Cloud FPGA

EENG 428
ENAS 968

bit.ly/cloudfpga

Lecture: Logic Synthesis

Prof. Jakub Szefer
Dept. of Electrical Engineering, Yale University

EENG 428 / ENAS 968
Cloud FPGA

Logic Synthesis

This lecture is mostly based on contents of Chapter 2, from "The Verilog Hardware Description Language" book [1], 5th edition. Example figures and (modified) code are from the textbook unless otherwise specified.

Topics covered:

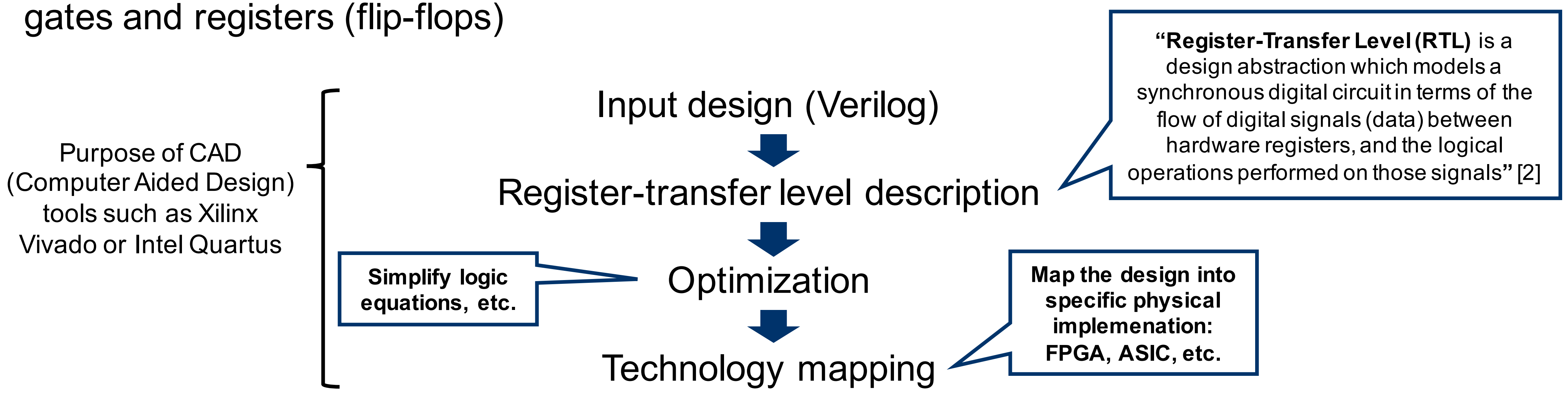
- Specifying combinatorial logic using gates and continuous **`assign`**
- Specifying combinatorial logic using procedural statements
- Inferring sequential elements
- Describing finite state machines
- Finite state machines and datapath

Thomas & Moorby's
The Verilog® Hardware Description Language
Fifth Edition
Extra Materials extras.springer.com

Logic Synthesis, Combinatorial Logic

Synthesis Overview

Goal of logic synthesis is to generate a hardware design that can be realized using logic gates and registers (flip-flops)



Purpose of CAD (Computer Aided Design) tools such as Xilinx Vivado or Intel Quartus

Input design (Verilog)

Register-transfer level description

Optimization

Technology mapping

"Register-Transfer Level (RTL) is a design abstraction which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers, and the logical operations performed on those signals" [2]

Simplify logic equations, etc.

Map the design into specific physical implemenation: FPGA, ASIC, etc.

Synthesizable subset of Verilog

- Constructs which can be mapped to digital logic
- Not all Verilog can be mapped to hardware
  - Recall Verilog was designed for testing, many parts are for simulation and testing

Combinatorial Logic Using Gates

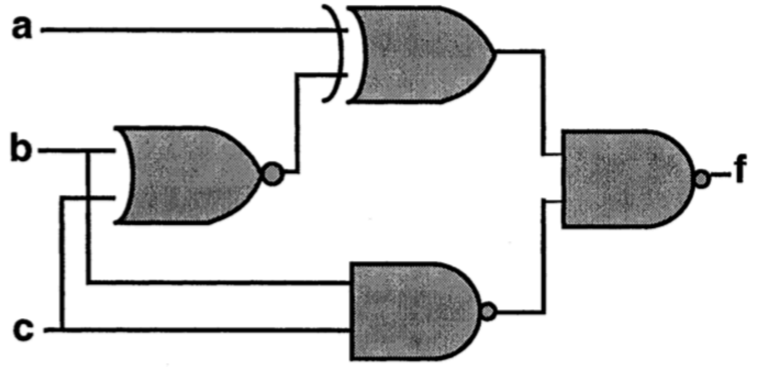
- Combinatorial logic can be explicitly specified using logic gates and interconnections between the gates: a structural specification
- Logic synthesis tool will then interpret the specification, optimize it, and map the design to logic gates

```
module synGate
( output f,
  input a, b, c
);

  wire a1, a2, a3, o1, f;

  and A (a1, a, b, c);
  and B (a2, a, ~b, ~c);
  and C (a3, ~a, o1);
  or D (o1, b, c);
  or E (f, a1, a2, a3);

endmodule
```

