Modeling Multi-Phase Systems and Level Sensitive Latches in VIS

Szu-Tsung Cheng
Rajeev Kumar Ranjan
Robert K. Brayton

December 5, 1996

This document describes how to model multi-phase latches as well as level sensitive latches using the "synthesizable subset of Verilog for `vl2mv`".

First, an example Verilog program is will be given (which consists of latches controlled by the rising or falling edges of a clock). The example also includes a level sensitive latch. We will convert the Verilog into BLIF-MV using `vl2mv` and simulate the extracted FSM using VIS. We will then demonstrate how to write a Verilog wrapper to pick up input vectors and apply it to the example circuit. A $Verilog-XL^{TM}$ simulation trace derived from the wrapper is shown. Finally, we compare the simulation outputs from VIS and $Verilog-XL$.

By default, `vl2mv` treats input programs as FSMs controlled by only one clock. For such programs, the clock of a system is merely used to synchronize all the state variables. Therefore, `vl2mv` simply drops these clocks, and state variables (BLIF-MV `.latch`) are allocated for register variables (Verilog `reg`).

However, a multi-phase (or level sensitive) design similar to the following (`two_phase.v`) can also be described:

```
module two_phase(clk, in1, in2, in3, in4, out1, out2, out3, out4);
input clk;
input [3:0] in1;
input [3:0] in2;
input [1:0] in3;
input [1:0] in4;
output [1:0] out1;
output out2;
output [1:0] out3;
output [1:0] out4;

reg [1:0] out1;
reg out2;
reg [1:0] out3;
reg [1:0] out4;

initial out1 = 0;
initial out2 = 0;
initial out3 = 0;

// Process 1 -- Phase 1 process, out1 is updated on rising edge of clk
always @(posedge clk)
    begin
    out1 = in1 + in2;
    end

// Process 2 -- Phase 2 process, out2 is updated on falling edge of clk
//              Note: ^ is bitwise exclusive XOR.
always @(negedge clk)
    begin
    out2 = in1[0] ^ in1[1];
    end

// Process 3 -- Phase 1 process, out3 is updated on rising edge of clk
```

```
always @(negedge clk)
    begin
    out3 = in3 - in4;
    end

// Process 4 --
// out4 is a level sensitive latch, which is transparent on clk==1
always @(clk or (in1 - in2))
    begin
    if (clk)
        out4 = in1 - in2;
    end

endmodule
```

Here different processes (Verilog **always** statements) are activated at different time points. In this case, the **-c** option of **vl2mv** should be used to direct **vl2mv** to interpret the circuit in "explicit clocking" mode [?]. In this mode, **vl2mv** converts event guards (Verilog **@** constructs) into edge-detectors [?]. This way, different clock phases can be detected by different event-detectors and thus fire different processes. Note, the extracted FSM can only serve as a model of the original program; it may look very unlike the corresponding hardware.

For reference, at the end of this document is the extracted BLIF-MV file for the above Verilog example. One can see that the process

```
always @(negedge clk)
    begin
    out3 = in3 - in4;
    end
```

is converted into

1. an edge-detector (which consists of a latch used to remember the previous value of clk and an xor gate used to detect value change),
2. a sequencer (or *timing machine* [?]) which serves as a program counter to pin-point the current @ of a sequential process)
3. the necessary logic used to implement in3 - in4 and
4. resolution logic (used to resolve the possible contention if multiple processes assign to the same variable).

It is easy to manually convert the process into a edge-triggered latch with input from a subtracter. It is not so obvious by looking at the BLIF-MV file.

"vl2mv -c -F two_phase.v" (explicit clocking, and no timed RQ-automata generation) can be used to extract a BLIF-MV file (two_phase.mv) from the above program. The -F option makes vl2mv throw away timing information (# in Verilog) in a program. If timing information is not thrown away, blif-mvt (a timed extension of blif-mv) will be generated. VIS does not recognize blif-mvt. Before applying -F option, make sure that delays (#) in the source program can be safely removed without affecting the behavior of the program.

