Synthesis of Combinational and Sequential Circuits with Verilog

Logic Synthesis

• Verilog is used
  – Model hardware for discrete-event simulation
  – Input to logic synthesis.
  – Construct testbenches
• Only a subset of Verilog constructs can be synthesised into an efficient circuit
• Exactly what can synthesize depends on tools and technologies

• Logic synthesis involves
  – Translating Verilog source to a netlist
  – Optimization for speed and circuit size
    • Detect and eliminate redundant logic
    • Detect combinatorial feedback loops
    • Exploit don’t cares
    • Collapse equivalent states
    • Optimize for a particular technology
• Some constructs easy to synthesize
  – Structural descriptions map to netlist
  – Assign statements to combinational logic
  – Case statement to multiplexers
• Behavioral (always @) blocks
  – Synthesis to combinational and sequential logic
  – Depends on sensitivity list
    • No sensitivity to both edges of the clock, changing sensitivity lists, etc.

• Others have no clear relation to hardware
  – Initial statements (for testbenches or simulation)
  – Delays
  – Other simulation oriented features not intended for synthesis

• Verilog is not like a typical programming language
  – Many things will be parallel rather than sequential
  – For loops may or may not synthesis (need to be static)
  – Consider what code might synthesis to (e.g. if else to a multiplexer)
  – Many different Verilog constructs to describe the same circuit.
always @ block

• Combinational logic
  – If sensitivity list contains all signals used in the equations within block then combinational logic synthesized
  – Variables modified should be defined on all paths through block
    • If not a latch is implied
  – Assignments should use =

always @ block

always @ (A or B) begin
  if (A == 2'b00) out = B;
end
– If A does not equal 2'b00 then Verilog will assume that out maintains its value -> unintended latch is synthesized
– Need to specify output for all paths (else, default)
– Put default behavior right at top of always.

always @ block

• Sequential logic
  – Sensitivity list contains clock edge
  – Reset optional.
• Output changes synchronised to clock edge
  – Use <= assignments
Combinational Logic

- Synthesizable combinational logic can be described by
  - A netlist of structural primitives or combinational modules
  - Continuous assignment statements
  - Level sensitive cyclic behaviours (assign @)

Structural models
- Instantiate primitives and other modules and connect them with wires
- Interconnect objects to create a larger structure with composite behaviour
- Can next model instantiations to create a top-own hierarchy
  - High level blocks consisting of major components
  - Decomposed into smaller functional units
  - Top level module for overall system

Module Add_half(sum, c_out,a,b);
  input  a,b;
  output c_out, sum;
  xor (sum,a,b);
  and (c_out,a,b);
endmodule
(Figure 4-1, Ciletti)
• Modules instantiated in another module must be named and have scope local to the modules
• Primitives predefined by truth tables; can also defined by user based on truth table
• Modules as structural objects
  – Interface with other modules through *ports*
  – Can be nested with arbitrary depth
  – Wires define connectivity

4-bit Comparator (Ciletti ex 4.5)
• Composed of two two-bit comparators
  – \( A_{\text{lt}} B = A_1'B_1+A_1'A_0'B_0 +A_0'B_1B_0 \)
  – \( A_{\text{gt}} B = A_1B_1'+A_0B_1'B_0' +A_1'A_0B_0' \)
  – \( A_{\text{eq}} B = A_1'A_0'B_1'B_0' + A_1'A_0B_1'B_0 + A_1A0'B1B0 + A1A0B1B0 \)
module compare_2_str (A_gt_B, A_lt_B, A_eq_B, A0, A1, B0, B1);
output A_gt_B, A_lt_B, A_eq_B;
input A0, A1, B0, B1;
nor (A_gt_B, A_lt_B, A_eq_B);
or (A_lt_B, w1, w2, w3);
and (A_eq_B, w4, w5);
and (w1, w6, B1);
and (w2, w6, w7, B0);
and (w3, w7, B0, B1);
not (w6, A1);
not (w7, A0);
xnor (w4, A1, B1);
xnor (w5, A0, B0);
endmodule

