Debugging Aids

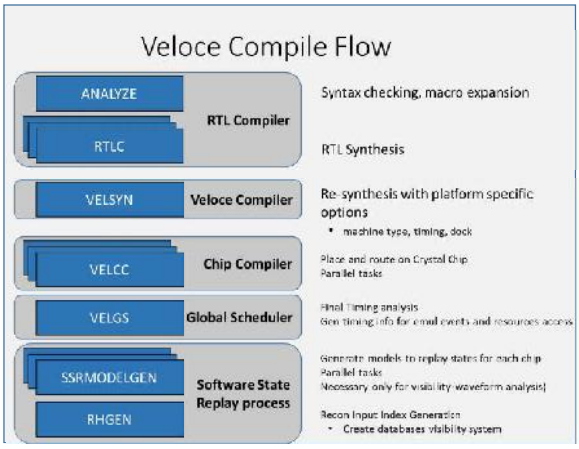
- Trace System
  - Circular buffer stores design input and internal node logic values
    - Unlimited depth through continuous upload
  - Selective upload
    - Don't need visibility of every signal
  - On demand visibility
  - Store ON/Store OFF management
- Features
  - Full visibility of entire design
  - All flops triggerable by default
  - RTL and source level debugging
  - Line break pointing and advanced triggers
  - Advanced verification using simulation time
  - Built in logic analyzer and graphical path browser
  - Waveform comparison
  - Check point save/restore
  - Force/release

More on Verbose Logs

- **Logs provide -**
- **Call Graph for all imported/exported DPI tasks/functions. The call graph gathers the following information:**
  - Simulation time
  - Direction of call (----> for HVL to HDL, <----- for HDL to HVL)
  - Appropriate argument values (for example, inputs for HVL to HDL)
  - Action for each call
  - Enhanced hang situation detection (on HDL/HVL side), an error message is issued after 1 M uclocks are detected for performing zero-delay activity, or
- **Log the software advance and hardware advance (from C side)**

Trace capturing

- Trace data collected in buffers frequently, but not continuously!
  - Snapshots at intervals of time
  - Continuous capture would fill memory more quickly
- Captured trace data when uploaded, used to generate full trace
  - *SSRmodelgen* produces equivalent C version of design
  - *Velwavegen* fills in details to create full trace using C version and captured data
- Once trace data uploaded, don't need to be connected



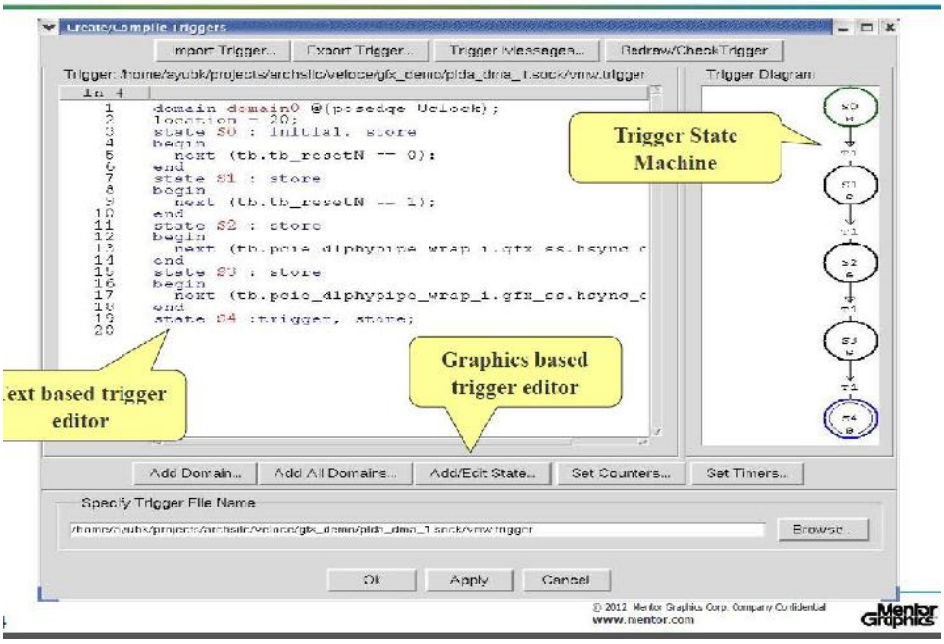
Waveform Capture

- Simulation time-based capture
  - Manually run for some number of cycles, stop and upload waveform data
- Line Breakpointing Capture
  - Put breakpoint on line of code to stop simulation and then manually upload
  - Not available by default
    - Add to tbx.config: `rtlc -debug` and `rtlc -preserve`
    - Right click some module, select view source, can add breakpoint
- Trigger capture
  - Enables capturing and uploading waveform during simulation

