Verilog Summary

- The following slides briefly review the main points in Verilog Modeling
  - Verilog Variables
  - Verilog Modules
  - Verilog Dataflow Modeling
  - Verilog Operators
  - Verilog Multiplexed Datapaths
  - Verilog FSM-based Control

- Important! This course does not teach how to do Verilog modeling. These materials are used only to freshen up your mind.

References

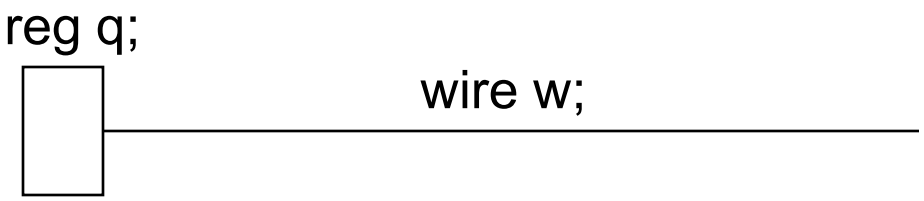
- Official Standard:
  IEEE Std 1364-2001
  IEEE Standard Verilog Hardware Description Language
  https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=954909

- Reference Guide:
  Verilog 2001 Quick Reference Guide by Sutherland HDL
  https://sutherland-hdl.com/pdfs/verilog_2001_ref_guide.pdf

- Textbook:
  Digital Design with RTL Design, VHDL, and Verilog by Frank Vahid
  https://www.amazon.com/Digital-Design-RTL-VHDL-Verilog/dp/0470531088

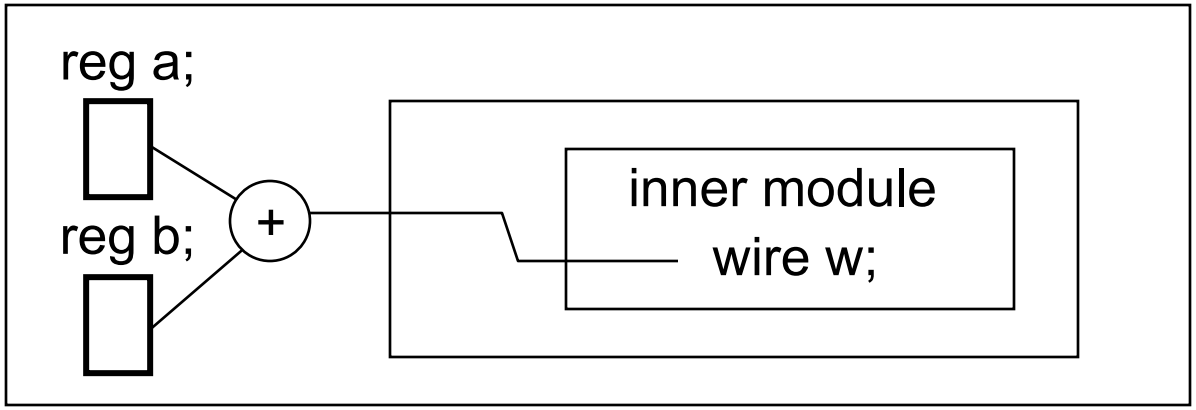
Verilog Variables

Wires and Regs



- A wire and a reg carry information around in a module
- A reg is a variable with storage
  - Assign value v at time t_1, then the value will remain v for $t > t_1$ until reg is re-assigned
- A wire does not have storage, but it reflects the value of a driver
  - Assign value v at time t_1, then the value is not valid for any other time then $t = t_1$

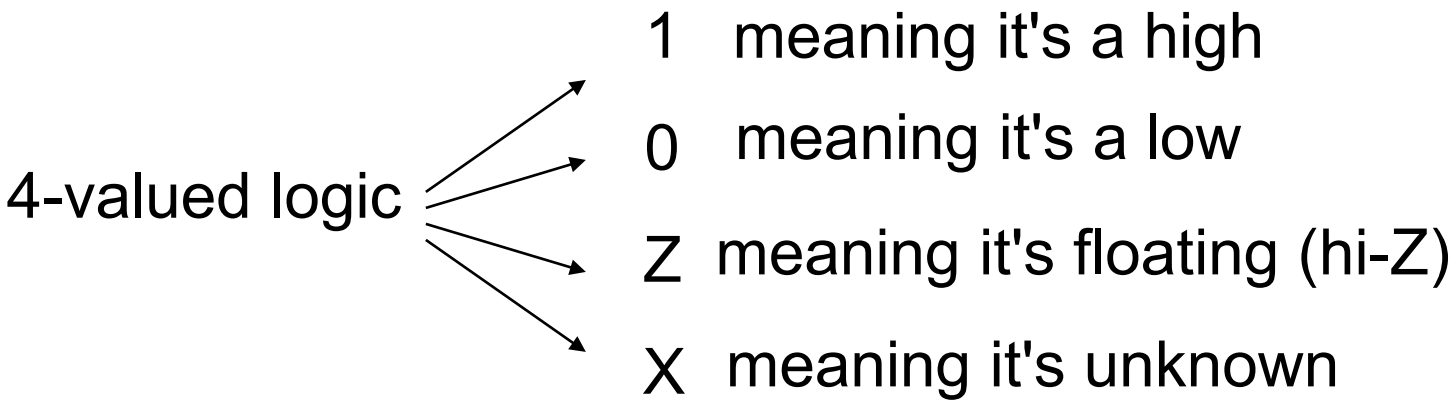
Wires and Regs



- A driver of a wire can be defined through expressions and multiple levels of hierarchy
- Eventually, all wires need a driver, otherwise their value is undefined

Value Levels

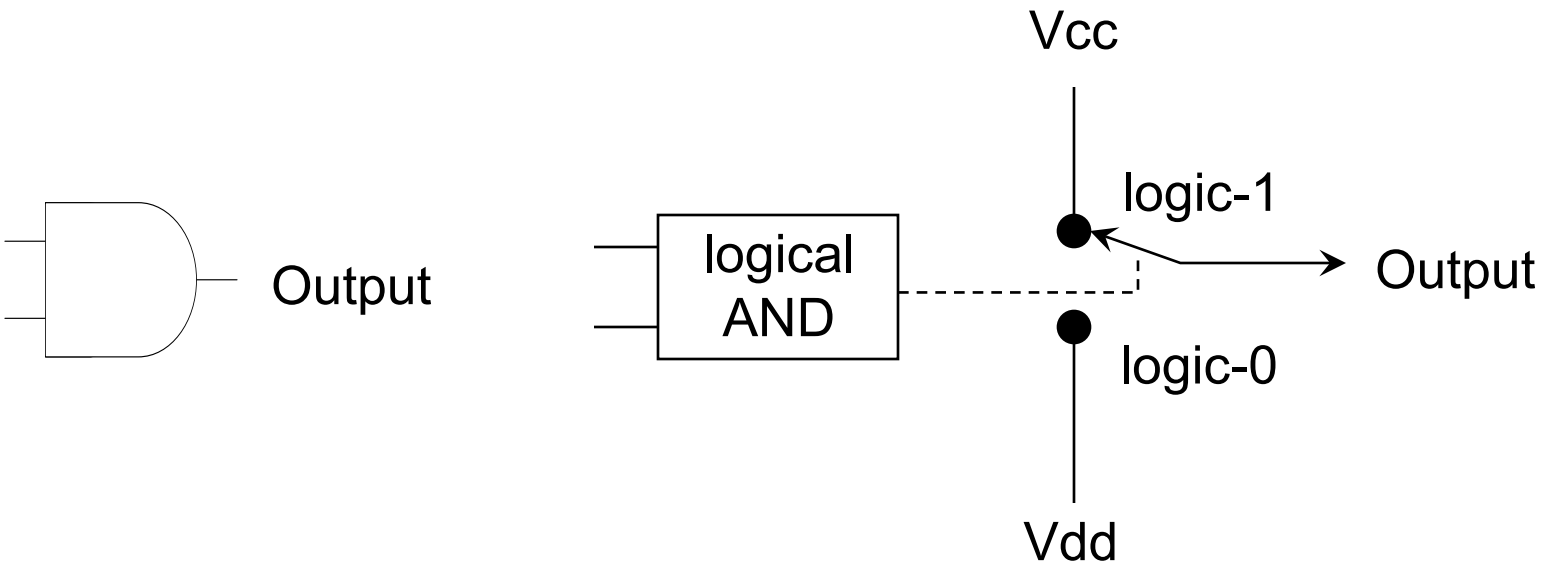
- Each wire can be at 1 of 4 logic levels



- Initially, variables start out at X
- X is contagious. E.g. logic operation (0 or X) = X
- Z is contagious. E.g. logic operation (1 and Z) = X

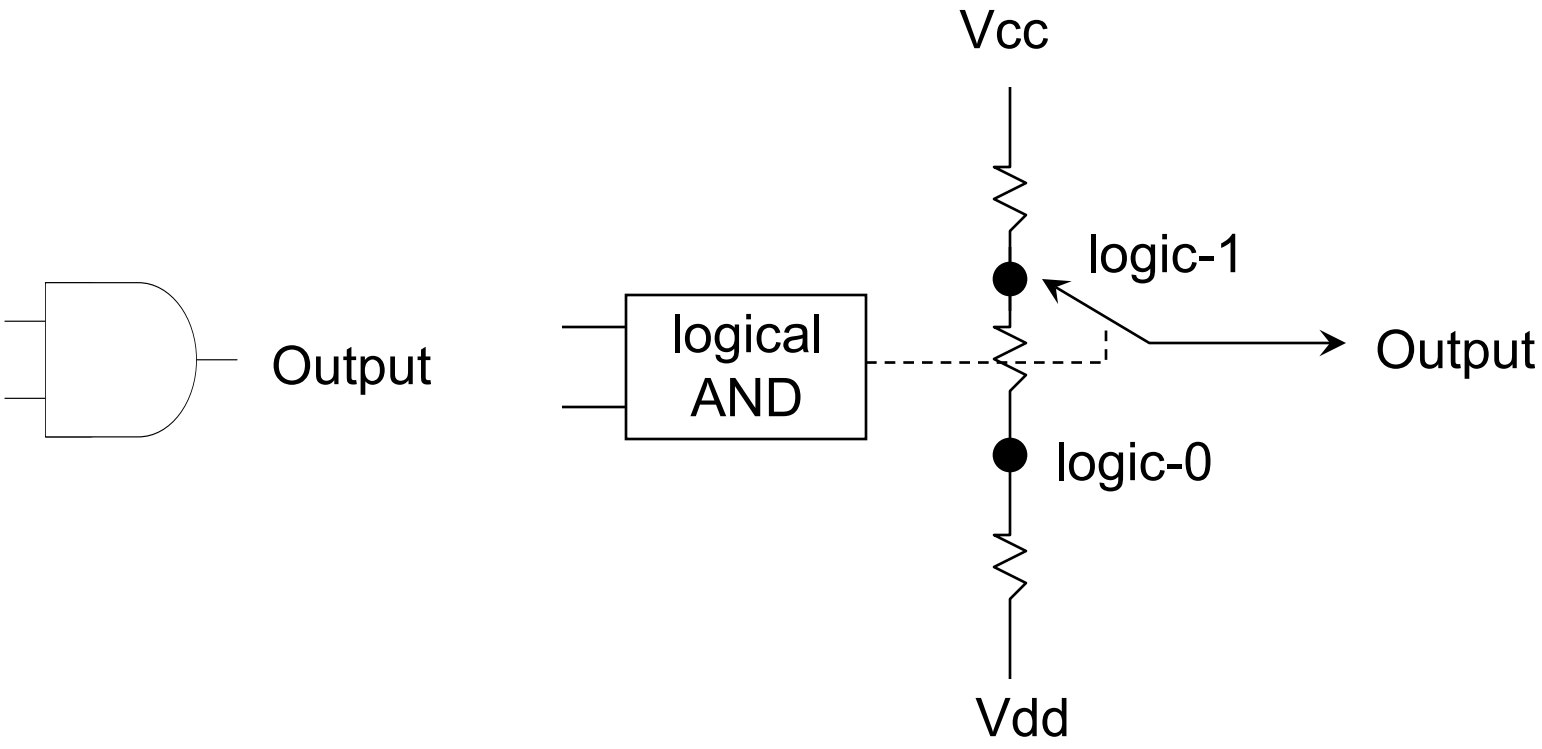
Driving Levels

- Ideally, a driver is a switch



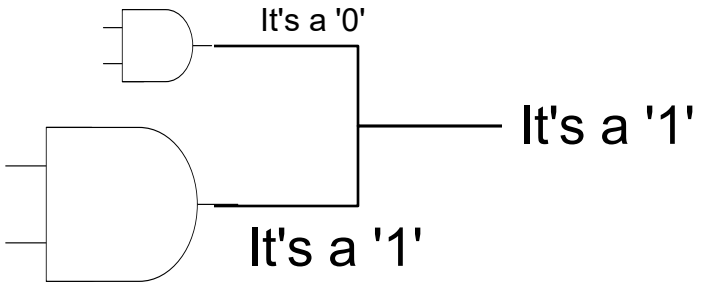
Driving Levels

- In reality, a driver has different strengths



Driving Levels

- A driving level enables to simulate the electrical effect of different driving strengths
  - Can simulate the effect of signal contention
  - Can implement wired-AND and wired-OR
  - Can simulate tri-state buses
  - Can simulate large gates and small gates



- Verilog offers support for this type of simulation
  - We will stick to wire, reg and a single driving level for now

Vectors

```
wire [7:0] b,c; // two 8-bit vectors
                // msb is b[7] and c[7]
reg [0:20] a;   // 21-bit reg, msb is a[0]

b[7]    // bit 7 of the b bit-vector
c[6:5]  // bit 6 and 5 of the c bit-vector
a[3+:4] // bit 3, 4, 5, 6 of a
```

Constants

```
<size>' <base> <number>
```

number of *bits*

b or B Binary
o or O Octal
h or H Hex
d or D Decimal

number in selected base
may use _ as a spacer
0-extended with 0 or 1 msb
x-extended with x msb
z-extended with z msb

For example ...

```
<size>’ <base> <number>

6’b010_111    gives 010111
8'b0110       gives 00000110
8’b1110       gives 00001110
4'bx01        gives xx01
16'H3AB       gives 0000001110101011
24            gives 0…0011000
5'O36         gives 11100
16'Hx         gives xxxxxxxxxxxxxxxx
8'hz          gives zzzzzzzz
```

Other Verilog Variables

- Besides reg, Verilog allows several other kinds of variables that have storage
- These are for test-benches, not for synthesis
  - `integer` (signed 32 bit)
  - `time` (unsigned 64 bit)
  - `real` (double precision)
  - `realtime`
- `reg` is unsigned; Verilog-2001 allows signed
  - `signed reg a[3:0];  // 4-bit two's compl`

Verilog Modules

Modules

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantations;
    dataflow statements;
    always blocks;
    initial blocks;

    tasks and functions;

endmodule
```

Modules

```
module name(portlist);
   port declarations;
   parameter declarations;

   wire declarations;
   reg declarations;
   variable declarations;

   module instantations;
   dataflow statements;
   always blocks;
   initial blocks;

   tasks and functions;

endmodule
```

- Direction of ports
- Module variants
  (module muffin;
    parameter banana_nut;)

Modules

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantations;
    dataflow statements;
    always blocks;
    initial blocks;

    tasks and functions;

endmodule
```