In the generated BLIF-MV file, one sees latches not only for out1, out2, out3, and out4 but also clk$prev0 clk$prev1, clk$prev2, clk$prev3, (used to check value change of clk) and _n46$prev4<0>, _n46$prev4<1>, _n46$prev4<2>, _n46$prev4<3>, (used to check value change of in1-in2). These are used in "edge-detectors"; one edge-detector per each process. In addition, <T>000001 (<T>000002, <T>000003, <T>000004) are created, which are process location counters (a state in a *timing machine*) used to keep track of the location of each process.

Next we simulate the extracted FSM using VIS' FSM simulator. We use input vectors to explicitly control the clk signal. The same technique can be used to exercise circuits controlled by more than one clock. Note that since register variable out4 is not initialized, we have "quasi-inputs" (out4<0>$INIT and out4<1>$INIT) for a nondeterministic initial state. Also note that currently the nondeterminism is not used by the VIS simulator (which we will observe later). in1<0> is the l.s.b. of vector in1 and in1<3> is the m.s.b. of in1.

Stimulus vector file (two_phase.vect):

```
.inputs  clk in1<0> in1<1> in1<2> in1<3> in2<0> in2<1> in2<2> in2<3> in3<0> in3<1> in4<0> in4<1> \\
    out4<0>$INIT out4<1>$INIT
.latches <T>000001 <T>000002 <T>000003 <T>000004 _n46$prev4<0> _n46$prev4<1> _n46$prev4<2> _n46$prev4<3> \\
    clk$prev0 clk$prev1 clk$prev2 clk$prev3 out1<0> out1<1> out2 out3<0> out3<1> out4<0> out4<1>
.outputs out1<0> out1<1> out2 out3<0> out3<1> out4<0> out4<1>
.initial <1> <2> <3> <4> 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1

.start_vectors

# clk in1<0> in1<1> in1<2> in1<3> in2<0> in2<1> in2<2> in2<3> in3<0> in3<1> in4<0> in4<1> out4<0>$INIT out4<1>$INIT
0  0 0 0 0  0 0 0 0  0 0  0 0  0 0
1  0 0 0 0  0 0 0 0  0 0  0 0  0 0
0  0 0 0 0  0 0 0 0  0 0  0 0  0 0
1  0 0 0 0  0 0 0 0  0 0  0 0  0 0
0  0 0 0 0  0 0 0 0  0 0  0 0  0 0
1  0 0 0 0  0 0 0 0  0 0  0 0  0 0
0  0 0 0 1  0 0 0 0  1 0  0 0  0 0
1  0 0 0 1  0 0 0 0  1 0  0 0  0 0
0  0 0 0 0  0 1 0 1  0 1  0 1  0 0
1  0 0 0 0  0 1 0 1  0 1  0 1  0 0
0  1 0 0 1  0 0 1 1  0 1  0 0  0 0
1  1 0 0 1  0 0 1 1  0 1  0 0  0 0
0  0 0 1 0  1 1 0 0  1 1  0 0  0 0
1  0 0 1 0  1 1 0 0  1 1  0 0  0 0
```

Having the BLIF-MV and input vector file, we can simulate the circuit using the following VIS command:

```
UC Berkeley, VIS Release 1.0 (compiled 4-Jan-96 at 11:18 AM)
vis> read_blif_mv two_phase.mv
Warning: Some variables are unused in model two_phase.
vis> flatten_hierarchy
vis> static_order
vis> build_partition_mdds
vis> simulate -i two_phase.vect -o two_phase.result
```

