Synchronous Verilog

Ching-Tsun Chou
⟨ctchou@fla.fujitsu.com⟩

Fujitsu Laboratories of America
3350 Scott Blvd., Bldg. 34
Santa Clara, CA 95054, U.S.A.

28 January 1997

Warnings!

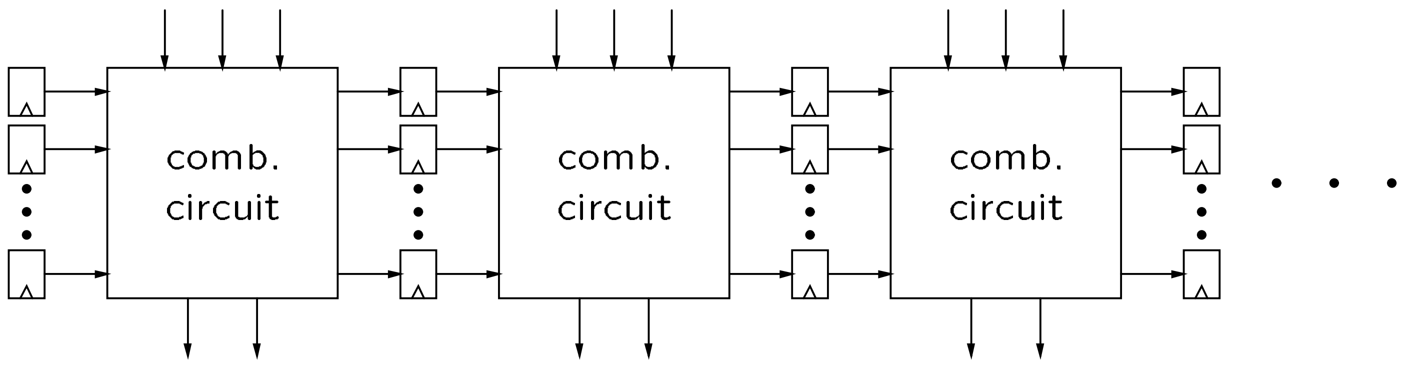
- This talk is *not* about finished work, but about ongoing research.
- In fact, some of my views have changed since I wrote the abstract for this talk last week.
- There will be a lot of handwaving in this talk; in particular, there will be no proof.
- But I will try to formulate, as precisely as I can, what seems to me to be worth proving.
- Your comments will be greatly appreciated!

Synchronous Abstraction

- A *synchronous system* engages in a series of *instantaneous* reactions with its environment, each of which observes inputs, produces outputs, and changes the system state in *zero time*.

$$S_0 \xrightarrow[O_1]{I_1} S_1 \xrightarrow[O_2]{I_2} S_2 \xrightarrow[O_3]{I_3} S_3 \quad \cdots$$

- More concretely, consider a synchronous circuit:



- In reality, reactions of course take time, but the *synchronous abstraction* can be used as long as there is no combinational loop and the clock period is long enough.

- The main advantage of the synchronous abstraction is the separation of *functional* and *timing* considerations.

- Most digital hardware designs are synchronous.
  - For simplicity, we do not consider multi-clock or multi-phase synchronous designs in this talk.

Synchronous Languages

- Their underlying model of computation is based on the synchronous abstraction. Examples:
  - Procedural: *Esterel*, *YASL*[*].
  - Dataflow: *Lustre*, *Signal*.
  - Graphical: *StateChart* (?), *Argos*.
- They are ideal for describing the *functional* aspect of synchronous hardware, because:
  - their semantics are at the right level of abstraction;
  - their constructs express powerful notions of control flow;
    - Examples: concurrency, sequencing, iteration, preemption, suspension, . . .
  - their programs can be naturally translated into synchronous circuits, which can be optimized to produce efficient hardware in many cases.
    - See G. Berry's draft book [1996] or my paper [CHDL, 1997][†] for circuit translation.
    - See E.M. Sentovich, H. Toma, and G. Berry [ICCAD, 1996] for circuit optimization.

[*]Yet Another Synchronous Language

[†]Poster presentation

Verilog and VHDL

- They are the two most widely used HDLs (hardware description languages).

- Their underlying model of computation is *discrete-event simulation*, which differs from the synchronous model in two (related) aspects:
  - *Nonatomicity:* A signal can (be observed to) assume more than one value in one clock cycle;
  - *Nondeterminism:* Given the same input history, a program may allow more than one execution.

- The discrete-event simulation model is at too low a level of abstraction for describing the *functional* aspect of synchronous hardware. Manifestations:
  - Problems with “race conditions”, which often need be fixed by kludges like zero-delays.
  - Pervasiveness of “Moore-type” parallel composition with latched outputs and one-cycle delay in communication.
  - Growing popularity of “cycle-based simulation” in functional verification.

