ECE 4514 Digital Design II Spring 2008

Lecture 8: Multiplexed Datapaths

A *Methods* Lecture

Patrick Schaumont

Today's topic

- An design method that uses dataflow modeling
- A *Design Method*
  - A canned sequence of steps, commands
  - Is easy to learn, easy to remember, easy to apply
  - Allows a designer to concentrate on *design* rather than on tools
- The method we will discuss is called

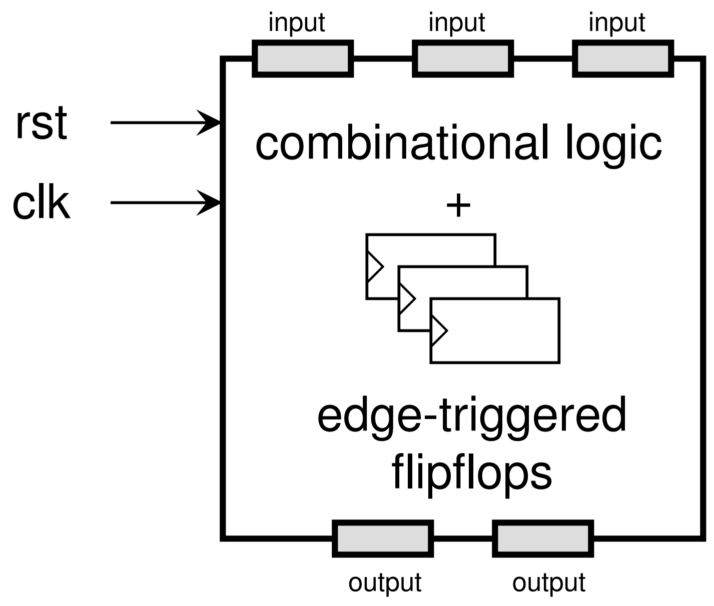
Multiplexed Datapaths

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 ?

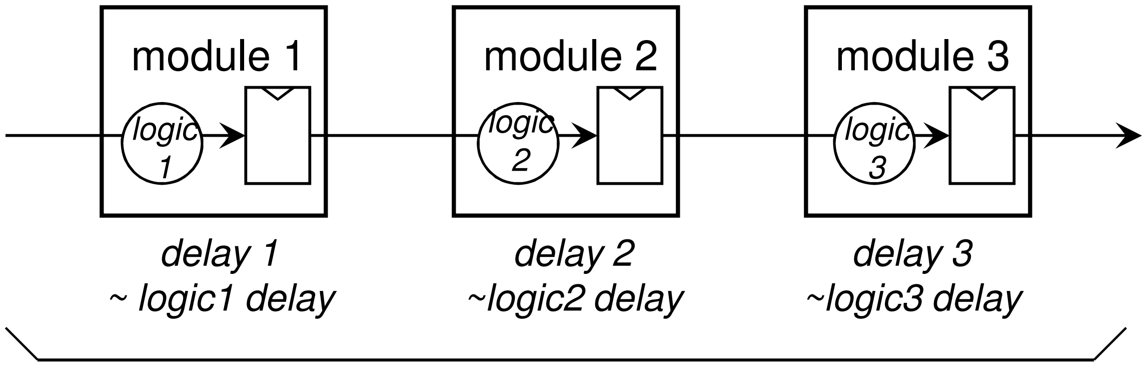
- A module with a single clk and rst input
- Zero or more inputs, one or more outputs
- Edge-triggered flip-flops
- Outputs depend *only* on registers

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

Why outputs depend only on registers ?

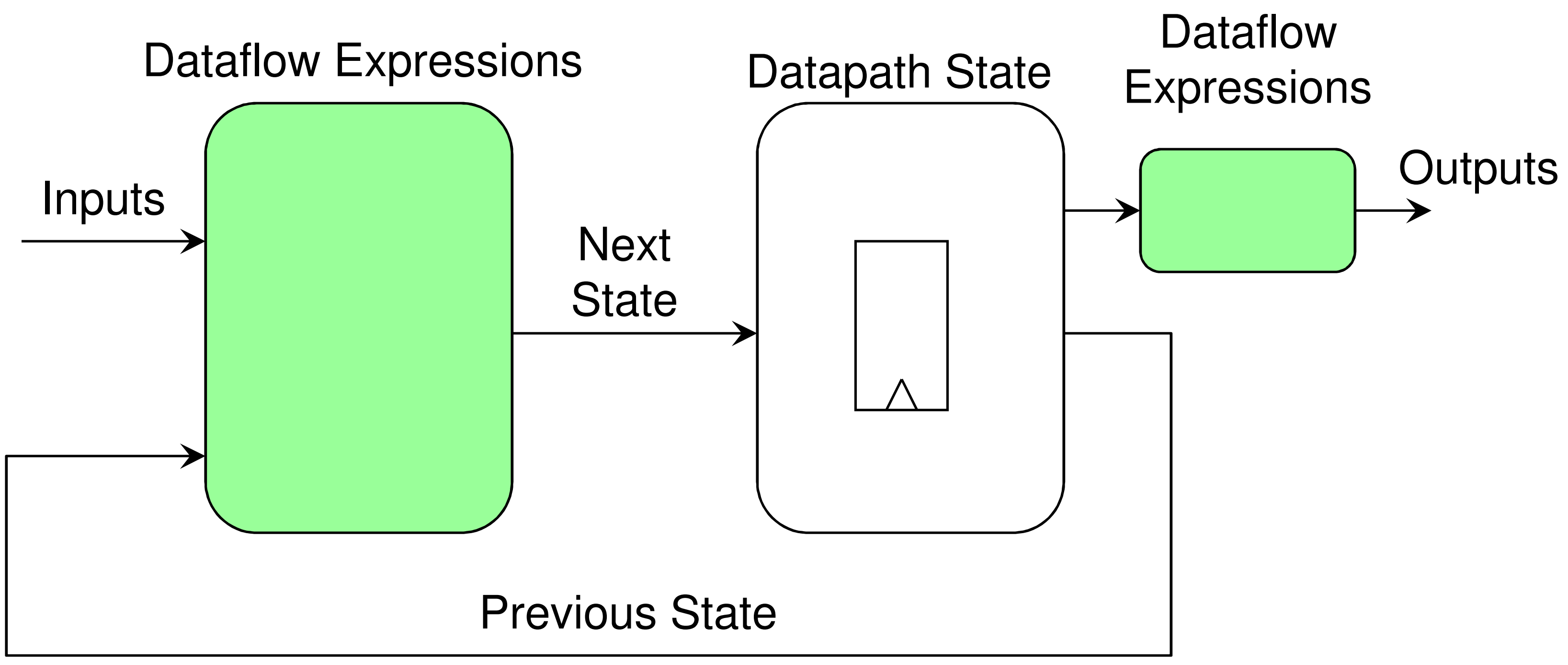
- A module with a single clk and rst input
- Zero or more inputs, one or more outputs
- Edge-triggered flip-flops
- Outputs depend *only* on registers

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



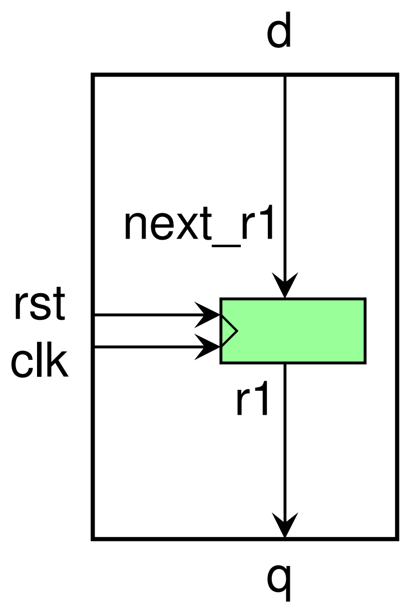
System Delay = max(delay1, delay2, delay3) + routing_delay

Template for a Multiplexed Datapath



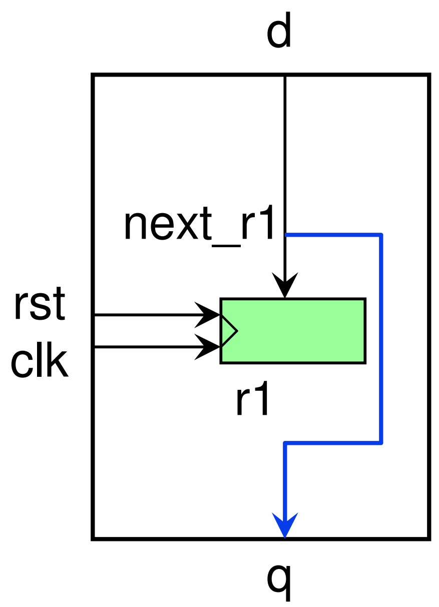
How to model an edge-triggered register

```
module a_module(q, rst, clk, d);
  output q;
  input rst;
  input clk;
  input 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

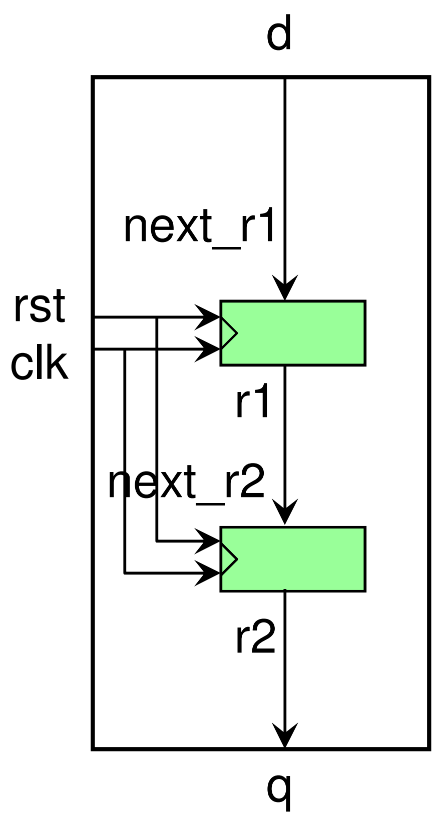
```
module a_module(q, rst, clk, d);
  output q;
  input rst;
  input clk;
  input d;
  wire next_r1;
  reg  r1;

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

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



No!