VIS simulation output (`two_phase.result`):

```
# UC Berkeley, VIS Release 1.0 (compiled 4-Jan-96 at 11:18 AM)
# Network: two_phase
# Input Vectors File: two_phase.vect


.inputs  clk in1<0> in1<1> in1<2> in1<3> in2<0> in2<1> in2<2> in2<3> in3<0> in3<1> in4<0> in4<1> \\
    out4<0>$INIT out4<1>$INIT
.latches <T>000001 <T>000002 <T>000003 <T>000004 _n46$prev4<0> _n46$prev4<1> _n46$prev4<2> _n46$prev4<3> \\
    clk$prev0 clk$prev1 clk$prev2 clk$prev3 out1<0> out1<1> out2 out3<0> out3<1> out4<0> out4<1>
.outputs out1<0> out1<1> out2 out3<0> out3<1> out4<0> out4<1>
.initial <1> <2> <3> <4> 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1

.start_vectors

# clk in1<0> in1<1> in1<2> in1<3> in2<0> in2<1> in2<2> in2<3> in3<0> in3<1> in4<0> in4<1> out4<0>$INIT out4<1>$INIT ; \\
    <T>000001 <T>000002 <T>000003 <T>000004 _n46$prev4<0> _n46$prev4<1> _n46$prev4<2> _n46$prev4<3> \\
    clk$prev0 clk$prev1 clk$prev2 clk$prev3 out1<0> out1<1> out2 out3<0> out3<1> out4<0> out4<1> ; \\
    out1<0> out1<1> out2 out3<0> out3<1> out4<0> out4<1>

0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ; <1>    <2>    <3>    <4>    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 ; 0 0 0 0 0 1 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ; <1>    <2>    <3>    <4>    0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 ; 0 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ; <1>    <2>    <3>    <4>    0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ; <1>    <2>    <3>    <4>    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ; <1>    <2>    <3>    <4>    0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ; <1>    <2>    <3>    <4>    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0
```

```
0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 ; <1>     <2>     <3>     <4>     0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0
1 0 0 0 1 0 0 0 0 1 0 0 0 0 0 ; <1>     <2>     <3>     <4>     0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 ; 0 0 0 1 0 0 0
0 0 0 0 0 0 1 0 1 0 1 0 1 0 0 ; <1>     <2>     <3>     <4>     0 0 0 1 1 1 1 1 0 0 0 1 0 0 0 ; 0 0 0 1 0 0 0
1 0 0 0 0 0 1 0 1 0 1 0 1 0 0 ; <1>     <2>     <3>     <4>     0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 ; 0 0 0 0 0 0 0
0 1 0 0 1 0 0 1 1 0 1 0 0 0 0 ; <1>     <2>     <3>     <4>     0 1 1 0 1 1 1 1 0 1 0 0 0 0 1 ; 0 1 0 0 0 0 1
1 1 0 0 1 0 0 1 1 0 1 0 0 0 0 ; <1>     <2>     <3>     <4>     1 0 1 1 0 0 0 0 1 1 0 1 0 1 ; 0 1 1 0 1 0 1
0 0 0 1 0 1 1 0 0 1 1 0 0 0 0 ; <1>     <2>     <3>     <4>     1 0 1 1 1 1 1 1 1 0 1 0 1 1 0 ; 1 0 1 0 1 1 0
1 0 0 1 0 1 1 0 0 1 1 0 0 0 0 ; <1>     <2>     <3>     <4>     1 0 0 0 0 0 0 0 1 0 0 1 1 1 0 ; 1 0 0 1 1 1 0
# Final State : <1>     <2>     <3>     <4>     1 0 0 0 0 0 0 0 1 0 0 1 1 1 0
```

Next, we show how to apply the same input vectors to simulate the circuit with $Verilog - XL^{TM}$. The following is a Verilog wrapper (`two_phase.wrapper`) used to pick up the input vectors, apply them to the example circuit, and dump the simulation result. Note that the values of `<T>000001`, `<T>000002`, `<T>000003`, `<T>000004` are constants `<1>`, `<2>`, `<3>`, `<4>`, respectively. Remember that `<T>000001` is for the first process. `<1>` corresponds to the only event guard in that process (`@(posedge clk)`). On positive edge of `clk`, *timing machine* transits from `<1>` to `<1>`. Therefore, the value of `<T>000001` stays constant at `<1>`. `<T>000002`, `<T>000003`, `<T>000004` are constant for the similar reason.