How to Introduce Synchronous HDLs into "The Real World"?

- Use a new language (eg, Esterel, YASL, . . .).
  - Small and fragmented user communities.
  - Lack of tools and experience.
  - Existing investments in Verilog or VHDL.
- Use a subset of Verilog or VHDL, but with a synchronous semantics.
  - People are already doing it!
    * Cycle-based simulation.
    * Synthesizable subsets.
    * Front-end of model checkers (eg, `vl2mv`).
  - But, what exactly does "Synchronous Verilog" or "Synchronous VHDL" mean?
    * The subject explored in this talk.

Synchronous Verilog

Synchronous Verilog, or *SV*, is a subset of Verilog with the following properties:

- One can read SV programs as if one were reading Esterel programs, with two main exceptions:
  - SV *variables* (by default) correspond to hardware registers, while Esterel *signals* correspond to combinational circuits;
  - Esterel has built-in constructs for *suspension*, while SV, like Verilog, does not.
- There is a translation $\mathcal{C}$, from SV programs to (ideal) synchronous circuits, which serves as the *synchronous semantics* of SV.
- There is another translation $\mathcal{V}$, from SV programs to Verilog programs, which serves as the *simulation semantics* of SV.
- For any SV program p,
  - every execution of $\mathcal{C}(p)$ is the abstraction of an execution of $\mathcal{V}(p)$, and
  - every execution of $\mathcal{V}(p)$, after abstraction, is an execution of $\mathcal{C}(p)$,

  where the abstraction ignores the "nonessential" order between the steps of $\mathcal{V}(p)$ within a step (ie, a clock cycle) of $\mathcal{C}(p)$.

Outline of the Rest of This Talk

- List the design decisions for SV and discuss their consequences.
- Define the translation $\mathcal{C}$ from SV programs to synchronous circuits, via a kernel language called *SVK* (*SV Kernel*).
- Outline the translation $\mathcal{V}$ from SV programs to Verilog programs and argue why the relationship between $\mathcal{C}$ and $\mathcal{V}$ specified on the last slide holds.
- Discussion and future work.

Design Decisions for SV

D0 No module, function, or task is used.

- Tantamount to assuming that all modules, functions, and tasks used are textually expandable and expanded.
- Call-by-value parameter passing in tasks is banned (for now).
- But shared variables between tasks are allowed.

D1 There is a single clock `clk` and the statement `@(posedge clk);` is the only timing control allowed.

- No delay (`#`), level-sensitive control (`wait`), intra-assignment control, or named event control is allowed.
- "Temporal abstraction" is not treated (yet).

Design Decisions for SV (continued)

D2 For every variable `X` and in every clock cycle:

- all reads on `X` return the same value;
- there is at most one blocking assignment to `X` and at most one non-blocking assignment to `X`.

There are 3 values associated with `X` in any cycle:

- `X.now` is the value returned by any read on `X` in this cycle;
- `X.next` is the value stored into `X` at the end of this cycle;
- `X.prev` is the value stored into `X` at the end of the previous cycle.

In any cycle, these values and the assignments are related as follows:

- if no `X = ···` and no `X <= ···`
  then `X.now` = `X.next` = `X.prev`
- if `X = V1` but no `X <= ···`
  then `X.now` = `X.next` = `V1`
- if `X <= V2` but no `X = ···`
  then `X.now` = `X.prev` and `X.next` = `V2`
- if `X = V1` and `X <= V2`
  then `X.now` = `V1` and `X.next` = `V2`

Consequences of D2

- A read on `X` always returns the "most up-to-date value" of `X`. For example, in SV,

```
X = 13;        ||        if (X == 13)
                           Y = 1;
                         else
                           Y = 2;
```

  always executes `Y = 1;`

  - Note that there is no one-cycle delay in communication, so SV supports "Mealy-type" parallel composition.
  - SV also supports "Moore-type" parallel composition via non-blocking assignments.
  - `vl2mv` in `VIS` adopts the same rule for blocking assignments, but disallows non-blocking assignments altogether.
  - IEEE Verilog Standard allows a *race condition* in this example, which can be "fixed" by using zero-delays to force a Verilog simulator to produce the desired execution:

```
X = 13;        ||        #0;
                         if (X == 13)
                           Y = 1;
                         else
                           Y = 2;
```

  - This is exactly the program generated by $\mathcal{V}$.

Consequences of D2 (continued)

- The following program is *not* in SV:

```
X = Y;        ||        Y = X;
```

  because of a combinational loop.

  - If the intention is to exchange the values of `X` and `Y` for the next cycle, then write:

    ```
    X <= Y;        ||        Y <= X;
    ```

    which is in SV.