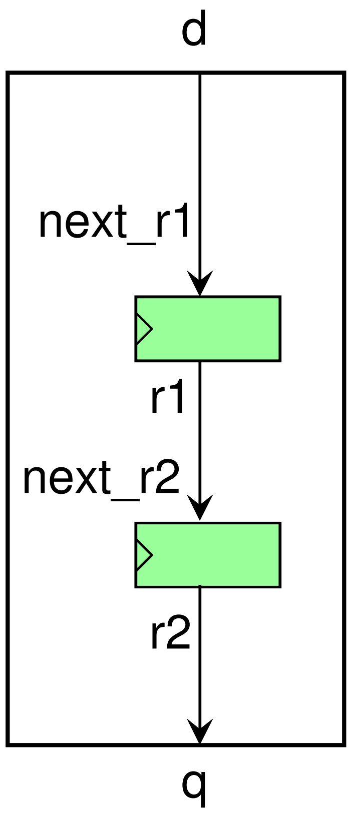
How to model two edge-triggered registers

```
module a_module(q, rst, clk, d);
  output q;
  input rst;
  input clk;
  input 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

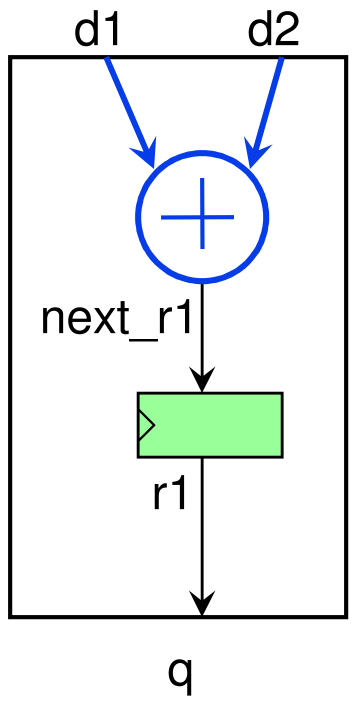
```
module a_module(q, rst, clk, d);
  output q;
  input rst;
  input clk;
  input d;
  wire next_r1, next_r2;
  reg  r1, r2;

  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

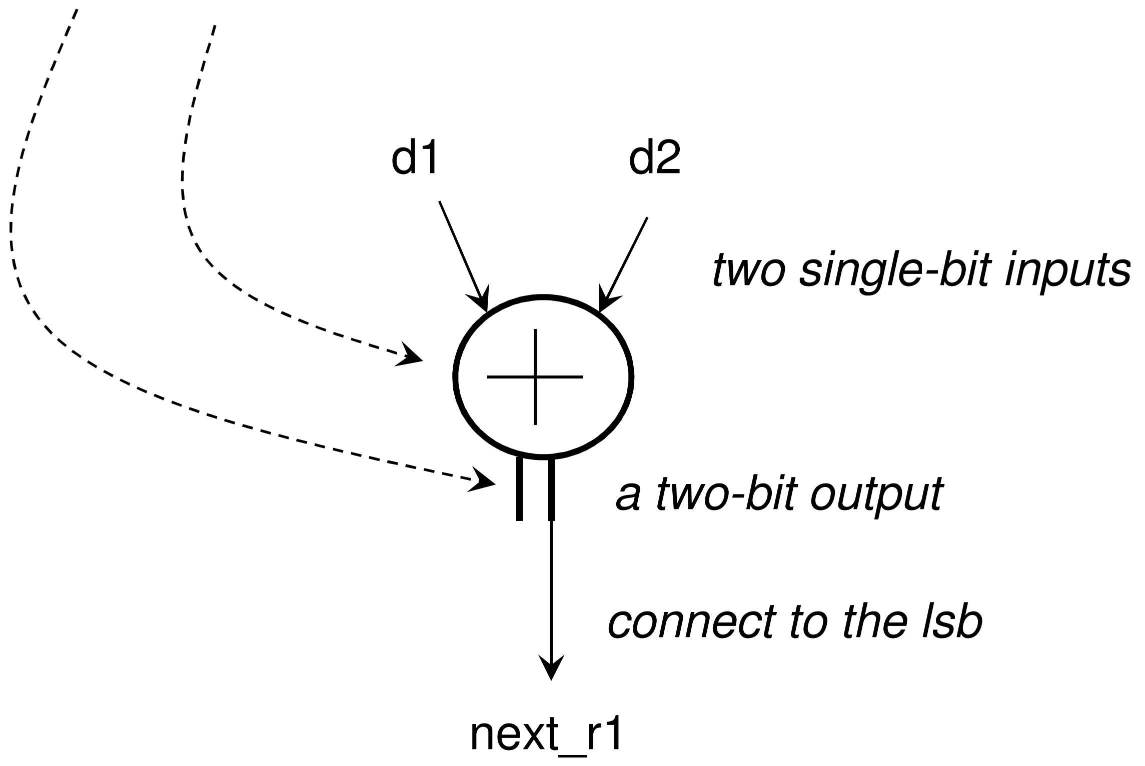
Add two inputs, capture in a register

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



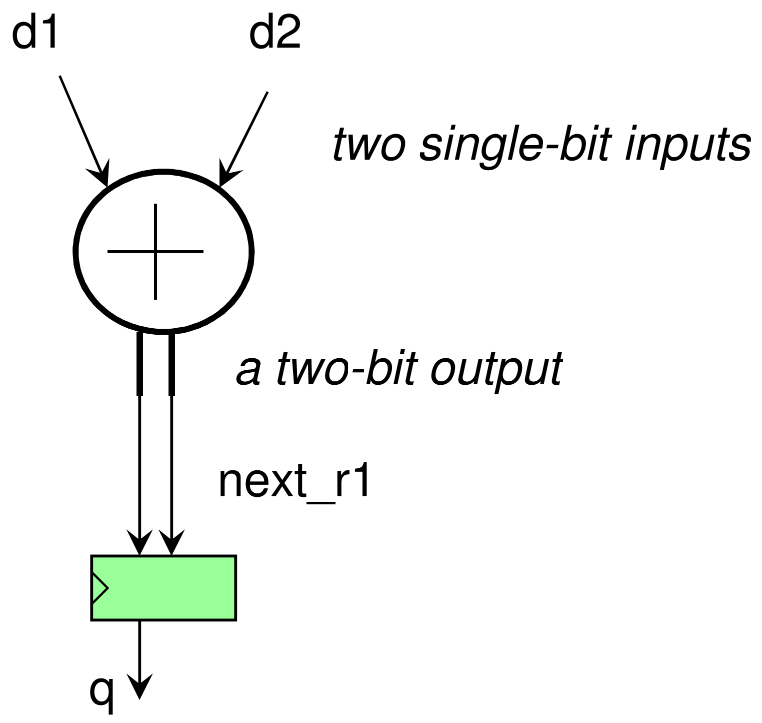
Take a close look at the addition

```
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];
```



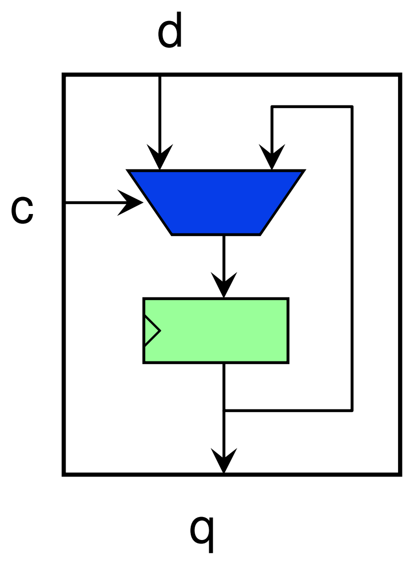
Not so good, will require a two-bit register

Better solution

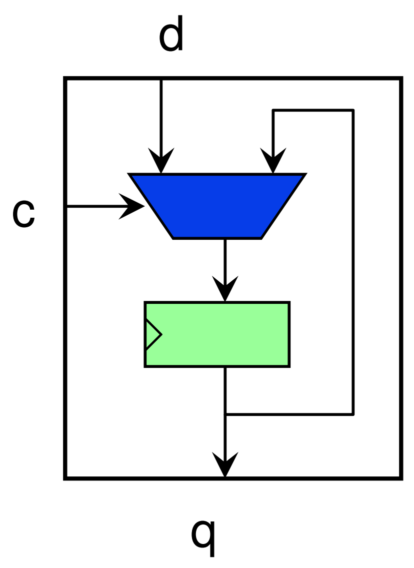
```
module get_carry(q, rst, clk, d1, d2);
  output q;
  input rst;
  input clk;
  input d1, d2;

  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

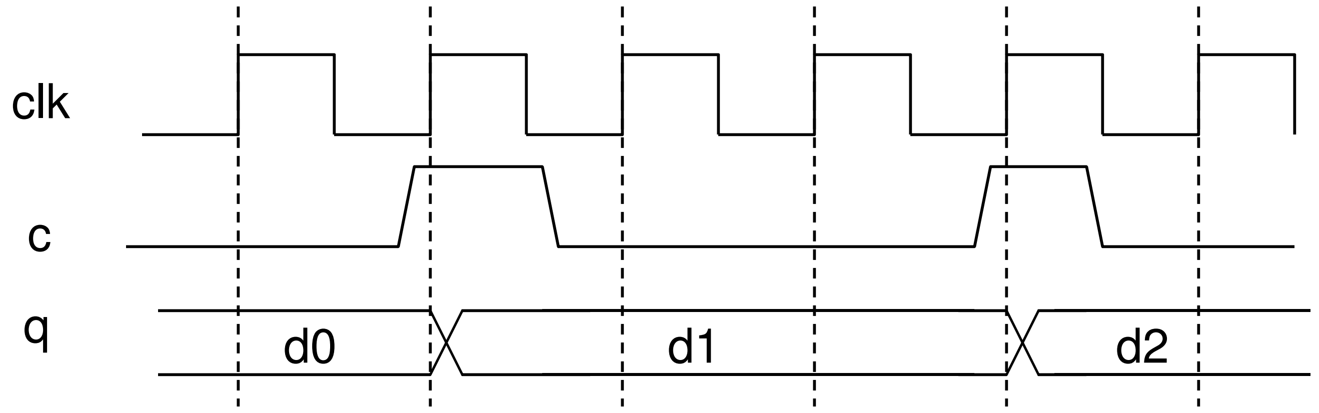
```
module a_module(q, rst, clk, d, c);
  output q;
  input rst;
  input clk;
  input d, c;
  wire next_r1;
  reg  r1;

  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

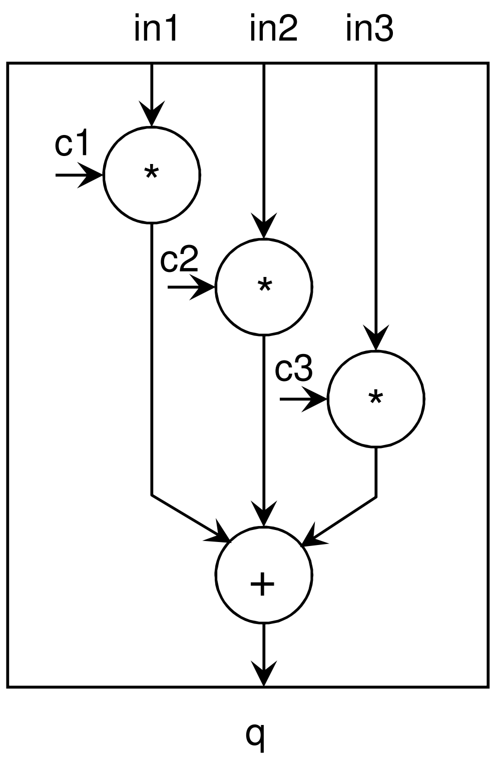


- Eg. d is only valid once every three clock cycles
- Capture d in register only when c = 1
- Otherwise, retain value of q
- c is a 'logical' clock signal (in contrast to the physical clock signal clk)

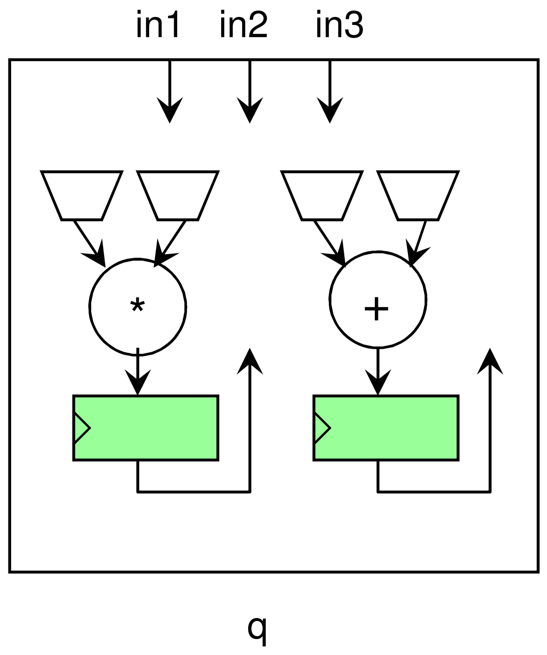


Muxed registers are key to Muxed Datapath

- Overall Idea



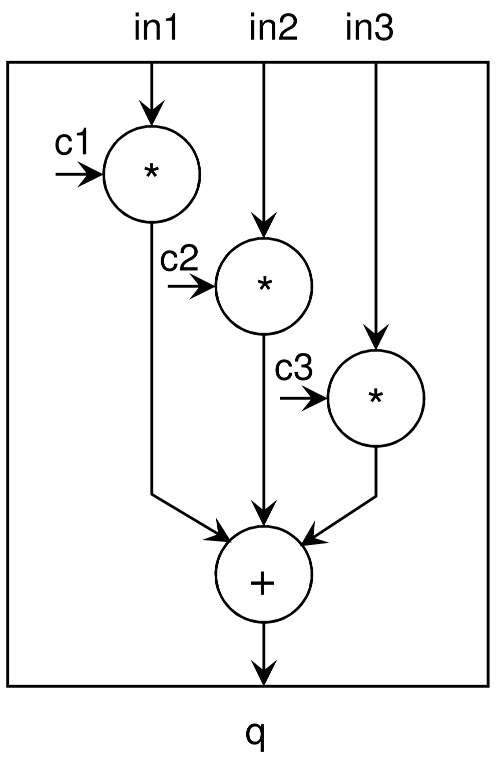
One clock cycle
3 Multiplier
2 Adders (2-input)