Special attention should be paid to the timing in the wrapper program about

- when to apply the input,
- when to let simulate time pass by (to let the simulator evaluate the example circuit),
- and when to dump the output.

```
// You got to change the following parameters
module two_phase_t ();
parameter numCycles = 14 ;

parameter numInputs =5;
parameter maxSigLen =4;
parameter MEMFILE = "two_phase.vect.wrapper" ;
// You don´t have to change this parameter
parameter numLines = numInputs * numCycles;
parameter clkHalfPeriod = 50;

integer fg;
integer i;

reg [maxSigLen - 1 :0] CORE [0:numLines - 1];

reg clk;
reg [3 : 0] in1;
reg [3 : 0] in2;
reg [1 : 0] in3;
reg [1 : 0] in4;
wire [1 : 0] out1;
wire out2;
wire [1 : 0] out3;
wire [1 : 0] out4;

// instantiate the example circuit.
two_phase itwo_phase(
.clk (clk),
.in1 (in1),
.in2 (in2),
.in3 (in3),
.in4 (in4),
.out1 (out1),
.out2 (out2),
.out3 (out3),
.out4 (out4));
```

```
initial
begin
  i = 0;
  fg = $fopen ("two_phase.vlog");
  $readmemh(MEMFILE,CORE);


 repeat (numCycles)
 begin
clk = CORE [ 0 * numCycles + i] ;
in1 = CORE [ 1 * numCycles + i] ;
in2 = CORE [ 2 * numCycles + i] ;
in3 = CORE [ 3 * numCycles + i] ;
in4 = CORE [ 4 * numCycles + i] ;

// Note: we dump output vectors here using the following wires
// since the circuit is NOT evaluated yet even though the input
// vector has been applied.  This is necessary to match VIS and Verilog-XL
// outputs.
// Remember that, in VIS, after inputs are applied and next state variables
// are evaluated, one cannot see the next state values until the next clock
// tick.  However, in Verilog-XL, after the rising/falling edge of clk occurs,
// the guarded statements are immediately executed and register variables
// are immediately updated.  That is, in a Verilog-XL wrapper, if we allow
// the circuit to be evaluated and then dump the value of registers, we
// see NEXT STATE values instead of CURRENT STATE values (which is what we
// see in VIS).

$fwrite(fg,
 " %h", clk,
 " %h", in1,
 " %h", in2,
 " %h", in3,
 " %h", in4,
 " %h", out1,
 " %h", out2,
 " %h", out3,
 " %h", out4,
"\n"
);
#10;  // give circuit time to evaluate
#clkHalfPeriod  i = i + 1;

  end // repeat
end // initial
endmodule
```

Verilog-XL input vector (`two_phase.vect.wrapper`, converted from `two_phase.vect`). The first 14 entries are for `clk`, then the next 14 for `in1`, the 14 follow are for `in2`, and so forth. The vector included listed in the document is beautified. In the real vector file, there should be only one number per line.

```
0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 0 0 0 0 0 8 8 0 0 9 9 4 4
0 0 0 0 0 0 0 0 a a c c 3 3
0 0 0 0 0 0 1 1 2 2 2 2 3 3
0 0 0 0 0 0 0 0 2 2 0 0 0 0
```

We can then invoke Verilog-XL using the command line "`verilog -q +noxl two_phase.wrapper two_phase.v`". $Verilog - XL^{TM}$ simulation output follows. The output columns, from left to right, are: `clk`, `in1`, `in2`, `in3`, `in4`, `out1`, `out2`, `out3`, `out4`.

```
0 0 0 0 0 0 0 0 x
1 0 0 0 0 0 0 0 x
0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0
0 8 0 1 0 0 0 0 0
1 8 0 1 0 0 0 1 0
0 0 a 2 2 0 0 1 0
1 0 a 2 2 0 0 0 0
0 9 c 2 0 2 0 0 2
1 9 c 2 0 2 1 2 2
0 4 3 3 0 1 1 2 1
1 4 3 3 0 1 0 3 1
```

Note the nondeterministic initial state of `out4` (which is `2´b11`) in VIS simulation trace and `x`'s (`2´bxx´`) in Verilog-XL trace. By ignoring the first two rows (initialization), we find the simulation trace for $Verilog - XL^{TM}$ matches that of VIS.

One should be careful in using "explicit clocking" in `vl2mv`. Since `vl2mv` converts each `@` construct into a bunch of edge-detectors, the extracted blif-mv file may contain a large number of state variables. Any questions or problems encountered should be addressed to `vis@ic.eecs.berkeley.edu`.

