Compiling Verilog into Automata

Szu-Tsung Cheng [1]
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Robert K. Brayton
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

May 18, 1994

[1]Supported by Siemens, CA Micro, Cadence, and Fujitsu

Contents

Acknowledgements

This work would not have been possible without the help of many other people. First of all, I would like to give my special thanks to my advisor, Robert K. Brayton, for his exceptional and inspirational guidance. I would like to thank Gary York for his guidance and excellent ideas in writing and organizing vl2mv. An important step in this work involves the interpretation of timing constructs in Verilog, which was a EECS290h project. I would like to thank Felice Baralin for advice and inspiring discussions during that time. I want to thank professor Alberto Sangiovanni-Vincentelli, professor Katherine A. Yelick, Adnan Aziz, Wan-teh Chang, Ramin Hojati, Shan-Hsi Huang, Timothy Kam, Joe King, Sriram Krishnan, William Lam, Wei-Yi Li, Rajeev Murgai, Rajeev Ranjan, Tom Shiple, Vigyan Singhal, Serdar Tasiran, Tiziano Villa, Huey-Yih Wang, Robert Wang, Chi-Po Wen, Ephrem Wu for helpful discussions. I want to thank those who used and kindly gave feedback regarding bugs, suggestions, comments for vl2mv. Development of vl2mv would have been slowed down without these suggestions. I also want to address my grateful thanks to Siemens, CA Micro, Cadence, and Fujitsu for generous support during this research. I thank my parents, my wife, Li-Ling Jeng, for their patience, support, and encouragement.

Chapter 1

Introduction

The Verilog [TM91] Hardware Description Language (Verilog HDL) is one of the most popular and widely used languages for digital design. Verilog allows mixed-level descriptions of hardware in terms of their static structures as well as dynamic behaviors. To facilitate description of dynamic behavior, Verilog has high level constructs like conditional, loop-control, process fork/join, such that designers can describe the behavior much the way they write programs in general programming languages. Verilog also has decorators facilitating the quantitative description of time. This makes it easy to specify delays associated with statements, gates, or modules.

BLIF-MV [BCH+91], a multi-valued extension of BLIF, is the input format used by HSIS [ABB+94], an integrated interactive hierarchical verification/synthesis system. Basic constructs in BLIF-MV consist of module declaration/instantiation, tables which allow descriptions of nondeterminism, and symbolic latches. At each "clock" cycle, each table updates its outputs according to the inputs it sees until a fix-point is reached. In the very beginning of the next cycle, all latches simultaneously update their present states according to their next state inputs. Then again, tables update their outputs accordingly.

The relationship between a behavioral description language like Verilog and a machine description language like BLIF-MV is similar to that between a high level programming language and an assembly language. In a high level language one can easily describe desired behavior in an abstract way while omitting irrelevant details, e.g. given a statement `o=a-b`, it does not matter how a subtraction is implemented if only `o` gets the difference between `a` and `b`. A low level language gives a detailed, well defined execution model where every aggregate operation is decomposed into a series of well defined primitives on elemental data units.

vl2mv is built to bridge between various existing designs in Verilog and powerful synthesis/verification algorithms in HSIS/SIS (Figure 1.1). vl2mv extracts a set of finite state machines (FSMs) which preserve the behavior of the source Verilog programs which is defined in terms of simulated results. Allocation of hardware gates to operators in Verilog (i.e., *resource binding*) is based on the assumption of unlimited resources. The resource pool consists

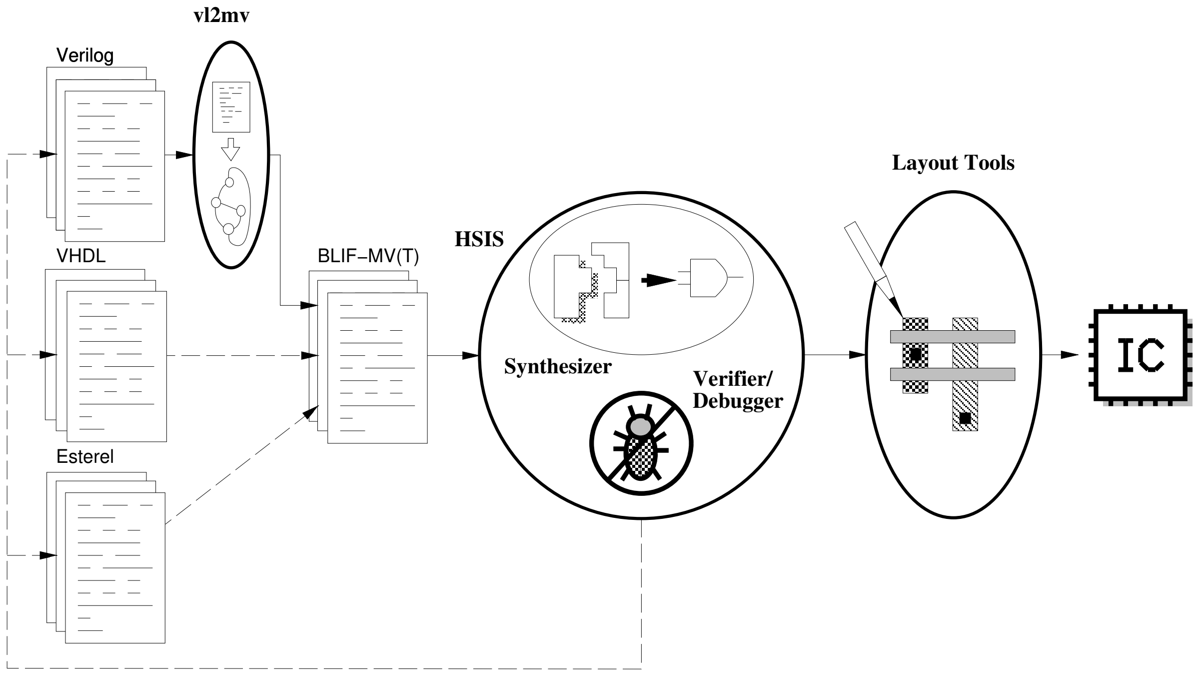


Figure 1.1: An Automatic Verification/Synthesis Path from Specification/Design to IC Fabrication.

of all possible gates expressible in one table in BLIF-MV. No scheduling is performed and no optimization is applied on the Verilog source. The extracted FSMs are not guaranteed to be "optimal" in any sense (area, speed, or a combination of the two). If it is desired to get an optimal implementation (e.g., a real circuit), then an advanced synthesis system like SIS [Sea92] can be applied to the generated FSMs to get a more compact design. Target BLIF-MV code can also be used for verification against desired properties. We also extend Verilog to make it possible to describe nondeterministic transitions. This allows us to specify both *system* and *task* processes using the same HDL (however, at this point, we still need the help of PIF files, Property Interchange Format [Ver], to exclude undesired behavior introduced by abstraction and to specify desired "sink" states and transitions). If it can be shown formally that a design satisfies the specifications (which can be expressed in CTL formulae [Eme90], or a set of automata using Verilog + PIF), then it can be used as input to a synthesis system to minimize it. Not only can vl2mv handle un-timed Verilog models, but it can also extract quantitative timing information from a timed Verilog program. The generated timed automata is in BLIF-MVT [BBC+], an extension of BLIF-MV with timing constructs.

The organization of the report is as follows. Section 2.1 gives terminology used throughout this report. Section 2.2 describes the synthesizable subset of Verilog that can be handled by vl2mv. Sections 3.1 and 3.2 describe basic constructs (unrelated with timing) and how they are compiled. In chapter 4, we extend the scope of the un-timed synthesizable subset of Verilog for vl2mv to include timing constructs. In this chapter, we first look at a very important issue in modeling transition systems and/or real circuits. The notions of *explicit* and *implicit clocking* are introduced. Section 4.2 details the algorithms used to extract timed/un-timed machines from a subset of Verilog with timing constructs. Chapter 5 presents various compiler functionality provided by vl2mv. These special functions include compiler directives (macros, file inclusion, section 5.1), compatibility checking (section 5.2.1), operator abstraction (section 5.2.2), table decomposition for non-blocking assignments (section 5.2.3), and source debugger support (section 5.3). Chapter 6 gives conclusions as well as future work of this project. The complete synthesizable subset of Verilog for vl2mv is given in appendix A.

Chapter 2

Basics

2.1 Definitions

In this section definitions that will be used later are introduced.

Definition 2.1.1 (Delays, Event controls) *A* delay *is a Verilog delay operator (*`#`*) which specifies the duration a program execution should be halted. An event control causes an execution pause until either a rising edge (*`@(posedge x)`*), a falling edge (*`@(negedge x)`*), or any one of both (*`@(x)`*) occurs. An event control is also referred to as an* edge event control.

Definition 2.1.2 (Simple statements, Composite statements) *A* simple statement *is a statement that contains no sub-statement other than itself, i.e.,* `if/else`*,* `case`*,* `for`*,* `begin/end` *statements are not simple statements (which are called* composite statements*). Note also that simple statements contain no delay or edge event control. Basically, a simple statement is a statement that can be executed in "zero hardware time".*

In the subset of timed Verilog that can be synthesized, assignment statements are the only simple statements. For example, `y <= a - b;` is a simple statement.

Definition 2.1.3 (Basic blocks) *A basic block is a linear sequence of simple statements that passes through no branches such as* `if/else`*,* `case`*, or* `for`*. Thus they do not contain any delays and/or event controls. Basic blocks may cross statement boundary, and may overlap.*

Definition 2.1.4 (Conditional statement, Conditional expression, Conditional branch) *A* conditional statement *is an* `if/else` *statement, a* `case` *statement, or a* `for` *statement. A* conditional (logical) expression *of a conditional statement is the Boolean expression which determines which* conditional branch *is taken on executing the conditional statement.*

An example of the above definition is shown in Figure 2.1.

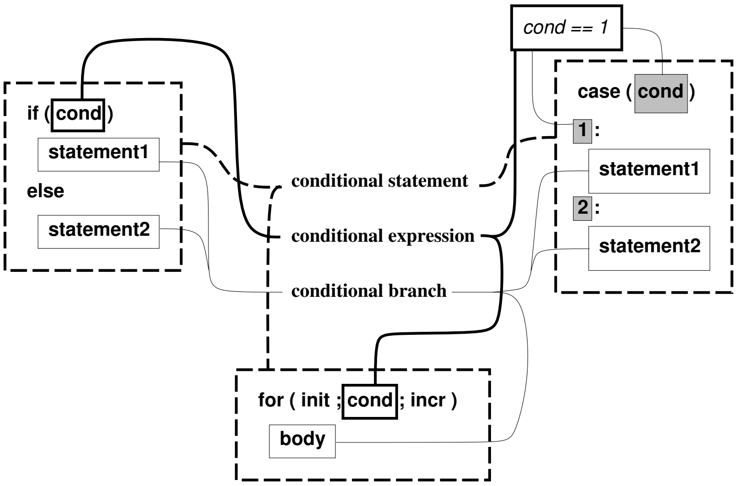


Figure 2.1: Definition of *conditional statement*, *conditional expression*, and *conditional branch*

Definition 2.1.5 (Conditional blocks, Significant and Insignificant) *A* conditional block *is a conditional statement (*`if/else` *statement,* `case` *statement, or* `for` *statement). It involves all branches of the conditional statement. A* significant conditional block *is a conditional block where at least one of its branch statements contains one or more delays or event controls; otherwise, it is* insignificant*.*

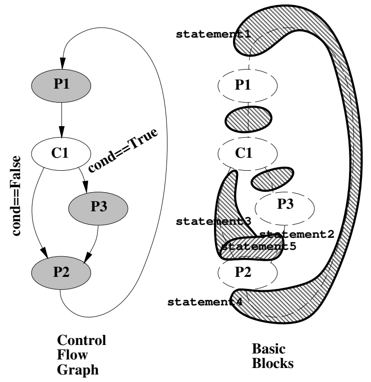
Definition 2.1.6 (Control Flow Graph (CFG)) *A* Control flow graph *is a directed multi-graph $G = (V_p + V_c, E)$. There is a distinct $p \in V_p$ for each delay (*`#`*) or edge event control (*`@(posedge x)` */* `@(negedge x)` */* `@(x)`*) in a Verilog program, a distinct $c \in V_c$ for each conditional statement (*`if/else`*,* `case`*, or* `for`*). A node $p \in V_p$, which has a corresponding delay/event control in the HDL program is called a* pause*. There is an edge $e = (v_1, v_2)$ iff there is a basic block between the expressions which v_1 and v_2 ($v_1, v_2 \in V_p + V_c$) stand for. Any edge $e = (v_s, v_d)$ where $v_s \in V_c$ is called a* context edge *and is labelled with the conditional expression of the branch which e stands for. The label is denoted by $L(e)$. A* legal *CFG is a control flow graph where there is at least one pause on every cycle.*

A control flow graph example for a timed Verilog program is shown in Figure 2.1.1.