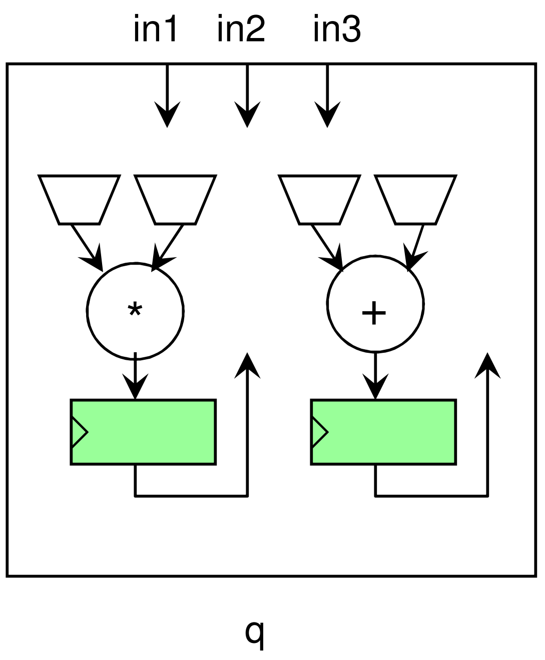


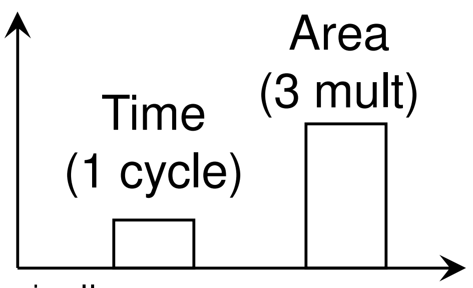
Three clock cycles
1 Multiplier
1 Adder
n Multiplexers

Muxed registers are key to Muxed Datapath

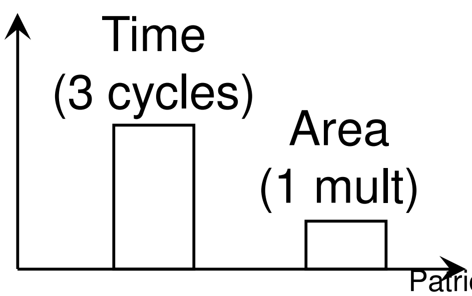
- Overall Idea





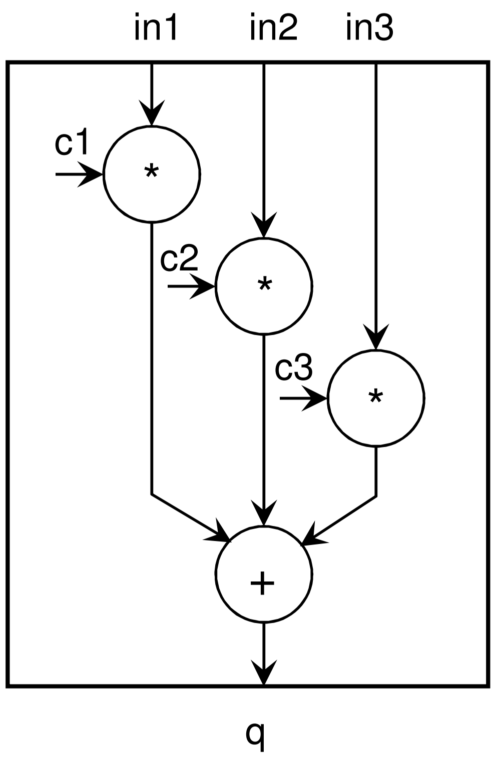


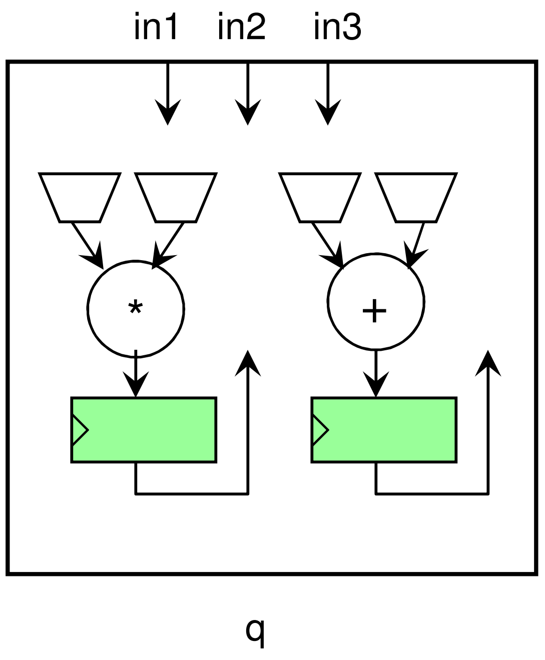
AREA TIME TRADEOFF



Muxed registers are key to Muxed Datapath

- Overall Idea

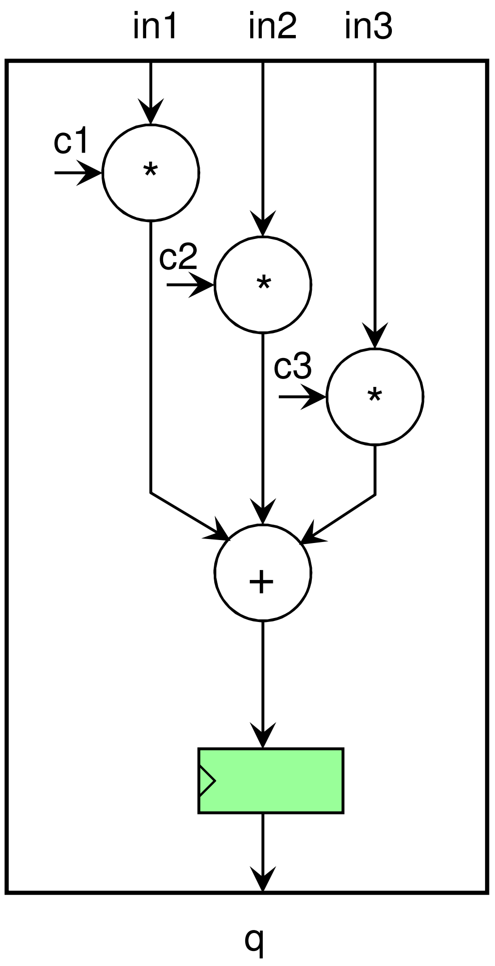




Pitfall #1: Time doesn't scale linearly
Time = cycles * period
Period ~ Critical Path
Critical Path$_{\text{non-muxed}}$ >> Critical Path$_{\text{muxed}}$

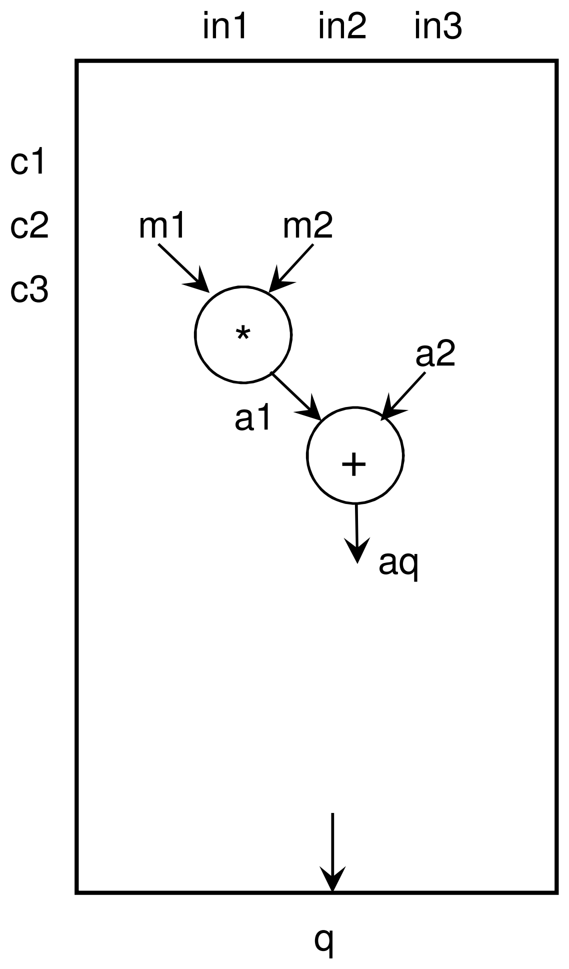
Pitfall #2: Area doesn't scale linearly
Area = Operations + Overhead
Overhead ~ muxes + regs
#-Muxes$_{\text{muxed}}$ >> #-Muxes$_{\text{non-muxed}}$

Let's try this



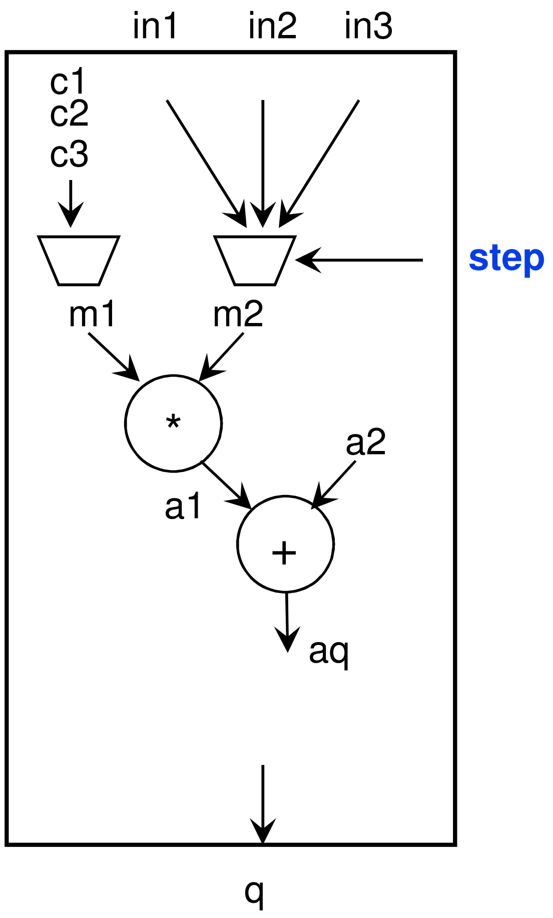
```
module vecmul(q, rst, clk, in1, in2, in3);
  output [7:0] q;
  input rst, rst;
  input [7:0] in1, in2, in3;
  parameter c1 = 8'd2;
  parameter c2 = 8'd4;
  parameter c3 = 8'd6;

  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
```

