Cadence: IXCOM Internals

Contents

1 SystemVerilog Elaboration 1

2 SystemVerilog Optimization and Transformation 1

2.1 Assignment operator transformation . . . 1
2.2 Dead code elimination . . . 2
2.3 Loop unrolling . . . 2
2.4 Constant propagation . . . 2
2.5 Tristate buffer transformation . . . 3
2.6 DISABLE transformation . . . 3
2.7 Continue/Break statement . . . 4
2.8 Multiple event transformation . . . 4
2.9 RTL blocking delay transformation . . . 6
2.10 Assignment Operator . . . 12

3 Primitives 12

1 SystemVerilog Elaboration

2 SystemVerilog Optimization and Transformation

2.1 Assignment operator transformation

- when there is a statement that contains an assignment operator expression, we hoist the expression to statements; for example, given

  ```
  a = c++ + b;
  ```

  we create the following statements:

  ```
  t0 = c;     // save the original value of c to t0
  c = c + 1;  // increment c;
  a = t0 + b; // use the saved value to compute a
  ```

- **complication #1: dependency issue**: sometimes we need to hoist non-assignment-operator expressions; given a task call

  ```
  foo(j, i = j++, ...);
  ```

  we should be careful when there are any dependencies between task arguments; therefore,

  ```
  // wrong transformation       // correct transformation
  t0 = j;                       t1 = j;
  j = j + 1;                    t0 = j;
  foo(j/**/, i = t0, ...);      foo(t1, i = t0, ...);
  ```

- **complication #2: inout arguments**: we may also need restore temporary values; let another task call be given as below, where the first argument is an **inout** argument

```
bar(b, ++b, c);
```

```
// wrong tranformation
t1 = b;
t0 = b;
b = b + 1;
bar(t1, t0, c);
```

```
// correct transformation
t1 = b;
t0 = b;
b = b + 1;
bar(t1, t0, c);
b = t1;
```

2.2 Dead code elimination

- two types of DC performed:
  - per-statement-type DC
  - remove unused definitions (liveness analysis)
- **DC optimization of DISABLE**: any statement that follws a disable statement can be deleted as long as the statement is within the disabled scope

```
begin: blk
  ...
  disable blk;
  ... // can be removed if disabled block is an ancestor scope
  ... // of the current scope
end
```

  There are a few complications:

  - note that, if the disabled scope is either an external scope or a forward scope, this optimization is not applicable
  - consider the following example:

```
begin: B1
  begin: B2
    if C
      disable B1;
    else
      disable B2;
  end // B2
end // B1
```

    in this case, after the **if-else** statement, the disabled scope is $\mathbf{B1} \cap \mathbf{B2} = \mathbf{B2}$. To support this, we need a simple dataflow analysis.

2.3 Loop unrolling

2.4 Constant propagation

- **complication #1: three special statements**: three types of statements complicates the implementation of constant propagation: **branches**, **loops**, and **scopes**
- **complication #2: time-consuming statements**: wait/event/delay acts as a barrier; since during the suspension of simulation, collected constants may become obsolete since other parts of the design may update the values of the corresponding variables

  - can be enhanced if the variable is not updated from outside
- **complication #3: function arguments**: for some functinos and tasks, some arguments are not eligible for constant propagation
- **implementation sketch**: two main procedures
  - **constant collector**: constants can be collected from 1) *explicit assignments* (e.g. `a = 1`) or 2) *implicit assignments* (e.g. in `if (a == 1) begin ... end`), the variable `a` has value 1 in the then-branch
  - **constant propagator**
- **context**: defined as a mapping from variable to constant values
  - implemented using a stack
  - new context is created when enter a new "scope"

2.5 Tristate buffer transformation

- let a process be given:

```
always @(posedge clk)
  if (cond)
    q <= 1'bz;
  else
    q <= d;
```

  in this case, for hotswapping, we need to store the two values (enable, data)

- **transformation**:

```
always @(posedge clk)
  if (cond)
    t_en <= 1;
  else begin
    t_en <= 0;
    t_d <= d;
  end
```

```
always @(t_en, t_d)
  if (t_en)
    q = 1'bz;
  else
    q = t_d;
```

2.6 DISABLE transformation

```
always @(posedge clk) begin
  S1;
  if (cond2 == 1'b1)
    disable blk;
  S2;
end
```

```
always @(posedge clk) begin
  disable_blk = 0;
  S1;
  if (cond2 == 1'b1) begin
    disable_blk = 1;
  end
  if (!disable_blk) begin
    S2;
  end
end
```

2.7 Continue/Break statement

```
for (i = 0; i < 10; i++) begin
  S1;
  if (cond1)
    continue;
  S2;
  if (cond2)
    break;
end

begin: blk1;
  for (i = 0; i < 10; i++) begin: blk2
    S1;
    if (cond1)
      disable blk2;
    S2;
    if (cond2)
      disable blk2;
  end
end
```