Verilog program	*Control Flow Graph*

```
always
   statement1;
   # 1
   if (cond1)
      # 3
      statement2;
   else
      statement3;
   statement5;
   # 2
   statement4;
```

$\Rightarrow$



$$G = \left(\begin{array}{c} V = V_p + V_c = \{P_1, P_2, P_3, C_1\}, \\ E = \left\{ \begin{array}{c} (P_1, C_1), \\ (C_1, P_3)_t, (C_1, P_2)_f, \\ (P_3, P_2), \\ (P_2, P_1) \end{array} \right\} \end{array} \right)$$

Figure 2.1.1: An example Verilog program and its control flow graph.

Note that since an `always` statement execution is wrapped-around, a basic block might cross the `always` boundary. There are three basic blocks in the above example. One contains statement 2, 5. One contains statement 3, 5. The last contains statement 1, 4. A control flow graph can be extracted in a depth first traversal along `always` loops. In the following discussion, CFGs are assumed to be legal. In addition, we use pauses and delay-operators/event-controls interchangeably whenever it does not cause any confusion.

2.2 Synthesizable Subset of Verilog for vl2mv

Conceptually, a design in a synthesizable subset of Verilog consists of a set of modules (either hardware or software). The first module encountered is regarded as the root module. All modules run in parallel and communicate with each other through a set of channels (which are essentially sets of wire variables declared in the modules that these channels belong to). It is assumed that communication through channels takes no time. Within each module, values on channels can be accessed through a set of ports. Ports can be either wires or registers. Through wire (register) ports a module can input/output from/to channels instantaneously (at a lag of one time unit). A wire (register) port has no (one) storage element associated with it.

A module consists of a set of *declarations*, *module instantiations*, *continuous assignments* (assignments begin with the key word `assign` which are always "active"; they can be thought of as combinational blocks), and *procedural blocks* (sequential blocks, sometimes referred to as `always` statements. Statements within a procedural block are executed sequentially); refer to Figure 2.2. Module instances, continuous assignments, and procedural blocks within a module run concurrently. Execution of each continuous assignment, basic block in a procedural block, and module instance is assumed to be atomic within each instant. Caution should be taken when there is more than one procedural block in the same module. A Verilog simulator treats each statement as atomic instead of each basic block (which may consist of several sub-statements). If there is more than one procedural block in a module, the simulated result may depend on how expressions from different blocks are interleaved by the simulator. For more details, refer to section 3.1.4.

Continuous assignments are treated as combinational circuits with their left hand side variables as primary outputs. A procedural block is an `always` statement. Delays/event-controls synchronize the transition of a system. The synthesizable subset of Verilog for vl2mv allows delays/event-controls to appear anywhere inside an `always` statement. It requires that no execution can "sneak through" any delay/event-control, i.e. no non-blocking delay/event-control is allowed. For more details about how synchronizing signals are synthesized and what compiler options should be used, refer to section 4.1. Variable updates are arbitrated by an *arbiter* (Figure 2.2). An arbiter is further controlled by a timing automaton if the source Verilog program is timed. In case there exists variable contention, the arbiter selects nondeterministically one of the values assigned to that variable as its new value. If there is no contention, the arbiter is simply a bus which connects the definition of a variable to destinations where it is consumed. Arbiters also regulate among non-blocking and blocking procedural assignments. Consider for example,

```
always
   begin
   if (x == 1);
      y <= 0;
   y = 1;
   end
```

In case `x` is 1, the value of `y` in the next time slot is 0 since the non-blocking assignment (`y<=1`) overwrites the blocking assignment (`y=1`) as time advances. When an arbiter is choosing the next state value for a register variable, it gives higher priority to non-blocking assignments than to blocking assignments.

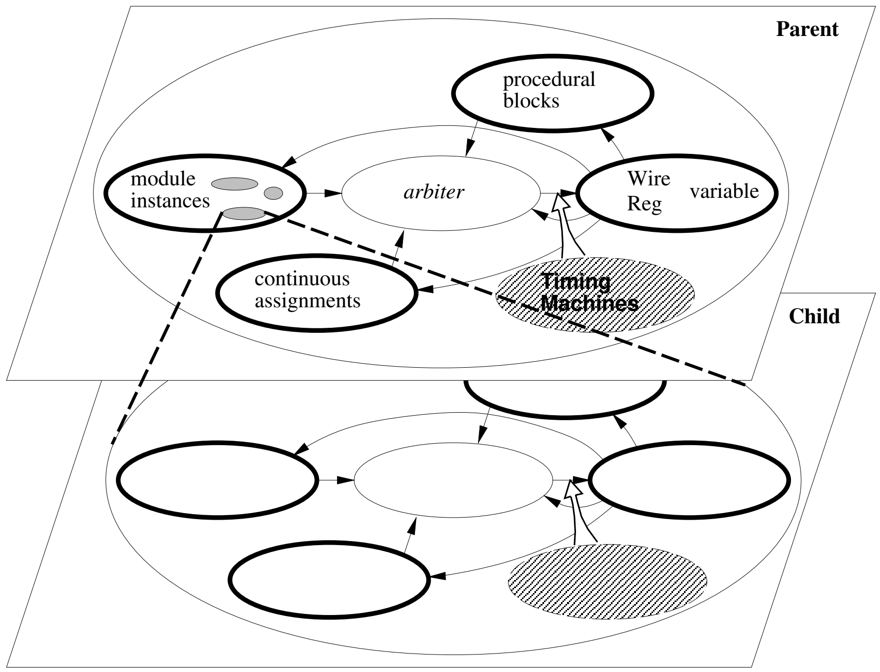


Figure 2.2: Module structure

Chapter 3

Translation of Untimed Components

3.1 Basic Constructs and Some Extensions

This section gives a detailed description of the untimed synthesizable subset of Verilog for vl2mv, some extensions to existing Verilog, and the way constructs are compiled. For the syntax and more details of Verilog, refer to [TM91]. The syntax of the complete synthesizable subset of Verilog for vl2mv can be found in appendix A.

3.1.1 Variable Declaration and Usage

Net Variables

Net variables are declared through `wire` declarations. For example, `wire[0:2] a;` declares a 3-bit wire variable `a`, which is composed of 3 1-bit elements, `a<0>`, `a<1>`, and `a<2>`. Any variable not decorated by `reg` is by default regarded as a net variable. Only net variables can appear at the left hand side of *continuous assignments*. Nets can be referred to from within continuous assignments, procedural blocks, or module instances. Nets do not store values, nor can they be initialized. They only transmit values from driver(s) (places where they are assigned) to destination(s) (where they are referred). The value of a net is determined by the output of its driver(s). If there is more than one driver for the same wire variable, the value of the net will be chosen nondeterministically. For example, the left hand side Verilog program below is compiled into the right hand side BLIF-MV table:

```
assign a = x;          .names x y a
                 ⇒     - - =x
assign a = y;          - - =y
```

In Verilog [TM91], nets cannot be used at the left hand side of *procedural assignments* (assignments in *procedural blocks*). However, vl2mv extends Verilog to allow non-blocking procedural assignments with wires as their left hand side variables. This extension is made for two purposes. First, in describing a transition system, it might be desirable to describe deadlocks. In original Verilog, if a variable is not assigned any value in the current simulation cycle then it retains the same content in the next simulation cycle. i.e. the transitions of a system in original Verilog are always *complete* (at any reachable state of a system, its next state is defined given any input). To describe deadlock, designers may either introduce an extra deadlock state (trap state) or use a system task (system tasks are simulator directives which can do file I/O, dump module status, monitor variable content, and suspend simulation) to explicitly assert that certain transitions should never happen. Second, sometimes it makes the description of transition system easier if it is allowed to combine assignments to registers and assignments to wires in the same statement. For example, if it is desired to say that if the current state of a machine is `green` then a combinational output `turn_on_green_light` should be asserted and the next state of the machine is `yellow`. Using original Verilog, we need two tests on the current state of the machine, one for continuous assignment to the wire variable and the other for procedural assignment to the register (state) variable, as shown in the following example:

```
assign turn_on_green_light = (state == green) ? 1 : 0;
always
    ...
    if (state == green) state <= yellow;
    ...
```

On the other, it would be easier if it is allowed to test the current state then make assignments to both register variable and wire variable, as shown in this example:

```
always
    ...
    if (state==green) then
        begin
        turn_on_green_light <= 1;
        state <= yellow;
        end
    else turn_on_green_light <= 0;
    ...
```

The caveat is that *programs using non-blocking assignemtns to wires do not simulate*. Besides, since a wire variable induces no delay, careless use of non-blocking assignments may result in non-intuitive program behavior. For example, suppose that varible `w` is a wire,

```
1 always @(posedge clk)
2    begin
3    if (w == 0)
4       state = 1;
5    w <= 1;
6    end
```

The statement in line 5 can actually affect the decision made in line 3 since no symbolic latch is allocated for `w` and there is no delay between the statement that assigns it and the statement that uses it.

The semantics for non-blocking assignments made to wires are as follows. Consider the example, `always @(posedge clk) if (state==0) w <= 0;` where `w` is a wire variable. It does not specify what value `w` should take when `state` is not 0. Note that no storage is allocated for `w` since it is a wire, it can not "remember" what its value was in the previous instant, i.e. `w` can not self-loop in the same "state". Two possible semantics, *deadlock* and *nondeterminism*, are proposed. With deadlock semantics, if the value of a wire variable is not specified in any instant, there is a deadlock. In the above example, there is a deadlock when `state` is not 0. Nondeterminism semantics simply means that the value of a wire will be nondeterministically chosen from its domain if its new value is not specified in a certain instant.

Register Variables

Register variables are declared through `reg` declarations. They can only be assigned through procedural assignments (`=`) and non-blocking assignments (`<=`). Register variables can be initialized with `initial`. If there is no initial statement associated with a `reg` variable, a reset circuit is generated which nondeterministically initializes the variable to be any of its domain values. There is one BLIF-MV symbolic latch allocated for each register variable component.

If there is more than one process running in parallel assigning to the same `reg` variable, its value in the next clock cycle will be chosen nondeterministically from those values assigned. For instance, given the following Verilog program with two `always` statements within the same module

Statement A_1 : `always @(posedge clk) if (`c_1`) a =` e_1;

Statement A_2 : `always @(posedge clk) if (`c_2`) a =` e_2;

$\Downarrow$

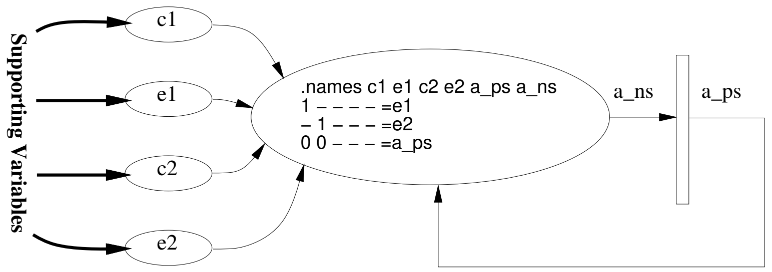


Figure 3.1.1:Conflict arbiter.

a circuit like the one in Figure 3.1.1 is generated, where `a_ps` denotes variable `a`'s present state and `a_ns` its next state.