- The following program is also *not* in SV:

```
X = X + 1;
```

  again due to a combinational loop.

  - If the intention is to increment `X` and to have the new value available in this cycle, then introduce a temporary variable `X1` and write:

    ```
    X1 = X + 1;
    X <= X1;
    ```

  - In general, in SV, a blocking assignment behaves just like a continuous assignment whose effect is restricted to one cycle.

Consequences of D2 (continued)

- Is the following program in SV?

```
X = Y;
Y = 3;
```

  Yes, but it is equivalent in SV to:

```
Y = 3;
X = Y;
```

  because blocking assignments are like single-cycle continuous assignments and so their order does not matter within one cycle.

  - The second program is what $\mathcal{V}$ generates from the first program.

- A similar example:

```
X = W;      ||      Z = Y;
Y = 3;              W = 4;
```

  from which $\mathcal{V}$ generates:

```
Y = 3;      ||      W = 4;
#0;                 #0;
X = W;              Z = Y;
```

  which forces the desired execution.

Design Decisions for SV (continued)

D3 The semantics of continuous assignments in SV is defined by the following reduction:

```
assign X = exp;    ≡    always begin
                          X = exp;
                          @(posedge clk);
                        end
```

- This reduction allows us to assume that, in SV, every wire is driven by a variable of the same name.
- The register introduced by $\mathcal{C}$ for the variable `X` on the RHS can be shown to be redundant, so a continuous assignment can still be implemented by a combinational circuit.

Design Decisions for SV (continued)

D4 In the cycle in which it is `disable`d, a `disable`d block should be allowed to progress as far as possible in all its threads.

For example:

```
X = 0; Y = 0; Z = 0; W = 0;
fork : B
  begin X = Z; disable B; Y = 2; end
  begin Z = 3; W = Y + 4; end
join
```

should leave `X == 3`, `Y == 0`, `Z == 3`, and `W == 4` at the end.

- The desired execution can be forced by inserting `#0` just before `X = Z;`
- **D4** is influenced by the semantics of trap-exit in Esterel.

Design Decisions for SV (continued)

D5 Let `B1` and `B2` be (the names of) two named blocks, with `B1` containing `B2`. If `disable B1` and `disable B2` are executed in the same cycle, then the former takes precedence over the latter.

For example:

```
X = 0; Y = 0;
begin : B1
  fork : B2
    disable B1;
    disable B2;
  join
  Y = 2;
end
X = 1;
```

should leave `X == 1` and `Y == 0` at the end.

- The desired execution can be forced by replacing `disable B2;` with `#0 disable B2;`
- **D5** is also influenced by the semantics of trap-exit in Esterel.
- This example is adapted from one in Esterel due to E.M. Sentovich.

SV Kernel

- Variables

$$X, Y, Z, W, \ldots$$

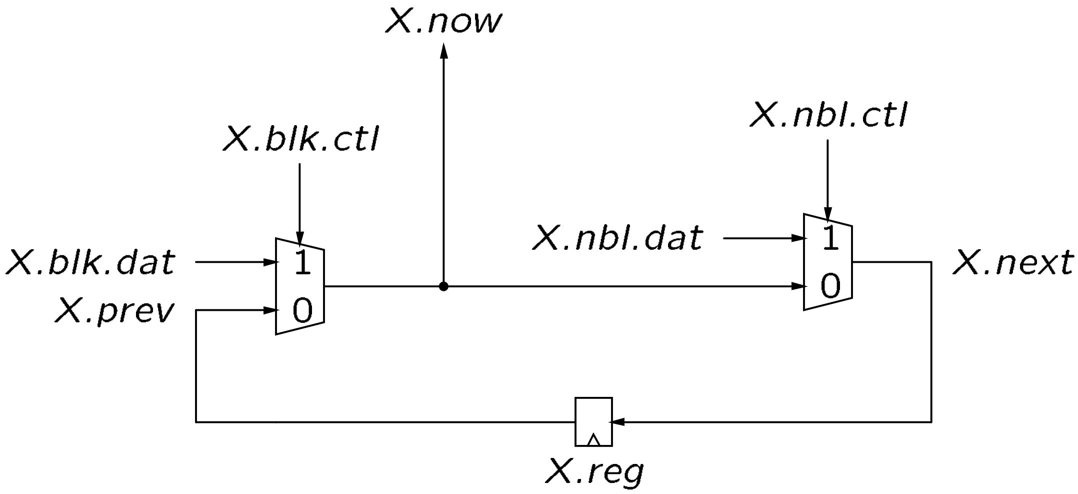
- Expressions

$$e \quad ::= \quad X \mid C \mid U\ e \mid e\ B\ f \mid e\ \mathbf{?}\ f\ \mathbf{:}\ g$$

