Event and Cycle Semantics of Hardware Description Languages

Mike Gordon
University of Cambridge Computer Laboratory

Version 1.4

September 1, 1999

This unfinished paper describes work in progress. It gives some details behind my Banff99 talk, though it is now slightly dated.

The work reported here is part of the Verilog Formal Equivalence (VFE) Project funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC).

Contents

1 Introduction **3**

2 Syntax **4**

2.1 Expressions 4

2.2 Event expressions 4

2.3 Statements 5

2.4 Abbreviations 5

2.4.1 Case statements 5

2.4.2 Continuous assignments 6

3 Semantic Pseudo-Code **6**

3.1 Pseudo-code instructions 6

3.2 The size of a statement 7

3.3 Translation algorithm 7

3.4 Example translations 8

4 Event Semantics **9**

4.1 Simulation algorithm 10

5 Trace semantics **13**

5.1 Combinational programs 13

5.1.1 Transparent latches 19

5.1.2 Summary 20

5.2 Flip-flops 21

5.3 RTL programs 23

Bibliography **25**

1 Introduction

Hardware description languages (HDLs) like VHDL and Verilog have a simulation oriented semantics based on events, i.e. changes to the values of wires and registers. This *event semantics* can accurately model detailed asynchronous behavour, but is very fine-grained and does not easily support formal verification.

Most practical formal methods (e.g. model checking and theorem proving) are oriented towards descriptions of systems in terms of their execution traces, which are sequences of states. One might characterise simulation semantics as 'edge-oriented' and trace semantics as 'level-oriented'. The relationship between the two views is obtained by accumulating the changes (events) during a simulation cycle to obtain the state holding at the end of the cycle. The sequence of states that the simulation cycle quiesces to at successive instants of simulation time is an abstraction called the *trace semantics*. If there are race conditions, then there may be several possible successor states to a given state (i.e. branching time). Trace semantics has the same timescale as event (simulation) semantics – namely simulation time – but abstracts away from the individual events within a single simulation cycle (delta-time).

Clocked sequential systems can also be viewed more abstractly in terms of the sequence of states held in registers during successive clock cycles. This view is called the *cycle semantics*. Certain kinds of hardware (e.g. transparent level sensitive latches) are rather badly approximated if only the states latched at clock edges are considered, so equivalences between such hardware is best done with trace semantics.

A cycle semantics determines the exact clock cycle that each register transfer occurs on. A more abstract view is the *behavioural semantics* [2] in which certain state transitions are regarded as not observable. With respect to behavioural semantics, functional equivalence is preserved by moving operations across certain sequences of clock cyles (e.g. within the same 'super state').

2 Syntax

The language described below is intended to embody the essence of the core of synthesisable Verilog, but with a lighter syntax.

A program consists of a set of concurrently executing non-terminating threads:

```
forever S1
forever S2
   ⋮
forever Sn
```

where $\mathcal{S}_1$, $\mathcal{S}_2$, ... , $\mathcal{S}_n$ are *statements*, which are built out of *expressions* and *event controls*.

Each thread[1] models a piece of hardware, with interconnection specified by coincidence of variable names.

2.1 Expressions

Expressions are built up from variables and constants using operators (unary, binary, conditional etc.). The metatavariables $\mathcal{V}$ and $\mathcal{E}$ will range over variables and expressions, respectively.

The structure of expressions is not specified in detail here. It is assumed that there are two Boolean constants `1` (*true* or *high*) and `0` (*false* or *low*), an infixed binary equality operator `=` and a conditional $\mathcal{E}?\mathcal{E}_1:\mathcal{E}_2$ (*if* $\mathcal{E}$ *then* $\mathcal{E}_1$ *else* $\mathcal{E}_2$). In examples, the C-like expression syntax of Verilog will be used.

It is assumed that each $\mathcal{E}$ expression has a *value* with respect to an assignment of values to each variable occurring in it. If s is such an assignment of values to variables, then the value of $\mathcal{E}$ is denoted by $[\![\mathcal{E}]\!]s$.

2.2 Event expressions

Event expressions $\mathcal{T}$ only occur as components of event controls `@(`$\mathcal{T}$`)`. They can be used both to delimit cycle boundaries and to specify combinational

[1] Threads correspond to `always`-blocks in Verilog.

logic.

$\mathcal{T}$	::=	$\mathcal{V}$	(Change of value)
	|	`posedge` $\mathcal{V}$	(Positive edge)
	|	`negedge` $\mathcal{V}$	(Negative edge)
	|	$\mathcal{T}_1$ `or` $\cdots$ `or` $\mathcal{T}_n$	(Compound sensitivity list)

2.3 Statements

The syntax of statements $\mathcal{S}$ is given by the BNF below.

$\mathcal{S}$	::=	$\mathcal{V}_1, \ldots, \mathcal{V}_n$ `:=` $\mathcal{E}_1, \ldots, \mathcal{E}_n$	(Assignment)
	|	$\mathcal{S}_1$`;` $\cdots$ `;` $\mathcal{S}_n$	(Sequencing block)
	|	`if` $\mathcal{E}$ `then` $\mathcal{S}_1$ {`else` $\mathcal{S}_2$}	(Conditional)
	|	`@(`$\mathcal{T}$`)` $\mathcal{S}$	(Event control)

In an assignment $\mathcal{V}_1, \ldots, \mathcal{V}_n$ `:=` $\mathcal{E}_1, \ldots, \mathcal{E}_n$, the variables $\mathcal{V}_1$, ... , $\mathcal{V}_n$ must be distinct. Each pair $(\mathcal{V}_i, \mathcal{E}_i)$ is called an *update*. The assignment is executed by updating in parallel each $\mathcal{V}_i$ with the value of $\mathcal{E}_i$.