Triggers

- Triggers operate with a user defined sequence graph of abstract hardware states tied to hardware execution
- Supports debug activites
  - monitor executing progress
  - stop the clock upon reaching state
  - dynamically control when trace data is captured
    - hwtrace on|off (default: trace is on)
- Defined in Verilog-like syntax:
  - Any state element can be a trigger input (guards)
  - Up to 4 transitions to other states
  - Support for repeat counts, global counters and timers
  - Can write and download triggers
- Top level declarations: state, timer, counter

Trigger Editor And Display



Trigger syntax

```
state name :attrib
  transition type
    (condition)
      repeat control
      arguments
```

```
timer  name
  begin
    value =  num
    target = state
  end
```

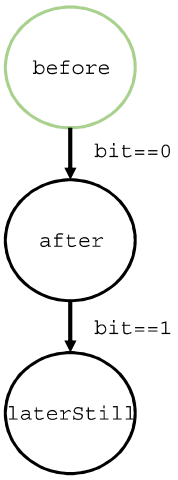
```
counter  name
```

- State
  - Attributes
    - initial, trigger, store, store if, nostore, stop, stop if
  - Transitions: next, jump
    - Expressions
    - Verbs: affect global counters and timers
      - Enable, set, reset, increment, decrement
- Timers
  - value
  - target
- Counter
  - name
- Domain
  - times of expression evaluation
    domain <domain name> @(<edge-expression>)

- Recommendations:
  - Keep definitions simple
    - Break up complexity
    - Don't do too much in one trigger
  - Use good names
  - Set up a timer to cause simulation to eventually stop
    - Helpful in case other triggers don't occur
  - Identify setup behavior that precedes interesting conditions
    - turn off store

Simple state definition example
signal edge* detection

```
state before
     next (bit == 0);

state after
     next (bit == 1);

state laterStill
     …
```



*not clock edge

state name :attrib
transition type
(condition)
repeat control
arguments

Simple Matches

```
state StateNameA
  next (ScalarName == 1'b0);

state StateNameB
  next (BusName[15:0] == 16'hABCD);

state StateNameC
  next (  ScalarName == 0
        && BusName[15:0] == 'hABCD
        && (yoma != 'hCAB0));
```

Simple Repetition

```
state SomeState
  next (foobar = 0) 37 times;
```

Match N contiguous times

use N times next transition and jump-to-self

```
state GroovyState begin
  next (one.two.address[31:0]
        =32'h0A0A0A0A) 49 times;
  jump (one.two.address[31:0]
        != 32'hA0A0A0A0A)
       target = 0;
end
```

Relative to present state
Could also use state name

Match exactly N contiguous times

```
state  BeginExactly
begin
  next (f) N times;     // f is the cond
  jump (!f) target = 0;
end

state CheckNoMore
begin
   next (f)
   jump (!f) target = 2;
end

state KeepLooking
  jump (!f) target = -2;

state nextOneAfterExactNcontigMatch
…
```


state name :attrib
transition type
(condition)
repeat control
arguments

Match min N max M contiguous times

```
state  BeginExactly
begin
  next (f) N times;
  jump (!f) target = BeginExactly;
end

state CheckNoMore
begin
   next (f) P times;         // P = M-N
   jump (!f) target = NextState;
end

state KeepLooking
  jump (!f) target = BeginExactly;

state nextOneAfterExactNcontigMatch
…
```

Conditional Triggers

state name :attrib
transition type
(condition)
repeat control
arguments

```
domain domain0 @(posedge clk);
state FirstCompare
  next (shift_testing_rising.src0.dataOut[15:0] == 16'h3232);
state SecondCompare
  next (shift_testing_rising.src0.dataOut[18:3] == 16'h3232);
state ThirdCompare
  next (shift_testing_rising.src0.dataOut[21:6] == 16'h3232);

state Finish: trigger;
```