Vector Declaration

Vector declaration is facilitated by annotating the type of the variable (input, output, reg, wire, etc) with the width (`[vector_lower_bound:vector_upper_bound]`) of the vector. For example,

```
input [0:7] a;
wire [0:7] a;
```

declares a 8-bit input wire vector `a` from bit 0 (l.s.b.) to bit 7 (m.s.b.). Note that if a variable is declared by more than one statement, their vector decorations have to be consistent, as shown in the above example.

Array Declaration

Arrays are declared by annotating variables with the `[lower_bound:upper_bound]` decorator. For example, to declare a 3-element array `a` where each element is a 1-bit scalar, one can write `reg a[0:2];`. For each array element access, a multiple-fanin multiplexor, controlled by the indexing variable, is generated in order to select the correct variable to be accessed. For example,

```
                          .names i<0> i<1> a[0] a[1] a[2] b
                          0 0 - - - =a[0]
assign b = a[i];   ⇒     0 1 - - - =a[1]
                          1 0 - - - =a[2]
                          1 1 - - - -
```

If an array variable is used as a left hand side variable, then a set of MUXes controlling each element of the array is generated. For example, given the following Verilog code fragment, the circuit in Figure 3.1.1 is generated.

```
always @(posedge clock)
   a[i] = b;
```

⇒

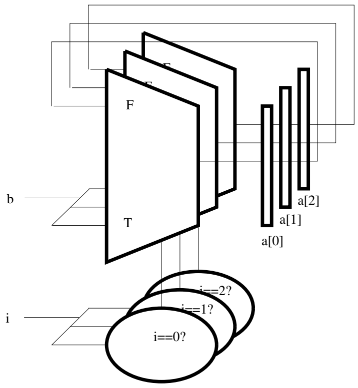


Figure 3.1.1: Circuit for array assignment.

Enumerated Type and Symbolic Variable Declaration

In certain phases of system design when the encoding of variables are not known yet, it might be desirable to specify and examine the value of some variables symbolically instead of hard-code them in the first place. vl2mv extends Verilog to allow users to declare symbolic variables using an enumerated type mechanism similar to the enumerated type in the C programming language [KR78]. The syntax of an enumerated type declaration is as follows (words enclosed by angle brackets represent nonterminals, + indicates one or more repetitions of the preceding terminal/nonterminal):

```
enum <enum_name> { <enumerator_name>+ }
```

It declares an enumeration name <enum_name>. It can be regarded as declaring a named set (with <enum_name> its name) consisting of the elements that follow. To declare a new type, one says:

```
typedef <type_specifier> <type_name> ;
```

which declares new type <type_name> to be of type <type_specifier>. As an example, assume that the state of a man could be studying, working, playing, eating, or sleeping. Then we can declare a type status_t which ranges over the possible states a man is in:

```
typedef enum status {studying, working, playing, eating, sleeping} status_t;
```

Enumeration name status can be omitted. If that is the case, status_t is a new type ranging over an anonymous set consisting of studying, working, playing, eating, and sleeping. To declare a wire variable of type status_t, one says:

```
status_t wire state_of_boss;
```

To declare a register of type status_t, on says:

```
status_t reg state_of_boss;
```

3.1.2 Concatenation

Concatenation in Verilog allows bits/vectors to be glued together to form an aggregate variable. vl2mv simply creates a vector of intermediate BLIF-MV variables with its components connected to constituents of the concatenation. For example,

```
wire b;
wire [0:1] c, d;
wire [0:4] a;
assign a = {2'b10, b, c & d};
```

is compiled into:

```
.names -> t<0> t<1>
0 1

.names b -> t<2>
- =b

.names c<0> d<0> -> t<3>
.def 0
1 1 1
.names c<1> d<1> -> t<4>
.def 0
1 1 1

.names t<0> t<1> t<2> t<3> t<4> ->
          a<0> a<1> a<2> a<3> a<4>
- - - - - =t<0> =t<1> =t<2> =t<3> =t<4>
```

where 2'b10 means a 2 bit binary constant 10.

Symbolic variables can not be concatenated.

3.1.3 Continuous Assignments

Continuous assignments have the form: assign <var> = <expr>;. Continuous assignments are always "active", i.e., whenever any one of their inputs changes, their output is updated instantaneously. Only wire variables can be used as the left hand side of continuous assignments. Continuous assignments provide an abstract means of describing combinational logic of a circuit. Continuous assignments are intended to describe combinational behavior of circuits instead of their implementation. Since *resource binding* is not implemented in vl2mv, for these continuous assignments vl2mv chooses arbitrarily BLIF-MV implementations consistent with the expressions. For example, given statement o = a+b;, it is not specified whether the addition should be implemented by a carry-ripple adder, look-ahead adder, or carry-skip adder. vl2mv chooses any circuit as long it is consistent with the definition of addition.

3.1.4 Procedural Assignments

Procedural statements (`=` within a procedural block), also referred to as *blocking assignments*, act like normal software programming language assignments in that they execute sequentially within a procedural block, changing the content of state variables, until the execution is blocked by a pause. vl2mv compiles procedural blocks based on the assumption that each basic block will be executed atomically given the delay/event-control of the block is satisfied. It is also assumed that execution of procedural assignments takes zero hardware time. All procedural blocks with active event controls get executed concurrently.

However, a Verilog simulator does not treat simple blocks as atomic. Instead, it executes basic statements atomically (a simple block may consist of several basic statements). This may introduce disagreement between automata traces and simulation results. For example, given the following two concurrent processes of the same module,

Process 1	Process 2

`always @(posedge clk)`
`begin`
`x=1; x=x+1;`
`end`
	`always @(posedge clk)`
	`x=4;`

due to process interleaving in a Verilog simulator, the possible traces of `x` that could be generated are $< 1, 2, 4, \ldots >$, $< 1, 4, 5, \ldots >$, $< 4, 1, 2, \ldots >$. The corresponding possible new value for x in the next clock cycle will be 4, 5, 2, depending on how a simulator schedules the processes. On the other hand, if the two procedural blocks are treated atomically, only $< 1, 2, 4, \ldots >$ or $< 4, 1, 2, \ldots >$ will be the possible execution sequence and the only possible values of `x` in the next clock cycle are 4 and 2. For the above example, vl2mv generates a circuit in Figure 3.1 where `x_ps` and `x_ns` denote the current and next state variable of register `x`, respectively. `x1` and `x2` are intermediate varaibles for `x`.

Hence, if there is more than one procedural blocks sharing the same `reg` variables, caution should be taken to make sure *the desired behavior does not depend on specific interleaving among processes.*

3.1.5 Non-blocking Assignments

Procedural assignments update variables instantly, hence in some cases race conditions might arise among multiple procedural assignments. Non-blocking procedural assignments provide a mechanism that defers the assignment without blocking the execution of statements in a block. On encountering a non-blocking assignment, the right hand side of the assignment is evaluated according to the most recent values of variables referred. Without changing the variable on the left hand side, program execution continues. Variables are updated simultaneously at the

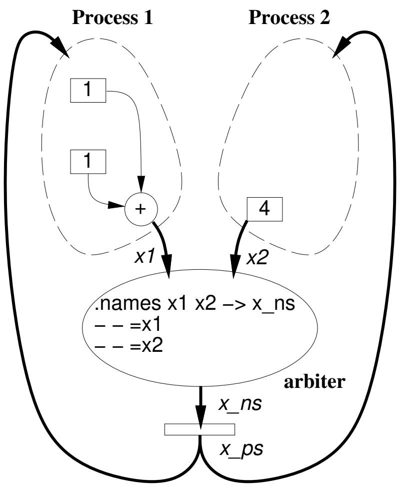


Figure 3.1: Concurrent Processes Accessing the Same Register Variable.

very beginning of the next time slot.

Non-blocking assignments also provide a way to introduce nondeterminism [BY93] on `reg` variables. If there is more than one non-blocking assignment in the current time slot assigning to the same register variable, then the value of that register variable in the next clock cycle will be nondeterministically chosen from those values assigned.

3.1.6 Interaction Between Blocking and Non-blocking Assignments

Blocking assignments "block" the execution of a program in the sense that they do not let the program execution continue until variables appear on the left hand sides are changed as indicated by the right hand sides. That is, blocking assignments have immediate effect on variables they touch.

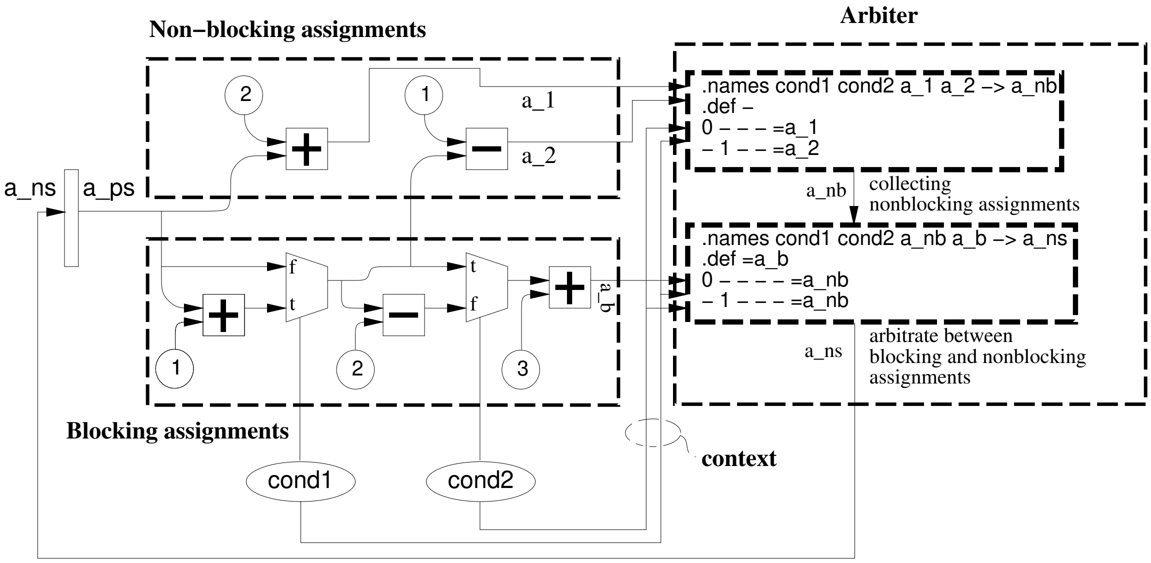
On the other hand, on executing non-blocking assignments, right hand sides are evaluated and, without modifying the content of the left hand side variable, it is "remembered" that there is a variable alteration suspended and the alteration should be realized at the very beginning of the next time slot.

Given that there are no pending alteration, the next state value of a register variable is derived from the last blocking assignment made to that variable. However, if there is one or more pending alteration due to the execution of non-blocking assignments, no matter what value of the last blocking assignment is, the next state value of a register is derived from the set of non-blocking assignments executed in the current time slot.

The following example illustrates these concepts more fully.

```
if (cond1)
   a = a + 1;
else
   a <= a + 2;
if (cond2)
   a <= a - 1;
else
   a = a - 2;
a = a + 3;
```

$\Downarrow$



where `a_ps` and `a_ns` are the present and next state variable of register variable `a`, respectively.