Sometimes the notation $\overline{\mathcal{V}}$ will be used for the vector of variables $\mathcal{V}_1$, ... , $\mathcal{V}_n$ and $\overline{\mathcal{E}}$ for the vector of expressions $\mathcal{E}_1$, ... , $\mathcal{E}_n$.

2.4 Abbreviations

The following constructs are derived in terms of the syntax given above.

2.4.1 Case statements

`case` ($\mathcal{E}$)
  $\mathcal{E}_1$`:` $\mathcal{S}_1$
  $\mathcal{E}_2$`:` $\mathcal{S}_2$
  $\vdots$
  $\mathcal{E}_n$`:` $\mathcal{S}_n$
  {`default:` $\mathcal{S}_{n+1}$}
`endcase`

abbreviates

`if (`$\mathcal{E}$`=`$\mathcal{E}_1$`)` $\mathcal{S}_1$ `else if (`$\mathcal{E}$`=`$\mathcal{E}_2$`)` $\mathcal{S}_2$ $\cdots$ `else if (`$\mathcal{E}$`=`$\mathcal{E}_n$`)` $\mathcal{S}_n$ {`else` $\mathcal{S}_{n+1}$}

2.4.2 Continuous assignments

`assign` $\overline{\mathcal{V}}$ `=` $\overline{\mathcal{E}}$

abbreviates

`forever @(`$\mathcal{V}_1$ `or` $\cdots$ `or` $\mathcal{V}_r$`)` $\overline{\mathcal{V}}$ `:=` $\overline{\mathcal{E}}$

where $\mathcal{V}_1, \ldots, \mathcal{V}_r$ are the variables occurring in any component expression of the vector $\overline{\mathcal{E}}$.

Note there there is a subtle difference between these continuous assignments and the ones in Verilog. However, the HDL used here is not rich enough for this difference to be significant.

3 Semantic Pseudo-Code

To simplify the presentation of the semantics, each thread is first compiled to a sequence of assembler-like pseudo instructions.

3.1 Pseudo-code instructions

Statements are compiled to pseudo-code consisting of sequences of instructions from the following instruction set:

$\overline{\mathcal{V}}$ `:=` $\overline{\mathcal{E}}$	assignment
`@(`$\mathcal{T}$`)`	event control
`go` n	unconditional jump to instruction n
`ifnot` $\mathcal{E}$ `go` n	jump to instruction n if $\mathcal{E}$ is not true

3.2 The size of a statement

The size function defined in this section is used in the translation algorithm described in 3.3. Let the size $|\mathcal{S}|$ of $\mathcal{S}$ be as defined below inductively on the structure of $\mathcal{S}$. It will turn out that $|\mathcal{S}|$ is the number of instructions that $\mathcal{S}$ is translated to.

$$
\begin{array}{ll}
|\overline{\mathcal{V}} \texttt{ := } \overline{\mathcal{E}}| & = 1 \\
|\mathcal{S}_1 \texttt{;} \cdots \texttt{;} \ \mathcal{S}_n| & = |\mathcal{S}_1| + \ \cdots \ + |\mathcal{S}_n| \\
|\texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}| & = |\mathcal{S}| + 1 \\
|\texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2| & = |\mathcal{S}_1| + |\mathcal{S}_2| + 2 \\
|\texttt{@(}\mathcal{T}\texttt{)}| & = 1
\end{array}
$$

3.3 Translation algorithm

The sequence $\langle \mathtt{i}_0, \ldots, \mathtt{i}_n \rangle$ of instructions that statement $\mathcal{S}$ is translated to is denoted by $\langle\!\langle \mathcal{S} \rangle\!\rangle p$, where p is the position of the first instruction (e.g. `go` p jumps to the start of the program).

To handle sequential blocks, it is convenient to define in parallel the translation of a sequence $\langle \mathcal{S}_1, \ldots, \mathcal{S}_N \rangle$ of statements (see the third and forth clauses of the definition below). The symbol $\frown$ denotes sequence concatenation.

$$
\begin{array}{ll}
\langle\!\langle \overline{\mathcal{V}} \texttt{ := } \overline{\mathcal{E}} \rangle\!\rangle p & = \langle \overline{\mathcal{V}} \texttt{ := } \overline{\mathcal{E}} \rangle \\
\langle\!\langle \mathcal{S}_1 \texttt{; } \mathcal{S}_2 \texttt{; } \ldots \texttt{; } \mathcal{S}_n \rangle\!\rangle p & = \langle\!\langle \mathcal{S}_1 \rangle\!\rangle p \frown \langle\!\langle \mathcal{S}_2 \texttt{; } \ldots \texttt{; } \mathcal{S}_n \rangle\!\rangle (p + |\mathcal{S}_1|) \\
\langle\!\langle \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S} \rangle\!\rangle p & = \langle \texttt{ifnot } \mathcal{E} \texttt{ go } p + |\mathcal{S}| + 1 \rangle \frown \langle\!\langle \mathcal{S} \rangle\!\rangle (p+1) \\
\langle\!\langle \texttt{if } \mathcal{E} \texttt{ then } \mathcal{S}_1 \texttt{ else } \mathcal{S}_2 \rangle\!\rangle p & = \langle \texttt{ifnot } \mathcal{E} \texttt{ go } p + |\mathcal{S}_1| + 2 \rangle \\
 & \quad \frown \langle\!\langle \mathcal{S}_1 \rangle\!\rangle (p+1) \\
 & \quad \frown \langle \texttt{go } p + |\mathcal{S}_1| + |\mathcal{S}_2| + 2 \rangle \\
 & \quad \frown \langle\!\langle \mathcal{S}_2 \rangle\!\rangle (p + |\mathcal{S}_1| + 2) \\
\langle\!\langle \texttt{@(}\mathcal{T}\texttt{)} \ \mathcal{S} \rangle\!\rangle \ p & = \langle \texttt{@(}\mathcal{T}\texttt{)} \rangle \frown \langle\!\langle \mathcal{S} \rangle\!\rangle (p+1)
\end{array}
$$