No delays are specified, e.g. no `and #5`, delays depend on physical implementation that the design targets

Logic Synthesis

Synthesized design is equivalent to the specification, but need not be identical to the gates used in Verilog



Different technologies have different gates, or even no gates (in FPGA design is mapped to Lookup Tables)

Combinatorial Logic Using `assign`

- Combinatorial logic can be alternatively specified using Boolean algebra-like statements
  - Verilog has more operators so can write simpler expression
  - Can write only using Boolean algebra operators (and, or, not) if desired

```
module synAssign
( output f,
  input a, b, c
);

assign f = (a & b & c) | (a & ~b & ~c) | (~a & (b | c));

endmodule
```

Logical expression for signal f

- **`assign`** can be driven by logical expressions, or using functions
- Functions in Verilog are like mini combinatorial logic modules

```
function  myfunction;
input a, b, c, d;
begin
  myfunction = ((a+b) + (c-d));
end
endfunction

...

assign f =  (myfunction (a,b,c,d)) ? e : 0;
```

Function example from http://www.asic-world.com/verilog/task_func1.html

More Combinatorial Logic Specification Options

- Combinatorial (and sequential) logic can be parametrized to adjust the width of the signals
  - E.g. use different width for different module instances
  - Default width if not overwritten when instantiated

```
addWithAssign #(16) myAdder (cout, s, a, b cin)

addWithAssign #( .WIDTH(16)) myAdder (cout, s, a, b cin)
```

- **`assign`** statements in synthesizable design cannot use don't care values on left-hand side
  - Right-hand side is okay, lets CAD tools optimize the design

There are only physical 1s and 0s, can't compare to don't care x; can't synthesize this

```
assign y = (a===1'bx) ? c : 1;

assign y = (a==b) ? 1'bx : c;
```

Okay to assign don't care value, CAD tools will pick 1 or 0 that optimizes the design best

```
module addWithAssign
#(parameter WIDTH = 4
)
( output carry,
  output [WIDTH-1:0] sum,
  input [WIDTH-1:0] A, B,
  input Cin
);

  assign {carry, sum} = A + B + Cin;

endmodule
```

Use { and } to concatenate signals

```
module muxWithAssign
#(parameter WIDTH = 4
)
( output [WIDTH-1:0] out,
  input [WIDTH-1:0] A, B,
  input sel
);

  assign out = (sel) ? A: B;

endmodule
```

C-like conditional selection

Function example from http://www.asic-world.com/verilog/task_func1.html

Combinatorial Logic Using Procedural Statements

- Combinatorial logic can be specified with **`always`** statements, if certain rules are followed
  - Sensitivity list has to include all the signals on the right-hand side in the always block
  - No **`posedge`** or **`negedge`** in sensitivity list
  - Left-hand side signals need to be assigned a value for all possible input combinations

```
module synCombinationalAlways
( output reg f,
  input a, b, c
);

  always @ (a, b, c)
    if (a == 1)
      f = b;
    else
      f = c;

endmodule
```

Alternate specification with `always @ (*)` and default value for `f`

```
module synAssignOutputFirst
( output reg f,
  input a, b, c
);

  always @ (*)
  begin
    f = c;
    if (a == 1)
      f = b;
  end

endmodule
```

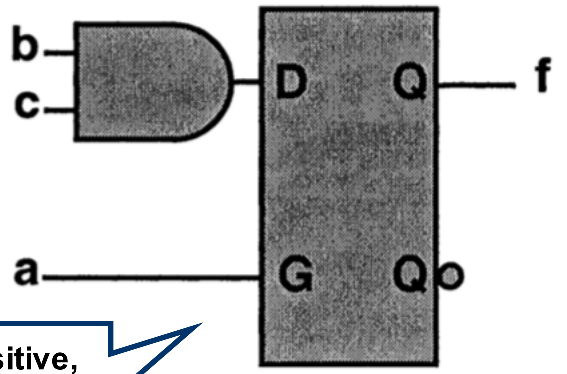
Use `@(*)` or `@*` to automatically derive the sensitivity list