3.1.7 Nondeterminism on Wire Variables

With non-blocking assignments on register variables it is possible to make the next state function nondeterministic. However, it is still desirable to specify nondeterminism on wire variables. We extend the system tasks in Verilog to describe nondeterministic wires. For example, given that `out` is a wire variable whose domain is `swallow`, `eat`, and `drink`, to say that `out` can be `eat` or `drink` nondeterministically, one can write:

```
assign out = $NDset(eat, drink);
```

Value of `$NDset` is nondeterministically chosen from its argument expressions whenever hardware time progresses. Hence, in the above example, the value of `out` is nondeterministically `eat` or `drink`. A BLIF-MV table similar to the following one is generated for the above example,

```
.names -> out
eat
drink
```

3.1.8 Statement Sequence

A sequence of statements is compiled as if there are unlimited resources of hardware. For each assignment, a new signal (BLIF-MV variable) is introduced to represent the new value of the left hand side variable. The next state signal associated with a register variable comes from the last assignment to that variable. For example, the following program is compiled into a circuit like the one in Figure 3.1.8.

```
(1)x = y+z;
(2)z = x-1;    ⇒
(3)y = z;
```

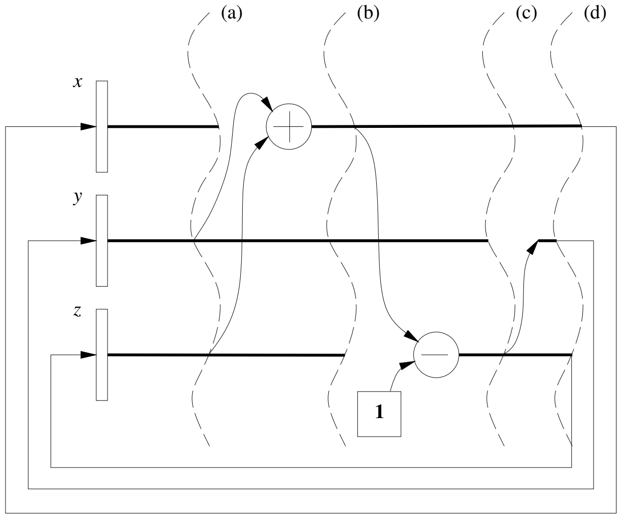


Figure 3.1.8: Circuit generated from sequence of statements. (a) Before any statement is encountered. (b) After `x=y+z;` is encountered. (c) After `z=x-1;` is encountered. (d) After `y=z;` is encountered.

3.1.9 If/Else, Case, and Conditional Statements

The first arguments to `if/else` statements are boolean expressions. Currently supported relational operators are `==` (equality), `!=` (inequality), `<` (less than), `>` (greater than), `=<` (less than or equal to), and `>=` (greater than or equal to). For symbolic variables, the only supported relational operators are `==` and `!=`. Conditional statements like `if/else`, `case` are compiled into a series of MUXes. For example, the following Verilog program is compiled into the circuit in Figure 3.1.9.

```
always @(posedge clk) begin
   case (st)
      0:  a = e0;
      1:  a = e1;
      2:  a = e2;
      default:;
   endcase
end
```

⇒

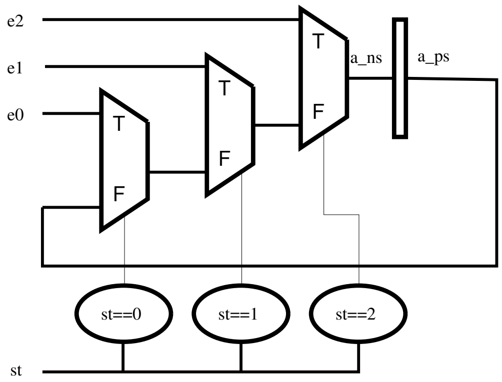


Figure 3.1.9: Circuit for case statement

3.2 Translation of Primitive Gates

The Verilog primitive gates (and, nand, or, nor, ...) are compiled into `__sis_<gate_name>_<num_of_inputs>`, e.g., a 3-input `and` gate is compiled into a library gate `__sis_and_3`. i.e., a Verilog program fragment

```
and(o, x, y, z);
```

is compiled into the BLIF-MV fragment

```
.subckt __sis_and3 o=o a=x b=y c=z
```

where the definition of `__sis_and_3` is

```
.model __sis_and3
# I/O ports
.inputs c
.outputs o
.inputs a
.inputs b

# assign o = a & b & c
# a & b
.names a b _n3
.def 0
1 1 1
# a & b & c
.names _n3 c _n4
.def 0
1 1 1
##l 11##
##v o##
```

```
.names _n4 o:raw_n2
0 0
1 1
# conflict arbitrators
.names o:raw_n2  o
0 0
1 1
# non-blocking assignments
# latches
.end
```

The library of gates supported is in `vl2mv/common/gen_lib/library.mv`.

3.3 Example: Dining Philosophers

The system models three philosophers sitting around a dining table. There is one chopstick between neighboring philosophers. Each philosopher can `THINK`, `EAT`, or `READ`. If anyone feels hungry, s/he must get a chopstick at each side in order to eat.

```
typedef enum { THINKING, EATING, READING, HUNGRY } status;

module diners(clk);
input clk;

status wire s0, s1, s2;

philosopher ph0 (clk, s0, s1, s2, READING);
philosopher ph1 (clk, s1, s2, s0, THINKING);
philosopher ph2 (clk, s2, s0, s1, THINKING);

endmodule

module philosopher(clk, out, left, right, init);
input clk;
input left, right, init;
output out;
status wire init;
status wire out, left, right;
status reg state;
```

```
assign out = (state==THINKING) ? $NDset(THINKING, HUNGRY) :
             (state==EATING)   ? $NDset(EATING, THINKING) :
             (state==READING)  ? $NDset(THINKING, READING) :
                                 $NDset(THINKING, EATING, READING, HUNGRY);

initial state = init;   // initialize state variable

always @(posedge clk) begin
    case(state)
        THINKING:
            begin
            if ((out == HUNGRY) && !((left == EATING) |
                                     (right == HUNGRY) |
                                     (right == EATING)))
              state = EATING;
            else if ((out == THINKING) && (right == READING))
              state = READING;
            end

        EATING:
            begin
            if ((out == THINKING) && !(right == READING))
              state = THINKING;
            else if (out == THINKING)
              state = READING;
            end

        READING:
            begin
            if ((out == READING) & (left == THINKING))
              state = THINKING;
            end

        default:;  // do nothing

        endcase
end
```

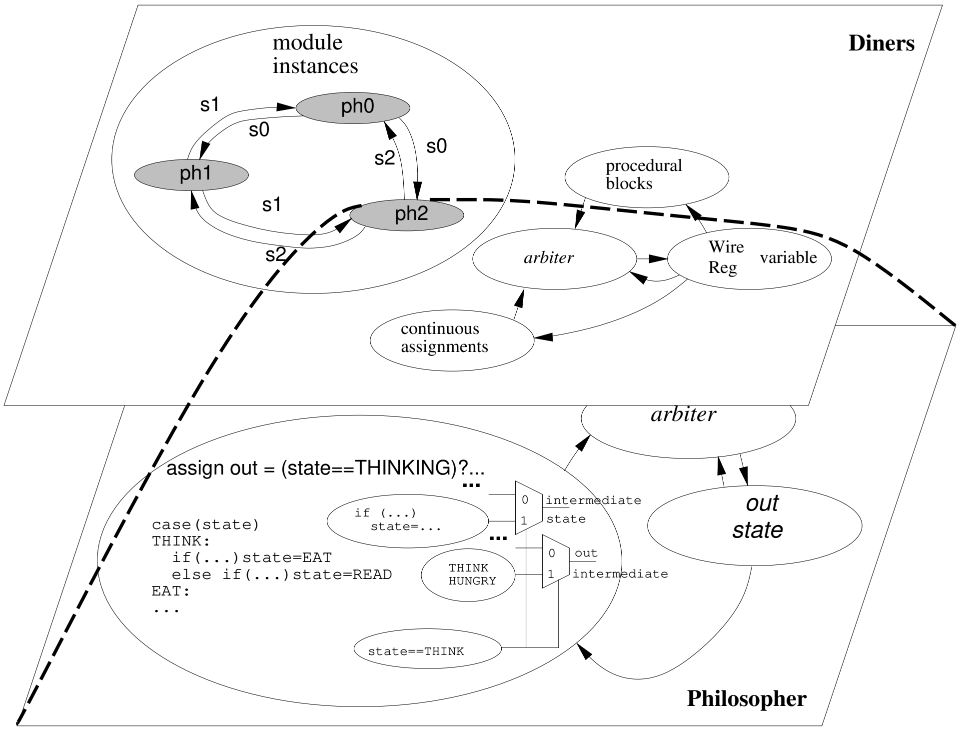


Figure 3.2: Circuit Structure for Dining Philosophers

```
59    endmodule
```

Part of the generated circuit is shown in Figure 3.2.

By using non-blocking assignments to wires, the above example can be rewritten as the following example. Note that continuous assignment to `out` is moved into the `always`-statement and `out` is assigned through non-blocking assignments (line 28, 29, 40, 41, and 50, 51). Although not compatible with the Verilog simulation, vl2mv would produce correct hardware if nondeterminism semantics are chosen.

```
1     typedef enum { THINKING, EATING, READING, HUNGRY } status;
2
3     module diners(clk);
4     input clk;
5
6     status wire s0, s1, s2;
7
8     philosopher ph0 (clk, s0, s1, s2, READING);
9     philosopher ph1 (clk, s1, s2, s0, THINKING);
10    philosopher ph2 (clk, s2, s0, s1, THINKING);
11
```

```
endmodule

module philosopher(clk, out, left, right, init);
input clk;
input left, right, init;
output out;
status wire init;
status wire out, left, right;
status reg state;

initial state = init;   // initialize state variable

always @(posedge clk) begin
    case(state)
        THINKING:
            begin
            out <= THINKING;
            out <= HUNGRY;
            if ((out == HUNGRY) && !((left == EATING) |
                                     (right == HUNGRY) |
                                     (right == EATING)))
              state = EATING;
            else if ((out == THINKING) && (right == READING))
              state = READING;
            end

        EATING:
            begin
            out <= EATING;
            out <= THINKING;
            if ((out == THINKING) && !(right == READING))
              state = THINKING;
            else if (out == THINKING)
              state = READING;
            end

        READING:
            begin
```

```
            out <= THINKING;
            out <= READING;
            if ((out == READING) & (left == THINKING))
              state = THINKING;
            end

        default:;  // do nothing

        endcase
end
endmodule
```

Chapter 4

Timing

4.1 Implicit Clocking vs. Explicit Clocking

In BLIF-MV, symbolic latches are implicitly controlled by a global clock which in some sense denotes the notion of "progression of time". Note that this "clock" need not to be a real wire as in the hardware sense. All symbolic latches transit to the next state indicated by the relevant transition tables at the same time.

Verilog is also used to specify such synchronous systems:

```
always @(posedge time)
   begin
   a = expr1;
   b = expr2;
   end
```

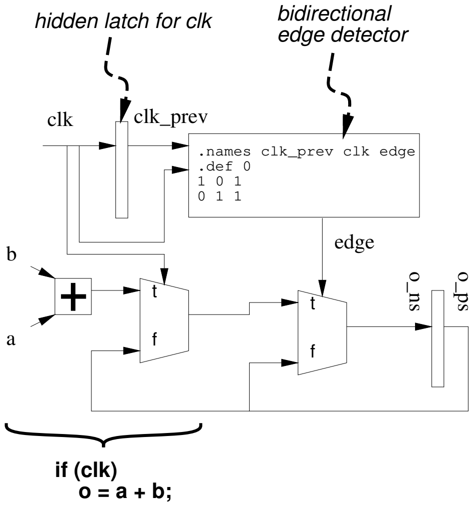
In this example, `time` represents only the notion of progression of time agreed on by all FSMs. `time` should not be interpreted literally as a true variable. It should be disregarded since BLIF-MV has already an implicit notion of time common to all tables and symbolic latches. The `time` signal is there only for two purposes. First, it explicitly says that all FSMs progress at the same pace set by `time`. The amount of time between consecutive time points is irrelevant. Second, the signal `time` is there for the correctness of Verilog simulation. Note that a Verilog simulator is an event-driven "passive scheduler". A simulator schedules events generated from Verilog modules and then sends them to modules which are sensitive to these events. Those statements whose sensitizing events appear (active statements) are executed which in turn generates more events which are scheduled by the simulator. A simulator itself does not generate any event. Instead, it coordinates between the producers and consumers of events. Hence, in general, when a designer writes such a synchronous system, s/he also needs to write a small clock generator (an event generator, which creates events along the time axis. The produced events invoke a chain of reactions among modules. The system runs into a stable state when there are no more events other than the clocking event. The next clocking event is then chosen by the simulator and simulation time is advanced accordng to the time stamp of the newly scheduled clocking event. A clock generator may be as simple as: `always time = #1 ~time;`, where `~` means bitwise logical inverse. This clock generator generates a clocking signal `time` with a cycle of two time units) in order to drive the whole system and make it simulatable. In this case, we call the system *implicitly clocked* since all transitions are implicitly synchronized by an invisible "time". For an implicitly clocked system, hardware resources (table, wires and/or latches) will not be allocated for a synchronizing variable (`time` in the previous example).