module Comp_4_str (A_gt_B, A_lt_B, A_eq_B, A3, A2, A1, A0, B3, B2, B1, B0);
output A_gt_B, A_lt_B, A_eq_B;
input A3, A2, A1, A0, B3, B2, B1, B0;
wire w1, w0;
Comp_2_str M1 (A_gt_B_M1, A_lt_B_M1, A_eq_B_M1, A3, A2, B3, B2);
Comp_2_str M0 (A_gt_B_M0, A_lt_B_M0, A_eq_B_M0, A1, A0, B1, B0);
or (A_gt_B, A_gt_B_M1, w1);
and (w1, A_eq_B_M1, A_gt_B_M0);
and (A_eq_B, A_eq_B_M1, A_eq_B_M0);
or (A_lt_B, A_lt_B_M1, w0);
and w0, A_eq_B_M1, A_lt_B_M0);
endmodule
Behavioural models

- Behavioural modeling using procedural assignments to describe the input output behaviour without specifying structure
  - Generally less structured, algorithmic
  - Register transfer level (RTL) operations specify logic with *implicit* binding to hardware (datapath structure)
  - RTL is most common level for design synthesis today -> increasingly more algorithmic or high-level behavioral synthesis

- For combinational logic, behavioural descriptions take the form of Boolean equations
  - Verilog equivalent is continuous assignment statement (e.g. Ciletti example 5.1, 5.13)
  - Assigns net (wire) to output of multilevel comb. circuit

```
module compare_2_CA0 (A_lt_B, A_gt_B, A_eq_B, A1, A0, B1, B0);
  input A1, A0, B1, B0;
  output A_lt_B, A_gt_B, A_eq_B;
  assign A_lt_B = (~A1) & B1 | (~A1) & (~A0) & B0 | (~A0) & B1 & B0;
  assign A_gt_B = A1 & (~B1) | A0 & (~B1) & (~B0) | A1 & A0 & (~B0);
  assign A_eq_B = (~A1) & (~A0) & (~B1) & (~B0) | (~A1) & A0 & (~B1) & B0 | A1 & A0 & B1 & B0 | A1 & (~A0) & B1 & (~B0);
endmodule

module compare_2_CA1 (A_lt_B, A_gt_B, A_eq_B, A, B);
  input [1: 0] A, B;
  output A_lt_B, A_gt_B, A_eq_B;
  assign A_lt_B = (A < B);
  assign A_gt_B = (A > B);
  assign A_eq_B = (A == B);
endmodule
```
module Mux_2_32_CA (mux_out, data_1, data_0, select);
parameter word_size = 32;
output [word_size -1: 0] mux_out;
input [word_size-1: 0] data_1, data_0;
input select;
assign mux_out = select ? data_1 : data_0;
endmodule

• ? synthesizes as multiplexer here

• Continuous assignment statements within a module are active concurrently
  – What if result of an assign is on RHS of another?
• For modeling and functional simulation
  – the simulator assumes an ‘inertial delay’;
    assignments take some time to change
  – Allows functional simulation to match timing simulations

• Level sensitive cyclic behaviour
  – Will synthesis to combinational logic if there is an output for every possible input combination
    (to inputs in sensitivity list)
    • Sensitivity list must be sensitive to every input
    • Every path through behaviour must assign value to every output
  – Defines functionality in device independent way but doesn’t define realization, timing
module compare_2_RTL (A_lt_B, A_gt_B, A_eq_B, A1, A0, B1, B0);
input A1, A0, B1, B0;
output A_lt_B, A_gt_B, A_eq_B;
reg A_lt_B, A_gt_B, A_eq_B;
always @ (A0 or A1 or B0 or B1) begin
    A_lt_B = ({A1,A0} < {B1,B0});
    A_gt_B = ({A1,A0} > {B1,B0});
    A_eq_B = ({A1,A0} == {B1,B0});
end
endmodule (Ciletti 5.15)

module mux_4pri (y, a, b, c, d, sel_a, sel_b, sel_c);
output y;
input a, b, c, d, sel_a, sel_b, sel_c;
reg y;
always @ (sel_a or sel_b or sel_c or a or b or c or d) begin
    if (sel_a == 1) y = a; else
    if (sel_b == 0) y = b; else
    if (sel_c == 1) y = c; else
        y = d;
end
endmodule