Use "default" value for all left-hand side signals

Avoiding Inferred Latches

- Combinatorial logic's output only depends on the current inputs
  - There is no memory
- If combinatorial logic is specified incorrectly, latches may be inferred
  - Logic will have memory
  - No-longer combinatorial logic

```
module synInferredLatch
( output reg f,
  input a, b, c
);

  always @(*)
    if (a == 1)
      f = b & c;

endmodule
```



Level-sensitive, or gated, latch; controlled by input a

Combinatorial Logic with `case` Statements

- A case statement can be used to specify the output of combinatorial logic in a manner similar to a truth table
  - Can make use of features like default case or don't care values to shorten the description of the logic and synthesize better design

```
module synCase
( output reg f,
  input a, b, c
);

  always @(*)
    case ({a, b, c})
      3'b000: f = 1'b0;
      3'b001: f = 1'b1;
      3'b010: f = 1'b1;
      3'b011: f = 1'b1;
      3'b100: f = 1'b1;
      3'b101: f = 1'b0;
      3'b110: f = 1'b0;
      3'b111: f = 1'b1;
      endcase

endmodule
```

```
module synCaseWithDefault
( output reg f,
  input a, b, c
);

  always @(a, b, c)
    case ({a, b, c})
      3'b000:   f = 1'b0;
      3'b101:   f = 1'b0;
      3'b110:   f = 1'b0;
      default:  f = 1'b1;
    endcase

endmodule
```

```
module synCaseWithDC
( output reg f,
  input a, b, c
);

  always @(*)
    case ({a, b, c})
      3'b001:   f = 1'b1;
      3'b010:   f = 1'b1;
      3'b011:   f = 1'b1;
      3'b100:   f = 1'b1;
      3'b110:   f = 1'b0;
      3'b111:   f = 1'b1;
      default:  f = 1'bx;
    endcase

endmodule
```

Synthesis Attributes

- Verilog code can be augmented with attributes, also called compiler directives
  - Not part of code
  - “Hints” to the compiler
- Example of attributes with **`case`**:

Be careful that attributes or directives can be tool specific, or may behave differently.

```
module synAttributes
( output reg f,
  input a, b, c
);

  always @(*)
  (* full_case, parallel_case *)
    case ({a, b, c})
      3'b001: f = 1'b1;
      3'b010: f = 1'b1;
      3'b011: f = 1'b1;
      3'b100: f = 1'b1;
      3'b110: f = 1'b0;
      3'b111: f = 1'b1;
    endcase

endmodule
```

full_case: hint that unspecified values should be treated as don’t cares, an no latch should be inferred

parallel_case: hint that the case statement has no overlapping cases

- Other examples: **`(* keep = true *)`** to prevent signal from being synthesized away or merged with other logic

```
(* keep = “true” *) wire sig1;
assign sig1 = in1 & in2;
assign out1 = sig1 & in2;
```

`keep` example from Vivado documentation

Combinatorial Logic with `casex` Statements

- The **`casex`** statement, allows for the use of x, z, or ? in the controlling expression or in a case item expressions
  - Can be used to specify don't cares for synthesis
  - For sythesis, x, z, or ? may only be specified in case item expressions

```
module synUsingDC
( output reg f,
  input a, b
);

  always @(*)
    casex ({a, b})
      2'b0?: f = 1;
      2'b10: f = 0;
      2'b11: f = 1;
    endcase

endmodule
```

Combinatorial Logic with Loop Statements

- The **for** loop in Verilog may be used to specify combinational logic
  - The **while** and **forever** loops are used for synthesizing sequential logic
  - The **generate** loops are used to generate Verilog code blocks with well defined patterns

```
module synXor8
( output reg [1:8] xout,
  input [1:8] xin1, xin2
);

  reg [1:8] i;

  always @(*)
    for (i = 1; i <= 8; i = i + 1)
      xout[i] = xin1[i] ^ xin2[i];

endmodule
```

```
module DigitalCorrelator
#(parameter dataWidth  = 40,
            countWidth = 6
)
( output reg [countWidth-1:0]matchCount = 0,
  input [dataWidth-1:0] message, pattern
);

  int i;

  always @(*) begin
    for (i = 0; i < dataWidth; i = i + 1)
      matchCount = matchCount + ~(message[i] ^ pattern[i]);
  end

endmodule
```

