Computer Architecture
Tutorial 12

Verilog HDL: A Brief Syntax Overview

Artem Burmyakov, Alexander Tormasov

November 11, 2021

INNOPOLIS UNIVERSITY

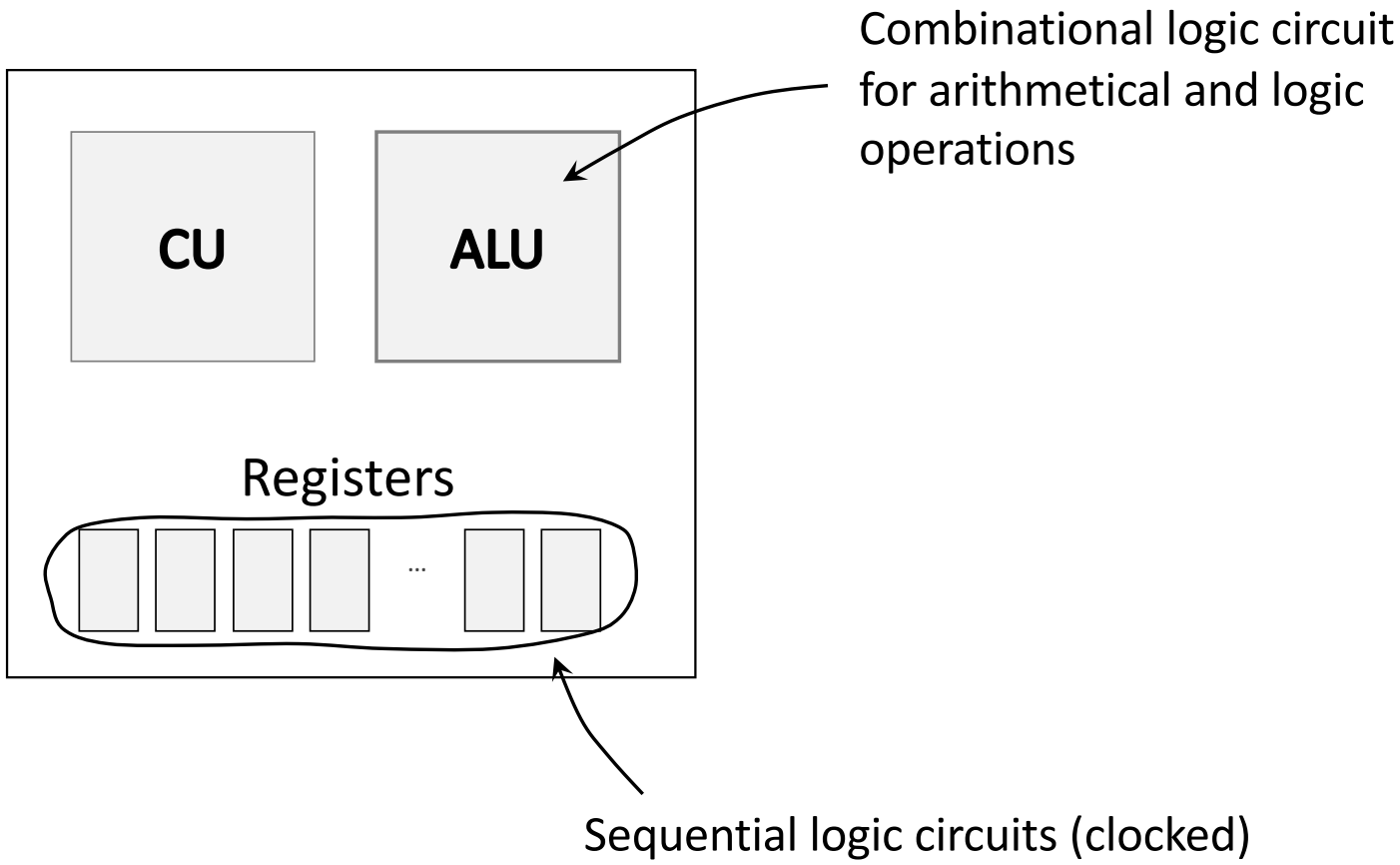
Recap: Characteristics of Combinational Logic Circuits

1) An output signal depends just on input signals
2) Comprised of logic gates, such as AND, OR, etc., and wires connecting them
3) Output signal is updated after the change of input signals, subject to propagation delay
4) Use no memory units, such as registers
5) Typically, no clock signal is present (exceptions might apply)

Key advantage: Fast (thus, used in such components, as processor ALU)

Key disadvantage: No support of inputs synchronization (will be discussed later)

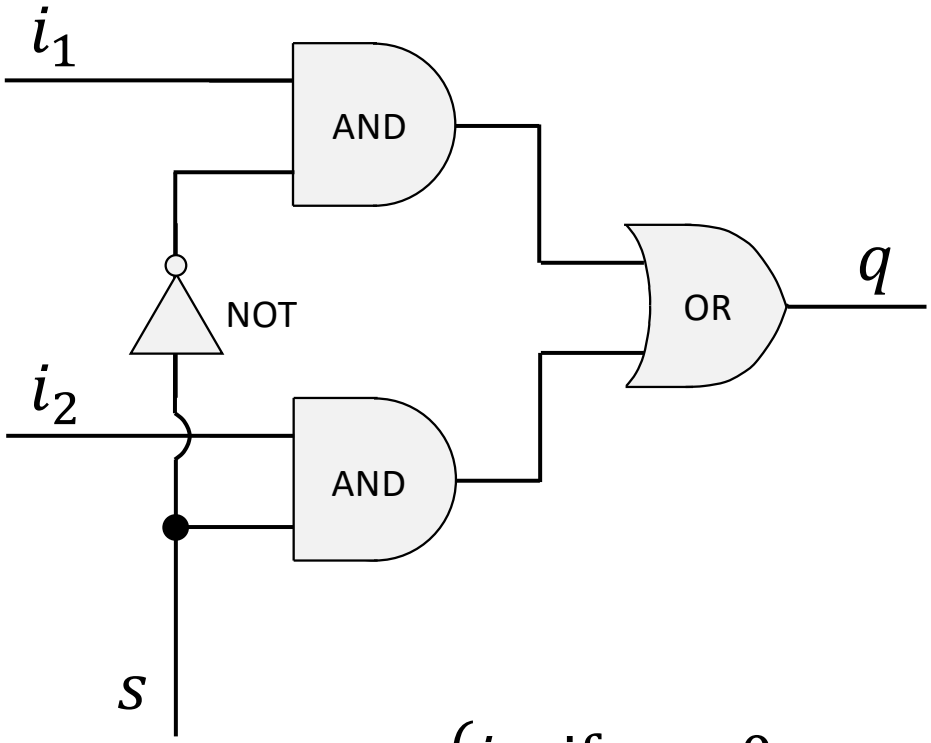
Types of Logic Circuits for CPU Components



