ECE 4514
Digital Design II
Spring 2008

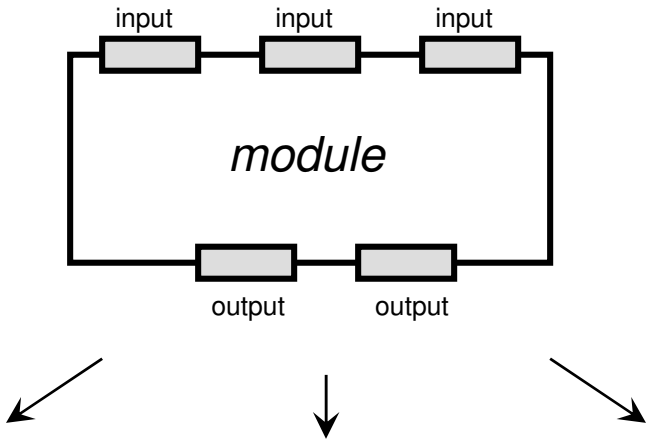
Lecture 7:
Dataflow Modeling

A *language* Lecture

Patrick Schaumont

Today's topic

- Dataflow Modeling

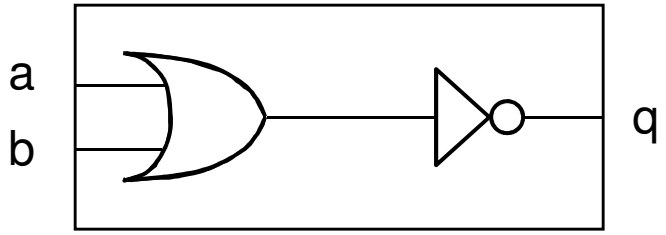


Model with submodules and gates
=
Structural

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

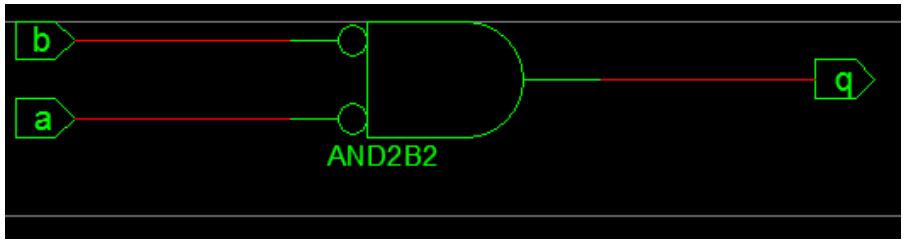
Behavioral - Procedural
assign reg

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

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

endmodule
```

Behavioral - Dataflow
assign wire

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

assign q = ~(a | b);

endmodule
```

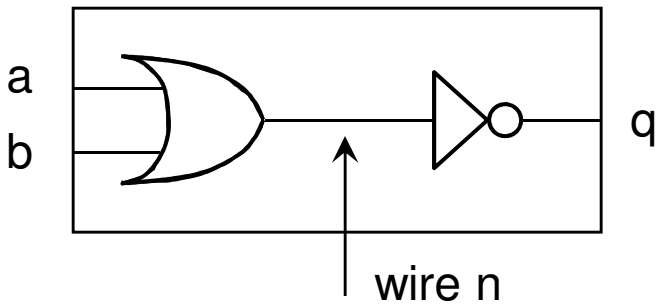
Note that both will end up as the same hardware



Two forms of assignment

- Net Declaration (*Palnitkar: Implicit Assignment)*

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

wire n = (a | b);
...
endmodule
```



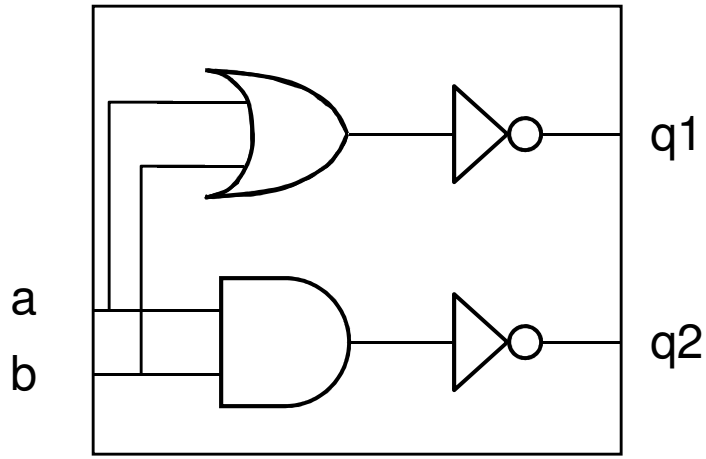
- Continuous Assignment

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

wire n;
assign n = (a | b);
...
endmodule
```