Logic Synthesis, Sequential Logic

Sequential Logic Elements

- Sequential elements are: **latches** and **flip-flops**
  - Latches and flip-flops store bits of data
  - Latches are level-sensitive
  - Flip-flops are edge-sensitive

Latch

```
module synLatchReset
( output reg Q,
  input g, d, reset
);

  always @(*)
    if (~reset)
      Q = 0;
    else if (g)
      Q = d;

endmodule
```

This is active-low reset, typically would write name as `reset_n` so it's clear it's active low (n = negative)

Flip-Flop

```
module synDFF
( output reg q,
  input clock, d
);

  always @(negedge clock)
    q <= d;

endmodule
```

Flip-Flop with Reset and Set

```
module synDFFwithSetReset
( output reg q,
  input d, reset, set, clock
);

  always @(posedge clock, negedge reset, posedge set)
  begin
    if (~reset)
      q <= 0;
    else if (set)
      q <= 1;
    else
      q <= d;
  end

endmodule
```

Rules for Sequential Logic Elements

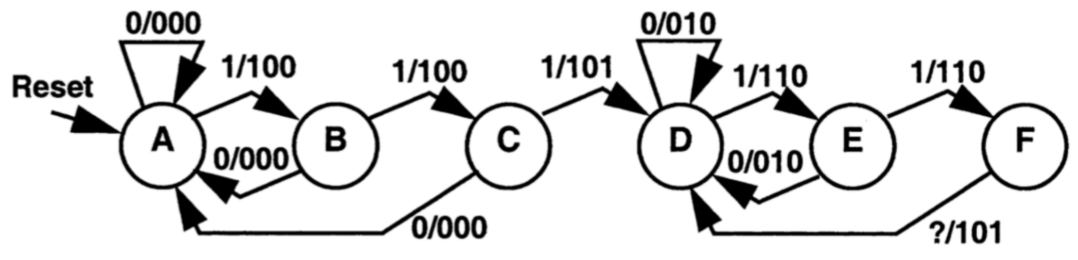
For synthesis of sequential logic, the **`always`** blocks need to follow some rules

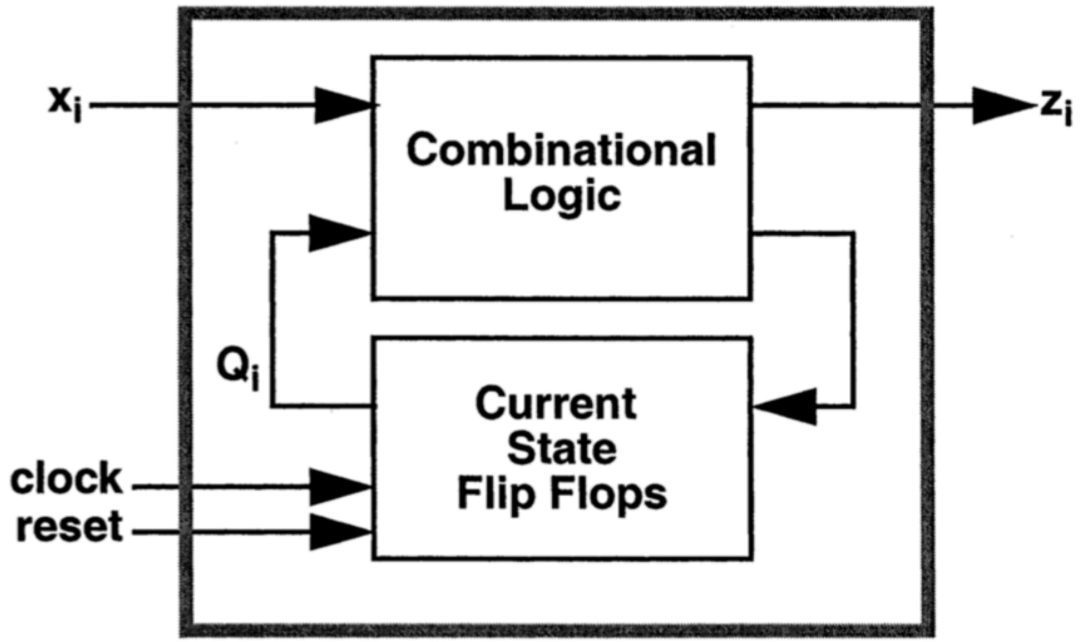
- The sensitivity list must contain only **`posedge`** or **`negedge`** of clock, reset, and set
  - Clock, reset, and set can have any names, clock could be xyz or foo
- Body of always block must be **`if ... else if ... else`**
  - Can skip else if when there is only reset, but no set
- Test reset and set conditions first (asynchronous reset and set)
- If **`posedge`** test for value, if **`negedge`** test for ~value
- Use non-blocking **<=** assignment operator