Appendix: `two_phase.mv`

We show the entire extrated blif-mv file for `two_phase.v`.

```
# vl2mv -y -c -F -Z two_phase.v
# version: 0.2
# date:    14:29:49 10/14/96 (PDT)
.model two_phase
# I/O ports
.inputs in1<0> in1<1> in1<2> in1<3>
.inputs in2<0> in2<1> in2<2> in2<3>
.inputs in3<0> in3<1>
.inputs in4<0> in4<1>
.inputs clk
.outputs out4<0> out4<1>
.outputs out3<0> out3<1>
.outputs out2
.outputs out1<0> out1<1>

.mv <$lc_ps$>000001, <$lc_ns$>000001 4 <0> <1> <1><PT> <>
.mv <$lc_ps$>000002, <$lc_ns$>000002 4 <0> <2> <2><PT> <>
.mv <$lc_ps$>000003, <$lc_ns$>000003 4 <0> <3> <3><PT> <>
.mv <$lc_ps$>000004, <$lc_ns$>000004 4 <0> <4> <4><PT> <>
# out1  = 0
.names out1$raw_n0<0>
0
.names out1$raw_n0<1>
0
# non-blocking assignments for initial
# out2  = 0
.names out2$raw_n1
0
# non-blocking assignments for initial
# out3  = 0
.names out3$raw_n2<0>
0
.names out3$raw_n2<1>
0
# non-blocking assignments for initial
```

```
# out1  = in1  + in2
# in1  + in2
.names _n9
0
.names in1<0> in2<0> _n9 _n8<0>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names _nb
0
.names in1<0> in2<0> _nb _na
.def 0
0 1 1 1
1 0 1 1
1 1 0 1
1 1 1 1
.names in1<1> in2<1> _na _n8<1>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names in1<1> in2<1> _na _nc
.def 0
0 1 1 1
1 0 1 1
1 1 0 1
1 1 1 1
.names in1<2> in2<2> _nc _n8<2>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names in1<2> in2<2> _nc _nd
.def 0
0 1 1 1
1 0 1 1
1 1 0 1
1 1 1 1
.names in1<3> in2<3> _nd _n8<3>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
.names _n8<0> out1$raw_n7<0>
- =_n8<0>
.names _n8<1> out1$raw_n7<1>
- =_n8<1>
.names <$1c_ps$>000001 _ne
.def 0
<0> 1
# feedback mux
.names out4<0> _n3<0>
- =out4<0>
```

```
.names out4<1> _n3<1>
- =out4<1>
.names out3<0> _n4<0>
- =out3<0>
.names out3<1> _n4<1>
- =out3<1>
.names out2 _n5
- =out2
.names out1<0> out1$raw_n7<0> _ne _n6<0>
0 - 1 0
1 - 1 1
- 0 0 0
- 1 0 1
.names out1<1> out1$raw_n7<1> _ne _n6<1>
0 - 1 0
1 - 1 1
- 0 0 0
- 1 0 1
# out2  = in1 [0] ^ in1 [1]
# in1 [0] ^ in1 [1]
.names in1<0> in1<1> _n16<0>
.def 0
0 1 1
1 0 1
.names _n16<0> out2$raw_n15
- =_n16<0>
.names <$1c_ps$>000002 _n17
.def 0
<0> 1
# feedback mux
.names out4<0> _n11<0>
- =out4<0>
.names out4<1> _n11<1>
- =out4<1>
.names out3<0> _n12<0>
- =out3<0>
.names out3<1> _n12<1>
- =out3<1>
.names out2 out2$raw_n15 _n17 _n13
0 - 1 0
1 - 1 1
- 0 0 0
- 1 0 1
.names out1<0> _n14<0>
- =out1<0>
.names out1<1> _n14<1>
- =out1<1>
# out3  = in3  - in4
# in3  - in4
.names _n1f
0
.names in3<0> in4<0> _n1f _n1e<0>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names _n21
0
.names in3<0> in4<0> _n21 _n20
```

```
.def 0
0 0 1 1
0 1 0 1
0 1 1 1
1 1 1 1
.names in3<1> in4<1> _n20 _n1e<1>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
.names _n1e<0> out3$raw_n1d<0>