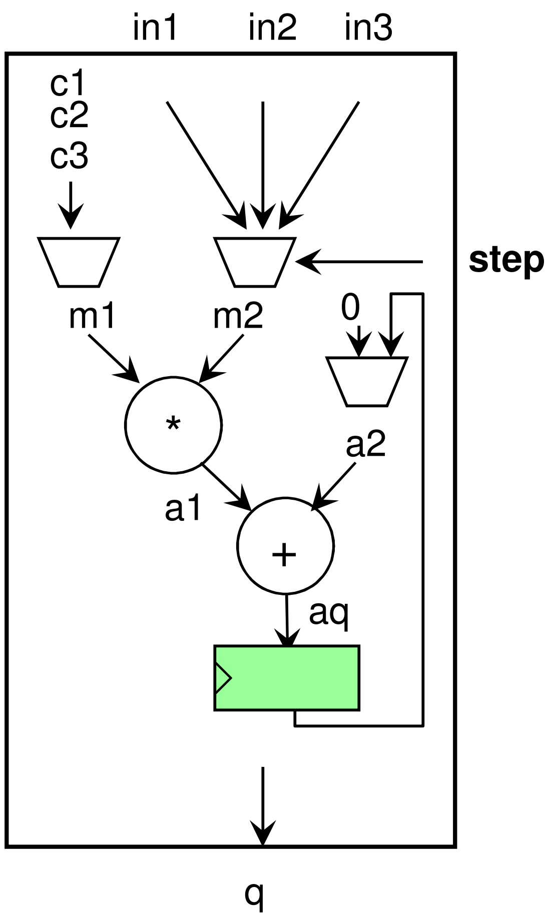
Add muxes on m1, m2. Add control input.



```
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
```

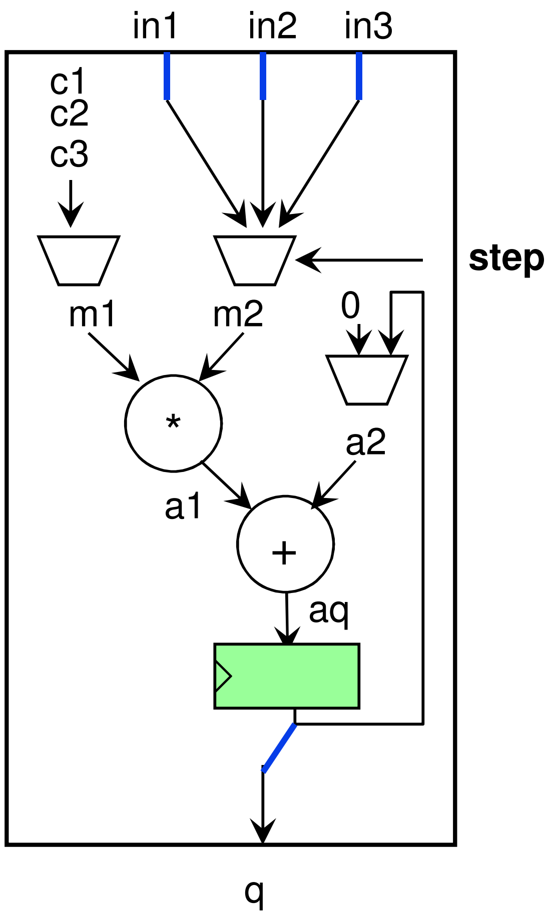
step is a control input

Add muxes on a2. Add accumulator reg.



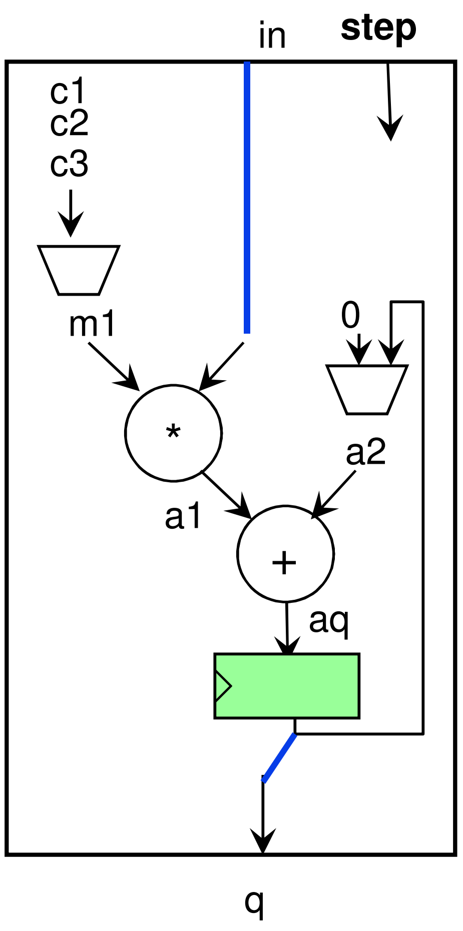
```
module vecmul(q, rst, clk, in1, in2, in3, step);
  output [7:0] q;
  input rst, rst;
  input [7:0] in1, in2, in3;
  input [1:0] step;
  parameter c1 = 8'd20;
  parameter c2 = 8'd40;
  parameter c3 = 8'd60;

  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
```

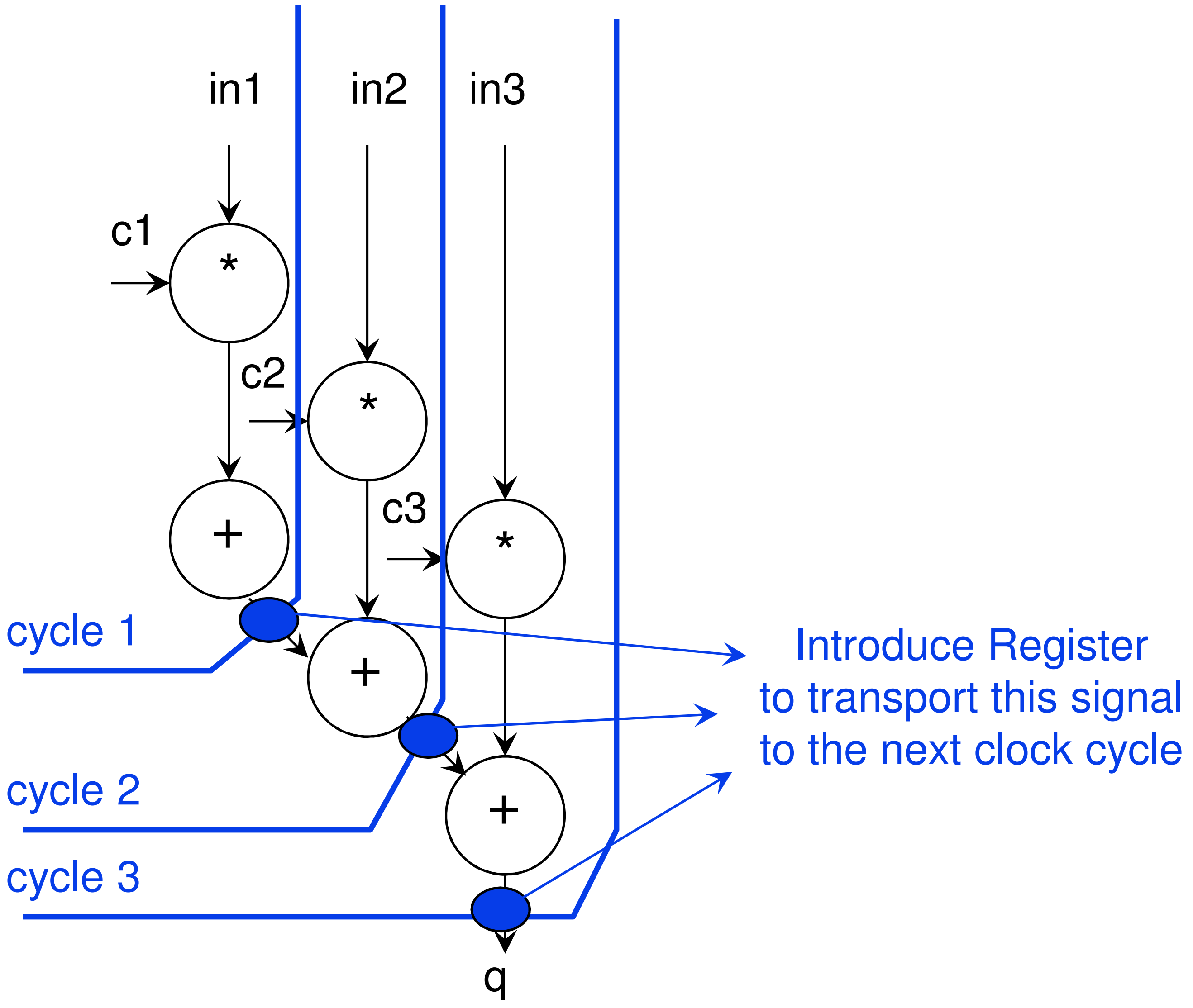
Simple Optimizations



```
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'


in1
in2
in3
c1
c2
c3
*
*
*
+
+
+
cycle 1
cycle 2
cycle 3
q
Introduce Register to transport this signal to the next clock cycle

Automatic Generation of Muxed Datapaths

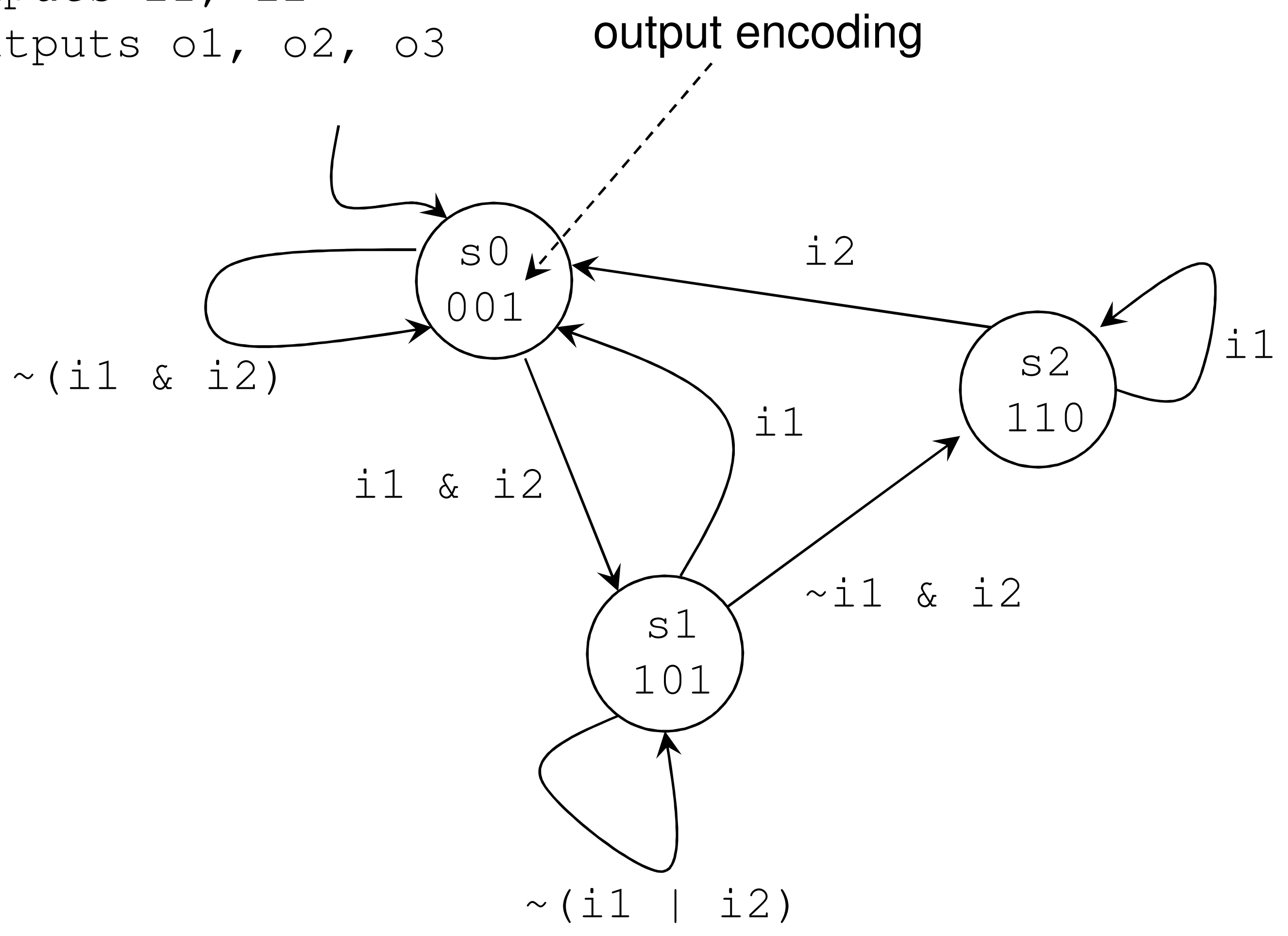
- There exists elaborate theory that will find the 'best' way of partitioning an operation graph
- It involves
  - Allocation: Deciding how much operators you need to implement all of the operations
  - Assignment: Deciding which operation will be execute on which operator
  - Scheduling: Decide the clock cycle during which an operation must be executed
- But in this course, we will work from you ingenuity

Can we design FSM using same Verilog Style?

- Yes!
- We can generate the *step* signal locally, if we want.
- We can also model an existing FSM using multiplexers and assign expressions.
  - Need to perform state assignment manually
  - Next-state encoding can be done with assign expressions

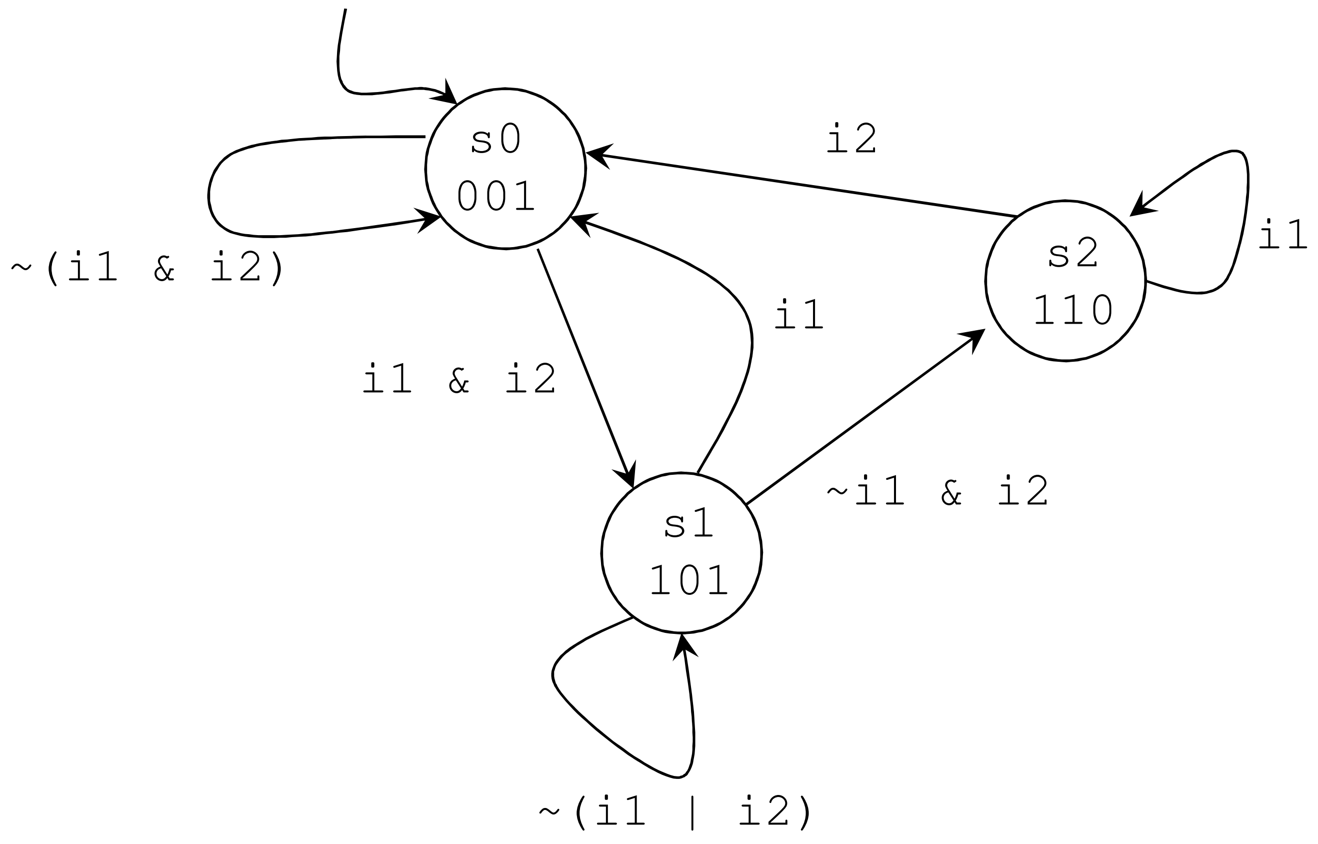
Example: Design this FSM as muxed datapath