A thread `forever` S is translated into the infinite loop:

$$\langle\!\langle \mathcal{S} \rangle\!\rangle 0 \frown \langle \texttt{go } 0 \rangle$$

3.4 Example translations

```
if ℰ
 then
  a := b;
  @(posedge clk) b := a;
 else
  a := b;
  b := a;
```

```
0:   ifnot ℰ go 5
1:   a := b
2:   @(posedge clk)
3:   b := a
4:   go 7
5:   a := b
6:   b := a
```

```
forever
 @(b or c) a := b + c;
```

```
0:   @(b or c)
1:   a := b + c
2:   go 0
```

```
forever
  @(posedge clk)
   total := data;
  @(posedge clk)
   total := total + data;
  @(posedge clk)
   total := total + data;
```

```
0:   @(posedge clk)
1:   total := data
2:   @(posedge clk)
3:   total := total + data
4:   @(posedge clk)
5:   total := total + data
6:   go 0
```

```
forever
 @(posedge clk)
  case state of
  0:
   total := data;
   state := 1;
  1:
   total := total + data;
   state := 2;
  default:
   total := total + data;
   state := 0;
```

```
 0:   @(posedge clk)
 1:   ifnot state = 0 go 5
 2:   total := data
 3:   state := 1
 4:   go 11
 5:   ifnot state = 1 go 9
 6:   total := total + data
 7:   state := 2
 8:   go 11
 9:   total := total + data
10:   state := 0
11:   go 0
```

4 Event Semantics

The event semantics is specified via a simulation cycle that non-deterministically executes threads according to the event controls present.

Consider the following program:

```
forever S_1
forever S_2
   ⋮
forever S_n
```

An *input* is a variable that occurs in one of $\mathcal{S}_1, \ldots, \mathcal{S}_n$, but does not occur on the left hand side of an assignment.

A *state variable* is a variable that occurs on the left hand side of an assignment. The values of the state variables are the *outputs* of the program.

It is assumed that each input is driven by the environment with a sequence of values – the elements of the sequence being the values at successive instants of simulation time. When time increases, the values being input at the new time may change from their previous value. This change may cause an event expression $\mathcal{T}$ to 'fire' and any threads waiting on `@(`$\mathcal{T}$`)` will then be enabled for execution. The simulation cycle (see 4.1 below) consists of repeatedly choosing an enabled thread, executing it until an event control is reached (an atomic step), and then enabling any new threads that fire. If several threads are simultaneously enabled, then the choice of which thread to advance is non-deterministic. If a state is reached in which there are no enabled threads, then the simulation cycle is said to have *quiesced*. When this happens the simulation time is incremented, the next set of inputs is assigned to the input variables and the whole process repeats.

Thus sequences of values on the inputs non-deterministically generate sequences of states (and hence a sequence of outputs). Such a sequence is called a *trace* and has the form:

$$\langle i_0, s_0\rangle\langle i_1, s_1\rangle\langle i_2, s_2\rangle \ \cdots$$

where s_0 is an initial state and $i_0\, i_1\, i_2 \ldots$ are inputs.

4.1 Simulation algorithm

The state of a thread during simulation consists of:

- the simulation time (a non-negative integer);
- the values of all variables;
- the value of the program counter (denoted by *pc*).

It is assumed that the environment to the program supplies a value for each input at each instant of simulation time.

The simulation algorithm can be succinctly described if some terminology is introduced.

The instruction pointed to by the program counter of a thread is called the *current instruction*.

A thread is *enabled* if its current instruction is not an event control `@(`$\mathcal{T}$`)`. Enabled threads can be executed immediately. Once a thread starts executing it continues until an event control is reached. This execution is atomic and is called *advancing* the thread (see below)

A thread is *listening* if its current instruction is an event control `@(`$\mathcal{T}$`)`. Note that if a thread is not enabled then it is listening.

When a variable or input changes, an event expression may be *triggered*:

- $\mathcal{V}$ is triggered if the new value of $\mathcal{V}$ differs from the previous one;
- `posedge` $\mathcal{V}$ is triggered if the new value of $\mathcal{V}$ is `1` and the previous value was not `1`;
- `negedge` $\mathcal{V}$ is triggered if the new value of $\mathcal{V}$ is `0` and the previous value was not `0`;
- $\mathcal{T}_1$ `or` $\cdots$ `or` $\mathcal{T}_n$ is triggered if any of the $\mathcal{T}_i$ are triggered ($1 \leq i \leq n$).

A thread is *triggered* if it is listening and the event control of the current instruction is triggered. A triggered thread is *fired* by incrementing its program counter.

If a thread is not executing, but is enabled, then its current instruction cannot be an unconditional jump, but must be an assignment or a conditional jump. This is because, by the definition of $\langle\langle \mathcal{S} \rangle\rangle$, unconditional jumps cannot occur immediately after event controls, so when a thread is triggered the current instruction cannot become an unconditional jump.

Threads are executed concurrently. When an enabled thread is run it advances without interruption until an event control is reached. Thus a thread never stops running at an unconditional jump `go` n. To see the need for this consider:

Thread 1

```
forever @(inp) w := !inp;

0: @(inp)
1: w := !inp
2: go 0
```

Thread 2

```
forever @(w or inp) outp := w & inp;

0: @(w or inp)
1: outp := w & inp
2: go 0
```

If Thread 2 could stop at instruction 2, then Thread 1 could interleave at that point and update to `w` whilst Thread 2 is not listening. By forcing all jumps to be taken immediately, this possibility is precluded.

The execution of a single instruction is performed as follows:

Instruction Execution

- $\mathcal{V}_1, \ldots, \mathcal{V}_n$:= $\mathcal{E}_1, \ldots, \mathcal{E}_n$
  For $1 \leq i \leq n$ change the $\mathcal{V}_i$ component of the state to the value of $\mathcal{E}_i$ in the current state, note all variables whose value changes, then increment the program counter;

- `go` n
  Set the program counter to n;

- `ifnot` $\mathcal{E}$ `go` n
  If $\mathcal{E}$ is true in the current state then increment the program counter, otherwise set it to n.