2.8 Multiple event transformation

Same events

```
/* two indentical events */
always @(posedge clk) begin
  S1;
  @(posedge clk);
  S2;
end

/* result */
reg [0:0] state;
initial state = 0;
always @(posedge clk) begin
  case (state) begin
    0: begin
         S1;
         state = 1;
       end
    1: begin
         S2;
         state = 0;
       end
  endcase
end
```

Different events

```
/* two non-indentical events */
```

```
always @(posedge clk) begin
  S1;
  @(negedge clk);
  S2;
end

/* result */
reg [0:0] state;

ixc_edge #(0) PEclk(peclkout, clk);
ixc_cap #(0, 0) Cap(capout0, pecklout, en0, reset0, set0, enxp0);

ixc_edge #(0) PEclk(neclkout, clk);
ixc_cap #(0, 0) Cap(capout1, necklout, en1, reset1, set1, enxp1);


initial begin
   en0 = 0;
   reset0 = 0;
   en1 = 1;
   reset1 = 0;
   state = 0;
end
always @(posedge capout0 or posedge capout1) begin
  case (state) begin
    0: begin
         reset0 = set0;
         S1;
         en1 = set1;
         state = 1;
       end
    1: begin
         reset1 = set1;
         S2;
         en0 = set0;
         state = 0;
       end
  endcase
end
```

Branching: same events

```
always @(posedge clk) begin
  if (cond) begin
    S1;
    @(posedge clk);
    S2;
  end
  S3;
end
```

```
/* result */
always @(posedge clk) begin
  case (state)
    0: begin
         en0 = 1;
         if (a) begin
           S1;
           state = 1;
           en0 = 0;
         end
         if (en0) begin
           S3;
           state = 0;
         end
       end
    1: begin
         S2;
         S3;
         state = 0;
       end
  endcase
end
```

2.9 RTL blocking delay transformation

Consider the following two DUT modules which contain #-delays inside processes.

```
module dut1;
  always begin
    #10;
    S1;
  end
endmodule
```

```
module dut2;
  always begin
    #20;
    S2;
  end
endmodule
```

To support the delay inside DUT modules, three components interact.

- **iscDelay modules**: transformed so that all #-delays are removed
- **ixc_time module**: collects information from iscDelay modules and computes the time until the earliest #-event in iscDelay modules
- **runtime and xc_top_incl.v**: collects information from a) **ixc_time** and **IUS** to compute the time until the earliest next event (either due to #-event in iscDelay or dut to TB-side event); when such computed time has passed, it triggers **eClk**

Note that above example is a simpliest possible case where 1) each module has only one #-delay process and 2) each module has onlye one #-delay.

Transformation of iscDelay module Each process with RTL delay is transformed into one which wakes up at posedge eClk. eClk makes posedge transition exactly when any #-delay-related event (i.e. simetime advances due to #-delay) occurs. In the above example, eClk is triggered at time 10, 20, 30, 40, ... The process in dut1 wakes up at time 10, 20, 30, while the process in dut2 wakes up at time 20, 40, 60, ...

```
module dut1;
  always @(posedge eClk)
    if (TDL2 == DELTA) begin
```

```
module dut2;
  always @(posedge eClk)
    if (TDL2 == DELTA) begin
```

```
      S1;                          S2;
      TDL2 = 10;                   TDL2 = 20;
    end                          end
    else                         else
      TDL2 = TDL2 - DELTA;         TDL2 = TDL2 - DELTA;
  assign TDM1 = TDL2;            assign TDM1 = TDL2;
endmodule                    endmodule
```

There are three important variables which are used for handling delays in DUT modules:

- **TDL2**: this is a process-specific variable, which contains *the time the process has to wait until its #-delay is consumed*
  - **TDL2 = 10** means that "my #-delay will be fully consumed at 10ns later" (or "I need to wak up 10 ns later and executed S1 (or S2)")
  - in case there are multiple processes with #-delays, we need multiple **TDL2** variable
- **DELTA**: this is shared by all iscDelay DUT modules, and contains the "time until the earliest next TB event (in particular, the event due to full consumption of #-delay) over the entire modules"1
  - **DELTA = 10** means that 10 ns has passed since the last #-event
  - **DELTA** value is updated by **ixc_time** and is propagated to each iscDelay module (more will be discussed in **ixc_time** section)
- **TDM2**: this is a module-specific variable, which contains "the time the module has to wait until any of its #-delay processes wake up next"
  - when there is only one #-delay process in the module, **TDM2 = TDL2**
  - when there are multiple #-delay processes in the module, **TDM2 = min(TDL2_1, TDL2_2, ..., TDL2_n)**.