Recap: Synchronous vs. Asynchronous Circuits

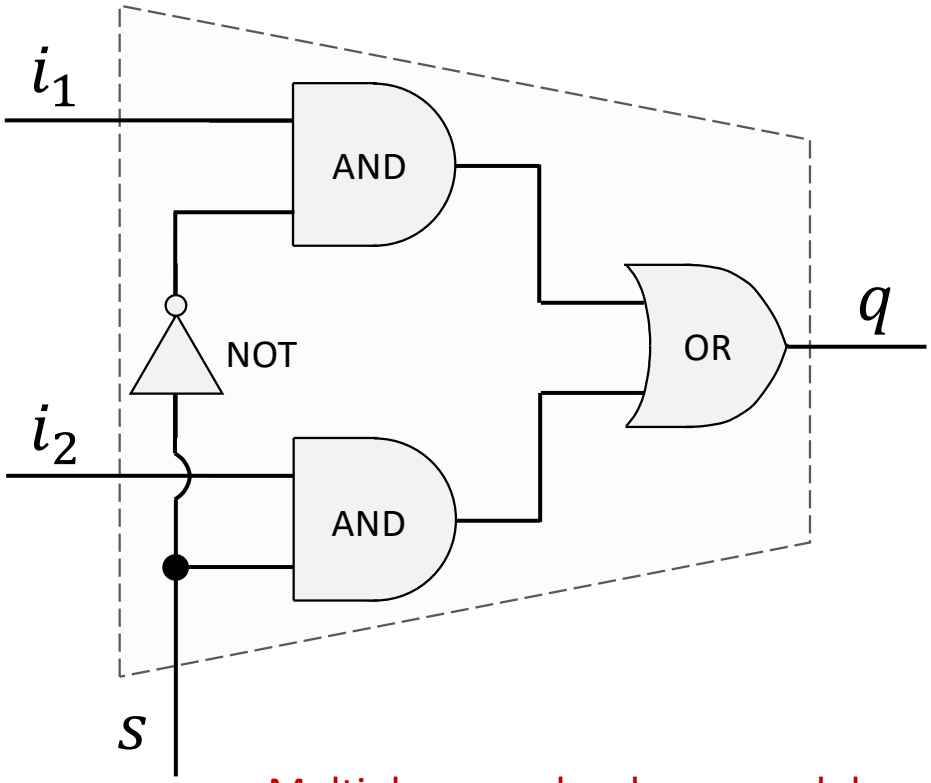
Characteristic	**Asynchronous** (like S/R latch)	**Synchronous** (like Flip-Flop)
Output Update Trigger	Any change of one of the inputs	A clock signal, together with other input signals
Clock Presence	No clock present	Clock governs the entire circuit activity
Reliability (for result correctness)	Less reliable (prone to incorrect result, subject to propagation delays)	Reliable (guarantees a correct result, independently of propagation delays)
Memory Element Presence	Not used	Used
Operation Speed	Faster (no clock signal for synchronisation delay is used)	Slower (due to overpessimistic synchronisation delays)
Power Consumption	Lower	Higher (e.g. due to the presence of flip-flops, consuming power for data storage)
Logical Complexity, Size (the number of elements)	Simpler, smaller (*Note:* an asynchronous circuit might behave as a synchronous, but at the price of a higher hardware implementation complexity)	More complex, larger
Sample Use Cases	Arithmetic-Logic Unit (ALU); Small fast peripheral circuits supporing CPU operation	Register files; Most of the circuits containing memory elements

2-to-1 Multiplexor: Digital Circuit by Using Logic Gates



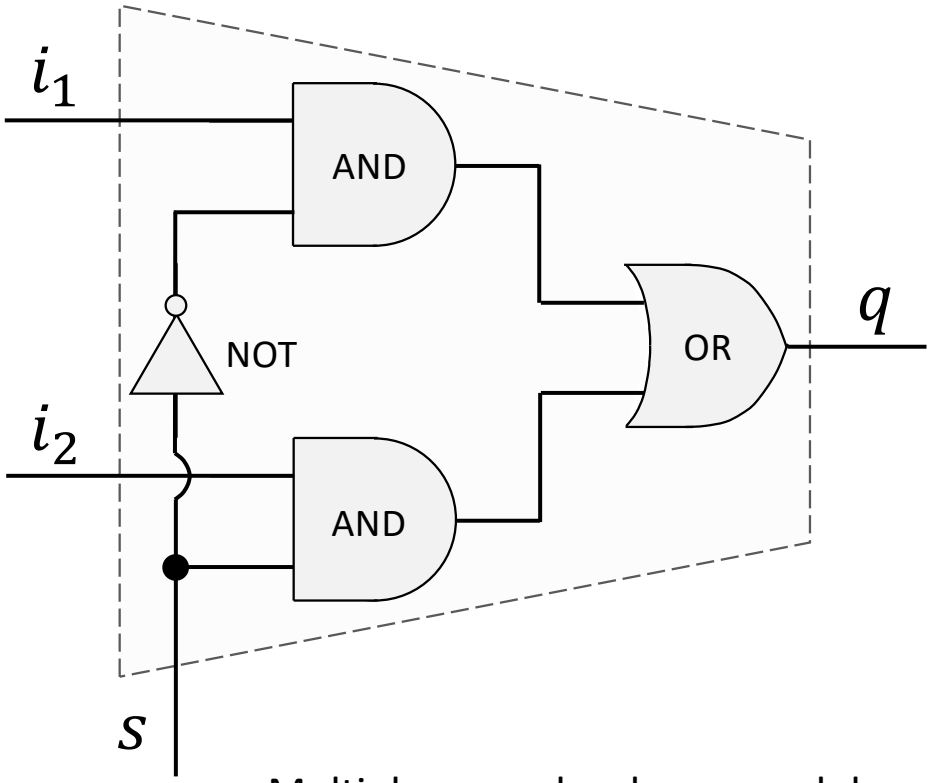
$$q = \begin{cases} i_1, \text{ if } s = 0 \\ i_2, \text{if } s = 1 \end{cases}$$

2-to-1 Multiplexor: Digital Circuit by Using Logic Gates



Multiplexer – a hardware module

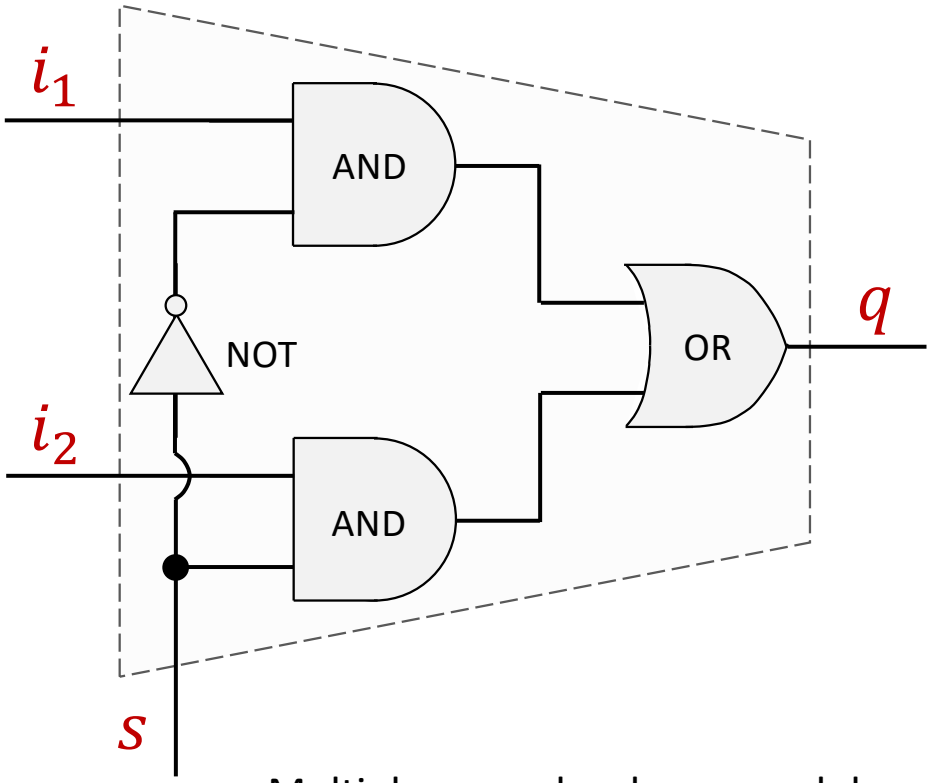
Multiplexor Implementation in Verilog



Multiplexer – a hardware module

```
1  module mux
9  endmodule
```

Multiplexor Implementation in Verilog



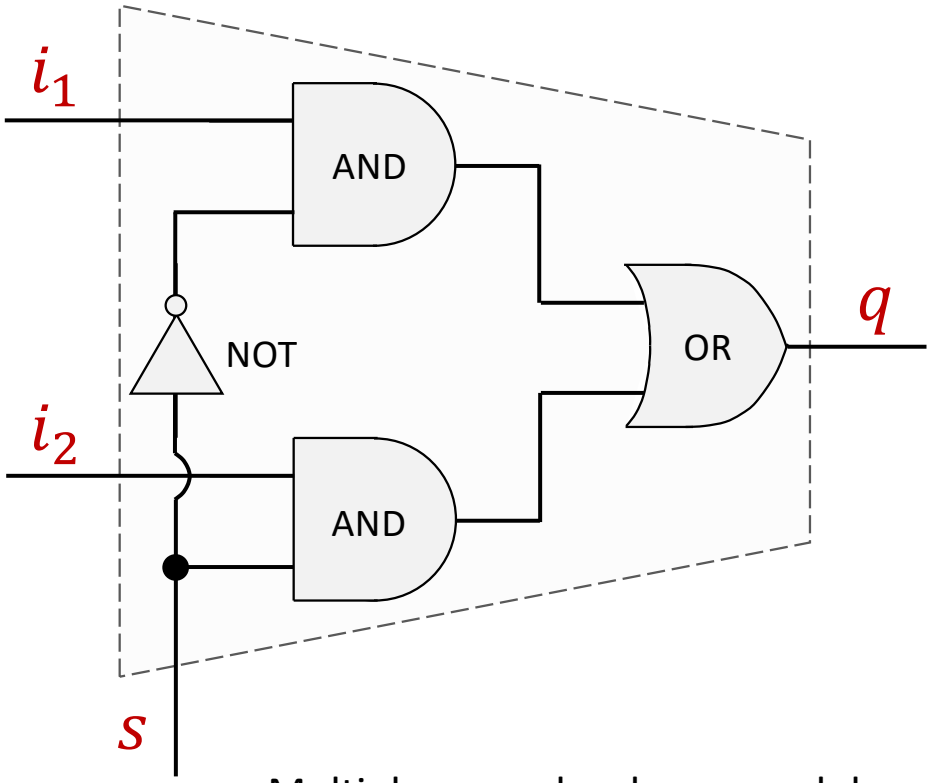
Multiplexer – a hardware module:

i_1, i_2, s – input pins;

q – output pin

```
1  module mux
9  endmodule
```

Multiplexor Implementation in Verilog



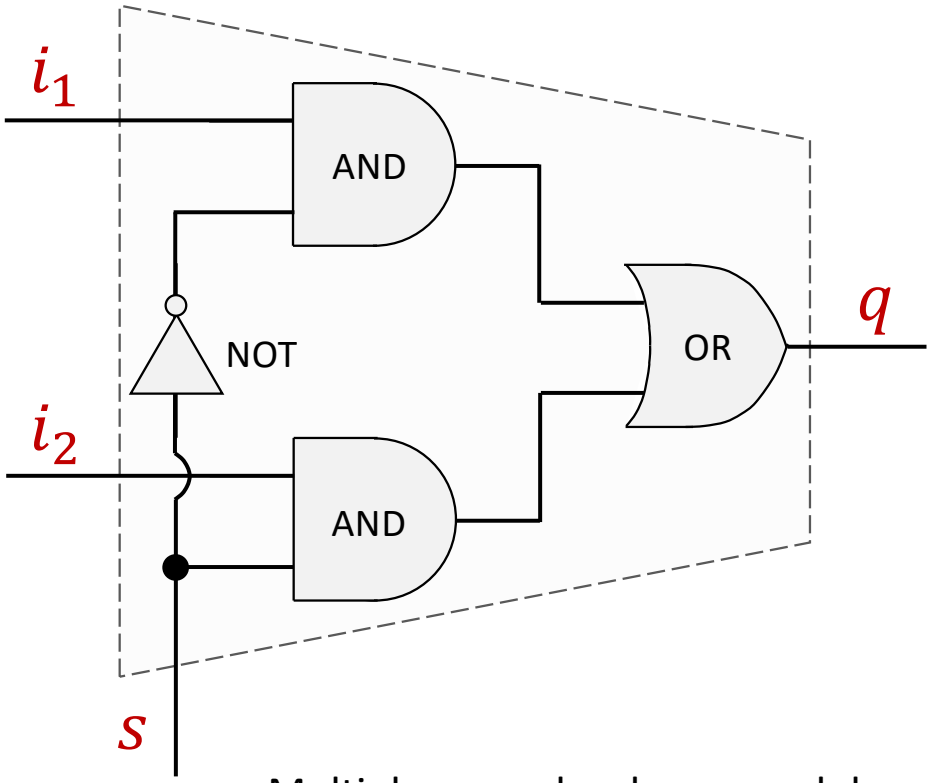
Multiplexer – a hardware module:

i_1, i_2, s – input pins;

q – output pin

```
1  module mux (i1, i2, s, q);
9  endmodule
```

Multiplexor Implementation in Verilog



Multiplexer – a hardware module:

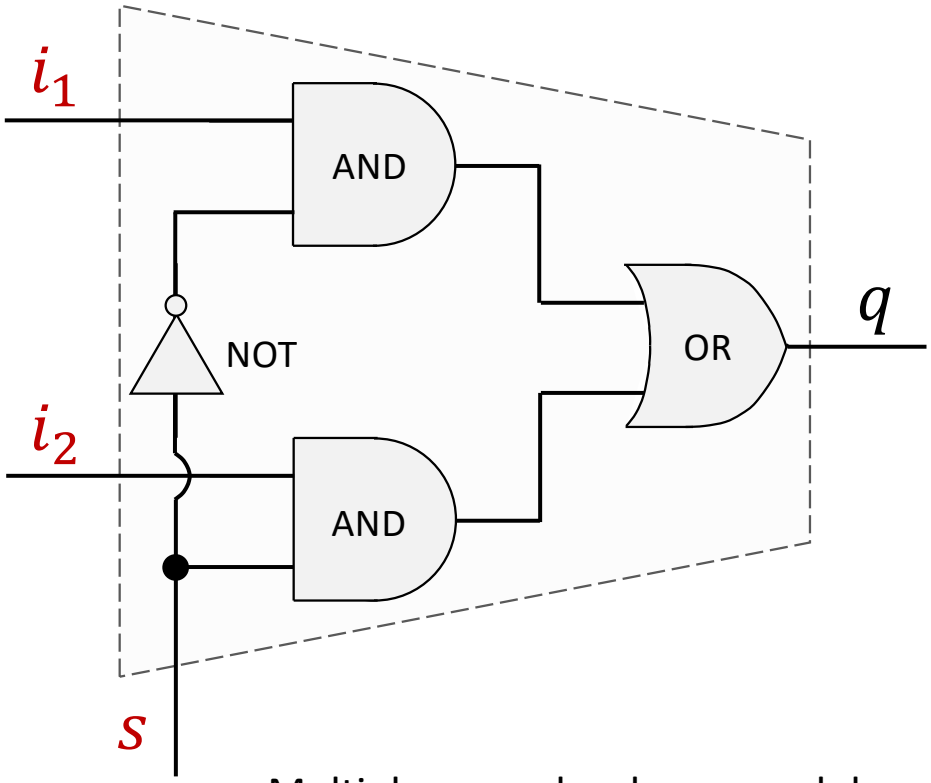
i_1, i_2, s – input pins;

q – output pin

The list of all input and output pins of a module

```
1  module mux (i1, i2, s, q);
9  endmodule
```

Multiplexor Implementation in Verilog



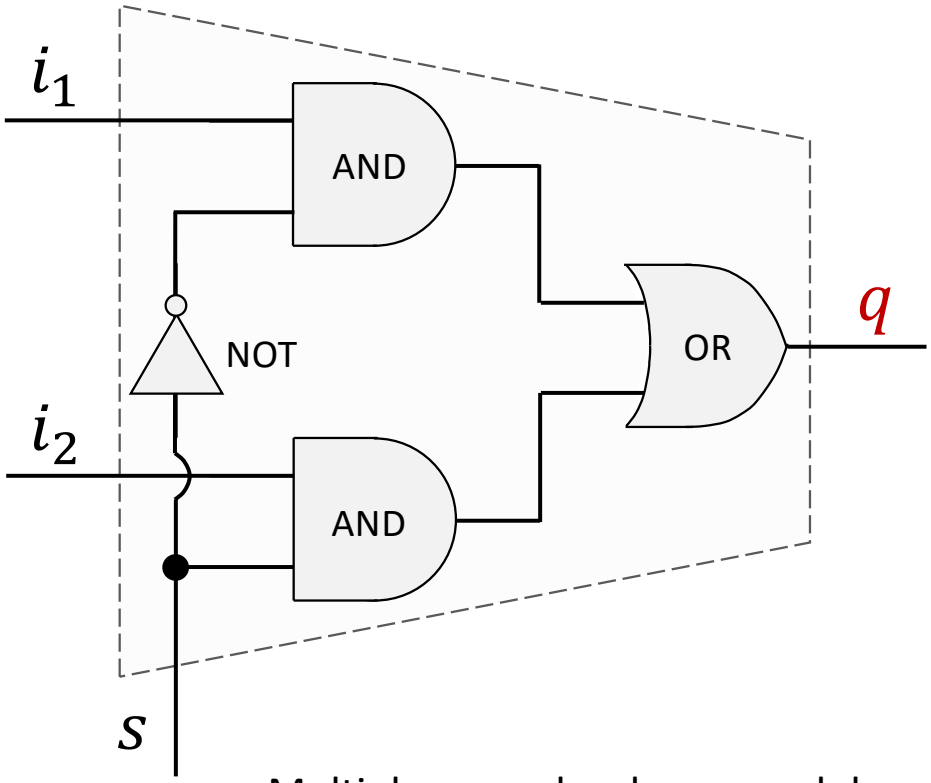
Multiplexer – a hardware module:

i_1, i_2, s – input pins;

q – output pin

```
1  module mux (i1, i2, s, q);

2     input i1, i2, s;

9  endmodule
```