When a thread fires these operations are repeated until an event control is reached and then any triggered threads (including possibly the one just

executed) are fired. This process is called advancing a thread.

Advancing a Thread

- A thread is advanced by repeatedly executing the current instruction until an event control is reached.
- This execution is considered to be an atomic step and is not interruptable.
- As a thread is advancing, the variables that are modified by assignments are noted.
- When the thread stops advancing any listening threads that are guarded by event controls containing modified variables are fired.

Note that states at which threads stop advancing are those for which the current instruction is an event control. Such states are called *stopping states*. When reasoning about the execution of programs, it is sufficient to just consider the transitions of threads between stopping states, since these are the uninterruptible atomic actions of the thread. The decomposition of these actions into smaller ones, like unconditional jumps, is just an artifact of the way the event semantics is presented.

Inititially simulation time is 0, each program counter is set to 0 and each variable has some initial value (which is not specified here, but would typically be an 'undefined' value, like `x` in Verilog). The simulation algorithm consists of iterating forever the simulation step shown in the box below.

Simulation Step

- **There is an enabled thread:**

  Non-deterministically choose an enabled thread and advance it.

- **No enabled threads – i.e. all threads listening:**

  Increment the simulation time, read in new input values and fire any triggered threads.

5 Trace semantics

For each program, initial state and sequence of inputs, the simulation algorithm determines a set of traces. Each member of such a trace consists of the inputs at each time together with a state the simulation cycle quiesces to. Note that since simulation is non-determinishtic, there may be several possible quiescent states (or none). Thus time is branching.

The simulation algorithm is hard to work with. In this section it is shown how a the traces of a number of common kinds of programs can be derived more directly.

5.1 Combinational programs

The idea of a combinational program is that it computes its outputs instantaneously from its inputs. A syntactic condition that ensures this will be defined. In preparation, consider a thread having the form `forever @(`$\mathcal{T}$`)`$\mathcal{S}$, where $\mathcal{S}$ contains no event controls. This translates to pseudo code of the form:

```
0: @(T)
⋮
n: go 0
```

where the pseudo instruction `1`, `2`, ..., $n-1$ are assignments or jumps (i.e. not event controls).

Since threads execute atomically from event control to event control, it is clear that such a pseudo program is equivalent to one of the form:

```
0: @(T
1: V := E
2: go 0
```

For example,

```
forever @(y or z) x := 1;
                  if x>y then x,y := y,x else y,z := 3,y
                  z := x+y
```

translates to:

```
0: @(y or z)
1: x := 1
2: ifnot x>y go 5
3: x,y := y,x
4: go 6
5: y,z := 3,y
6: z := x+y
7: go 0
```

which is equivalent to:

```
0: @(y or z)
1: x,y,z :=
    x>y ? y : 1,
    x>y ? 1 : 3,
    (x>y ? y : 1) + (x>y ? 1 : 3)
2: go 0
```

A formal description of the straightforward conversion of combinational threads into a single parallel assignment, as illustrated by this example, is not given here.

A *weakly combinational* program has the form:

`forever @(`$\mathcal{T}_1$`)` $\overline{\mathcal{V}}_1$`:=`$\overline{\mathcal{E}}_1$ $\cdots$ `forever @(`$\mathcal{T}_n$`)` $\overline{\mathcal{V}}_n$`:=`$\overline{\mathcal{E}}_n$

and satisfies three conditions:

1. $\mathcal{T}_i$ consists of all the variables occurring in $\overline{\mathcal{E}}_i$ `or`-ed together;
2. no variable occurs more than once on the left of an assignment (no clashing assignments);
3. after performing any update $(\mathcal{V}, \mathcal{E})$, the value of $\mathcal{V}$ in the resulting state equals the value of $\mathcal{E}$.

Updates satisfying condition 3 above are called idempotent. More formally, $(\mathcal{V}, \mathcal{E})$ is *idempotent* if

$$[\![\mathcal{V}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s]) = [\![\mathcal{E}]\!]s = [\![\mathcal{E}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

A sufficient condition that $(\mathcal{V}, \mathcal{E})$ be idempotent is that $\mathcal{V}$ not occur in $\mathcal{E}$.

If the simulation cycle of a weakly combinational program quiesces then it does do with a a unique value for all the variables. Showing quiescence requires an additional condition (loop-free) which is defined later.

A weakly combinational program generates n threads:

Thread 1			**Thread** n
0:	@($\mathcal{T}_1$)		0: @($\mathcal{T}_n$)
1:	$\overline{\mathcal{V}}_1$:= $\overline{\mathcal{E}}_1$	$\cdots$	1: $\overline{\mathcal{V}}_n$:= $\overline{\mathcal{E}}_n$
2:	go 0		2: go 0

Suppose thread i has program counter pc_i. Let Inv be the n-ary conjunction $\bigwedge_{i=1}^{n} Inv_i$, where Inv_i is the statement:

$$(pc_i = 0 \;\Rightarrow\; \overline{\mathcal{V}}_i = \overline{\mathcal{E}}_i) \;\wedge\; (\overline{\mathcal{V}}_i \neq \overline{\mathcal{E}}_i \;\Rightarrow\; pc_i = 1)$$

Inv_i asserts two things: (i) if the i-th thread is at instruction 0 then the value of the vector of variables $\overline{\mathcal{V}}_i$ equals the value of the vector of expressions $\overline{\mathcal{E}}_i$ and (ii) if any of these values are not equal then the thread is at instruction 1.