After the transformation, each process with #-delay performsn the following tasks.

1. each process with #-delay checks if the #-delay in the given process has been consumed; this can be checked by comparing **TDL2** value and **DELTA** value
   - if so, execute S1 (or S2)
   - otherwise, update **TDL2** with **TDL2 - DELTA** (**DELTA** time has passed since the last eClk; so we only need to wait **TDL2 - DELTA** time until wakeup)
2. update the **TDM1** value which is the time until the next wakeup
   - **TDM1** values from each iscDelay DUT modules will be collected later by **ixc_time** and will be used to compute **DELTA** (more will be discussed in **ixc_time** sectino)

ixc_time.v: Computing DUT-side delays The role of **ixc_time** module is:

1. to compute the time until the next #-event in iscDelay modules,
2. to inform the runtime of the next iscDelay time (computed above), and
3. to compare the next iscDelay time (computed above) and the next TB time and save the minmum of these to **ixc_time.delta**
   - note that **ixc_time.delta** will be read by all iscDelay modules (and copied into module var **DELTA**

After the transformation of iscDelay modules, each such module will updazte its **TDM2** variable (the time until any of its #-delay process will wake up). The **ixc_time** module collects all **TDM2** values from all iscDelay modules, and compute the minimum of these values, denoted by **nextClkTime**.

The following code is a simplified versin of **ixc_time** module:

```
module ixc_time;
  // compute the delay until the next "event"
  //   - minT: time until next iscDelay-event
  //           (computed in previous eClk tick)
  //   - nextTbTime: time until next event
  //           (computed by xc_top_incl.v)
  assign delta = min(minT, nextTbTime);

  // compute minimum of TDM1 values over all iscDelay modules
  assign minT = min(TDM1_1, TDM1_2, ..., TDM1_n);

  // nextClkTime is the next time when #-event will happen in any
  // of iscDelay modules
  assign nextClkTime = lastClkTime + minT;
endmodule
```

xc_top_incl.v and runtime

```
bit eClk;
event runSwEclk;
import "DPI-C" pure function int xcMatchWriteTbTime();
bit schSwEclk = 0;
bit eotSwEclk;
bit [63:0] nextTbTimeIUS = {64{1'b1}};
assign ixc_time.nextTbTime = nextTbTimeIUS;

// whenever, ixc_time computes the next iscDelay time, forward this
// info to the runtime, so it can properly compute the time to next
// earliest
always @(ixc_time.nextClkTimePO)
  xcNextClkTime(ixc_time.nextClkTimePO);

xc_sch_eval xc_sch_sw_eclk(EotSwEcl, schSwEclk);
always @(runSwEclk) begin:swclk
  longint delay;
  while (1) begin
    schSwEclk = ~schSwEclk;
    @eotSwEclk;

    delay = xcNextEclk();
    if (delay == 64'b0) break;

    nextTbTimeIUs = $time + delay;
    #(delay);
    if (cpi_capture_enable)
      void'(xcMatchWriteTbTime());
    eClk = 1;
    eClk <= 0;
  end
end
```

S/W run: Interaction between 3 components

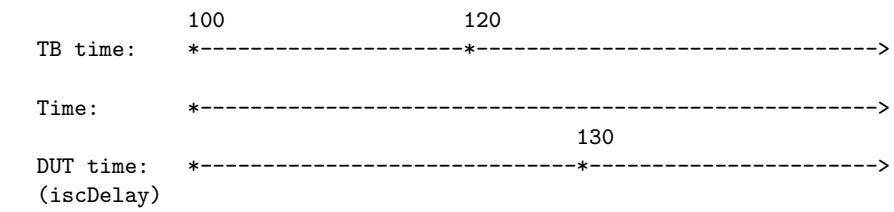
```
/* xc_top_incl.v */
always
  while (1) begin
    // schedule at program block
    delay = xcNextEclk();
    nextTbTime = $time + delay;
    #(delay);
    eClk = 1;
    eClk <= 0;
  end
```

```
/* ixc_time.v */
minT = min(dut1.TDM2, dut2.TDM2);
delta = min(minT, nextTbTime-lastClkTime);
nextClkTime = lastClkTime + minT;
nextDutTime = min(nextClkTime, nextTbTime);
always @(posedge eClk)
  lastClkTime <= min(nextClkTime,
                     nextTbTime);
```

```
/* iscDelay dut1 */
DELTA = ixc_time.delta;
always @(posedge clk)
  if (TDL2 == DELTA)
    S1; TDL2 = 10;
  else
    TDL2 = TDL2 - DELTA:
assign TDM2 = TDL2;
```

```
/* iscDelay dut2 */
DELTA = ixc_time.delta;
always @(posedge eClk)
  if (TDL2 == DELTA)
    S2; TDL2 = 20;
  else
    TDL2 = TDL2 - DELTA;
assign TDM2 = TDL2;
```

Let a simulation be given as shown in the diagram below:



We assume there was an event (either TB-event or iscDelay-event) at time 100. At time 100, eClk should have been triggered which had caused the computation of main variables.