- Statements

$$\begin{array}{rcll}
p & ::= & X = e & \text{(blocking assign.)} \\
 & \mid & X \Leftarrow e & \text{(non-blocking assign.)} \\
 & \mid & \textbf{skip} & \text{(do nothing)} \\
 & \mid & \textbf{pause} & \text{(pause a cyle)} \\
 & \mid & \textbf{exit}\ p\ \textbf{when}\ e & \text{(preemption)} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ p\ \textbf{else}\ q & \text{(conditional)} \\
 & \mid & p \parallel q & \text{(concurrency)} \\
 & \mid & p \circ q & \text{(sequencing)} \\
 & \mid & \textbf{forever}\ p & \text{(iteration)}
\end{array}$$

Implementing Variables in Circuits

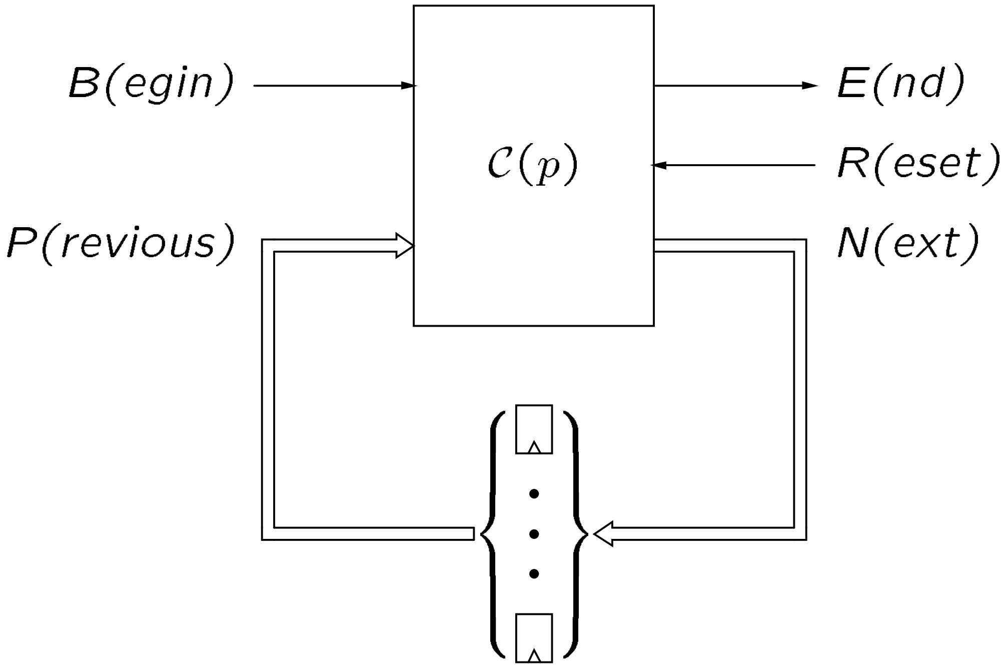


- $X.blk.ctl$ = the OR of all $X.blk.ctl.i$, where i ranges over all blocking assignments to X.

- $X.blk.dat$ = the $X.blk.dat.i$ if there is a unique i such that $X.blk.ctl.i = 1$; otherwise, $X.blk.dat$ is undefined.

- Similar remarks apply to $X.nbl.ctl$, $X.nbl.dat$, and non-blocking assignments to X.

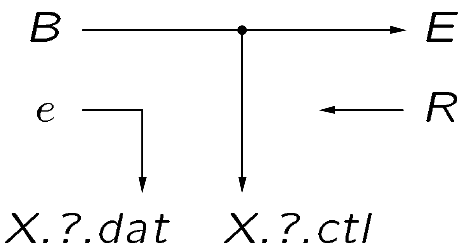
Implementing Statements in Circuits



- The number of control latches is exactly the number of **pause**'s in p.
- In the first cycle of p, $B = 1$ and $P = \vec{0}$.
- In all subsequent cycles of p, $B = 0$ and $P \neq \vec{0}$.
- p signifies (non)termination by setting $E = 1$ and $N = \vec{0}$ (resp. $E = 0$ and $N \neq \vec{0}$).
  - *Note:* p can terminate in the same cycle in which it is started.
- If $R = 1$, then $N = \vec{0}$, which implies that $\mathcal{C}(p)$ will be idle from the next cycle onward (unless it is re-started by $B = 1$). The behavior of $\mathcal{C}(p)$ is not otherwise affected by $R = 1$.