On the other hand, for certain designs/circuits, the operation of a system depends explicitly on several phases (rising edge, falling edge, 1-level, 0-level) of one or more synchronizing signals (which we generally refer to as clocks). Equivalently, designers may write a mixed gate/high-level description of a design in Verilog using gate primitives. In these cases, the clock signals should be interpreted literally and hardware resources should be allocated. We call a design *explicitly clocked* when synchronizing signals are to be compiled literally into hardware.

For implicitly clocked designs, one symbolic latch (or state variable) is allocated for each `reg` variable. Synchronization variables are dropped. For explicitly clocked systems, each `reg` variable is modeled by a symbolic latch along with some extra logic to emulate the clocking mechanism. For example, assume `o` is a register variable,

```
always @(clk)
   if (clk)
      o = a + b;
```

is compiled into



according to explicit semantics. `o_ps` and `o_ns` are the current and next state variables for register `o`, respectively. Refer to section 4.2 for more details of various kinds of edge detectors.

By default, implicit clocking semantics are assumed. Option `-c` or the presence of timing information (delay operators, `#`, etc.) in Verilog makes vl2mv switch to explicit clocking semantics.

4.2 Timed Program = Timing Machines + Untimed Machines

A timed Verilog module can be logically separated into two sets of machines; *timing machines* and *untimed machines*. Timing machines determine how long the program (or the resulting product machine) can stay in a certain state. It controls the timing for updating register and wire variables. Timing machines use the control information (value of logical expressions in conditional statements) from untimed machines along with values of *timers* in BLIF-MVT to determine their next states. Untimed machines use control information and transitions of timing machines to determine whether "hardware" registers/latches self-loop or go to the next state.

4.2.1 System Modeling

Execution of a Verilog program consists of a sequence of two alternating phases, *computation phase* and *idling phase*, as shown in Figure 4.1. Execution of any statement other than pause takes "zero hardware time".

The purpose of the computation phase is to compute the new hardware state while halting the progression of time. In the computation phase, timing machines reset the timers for the next pause where it is going to stay and untimed machines compute next states for `reg` variables. In this phase, untimed machines emulate the execution of "active statements" – the segments of program that will be executed between the current pause and the next one. This is done in the following way. First, continuous assignments calculate their outputs according to their operands until a stable value is reached. Then procedural blocks compute the next states for `reg` variables that are touched by the active statements during the execution. Basically, untimed machines are still timed machines, but there are no timing constraints on the state transitions of those machines. Hardware time does not progress during the computation phase.

The purpose of the idling phase, as opposed to that of the computation phase, is to let time elapse while fixing the hardware state. In the idling phase, hardware time can pass while latches in untimed machines keep self-looping. The purpose of this phase is to let the system stay in a state for some period of time indicated in the corresponding pause. Loosely speaking, hardware status progression and time progression are alternated and not overlapped.

4.2.2 Timing Machines

In this section algorithms for extracting timing machines from Verilog programs are presented. All assume the existence of a control flow graph for each `always` statement. Note that the first time a simulator executes an `always` statement, it must start from the first statement in the procedural block guarded by `always`. It is possible that this statement is not the

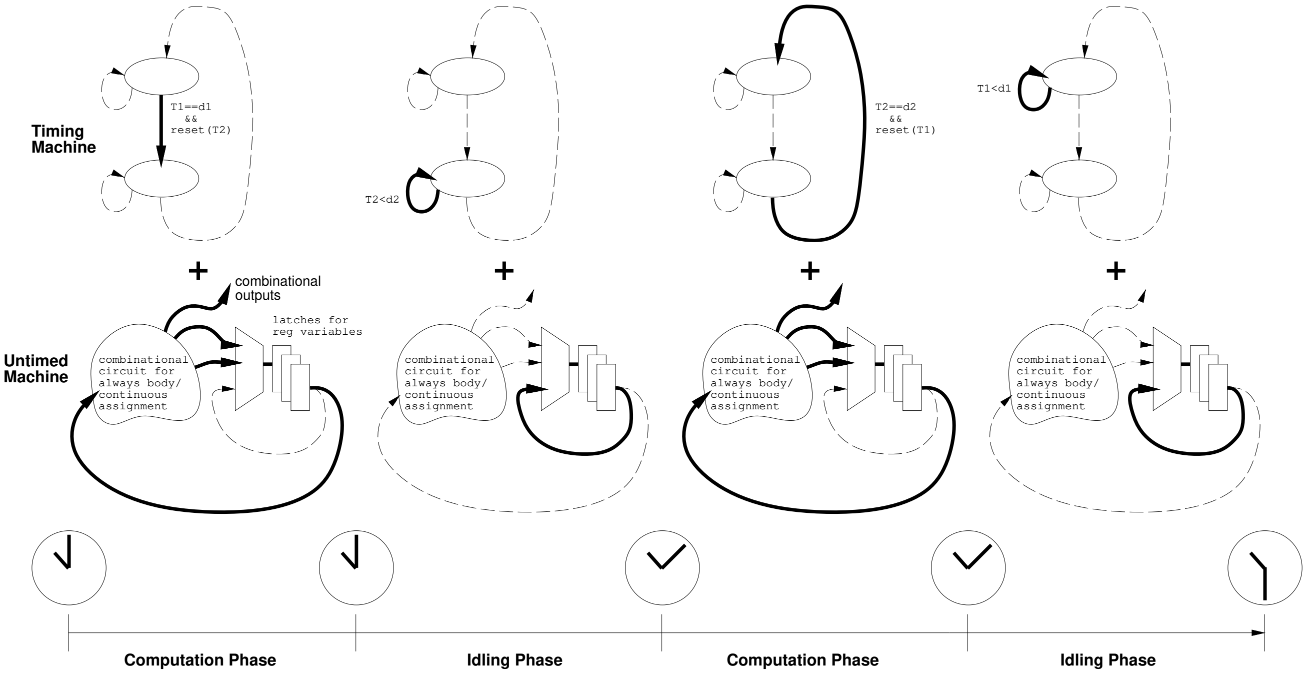


Figure 4.1: System Modeling

first statement that will be executed in the executions following the initial one. Consider for example,

```
always
   statement1;
   # 1
   statement2;
```

A possible execution of the program consists of the sequence `statement1`, `delay`, `statement2`, `statement1`, `delay`, `statement2`, `statement1`, ... When a simulator executes the program for the first time, only `statement1` is executed. Afterwards, `statement2`, `statement1` are always executed in order, without any interrupt. In order to emulate this irregularity, we have to create special circuit/transition tables. In the following two subsections, we first show how to extract timing machines from control flow graphs without considering the initial execution problem. Then we supplement the timing machines with auxiliary states and untimed machines with muxes to emulate initial execution.

`always` Loops

In the following algorithms, p_i denotes a pause node in the control flow graph, and s_i the state corresponding to p_i in the timing machine being generated. In the following sections and figures, p_{ps} and p_{ns} denote the current and next state of a timing machine, respectively.

for each $p_s, p_d \in V_p$ **do**

for each simple path $p : p_s \rightsquigarrow p_d, p \cap (V_p - \{p_s, p_d\}) = \phi$ **do** /* i.e. no pauses in between */

Let $C = \{l \mid l = L((c, v)), c \in V_c, c \in p\}$

if p_s's corresponding delay is of the form `#`δ,

then put a transition $s_s \rightarrow s_d$ labelled with C, $T_s == \delta(p_s)$[1],and $T_d = 0$

and a self-loop $s_s \rightarrow s_d$ labelled with $T_s < \delta(p_s)$[2]

if p_s's corresponding delay is of the form `#`$(\delta_{min} : \delta_{max})$,

then put a transition $s_s \rightarrow s_d$ labelled with C, $\delta_{min}(p_s) \leq T_s \leq \delta_{max}(p_s)$[1], and $T_d = 0$

and a self-loop $s_s \rightarrow s_d$ labelled with $T_s < \delta_{max}(p_s)$[2]

if p_s's corresponds to an edge event control (`@(posedge x)/@(negedge x)/@(x)`),

then put a transition $s_s \rightarrow s_d$ sensitized by the corresponding edge detector

and a self-loop $s_s \rightarrow s_d$ when edge detector gives false

od

od

This naive algorithm investigates all the paths from source pause p_s to destination pause p_d. However, values of insignificant conditional expressions can be regarded as don't cares for the transitions of a timing machine. For example, consider the following two program segments,

```
# 10 icon = DOG; (pause1)
if (showPicture==True)
   image = CAT;
else
   image = PEOPLE;
# 20 icon = FOOD; (pause2)
```

```
# 10 icon = DOG; (pause1)
if (showPicture==True)
   # 4.5 image = CAT; (pause3)
else
   image = PEOPLE;
# 20 icon = FOOD; (pause2)
```

In the left hand side, regardless of the outcome of `showPicture==True`, pause 1 transits to pause 2 when pause1's timer counts to 10. However, in the right hand side, the result of `showPicture==True` is not a don't care anymore. A naive approach considering all the paths from p_s to p_d spends time exponential in the number of non-overlapping insignificant conditional blocks between them on computing paths corresponding to don't cares. In addition, generated BLIF-MV tables will also be extremely large. This can be optimized in the following way. A preprocess can be imposed on the control flow graph to eliminate all insignificant conditional blocks. Insignificant conditional blocks can be eliminated by a bottom-up graph reduction, which takes time proportional to the size of the control flow graph.

graph-reduce (s, a statement)

[1]This kind of timing constraints are called *Time-out-constraints.*

[2]This kind of timing constraints are called *Idling-constraints.*

if (s is a delayed statement) return true; **fi**
if (s is a `if/else`, `case`, or `begin/end`)
  contain-pause = false;
  for each branch-statement/sub-statement s_i in s **do**
    contain-pause = contain-pause **or** graph-reduce(s_i)
  od
  if (contain-pause==false)
    add edge from s's immediate predecessor to s's immediate successor
    remove s and all edges incident to it
  fi
fi

No vertex/edge is examined more than once by the above algorithm. The time complexity of the algorithm is $O(\mid G \mid)$, where G is the control flow graph.

Initialization of an `always` Loop

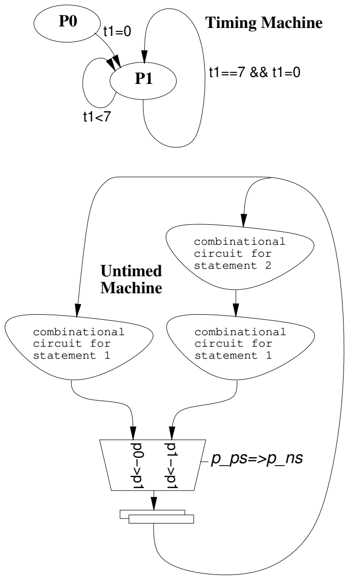
The first time a program gets executed, it is started from the first statement in the `always` statement. However, this may not hold for executions that follows. Hence, some effort to take care of the initial execution of an `always` statement is required.

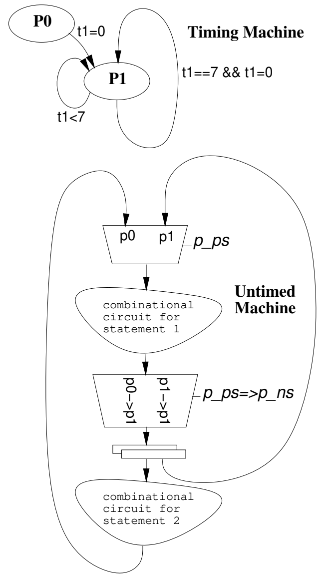
A naive implementation may duplicate the logic for statements just for the initial execution. However, we can create a set of muxes which select the present states of `reg` variables or values assigned by "trailing statements" that are executed before the first statement in a procedural block if there is no pause separating them. The outputs of these muxes are used for program variable references from leading statements inside `always` statements. The set of muxes is controlled by the starting state of the timing machine. The following example shows both alternatives for a Verilog program segment.