- local communication
- local storage
- local storage (testbench)

within the confines
of this module and this
level of the design hierarchy

Modules

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantiations;          - structural
    dataflow statements;            - dataflow
    always blocks;                  - behavioral
    initial blocks;                 - behavioral

    tasks and functions;            - behavioral

endmodule
```

Example - 4-bit counter in Verilog

```
module tflopcounter(q, clk, reset);

   output [3:0] q;
   input clk, reset;

   T_FF tff0(q[0], clk, reset);
   T_FF tff1(q[1], q[0], reset);
   T_FF tff2(q[2], q[1], reset);
   T_FF tff3(q[3], q[2], reset);

endmodule
```

4 module instantiations

Example - Testbench

```
`timescale 10ns / 10ps;

module stimulus;
  reg clk;
  reg reset;
  wire [3:0] q;

  tflopcounter my_counter(q, clk, reset);

  initial
    clk = 1'b0;

  always
    #5 clk = ~clk;

  initial
  begin
    reset = 1'b1;
    #15  reset = 1'b0;
    #180 reset = 1'b1;
    #10  reset = 1'b0;
    #20 $finish;
  end

  initial
    $monitor($time, "  Output q = %d", q);
endmodule
```

module instantiation

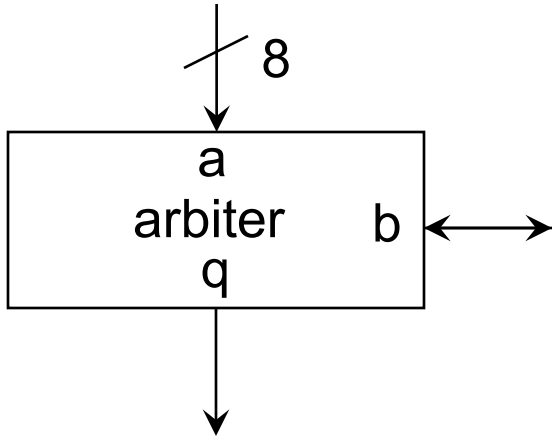
3 initial blocks
1 always block

Port/Parameter Declarations

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantations;
    dataflow statements;
    always blocks;
    initial blocks;

    tasks and functions;

endmodule
```

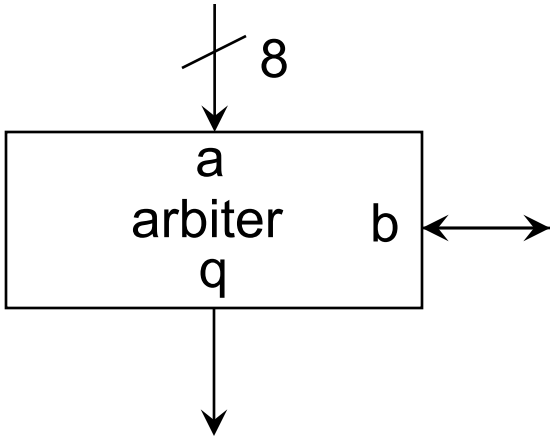
Port List

```
module arbiter(q, a, b);
   output [7:0] q;
   input a;
   inout b;

endmodule
```



Port List

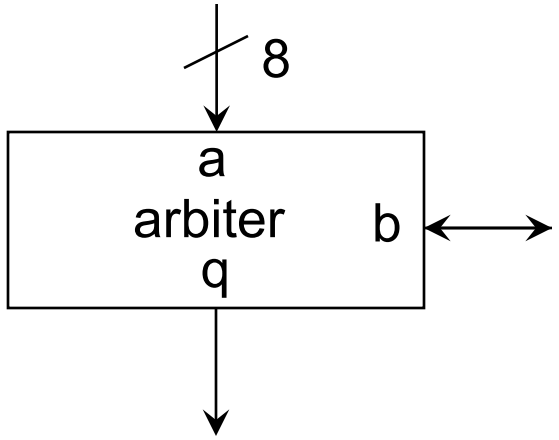
```
module arbiter(q, a, b);
   output [7:0] q;
   reg    [7:0] q;
   input a;
   inout b;

endmodule
```



- inputs, outputs, inouts are wires by default
- output ports can also be reg

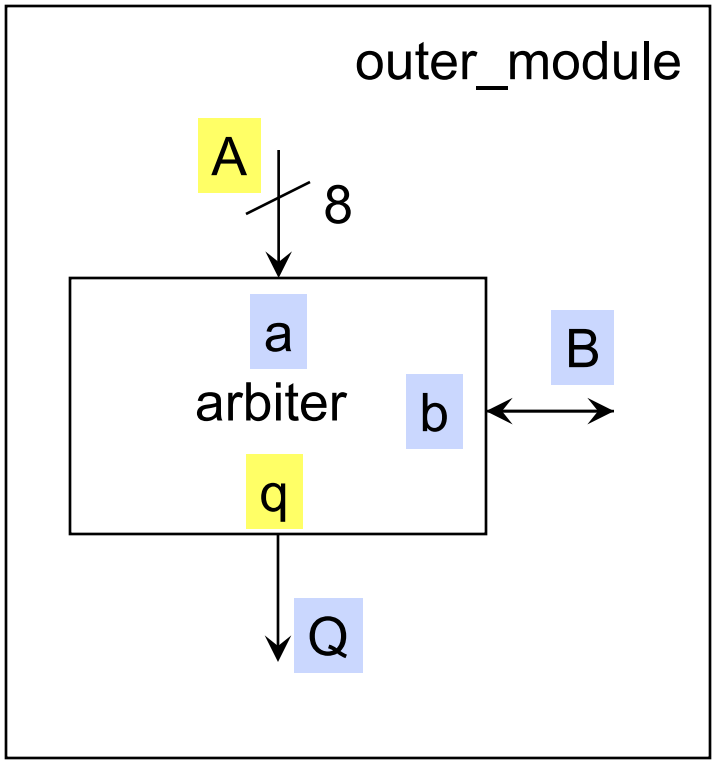
Port List

```
module arbiter(
   output reg [7:0] q,
   input wire a,
   inout wire b);

endmodule
```



Shorthand (Verilog 2001)

Port Reg Rules



A and q MAY be a reg

a, b, B, Q MUST be a wire

Why is this? Hierarchy is a imaginary boundary.
In other words, q and Q, b and B, a and A are the same value.
Hence, we can have only a single storage location for a single value.

Module Parameters

- Define and use symbolic constants in the design of a module
  - The number of bits in a variable
  - The propagation delay of a gate
  - Etc ..

```
module arbiter(
   output reg [7:0] q,
   input wire a,
   inout wire b);

   parameter WIDTH = 32;
   reg [WIDTH-1:0] internal;

endmodule
```

Module Parameters

- Define and use symbolic constants in the design of a module
  - The number of bits in a variable
  - The propagation delay of a gate
  - Etc ..

```
module arbiter
  #(parameter WP = 8)
   (output reg [WP-1:0] q,
    input wire a,
    inout wire b);

  parameter WIDTH = 32;
  reg [WIDTH-1:0] internal;

endmodule
```

Module Instantation and Port Matching

```
module outer_module;
  wire [7:0] q;
  wire a, b;

  arbiter A1(q, , b);
endmodule
```

Positional Matching

unconnected port

```
module outer_module;
  wire [7:0] Q;
  wire A, B;

  arbiter A1(.q(Q),
             .b(B));
endmodule
```

Name-based Matching

unconnected port
remains unnamed

initial and always blocks

```
module name(portlist);
    port declarations;
    parameter declarations;

    wire declarations;
    reg declarations;
    variable declarations;

    module instantations;
    dataflow statements;
    always blocks;
    initial blocks;

    tasks and functions;

endmodule
```

Initial and Always Block Statement

```
module initalways;
reg a, b;

initial
begin
   #10 a = 1;
   #10 a = 0;
end

always
begin
   #10 b = 1;
   #10 b = 0;
end

endmodule;
```

The begin .. end is a block statement
Statements within this block execute sequentially

The initial block is a behavioral construct
It executes a single time when t = 0

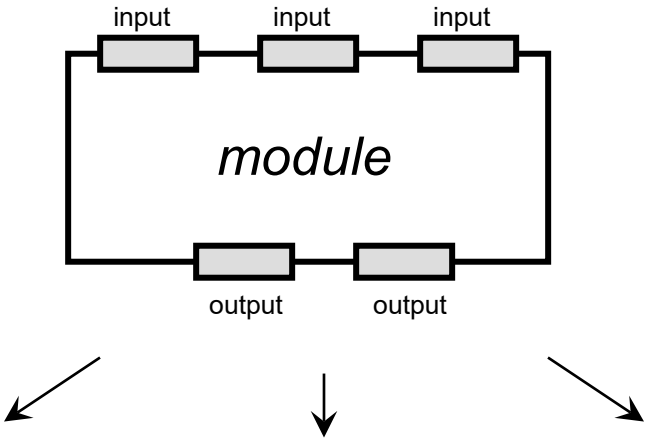
The always block is a behavioral construct
It executes repeatedly

Execution of always can be controlled through sensitivity list

Verilog Dataflow Modeling

Today's topic

- Dataflow Modeling

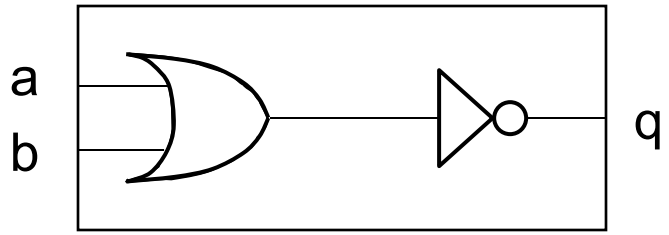


Model with submodules and gates
=
Structural

Model with always and initial blocks
=
Behavioral
Procedural

Model with assign statements
=
Behavioral
Dataflow

Example



Structural

```
module nand(q, a, b)
output q;
input a, b;

wire n;

and G1(n, a, b);
not  G2(q, n);
endmodule
```

Behavioral Procedural

```
module nand(q, a, b)
output q;
reg q;
input a, b;

always @(a or b)
  q = ~(a | b);

endmodule
```

Behavioral Dataflow

```
module nand(q, a, b)
output q;
input a, b;

assign q = ~(a | b);

endmodule
```

Key differences with procedural code

- Must assign nets (wires) instead of registers

Behavioral - Procedural
assign reg

```
module nand(q, a, b)
output q;
reg q;
input a, b;

always @(a or b)
  q = ~(a | b);

endmodule
```

Procedural Assignment

Behavioral - Dataflow
assign wire

```
module nand(q, a, b)
output q;
input a, b;

assign q = ~(a | b);

endmodule
```

Continuous Assignment

Key differences with procedural code

- Must assign nets (wires) instead of registers

Behavioral - Procedural
assign reg

```
module nand(q, a, b)
output q;
reg q;
input a, b;

always @(a or b)
  q = ~(a | b);

endmodule
```

Any change in a or b will cause the always block to re-execute

The procedural assignment evaluates as a result of always block execution

Behavioral - Dataflow
assign wire

```
module nand(q, a, b)
output q;
input a, b;

assign q = ~(a | b);

endmodule
```

Any change in a or b will cause the expression to re-evaluate

Key differences with procedural code

- Must assign nets (wires) instead of registers

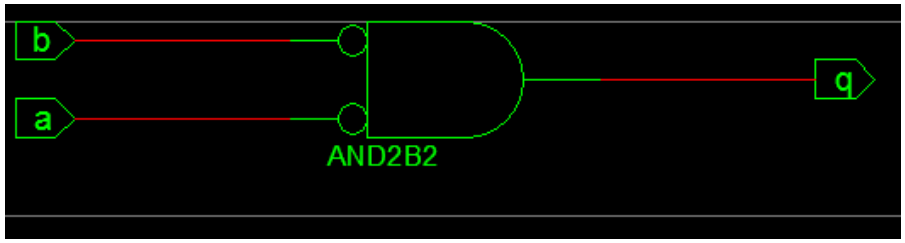
Behavioral - Procedural
assign reg

```
module nand(q, a, b)
output q;
reg q;
input a, b;

always @(a or b)
  q = ~(a | b);

endmodule
```

Behavioral - Dataflow
assign wire

```
module nand(q, a, b)
output q;
input a, b;

assign q = ~(a | b);

endmodule
```

Note that both will end up as the same hardware



Key differences with procedural code

```
module wand(q, a, b, c)
output q;
reg q;
input a, b;

always @(a or b or c)
  begin
  q = a;
  q = q & b;
  q = q & c;
  end

endmodule
```

```
module wand(q, a, b, c)
output q;
input a, b, c;

assign q = a;
assign q = q & b;
assign q = q & c;

endmodule
```

- And-circuit: which is correct, and which is wrong?

Key differences with procedural code

```
module chk(q, a, b, c)
output q;
reg q;
input a, b;

always @(a or b or c)
  begin
  q = a;
  q = q & b;
  q = q & c;
  end

endmodule
```

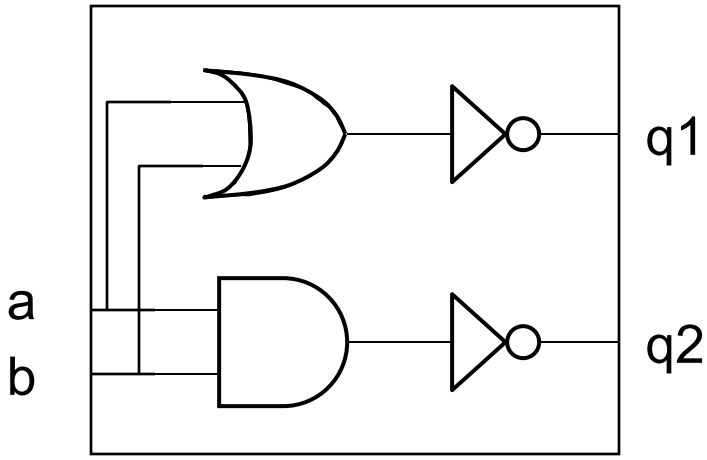
```
module chk(q, a, b, c)
output q;
input a, b, c;

assign q = a;
assign q = q & b;
assign q = q & c;

endmodule
```

- And-circuit: which is correct, and which is wrong?
- Left hand side means three possible assignments to q, results in X (unknown)

Assign two wires

Concurrent Assign statements

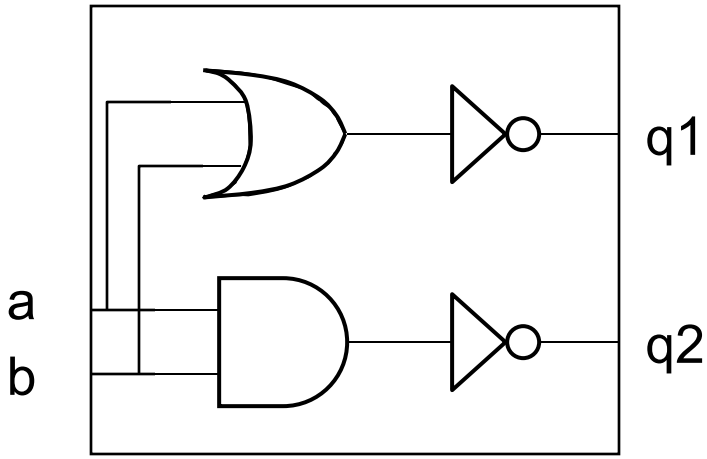
```
module nand(q1, q2, a, b)
output q1, q2;
input a, b;

assign q1 = ~(a | b);
assign q2 = ~(a & b);
...
endmodule
```



