Intensive one week course in

Hardware Description Languages for Synchronous Hardware Design

Verilog II

Gordon J. Pace

September 2000

How do these Verilog programs behave?

- ```
  initial v=1;
  initial v=0;
  ```
- ```
  initial @(posedge v) v=0;
  initial begin v=0; v=1; end
  ```


Verilog: A concurrent programming language

- The simulator *does not* interpret your code as hardware and try to emulate the circuit.
- Verilog is just a concurrent language whose semantics are defined in terms of how the simulator works.

The simulation cycle

1. Transform program into one flat module.
2. Create one parallel thread per module item.
3. Initialise variables.
4. Run along threads which are enabled.
5. Enable threads delayed to the current time (if any go to 4).
6. Choose the smallest time such that there is a thread delayed to that time (If there is no such time go to 7). Advance time to the new time and enable all threads delayed to this time. Go to 4.
7. Terminate simulation.

Step 1: Flattening the modules

```
module delay(out,clk,in);
  always
    @(posedge clk)
    { mem, out } = { in, mem };
endmodule
module clock(clk);
  initial
    clk=0;
    forever #5 clk=~clk;
endmodule
module main;
  delay D1(middle,clk,in    );
  delay D2(out,   clk,middle);
  clock C(clk);
endmodule
```

```
module main;
 initial
   while (1)
     @(posedge clk)
     { D1.mem, middle } = { in, D1.mem };

 initial
   while (1)
     @(posedge clk
     { D2.mem, out } = { middle, D2.mem };

  initial
    clk=0;
    forever #5 clk=~clk;
```


The simulation state

- The values of the variables
- The current time *now*
- Position in the threads
- Threads can be:
  - `enabled`
  - `delayed to time` t
  - `guarded by @g` (or `wait e`)
  - `finished`

Simulation examples (1)

```
initial
   v = 0;
   v = 1;

initial
   @(posedge v)
   w = 1;
```

Variable values:
`v =`
`w =`

States of threads:
`Thread 1:`
`Thread 2:`

Simulation examples (2)

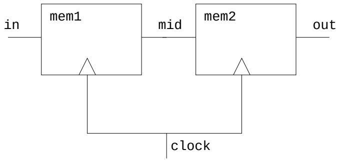
```
initial
   #0
   v = 0;
   v = 1;

initial
   @(posedge v)
   w = 1;
```

Variable values:
`v =`
`w =`

States of threads:
`Thread 1:`
`Thread 2:`

Revisiting the delay

Consider two delays in sequence:




```
always
  @(posedge clock)
  {mem1,mid} = {in,mem1};
always
  @(posedge clock)
  {mem2,out} = {mid,mem2};
```

This is non-deterministic (and incorrect)!

Non-blocking assignments

```
v <= e; P
```

- Evaluate `e` and enable `P`, but
- perform the assignment only when no enabled processes (or processes delayed to the current time) remain.

The intuition: Perform all calculations with the current values, and only perform the update at the ‘end’ of the the current simulation time.

Non-blocking assignments: Examples

- Swap values:

```
initial
  v <= w; w <= v
```

- Interaction with other processes:

```
initial
  v  = 0;
  v <= 1;

initial
  @(posedge v) ...
```


Non-blocking assignments: Examples (1)

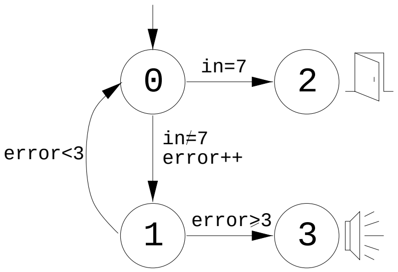
- The delay revisited once again:

```
always
  @(posedge clock)
  mid <= in;

always
  @(posedge clock)
  out <= mid;
```

The updated simulation cycle

4. Run along threads which are enabled.
5. Enable threads delayed to the current time (if any go to 4).
6. Choose the smallest time such that there is a thread delayed to that time (If there is no such time go to 7). Advance time to the new time and enable all threads delayed to this time. Go to 4.
7. Perform *one* non-blocking assignment from the assignment queue and go to 4. If none, go to 8.
8. Terminate simulation.

State machine encoding




```
initial begin
  open  = 0;
  siren = 0;
  error = 0;
  state = 0;
end;
```

```
always
  @(posedge clock)
  case (state)
    0: if (input==7)
          state <= 2;
       else begin
          state <= 1;
          error <= error+1;
       end
    1: state <= (error >= 3)?3:0;
    2: open <= 1;
    3: siren <= 1;
  endcase;
```


Synthesis: The idea

- Writing behavioural descriptions is easy.
- Writing gate-level descriptions is tedious and time consuming.
- Why don't we *compile* our high level descriptions into gate-level implementations?

In hardware design circles, this is called *synthesis.*

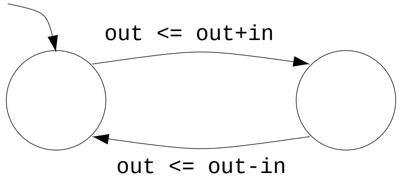
Synthesis: Restrictions

- The process may have only one global clock (and possibly a reset wire).
- Only one process may write to a signal.
- Variables (assigned by `=`) may not be read by parallel processes, but signals (assigned by `<=`) may.
- Timing can only be performed via edge detection of the `clock` and `reset` signals.
- Loops must have a `@(posedge clock)` through every path they may take.

Synthesis: Examples (1)

An add/subtract register

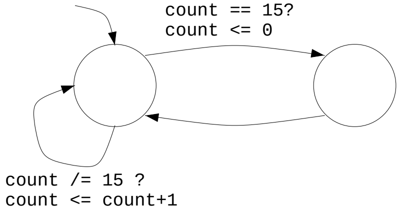
```
always
  @(posedge clock) out <= out+in;
  @(posedge clock) out <= out-in;
```



Synthesis: Examples (2)

A counter

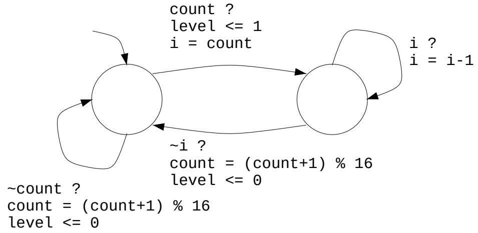
```
always
  @(posedge clock)
  if (count==15)
    count <= 0; @(posedge clock);
  else
    count <= count+1;
```



Synthesis: Examples (3)

Another counter

```
always
  @(posedge clock)
  level <= 1; i = count;
  while (i)
    @(posedge clock) i=i-1;
  count = (count + 1) % 16;
  level <= 0;
```



Conclusions

- The simulation semantics of Verilog are *complicated!*
- Separating variables from signals is a good approach, but may still cause intractable code.
- Synthesisable code is still not as easy as behavioural code to write, but is a step towards fast development of large circuits.