Verilog program	Duplicate circuit for initial execution	Use Mux to emulate initial execution

```
always
   statement1;
   # 7
   statement2;
```

$\Rightarrow$





In a control flow graph, for each `always` statement, a distinct dummy node p_0 is introduced which corresponds to the initial execution of a procedural block. The new edge set for the new CFG is the union of edge set E in the original CFG and $E' = \{(p_0, v) \mid$ there is a basic block between `always` and the expression v stands for $\}$. The set of newly introduced dummy nodes are called *initial pauses*. In the timing machine, we put an initial state s_0 which represents the initial run over the always loop. The algorithm in the previous section can be easily adapted to find the initial transition in the timing machine from s_0.

for each $p_d \in V_p$ **do**
    for each simple path $p : p_0 \rightsquigarrow p_d, p \cap (V_p - \{p_d\}) = \phi$ **do**
        Let $C = \{l \mid l = L((c, v)), c \in V_c, c \in p\}$
            put a transition $s_0 \rightarrow s_d$ labelled with C, $T_d = 0$.
    od
od

The techniques in the previous section, used to improve the performance of extracting transition structures of timing machines, can also be applied.

Delay Operators/Edge Event Controls

A delay operator is translated into a pause state s. A distinct BLIF-MVT timer T_p is allocated for each pause except the initial pause. State s keeps self looping until the specified delay time elapses. On exit, the outgoing transition resets the next timer to be used for the next delay operator/edge event control that will be encountered. The delay time in the delay operator

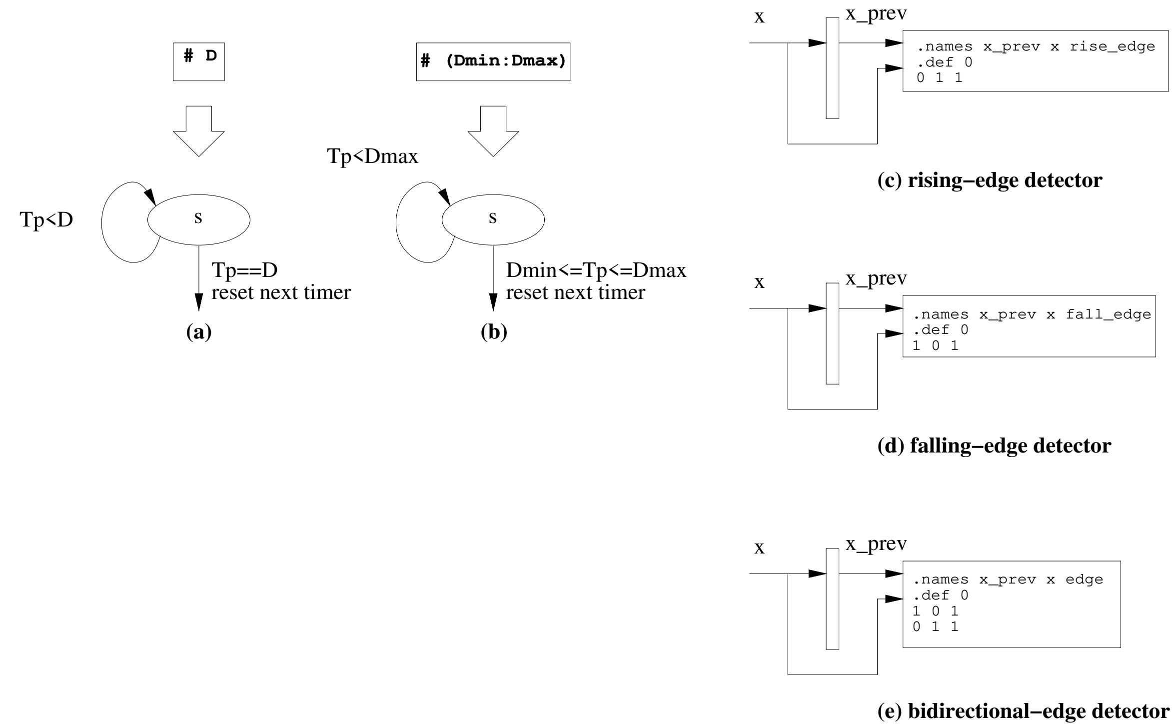


Figure 4.2: (a), (b) Converting delays into timing constraints. (c), (d) Rising/falling edge detectors. (e) Bidirectional edge detector.

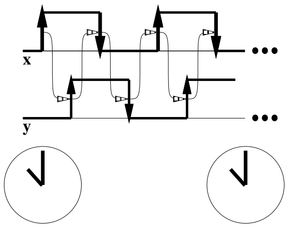
determines the timing constraints associated with transitions out of that pause, as shown in Figure 4.2. Figure 4.2.a shows a translation for a simple delay (`# D`, which means that program has to stay in s for `D` time units), Figure 4.2.b shows a translation for a complex delay (`# (Dmin:Dmax)`, which means that the program has to stay in s for t units, $Dmin \leq t \leq Dmax$).

On the other hand, an edge event control (`@(posedge x)`, `(negedge x)`, or `@(x)`) is translated into an *edge detector* (Figure 4.2, c, d, and e). An edge detector is basically a hidden symbolic latch along with a small table. The hidden symbolic latch stores the value of a signal in the previous instant while the table checks if the desired transition (rising, falling, or bidirectional) happens.

However, a simulator might generate behaviors that are not reproducible by the targeted hardware which use edge detectors to emulate edge event controls. Further, if two concurrent processes are to access the each other's state variables, a simulator can generate behaviors such that one of them can read the updated state variable at the very instant. For timed machines, concurrent processes can not access next state values of the other processes in the same instant. Consider the following two examples, *example 1*

```
always
  if (first)
    begin
    x = 1;
    first = 0;
    end
  else
    begin
    @(posedge y)
    x = 0;
    @(negedge y)
    x = 1;
    end
```

```
always
  begin
  @(posedge x)
  y = 1;
  @(negedge x)
  y = 0;
  end
```



example 2

```
always              always
  #3 x = y;           #3 y = x;
```

Simulated result:
when time>3, either
x overwrites y or
y overwrites x,
but not both (x and y swapped)

In the first example, a simulator can show a sequence where `x` oscillates between 0 and 1, while hardware time is *halted*. On the other hand, for circuits extracted from the subset of timed Verilog, if edge detectors are used to emulate edge event controls, similar behaviors can be reproduced except that hardware time *advances* (in timed machines, time progresses strictly and monotonically). In the second example, two processes try to assign each other's state value to their local state variable at the same time. If the execution is interleaved, one of `x` and `y` gets overwritten, but not both. However, `x` and `y` are swapped in the concurrent machines generated from direct compilation.

Hence, some restrictions are added to prune away these programs that can lead to disagreement between simulator and generated hardware.

- Any state (`reg`) variable should not change its value more than once in any hardware instant.
- Two or more concurrent processes accessing each other's state variable should not modify the local state variable that is accessed by the other process at the same time. If so, at least one of the modifications to the state variables should be made through a non-blocking assignment.

4.2.3 Untimed Machines

The untimed machine for a module is basically similar to the one obtained by removing all delay operators. However, some control points are added in order to make it controllable by the timing machine.

Continuous Assignment

In the synthesizable subset of timed Verilog, it is forbidden to apply any delay operator on continuous assignments. Hence, a continuous assignment is still essentially a combinational logic (with no delay) which can be modeled by pure BLIF-MVT tables without supplementary symbolic latches or timers.

Sequence of Statements

Within a `begin/end` block, a sequence of statements is executed in order. However, due to the pauses inside these statements, they might not be executed in the same hardware time. We refer to the segments of code executed in a particular point of time as *active*. Next state values of timing machines are used to determine which segment of the code is active and should affect the next states of `reg` variables. Segments of combinational circuits for different statements are fed into a *segment selector* which selects the active segments of statements. Due to the hierarchical structures of statements (a statement may be composed of sub-statements), segment selectors also have similar hierarchical structures (Figure 4.3). The following algorithm incrementally builds segment selectors for simple/composite statements. We say that a transition is taken due to *time-out* if that taken transition is labelled with a time-out timing constraint (section 4.2.2) and the constraint is satisfied.

segment-selector (composite-statement)
    for each sub-statement s in composite-statement **do**
        Let p_{ns} be next state of the timing machine.
        if (s is composite)
            p_l = segment-selector(s)
            d_l = set of pauses in s
            add one branch in segment-selector which says
            If there is an inter-pause transition due to time-out and if $p_{ns} \in d_l$
                then value of p_l *is taken.*
            continue;
        fi
        if (s is delayed simple)
            let d be the pause controlling s
            let p be the set of value terminals available for all `reg` variables immediately before d
            add one branch in segment-selector which says
            If there is an inter-pause transition due to time-out and if $p_{ns} == d$
                then value of p *is taken.*
            continue;
        fi

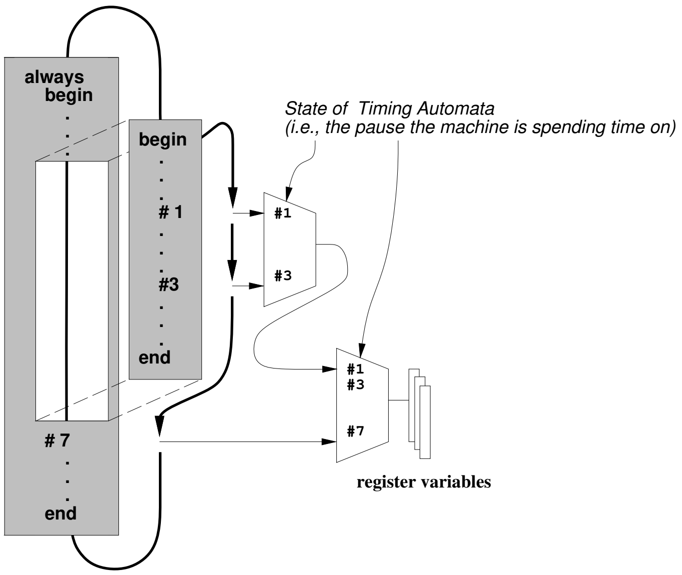


Figure 4.3: Hierarchy of segment selector

    if (s is simple)
        continue;
    fi
od

`if/else` and `case` Conditional Statements

Given that there is no pause in all branches of an `if/else` statement or a `case` statement, extracting a circuit from the conditional statement is straightforward. One mux (which is referred to as a *conditional mux*) is created for each variable and is controlled by the logical expression of the conditional statement. Inputs to the mux correspond to the new values assigned within each branch. The output of a mux is used in further reference to the variable. When the program enters one branch of a conditional statement and stays in a pause of the branch, it must "remember" which branch it was in and preempts the conditional mux appropriately when control is going to be transferred to the statement following the conditional statement. For example,

```
if (state==Eating)
   begin
   month = Feb;
   state = Playing;
   # 3 ;
   end
```

```
6  else
7     month = Mar;
8  x = ...  month ...
```

Assume that the program starts at time unit 0 and state is initially Eating. At the time line 8 refers to the content of variable month, the truth value of the conditional expression state==Eating is changed. So choosing the value of month based on the value of state at time unit 3 (which is Playing) is incorrect. In general, it is predetermined which branch should be taken on executing a statement immediate following an if/else or case statement, hence a conditional mux should interrogate both the current state of a timing machine as well as the outputs of logic expressions of the conditional statements to make its decision, as shown in Figure 4.2.3.(a).

for Loops

A for-loop statement consists of four parts, *initialization*, *loop condition test*, *increment*, and *loop body*. vl2mv does not support general forms of for-loop due to its dynamic nature, though symbolic simulation and fix-point computation might help to find transition relations for it. vl2mv supports two restricted forms of for-loop. First, if the lower-bound, upper-bound, and increment of the loop can be determined at compile-time, then loop unrolling can expand the for loop into a straight-line of codes. Second, if on every simple cycle along the CFG for the loop there is a delay/event-control (i.e., the CFG for the loop is legal), then it is also synthesizable.

for Loop Unrolling

vl2mv can unroll a very simple form of for loop. To use the capability, it is necessary to ensure that all expressions appearing in initialization, loop condition, and increment of the for statement are able to be evaluated during compile-time. What vl2mv does when encountering an expression like for (i=<init>; <cond>; i=<inc>) <loop-body>; is that, it first evaluates <init>, assigns the result to i, then repeatedly evaluates <cond> and <inc>, assigning new value to i according to <inc> until <cond> becomes false. For each value i can have, vl2mv duplicates <loop-body> and substitutes i with its associated value.