Assign two wires

Single Assign statement with a vector assignment

```
module nand(q1, q2, a, b)
output q1, q2;
input a, b;

assign {q1, q2} = {~(a | b),
                   ~(a & b)};
...
endmodule
```



Useful to partition results from an expression:

```
reg a, b, c;
wire c0, d;
assign {c0, d} = a + b + c;
```

Verilog Operators

Operand Types

```
assign {c0, d} = a + b;
```

Left-Hand Side = Right-Hand Side

nets (wire)

nets (wire)
variable (reg)
parameters
numbers
function call

Operand Bit-select and Part-select

```
wire [7:0] n;
```

`n[3:0]`	Bit 3, 2, 1, 0
`n[3]`	Bit 3
`n[x]`	X
`n[3 +:2]`	Bit 3, 4

Operand Memory, Memory Indexing

```
wire [7:0] n[0:99];
```

`n[15]`	Element 16
`n[15][3]`	Bit 3 of Element 16
`n[15][0:3]`	Bits 0:3 of Element 16
`n[15:0]`	Illegal

Operand Types

Left-Hand Side = Right-Hand Side

'net' can be

- net
- net bit-select [n] or part-select [n:m]
- indexed net
- indexed net bit-select or part-select
- concatenated net

```
wire [7:0] n[0:99];
{n[15][0:3], n[2]} = 12_bit_expr;
```

Operators

- Arithmetic
- Bitwise
- Reduction
- Logical
- Relational
- Shift
- Selection
- Concatenation & Replication

We will only discuss Verilog-specific Operators

General Rules for Arithmetic Precision

- Assignments will not loose precision if the target is large enough
  - If a and b are 15 bit, and c is 16 bit, then c = a + b will not loose precision
- Assignments will loose precision if the target is not large enough
  - If a and b are 15 bit, and c is 8 bit, then c = a + b captures the 8 lsb of the addition
- Standalone expressions *may* loose precision
  - If a and b are 15 bit, then the standalone expression a + b uses 15 bit (max wordlength over a and b), and thus may loose precision
  - Standalone expressions occur in some system calls like e.g. $display("a+b=%h", a+b);

Reduction Operator

- Condense all bits from a vector into a single bit using a specified logical operation

```
assign q1 = &a;  // reduction-and
assign q2 = |b;  // reduction-or
assign q3 = ^c;  // reduction-xor
assign q4 = ~&d; // reduction-nand
assign q5 = ~|e; // reduction-nor
assign q6 = ~^f; // reduction-xor
```

- Similar to an N-input gate of the specified type, where N equals the wordlength of the operand

- Examples
  - `|(4'b0001) =`
  - `^(4'b0111) =`
  - `~|(2'b11) =`
  - `&(2'b1x) =`

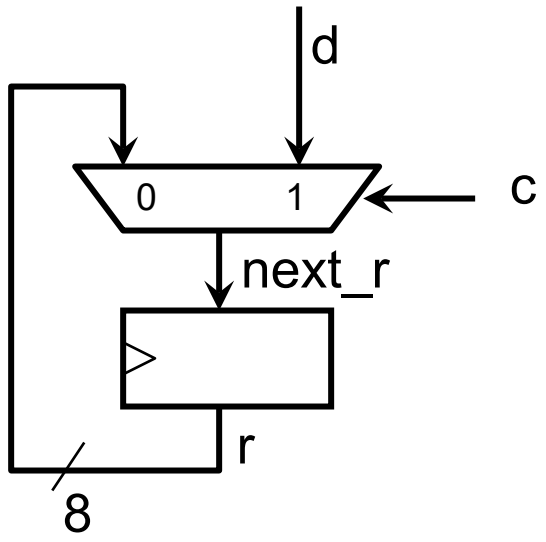
Reduction Operator

- Condense all bits from a vector into a single bit using a specified logical operation

```
assign q1 = &a;  // reduction-and
assign q2 = |b;  // reduction-or
assign q3 = ^c;  // reduction-xor
assign q4 = ~&d; // reduction-nand
assign q5 = ~|e; // reduction-nor
assign q6 = ~^f; // reduction-xor
```

- Similar to an N-input gate of the specified type, where N equals the wordlength of the operand
- Examples
  - `|(4'b0001) = 1`
  - `^(4'b0111) = 1`
  - `~|(2'b11) = 0`
  - `&(2'b1x) = x`

Selection operator

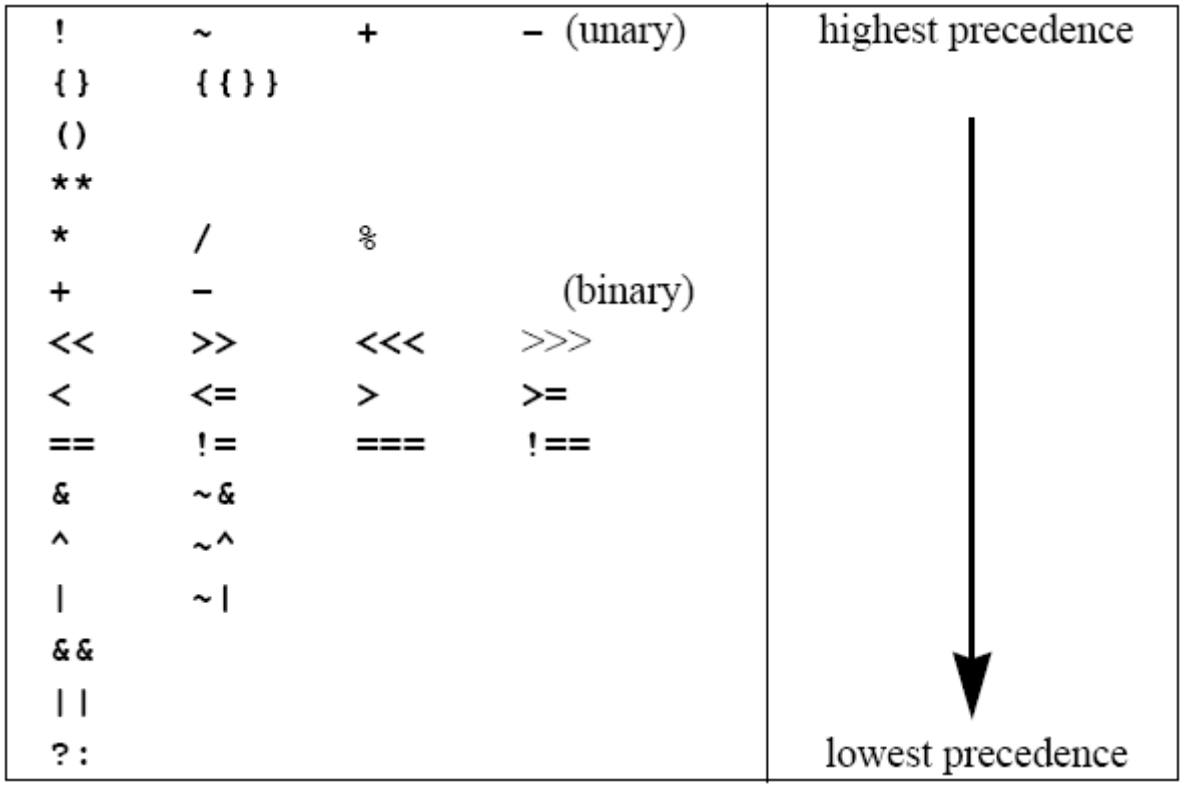



```
wire [7:0] d;
wire [7:0] next_r;
reg  [7:0] r;

wire c;

always @(posedge clk)
  r = next_r;

assign next_r = c ? d : r;
```

Concatenation and Replication

- Form wire bundles from single wires or smaller wire bundles
  - {a, b[3:2], c} = {a, b[3], b[2], c} = a 4-bit vector
- Concatenation can be replicated
  - 4{a} = {a, a, a, a}
  - {a, 3{b,c}} = {a, b, c, b, c, b, c}

Operator Precedence



Example 1

- Write a 3-to-1 multiplexer
  - Inputs D0, D1, D2 (4-bit)
  - Select input S (2-bit)
  - Output Y (4-bit)

S	Y
0	D0
1	D0
2	D1
3	D2

Example 1

- Write a 3-to-1 multiplexer
  - Inputs D0, D1, D2 (4-bit)
  - Select input S (2-bit)
  - Output Y (4-bit)

S	Y
0	D0
1	D0
2	D1
3	D2

```
module mux(output [7:0] Y,
           input  [1:0] S,
           input  [7:0] D0, D1, D2);

  assign Y = (S == 0) ? D0 :
             (S == 1) ? D0 :
             (S == 2) ? D1 : D2;
endmodule
```

Example 1

- Write a 3-to-1 multiplexer
  - Inputs D0, D1, D2 (4-bit)
  - Select input S (2-bit)
  - Output Y (4-bit)

S	Y
0	D0
1	D0
2	D1
3	D2

```
module mux(output [7:0] Y,
           input  [1:0] S,
           input  [7:0] D0, D1, D2);

  assign Y = (S[1]) ? ((S[0]) ? D1 : D2) : D0;

endmodule
```

Example 2

- Develop an 8-bit * 8-bit multiplier with 8-bit output, reflecting the 8 msb of the multiplication

Example 2

- Develop an 8-bit * 8-bit multiplier with 8-bit output, reflecting the 8 msb of the multiplication

```
module mul(output [7:0] Y,
           input  [7:0] D0, D1);

  wire   [15:0] mul_out;

  assign mul_out = D0 * D1;
  assign Y = mul_out[15:8];

endmodule
```

Example 3

- Develop a rounding module with 8-bit input and 4-bit output. The 4-bit input captures the *rounded* output of the 8 input bits. That is, if the 4 lsb of the input are bigger than 0111, then the output should be incremented.

Example 3

- Develop a rounding module with 8-bit input and 4-bit output. The 4-bit input captures the *rounded* output of the 8 input bits. That is, if the 4 lsb of the input are bigger than 0111, then the output should be incremented.

Rounding Trick: Add 1/2 lsb

1001	1100	8-bit input
0000	1000	add 1/2 lsb of 4-bit output
1010		4-bit output

Example 3

- Develop a rounding module with 8-bit input and 4-bit output. The 4-bit input captures the *rounded* output of the 8 input bits. That is, if the 4 lsb of the input are bigger than 0111, then the output should be incremented.

```
module round(output [3:0] Y,
             input  [7:0] D);

  wire [7:0] A;

  assign A = D + 4'b1000;
  assign Y = A[7:4];

endmodule
```

Example 4

- Develop a multiply-accumulate module with 40-bit accumulator and 2x16-bit inputs. The accumulator register is triggered on pos clock edge and has a negative asynchronous reset.

Example 4

- Develop a multiply-accumulate module with 40-bit accumulator and 2x16-bit inputs. The accumulator register is triggered on pos clock edge and has a negative asynchronous reset.

```
module mac(output reg [39:0] Y,
           input             clk, rst,
           input [15:0]      D0, D1);

  wire   [39:0] next_Y;

  always @(posedge clk or negedge rst)
    Y = (rst) ? next_Y : 0;

  assign next_Y = Y + (D0 * D1);

endmodule
```

Example 5

- Design a programmable shifter that will shift an 8-bit word over 0 .. 3 positions to the right, inserting zeroes at the msb side

Example 5

- Design a programmable shifter that will shift an 8-bit word over 0 .. 3 positions to the right, inserting zeroes at the msb side

```
module shft(output [7:0] Y,
            input  [1:0] s,
            input  [7:0] a);

  assign Y = (s == 2'b11) ? (a << 3):
             (s == 2'b10) ? (a << 2) :
             (s == 2'b01) ? (a << 1) :
              a

endmodule
```

Example 5

- Design a programmable shifter that will shift an 8-bit word over 0 .. 3 positions to the right, inserting zeroes at the msb side

```
module shft(output [7:0] Y,
            input  [1:0] s,
            input  [7:0] a);
  wire [7:0] Y1;

  assign Y1 = (s[0]) ? (a  >> 1) : a;
  assign Y  = (s[1]) ? (Y1 >> 1) : Y1;

endmodule
```

Example 5

- Design a programmable shifter that will shift an 8-bit word over 0 .. 3 positions to the right, inserting zeroes at the msb side

```
module shft(output [7:0] Y,
            input  [1:0] s,
            input  [7:0] a);
  wire [7:0] Y1;

  assign Y1 = (s[0]) ? {1'b0,a[7:1]}  : a;
  assign Y  = (s[1]) ? {1'b0,Y1[7:1]} : Y1;

endmodule
```

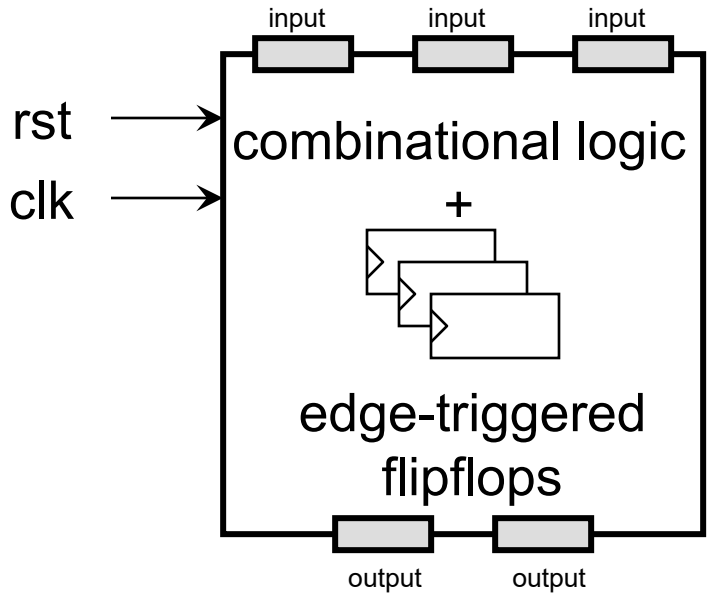
Verilog Multiplexed Datapaths

Multiplexed Datapaths - Disclaimer

- We will cover very specific Verilog modeling guidelines to build hardware modules called 'multiplexed datapaths'
- Not the only way to build hardware in Verilog
  - We will see other 'design methods' later
- But, if you use the 'multiplexed datapath' method, you MUST stick to the following guidelines

What is a multiplexed datapath ?

- A module with a single clk and rst input
- Zero or more inputs, one or more outputs
- Edge-triggered flip-flops
- Outputs depend *only* on registers



Why single-clock ?

- A module with a single clk and rst input
- Zero or more inputs, one or more outputs
- Edge-triggered flip-flops
- Outputs depend *only* on registers

Simplifies generation and distribution of clock signal in the implementation

Why edge-triggered flip-flops ?