Assign two wires

Concurrent Assign statements

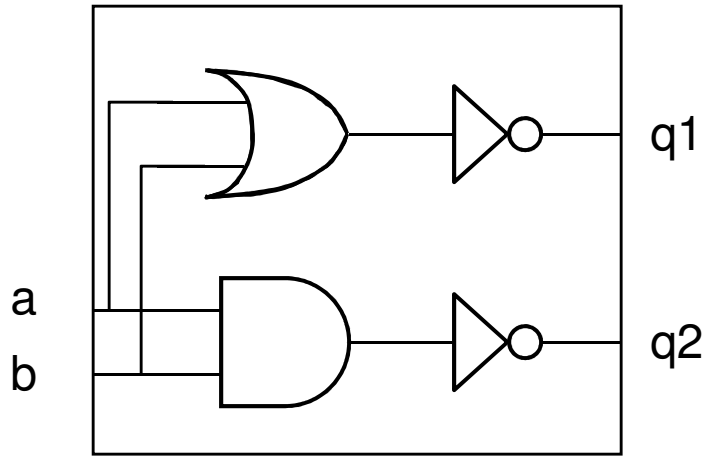
```
module nand(q1, q2, a, b)
output q1, q2;
input a, b;

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



Assign two wires

Single Assign statement with two independent assignments

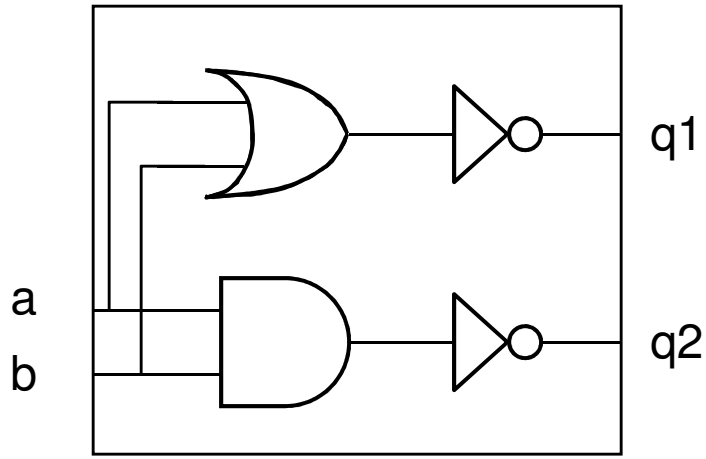
```
module nand(q1, q2, a, b)
output q1, q2;
input a, b;

assign q1 = ~(a | b),
       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;
```

Delay in continuous assignments

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

wire n;
assign #10 n = ~(a | b);
...
endmodule
```

Changes to n take an *inertial delay* of 10 units

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

wire n;
assign (#10, #15) n = ~(a | b);
...
endmodule
```

Changes to n take an *inertial delay* of 10 units for rising edges and 15 units for falling edges

Delay in continuous implicit assignments

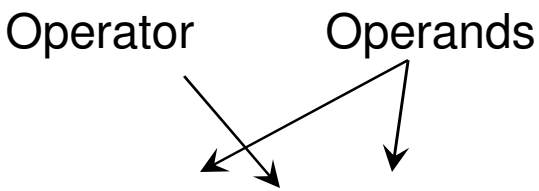
```
module nand(q, a, b)
output q;
input a, b;

wire #10 n = ~(a | b);
...
endmodule
```

Changes to n take an *inertial delay* of 10 units

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

wire #10 n;
assign #10 n = ~(a | b);
...
endmodule
```

Changes to n take an *inertial delay* of 20 units

Operands and Operators




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

Left-Hand Side = Right-Hand Side

- What we will cover:
  - What operand types can I use?
  - What operators are available ? What is their precedence ?
  - What are the rules for expression precision ?