```
Inputs i1, i2
Outputs o1, o2, o3
```



Example: Design this FSM as muxed datapath

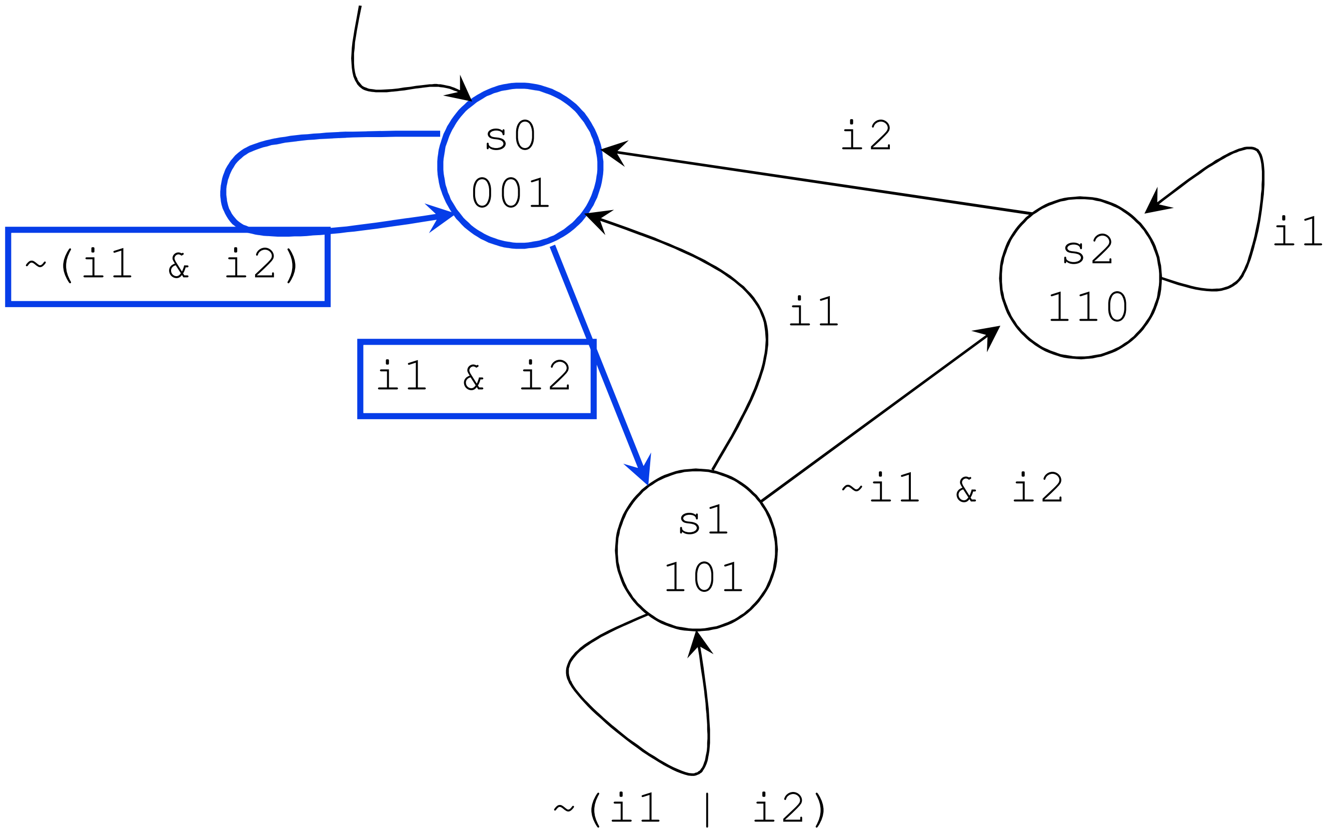
```
Inputs i1, i2
Outputs o1, o2, o3
```



Always verify if state transitions are complete

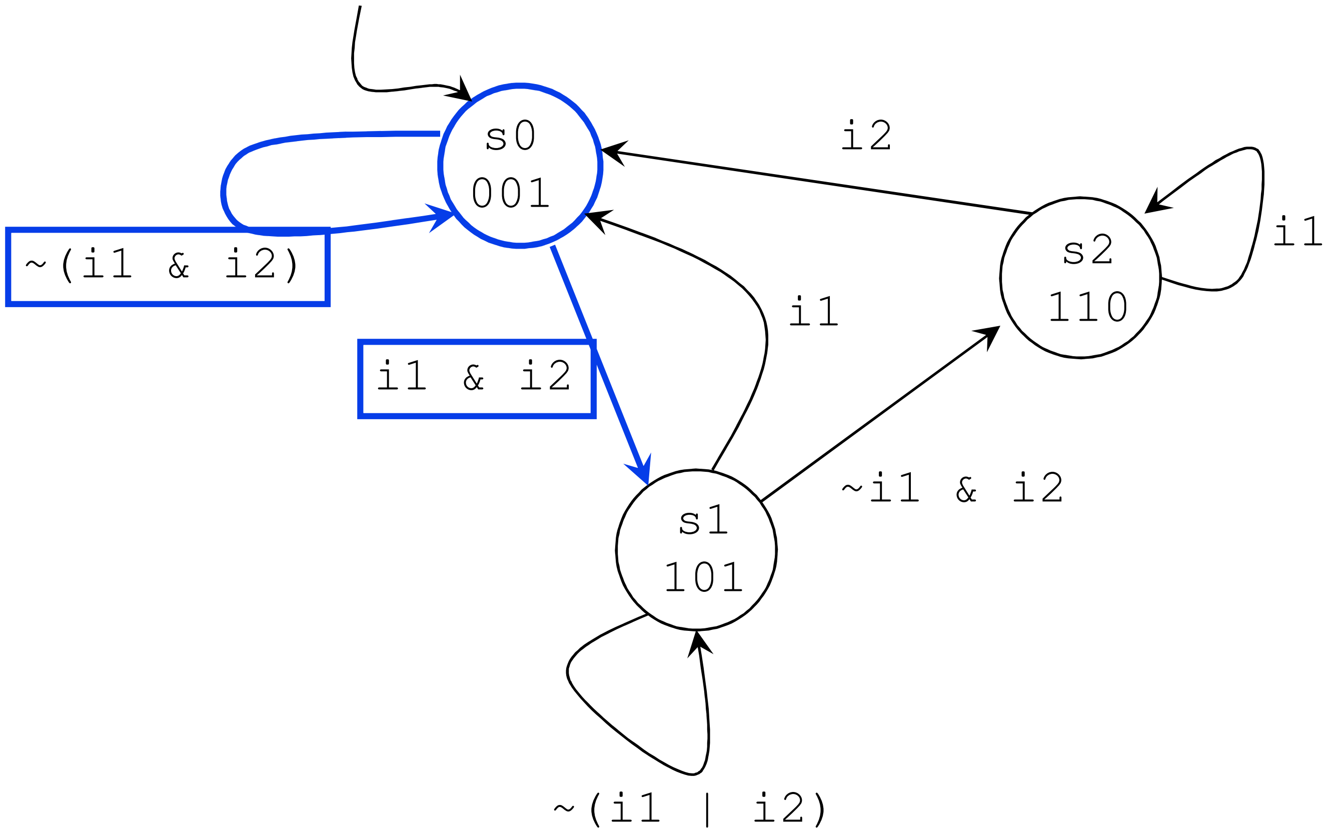
Example: Design this FSM as muxed datapath