Multiplexor Implementation in Verilog



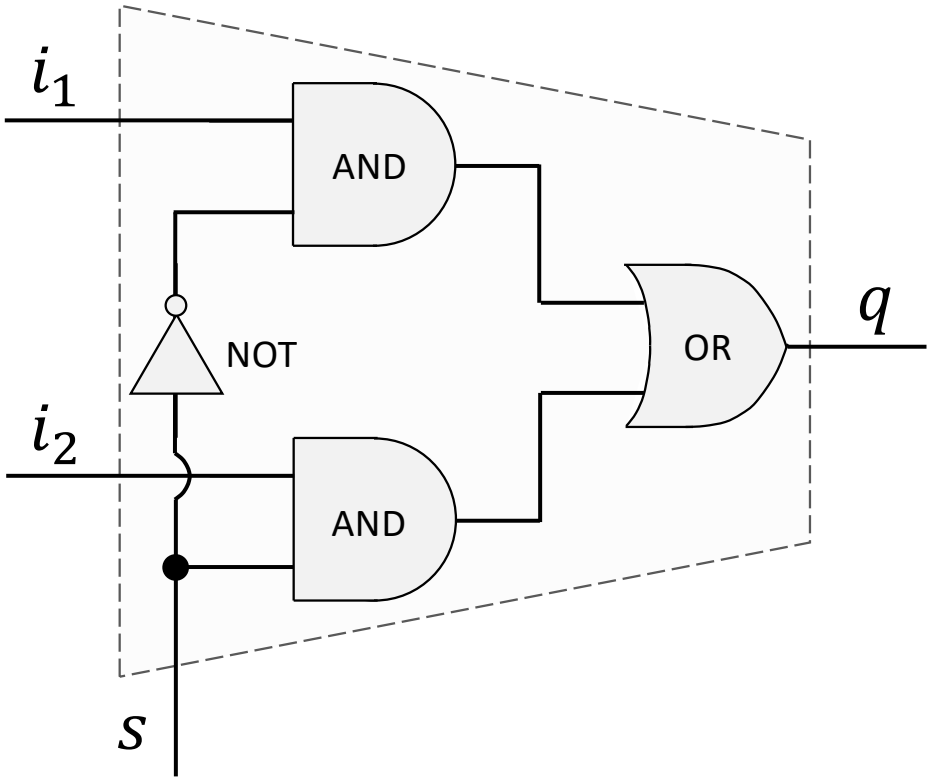
Multiplexer – a hardware module:

i_1, i_2, s – input pins;

q – output pin

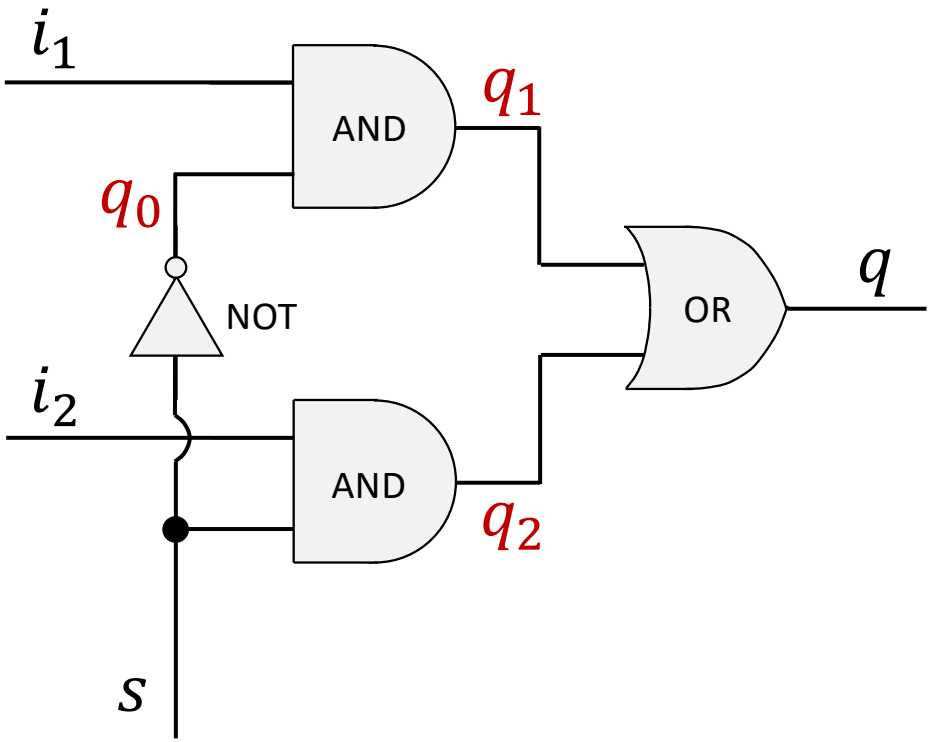
```
1  module mux (i1, i2, s, q);

2     input i1, i2, s;
3     output q;

9  endmodule
```

Multiplexor Implementation in Verilog




```
1  module mux (i1, i2, s, q);

2      input i1, i2, s;
3      output q;

9  endmodule
```

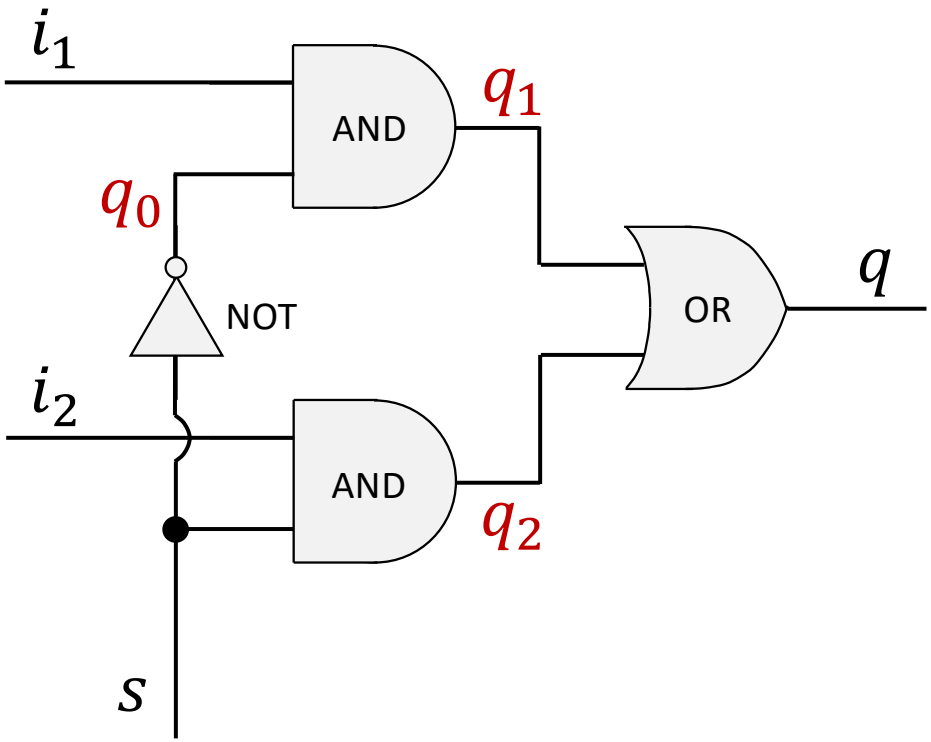
Multiplexor Implementation in Verilog




```
1  module mux (i1, i2, s, q);

2     input i1, i2, s;
3     output q;
9  endmodule
```