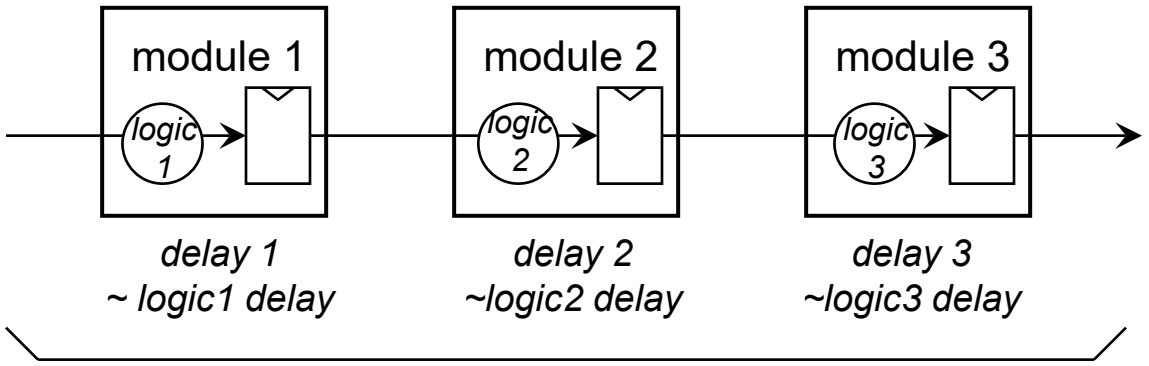
- A module with a single clk and rst input
- Zero or more inputs, one or more outputs
- Edge-triggered flip-flops
- Outputs depend *only* on registers

A single type of storage module simplifies test
A single type of reset signal simplifies initialization

Why outputs depend only on registers ?

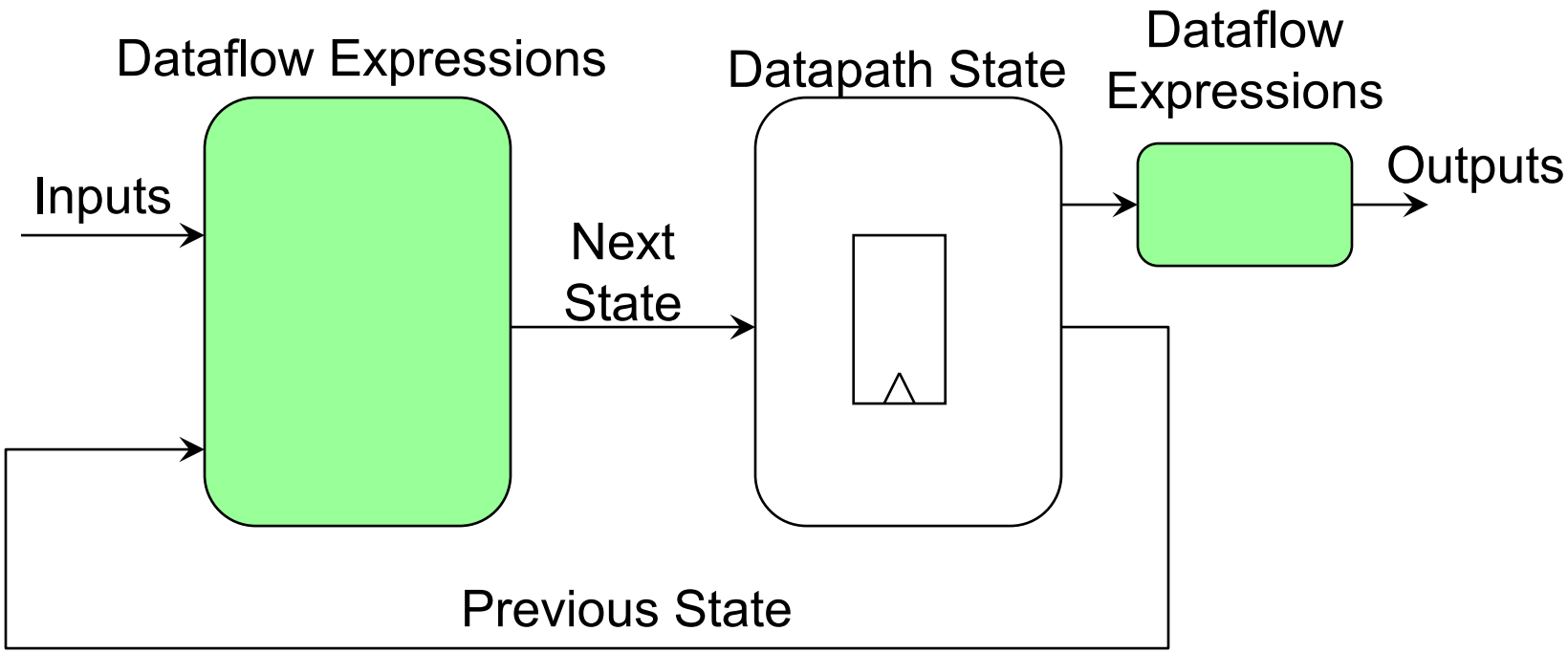
- A module with a single clk and rst input
- Zero or more inputs, one or more outputs
- Edge-triggered flip-flops
- Outputs depend *only* on registers

Make sure that system-level delay, i.e. critical path,
is just max(module-level delay), not sum(module-level delay)



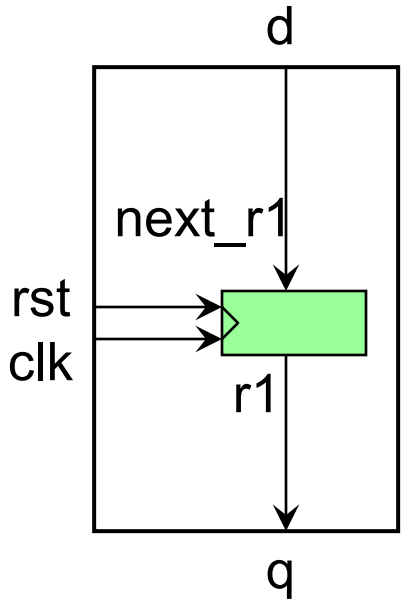
System Delay = max(delay1, delay2, delay3) + routing_delay

Template for a Multiplexed Datapath



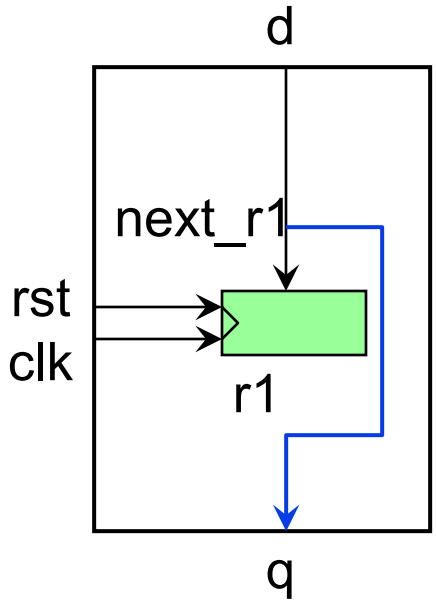
How to model an edge-triggered register

```
module a_module(output wire q,
                input wire rst,
                input wire clk,
                input wire d);
  wire next_r1;
  reg  r1;

  always @(posedge clk or negedge rst)
    if (rst)
      r1 = next_r1;
    else
      r1 = 0;

  assign next_r1 = d;
  assign q = r1;
endmodule
```



How to model an edge-triggered register

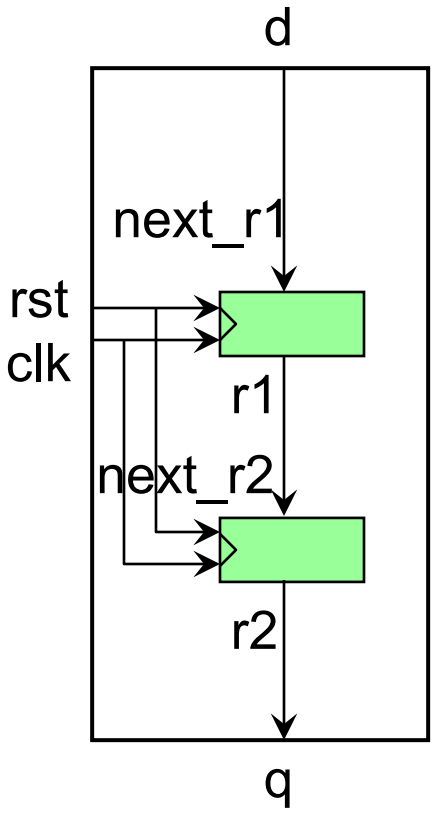
```
module a_module(output wire q,
                input wire rst,
                input wire clk,
                input wire d);
  wire next_r1;
  reg  r1;

  always @(posedge clk or negedge rst)
    if (rst)
      r1 = next_r1;
    else
      r1 = 0;

  assign next_r1 = d;
  assign q = next_r1;
endmodule
```



No!

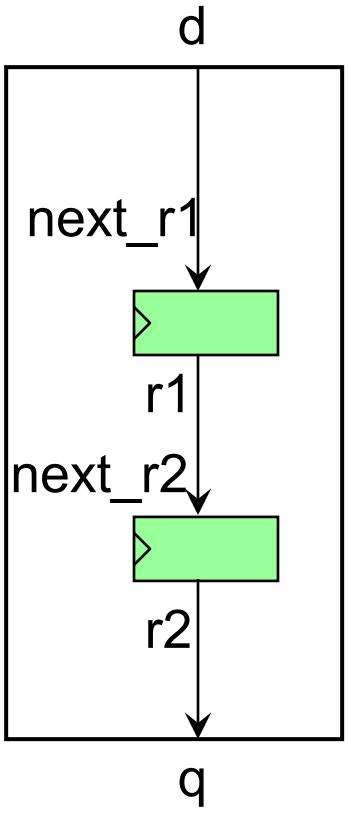
How to model two edge-triggered registers

```
module a_module(output wire q,
                input wire rst,
                input wire clk,
                input wire d);
  wire next_r1, next_r2;
  reg  r1, r2;

  always @(posedge clk or negedge rst)
    if (rst) begin
      r1 = next_r1;
      r2 = next_r2;
    end else begin
      r1 = 0;
      r2 = 0;
    end
  assign next_r1 = d;
  assign next_r2 = r1;
  assign q = r2;
endmodule
```



How to model two edge-triggered registers

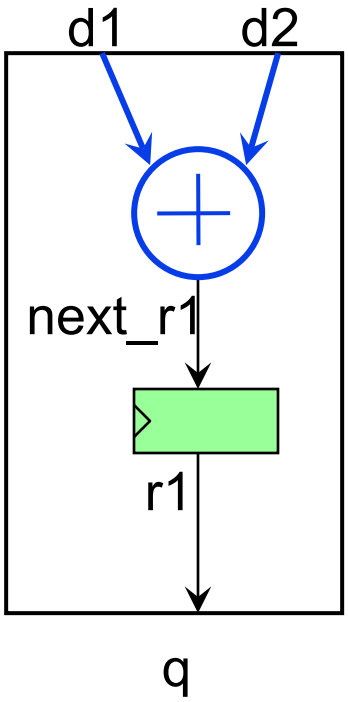
```
module a_module(output wire q,
                input wire rst,
                input wire clk,
                input wire d);
  wire next_r1, next_r2;
  reg  r1, r2;

  always @(posedge clk or negedge rst)
    if (rst) begin
      r1 = next_r1;
      r2 = next_r2;
    end else begin
      r1 = 0;
      r2 = 0;
    end
  assign next_r1 = d;
  assign next_r2 = r1;
  assign q = r2;
endmodule
```



Will leave out drawing of clk net and rst net from now

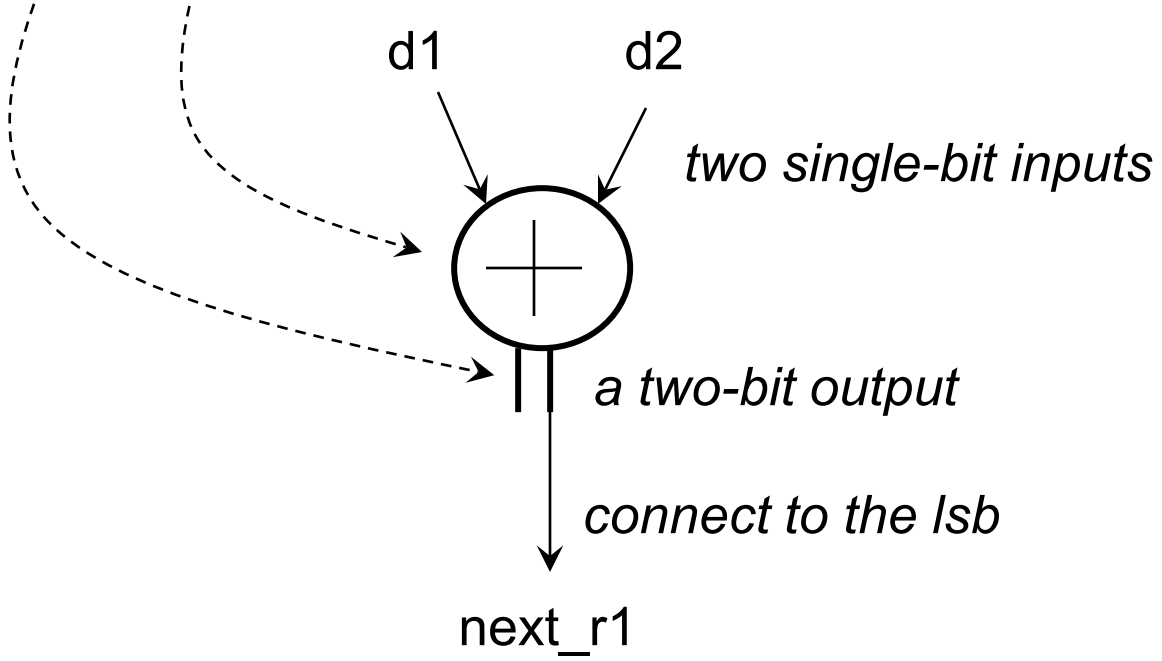
Add two inputs, capture in a register

```
module a_module(output wire q,
                input wire rst,
                input wire clk,
                input wire d1, d2);
  wire next_r1;
  reg  r1;

  always @(posedge clk or negedge rst)
    if (rst)
      r1 = next_r1;
    else
      r1 = 0;

  assign next_r1 = d1 + d2;
  assign q = r1;
endmodule
```



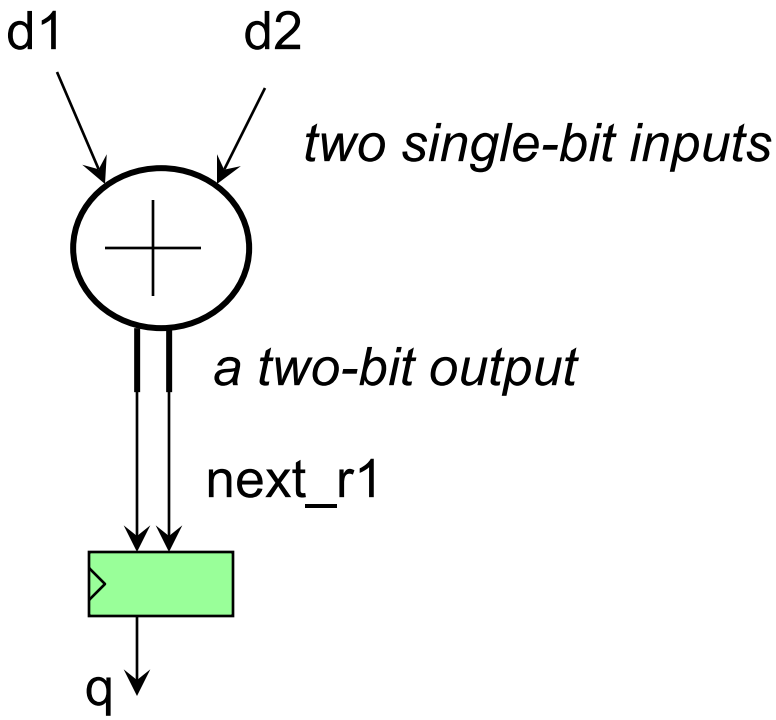
Take a close look at the addition

```
input d1, d2;
wire next_r1;

assign next_r1 = d1 + d2;
```