```
Inputs i1, i2
Outputs o1, o2, 03
```



Example: Design this FSM as muxed datapath

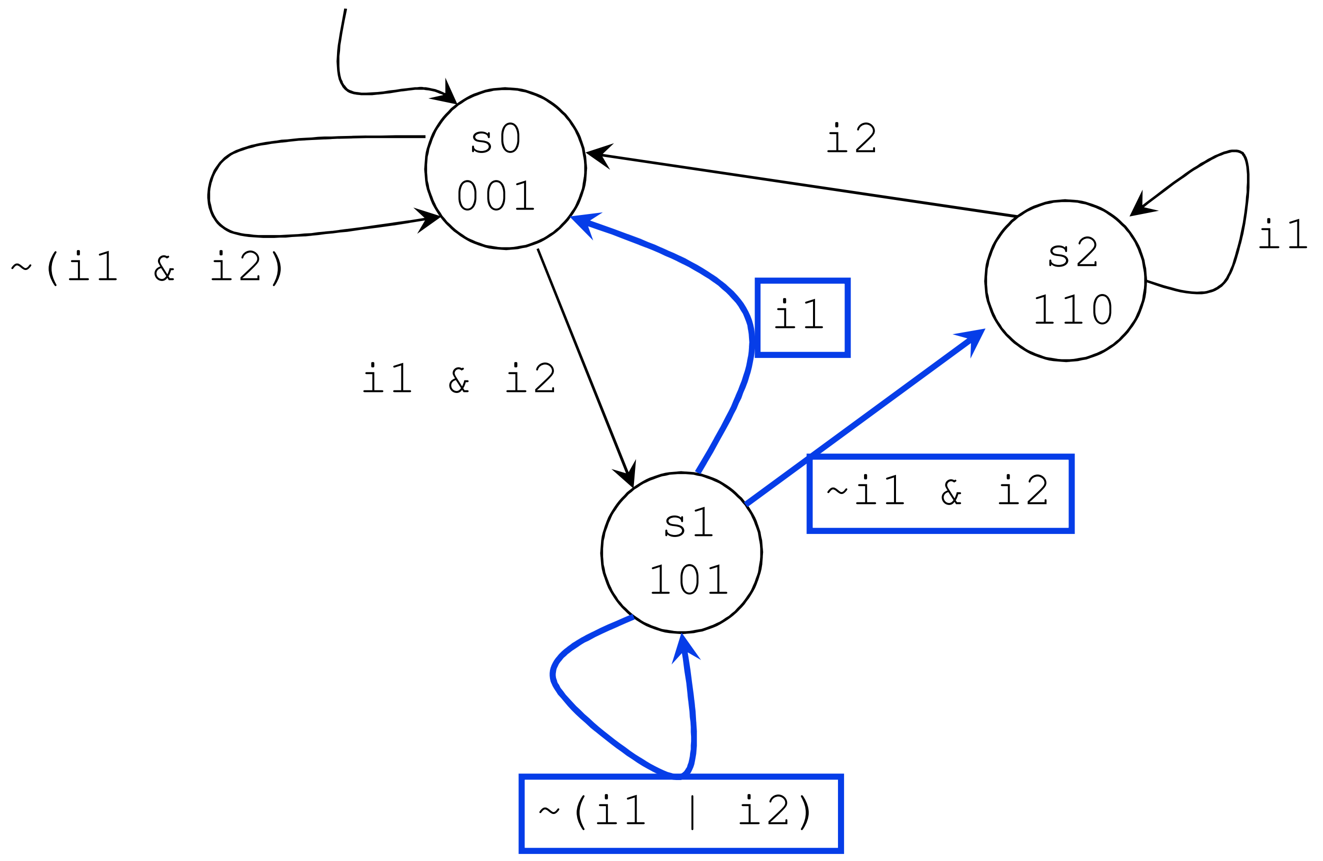
```
Inputs i1, i2
Outputs o1, o2, o3
```



OK, ~(i1 & i2) | (i1 & i2) = 1
and ~(i1 & i2) & (i1 & i2) = 0

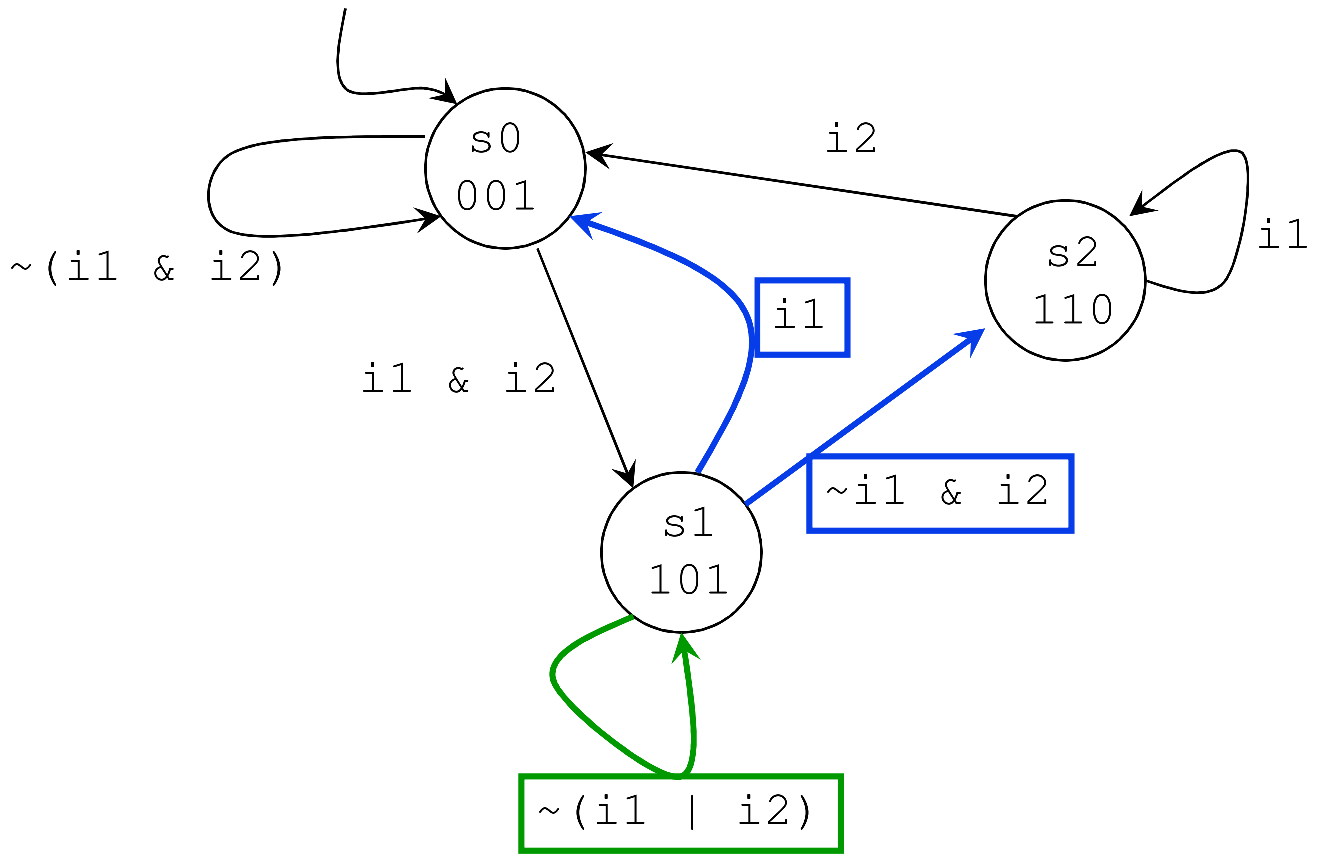
Example: Design this FSM as muxed datapath

```
Inputs i1, i2
Outputs o1, o2, o3
```



Example: Design this FSM as muxed datapath

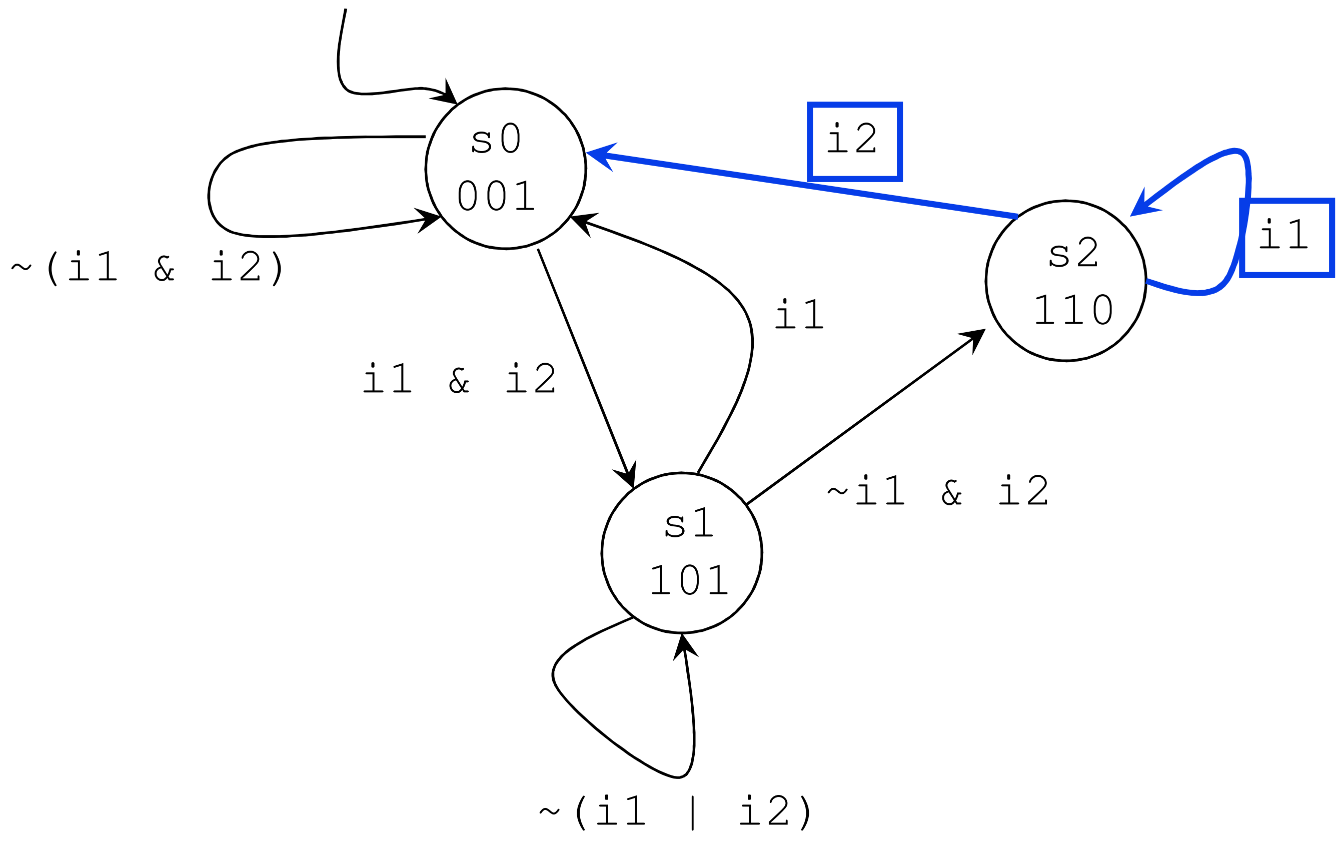
```
Inputs i1, i2
Outputs o1, o2, o3
```



OK, since ~(i1 | i2) = ~i1 & ~i2,
the all transitions are covered exactly

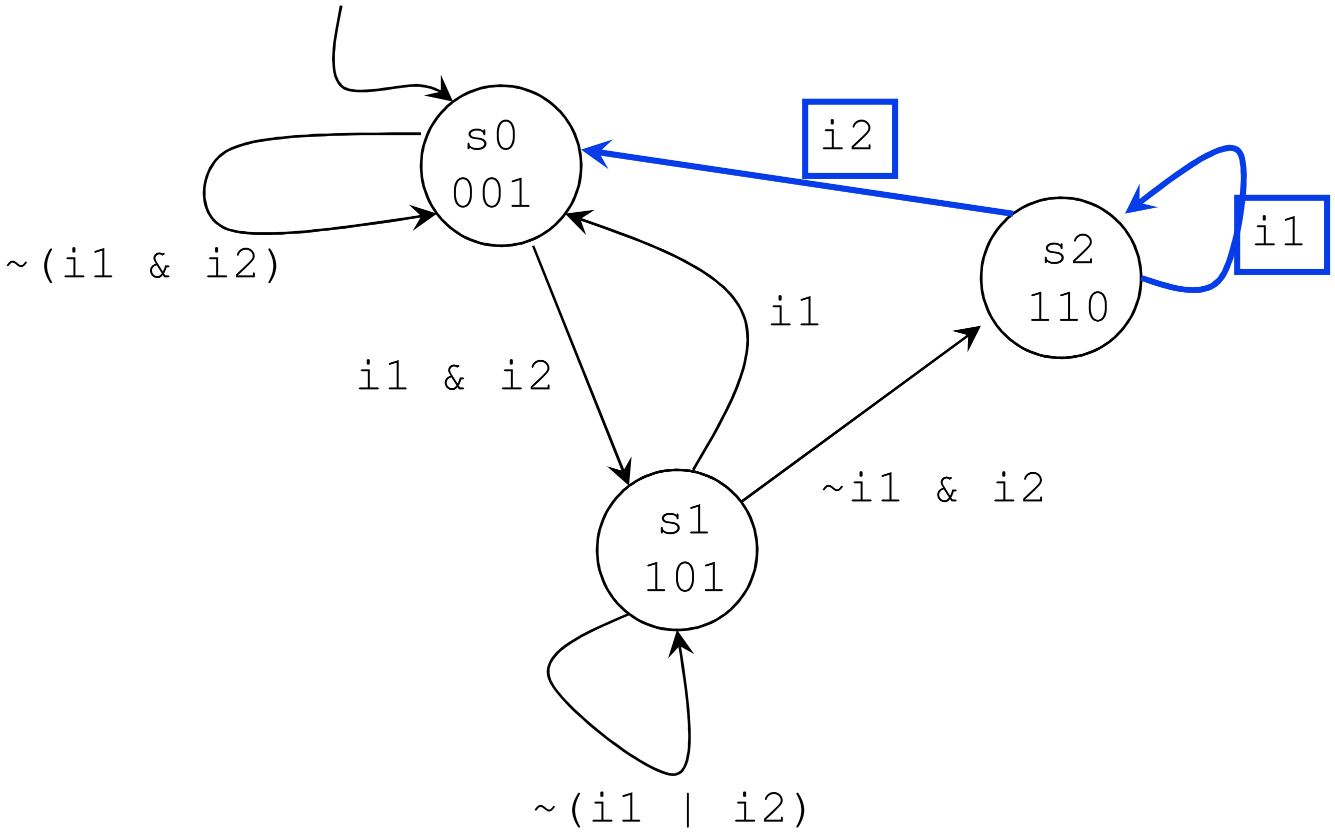
Example: Design this FSM as muxed datapath