Describing Finite State Machines

- Finite State Machines (FSM) are made of combinatorial and sequential logic
- Represent behavior of a system as a set of finite states
  - The state is encoded in the flip-flop
  - Alternatively, values of all the flip-flops in a design determine the current state of the FSM

- Behavior of FSM can be specified by a state transition diagram, and later transferred to Verilog





Describing Finite State Machines

- Example FSM in Verilog:

```
module fsm
( input i, clock, reset,
  output reg [2:0] out
);

  reg [2:0] currentState, nextState;

  localparam [2:0] A = 3'b000,
                   B = 3'b001,
                   C = 3'b010,
                   D = 3'b011,
                   E = 3'b100,
                   F = 3'b101;
```

Local parameters, not to be confused with module parameters, can only be defined inside module

Define local parameters to assign names to states of the FSM

Combinatorial logic

```
always @(*)
  case (currentState)
    A: begin
      nextState = (i == 0) ? A : B;
      out = (i == 0) ? 3'b000 : 3'b100;
    end
    B: begin
      nextState = (i == 0) ? A : C;
      out = (i == 0) ? 3'b000 : 3'b100;
    end
    C: begin
      nextState = (i == 0) ? A : D;
      out = (i == 0) ? 3'b000 : 3'b101;
    end
    D: begin
      nextState = (i == 0) ? D : E;
      out = (i == 0) ? 3'b010 : 3'b110;
    end
    E: begin
      nextState = (i == 0) ? D : F;
      out = (i == 0) ? 3'b010 : 3'b110;
    end
    F: begin
      nextState = D;
      out = (i == 0) ? 3'b000 : 3'b101;
    end
    default: begin
      nextState = A;
      out = (i == 0) ? 3'bxxx : 3'bxxx;
    end
  endcase
```

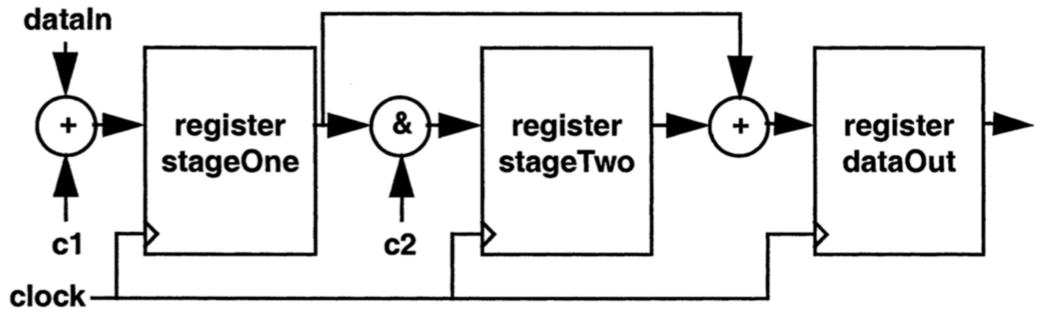
Sequential logic

```
  always @(posedge clock or negedge reset)
    if (~reset)
      currentState <= A;
    else
      currentState <= nextState;

endmodule
```

FSM Description with Multiple `always` Blocks

- Multiple **`always`** blocks can be used in parallel to describe different parts of the system

- Textbook example of a pipeline with three registers
  - Each register is controlled by separate **`always`** statement
  - All **`always`** statements are working in parallel