- =_n1e<0>
.names _n1e<1> out3$raw_n1d<1>
- =_n1e<1>
.names <$1c_ps$>000003 _n22
.def 0
<0> 1
# feedback mux
.names out4<0> _n19<0>
- =out4<0>
.names out4<1> _n19<1>
- =out4<1>
.names out3<0> out3$raw_n1d<0> _n22 _n1a<0>
0 - 1 0
1 - 1 1
- 0 0 0
- 1 0 1
.names out3<1> out3$raw_n1d<1> _n22 _n1a<1>
0 - 1 0
1 - 1 1
- 0 0 0
- 1 0 1
.names out2 _n1b
- =out2
.names out1<0> _n1c<0>
- =out1<0>
.names out1<1> _n1c<1>
- =out1<1>
.names clk _n29
- =clk
# out4  = in1  - in2
# in1  - in2
.names _n2c
0
.names in1<0> in2<0> _n2c _n2b<0>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names _n2e
0
.names in1<0> in2<0> _n2e _n2d
.def 0
0 0 1 1
0 1 0 1
0 1 1 1
1 1 1 1
.names in1<1> in2<1> _n2d _n2b<1>
```

```
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names in1<1> in2<1> _n2d _n2f
.def 0
0 0 1 1
0 1 0 1
0 1 1 1
1 1 1 1
.names in1<2> in2<2> _n2f _n2b<2>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names in1<2> in2<2> _n2f _n30
.def 0
0 0 1 1
0 1 0 1
0 1 1 1
1 1 1 1
.names in1<3> in2<3> _n30 _n2b<3>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
.names _n2b<0> out4$clk_n2a$true<0>
- =_n2b<0>
.names _n2b<1> out4$clk_n2a$true<1>
- =_n2b<1>
# if/else (clk )
.names out4$clk_n2a$true<0> out4<0> clk out4$clk$raw_n33<0>
0 - 1 0
1 - 1 1
- 0 0 0
- 1 0 1
.names out4$clk_n2a$true<1> out4<1> clk out4$clk$raw_n33<1>
0 - 1 0
1 - 1 1
- 0 0 0
- 1 0 1
.names <$lc_ps$>000004 _n36
.def 0
<0> 1
# feedback mux
.names out4<0> out4$clk$raw_n33<0> _n36 _n25<0>
0 - 1 0
1 - 1 1
- 0 0 0
- 1 0 1
.names out4<1> out4$clk$raw_n33<1> _n36 _n25<1>
0 - 1 0
1 - 1 1
- 0 0 0
- 1 0 1
.names out3<0> _n26<0>
```

```
- =out3<0>
.names out3<1> _n26<1>
- =out3<1>
.names out2 _n27
- =out2
.names out1<0> _n28<0>
- =out1<0>
.names out1<1> _n28<1>
- =out1<1>
# conflict arbitrators
.names <$lc_ps$>000004 _n29 _n39
.def 0
(<4>) 1 1
.names _n39 out4$clk$raw_n33<0> out4$clk$raw_n33<1> out4<0> out4<1> -> _n3a<0> _n3a<1>
1 - - - - =out4$clk$raw_n33<0> =out4$clk$raw_n33<1>
0 - - - - =out4<0> =out4<1>
.names <$lc_ps$>000003 _n3b
.def 0
(<3>) 1
.names _n3b out3$raw_n1d<0> out3$raw_n1d<1> out3<0> out3<1> -> _n3c<0> _n3c<1>
1 - - - - =out3$raw_n1d<0> =out3$raw_n1d<1>
0 - - - - =out3<0> =out3<1>
.names <$lc_ps$>000002 _n3d
.def 0
(<2>) 1
.names _n3d out2$raw_n15 out2 _n3e
1 0 - 0
1 1 - 1
0 - 0 0
0 - 1 1
.names <$lc_ps$>000001 _n3f
.def 0
(<1>) 1
.names _n3f out1$raw_n7<0> out1$raw_n7<1> out1<0> out1<1> -> _n40<0> _n40<1>
1 - - - - =out1$raw_n7<0> =out1$raw_n7<1>
0 - - - - =out1<0> =out1<1>
# non-blocking assignments
# latches
.r out4<0>
-
.r out4<1>
-
.latch _n3a<0> out4<0>
.latch _n3a<1> out4<1>
.r out3$raw_n2<0> out3<0>
.def 0
1 1
.r out3$raw_n2<1> out3<1>
.def 0
1 1
.latch _n3c<0> out3<0>
.latch _n3c<1> out3<1>
.r out2$raw_n1 out2
0 0
1 1