```
Inputs i1, i2
Outputs o1, o2, o3
```



Example: Design this FSM as muxed datapath

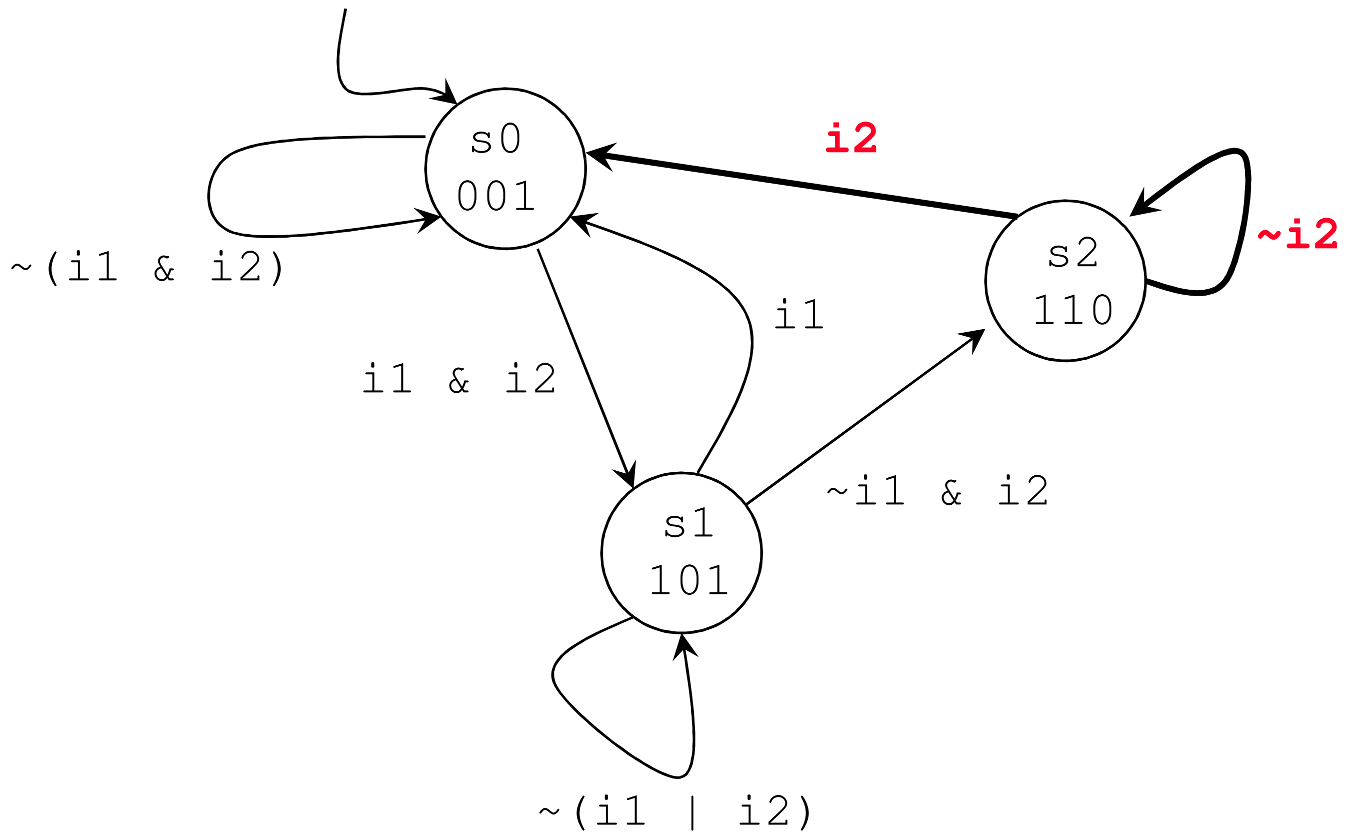
```
Inputs i1, i2
Outputs o1, o2, o3
```



Ambiguous! What if i1 = 1 and i2 = 1?
What if i1 = 0 and i2 = 0 ?

Example: Design this FSM as muxed datapath

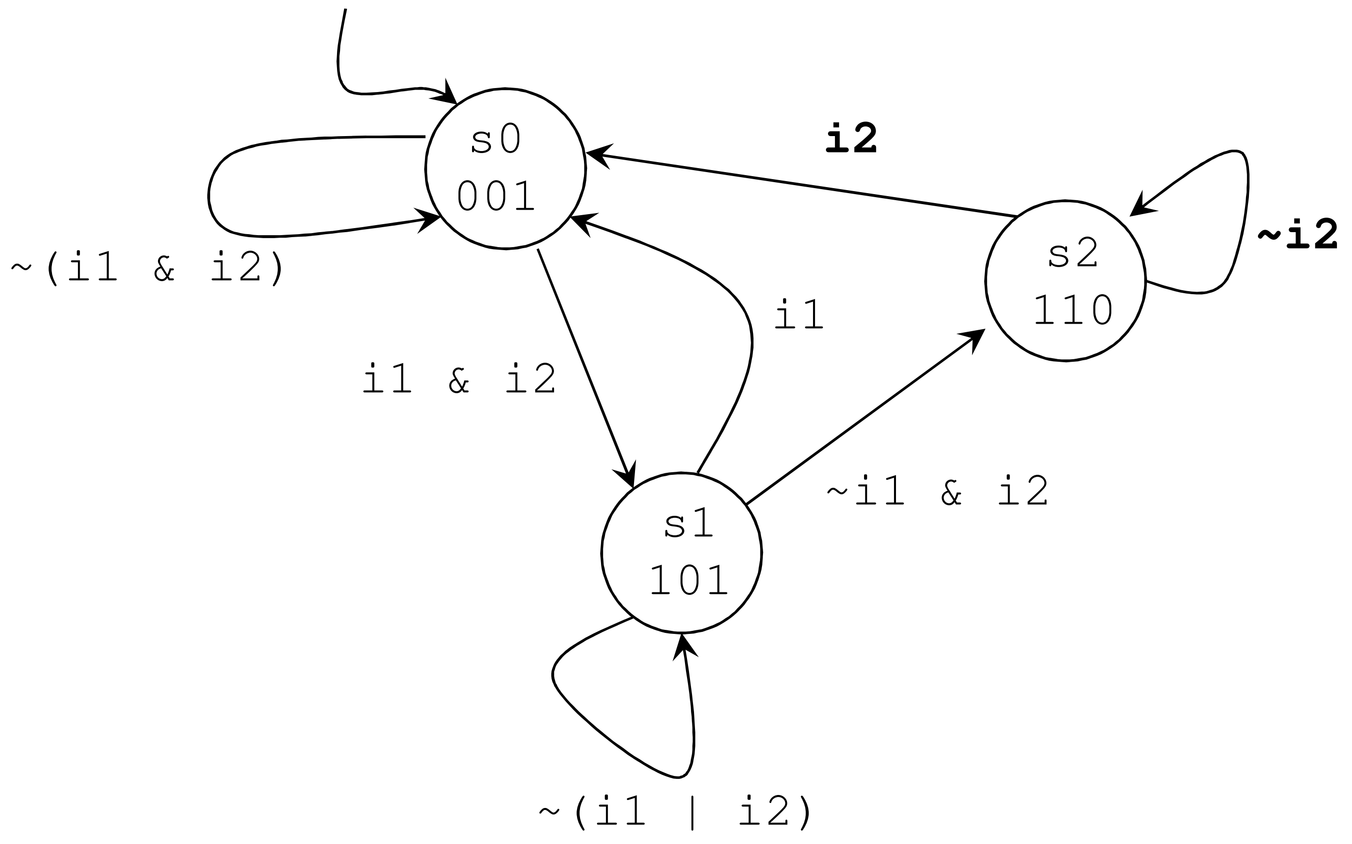
```
Inputs i1, i2
Outputs o1, o2, o3
```



Resolved (by modifying conditions - required change will depend on the application)

Example: Design this FSM as muxed datapath