Operand Types

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

Left-Hand Side = Right-Hand Side

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

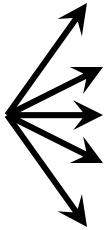
```
wire [7:0] n[0:99];
```

`n[15]`	Element 16
`n[15][5]`	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 or part-select
- indexed net
- indexed net bit-select or part-select
- concatenated net

```
wire [7:0] n[0:99];
{n[15][0:3], n[2]} = 5_bit_expr;
```

Operators

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

Arithmetic Operators

- Commonly used in synthesis: + (add), - (subtract), * (multiply)
- Not commonly used in synthesis: / (divide), ** (power), % (modulo)
- Binary Expressions: a + b, a - b, a * b
- Unary Expressions: +b, -a
- Specifying constant operands
  - a **+ 12**: add integer 12 to a
  - a **+ 16'd12**: add a 16-bit integer 12 to a

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);

Bitwise Operators

- & (and), | (or), ~ (negate), ^ (xor), ^~ (xnor) - also ~^
- Perform a bit-by-bit comparison.
  If input is N-bit, then output is N-bit.
- Similar truth table as for gates

AND

&	0	1	x	z
0	0	0	0	0
1	0	1	x	x
x	0	x	x	x
z	0	x	x	x

OR

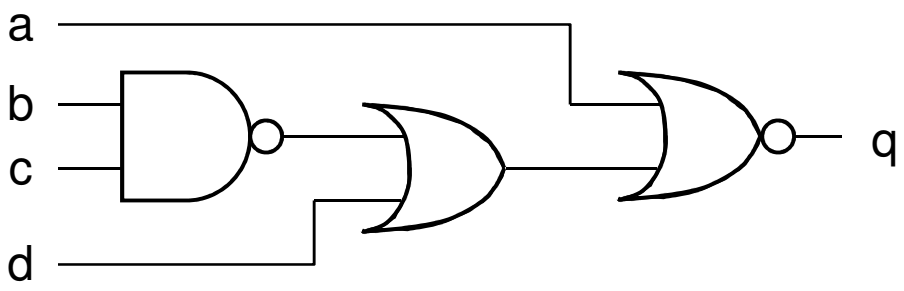
|	0	1	x	z
0	0	1	x	x
1	1	1	1	1
x	x	1	x	x
z	x	1	x	x

NOT

~	
0	1
1	0
x	x
z	x

Bitwise Operators

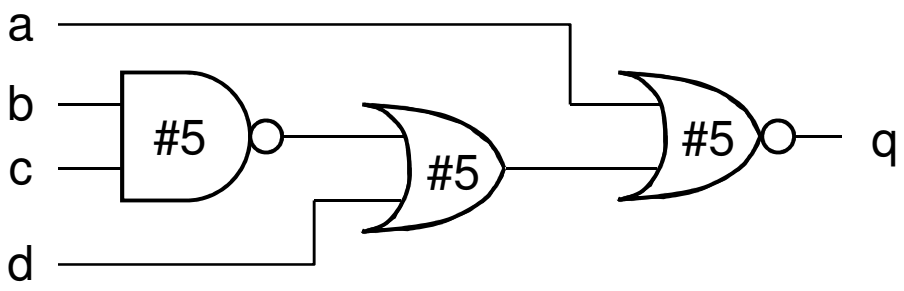
- Gate networks can be written as expressions with bitwise operators




```
assign q = ~(a | (d | ~(b & c)));
```

Bitwise Operators

- How to model gates-with-delays in expressions ?