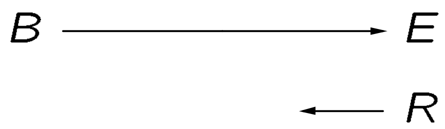
Implementing Statements in Circuits (continued)

- $\mathcal{C}(X = e)$ and $\mathcal{C}(X \Leftarrow e)$

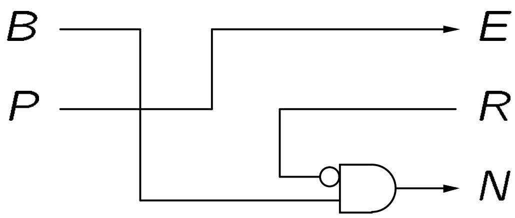


where *?* is *blk* or *nbl*.

- $\mathcal{C}(\textbf{skip})$

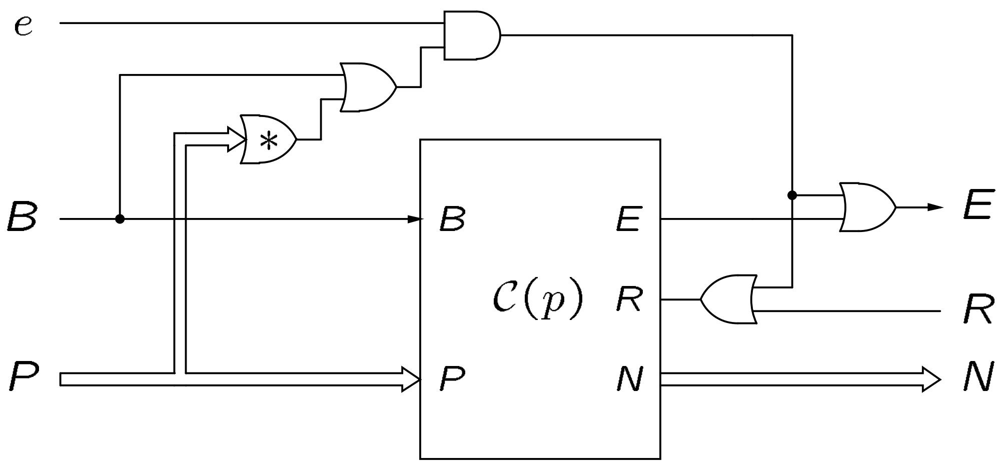


- $\mathcal{C}(\textbf{pause})$



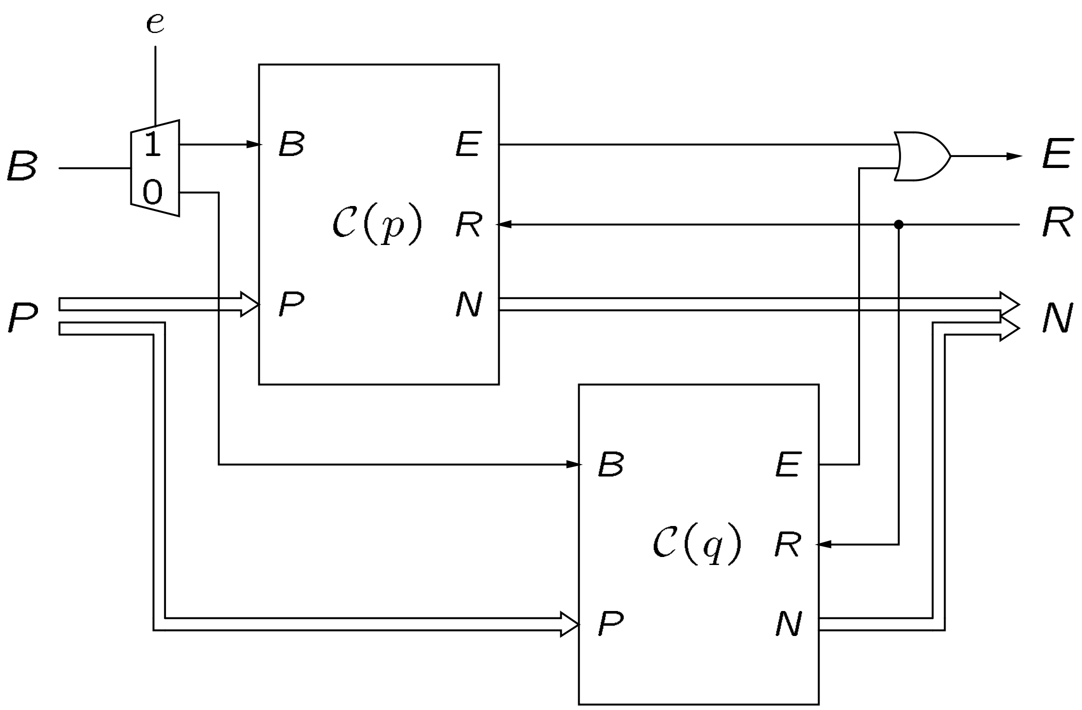
Implementing Statements in Circuits (continued)