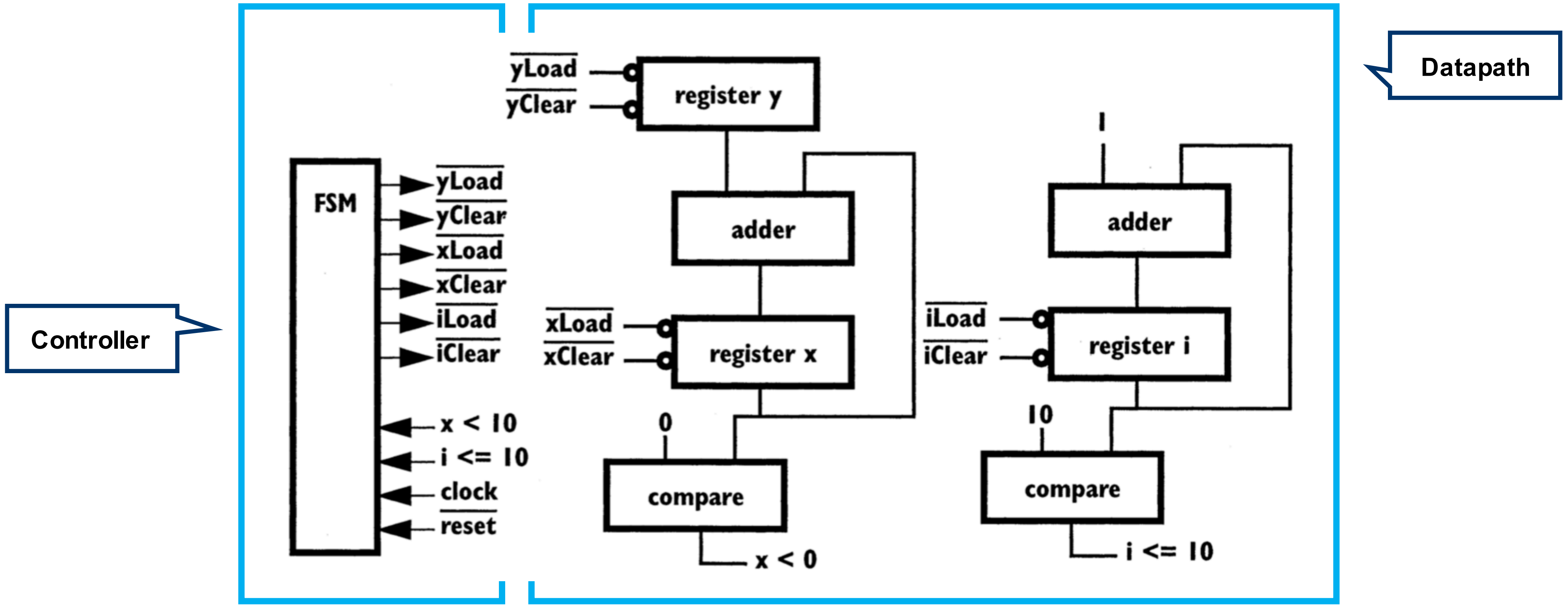
```
module synPipe
( input [7:0] dataIn, c1, c2,
  input clock,
  output reg [7:0] dataOut
);

  reg [7:0] stageOne;
  reg [7:0] stageTwo;

  always @ (posedge clock)
    stageOne <= dataIn + c1;

  always @ (posedge clock)
    stageTwo <= stageOne & c2;

  always @ (posedge clock)
    dataOut <= stageTwo + stageOne;

endmodule
```

Datapath and Controller FSM

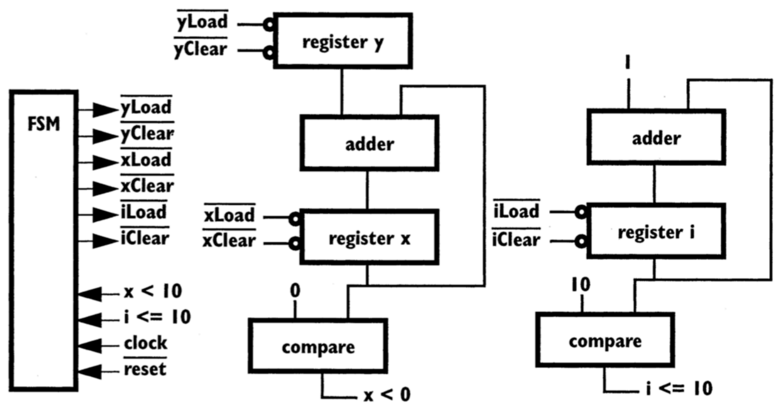
Computational task can be often broken down into datapath and controller FSM

- **Datapath** is the logic that performs the computation
- **Controller FSM** controls the steps of the computation

Many CPUs or microcontrollers are designed in this way: a computation pipeline and controoler