How to get the carry bit ?

```
input d1, d2;
wire [1:0] next_r1;
reg [1:0] r1;

assign next_r1 = d1 + d2;
assign q = r1[1];
```



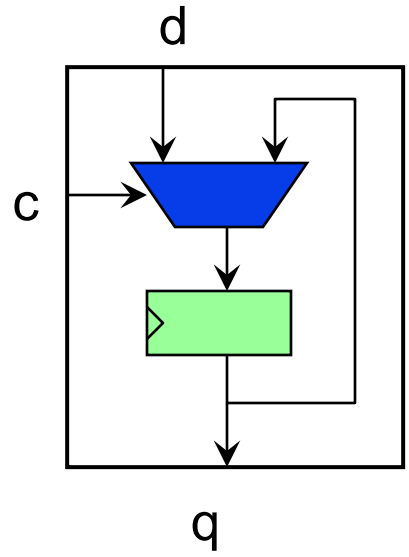
Not so good, will require a two-bit register

Better solution

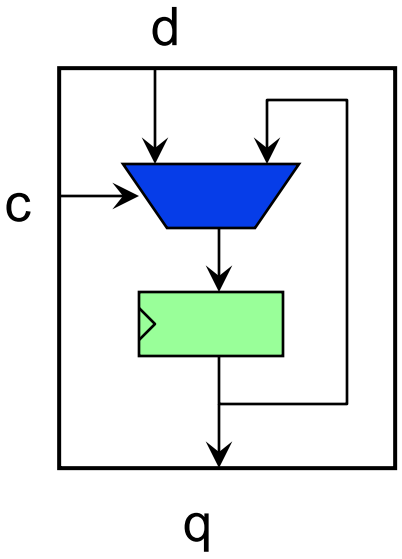
```
module a_module(output wire q,
                input wire rst,
                input wire clk,
                input wire d1, d2);

  wire next_r1;
  reg  r1;
  wire dummy;

  always @(posedge clk or negedge rst)
    if (rst)
      r1 = next_r1;
    else
      r1 = 0;

  assign {next_r1, dummy} = d1 + d2;
  assign q = r1;
endmodule
```

How to model an multiplexed register

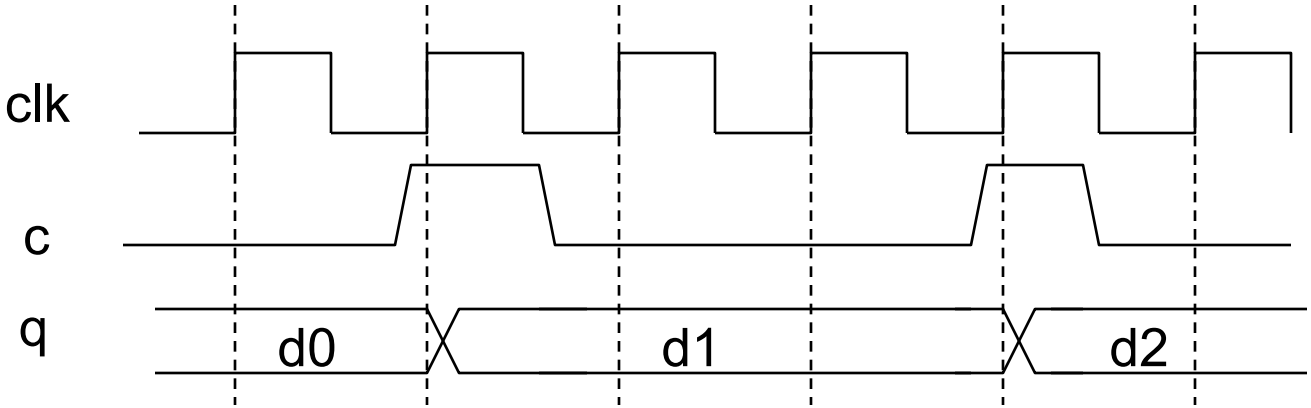
```
module a_module(output wire q,
                input wire rst,
                input wire clk,
                input wire d, c);
  wire next_r1;
  reg  r1;

  always @(posedge clk or negedge rst)
    if (rst)
      r1 = next_r1;
    else
      r1 = 0;

  assign next_r1 = c ? d : r1;
  assign q = r1;
endmodule
```



Multiplexer allows controlled update

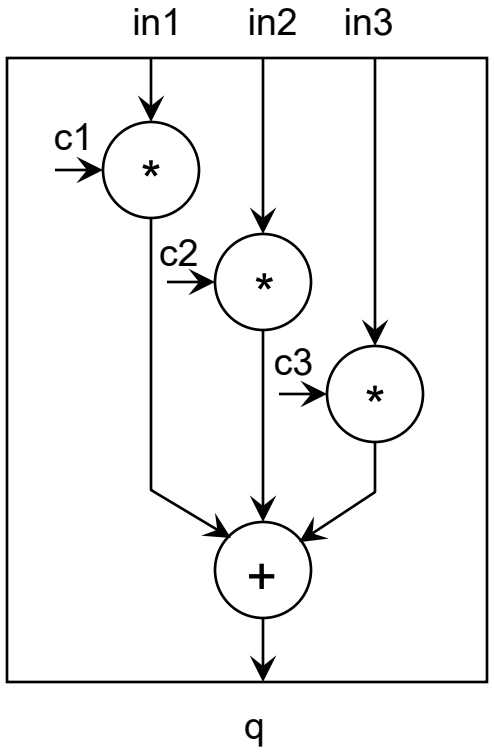


- Eg. d is only valid once every three clock cycles
- Capture d in register only when c = 1
- Otherwise, retain value of q
- c is a 'logical' clock signal (in contrast to the physical clock signal clk)

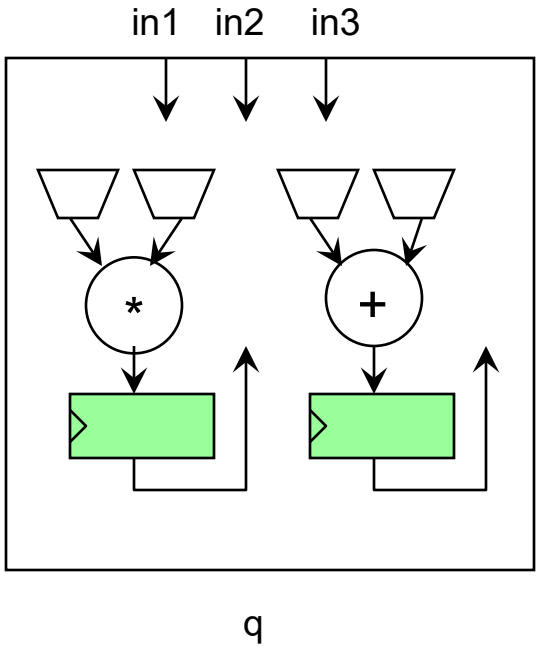


Muxed registers are key to Muxed Datapath

- Overall Idea



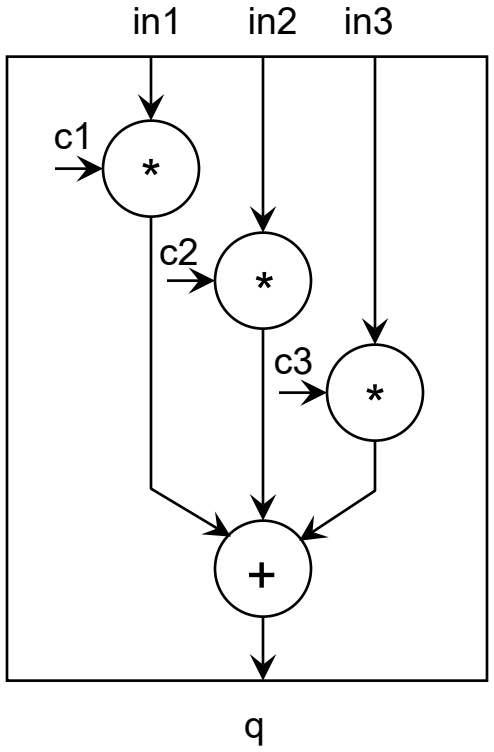
One clock cycle
3 Multiplier
2 Adders (2-input)

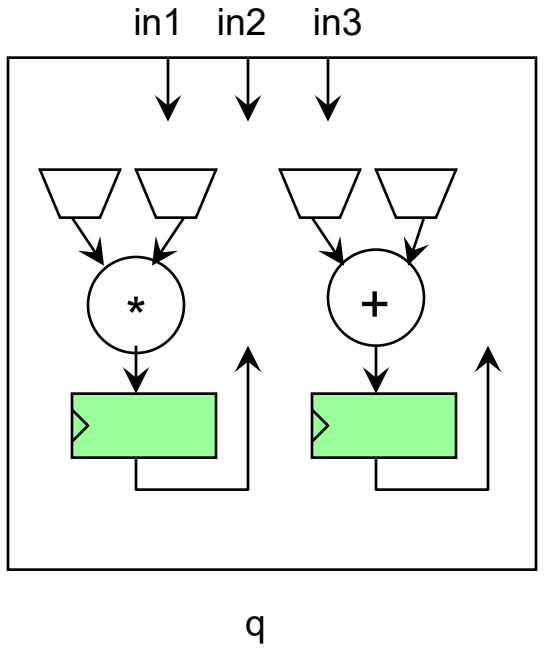


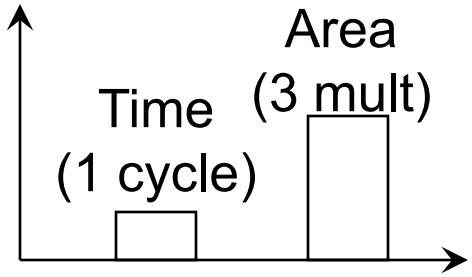
Three clock cycles
1 Multiplier
1 Adder
n Multiplexers

Muxed registers are key to Muxed Datapath

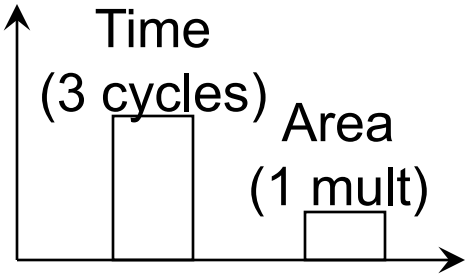
- Overall Idea



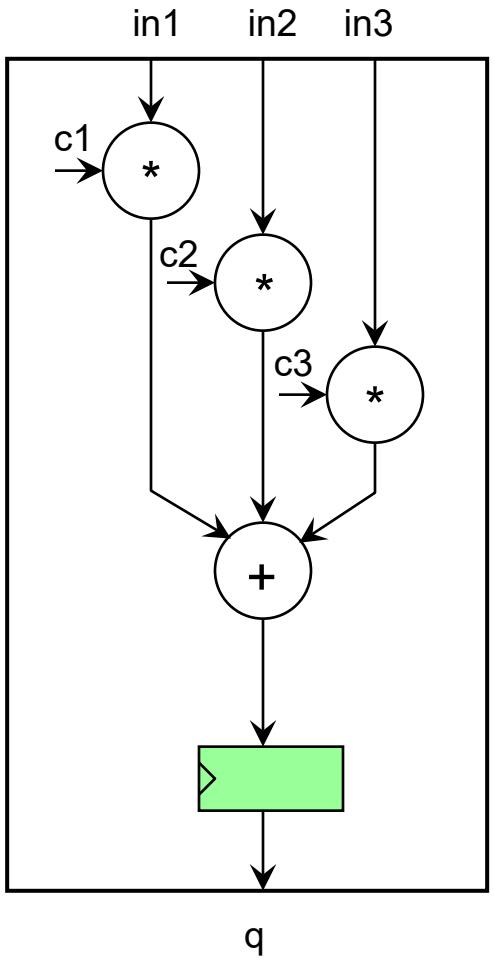




AREA
TIME
TRADEOFF

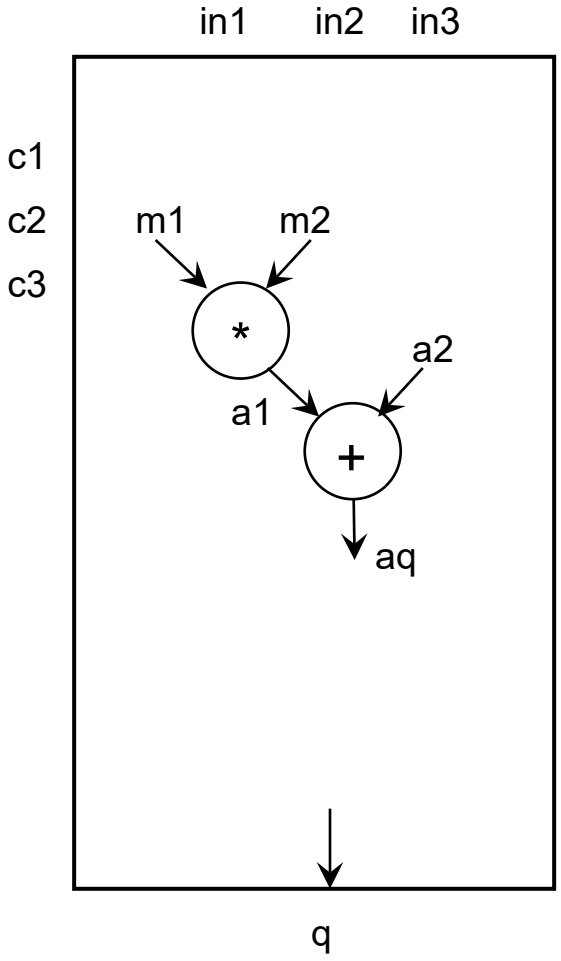


Let's try this



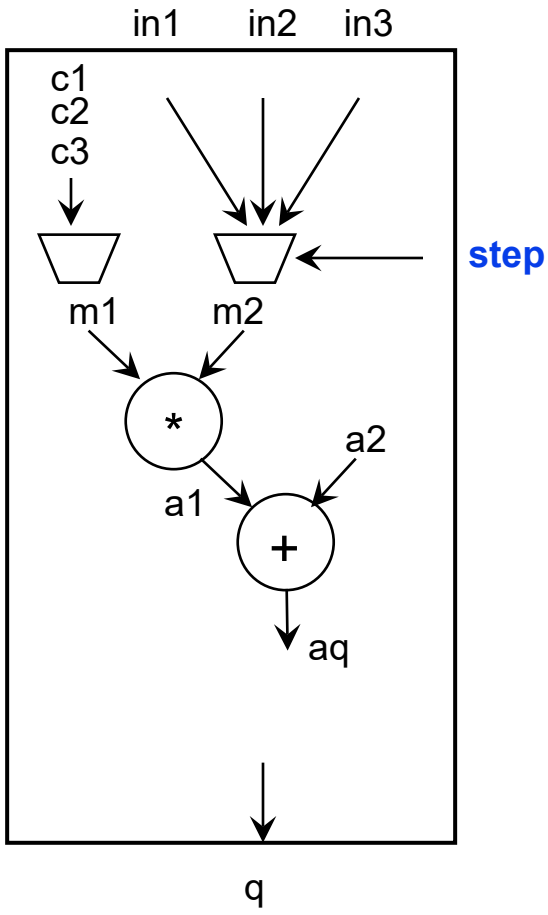

```
module vecmul(q, rst, clk, in1, in2, in3);
  output [7:0] q;
  input rst, rst;
  input [7:0] in1, in2, in3;
  parameter c1 = 8'd2;
  parameter c2 = 8'd4;
  parameter c3 = 8'd6;

  wire next_r1;
  reg  r1;

  always @(posedge clk or negedge rst)
    if (rst)
      r1 = next_r1;
    else
      r1 = 0;

  assign next_r1 = c1 * in1 +
                   c2 * in2 +
                   c3 * in3;
  assign q = r1;
endmodule
```