- $\mathcal{C}(\textbf{exit}\ p\ \textbf{when}\ e)$



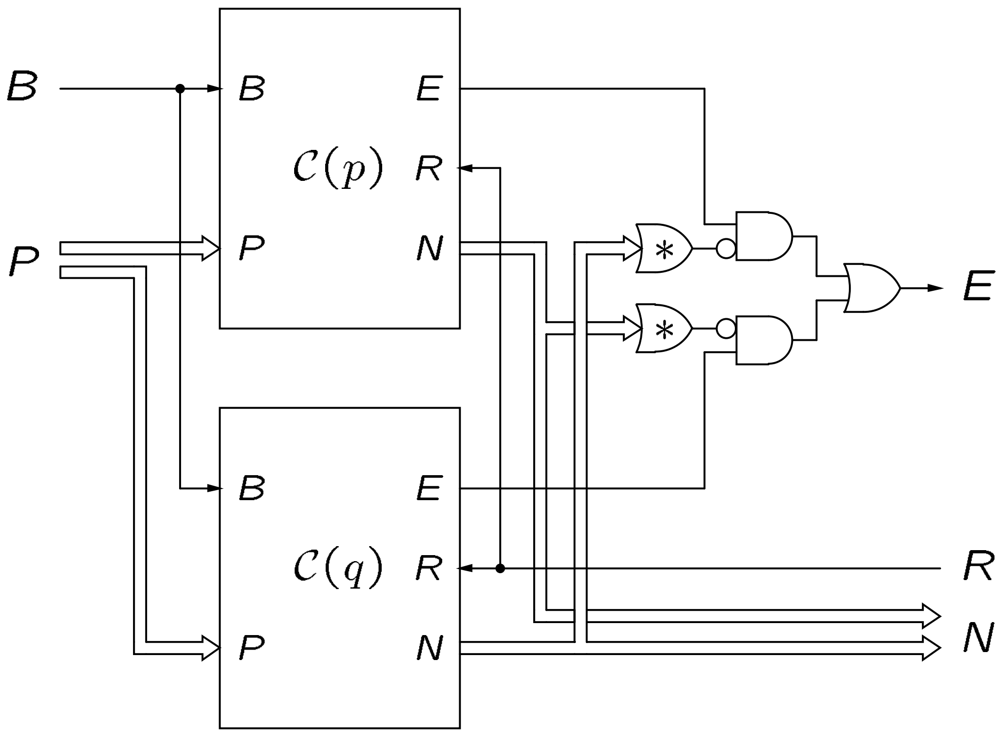
Implementing Statements in Circuits (continued)

- $\mathcal{C}(\textbf{if } e \textbf{ then } p \textbf{ else } q)$



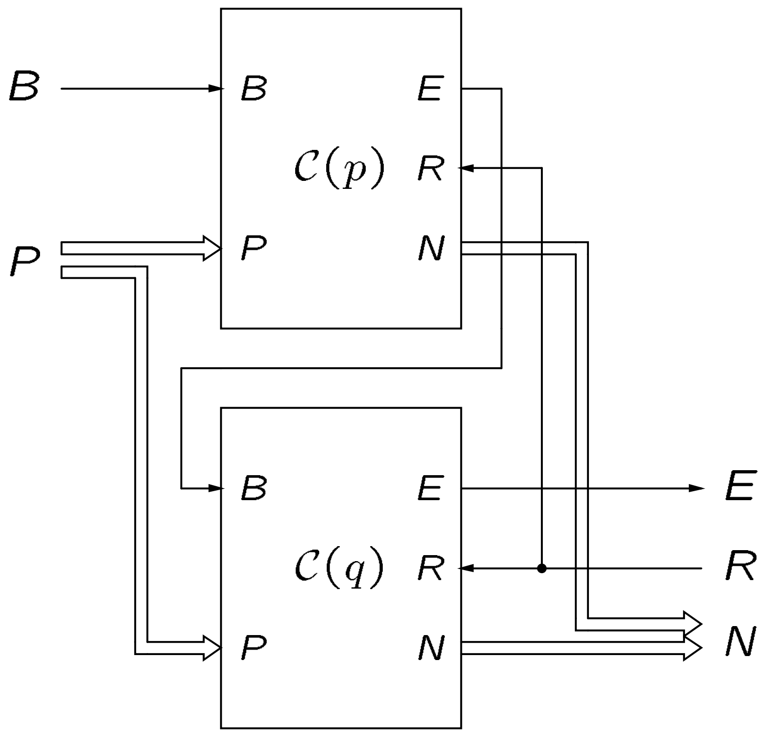
Implementing Statements in Circuits (continued)