Datapath and Controller FSM



- The example silly computation has four datapath components

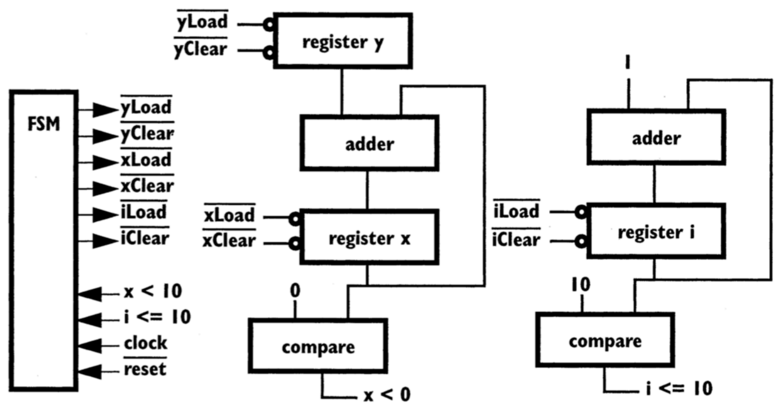
```
module adder
#(parameter Width = 8
)
( input [Width-1:0] a, b,
  output [Width-1:0] sum
);

  assign sum = a + b;

endmodule
```

```
module register
#(parameter Width = 8
)
( output reg [Width-1:0] out,
  input [Width-1:0] in,
  input clear, load, clock
);

  always @(posedge clock)
    if (~clear)
      out <= 0;
    else if (~load)
      out <= in;

endmodule
```

```
module compareLT
#(parameter Width = 8
)
( input [Width-1:0] a, b,
  output out
);

  assign out = a < b;

endmodule
```

```
module compareLEQ
#(parameter Width = 8
)
( input [Width-1:0] a, b,
  output out
);

  assign out = a <= b;

endmodule
```

Datapath and Controller FSM

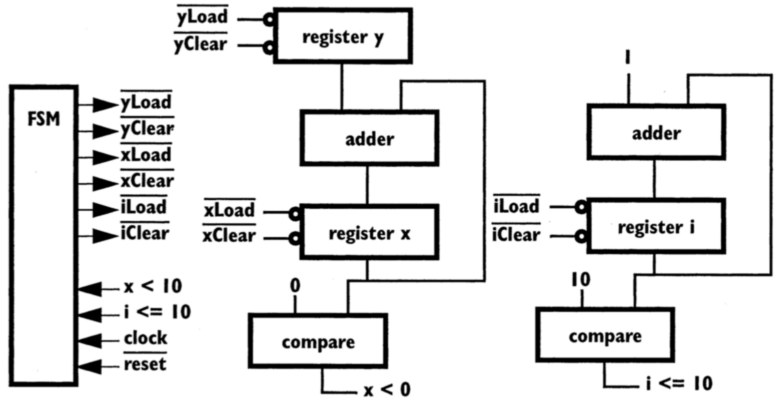


- The example silly computation has one FSM

```
module fsm
( input LT, LEQ, ck, reset,
  output reg yLoad, yClear, xLoad, xClear, iLoad, iClear
);

  reg [2:0] cState, nState;

  always @(posedge ck, negedge reset)
    if (~reset)
      cState <= 0;
    else cState <= nState;

  always @(cState, LT, LEQ)
    case (cState)
      3'b000 : begin // state A
        yLoad = 1; yClear = 1; xLoad = 1; xClear = 0;
        iLoad = 1; iClear = 0; nState = 3'b001;
      end
      3'b001 : begin // state B
        yLoad = 1; yClear = 1; xLoad = 0; xClear = 1;
        iLoad = 0; iClear = 1; nState = 3'b010;
      end
      3'b010 : begin // state C
        yLoad = 1; yClear = 1; xLoad = 1; xClear = 1;
        iLoad = 1; iClear = 1;
        if (LEQ) nState = 3'b001;
        if (~LEQ & LT) nState = 3'b011;
        if (~LEQ & ~LT) nState = 3'b100;
      end
      3'b011 : begin // state D
        yLoad = 1; yClear = 0; xLoad = 1; xClear = 1;
        iLoad = 1; iClear = 1; nState = 3'b101;
      end
      3'b100 : begin // state E
        yLoad = 1; yClear = 1; xLoad = 1; xClear = 0;
        iLoad = 1; iClear = 1; nState = 3'b101;
      end
      default : begin
        yLoad = 1; yClear = 1; xLoad = 1; xClear = 1;
        iLoad = 1; iClear = 1; nState = 3'b000;
        $display ("Oops, unknown state: %b", cState);
      end
    endcase

endmodule
```