Now, we reduce the number of *, + operations




```
module vecmul(q, rst, clk, in1, in2, in3);
  output [7:0] q;
  input rst, rst;
  input [7:0] in1, in2, in3;
  parameter c1 = 8'd20;
  parameter c2 = 8'd40;
  parameter c3 = 8'd60;

  wire next_r1;
  reg  r1;

  wire [7:0] m1, m2;
  wire [10:0] a1, a2, aq; // 16-bit worst case

  assign aq = a2 + a1;
  assign a1 = m1 * m2;

endmodule
```

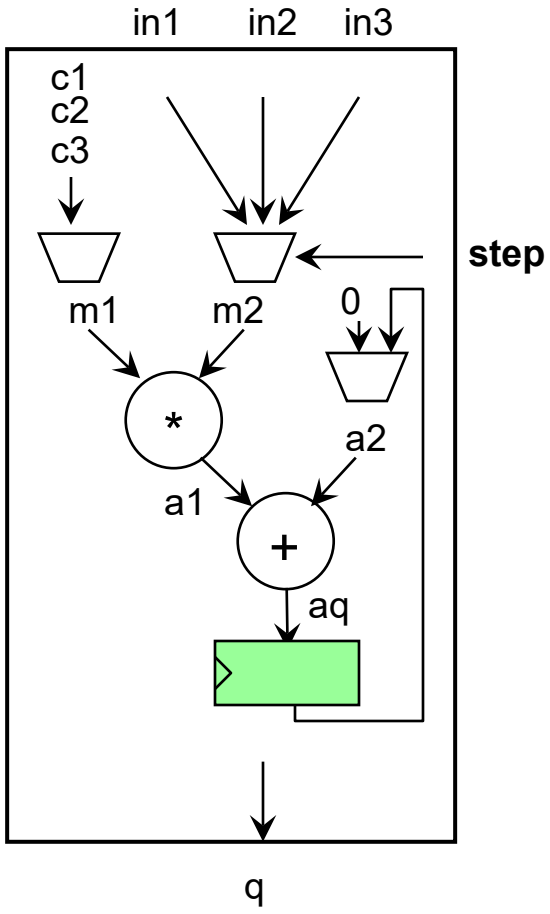
Add muxes on m1, m2. Add control input.




```
module vecmul(q, rst, clk, in1, in2, in3, step);
  output [7:0] q;
  input rst, rst;
  input [7:0] in1, in2, in3;
  input [1:0] step;
  parameter c1 = 8'd20;
  parameter c2 = 8'd40;
  parameter c3 = 8'd60;

  wire next_r1;
  reg  r1;

  wire [7:0] m1, m2;
  wire [10:0] a1, a2, aq; // 16-bit worst case

  assign aq = a2 + a1;
  assign a1 = m1 * m2;
  assign m1 = (step == 0) ? c1 :
              (step == 1) ? c2 : c3;
  assign m2 = (step == 0) ? in1 :
              (step == 1) ? in2 : in3;
endmodule
```

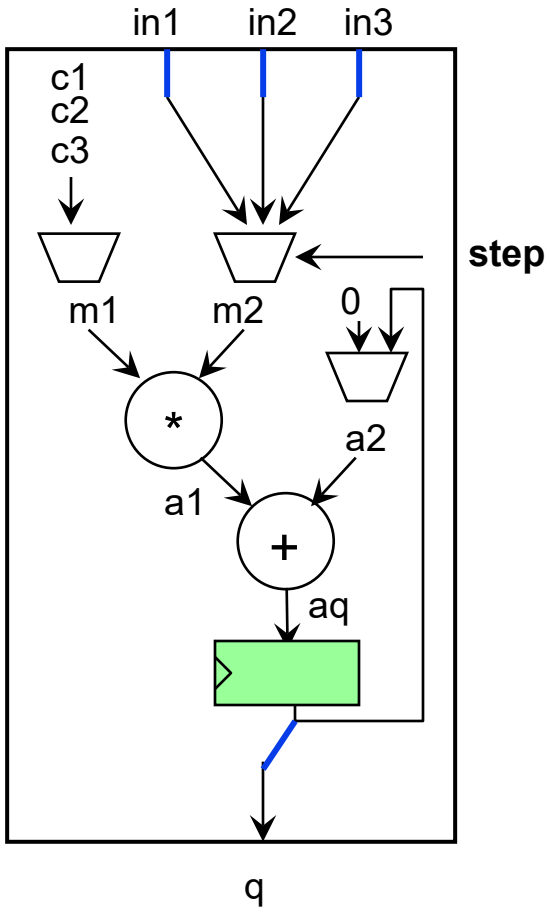
step is a control input

Add muxes on a2. Add accumulator reg.



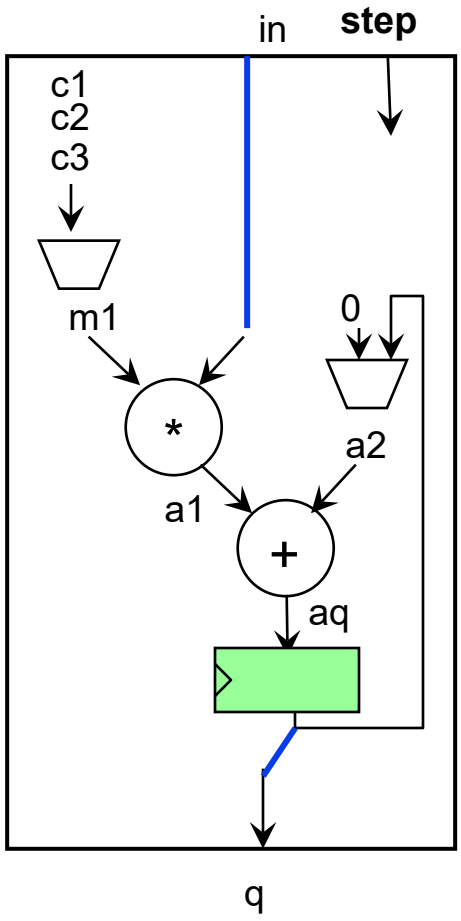

```
module vecmul(q, rst, clk, in1, in2, in3, step);
  output [7:0] q;
  input rst, rst;
  input [7:0] in1, in2, in3;
  input [1:0] step;
  parameter c1 = 8'd20;
  parameter c2 = 8'd40;
  parameter c3 = 8'd60;

  wire next_r1;
  reg  r1;

  wire [7:0] m1, m2;
  wire [10:0] a1, a2, aq; // 16-bit worst case
  reg [10:0] aq_reg;

  always @(posedge clk) aq_reg = aq;

  assign aq = a2 + a1;
  assign a1 = m1 * m2;
  assign m1 = (step == 0) ? c1 :
              (step == 1) ? c2 : c3;
  assign m2 = (step == 0) ? in1 :
              (step == 1) ? in2 : in3;
  assign a2 = (step == 0) ? 0 : aq_reg;
endmodule
```

System Interconnect.




```
module vecmul(q, rst, clk, in1, in2, in3, step);
  output [7:0] q;
  input rst, rst;
  input [7:0] in1, in2, in3;
  input [1:0] step;
  parameter c1 = 8'd20;
  parameter c2 = 8'd40;
  parameter c3 = 8'd60;

  wire next_r1;
  reg  r1;

  wire [7:0] m1, m2;
  wire [10:0] a1, a2, aq; // 16-bit worst case
  reg [10:0] aq_reg;

  always @(posedge clk) aq_reg = aq;

  assign aq = a2 + a1;
  assign a1 = m1 * m2;
  assign m1 = (step == 0) ? c1 :
              (step == 1) ? c2 : c3;
  assign m2 = (step == 0) ? in1 :
              (step == 1) ? in2 : in3;
  assign a2 = (step == 0) ? 0 : aq_reg;
  assign q = aq_reg;
endmodule
```

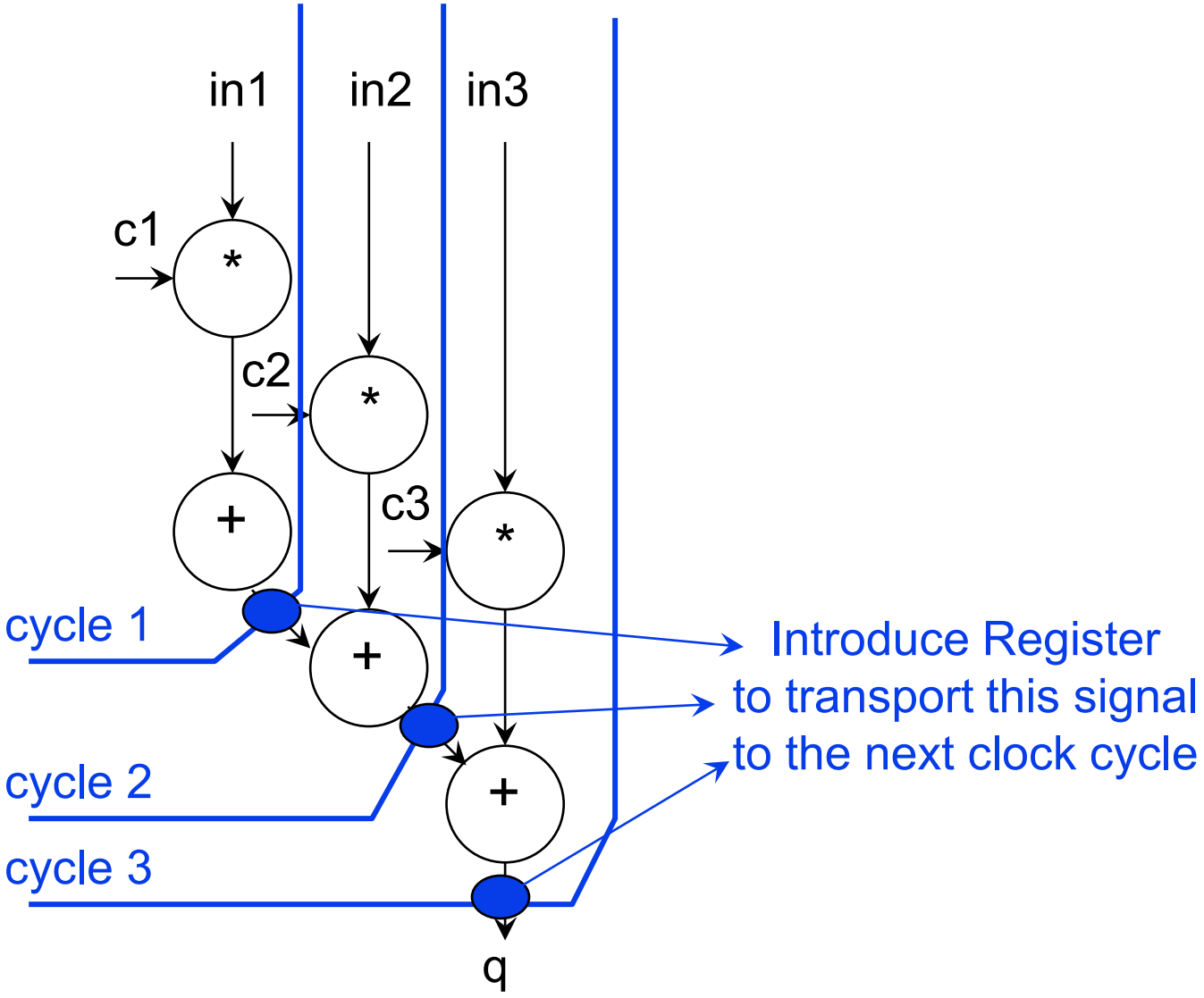
Simple Optimizations




```
module vecmul(q, rst, clk, in, step);
  output [7:0] q;
  input rst, rst;
  input [7:0] in;
  input [1:0] step;
  parameter c1 = 8'd20;
  parameter c2 = 8'd40;
  parameter c3 = 8'd60;

  wire next_r1;
  reg  r1;

  wire [7:0] m1;
  wire [10:0] a1, a2, aq; // 16-bit worst case
  reg [10:0] aq_reg;

  always @(posedge clk) aq_reg = aq;

  assign aq = a2 + a1;
  assign a1 = m1 * in;
  assign m1 = (step == 0) ? c1 :
              (step == 1) ? c2 : c3;
  assign a2 = (step == 0) ? 0 : aq_reg;
  assign q = aq_reg;
endmodule
```

Input is now sequential: in1, in2, in3 can share an input port

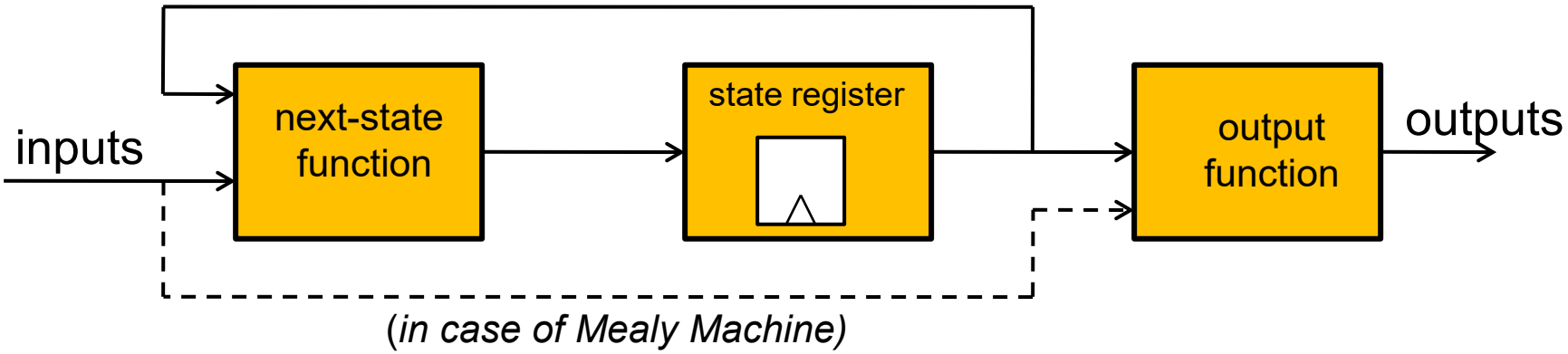
Result of 'Muxed Datapath'