- $\mathcal{C}(p \parallel q)$



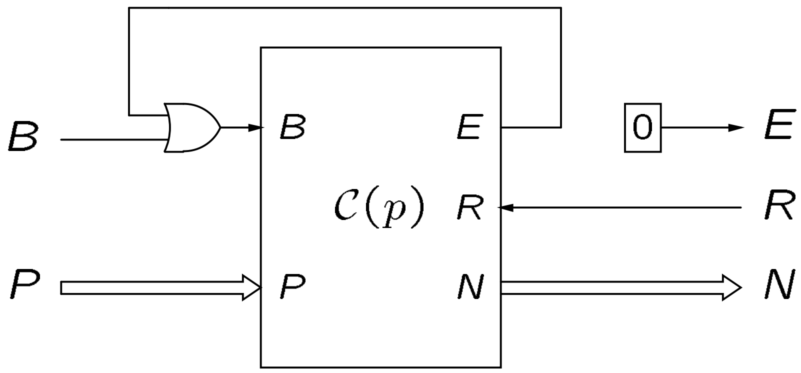
Implementing Statements in Circuits (continued)

- $\mathcal{C}(p \circ q)$



Implementing Statements in Circuits (continued)

- $\mathcal{C}($**forever** $p)$

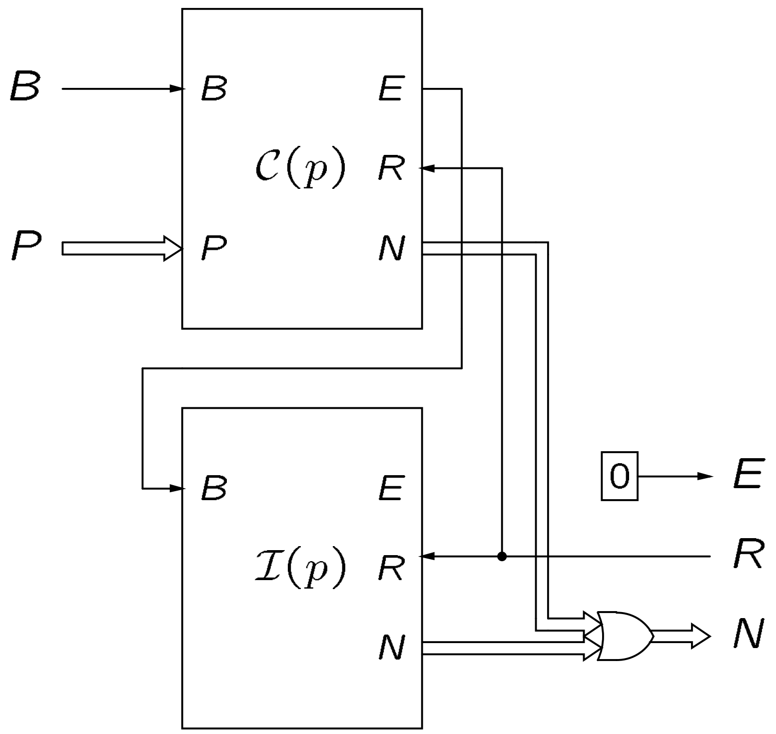


- There must not be a combinational path from *B* to *E* in $\mathcal{C}(p)$, i.e., p must not be able to terminate in the same cycle in which it is started.

- This translation may suffer from the so-called *schizophrenia*—the conflict between the last cycle of one iteration of p and the first cycle of the next iteration of p in the same circuit.

Implementing Statements in Circuits (continued)

- $\mathcal{C}$(**forever** p)



- This translation does not suffer from schizophrenia due to the duplication of $\mathcal{I}(p)$, which is the part of $\mathcal{C}(p)$ that can be active in the first cycle of p.

- But the circuit duplication may lead to circuit size being quadratic, instead of linear, in the program size (though note that the number of control latches is not affected).

- This translation should be used when there is any wire in $\mathcal{C}(p)$ that is reachable from both *B* and *E* through only combinational gates.

Reducing SV to SVK

- `@(posedge clk);` $\equiv$ **pause**

- Module composition of `initial` p_1, ..., `initial` p_n $\equiv$ `initial fork` $p_1 \cdots p_n$ `join`

- Continuous assignments are reduced to `always` loops as specified earlier.

  `always` $\equiv$ `initial` **forever**

- `begin` $p_1 \cdots p_n$ `end` $\equiv$ $p_1 \circ \cdots \circ p_n$ if $n > 0$

  `begin end` $\equiv$ **skip**

- `fork` $p_1 \cdots p_n$ `join` $\equiv$ $p_1 \parallel \cdots \parallel p_n$ if $n > 0$

  `fork join` $\equiv$ **skip**

- `if (`e`)` p `else` q $\equiv$ **if** e **then** p **else** q

  `if (`e`)` p $\equiv$ **if** e **then** p **else skip**

  `case` statements are reduced to `if` statements in the obvious way.

