XRTL

Mentor Graphics eXtended RTL

- Goal to ensure high performance on emulator
- Subset of SystemVerilog --some behavioral constructs

  Major special extensions:
  - Clock and reset specifications
    - Clock synchronous RTL modeling
  - non-RTL constructs
    - Initial blocks for signal initialization
    - Named event triggers/synchronous for signaling mechanisms
    - Procedural assignments on a register from multiple processes
  - Implicit state machines within initial blocks

- SystemVerilog DPI functions for communication
  - Imported/Exported functions and tasks
  - SCE-MI 2.0 DPI compliant transaction pipes

XRTL

Mentor Graphics eXtended RTL

- Rules for writing code for synthesis
  - Initializations
  - FSMs
  - Loops
  - Multiple drivers
  - Events
  - Waits
  - Clocks and Reset
    - Variable clock delay
    - Clocked tasks
  - Display, I/O, system functions

Initializations

- Simple initial blocks

```
parameter ALL_OK = 1;
initial begin
  receivedCount <= 0;
  receivedStatus <= ALL_OK;
end
```

- Initializations in initial implicit-state-machine

initializations

Initial FSM

```
initial begin
  reset <= 0;
  count <= 1;
  @(posedge clk);
  while(count < reset_cycles)
   begin
       count <= count + 1;
       @(posedge clk);
  end
end
```


Initializations (Cont.)

- Initialization during declaration

```
reg [1:0] state = 2'b00;
```

- All initialized variables must be clocked registers
- Blocking/Non-blocking assignments generally synthesized as non-blocking.
  - Good practice to use non-blocking to keep simulation to synthesis compatibility
- Always use a non-blocking assignment if the variable will be read in any process/assign/declaration other than the one that assigns it.

XRTL FSM : Explicit State Machine

```
always @(posedge clock)
begin
   if( Reset == 1)
     TokenOut <= 0;
   else if( TokenOut != 0 )
       UploadEgress( TokenOut );
end
```

```
always @(posedge clock or posedge Reset)
   begin
    if( Reset == 1)
      TokenOut <= 0;
    else if( TokenOut != 0 )
        UploadEgress( TokenOut );
  end
```

Not an XRTL FSM
(two conditions for the same state)

XRTL FSM: Implicit State Machine

- **Part of RTL subset, not used as often for designs**
- **Make transactor coding simpler**
  - No need to enumerate states explicitly, they are implicit
  - Enable pipelining through multi-cycle loops
- **Rules for RTL implicit machines:**
  - First statement has to be a wait on clock
  - All wait on clock statements have to be using same edge of same clock

```
initial begin
  reset = 0;
  count = 0;
end

always begin
  @(posedge clk); // state-0
  while(count < reset_cycles)
   begin
       count = count + 1;
       @(posedge clk);// state-1
  end
  reset <= 0;
end
```

Additional Forms of Implicit Machines in XRTL

```
initial forever begin
..
end
```

Initial-forever blocks;
Same as always blocks

Imp-machines in initial blocks

Initial imp-machines with initializations

```
initial begin
  reset <= 1;
  count = 0;
  @(posedge clk);  // state-0
  while(count < reset_cycles)
   begin
        count = count + 1;
        @(posedge clk); // state-1
  end
  reset <= 0;
end
```

```
always begin
  wait(data_available);
  @(posedge clock);
  ..
end
```

Implicit machines that start with synchronous wait/event

Transforming an Explicit Machine to Implicit Machine

```
initial state = STATE_0;
always @(posedge Clock)
   begin
     case( state )
       STATE_0: begin
         port0 <= data;
         state <= STATE_1;
       end
      STATE_1: begin
         port1 <= data;
         state <= DONE;
       end
       DONE: begin end
     endcase
end
```

```
always begin
  @(posedge clock ); // STATE_0
  port0 <= data;
  @(posedge clock ); // STATE_1
  port1 <= data;
  while(1)
    @(posedge clock ); // DONE
end
```

Loops: Different Types (Cont.)