Ex. Ciletti 6.10

• If the assign statement or level sensitive always @ has feedback a latch is synthesized
  – assign Q = s ? D:Q;
• Feedback free netlist of combinational logic will form latch free combinatorial logic
  – REG variables do not always mean flip-flops
  – Accidental synthesis of latches can occur in procedural combinational logic blocks
– If an output is not specified for some combination of inputs in the sensitivity list
  • Verilog assumes current values of variables set in procedural assignments is maintained
  • Implies memory -> latch synthesized
  • Likewise all RHS operands of procedural assignments should be in sensitivity list
  • Also RHS operand of procedural assignment must not be on LHS of the expression (explicit feedback)
  • Take advantage of blocking assignments - put default assignments first in combinatorial assign blocks

Synthesis of latches

• If one wants a latch in a level sensitive behaviour
  – Use explicit feedback of output of assign to feedback in an assign, case, if or ?
  – Implicitly use feedback by not specifying cases where latched variable does not change

Tri-state combinational logic

```verilog
module Uni_dir_bus (data_to_bus, bus_enable);
  input bus_enable;
  output [31: 0] data_to_bus;
  reg [31: 0] ckt_to_bus;

  assign data_to_bus = (bus_enabled) ? ckt_to_bus : 32'bz;

  // Description of core circuit goes here to drive ckt_to_bus
endmodule
```
Flip-Flops and Registers

- Verilog uses cyclic behaviour (always statements) to model edge sensitive sequential circuits
  - Use procedural statements
  - Called cyclic behaviours since after completion they execute again
  - Can model both level sensitive (latch) and edge-sensitive (flip-flop) memory

- Sensitivity list of an always block can be sensitive to clock edges (posedge or negedge)
- Clock, clk etc are not keywords and need to be defined and assigned. Synthesis tool must infer the synchronising signal and whether a flip flop is required.

```vhdl
module my_dff (q, qbar, d, clk, clr);
  input d, clk, clr;
  output q, qbar;
  assign qbar = ~q;
  always @ (posedge clk) begin
    if (clr == 0) q <= 0;
    else q <= d;
  end
endmodule
```
// asynchronous clear
module my_dff (q, qbar,d, clk, clr);
  input d, clk, clr;
  output q, qbar;
  assign qbar = ~q;
  always @ (posedge clk or negedge clr)
  begin
    if (clr == 0) q <= 0;
    else q <= d;
  end
endmodule

• Registers can be modeled as vectored flip-flops (Ciletti 5.17)

module shiftreg_nb (A, E, clk, rst);
  output A;
  input E;
  input clk, rst;
  reg A, B, C, D;
  always @ (posedge clk or posedge rst)
  begin
    if (rst) begin
      A <= 0; B <= 0; C <= 0; D <= 0;
    end
    else begin
      A <= B; // D <= E;
      B <= C; // C <= D;
      C <= D; // B <= D;
      D <= E; // A <= B;
    end
  end
endmodule

Register Transfer Level

• Synchronous sequential machine
• At each active clock, registers store data generated as result of previous clock
• Describe operations based dataflow between registers
  – Specific for a particular architecture
  – Combinational part - blocking assignments
  – Sequential part- non-blocking
Counters

- Important RTL component
- Generates sequence of binary words
- Many variants
  - Binary, BCD, gray-code, ...
  - Up, down, up-down, count enable
  - Parallel load

Usually use state machine implementation
- State based on control operations (counting up, down, idle) rather than count itself. Can often be simplified to counting/idle.
- More flexible
- Generalizes to other word lengths
- Count maintained in separate register than the state
- Ciletti Ex. 5.40

```verilog
module Up_Down_Implicit1 (count, up_dwn, clock, reset_);
output [2: 0] count;
input [1: 0] up_dwn;
input clock, reset_;
reg [2: 0] count;

always @(negedge clock or negedge reset_)
  if (reset_ == 0) count <= 3'b0;
  else if (up_dwn == 2'b00 || up_dwn == 2'b11) count <= count;
  else if (up_dwn == 2'b01) count <= count + 1;
  else if (up_dwn == 2'b10) count <= count - 1;
endmodule
```
Shift Registers

```verilog
module Shift_reg4 (Data_out, Data_in, clock, reset);
  output Data_out;
  input Data_in, clock, reset;
  reg [3: 0] Data_reg;
  assign Data_out = Data_reg[0];
  always @ (negedge reset or posedge clock)
  begin
    if (reset == 1'b0) Data_reg <= 4'b0;
    else Data_reg <= {Data_in, Data_reg[3:1]};
  end
endmodule
```