Note that Inv is a predicate on states. This could me made explicit:

$$\begin{aligned} Inv(s) \;\equiv_{Def}\; \bigwedge_{i=1}^{n} \;& (\llbracket pc_i \rrbracket s = 0 \;\Rightarrow\; \llbracket \overline{\mathcal{V}}_i \rrbracket s = \llbracket \overline{\mathcal{E}}_i \rrbracket s) \\ & \wedge \\ & (\llbracket \overline{\mathcal{V}}_i \rrbracket s \neq \llbracket \overline{\mathcal{E}}_i \rrbracket s \;\Rightarrow\; \llbracket pc_i \rrbracket s = 1) \end{aligned}$$

However, where the state is clear from context the less formal implicit-state notation will be used.

Suppose Inv holds. There are two cases to consider: there are some enabled threads, or there are no enabled threads.

1. If there are some enabled threads, then one of them, say thread i, is chosen and executed. Since all updates are idempotent, the execution will cause the value of $\overline{\mathcal{V}}_i$ to become equal the value of $\overline{\mathcal{E}}_i$, hence Inv_i will hold immediately after the assignment is executed. Next, other threads

may be fired (including, possibly, thread i itself). The following case analysis established Inv continues to hold after this.

(a) If thread j is enabled but not selected for execution, then $pc_j = 1$, so Inv_j remains vacuously true.

(b) If a thread j is triggered by the execution of thread i (the case $j{=}i$ is possible), then thread j becomes enabled with $pc_j = 1$ and so Inv_j becomes vacuously true.

(c) If a thread j is listening but not triggered then it must be the case that no variable changed by executing thread i occurs in $\overline{\mathcal{E}}_j$, so the value of $\overline{\mathcal{E}}_j$ is unchanged. Also pc_j remains equal to 0.

   i. If $j = i$ then as all updates are idempotent the execution of $\overline{\mathcal{V}}_i$:= $\overline{\mathcal{E}}_i$ makes the value of each variable in $\overline{\mathcal{V}}_i$ equal to the value of the corresponding expression in $\overline{\mathcal{E}}_i$ and hence Inv_i holds.

   ii. If $j \neq i$, the variables in $\overline{\mathcal{V}}_j$ are distinct from those in $\overline{\mathcal{V}}_i$, so the value of $\overline{\mathcal{V}}_j$ is unchanged and thus Inv_j remains true.

2. If there are no enabled threads, then time will advance and if the inputs changes some threads may be fired. If no threads are fired, then the values of all variables occurring in any $\overline{\mathcal{E}}_i$ $(1 \leq i \leq n)$ will be unchanged. Since the environment only changes inputs, the value of $\overline{\mathcal{V}}_i$ is unchanged. Thus Inv continues to hold.

   If thread i is fired then $pc_i = 1$ and Inv_i becomes vacuously true.

Thus Inv is an invariant of each simulation step. If simulation quiesces, then all program counters will be 0 and so for each i the value of $\overline{\mathcal{V}}_i$ will equal the value of $\overline{\mathcal{E}}_i$.

To establish that the simulation of combinational programs always quiesces – i.e. doesn't go into an infinite loop without advancing time – it is sufficient to exhibit a 'variant' that decreases on each iteration of the simulation algorithm for which there is an enabled thread. Such a variant shows that eventually a simulation state must be reached in which there are no enabled threads, and hence time advances. A suitable variant uses the Dershowitz-Manna multiset ordering [1]. Under this ordering multiset M_2 is less than multiset M_1 if M_2 can be obtained from M_1 by removing a finite number of elements

and replacing each of them by a finite number of strictly smaller elements. This ordering is well-founded if the ordering on the elements is.

The idea of the termination proof is to show that each chain of events – one causing the next – is finite. Define $(\mathcal{V},\mathcal{E})\nearrow(\mathcal{V}',\mathcal{E}')$ to mean, roughly, that doing the update $(\mathcal{V},\mathcal{E})$ will immediately trigger a thread contain the update $(\mathcal{V}',\mathcal{E}')$.

$$(\mathcal{V},\mathcal{E})\nearrow(\mathcal{V}',\mathcal{E}') \quad \equiv_{Def} \quad \exists\, s.\ \ [\![\mathcal{E}']\!]s \neq [\![\mathcal{E}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

where $s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s]$ denotes the state optained from s by changing the value of variable $\mathcal{V}$ to the value of $\mathcal{E}$ and leaving the values of all other variables unchanged.

The relation $\nearrow$ is defined semantically, but for weakly combinational programs it implies a syntactic relation $\nrightarrow$ defined by:

$$(\mathcal{V},\mathcal{E})\nrightarrow(\mathcal{V}',\mathcal{E}') \quad \equiv_{Def} \quad \mathcal{V} \textit{ occurs in } \mathcal{E}' \textit{ and } \mathcal{V} \neq \mathcal{V}'$$

If $(\mathcal{V},\mathcal{E})\nearrow(\mathcal{V}',\mathcal{E}')$ then $(\mathcal{V},\mathcal{E})\nrightarrow(\mathcal{V}',\mathcal{E}')$. Assume that $(\mathcal{V},\mathcal{E})\nrightarrow(\mathcal{V}',\mathcal{E}')$ doesn't hold, then there are two cases:

Case 1: $\mathcal{V}$ does not occur in $\mathcal{E}'$.

In this case it is clear that for all s:

$$[\![\mathcal{E}']\!]s \ = \ [\![\mathcal{E}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

which contradicts

$$\exists\, s.\ \ [\![\mathcal{E}']\!]s \neq [\![\mathcal{E}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

hence it can't be the case that $(\mathcal{V},\mathcal{E})\nearrow(\mathcal{V}',\mathcal{E}')$.

Case 2: $\mathcal{V} = \mathcal{V}'$.

There are no clashing assignments, thus $\mathcal{E} = \mathcal{E}'$, so if $(\mathcal{V},\mathcal{E})\nearrow(\mathcal{V}',\mathcal{E}')$ then $(\mathcal{V},\mathcal{E})\nearrow(\mathcal{V},\mathcal{E})$, i.e. $\exists\, s.\ [\![\mathcal{E}]\!]s \neq [\![\mathcal{E}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$, which contradicts the requirement that $(\mathcal{V},\mathcal{E})$ be idempotent.