Legal for Loops

Figure 4.2.3.(b), (c) gives an abstract translation from a legal for loop into a circuit. Similar to if/else or case statements, conditional muxes for for loops query the current states of timing machines to designate one of their branches as a legal source for variable references from statements in the main-loop or for-loop.

The initialization and increment part of the `for` loop are directly compiled into combinational circuits. Then a test is made to see if the specified condition is satisfied. The test result is used in the timing machine to determine pause transitions.

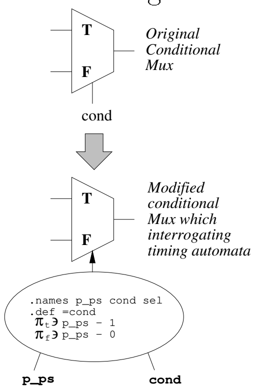


Figure 4.2.3:(a) Conditional mux. p_{ps} is the present state of the timing machine. $\pi_t(\pi_f)$ denotes the set of pauses in the *true* (*false*) branch of a conditional statement which the mux stands for.

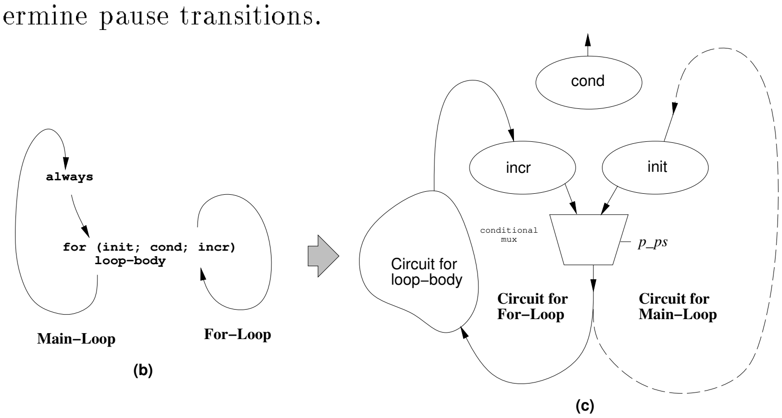


Figure 4.2.3:(b), (c) Circuit for `for` loop

4.3 Example

Figure 4.4 shows an abstracted example Verilog program and its corresponding timed/untimed machine. Note that there are two transitions $P_2 \rightarrow P_2$; $P_2 \xrightarrow{t_2 \le 30} P_2$ (an idling transition) stands for the transition that lets time pass and $P_2 \xrightarrow{20 \le t_2 \le 30 \&\& reset(t_2)} P_2$ (a time-out transition) stands for the transition that updates hardware states. Only the latter can sensitize the $P_2 \rightarrow P_2$ entry in the segment selector.

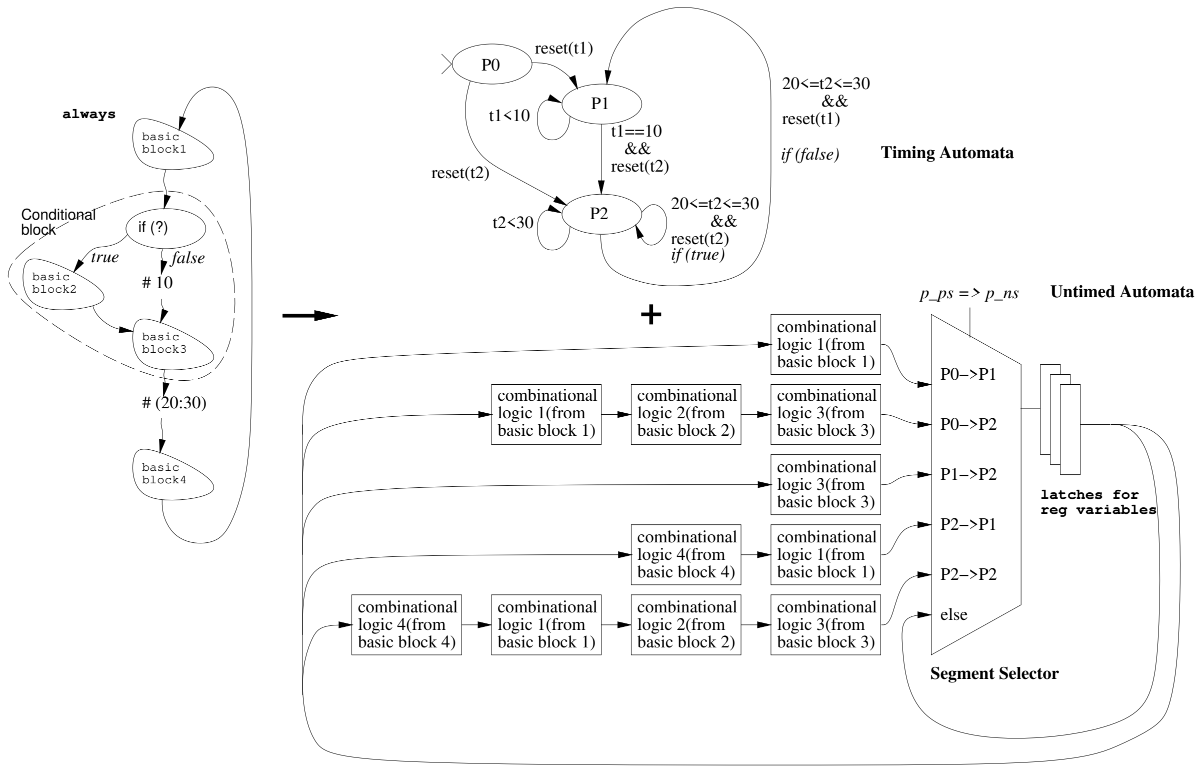


Figure 4.4: Example circuit

Chapter 5

Other Aspects of vl2mv

In this chapter miscellaneous vl2mv functionalities are introduced. In section 5.1 compiler directives that can be handled by vl2mv are detailed. Section 5.2 describes alternative ways in which vl2mv generates BLIF-MV. All features presented in this section can be controlled by compiler options.

5.1 Compiler Directives

5.1.1 Macros

A macro definition is started with the complier directive `` `define `` followed by the name of the macro and then the body of the macro. Invocation of a macro is started with left-quote (`) followed by the name of the macro. For example,

```
`define WIRE wire
`define VAR out
`define BIT_RNG [0:2]
`define ASSIGN assign
`define EQUAL =

`define WANDERING 2
`WIRE `VAR `BIT_RNG ;
`ASSIGN `VAR `EQUAL `WANDERING ;
```

is in effect the same as

```
wire out[0:2] ;
assign out = 2;
```

5.1.2 File Inclusion

To include another file, use the `‘include` directive. For example,

```
‘include /usr/lib/vl2mv/vlr.h
```

includes file `/usr/lib/vl2mv/vlr.h` in place where the directive occurs.

5.2 Other vl2mv Features

5.2.1 Compatibility Checking

Option `-C` makes vl2mv accept only those programs using the syntax defined in [TM91]. It rejects Verilog extensions defined in section 3.1.1 (e.g. enumerated type declaration for symbolic variable).

5.2.2 Abstraction of Operators

By default, high level operators like addition (`+`), subtraction (`-`), greater than (`>`), etc. are compiled into a flattened circuit in place, in the modules where they are used. It might be desirable to abstract these operators as sub-circuits/macros to make the resulting file more readable and compact. `-a` serves the purpose of abstracting high level operators. The existing design hierarchy of the source program is not influenced by operator abstraction. For example, assume that `a`, `b`, and `o` are all 1-bit scalars. `o=a+b` is compiled into circuit shown in Figure 5.1.a without the `-a` option (abstraction flag). The adder becomes a `.subckt` call when the abstraction flag is turned on, refer to Figure 5.1.b. In addition, users can choose to use either `.subckt` or `.macro` to instantiate an abstracted operator by using compiler option `-m` which places `.macro` calls in places where an operator is abstracted. `.macro` expands the sub-circuit in place. The `.subckt` introduces another level of hierarchy.

5.2.3 Table Decomposition for Non-Blocking Assignments

By default, all non-blocking assignments to a variable, along with those guards determining effectiveness of those assignments, are collected into a single large table in the module being processed. Sometimes this results in huge tables if there are many non-blocking assignments to the same wire/register variable, and consequently, degrades the performance of HSIS in processing the BLIF-MV file. In this case, the option `-T <table-width>` may alleviate the problem by decomposing those huge tables for non-blocking assignments into several smaller tables. `<table-width>` denotes the threshold number of nonblocking expressions that can be put into a single final table for non-blocking assignment. For example, consider the following program fragment:

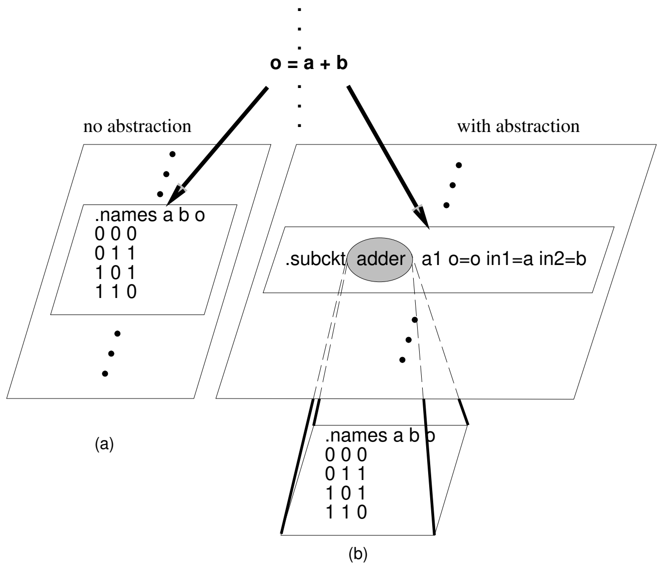


Figure 5.1: Circuit Abstraction

```
always @(posedge clk)
   begin
   if (c==0) begin a <= yellow; a <= green; end;
   if (c==1) begin a <= red; a <= white; end;
   endcase
   end
```

Without the `-T` option, the preceding program is compiled into the circuit as shown in Figure 5.2. With the `-T 2` option, the resulting circuit looks like Figure 5.3 where `a_ps` and `a_ns` represent present and next state variables of register `a`, respectively. The tradeoff is that when a large table is decomposed, many more intermediate variables (e.g., `set0`, `set1`, `t0`, `t1`, etc.) are introduced temporarily. However, huge number of intermediate variables may also decrease the performance of HSIS when reading in BLIF-MV.