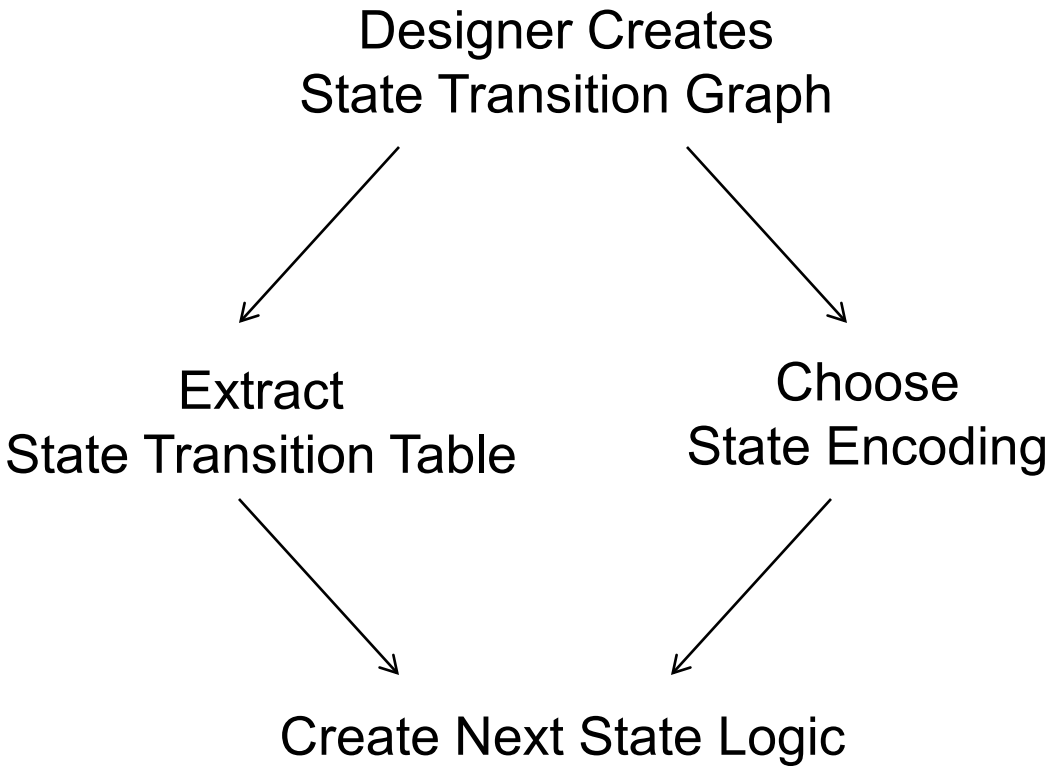
Verilog FSM-based Control

Finite State Machine Template

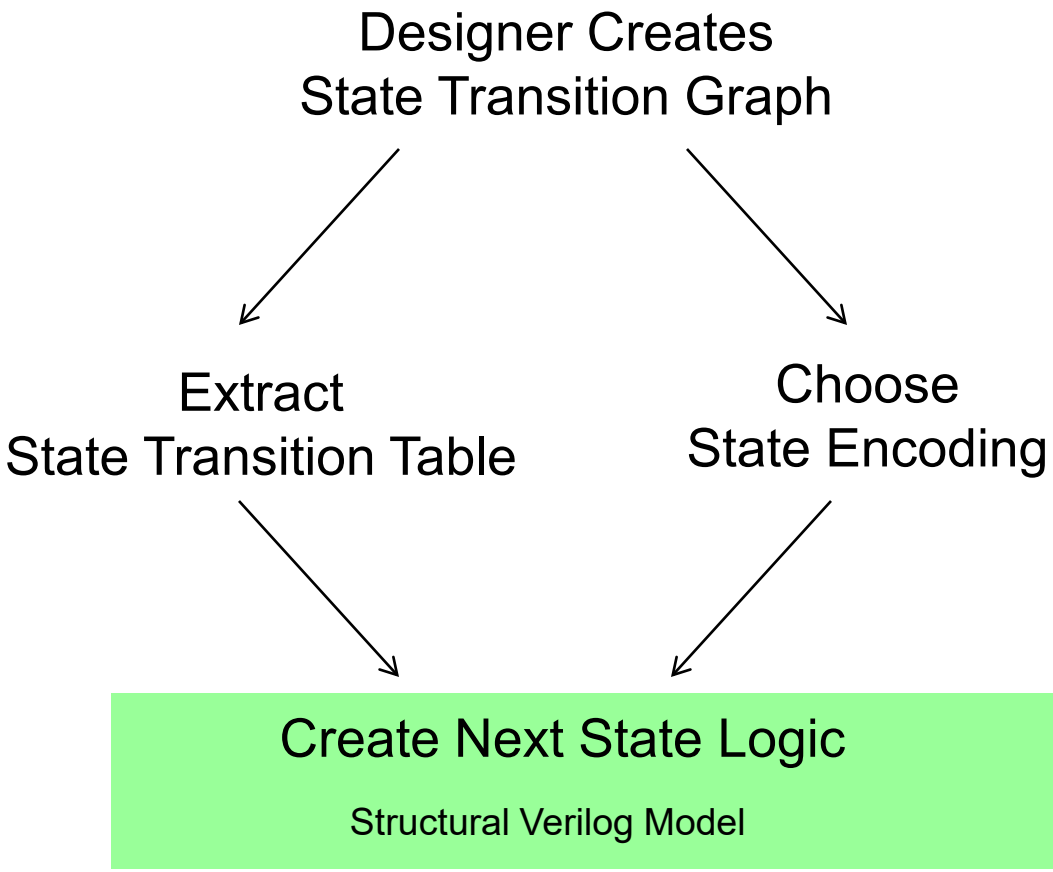
- Sequential Machine defined by
  - Set of Inputs and Outputs
  - Set of States
  - Initial State (reset function)
  - State Transitions



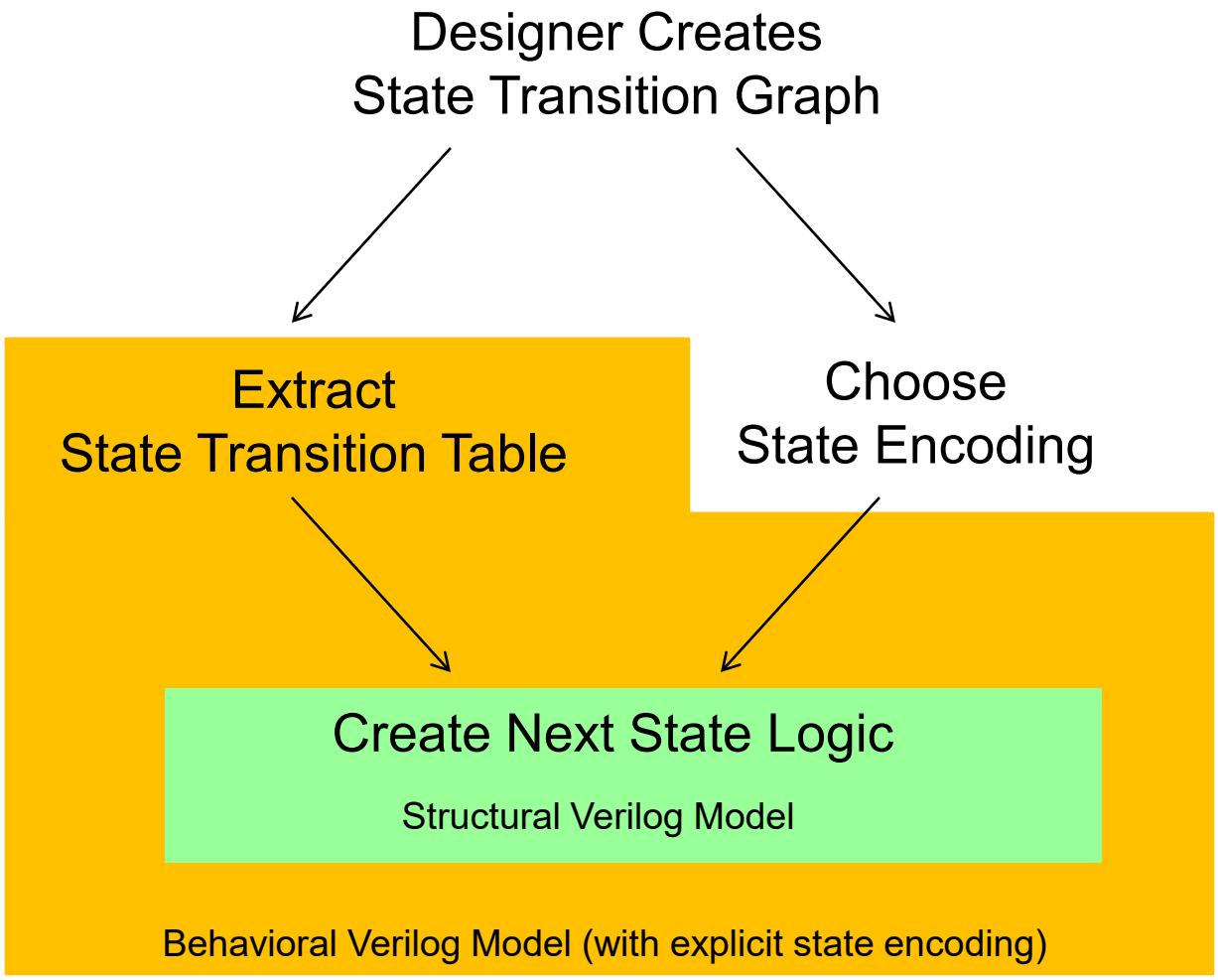
Design Process



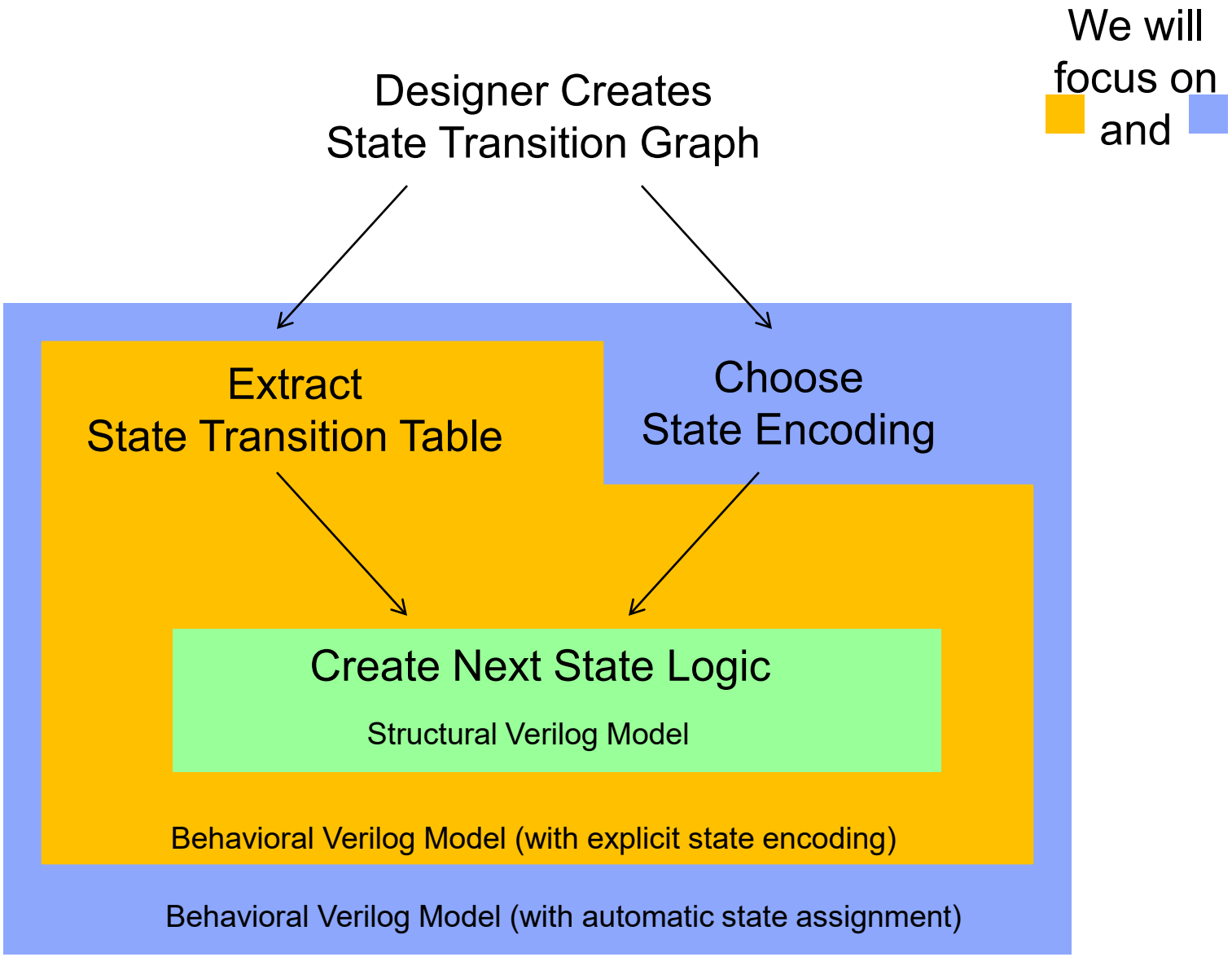
Design Process



Design Process



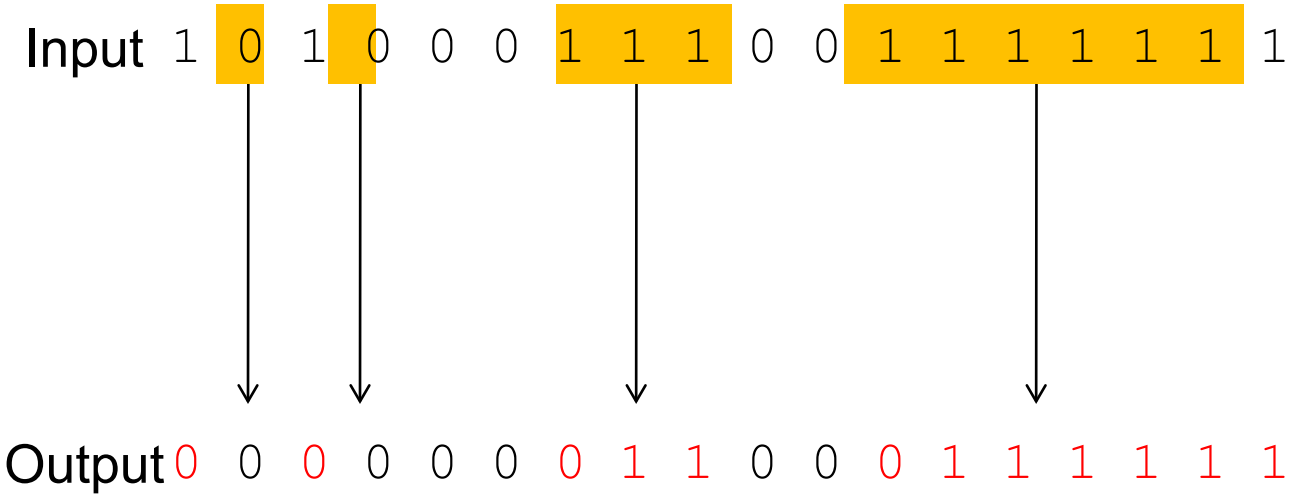
Design Process



Example FSM Design

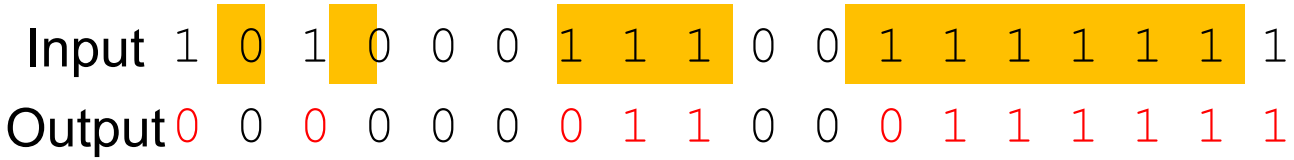
- *Create an FSM that turns the first '1' of a string of consecutive '1' into a '0'.*

Each clock cycle, a new input is provided



Example FSM Design

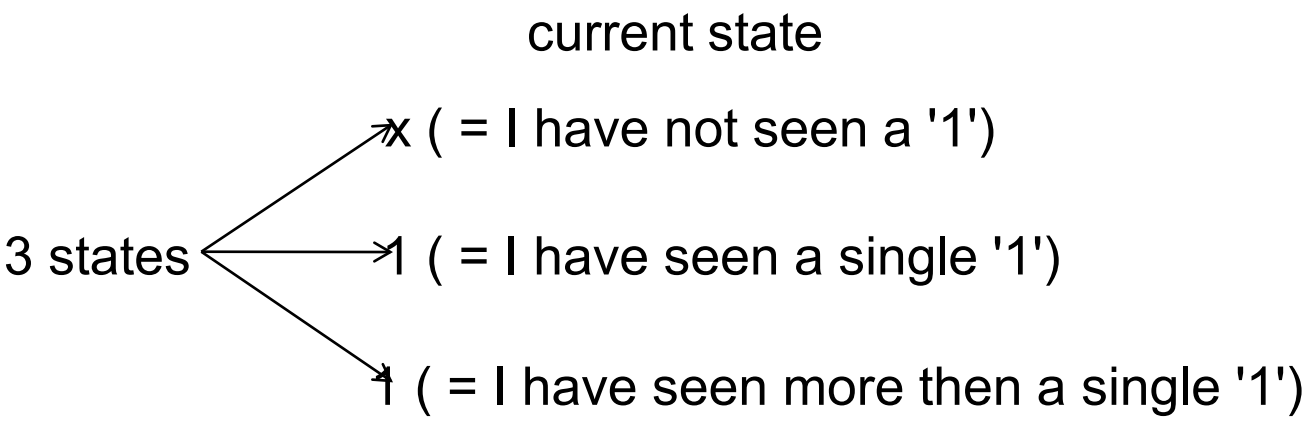
- How many states?