```
Inputs i1, i2
Outputs o1, o2, o3
```

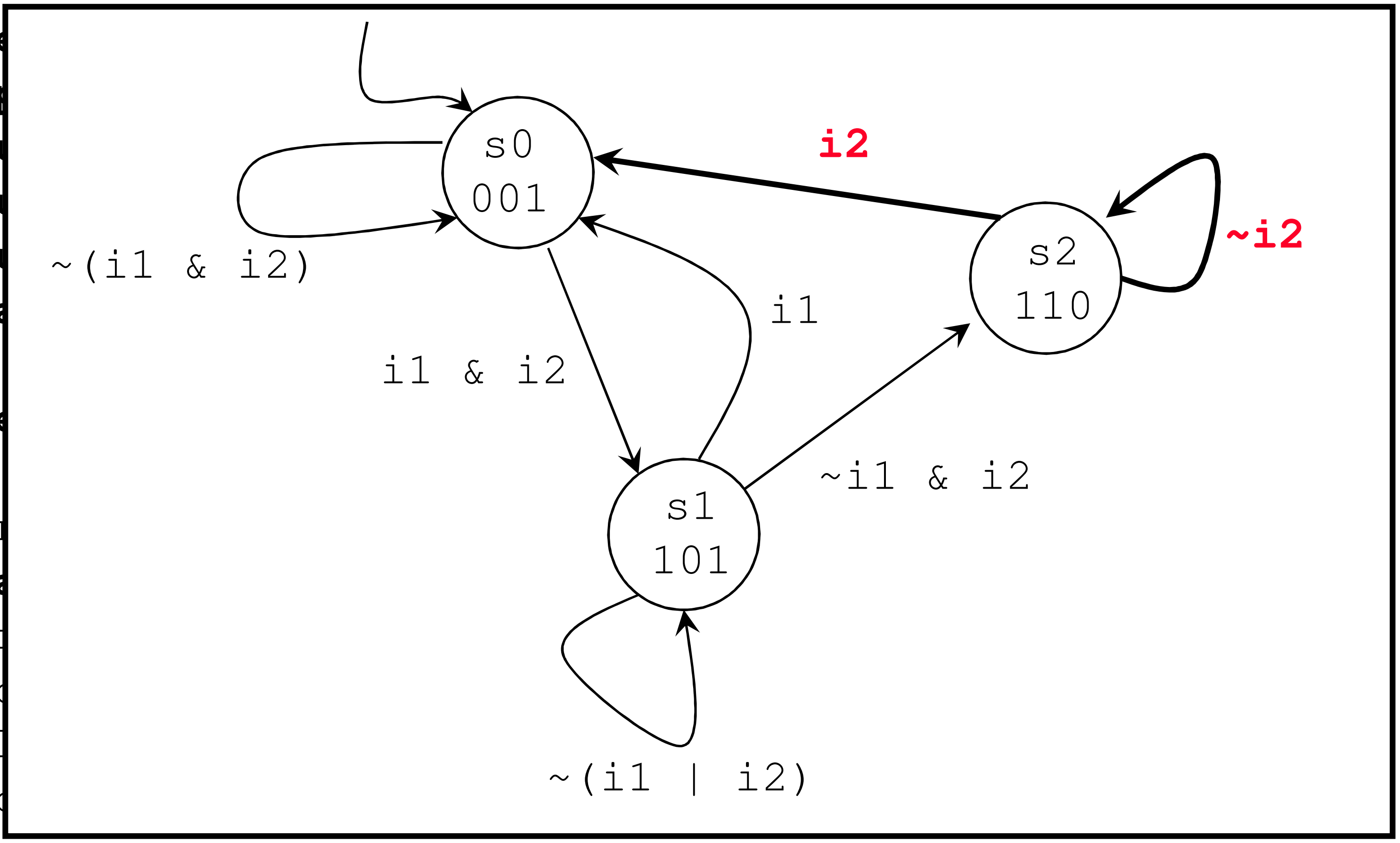


Example: Design this FSM as muxed datapath

```
module fsm(o, rst, clk, i);
  output [2:0] o;
  input rst;
  input clk;
  input [1:0] i;
  parameter s0 = 3'b001, s1 = 3'b101, s2 = 3'b110;
  reg [2:0] o; // state register
  wire [2:0] next_o;

  // rst not shown in always block
  always @(posedge clk)
    if (rst)
     o = next_o;
    else
     o = s0;

  assign next_o = (o == s0) ? (&i ? s1 : s0) :
                  (o == s1) ? (i1 ? s1 : (i2 ? s2 : s1)) :
                  (o == s2) ? (i2 ? s0 : s2);
endmodule
```

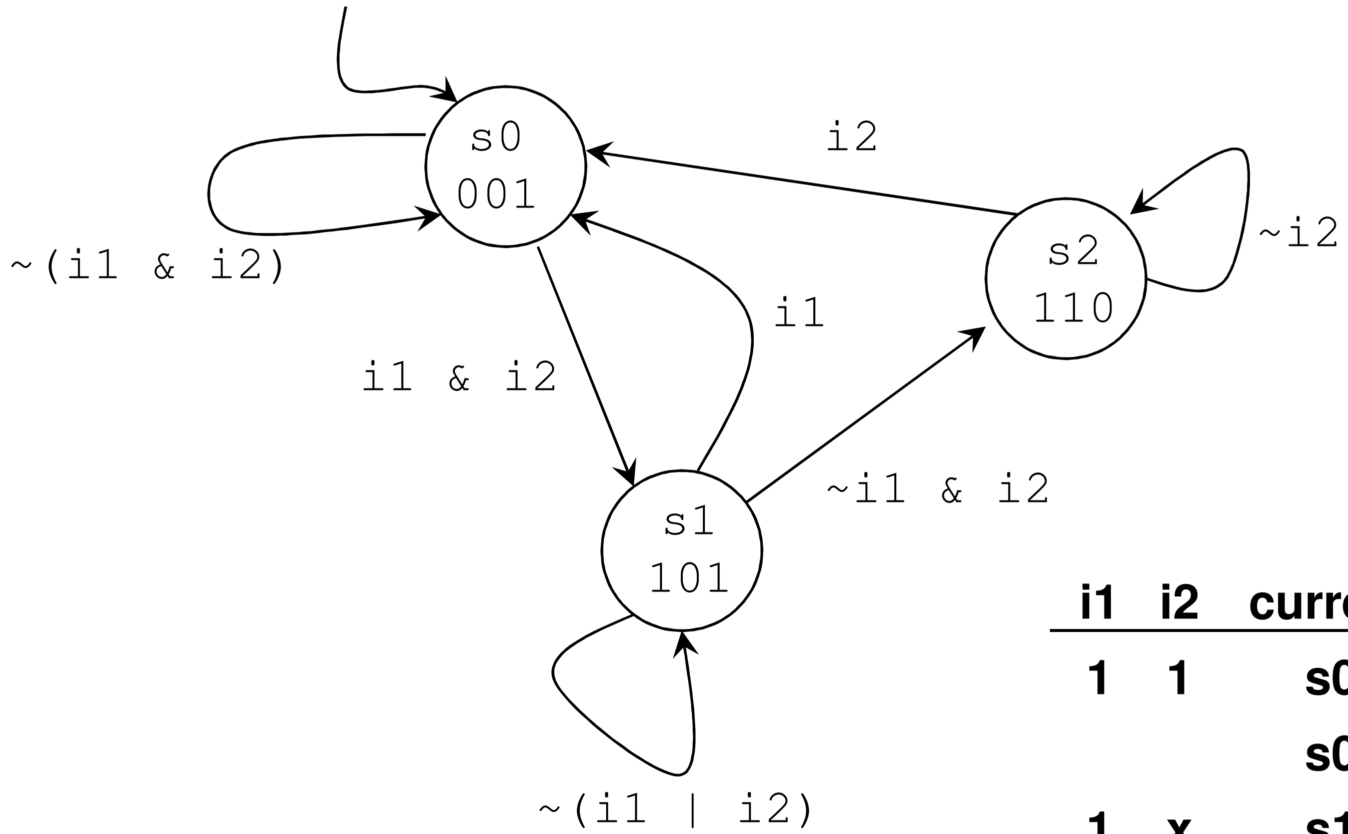
Example: Design this FSM as muxed datapath

```
module
   outp
   inpu
   inpu
   inpu
   para
   reg
   wire

   // i
   alwa
      i
       c
      e
       c
```



```
   assign next_o = (o == s0) ? (&i ? s1 : s0) :
                   (o == s1) ? (i1 ? s1 : (i2 ? s2 : s1)) :
                   (o == s2) ? (i2 ? s0 : s2);

endmodule
```

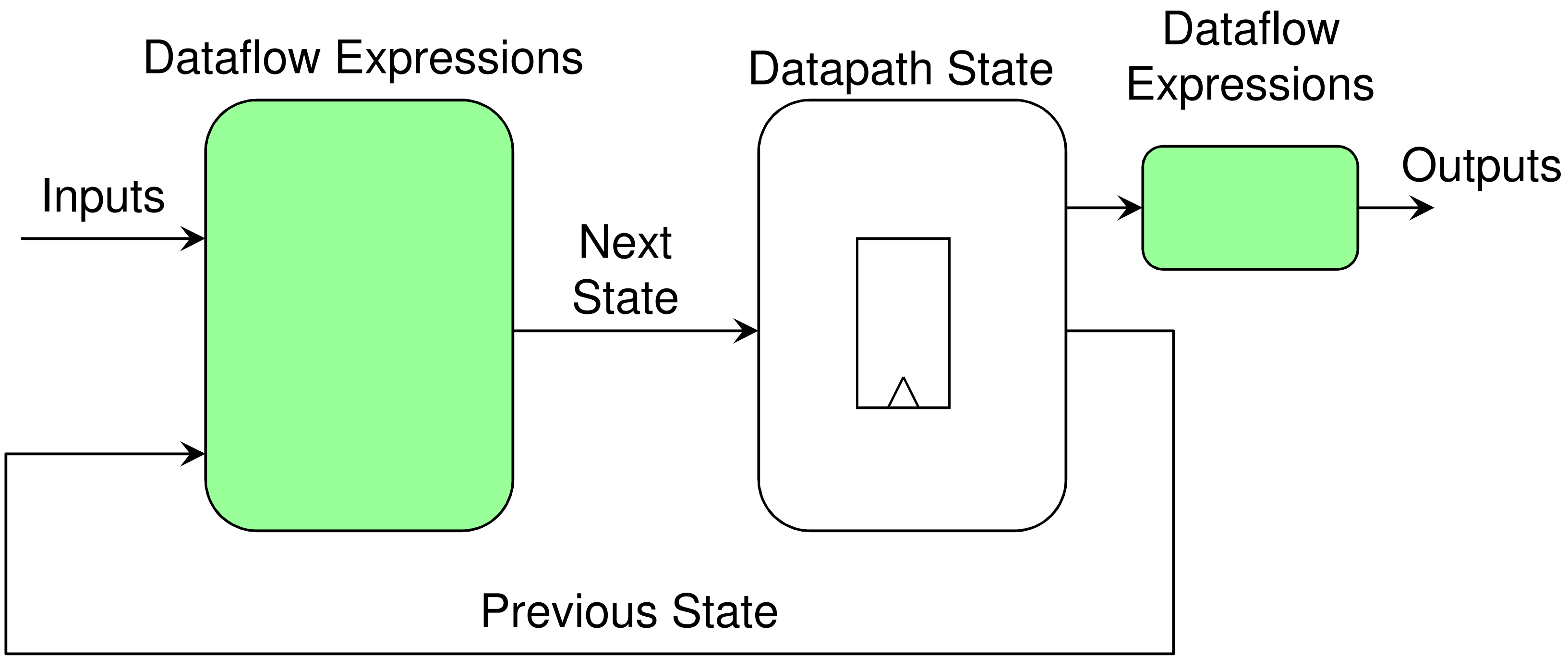
Note that FSM can be given by table



i1	i2	current	next	out
1	1	s0	s1	001
		s0	s0	001
1	x	s1	s0	101
0	1	s1	s2	101
0	0	s1	s1	101
x	0	s2	s2	110
x	1	s2	s0	110

Summary

- Multiplexed Datapath
  - Model Registers with reg and wire
  - Model Datapath logic using assign expressions
  - Use multiplexers to control register update


Dataflow Expressions
Datapath State
Dataflow Expressions
Inputs
Next State
Outputs
Previous State