In both cases it follows that it is not the case that $(\mathcal{V},\mathcal{E})\nearrow(\mathcal{V}',\mathcal{E}')$.

Thus by contradiction $(\mathcal{V},\mathcal{E})\nearrow(\mathcal{V}',\mathcal{E}')$ entails $(\mathcal{V},\mathcal{E})\nrightarrow(\mathcal{V}',\mathcal{E}')$.

Define a $\nearrow$-*chain* in a set of assignments to be a sequence $(\mathcal{V}_1,\mathcal{E}_1)$, ... , $(\mathcal{V}_l,\mathcal{E}_l)$ of updates such that for $1 \leq p < l$:

(i) $\mathcal{V}_p$ is updated to $\mathcal{E}_p$ by one of the assignments in the set;

(ii) $(\mathcal{V}_p, \mathcal{E}_p) \nearrow (\mathcal{V}_{p+1}, \mathcal{E}_{p+1})$.

Call a set of assignments *loop-free* if the lengths of all $\nearrow$-chains are bounded.

Define a program to be *combinational* if it is weakly combinational and loop free.

Note that since every $\nearrow$-chain is also a $\nearrow\!\!\!\!\nearrow$-chain, if there is an infinite $\nearrow$-chain then there is also an infinite $\nearrow\!\!\!\!\nearrow$-chain. So checking that all $\nearrow\!\!\!\!\nearrow$-chains are finite ensures the program is loop-free and hence combinational.

Define the *height* of $\mathcal{V}$ to be the length of the longest $\nearrow$-chain in the assignments $\overline{\mathcal{V}}_1 := \overline{\mathcal{E}}_1, \ldots, \overline{\mathcal{V}}_n := \overline{\mathcal{E}}_n$, whose first element is $(\mathcal{V}, \mathcal{E})$, for some $\mathcal{E}$.

To show each simulation cycle quiesces, consider the sequence of steps that happen after the advance of simulation time. If no inputs change then the cycle is already quiescent and there is nothing to show. If some inputs change, then some threads will fire and become enabled with their program counters at 1.

The subsequent quiescence (termination) of the simulation algorithm can be verified by taking as variant the multiset of the heights of those variables $\mathcal{V}$ such that the value of $\mathcal{V}$ differs from the value of $\mathcal{E}$. Note that by *Inv* all such updates must be part of an enabled thread.

Suppose some threads are enabled and one of them, thread i say, is selected for execution. All the variables in $\overline{\mathcal{V}}_i$ are updated with the corresponding variables in $\overline{\mathcal{E}}_i$ by the assignment and the thread advances to instruction 0. Next, every thread j such that $\mathcal{T}_j$ contains a variable modified by executing thread i is fired and so becomes enabled.

If an update $(\mathcal{V}, \mathcal{E})$ was done by executing thread i then, since it is idempotent, the value of $\mathcal{V}$ will equal the value of $\mathcal{E}$ after the update and so the height of $\mathcal{V}$ will be removed from the multiset. This deleted height will be replaced with the heights of other variables $\mathcal{V}'$ such that there is an update $(\mathcal{V}', \mathcal{E}')$ and the value of $\mathcal{V}'$ becomes different from the value of $\mathcal{E}'$. Suppose s is the state in which $(\mathcal{V}, \mathcal{E})$ was done. The height of $\mathcal{V}'$ will be added if $[\![\mathcal{V}']\!]s = [\![\mathcal{E}']\!]s$ but:

$$[\![\mathcal{V}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s]) \neq [\![\mathcal{E}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$$

There are two cases to consider:

Case 1: If $\mathcal{V} \neq \mathcal{V}'$ this equation is:

$[\![\mathcal{V}']\!]s \neq [\![\mathcal{E}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$

i.e.

$[\![\mathcal{E}']\!]s \neq [\![\mathcal{E}']\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$

hence $(\mathcal{V}, \mathcal{E}) \nearrow (\mathcal{V}', \mathcal{E}')$.

Case 2: If $\mathcal{V} = \mathcal{V}'$ then (as each variable can only be on the left of at most one update) $\mathcal{E} = \mathcal{E}'$ and hence:

$[\![\mathcal{V}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s]) \neq [\![\mathcal{E}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$

i.e.

$[\![\mathcal{E}]\!]s \neq [\![\mathcal{E}]\!](s[\mathcal{V} \leftarrow [\![\mathcal{E}]\!]s])$

but this is impossible as $(\mathcal{V}, \mathcal{E})$ is assumed idempotent.

Thus each added height – i.e. the heights of each such $\mathcal{V}'$ – will be less than the height of $\mathcal{V}$, so the multiset decreases in the Dershowitz-Manna ordering.

Each step of the simulation algorithm decreases the multiset. As long as there are enabled threads the simulation cycle continues, so by the invariant *Inv* and the well-foundedness of the multiset ordering eventually there must be no enabled threads, i.e. the simulation quiesces.

5.1.1 Transparent latches

Consider:

```
forever @(clk or d) if clk then q := d
```

If this is converted to an equivalent assignment, the result is:

```
forever @(clk or d) q := clk ? d : q
```

The update $(\texttt{q}, \texttt{clk ? d : q})$ is idempotent, since clearly:

$\forall\, s.\ [\![\texttt{clk ? d : q}]\!]s \ = \ [\![\texttt{clk ? d : q}]\!](s[\texttt{q} \leftarrow [\![\texttt{clk ? d : q}]\!]s])$

It is not the case that:

$(\texttt{q}, \texttt{clk ? d : q}) \not\nearrow (\texttt{q}, \texttt{clk ? d : q})$

hence the latch is loop-free.