q_0, q_1, q_2 - internal wires of our hardware module

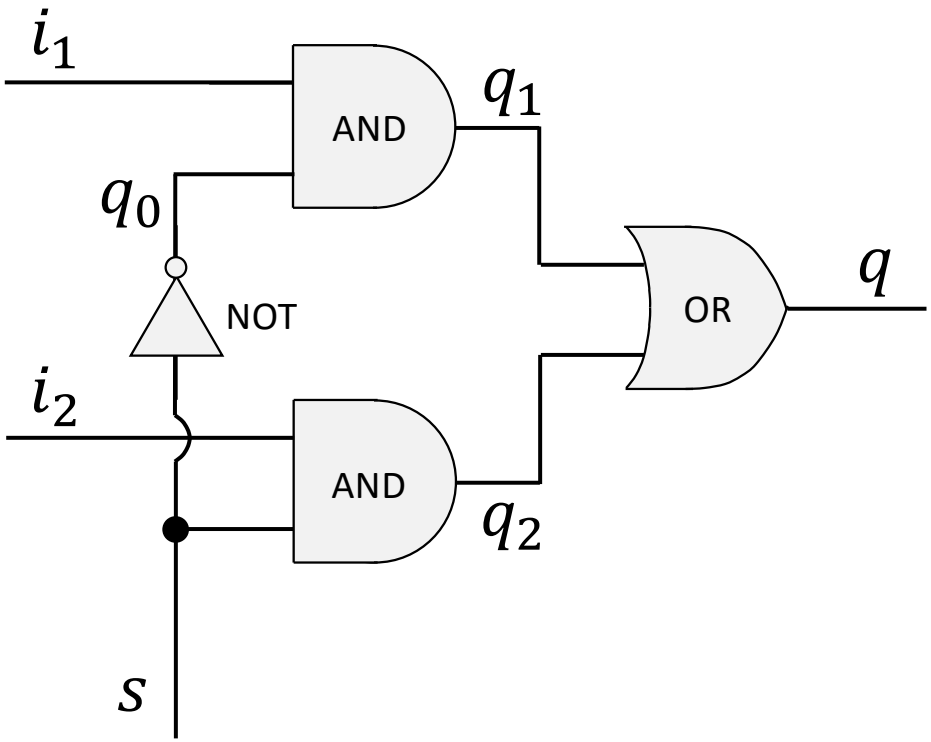
Multiplexor Implementation in Verilog




```
1  module mux (i1, i2, s, q);

2      input i1, i2, s;
3      output q;

4      wire q0, q1, q2;

9  endmodule
```

q_0, q_1, q_2 - internal wires of our hardware module

Multiplexor Implementation in Verilog

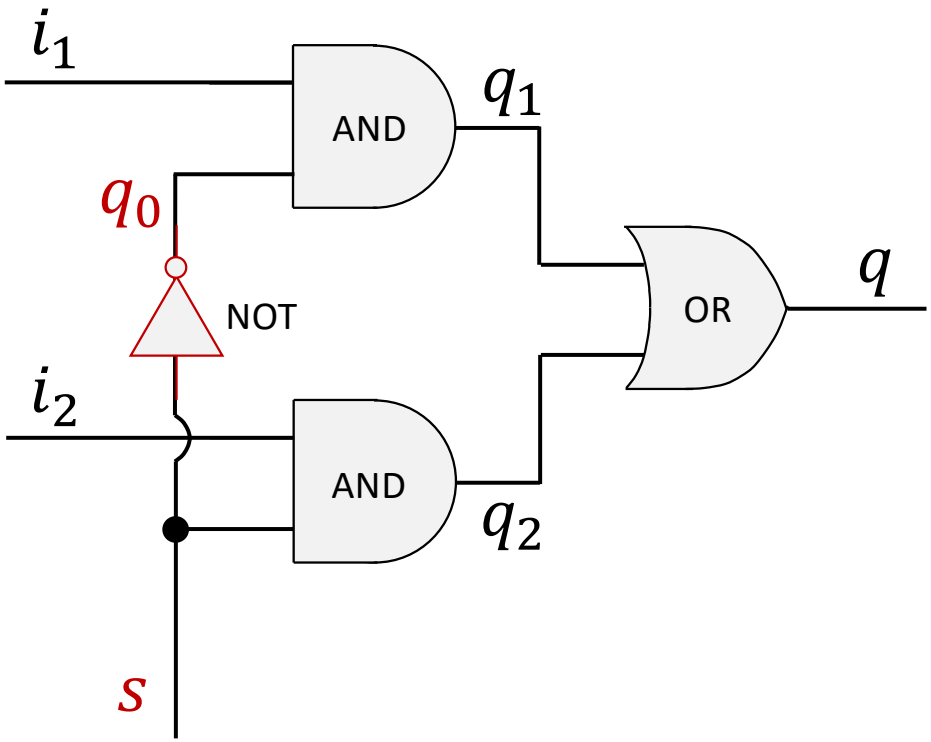


Input and output pins are always of type “wire”

```
1  module mux (i1, i2, s, q);

2     input i1, i2, s;
3     output q;

4     wire q0, q1, q2;









9  endmodule
```

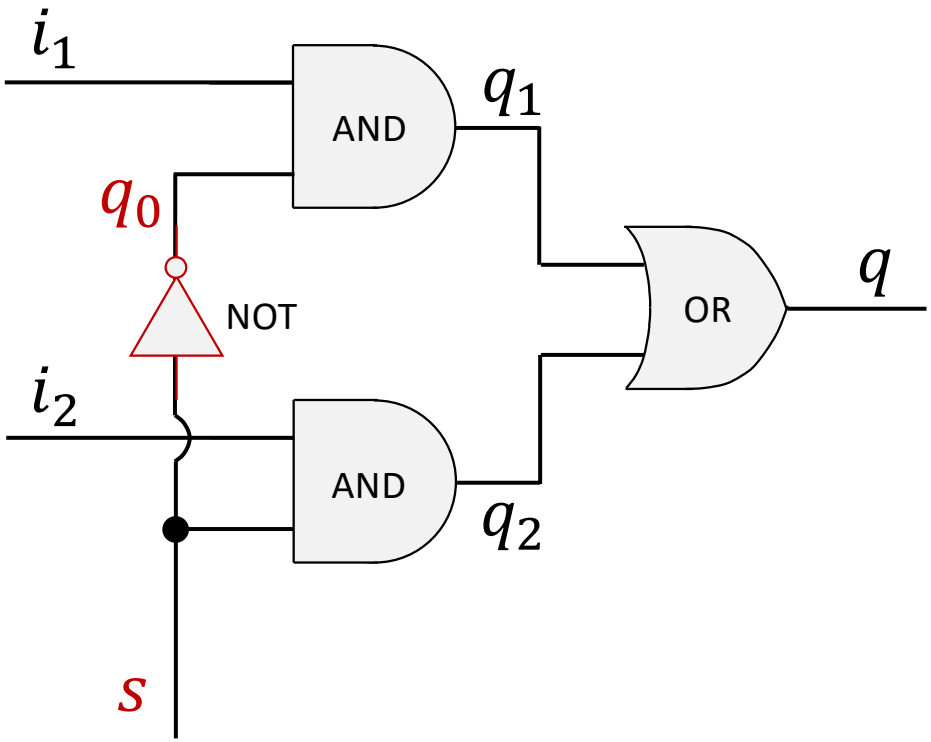
q_0, q_1, q_2 - internal wires of our hardware module

Multiplexor Implementation in Verilog




```
1  module mux (i1, i2, s, q);

2      input i1, i2, s;
3      output q;

4      wire q0, q1, q2;

9  endmodule
```

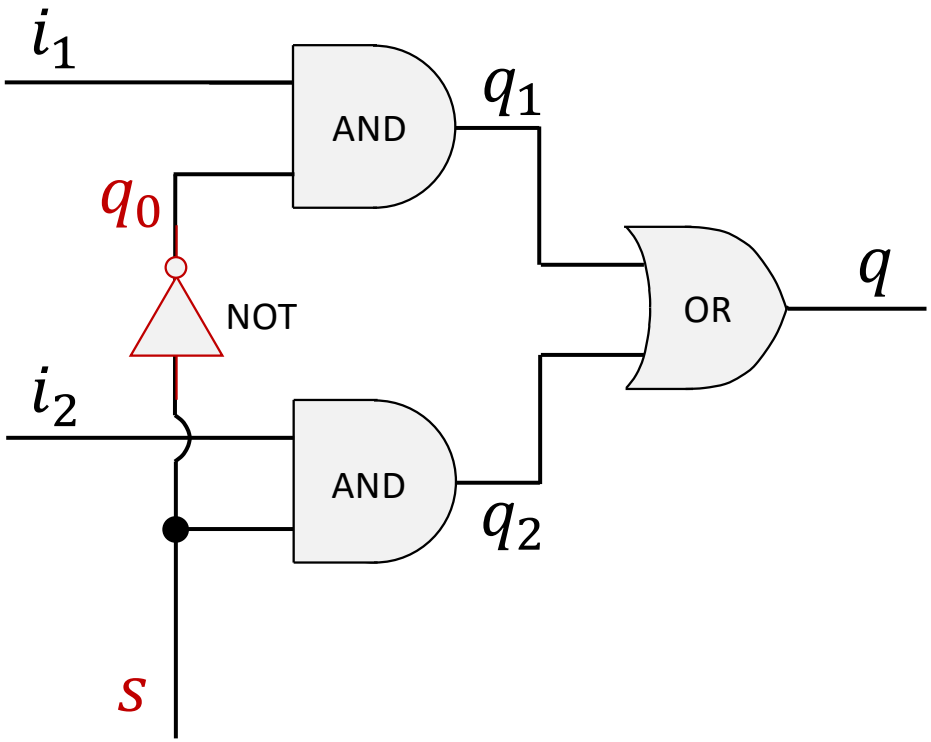
Multiplexor Implementation in Verilog




```
1  module mux (i1, i2, s, q);

2      input i1, i2, s;
3      output q;

4      wire q0, q1, q2;

9  endmodule
```

Verilog provides built-in primitive hardware modules (not, add, or, etc.)