Example FSM Design

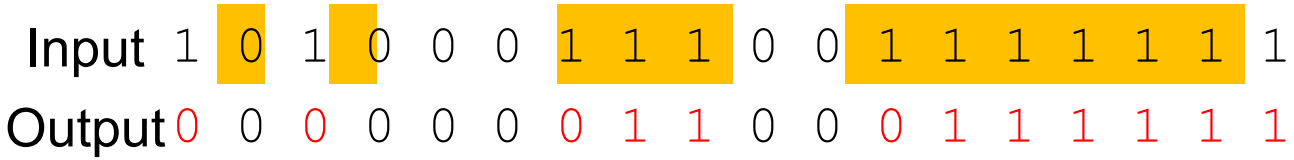
- How many states?

Input	1	0	1	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1
Output	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1	1	1



Example FSM Design

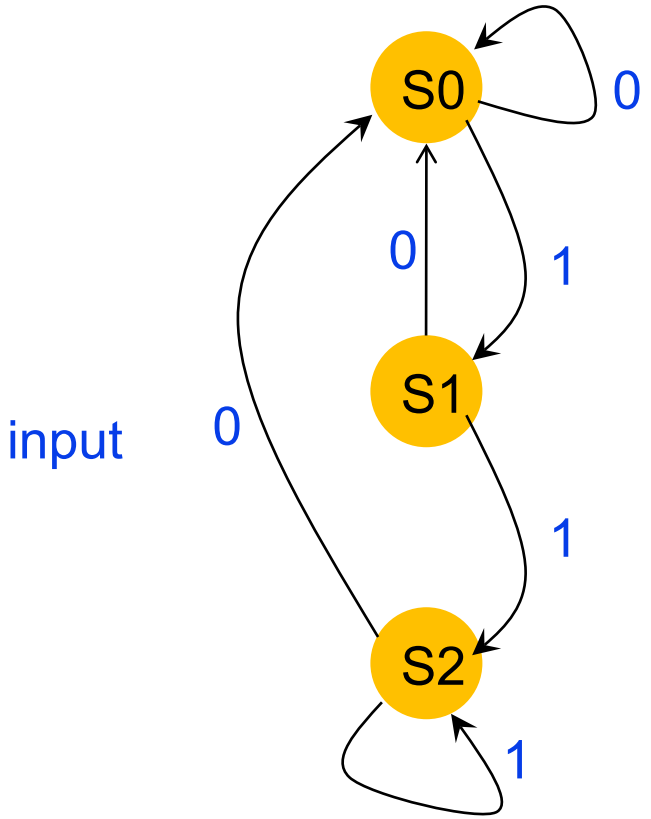
- How many states?



Example FSM Design

- How many states?

Input	1	0	1	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1
Output	0	0	0	0	0	0	0	1	1	0	0	0	1	1	1	1	1	1



output
S0 -> 0
S1 -> 0
S2 -> 1

Verilog Mapping of FSM - Generic Ideas

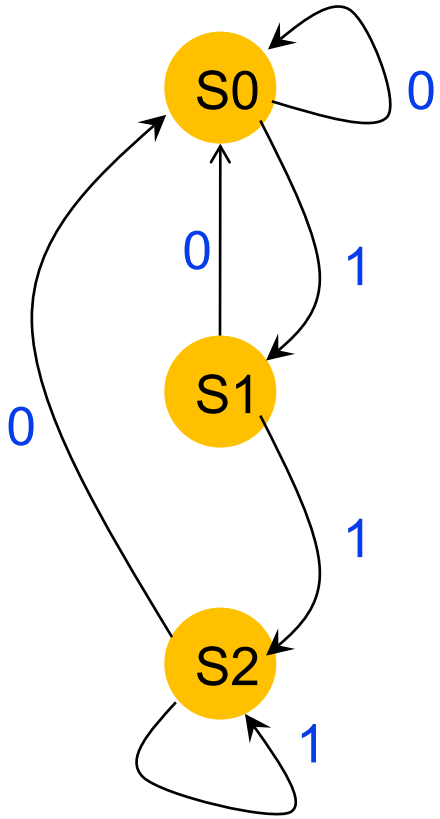
- Use 'parameter' to express state encoding

```
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;
```

- This is a *symbolic* encoding, not necessarily the encoding that will be used by the synthesis tool
- Use standard conventions for coding logic
  - Use non-blocking <= for registers, specify correct reset and edge-triggered behavior
  - Use blocking = for combinational logic, make sure sensitivity lists are complete

Example FSM Design - Verilog Mapping I

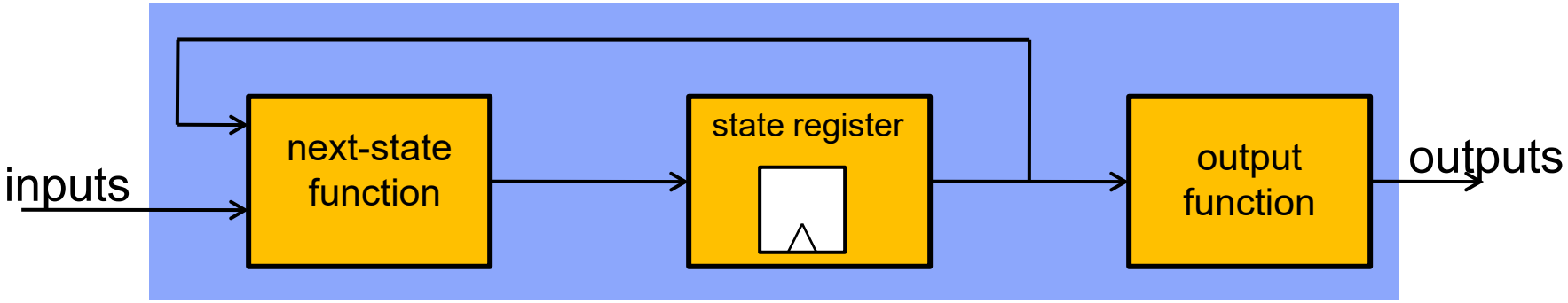
```
module fsm(q, i, clk, rst);
  input i, clk, rst;
  output q;
  reg q;
  reg [1:0] state;
  parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;

  always @(posedge clk or posedge rst)
    if (rst) begin
      state <= s0; q <= 1'b0;
    end else begin
      case (state)
        s0: if (i == 1'b1) begin
              state <= s1;
              q     <= 1'b0;
            end else begin
              state <= s0;
              q     <= 1'b0;
            end
        s1: ..
        s2: ..
      endcase
    end
endmodule
```



output
S0 -> 0
S1 -> 0
S2 -> 1

Example FSM Design - Verilog Mapping I

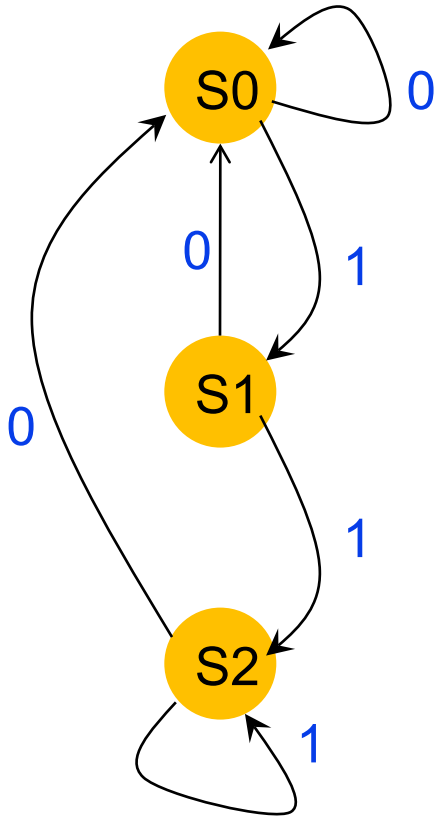
- One single always block
  - Next-state logic and state update in same block
  - Output will always have a register



Example FSM Design - Verilog Mapping II

```
reg q;
reg [1:0] state;
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;

always @(posedge clk or posedge rst)
  if (rst)
    state <= s0;
  else
    case (state)
      s0: if (i == 1'b1)
            state <= s1;
          else
            state <= s0;
      s1: ..
    endcase
```

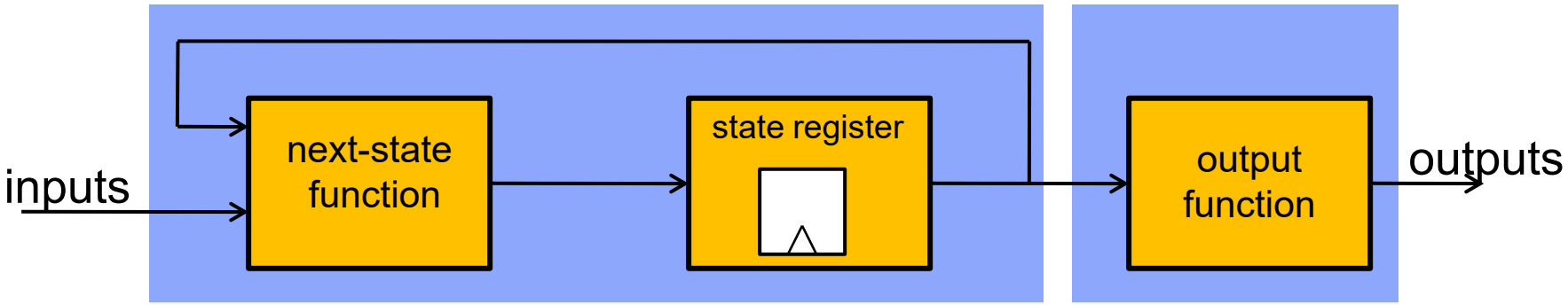
```
always @(state)
   q = 1'b0;    // default assignment
   case (state)
     s0: q = 1'b0;
     s1: q = 1'b0;
     s2: q = 1'b1;
   endcase
endmodule
```



output
S0 -> 0
S1 -> 0
S2 -> 1

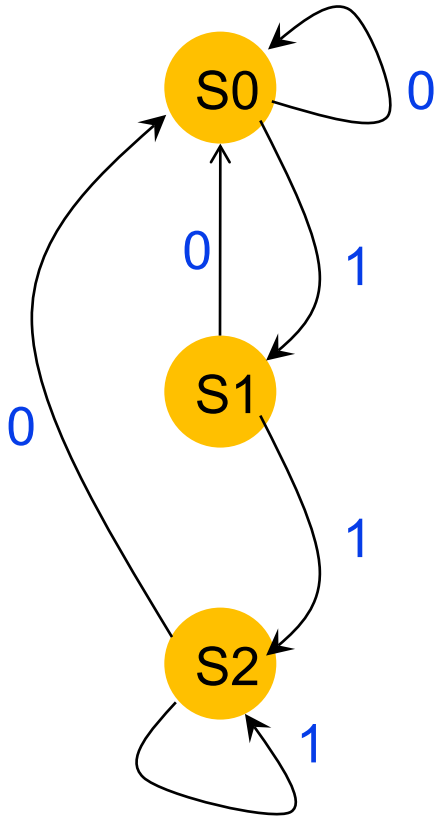
Example FSM Design - Verilog Mapping II

- Two always block
  - Next-state logic and state update in same block
  - Output can be combinational



Example FSM Design - Verilog Mapping III

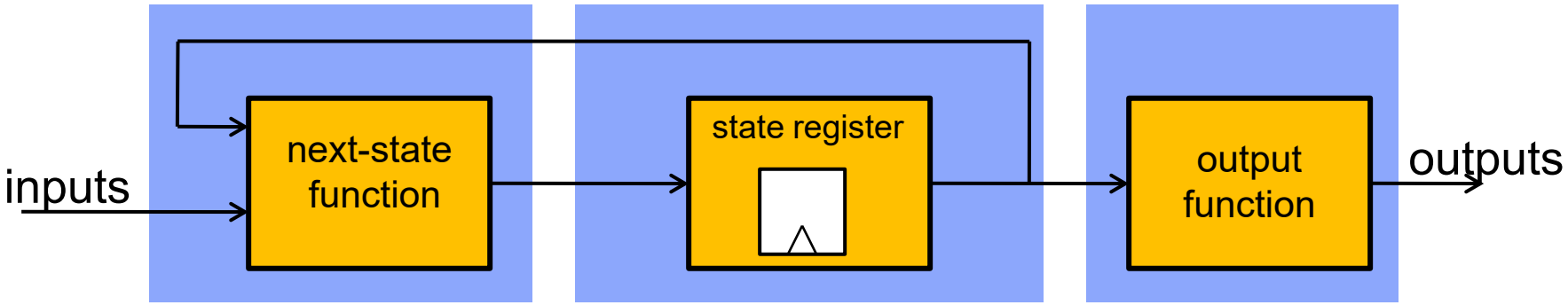
```
reg q;
reg [1:0] state, next_state;
parameter s0 = 2'b00, s1 = 2'b01, s2 = 2'b10;

always @(posedge clk or posedge rst)
  if (rst)
    state <= s0;
  else
    state <= next_state;

always @(state or i) begin
    // next state encoding
    next_state = s0;
    case (state)
      s0: if (i == 1'b1)
            next_state = s1;
          else
            next_state = s0;
      s1: ..
    endcase
end

always @(state)
  // output encoding
endmodule
```



output
S0 -> 0
S1 -> 0
S2 -> 1

Example FSM Design - Verilog Mapping III

- Three always block
  - Next-state logic and state update in same block
  - Output can be combinational



One, two, three always block

	1 always block	2 always block	3 always block	
Combinational Output		√	√	
State Update separate from next-state Logic			√	