Define an update $(\mathcal{V}, \mathcal{E})$ to be *latch-like* if $\mathcal{E}$ is either $\mathcal{V}$ or a nested conditional expression in which the only occurrences of $\mathcal{V}$ are in the then or else part of conditionals. This is a static (syntactically checkable) property that obviously implies simplicity.

Define a thread `forever @(`$\mathcal{T}$`)` $\overline{\mathcal{V}}$`:=`$\overline{\mathcal{E}}$ to be latch-like if every update $(\mathcal{V}, \mathcal{E})$ in the assignment is either latch-like or has $\mathcal{V}$ not occurring in $\mathcal{E}$. Note that all such updates will be idempotent.

Consider a weakly combinational program:

`forever @(`$\mathcal{T}_1$`)` $\overline{\mathcal{V}}_1$`:=`$\overline{\mathcal{E}}_1$ $\cdots$ `forever @(`$\mathcal{T}_n$`)` $\overline{\mathcal{V}}_n$`:=`$\overline{\mathcal{E}}_n$

where each thread is latch-like. A sufficient conditional for this to be combinational is that every chain:

$(\mathcal{V}_1, \mathcal{E}_1) \nearrow\!\!\!\!\nearrow (\mathcal{V}_2, \mathcal{E}_2) \nearrow\!\!\!\!\nearrow (\mathcal{V}_3, \mathcal{E}_3) \ \ldots$

is finite.

The fact that loop-free latch-like threads are combinational shows that the use here of the term is a bit non-standard. One might expect that "combinational" would denote circuits with no memory.

5.1.2 Summary

If $\mathcal{S}_1, \ldots, \mathcal{S}_n$ contain no event controls then any program of the form:

`forever @(`$\mathcal{T}_1$`)` $\mathcal{S}_1$ $\cdots$ `forever @(`$\mathcal{T}_n$`)` $\mathcal{S}_n$

is equivalent to a program of the form:

`forever @(`$\mathcal{T}_1$`)` $\overline{\mathcal{V}}_1$`:=`$\overline{\mathcal{E}}_1$ $\cdots$ `forever @(`$\mathcal{T}_n$`)` $\overline{\mathcal{V}}_n$`:=`$\overline{\mathcal{E}}_n$

Such a program is combinational if it is weakly combinational and loop-free. A sufficient condition is that every thread is latch-like and there are no infinite $\nearrow\!\!\!\!\nearrow$-chains.

Each simulation step of a combinational program preserves the invariant:

$$\bigwedge_{i=1}^{n} \ (pc_i = 0 \ \Rightarrow \ \overline{\mathcal{V}}_i = \overline{\mathcal{E}}_i) \ \wedge \ (\overline{\mathcal{V}}_i \neq \overline{\mathcal{E}}_i \ \Rightarrow \ pc_i = 1)$$

and as long as inputs do not change, each step reduces the size of the variant and thus the simulation cycle must quiesce with all threads listening at pseudo-instruction 0 and hence with $\overline{\mathcal{V}}_i = \overline{\mathcal{E}}_i$.

Thus if s_0 is an initial state and $i_0 i_1 i_2 \ldots$ are the inputs, then the traces of this combinational program are sequences:

$$\langle i_0, s_0 \rangle \langle i_1, s_1 \rangle \langle i_2, s_2 \rangle \ \cdots$$

where the value of each variable in $\overline{\mathcal{V}}_i$ in state s_{t+1} is the value of the corresponding expression in $\overline{\mathcal{E}}_i$ evaluated with inputs i_{t+1} and state s_t.

5.2 Flip-flops

A flip-flop is a single register with input `d` and output `q` that is clocked on the positive edge of a clock `clk` (which is assumed distinct from `d` and `q`). This is modelled by:

```
forever @(posedge clk) q := d
```

It is clear that the effect of the simulation algorithm is to update the value of `q` with the value input on `d` at each simulation time that triggers `posedge clk` (i.e. at each time that `clk` is high, but was not high at the preceding time).

If $f(t)$ denotes the value of input or variable f at simulation time t, then the trace semantics of the register above is:

$$\texttt{q}(t{+}1) \ = \ \mathit{if} \ \texttt{clk}(t{+}1) = \texttt{1} \ \wedge \ \texttt{clk}(t) \neq \texttt{1} \ \mathit{then} \ \texttt{d}(t) \ \mathit{else} \ \texttt{q}(t)$$

Define:

$$(\texttt{posedge} \ \mathit{clk})(t+1) \ \equiv_{\mathit{Def}} \ \mathit{if} \ \texttt{clk}(t{+}1) = \texttt{1} \ \wedge \ \texttt{clk}(t) \neq \texttt{1}$$

then the trace semantics can be written:

$$\mathtt{q}(t{+}1) \;=\; \mathit{if}\ (\mathtt{posedge\ clk})(t+1)\ \mathit{then}\ \mathtt{d}(t)\ \mathit{else}\ \mathtt{q}(t)$$

A bank of n registers can be represented by a single parallel assignment:

```
forever @(posedge clk) q̄ := d̄
```

where $\overline{\mathtt{q}} = (\mathtt{q}_1, \ldots, \mathtt{q}_n)$ and $\overline{\mathtt{d}} = (\mathtt{q}_1, \ldots, \mathtt{d}_n)$ and $\mathtt{q}_i \neq \mathtt{q}_j$ if $i \neq j$. The trace semantics is:

$$\textstyle\bigwedge_{i=1}^{n}\ \mathtt{q}_i(t{+}1) \;=\; \mathit{if}\ (\mathtt{posedge\ clk})(t+1)\ \mathit{then}\ \mathtt{d}_i(t)\ \mathit{else}\ \mathtt{q}(t)$$

An alternative representation, with the same trace semantics, is to have n separate threads; one for each individual register.

```
forever @(posedge clk) q1 := d1
  ⋮
forever @(posedge clk) qn := dn
```

The trace equivalence of the two representations is assumed clear.