$display (print) statement will only show up in simulation

Datapath and Controller FSM



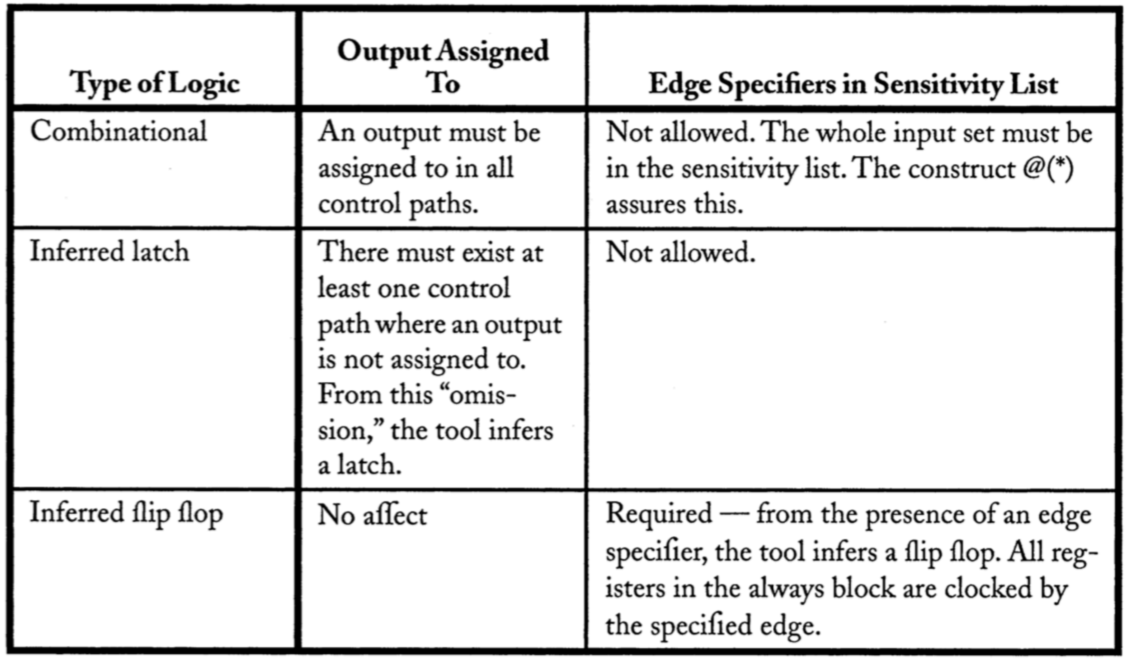
- The example silly computation has one top-level module to combine datapath and FSM

```
module sillyComputation
#(parameter Width = 8
)
( input ck, reset,
  input [Width-1:0] yIn,
  output [Width-1:0] y, x
);

  wire [Width-1:0] i, addiOut, addxOut;
  wire yLoad, yClear, xLoad, xClear, iLoad, iClear;

  register #(Width) I(i, addiOut, iClear, iLoad, ck),
                    Y(y, yIn, yClear, yLoad, ck),
                    X(x, addxOut, xClear, xLoad, ck);

  adder #(Width) addI(addiOut, 8'b1, i),
                 addX(addxOut, y, x);

  compareLT #(Width) cmpX (x, 8'b0, xLT0);

  compareLEQ #(Width) cmpI (i, 8'd10, iLEQ10);

  fsm ctl (xLT0, iLEQ10, yLoad, yClear,
           xLoad, xClear, iLoad, iClear, ck, reset);

endmodule
```

Synthesizable Verilog Summary

Not all of Verilog is synthesizable, designs to be implemented on FPGAs (or ASICs) need to follow rules to make sure the code synthesizes as desired:

- No gate or other delays with #
- Can use functions for combinatorial logic (functions have no delays #); do not use tasks, tasks have delays # and are used mostly for writing testbenches
- No don't cares x in left-hand side of an assign
- If using always to define combinatorial logic ensure sensitivity list has all right-hand values and all left-hand values are assigned something for each input



Type of Logic	Output Assigned To	Edge Specifiers in Sensitivity List
Combinational	An output must be assigned to in all control paths.	Not allowed. The whole input set must be in the sensitivity list. The construct @(*) assures this.
Inferred latch	There must exist at least one control path where an output is not assigned to. From this "omission," the tool infers a latch.	Not allowed.
Inferred flip flop	No affect	Required — from the presence of an edge specifier, the tool infers a flip flop. All registers in the always block are clocked by the specified edge.

References

1. Donald E. Thomas and Philip R. Moorby. " The Verilog Hardware Description Language, Fifth Edition." Springer. 2002
2. "Register-transfer level" Wikipedia, The Free Encyclopedia. Available at: https://en.wikipedia.org/w/index.php?title=Register-transfer_level