- **for : explicit number of iterations, explicit index-variable(s)**
  - for(int i = 0; i <= j; i = i + 1) begin .. end

```
for(int i = 0; i < MAXDATA; i =i +1)
   begin
      if(loc == BUFSIZE) begin
         loc = 0;
         ->fetch_more;
         @(posedge clk);
      end
      data <= buffer[loc];
      loc = loc + 1;
      ....
   end
```

- **Iterations maybe clocked (unconditionally OR conditionally)**

Unrolled Loops

- **Loops with conditional clocks OR no clocks are unrolled**

```
reg [4:0] count;
..
for(i=0;i < count; i = i+ 1) begin
  if(request[i] == 1) process(i);
end
```

```
i = 0;
while(i<32) begin
   if(request[i] == 1) process(i);
   i = i + 1;
end
```

```
input [31:0] count;
..
for(i=0;i < count; i = i+ 1) begin
if(request[i] == 1) process(i);
end
```

- **Consequently, unrolled loops with unbounded iterations are illegal, unless if they are inside an export-call**
- **RTLC/TBX unroll loops and after the iterations go beyond 5000 they give an error, this number can be changed with –max_loop_cnt**

Data Dependent Loops in Export Calls

```
reg [7:0] mem [1<<19:1];
export "DPI-C" function void burst_write;
 function void burst_write;
   input int length;
   input bit [255:0] data;
   input int start; // start address
   begin
       for(int i = 0; i < length; i = i + 1) begin
           mem[start+i] = data[(8*i)+7 -: 8];
       end
   end
 endfunction
```

- **Only allowed in zero time export-functions/tasks**

Loops: Different Types

- while : **most generic**
  - while(!busy) begin .. end
- repeat : **explicit number of iterations, with an implicit index-variable**
  - repeat(count) @(posedge clk);
- forever : **infinite range**
  - forever begin .. end

```
while(data_available)
  begin
     data <= getdata(..);
     valid <= 1;
     @(posedge clk);
  end
```

Multiple Drivers

- **Follow last driver wins semantics**
- **Assigning block must not be**
  - Combinatorial always
  - Asynchronous always
- **Assigning blocks can be**
  - Clocked export task
  - Initializing block
  - 0-time-export task/function
  - XRTL FSM

```
reg wait_for_C;

initial wait_for_C <= 0;

export "DPI-C" task go;
task go;
   begin
      wait_for_C <= 1;
   end
endtask

always begin
    @(posedge clk)

   while(!wait_for_C)
  @(posedge clk);
     wait_for_C <= 0;...
end
```


Cyclized Signaling: Named event

- **A signaling mechanism to initiate operations in FSM**
- **Event wait must be followed by @(posedge clock)**
- **Event trigger must be in:**
  - an export call
  - any synchronous clocked process
- **Combined with export-call trigger, provides an easy C->HDL signaling mechanism**

```
event serviceCallDetected;

initial begin
    @(posedge clock );
    forever begin
      // Wait for event to occur
      @( serviceCallDetected );
        // Always required by XRTL
      @( posedge clock );
      ...
    end
```

- **Clocks associated with named-event can be any clock.**

Cyclized Signaling: C->HDL signal

- Initialization during declaration

```
reg wait_for_C;

initial wait_for_C = 0;

export "DPI-C" task go;
task go;
begin
    wait_for_C = 1;
end
endtask

always begin
    @(posedge clk)
    wait_for_C = 0;
    while(!wait_for_C)
        @(posedge clk);
    ....
end
```

```
event wait_for_C;

export "DPI-C" task go;
task go;
begin
    ->wait_for_C;
end
endtask

always begin
    @(wait_for_C);
    @(posedge clk);
    ...
end
```

More intuitive way of C signaling to HDL

- Initializations in initial implicit-state-machine

Cyclized Signaling: Waking up a process

- Note the forever between event and @(posedge clock)
- Once the event is triggered, forever loop takes over

```
event start;

export "DPI-C" task go;
task go;
begin
    ->start;
end
endtask
```

```
initial begin
    @(start);
    forever begin
        @(posedge clk);
        ...
    end
end
```

Process

Cyclized Signaling : Wait

- Must be followed by a clock, same as event
- Good way to sense and react to, some HDL activity
- For signaling, events are more intuitive
- Wait/event can also be used in clocked export tasks, but they can not be the first statement

```
module
   transactor(mii_ready, ..
  input mii_ready;
  ...

  always begin
    wait(mii_ready);
     @(posedge clock);
    ...
  end
```

```
 initial begin
    wait(pci_init);
    forever begin
        @(posedge clock);
        ...
    end
  end
```