Consider now some combinational logic, say an inverter, connected to the input of a flip-flop. This could be represented with one thread by:

```
forever @(posedge clk) q := !d
```

or with the inverter and flip-flop in different threads:

```
forever @(d) a := !d
forever @(posedge clk) q := a
```

If `d` changes at the same time as a positive edge on `clk` then the second representation, but not the first, might store the negation of the unchanged (i.e. previous) value of `d`. This would happen if the flip-flop thread was executed first. The subsequent execution of the inverter thread will set `a` to the negation of the new value of `d`, but since the flip-flop is not sensitive to changes in `a` it won't update the value of `q`.

Thus the single thread representation is more robust and generally to be preferred.

The trace semantics of the single thread version is:

$$\texttt{q}(t{+}1) \;=\; \mathit{if}\ (\texttt{posedge clk})(t+1)\ \mathit{then}\ \texttt{!d}(t)\ \mathit{else}\ \texttt{q}(t)$$

and of the two threads is:

$$\begin{array}{l}(\forall\, t.\ \texttt{a}(t) \;=\; \texttt{!d}(t)) \\ \wedge \\ (\forall\, t.\ \texttt{q}(t{+}1) \;=\; \mathit{if}\ (\texttt{posedge clk})(t+1)\ \mathit{then}\ \texttt{a}(t)\ \mathit{else}\ \texttt{q}(t))\end{array}$$

If it is assumed that inputs never change on clock edges[2] then the two representations are produce the same signal trace on `q`.

If the combinational logic is connected to the output of the flip-flop the problem doesn't arise. Thus no matter what order the threads in:

```
forever @(posedge clk) a := d
forever @(a) q := !a
```

are executed, the final quiescent state will be the same – namely `q` will equal the negation of the value of `d`.

$$\begin{array}{l}(\forall\, t.\ \texttt{a}(t{+}1) \;=\; \mathit{if}\ (\texttt{posedge clk})(t+1)\ \mathit{then}\ \texttt{d}(t)\ \mathit{else}\ \texttt{q}(t)) \\ \wedge \\ (\forall\, t.\ \texttt{q}(t) \;=\; \texttt{!a}(t))\end{array}$$

However, with this example there is a hazard: if the inverter executes first then `q` will be set to the negation of the old value of `a`. The subsequent execution of the flip-flop will update `a` and might then change the value of `q`, since the inverter is sensitive to changes in `a`. The hazard is only visible at the event semantics level; it is invisiable with trace semantics.

5.3 RTL programs

Register Transfer Level (RTL) hardware consists of combinational logic and flip-flops.

[2]This assumption on the environment can be hard or impossible to enforce if the inputs are asynchronous – however clever analogue engineering can make it true with high probablility.

An HDL representation of this is a program that can be partitioned into a set of threads that are a combinational program plus a set of flip-flops of the form:

```
forever @(posedge clk) V̄ := Ē
```

It is assumed that there is a single input clock `clk` and that the environment is such that no other inputs change when `clk` has a positive edge. Also it is assumed no two distinct flip-flops drive the same ourput (no clashing assignments).

Thus an RTL program has the form:

```
forever @(T1) V̄1 := Ē1
  ⋮
forever @(Tm) V̄m := Ēm

forever @(posedge clk) V̄m+1 := Ēm+1
  ⋮
forever @(posedge clk) V̄m+n := Ēm+n
```

where the first m threads constitute a combinational program and the last n threads are flip-flops.

A simulation cycle that starts with a positive edge on `clk` will (by assumption) be one for which none of the other inputs change. Thus the positive edge will enable all the flip-flops and none of the other threads. After m simulation steps (i.e. one to execute each flip-flop) the output variables will have been updated and the cycle will quiesce.

A simulation cycle that that doesn't start with a positive edge on `clk` will not enable any of the flip-flops. The rest of the threads will simulate as combinational logic and, by the analysis in 5.1, will quiesce with each variable set to the value of the expression that updates it.

Thus the trace semantics of the RTL program above is:

$$
\begin{array}{l}
(\forall\, t.\ \overline{\mathcal{V}}_1(t) \ = \ \overline{\mathcal{E}}_1(t)) \\
\wedge \\
\vdots \\
\wedge \\
(\forall\, t.\ \overline{\mathcal{V}}_m(t) \ = \ \overline{\mathcal{E}}_m(t)) \\
\wedge \\
(\forall\, t.\ \overline{\mathcal{V}}_{m+1}(t{+}1) \ = \ \mathit{if}\ (\texttt{posedge clk})(t+1)\ \mathit{then}\ \overline{\mathcal{E}}_{m+1}(t)\ \mathit{else}\ \overline{\mathcal{V}}_{m+1}(t)) \\
\wedge \\
\vdots \\
\wedge \\
(\forall\, t.\ \overline{\mathcal{V}}_{m+n}(t{+}1) \ = \ \mathit{if}\ (\texttt{posedge clk})(t+1)\ \mathit{then}\ \overline{\mathcal{E}}_{m+n}(t)\ \mathit{else}\ \overline{\mathcal{V}}_{m+n}(t))
\end{array}
$$

To be continued ...

Acknowledgements

This work is funded by the U.K. Engineering and Physical Sciences Research Council (EPSRC) as project GR/K57343. The principal Investigator is Dr. David Greaves.

We are grateful to Synopsys, Inc. for providing us with their software and for ongoing discussions on the semantics of synthesizable Verilog.

References

[1] Nachum Dershowitz and Zohar Manna. Proving termination with multiset orderings. *Communications of the ACM (CACM)*, 22(8):465–476, August 1979.

[2] D. Knapp, T. Ly, D. MacMillen, and R Miller. Behavioural synthesis methodology for HDL-based specification and validation. In *Proc. of 32nd DAC*, San Francisco, June 1995. `http://www.synopsys.com/pubs/research/ATG_index.html`.