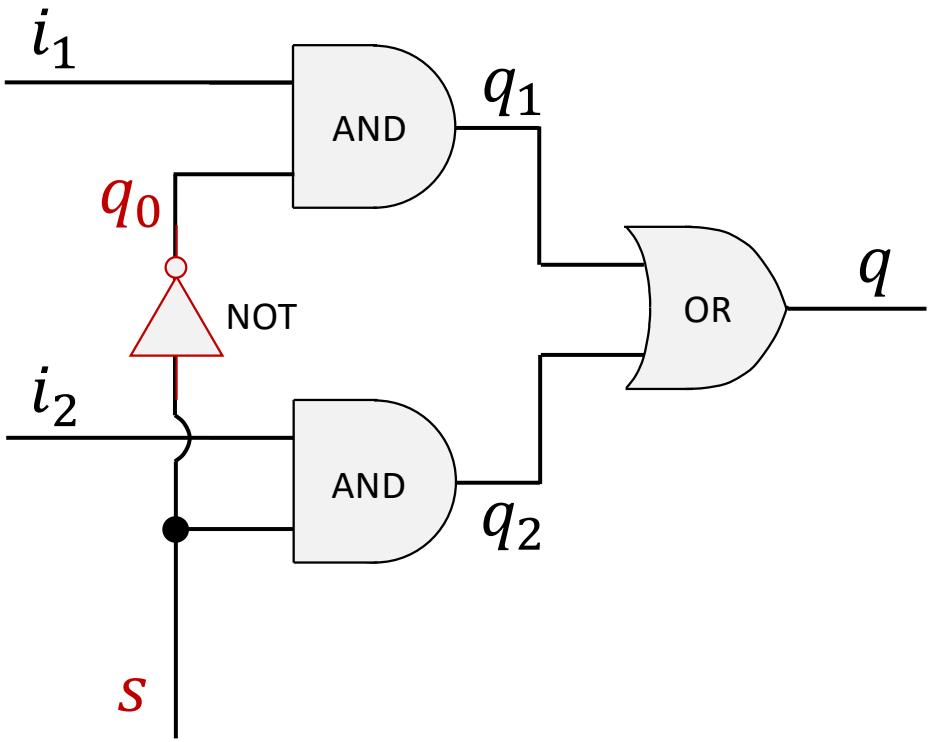
Multiplexor Implementation in Verilog




```
1  module mux (i1, i2, s, q);

2      input i1, i2, s;
3      output q;

4      wire q0, q1, q2;

5      not(q0, s);




9  endmodule
```

Verilog provides built-in primitive hardware modules (not, add, or, etc.)

Multiplexor Implementation in Verilog

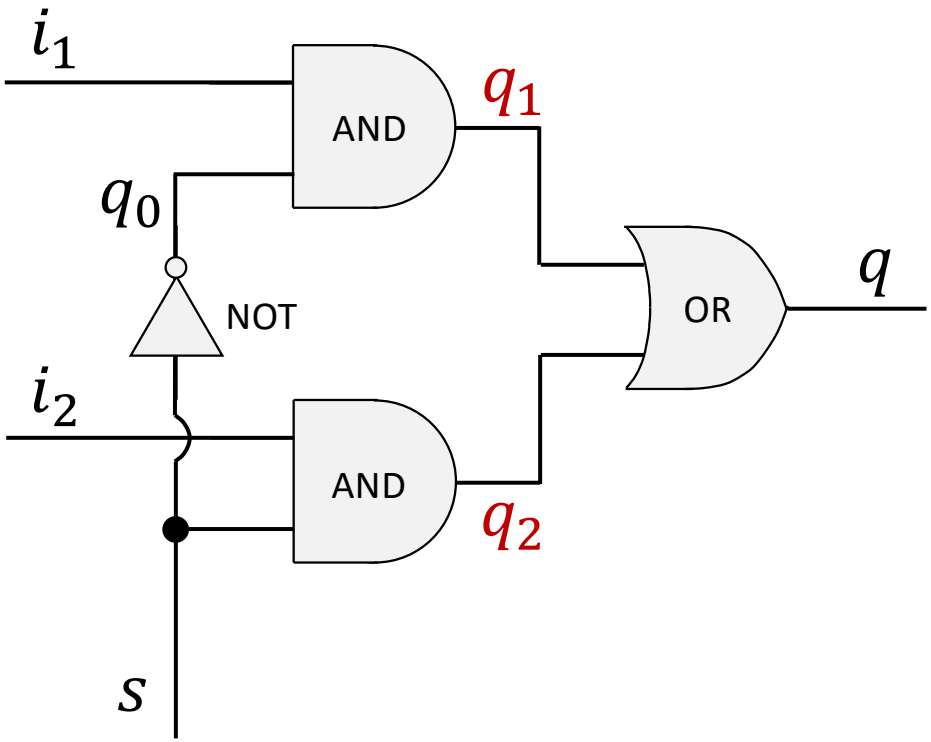



```
1  module mux (i1, i2, s, q);

2      input i1, i2, s;
3      output q;

4      wire q0, q1, q2;

5      not(q0, s);




9  endmodule
```

Verilog provides built-in primitive hardware modules (not, add, or, etc.)

Output pins (wires) are listed first (similarity to MIPS assembler)

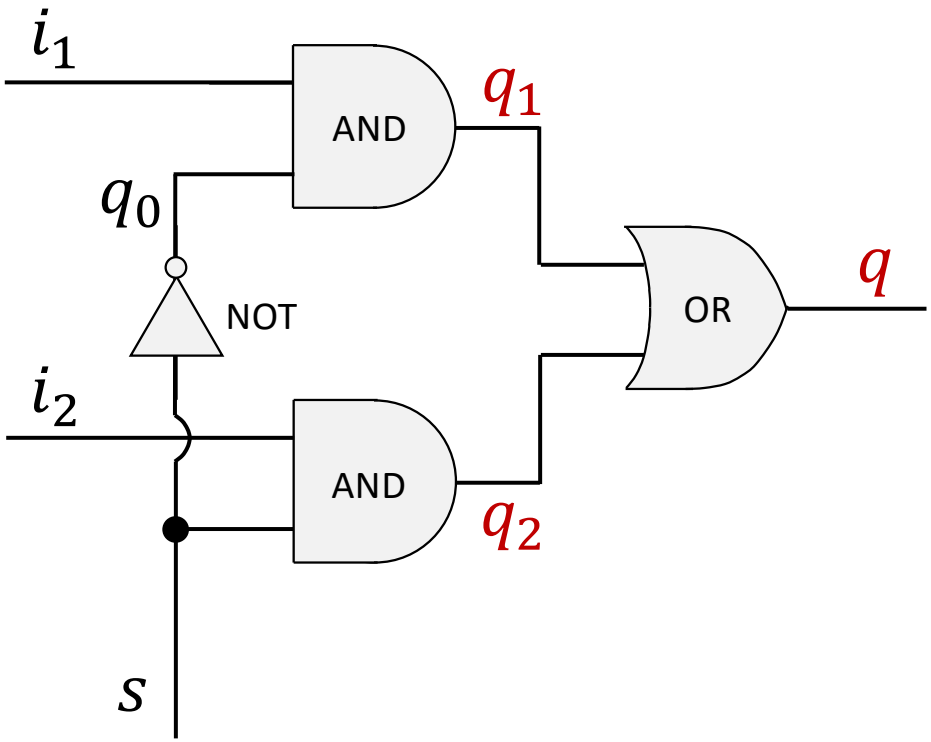
Multiplexor Implementation in Verilog




```
1  module mux (i1, i2, s, q);

2      input i1, i2, s;
3      output q;

4      wire q0, q1, q2;

5      not(q0, s);
6      and(q1, i1, q0);
7      and(q2, i2, s);


9  endmodule
```

Verilog provides built-in primitive hardware modules (not, add, or, etc.)

