.Marshner, M. Saahdad, "VHDL - The Language", IEEE Design & Test, April, pp.28-41.

34. L.I.Maissel, H.Ofek, "Hardware design and description language in IBM", IBM. J. of RES. DEVELOP. Vol.28, No.5, September 1984, pp.557-563.

35. K.J.Lieberherr, "Chip Design in ZEUS and a proposal for a Standard Benchmark Set for Hardware Description Languages", Proceedings, 22th Design Automation Conference, 1984, pp.66-72.

36. W.Hahn, "Computer Design Language - Version MUNICH (CDLM): A Modern Multi-Level Language", Proceedings, 20th Design Automation Conference, 1983, pp.4-11.

37. S.Y.Su, "A Survey of Computer Hardware Dscription Languages in the U.S.A., IEEE Computer, December 1974, pp.45-51.

38. J.R.Duley, D.L.Dietmeyer, "A Digital System Language (DDL)", IEEE Trans. on Computer, vol. c-17, September 1968, pp.850-861.

39. W.M.VanCleemput, "An hierarchical language for the structural description of digital systems", Proceedings, 14th Design Automation Conference, 1977, pp.377-385.

40. W.Johnson, J.Crowley, M.Steger, E.Woosley, "Mixed-Level Simulation from a Hierarchical Language", J. of Digital Systems, vol.IX, Issue 3, pp.305-335.

41. F.J.Hill, R.E.Swanson, M.Masud, Z.Navabi, "Structure Specification with a Procedural Hardware Description Language", Trans. on Computers, vol. c-30, No.2, February 1981, pp.157-161.

42. D.R.Coelho, "Behavioral Simulation of LSI and VLSI Circuits", VLSI Design, February 1984. pp.42-51.

43. C.H.Cho, "Multimodule Simulation Technique for Chip Level Modeling", Masters Thesis, EE Department, Virginia Polytechnic Institute and State University, March 1986.

44. VHDL Design Analysis and Justification, Document No. IR-MD-018-1, July 1984.

The vita has been removed
from the scanned document