5.2.4 HSIS System Calls

For some functions like `and`, `or`, `add`, etc., it is well-known how to build BDDs [Bry86], [BRB90] for them efficiently. Users can call supported MDD [KB90] functions directly to speedup the BDD building process. vl2mv can compile `and`, `nand`, `or`, `nor`, `xor`, `xnor`, `add`, and `minus` into MDD calls of the form `_hsis_<function_name>` by giving the compiler option `-h`. HSIS system calls can also be viewed as operator abstraction except that vl2mv does not write down the definitions for these operators. HSIS knows the definition of these abstracted operators and has MDD constructor routines to build MDDs for them efficiently. For example, assume that

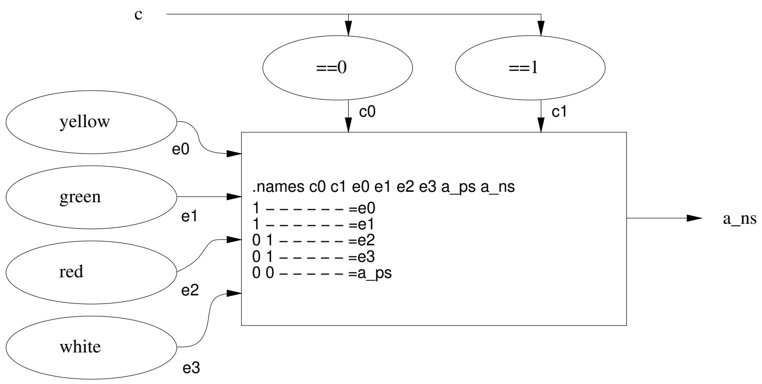


Figure 5.2: Single Table Non-blocking Assignment

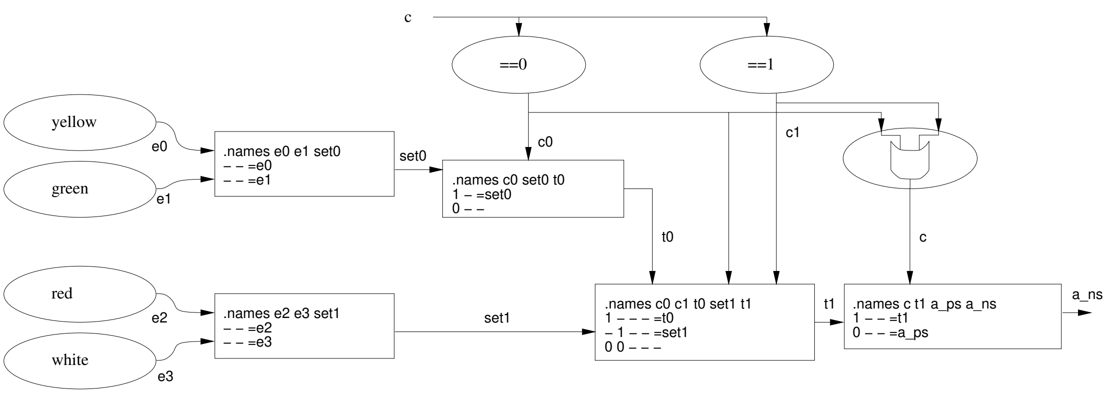


Figure 5.3: Multiple Table Non-blocking Assignment

x, y and o are single bit register variables. For various vl2mv options, the following BLIF-MV tables (table 5.1) are generate.
However, this BLIF-MV file is only useful to the HSIS users because only HSIS has these MDD constructor programs available.

5.3 Source Debugging Support

Generated BLIF-MV tables can be annotated with debugging information, such as the line number in the source at which the assignment is made, the variable being touched by the assignment, and the context in which the assignment is made. By providing the compiler option -g, vl2mv puts debugging information in the generated file. We intend to use this in the future to provides an enhanced debugging environment and a complete linkage between the front-end HDL and the powerful back-end verification/synthesis engine.

Compiler option given	BLIF-MV generated

without `-h`, `-a`	`.c_in`
	`0`
	`.names x y c_in o`
	`.def 0`
	`0 0 1 1`
	`0 1 0 1`
	`1 0 0 1`
	`1 1 1 1`

with `-h` option	`.subckt _hsis_add_1_1 adder o=o i1=x i2=y`

with `-a` option	`.subckt __plus<<1$1>> adder o=o a=x b=y`
	`.`
	`.`
	`.`
	`.model __plus<<1$1>>`
	`.inputs a, b`
	`.outputs o`
	`.names c_in`
	`0`
	`.names a b c_in o`
	`.def 0`
	`0 0 1 1`
	`0 1 0 1`
	`1 0 0 1`
	`1 1 1 1`
	`.end`

Table 5.1: BLIF-MV generated from code fragment `o=a+b;`

Chapter 6

Conclusions

vl2mv addresses the problem of compiling a subset of a HDL into a set of automata. With it, engineers can design in high level Verilog HDL and still have state of the art verification/synthesis algorithms to help verify and optimize designs. With this unified power, hopefully design errors can be detected and eliminated before fabrication and circuits can be optimized automatically so that manufacturers can reduce the turn-around time and increase productivity as well as the reliability of their products.

6.1 Future Work

This project has addressed, among other things, the problem of expressing a subset of the Verilog timing constructs to HSIS. So far, we can handle blocking timing constructs. Algorithms presented in this report has been incorporated into vl2mv, which bridges the gap between a Verilog design (of course, in the synthesizable subset) and HSIS. A delayed non-blocking statement, like `#7 x <= y;`, (which says that 7 time units later, `x <= y` will be executed; `x` will get `y`'s content at time 7^+) is allowed in the synthesizable subset for vl2mv. However, for non-blocking delayed statement like `x <= #7 y;` (which says `x` is going to be assigned to the value of current `y` after 7 time units, without blocking program execution), it is still an open question how to express this in timed automata. One timer per pause scheme is not valid in the context of non-blocking delayed statements. Actually, the number of timers that should be allocated might depend on the program context. For example,

```
always
   begin
   x <= #17 y;
   #7 y = y + 1;
   end
```

Since there might be 3 active copies of `x <= #17 y;` running concurrently, at least 3 timers are needed for the same statement. In addition, how and when to activate each timer for that particular statement raises another problem. Further research is required to address the feasibility of compiling such constructs.

RQ timed automata [LB93], [LB94] is a subclass of timed automata which imply the *simple path property* that makes it efficient to rule out paths which can not meet timing constraints. A lot of practical examples are known to have the simple path property. It is desirable to know if the timed synthesizable subset of Verilog for vl2mv can be compiled into set of timed machines having the simple path property.

Appendix A

Syntax

In the synthesizable subset of timed Verilog, a delay operator can not be applied to continuous assignments and primitive gates. No non-blocking delayed statement is allowed. Delay operators can be applied to either procedural statements or non-blocking assignments. All delays should be positive. Only one `always` statement per module is allowed if the program is timed.

The following grammar is defined in Extended Backus-Normal Form (BNF). Capitalized words represent terminal tokens. Words in angle brackets represent non-terminals. `::=` is read as "is defined as". `|` is read as "or". `*` represents Kleene closure of the preceding terminal/non-terminal. `+` represents one or more repetitions of the preceding terminal/non-terminal. `?` represents zero or one copy of the preceding terminal/non-terminal. Other symbols (e.g., "(", ")", "@", ",", ";", etc.) represent tokens literally as they are in the input stream.

```
<program> ::= <module_or_type>*

<module_or_type> ::= <module> | <type>

<module> ::= MODULE <id> ( <port>* ) ; <module_items> ENDMODULE

<type> ::= TYPEDEF <type_spec> <type_name> ;

<type_spec> ::= ENUM <id>? { <enumerator_list> }

<type_name> ::= <id>

<enumerator_list> ::= <enumerator> | <enumerator_list> , <enumerator>

<enumerator> ::= <id> | <id> = <expression>
```

```
<port> ::= <id>

<module_items> ::= <module_item>* <always_statement>

<module_item> ::= <continuous_assignment>
                | <reg_declaration>
                | <net_declaration>
                | <gate_instantiation>
                | <module_instantiation>

<module_instantiation> ::= <id> ( <port>* )

<gate_instantiation> ::= AND ( <port>+ )
                       | NAND ( <port>+ )
                       | OR ( <port>+ )
                       | NOR ( <port>+ )
                       | XOR ( <port>+ )
                       | XNOR ( <port>+ )
                       | XNOT ( <port> )

<always_statement> ::= ALWAYS <statement>

<continuous_assignment> ::= ASSIGN <assignment> ;

<reg_declaration> ::= <range>? <list_of_vars> ;

<net_declaration> ::= <range>? <list_of_vars> ;

<assignment> ::= <lhs> = <expression>
               | <lhs> <= <expression>

<statement> ::= BEGIN <list_of_statements> END
              | <assignment>
              | <delayed_statement>
              | <event_control_statement>
              | IF ( <expression> ) <statement>
              | IF ( <expression> ) <statement> ELSE <statement>
              | CASE ( <expression> ) <case_item>+ ENDCASE
```

```
<event_control_statement> ::= @ ( event_expression )

<event_expression> ::= POSEDGE IDENTIFIER
                     | NEGEDGE IDENTIFIER
                     | IDENTIFIER
                     | <event_expression> OR <event_expression>

<delayed_statement> ::= <delay_control> <statement>

<delay_control> ::= # NUMBER
                  | # ( NUMBER : NUMBER )

<expression> ::= IDENTIFIER
               | NUMBER
               | UNARY_OPERATOR <expression>
               | <expression> BINARY_OPERATOR <expression>
               | <expression> '?' <expression> : <expression>
               | SYSTEM_NDSET ( <set_components> )

<case_item> ::= <expressions> : <statement>
             | DEFAULT : <statement>

<list_of_vars> ::= IDENTIFIER
                 | <list_of_vars> , IDENTIFIER

<list_of_statements> ::= <statement>
               | <statement> ; <list_of_statement>

<set_components> ::= <set_component>
                   | <set_components> , <set_component>

<set_component> ::= <expression>
```

Bibliography

[ABB+94] Adnan Aziz, Felice Balarin, Robert K. Brayton, Szu-Tsung Cheng, Ramin Hojati, Sriram C. Krishnan, Rajeev K. Ranjan, Alberto L. Sangiovanni-Vincentelli, Thomas R. Shiple, Vigyan Singhal, Serdar Tasiran, and Huey-Yih Wang. HSIS: A BDD-based environment for formal verification. In *DAC94*, San Diego, CA, June 1994.

[BBC+] Felice Balarin, Robert K. Brayton, Szu-Tsung Cheng, Desmond A. Kirkpatrick, Alberto L. Sangiovanni-Vincentelli, and Ephrem Wu. A verification tool for real-time systems. Submitted to ICCAD'95.

[BCH+91] R. K. Brayton, M. Chiodo, R. Hojati, T. Kam, K. Kodandapani, R. P. Kurshan, S. Malik, A. Sangiovanni-Vincentelli, E. M. Sentovich, T. Shiple, K. J. Singh, and H.-Y. Wang. BLIF-MV: An interchange format for design verification and synthesis. Memorandum UCB/ERL M91/97, University of California at Berkeley, 1991.

[BRB90] K. S. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a bdd package. In *Proceedings of the 27th Design Automation Conference*. IEEE, 1990.

[Bry86] R. E. Bryant. Graph based algorithms for boolean function manipulation. In *IEEE Transaction on Computers, C-35(8)*. IEEE, 1986.

[BY93] Felice Balarin and Gary York. Verilog HDL modeling styles for formal verification. In *CHDL*. North-Holland, 1993.

[Eme90] E. A. Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science*. Elsevier Science Publishers, 1990.

[KB90] Timothy Kam and Robert K. Brayton. Multi-valued decision diagrams. Memorandum UCB/ERL M90/125, University of California at Berkeley, 1990.

[KR78] Brian W. Kernighan and Dennis M Ritchie. *The C Programming Language*. Prentice-Hall, Inc., 1978.

[LB93] William Lam and Robert K. Brayton. Alternating RQ timed automata. In *International Conference on Computer Aided Verification*, 1993.

[LB94] William Lam and Robert K. Brayton. Criteria for the simple path property in timed automata. In *Computer Aided Verification*, 1994.

[Sea92] Ellen M. Sentovish and et al. Sequential circuit design using synthesis and optimization. In *Proceedings of IEEE ICCD*. IEEE, October 1992.

[TM91] Donald E. Thomas and Philip R. Moorby. *The Verilog Hardware Description Language*. Kluwer Academic Publishers, Nowell, Massachusetts, 1991.

[Ver] Berkeley Verifiers. A property interchange format for verification using hsis. A part of HSIS document.