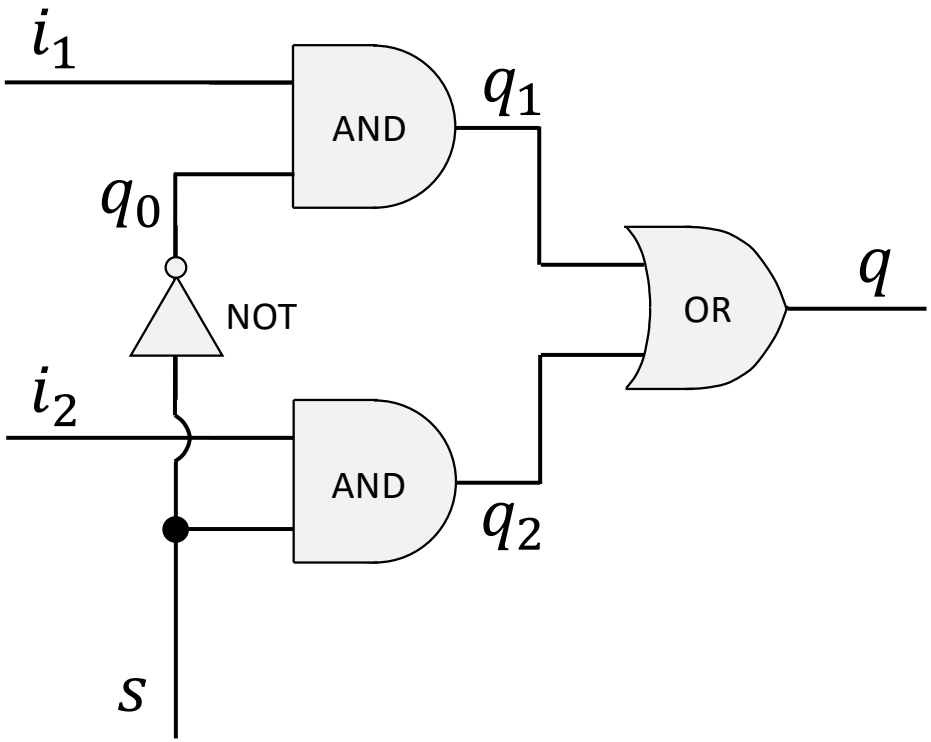
Multiplexor Implementation in Verilog




```
1  module mux (i1, i2, s, q);

2      input i1, i2, s;
3      output q;

4      wire q0, q1, q2;

5      not(q0, s);
6      and(q1, i1, q0);
7      and(q2, i2, s);
8      or(q, q1, q2);

9  endmodule
```

Verilog provides built-in primitive hardware modules (not, add, or, etc.)

Multiplexor Implementation in Verilog




```
1  module mux (i1, i2, s, q);

2      input i1, i2, s;
3      output q;

4      wire q0, q1, q2;

5      not(q0, s);
6      and(q1, i1, q0);
7      and(q2, i2, s);
8      or(q, q1, q2);

9  endmodule
```

Syntax Variation for Module Declaration in Verilog

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output q;
4      wire q0, q1, q2;
5      not(q0, s);
6      and(q1, i1, q0);
7      and(q2, i2, s);
8      or(q, q1, q2);
9  endmodule
```

Syntax Variation for Module Pins Declaration in Verilog

```
1  module mux (
2      input i1,
3      input i2,
4      input s,
5      output q
6  );
7      wire q0, q1, q2;
8
8      not(q0, s);
9      and(q1, i1, q0);
10     and(q2, i2, s);
11     or(q, q1, q2);
12
12 endmodule
```

```
1  module mux (i1, i2, s, q);
2
2      input i1, i2, s;
3      output q;
4
4      wire q0, q1, q2;
5
5      not(q0, s);
6      and(q1, i1, q0);
7      and(q2, i2, s);
8      or(q, q1, q2);
9
9  endmodule
```

These two declarations are equivalent

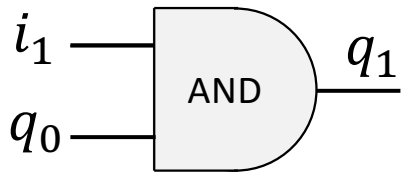
Some Verilog Operators

Operator Symbol	**Function/Purpose**	**Operands**
~	Bitwise negation	Unary
&	Bitwise AND	Binary
|	Bitwise OR	Binary
^	Bitwise XOR	Binary
^~	Bitwise XNOR	Binary
?:	Conditional	Ternary

Some Verilog Operators

Operator Symbol	Function/Purpose	Operands
~	Bitwise negation	Unary
&	Bitwise AND	Binary
|	Bitwise OR	Binary
^	Bitwise XOR	Binary
^~	Bitwise XNOR	Binary
?:	Conditional	Ternary

Example of possible implementations:

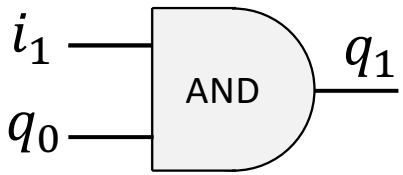


Block diagram impl.

Some Verilog Operators

Operator Symbol	Function/Purpose	Operands
~	Bitwise negation	Unary
&	Bitwise AND	Binary
|	Bitwise OR	Binary
^	Bitwise XOR	Binary
^~	Bitwise XNOR	Binary
?:	Conditional	Ternary

Example of possible implementations:



Block diagram impl.

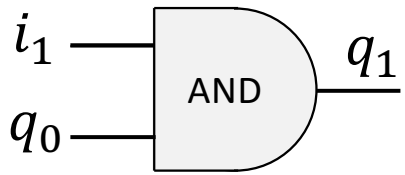
and(q1, i1, q0);

Verilog implementation by using “and” primitive

Some Verilog Operators

Operator Symbol	Function/Purpose	Operands
~	Bitwise negation	Unary
&	Bitwise AND	Binary
I	Bitwise OR	Binary
^	Bitwise XOR	Binary
^~	Bitwise XNOR	Binary
?:	Conditional	Ternary

Example:



Block diagram impl.

and(q1, i1, q0);

Verilog implementation by
using “and” primitive

assign q1 = (i1 & q0);

Verilog implementation by
a continuos assignment of an expression

Continuous Assignment in Verilog

assign q1 **=** (i1 & q0);

Whenever some input is changed, output q1 is updated as well

Multiplexer Implementation: Alternatives

Implementation by using “and”, “or”, and “not” primitives:

```
1  module mux (i1, i2, s, q);
2
3      input i1, i2, s;
4      output q;
5
6      wire q0, q1, q2;
7
8      not(q0, s);
9      and(q1, i1, q0);
10     and(q2, i2, s);
11     or(q, q1, q2);
12
13 endmodule
```

Implementation by using continuos assignments and “&”, “|”, and “~” operators:

```
1  module mux (i1, i2, s, q);
2
3      input i1, i2, s;
4      output q;
5
6      wire q0, q1, q2;
7
8      assign q0 = (~s);
9      assign q1 = (i1 & q0);
10     assign q2 = (i2 & s);
11     assign q = (q1 | q2);
12
13 endmodule
```

Multiplexer Implementation: Alternatives

Implementation by using “and”, “or”, and “not” primitives:

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output q;
4      wire q0, q1, q2;
5      not(q0, s);
6      and(q1, i1, q0);
7      and(q2, i2, s);
8      or(q, q1, q2);
9  endmodule
```

Implementation by using continuos assignments and “&”, “|”, and “~” operators:

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output q;
4      wire q0, q1, q2;
5      assign q0 = (~s);
6      assign q1 = (i1 & q0);
7      assign q2 = (i2 & s);
8      assign q = (q1 | q2);
9  endmodule
```

Multiplexer Implementation: Syntax Variations

A more *compact* program code:

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output q;
4      assign q = (i1 & ~s) | (i2 & s);
5  endmodule
```

Implementation by using continuos assignments and “&”, “|”, and “~” operators:

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output q;
4      wire q0, q1, q2;
5      assign q0 = (~s);
6      assign q1 = (i1 & q0);
7      assign q2 = (i2 & s);
8      assign q = (q1 | q2);
9  endmodule
```

Statement “assign” vs. Procedural Block “always” in Verilog

Implementation based on continuous assignment:

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output q;
4      assign q = (i1 & ~s) | (i2 & s);
5  endmodule
```

← We force q to be updated whenever i1, or i2, or s changes

Statement “assign” vs. Procedural Block “always” in Verilog

Implementation based on continuous assignment:

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output q;
4      assign q = (i1 & ~s) | (i2 & s);
5  endmodule
```

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output reg q;
4      always @ (i1 or i2 or s)
5      begin
6          q = (i1 & ~s) | (i2 & s);
7      end
8  endmodule
```

Procedural Block “always” in Verilog

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q);
2     input i1, i2, s;
3     output reg q;
4     always @ (i1 or i2 or s)
5     begin
6        q = (i1 & ~s) | (i2 & s);
7     end
8  endmodule
```

Update q whenever i1, i2, or s changes →

Procedural Block “always” in Verilog

Only “regs” can be modified within an “always” code block→

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q);
2     input i1, i2, s;
3     output reg q;
4     always @ (i1 or i2 or s)
5     begin
6        q = (i1 & ~s) | (i2 & s);
7     end
8  endmodule
```

Sensitivity List in Procedural Block “always”

Sensitivity list of “always” block → (contains i1, i2, and s pins):

Execute the body of the code block whenever one of the inputs in sensitivity list changes

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q);

2     input i1, i2, s;
3     output reg q;

4     always @ (i1 or i2 or s)
5     begin
6        q = (i1 & ~s) | (i2 & s);
7     end

8  endmodule
```

Sensitivity List in Procedural Block “always”

Sensitivity list of “always” block: → i2 is omitted

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q);
2     input i1, i2, s;
3     output reg q;
4     always @ (i1 or s)
5     begin
6        q = (i1 & ~s) | (i2 & s);
7     end
8  endmodule
```

Sensitivity List in Procedural Block “always”

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q);
2     input i1, i2, s;
3     output reg q;
4     always @ (*)
5     begin
6        q = (i1 & ~s) | (i2 & s);
7     end
8  endmodule
```

“wildcard” operator (*) at sensitivity list →

Forces to automatically determine the dependencies of q, and to update q whenever some of those input dependencies changes

Blocking and Non-Blocking Assignments in Verilog

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q);
2     input i1, i2, s;
3     output reg q;
4     always @ (*)
5     begin
6        q = (i1 & ~s) | (i2 & s);
7     end
8  endmodule
```

Assignment statement →

Blocking and Non-Blocking Assignments in Verilog

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q, q1);
2     input i1, i2, s;
3     output reg q, q1;
4     always @ (*)
5     begin
6        q <= (i1 & ~s) | (i2 & s);
7        q1 <= (i1 & ~s) & (i2 & s);
8     end
9  endmodule
```

“Blocking” assignment statements:

- First, line 6 is executed;
- After completing line 6, line 7 is executed

Blocking and Non-Blocking Assignments in Verilog

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q, q1);
2     input i1, i2, s;
3     output reg q, q1;
4     always @ (*)
5     begin
6        q <= (i1 & ~s) | (i2 & s);
7        q1 <= (i1 & ~s) & (i2 & s);
8     end
9  endmodule
```

“Non-Blocking” assignment statements:

- Lines 6 and 7 execute concurrently

Blocking and Non-Blocking Assignments in Verilog

Implementation based on “always @ ” procedural block:

```
1  module mux (i1, i2, s, q, q1);
2     input i1, i2, s;
3     output reg q, q1;
4     always @ (*)
5     begin
6        q <= (i1 & ~s) | (i2 & s);
7        q1 <= (i1 & ~s) & (i2 & s);
8     end
9  endmodule
```

Whenever possible, non-blocking assignments are prefered

Blocking and Non-Blocking Assignments in Verilog

Code with blocking assignments:

```
1  module mux (i1, i2, s, q, q1);
2      input i1, i2, s;
3      output reg q, q1;
4      always @ (*)
5      begin
6          q = (i1 & ~s) | (i2 & s);
7          q1 = ~q;
8      end
9  endmodule
```

Code with non-blocking assignments:

```
1  module mux (i1, i2, s, q, q1);
2      input i1, i2, s;
3      output reg q, q1;
4      always @ (*)
5      begin
6          q <= (i1 & ~s) | (i2 & s);
7          q1 <= ~q;
8      end
9  endmodule
```

What is the difference ?

Blocking and Non-Blocking Assignments in Verilog

Code with blocking assignments:

```
1  module mux (i1, i2, s, q, q1);
2      input i1, i2, s;
3      output reg q, q1;
4      always @ (*)
5      begin
6          q = (i1 & ~s) | (i2 & s);
7          q1 = ~q;
8      end
9  endmodule
```

Code with non-blocking assignments:

```
1  module mux (i1, i2, s, q, q1);
2      input i1, i2, s;
3      output reg q, q1;
4      always @ (*)
5      begin
6          q <= (i1 & ~s) | (i2 & s);
7          q1 <= ~q;
8      end
9  endmodule
```

Because of non-blocking assignments used, either line 6 or line 7 can be completed first;
Thus, the resulted value for q1 is non-determenistic

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output reg q;
4      always @ (*)
5      begin
6          q <= (i1 & ~s) | (i2 & s);
7      end
8  endmodule
```

If-Else Statement

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output reg q;
4      always @ (*)
5      begin
6          q <= (i1 & ~s) | (i2 & s);
7      end
8  endmodule
```

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output reg q;
4      always @ (*)
5      begin
6          if (s == 0) begin
7              q <= i1;
8          end
9          else if (s == 1) begin
10             q <= i2;
11         end
12     end
13 endmodule
```

If-Else Statement

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output reg q;
4      always @ (*)
5      begin
6          q <= (i1 & ~s) | (i2 & s);
7      end
8  endmodule
```

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output reg q;
4      always @ (*)
5      begin
6          if (s == 0) begin
7              q <= i1;
8          end
9          else if (s == 1) begin
10             q <= i2;
11         end
12     end
13 endmodule
```

If-Else Statement

```
1  module mux (i1, i2, s, q);
2      input i1, i2, s;
3      output reg q;
4      always @ (*)
5      begin
6          if (s == 0) begin
7              q <= i1;
8          end
9          else if (s == 1) begin
10             q <= i2;
11         end
12     end
13 endmodule
```

More compact code:

```
1  module mux (i1, i2, s, q);
2
2     input i1, i2, s;
3     output reg q;
4
4     always @ (*)
5     begin
6        if (s == 0) q <= i1;
9        else q <= i2;
12    end
13
13 endmodule
```

```
1  module mux (i1, i2, s, q);
2
2     input i1, i2, s;
3     output reg q;
4
4     always @ (*)
5     begin
6        if (s == 0) begin
7           q <= i1;
8        end
9        else if (s == 1) begin
10          q <= i2;
11       end
12    end
13
13 endmodule
```

If-Else Statement

```
1   module mux (i1, i2, s, q);
2       input i1, i2, s;
3       output reg q;
4       always @ (*)
5       begin
6           if (s == 0) q <= i1;
9           else q <= i2;
12      end
13  endmodule
```

If-Else Statement

```
1   module mux (i1, i2, s, q);
2
2       input i1, i2, s;
3       output reg q;
4
4       always @ (*)
5       begin
6           if (s == 0) q <= i1;
9           else q <= i2;
12      end
13
13  endmodule
```

Case Statement

```
1   module mux (i1, i2, s, q);
2
2       input i1, i2, s;
3       output reg q;
4
4       always @ (*)
5       begin
6           case (s)
7               0:   q <= i1;
8               1:   q <= i2;
9           endcase
10      end
11
11  endmodule
```

If-Else Statement

```
1   module mux (i1, i2, s, q);
2       input i1, i2, s;
3       output reg q;
4       always @ (*)
5       begin
6           if (s == 0) q <= i1;
9           else q <= i2;
12      end
13  endmodule
```

Case Statement

```
1   module mux (i1, i2, s, q);
2       input i1, i2, s;
3       output reg q;
4       always @ (*)
5       begin
6           case (s)
7               0:   q <= i1;
8               1:   q <= i2;
9           endcase
10      end
11  endmodule
```