.latch _n3e out2
.r out1$raw_n0<0> out1<0>
.def 0
1 1
.r out1$raw_n0<1> out1<1>
.def 0
```

```
1 1
.latch _n40<0> out1<0>
.latch _n40<1> out1<1>
# quasi-continuous assignment
# timing automaton: time flies
.latch clk clk$prev0
.r clk$prev0
0
.names clk$prev0 clk _n41
.def 0
0 1 1
.mv <T>000001 3 <0> <1> <1><PT>
.mv <_>000001 3 <0> <1> <1><PT>
.latch <_>000001 <T>000001
.r <T>000001
<1>
.names _n41 <T>000001 -> <$lc_ps$>000001 <$lc_ns$>000001 <_>000001
.def <> <> =<T>000001
0  <1> <> <> =<T>000001
1  <1> <1> <1> <1>
- <0> <0> <1> <1>
# timing automatan: time flies
.latch clk clk$prev1
.r clk$prev1
0
.names clk$prev1 clk _n42
.def 0
1 0 1
.mv <T>000002 3 <0> <2> <2><PT>
.mv <_>000002 3 <0> <2> <2><PT>
.latch <_>000002 <T>000002
.r <T>000002
<2>
.names _n42 <T>000002 -> <$lc_ps$>000002 <$lc_ns$>000002 <_>000002
.def <> <> =<T>000002
0  <2> <> <> =<T>000002
1  <2> <2> <2> <2>
- <0> <0> <2> <2>
# timing automaton: time flies
.latch clk clk$prev2
.r clk$prev2
0
.names clk$prev2 clk _n43
.def 0
1 0 1
.mv <T>000003 3 <0> <3> <3><PT>
.mv <_>000003 3 <0> <3> <3><PT>
.latch <_>000003 <T>000003
.r <T>000003
<3>
.names _n43 <T>000003 -> <$lc_ps$>000003 <$lc_ns$>000003 <_>000003
.def <> <> =<T>000003
0  <3> <> <> =<T>000003
1  <3> <3> <3> <3>
- <0> <0> <3> <3>
# timing automatan: time flies
.latch clk clk$prev3
.r clk$prev3
0
.names clk$prev3 clk _n45
.def 0
```

```
1 0 1
0 1 1
# in1  - in2
.names _n47
0
.names in1<0> in2<0> _n47 _n46<0>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names _n49
0
.names in1<0> in2<0> _n49 _n48
.def 0
0 0 1 1
0 1 0 1
0 1 1 1
1 1 1 1
.names in1<1> in2<1> _n48 _n46<1>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names in1<1> in2<1> _n48 _n4a
.def 0
0 0 1 1
0 1 0 1
0 1 1 1
1 1 1 1
.names in1<2> in2<2> _n4a _n46<2>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
# carry/borrow
.names in1<2> in2<2> _n4a _n4b
.def 0
0 0 1 1
0 1 0 1
0 1 1 1
1 1 1 1
.names in1<3> in2<3> _n4b _n46<3>
.def 0
0 0 1 1
0 1 0 1
1 0 0 1
1 1 1 1
.latch _n46<0> _n46$prev4<0>
.r _n46$prev4<0>
0
.names _n46$prev4<0> _n46<0> _n4c<0>
.def 0
1 0 1
0 1 1
.latch _n46<1> _n46$prev4<1>
.r _n46$prev4<1>
```

```
0
.names _n46$prev4<1> _n46<1> _n4c<1>
.def 0
1 0 1
0 1 1
.latch _n46<2> _n46$prev4<2>
.r _n46$prev4<2>
0
.names _n46$prev4<2> _n46<2> _n4c<2>
.def 0
1 0 1
0 1 1
.latch _n46<3> _n46$prev4<3>
.r _n46$prev4<3>
0
.names _n46$prev4<3> _n46<3> _n4c<3>
.def 0
1 0 1
0 1 1
.names _n4c<0> _n4c<1> _n4c<2> _n4c<3> _n4d
.def 1
0 0 0 0 0
.names _n45 _n4d _n44
.def 1
0 0 0
.mv <T>000004 3 <0> <4> <4><PT>
.mv <_>000004 3 <0> <4> <4><PT>
.latch <_>000004 <T>000004
.r <T>000004
<4>
.names _n44 <T>000004 -> <$lc_ps$>000004 <$lc_ns$>000004 <_>000004
.def <> <> =<T>000004
0  <4> <> <> =<T>000004
1  <4> <4> <4> <4>
- <0> <0> <4> <4>
.end
```