Don't Care Conditional Triggers

```
state First_Dont_Care_Check: initial
  next (shift_test_rising.src0.dataOut[3:0] == 4'bx0x1);
state Second_Dont_Care_Check
  next (shift_test_rising.src0.dataOut[3:0] == 4'b1x0x);

state Finish: trigger;
```


Timer Example

```
timer TooLong
Begin
  value = 100_000_000;
  target = TookTooLong;
End
…
state Origin
  next(SomethingOrOther ==16'hABCD)
     enable timer TooLong;
…
state NormalTrigger: trigger;
state TookTooLong : trigger;
```

Counting clocks

- Count default clock edges in default domain:

```
state WaitAround next (1) 100 times;
```

condition: always true

- Count explicit clock edges in explicit domain:

```
state WaitAround
begin
   next (1) 100 times;
   domain Simple @(posedge YoCLK);
end
```


Global Counters

```
increment
decrement
reset
```

```
counter Loopy;
state Start
begin
  next(1)
  reset counter Loopy;
end
…
state StartOfLoop
  next(SomethingOrOther == 32'h1234)
       increment counter Loopy;
…
state EndOfLoop
begin
  next(1) condition (Loopy == 17);
  jump(1) target = StartOfLoop
```

Example Edge Detection Trigger

```
domain domain0 @(posedge clk);
//consider signal shift_test_rising.sr0.dataOut[0] rise/fall edge
   has to detect
state Start
begin
 jump (shift_test_rising.sr0.dataOut[0] == 1'b0) target = First0;
 jump (shift_test_rising.sr0.dataOut[0] == 1'b1) target = First1;
End

state First0
jump (shift_test_rising.sr0.dataOut[0] == 1'b1) target =
   RisingEdge;

state First1
jump (shift_test_rising.sr0.dataOut[0] == 1'b0) target =
   FallingEdge;
state RiseingEdge: trigger;
state FallingEdge: trigger;
```


Absolute Jump Trigger

- In this type of trigger the target states of the jump instruction are specified by their exact names.

```
state S1: initial
begin
 jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b111 ) target = S1;
 jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b101 ) target = S3;
end
state S2
begin
jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b000 ) target = S1;
jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b011 ) target = final;
end
state S3
 jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b100) target = S4;
state S4
 jump( shift_test_rising.sr1.dataOut[2:0] == 3'b010 ) target = S2;
state final:trigger;
```

Relative Jump Trigger Example

- The target state of the jump instruction can be specified by it's position relative to the current state where jump is instruction is encountered.

```
state S1: initial
begin
 //self target using 0
 jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b111 ) target = 0;
 jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b101 ) target = 2;
end
state S2
begin
 jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b000 ) target = -1;
 jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b011 ) target = 3;
end
state S3
 jump ( shift_test_rising.sr1.dataOut[2:0] == 3'b100) target = 1;
state S4
 jump( shift_test_rising.sr1.dataOut[2:0] == 3'b010 ) target = -2;
state final:trigger;
```


Expression Logic Example

```
state S0:initial
 next((shift_test_rising.sr0.dataOut[3:0] == 4'b1010) &&
       (shift_test_rising.sr1.dataOut[3:0] == 4'b1100 ||
   shift_test_rising.sr1.dataOut[3:0]  != 4'b0101) ) 2 times;
state S1
begin
 next (shift_test_rising.sr0.dataOut[3:0] == 4'b1100 &&
   shift_test_rising.sr1.dataOut[3:0]  == 4'b1100) 2 times;
 jump (shift_test_rising.sr0.dataOut[3:0] == 4'b1100 &&
   shift_test_rising.sr1.dataOut[3:0]  == 4'b0011 &&
   shift_test_rising.sr1.dataOut[7:4]  == 4'b1100 &&
   shift_test_rising.sr0.dataOut[7:4] == 4'b1010) target = 0;
end
state S2
begin
 next (shift_test_rising.sr0.dataOut[3:0] != 4'b1100 ||
   shift_test_rising.sr1.dataOut[3:0] != 4'b0011) 2 times;
 jump (shift_test_rising.sr0.dataOut[3:0] == 4'b1100 ||
   shift_test_rising.sr1.dataOut[3:0]  == 4'b0011 ||
   shift_test_rising.sr1.dataOut[1:0]  != 2'b11) target = 0 ;
end
state Final: trigger;
```