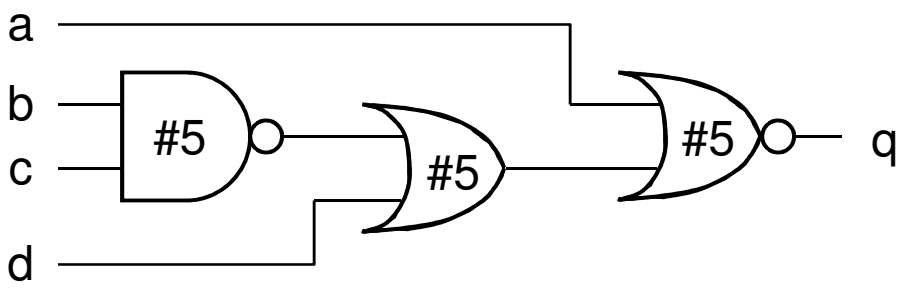



```
assign q = #15 ~(a | (d | ~(b & c)));
```

???

Bitwise Operators

- How to model gates-with-delays in expressions ?



Better:

```
assign q1 = #5 ~(b & c);
assign q2 = #5 q1 | d;
assign q  = #5 q2 | a;
```

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`

Logical Operators

- || logical or, && logical and, ! logical not
- Outcome is a single bit: 1, 0, X
- When inputs are bitvectors:
  - Effect will be to reduce-OR operands, then perform a bitwise or/and/not corresponding to logical operation
  - Example
    - `(4'b1100) && (4'b0011) =`
    - `!(4'b100x) =`

Logical Operators

- || logical or, && logical and, ! logical not
- Outcome is a single bit: 1, 0, X
- When inputs are bitvectors:
  - Effect will be to reduce-OR operands, then perform a bitwise or/and/not corresponding to logical operation
  - Example
    - `(4'b1100) && (4'b0011) = 1`
    - `!(4'b100x) = 1`

Relational Operators

- a < b, a > b, a <= b, a >= b

- Result is 1'b1 if true, 1'b0 if false, 1'bx if either a or b x
- signed comparison requires both a and b to be signed, otherwise unsigned comparison
- Note that the result of a signed comparison may be *different* than that of an unsigned comparison
  - (4'bs1100 < 4'bs0111) but (4'b1100 > 4'b0111)

means: a two's complement constant

Relational operators - Equality tests

- a ==b, a != b tests a *logical equality*
  - will be 1 or 0 when a and b are fully known
  - when any bit of a or b is X, then the result is X

4'b101X != 4'b101X

- a === b, a !== b tests a *case equality*
  - will always be 1 or 0
  - will include X's and Z's in the comparison

4'b101X === 4'b101X

Shift Operator

- Logical shift-left <<, Logical shift-right >>
- Arithmetic shift-left <<<, Arithmetic shift-right >>>
- Difference between logical and arithmetic shift:
  - Sign-bit is preserved on arithmetic shift-right of signed operand

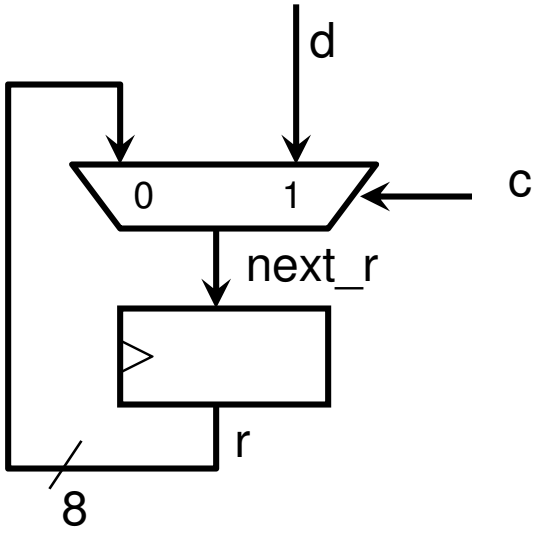
Selection Operator

- Ternary operator ? :
- Shorthand notation for a multiplexer
  - `assign q = c ? a : b;`
- c is or-reduced.
  - If result is 1, then q = a.
  - If result is 0, then q = b.
  - If result is x, then q is combined, bit by bit, from a and b.

```
a    1110011
b    0000001
     -------
c    xxx00x1
```

- **Important operation! (see lecture 8)**

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 Guessing Game

```
&&

|

~|

===

>@

5{ }

>=
```

Operator Guessing Game

`&&`	logical AND
`|`	bitwise/reduction OR
`~|`	reduction NOR
`===`	case equality
`>@`	???
`5{ }`	replication
`>=`	relational

Operator Precedence

+ - ! ~ & ~& | ~| ^ ~^ ^~ (unary)	Highest precedence
**	
* / %	
+ - (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& (binary)	
^ ^~ ~^ (binary)	
| (binary)	
&&	
||	
?: (conditional operator)	
{} {{}}	Lowest 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(Y, S, D0, D1, D2);
  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(Y, S, D0, D1, D2);
  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(Y, D0, D1);
  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 output reflects 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(Y, D);
  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.