- lastClkTime = 100;
- nextTbTime = 100;
- nextClkTime = 130;

Now, another while loop iteration in the process at **xc_top_incl.v** is about to begin. (i.e. the first statement "delay = xcNextEclk()" in the while loop of **xc_top_incl.v** is about to be executed)

An example sequence of interactions between these components are given below. Leftmost column shows the activity number, and the rest three columns represent three components.

	xc_top_incl.v	ixc_time	iscDelay module
		simtime: 100	
@1	delay = xcNextEclk()		
@2	nextTbTimeIUS = $time + delay		
@3	#(delay); eClk generator process suspended		
@4	assign ixc_time.nextTbTime = nextTbTimeIUS		
@5		determine if next event is TB-event or iscDelay event (by comparing **nextTbTime** and **nextClkTime**	
@6		update **delta** using the **min(nextTbTime, nextClkTime)**; delta means "time to next event"	
@7			DELTA = ixc_time.delta
		simtime: 120	
@8	delay consumed; eClk generator process wakes up		
@9	eClk = eClk		
@10			@(eClk) if TDL2 delay consumed; executed S1; update **TDL2**
@11			TDM2 = min{TDL2}
@12		minT = min{dut.TDM2}	
@13		nextClkTime = lastClkTime + minT	
@14		@(eClk) lastClkTime ← min(nextTbTime, nextClkTime)	
@15	xcNextClkTime(ixc_time.nextClkTime)		

S/W-side optimization

```
/* original */
always begin
  #10;
  S1;
  #20;
  S2;
end

/* HW process */
always @(posedge eClk) begin
  if (TDL2 == DELTA) begin
    case (state)
      0: begin
           S1;
           state = 1;
           TDL2 = 20;
         end
      1: begin
           S2;
           STATE = 0;
           TDL2 = 10;
         end
```

```
     endcase
  end
  else
    TDL2 = TDL2 - DELTA;

/* SW process */
always begin
  #(TD);
  if (!xc_top.hwOutInit) begin /* S/W run */
    case (state) begin
      0: begin
           S1;
           state = 1;
           TD = 20;
         end
      1: begin
           S2;
           state = 0;
           TD = 10;
         end
    endcase
  end
  else begin /* H/W run (wait until swapout */
    @(negedge xc_top.hwOutInit);
    TD = TDL2; // read swapout value of TDL2 and update TD
  end
  localtime = $time;
end

always (posedge xc_top.cpi_capture_enable) begin
  /* compute the value of TDL2 to download to HW */
  TDL2 = TD - ($time - localtime);
end

initial begin
  localtime = $time;
  TD = 10;
end
```

The following is brief explanation of the S/W process;

- **TDL2** and **state** are the only variables which needs to be swapped in and out
  - this means that when we switch from S/W process to H/W process (i.e. when swapin), **TDL2** and **STATE** value should be correctly set
  - **TDL2** are computed on-the-fly when swapin
  - **state** always matches between H/W process and S/W process and we don't need recomputation before swapin
- **TD** is used only in SW process and it means "time until next #-event"
  - **TD** is the "SW couterpart" of **TDL2** (i.e. **TD** is for S/W, **TDL2** is for H/W)
  - when we switch from H/W process to S/W process (i.e. swapout) **TD** should be correctly set using the **TDL2** value which was just updated from H/W.

2.10 Assignment Operator

3 Primitives

```
module ixc_edge(ev, s);
  parameter DIR = 0; /* 0: posedge, 1: negedge, 2: duedge */
  parameter ASYNC = 0;
  output ev;
  input s;

  wire fclk;  // quickturn fast_clock fclk;
  wire xc_top_eventOn // quickturn name_map xc_top_eventOn xc_top.eventOn

  generate
    if (ASYNC) begin
      reg _zzsr = 0;
      always @(posedge fclk) begin
        if (xc_top_eventOn)
          _zzsr <= s;
      end
      assign ev = xc_top_eventOn &
                  (((DIR == 0) & (s & ~_zzsr)) ||
                   ((DIR == 1) & (~s & ~_zzsr)) ||
                   ((DIR == 2) & (s ^ _zzsr)));
    end
    else begin
      if (DIR == 0) begin
        Q_PEDECT detect(ev, s, 1'b1);
      end
      else if (DIR == 1) begin
        Q_NEDECT detect(ev, s, 1'b1);
      end
      else if (DIR == 2) begin
        Q_EVECT detect(ev, s, 1'b1);
      end
    end
  endgenerate

endmodule
```