Clock and Reset Rules

- Only allowed in XRTL modules
- Clock must be a scalar register
- Clock-assignment must be blocking
- One initial block -> one clock
- Clock must not have multiple drivers
- Clock can only be inverted or assigned a constant
- Support both static and variable delays (restrictions apply)
  - Parameters can be used
- 'forever' must be used
- Must comply with one of 4 possible formats

Reset Specification

Special pragma

```
// tbx clkgen
 initial begin
   reset = 1;
   #100 reset = 0;
 end
```

Clock Specification #1

```
 // tbx clkgen
 initial begin
   clock = 0;
    #5;  // optional phase delay
   forever #10 clock = ~clock;
 end
```

#Delays in clock/reset specifications must be constants or parameters

Clock Specification #2

Optional inactive clock edge optimization

```
// tbx clkgen inactive_negedge
  initial begin
    clock = 0;
    forever begin
      #10 clock = 1;
      #15 clock = 0;
    end
  end
```

Assignments have to be blocking

Clock Generator Examples

hash delays only in tbx clkgen pragmas

```
// tbx clkgen
initial begin
    clk = 0;
    forever #5 clk = ~clk;
end
```

Add phase/duty cycle

```
// tbx clkgen
initial begin
    clk = 0;
    #3;
    forever begin
      clk = 1;
      #4;
      clk = 0;
      #6;
    end
end
```

Add phase

```
// tbx clkgen
initial begin
    clk = 0;
    #3;
    forever begin
        clk = ~clk;
        #5;
    end
end
```

Add duty cycle

```
// tbx clkgen
initial begin
    clk = 0;
    forever begin
      #6 clk = 1;
      #4 clk = 0;
end
```


Reset Generation/$finish()

```
// tbx clkgen
initial begin
    reset = 1;
    #10;
    reset = 0;
    #10000 $finish();
end
```

In //tbx clkgen block

```
always @(posedge clock)
begin
    if(all_done) $finish();
    ...
end
```

In XRTL FSM

Variable Clock Delay

- TBX supports variable delays inside clock generator
- Rules:
  - Delay variable must be a shortint
  - Delay variable must initialized during declaration
    - e.g.: shortint d = 0;
  - Can only be modified non-blockingly after time 0

```
shortint d = 10;
reg clk;
event slow_down;
// tbx clkgen
initial begin
    clk = 0;
    forever begin
    #(d) clk = ~clk;
    end
end
```

```
always begin
    @(slow_down);
    @(posedge clk);
    d <= d * 2;
end
```


Clocked Tasks

- **A task with a wait-on-clock statement inside it**
  - **The clock statements can be put anywhere in the clocked tasks**

```
task ProcessData;
input [7:0] in;
output [7:0] out;
begin
    data_in = in;
    data_rdy = 1;
    @(posedge clk);
    data_rdy = 0;
    while (!data_processed)
        @(posedge clk);
    out = data_out;
end
endtask
```

Same edge of the same clock must be used inside the clocked tasks as the clock and edge used inside the initiating block.

```
always @(posedge clk) ProcessData(..); ----> CORRECT
always @(negedge clk) ProcessData(..); ----> INCORRECT
always @(podedge new_clk) ProcessData(..);----> INCORRECT
```

Limitations for Clocked Tasks

- **TBX does not support z write inside clocked task and an error message is issued**
- **Explicit loop breaking in case of clocked task is required**
  - Loops that are calling clocked tasks, and do not have a wait-on-clock in its main path, are treated as combinational loops
  - The only work-around is to add an explicit wait-on-clock inside the loop

```
task apply_stim;
begin
   @(posedge clock);
    data = inp;
    while(!consumed) @(posedge clock);
end
endtask
always begin
    @(data_recd);
    while(more_data)
    begin
        inp = d[idx++];
        apply_stim;          <-- @(posedge clock)
    end
end
```


Display/File-I/O System Tasks and Functions

- **Following Display/File IO calls are supported:**

- **$display, $displayb, $displayo, $displayh**
- **$fdisplay, $fdisplayb, $fdisplayo, $fdisplayh**
- **$write, $writeb, $writeo, $writeh**
- **$fwrite, $fwriteb, $fwriteo, $fwriteh**
- **$fopen**
- **$fclose**
  - Use rtlc -compile_display option in tbx.config file
  - Use tbxrun –disable_display to disable the above tasks during runtime

- **Note: Use of –compile_display can lead to performance, area and compile time issues. Use with discretion**