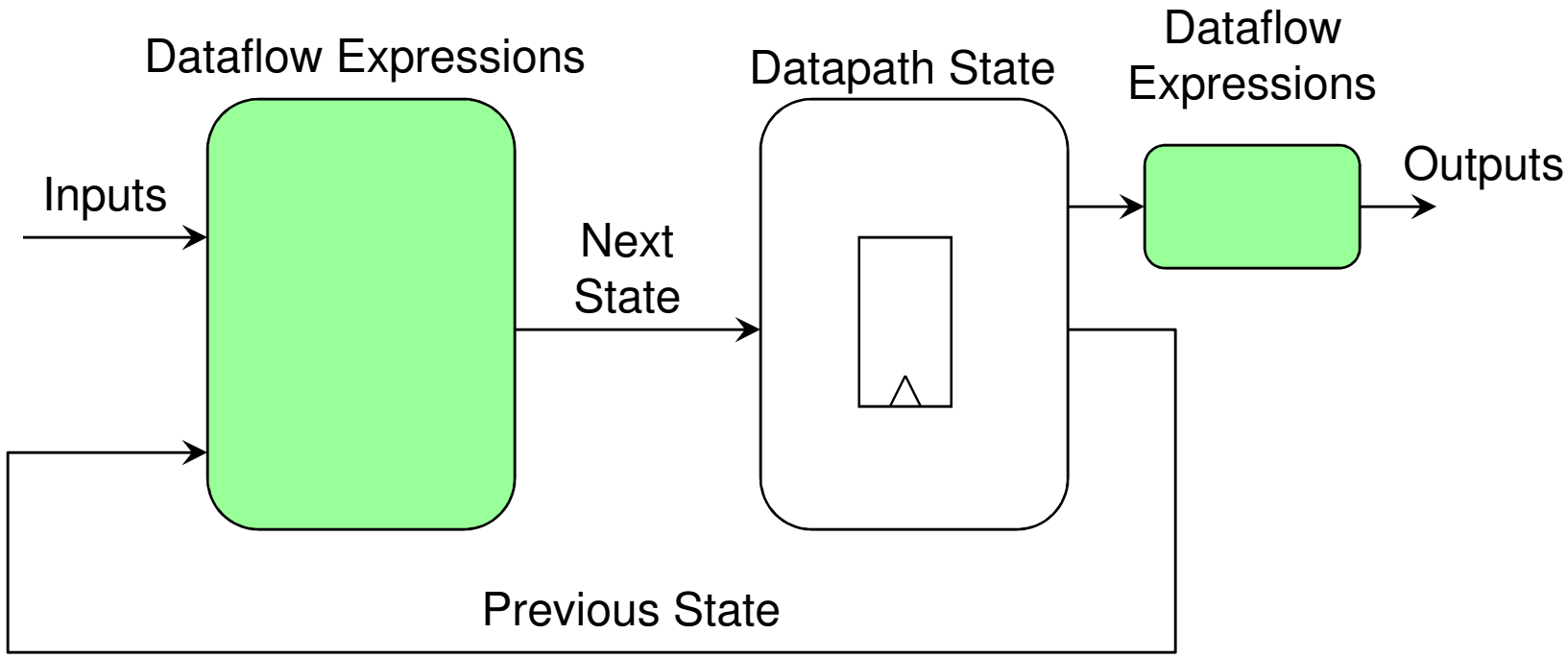
Example 4

- 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(Y, clk, rst, D0, D1);
  output [39:0] Y;
  input  clk;
  input  rst;
  input  [15:0] D0, D1;
  reg    [39:0] Y;
  wire   [39:0] next_Y;