- `while (`e`)` p $\equiv$
  `begin:T` **forever** $\{$**if** e **then** `disable T` **else** $p\}$ `end`
  - Direct translation would be more efficient.

Reducing SV to SVK (continued)

- Assume WLOG that no block name is declared twice or used as a variable name, so that for each block name `T`, a fresh variable `T` can be created. `T` should have initial value 0 and allow multiple assignments to it in one cycle, which are OR'ed.

  $\texttt{begin:T} \cdots \texttt{end} \;\equiv$
  $\quad$ **exit** $\texttt{begin} \cdots \texttt{end}$ **when** $(\texttt{T}) \;\circ$
  $\quad$ **if** $(\texttt{T}_1 \;||\; \cdots \;||\; \texttt{T}_n)$ **then loop else skip**

  $\texttt{disable T} \;\equiv$
  $\quad \{\ \texttt{T} = 1 \;\circ\; \texttt{T} \Leftarrow 0\ \}$ if `disable T` is outside block `T`
  $\quad \{\ \texttt{T} = 1 \;\circ\; \texttt{T} \Leftarrow 0 \;\circ\;$ **loop** $\}$ otherwise

  where **loop** $\equiv$ **forever pause** is the infinite loop and $\texttt{T}_1, \ldots, \texttt{T}_n$ are the names of named blocks containing block `T`.

  - A similar construction works for named `fork-join` blocks.
  - Note that if $n = 0$, then $(\texttt{T}_1 \;||\; \cdots \;||\; \texttt{T}_n)$ is equivalent to false.
  - No register need in fact be allocated for `T`, because 1 is never stored into the register.
  - No register need be allocated for the **loop**'s either, since the register is immediately reset.
  - Actually `T` is better modeled as an Esterel *signal* instead of as a variable.

What Exactly Is $\mathcal{V}$?

- For any SV program p, if $\mathcal{C}(p)$ contains no combinational loop, then there is a partial order for evaluating the combinational gates of $\mathcal{C}(p)$. $\mathcal{V}$ tries to realize that partial order.

- The real question is:

  > Is it always possible to "massage" the program text of p using such "local" transformations as inserting zero-delays and exchanging adjacent assignments, so that that partial order is realized?

  - I think the answer is yes, but I have no proof.

- If the answer is yes, then the relation between $\mathcal{C}$ and $\mathcal{V}$ specified earlier is clearly satisfied.

What about VHDL?

- I am not very familiar with VHDL.

- The control constructs of VHDL are less flexible than those of Verilog.
  - Flat model of concurrency: No `fork-join` nested within `begin-end`.
  - No general preemption construct `disable`: Only `next` and `exit`.

- The simulation semantics of VHDL is less permissive than that of Verilog.
  - In VHDL, the execution of a stretch of sequential code without `wait` can *not* be interleaved with the executions of any other such stretches; in Verilog, it can.
    - Thanks to W.C. Baker for pointing this out.
  - So the "inserting zero-delays" trick is harder to use. (Are there "zero-delays" in VHDL?)

- W.C. Baker investigated "Synchronous VHDL" in his PhD Thesis [1996].
  - As I understand it, Synchronous VHDL is the subset of VHDL for which the synchronous semantics and the simulation semantics agree.
  - In contrast, Synchronous Verilog is really a new language different from, but I hope well-related to, both Esterel and Verilog.

Future Work

- Find out whether, from the perspective of "The Real World", the kind of problems I have addressed in this talk is interesting and relevant.
- Find out whether, again from the perspective of "The Real World", the particular approach I have taken is reasonable and promising.
- Technical problems to be solved:
  - How to deal with multiple assignments in a single cycle?
    - Prevent it by "coding style".
    - Detect it at compile-time.
    - Detect it at run-time.
    - Use resolution functions.
  - How to deal with "benign" combinational loops?
    - Are they common in practice?
    - Is the approach taken by Esterel ("causality analysis" using BDD) reasonable?
    - What should $\mathcal{V}$ do if there are "benign" combinational loops?
  - What exactly is $\mathcal{V}$?

Acknowledgments

I would like to thank:

- Mike Gordon for writing "The Semantic Challenge of Verilog HDL" [LICS, 1995];
- Daryl Stewart for pointing out the existence of `disable` in Verilog;
- Wendell Baker for explaining Synchronous VHDL to me;
- Elizabeth Wolf for pointing out the existence of Wendell Baker's work;
- Szu-Tsung Cheng for answering my questions about `vl2mv`;
- Mike, Daryl, Wendell, and Ellen Sentovich for commenting on my ramblings about SV.