- Universal shift register (Ciletti 5.46)
  - Shift left
  - Shift right
  - Parallel load
  - Parallel out

```verilog
module Universal_Shift_Reg (Data_Out, MSB_Out, LSB_Out, Data_In,
    MSB_In, LSB_In, s1, s0, clk, rst);
  output [3: 0] Data_Out;
  output MSB_Out, LSB_Out;
  input [3: 0] Data_In;
  input MSB_In, LSB_In;
  input s1, s0, clk, rst;
  reg Data_Out;
  assign MSB_Out = Data_Out[3];
  assign LSB_Out = Data_Out[0];
endmodule
```
always @ (posedge clk) begin
  if (rst) Data_Out <= 0;
  else case (s1, s0)
    0: Data_Out <= Data_Out; // Hold
    1: Data_Out <= [MSB_In, Data_Out[3:1]]; // Serial shift from MSB
    2: Data_Out <= [Data_Out[2:0], LSB_In]; // Serial shift from LSB
    3: Data_Out <= Data_In; // Parallel Load
  endcase
end
endmodule

Memories

- Arrays of registers can form register files (Cilleti 5.47)

module Register_File (Data_Out_1, Data_Out_2, Data_in, Read_Addr_1, Read_Addr_2, Write_Addr, Write_Enable, Clock);
output [31: 0] Data_Out_1, Data_Out_2;
input [31: 0] Data_in;
input [4: 0] Read_Addr_1, Read_Addr_2, Write_Addr;
input Write_Enable, Clock;
reg [31: 0] Reg_File [31: 0]; // 32 bit x32 word memory declaration
assign Data_Out_1 = Reg_File[Read_Addr_1];
assign Data_Out_2 = Reg_File[Read_Addr_2];
always @ (posedge Clock) begin
  if (Write_Enable) Reg_File[Write_Addr] <= Data_in;
end
endmodule

State Machine Synthesis

- Explicit state machines have explicit storage of state variables and explicit logic to generate next state
- Typically at least two (or three) always blocks
  - Level sensitive, next state (and output) generator
    - Use = for combinatorial logic
    - Decode all possible states
  - Edge sensitive, synchronous state transitions
    - Use <= for synchronous transitions
• Usually restrictions on state machine implementation for synthesis
  – Can assign a variable in state assignments
  – Must set entire state register at once
  – …
• Examples Brown A.40, A41
• Avoid having one variable assigned in more than one always block (possible races)

Synthesis of Loops
• Loops in an always statement can often be synthesized if ‘static’ or ‘data-independent’
  – Number of iterations can be determined at compile time
  – Static loops can often be synthesized
  – Sometimes only for loops supported

• If for loop does not depend on data and does not have timing controls embedded
  – Synthesises to regular, repeated combination circuit (Ciletti 6.37)
• If static, with embedded event controls sequential logic can be synthesized
• Non-static loops cannot be synthesized
module comparator (a_gt_b, a_lt_b, a_eq_b, a, b);
// Alternative algorithm
parameter size = 2;
output a_gt_b, a_lt_b, a_eq_b;
input [size: 1] a, b;
reg a_gt_b, a_lt_b, a_eq_b;
integer k;

always @(a or b) begin: compare_loop
  for (k = size; k > 0; k = k-1) begin
    if (a[k] != b[k]) begin
      a_gt_b = a[k];
      a_lt_b = ~a[k];
      a_eq_b = 0;
      disable compare_loop;
    end // if
  end // for loop
  a_gt_b = 0;
  a_lt_b = 0;
  a_eq_b = 1;
end // compare_loop
endmodule