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

  assign next_Y = Y + (D0 * D1);

endmodule
```

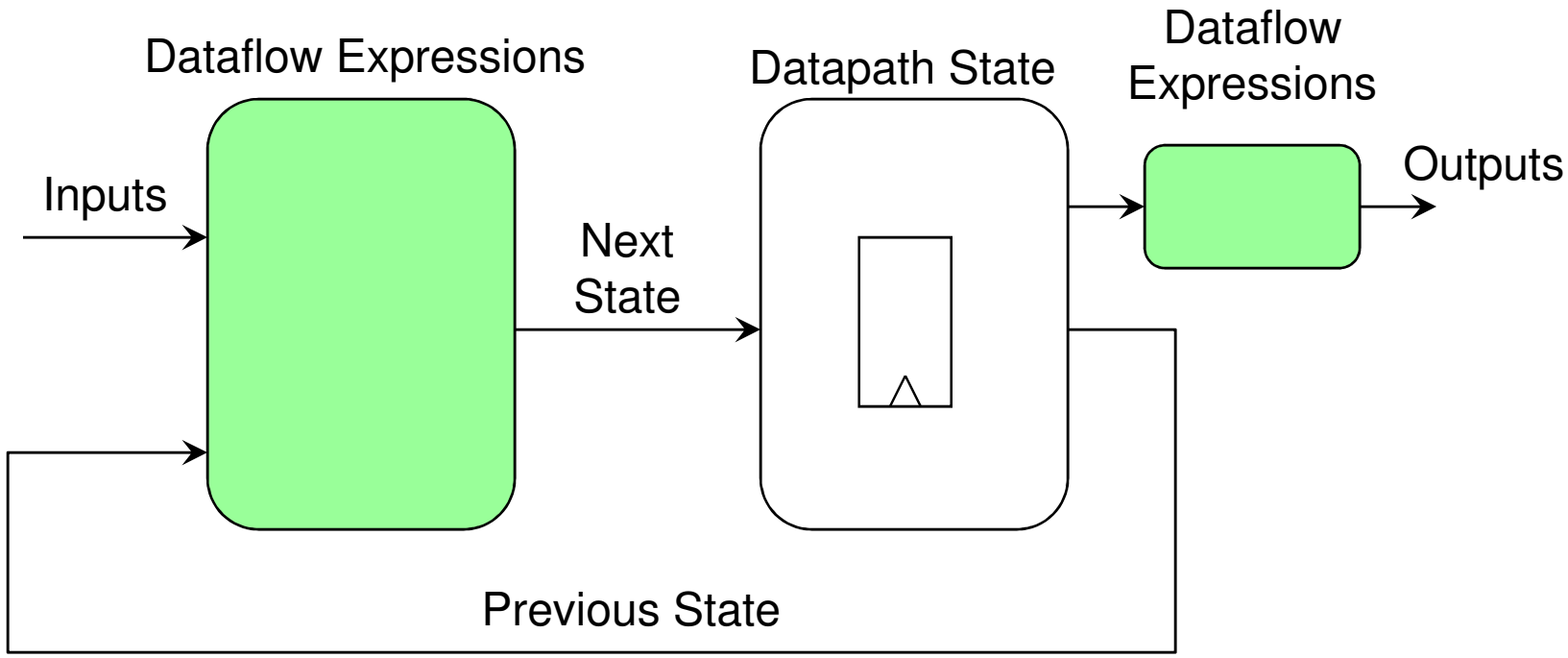
Dataflow Expressions

- Dataflow expressions are VERY powerful
- They can be used as a generic datapath modeling technique



Dataflow Expressions

- Looks like a Moore FSM
  - But, Dataflow Expressions can easily capture arithmetic. So, more flexibility than FSM.
  - See Lecture 8 - Multiplexed Datapaths



Summary

- Dataflow Modeling = Modeling using continous assignments to nets.
- Rich set of operators available
  - Arithmetic, Logical, Relational, Bitwise, Reduction, Shift, Concatentation, Replication, Selection.
- Precision - expressions will not loose precision if their target is big enough.
- Delay - assignments follow inertial model.