Expression Array Example

```
state S0:initial
 next(shift_test_rising.sr0.dataOut[10 - 3: 3 + 1] == 4'b1010);

state S1
 next (shift_test_rising.sr0.dataOut[(2 * 5 + 3) / 6 - 1] == 1'b1);

state S2
 next (shift_test_rising.sr0.dataOut[9/3: 9%3] != 4'b1100 );

state Final: trigger;
```


Expression Reduction Example

```
state Start:initial
 next(shift_test_rising.sr0.dataOut[3:0] == 4'hf);

state S0
 next(&shift_test_rising.sr0.dataOut[3:0] ) 2 times;

state S1
 next (|shift_test_rising.sr0.dataOut[3:0]) 2 times;

state S2
 next (~|shift_test_rising.sr0.dataOut[3:0]) 2 times;

state S3
 next (~&shift_test_rising.sr0.dataOut[3:0]) 2 times;

state Final: trigger;
```

Triggering – store/nostore

hdl

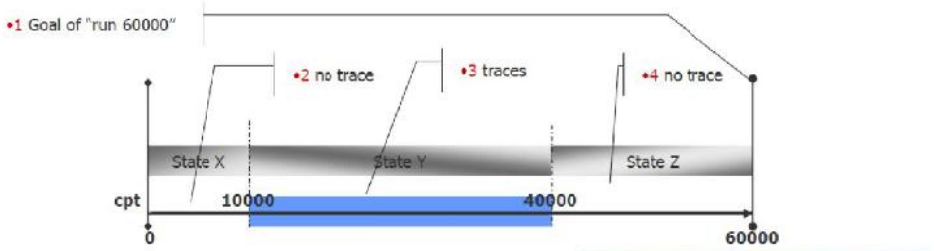
```
# trace depth == 40000

reg [] cpt;

always @(posedge clk1x)
  cpt <= cpt + 1;
```

tcl

```
trigger download my.trigger
run 60000
upload –tracedir mywave
```

my.trigger

```
state X : initial, nostore
begin
  next ( Tb.cpt[ ] == 10000);
end
state Y : store
begin
  next ( Tb.cpt[ ] == 40000);
end
State Z : nostore;
```



Triggering – trigger

hdl

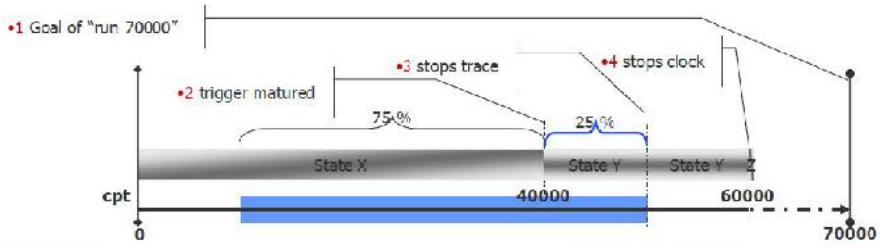
```
# trace depth == 40000

reg [] cpt;

always @(posedge clk1x)
  cpt <= cpt + 1;
```

tcl

```
trigger download my.trigger \
  –position 75        (3)
(1) run 70000
upload –tracedir mywave
```

my.trigger

```
state X : initial, store
begin
  next ( Tb.cpt[ ] == 40000);
end
state Y : trigger, (2) store
begin
  next ( Tb.cpt[ ] == 60000);
end
State Z : stop;   (4)
```



Triggering – trigger (cond't)

hdl

```
# trace depth == 40000

reg [] cpt;

always @(posedge clk1x)
  cpt <= cpt + 1;
```

tcl

```
trigger download my.trigger \
  -position 75          (3)
(1) run 70000
upload -tracedir mywave
```

my.trigger

```
state X: initial, store
begin
  next (Tb.cpt[] == 20000);
end
state Y: trigger (2) store
begin
  next (Tb.cpt[] == 60000);
end
State Z: stop;   (4)
```