THE UNIVERSITY OF TEXAS AT DALLAS

CS4341 Digital Logic & Computer Design

Lecture Notes 21

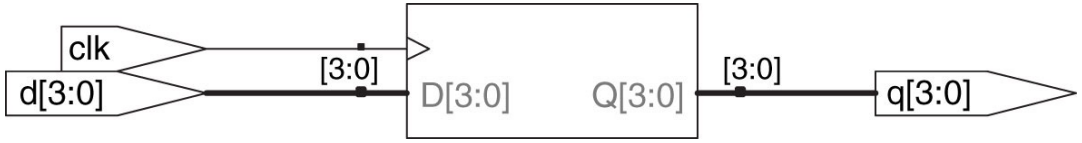
Omar Hamdy
Assistant Professor
Department of Computer Science

Review: VHDL – S. Logic: Registers

- Signals keep their old values until an “event” in the “sensitivity” list takes place that explicitly cause them to change.

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flop is
  port (clk: in STD_LOGIC;
  d: in STD_LOGIC_VECTOR (3 downto 0) ;
  q: out STD_LOGIC_VECTOR (3 downto 0)) ;
end;
architecture synth of flop is
begin
  process (clk) begin
    if clk’event and clk = ‘1’ then
      q <= d;
    end if;
  end process;
end;
```

```
process (sensitivity list) begin
  statement;
end process;
```




```
if RISING_EDGE (clk) then
  q <= d;
end if;
```

VHDL – S. Logic: Resettable Registers

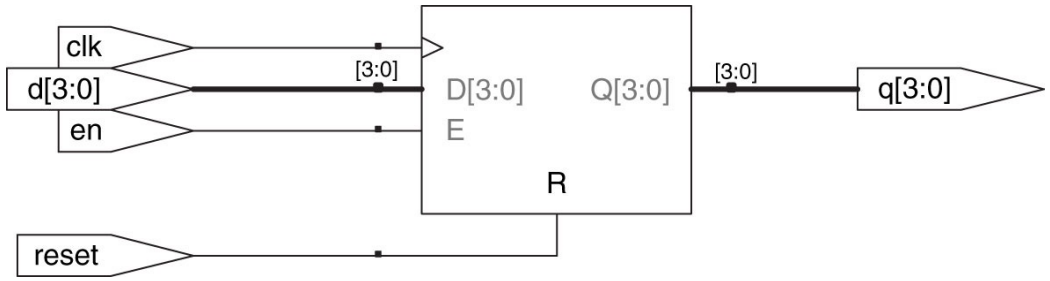
- Reset is a good practice at the start of the system.
- Reset can be synchronous or asynchronous (not easy to identify from the schematic diagram)

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is
  port (clk. reset: in STD_LOGIC;
  d: in STD_LOGIC_VECTOR (3 downto 0) ;
  q: out STD_LOGIC_VECTOR (3 downto 0)) ;
end;
architecture asynchronous of flopr is
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= "0000";
    elsif clk' event and clk  '1' then
      q <= d;
    end if;
end process;
end;
```

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopr is
  port (clk. reset: in STD_LOGIC;
  d: in STD_LOGIC_VECTOR (3 downto 0) ;
  q: out STD_LOGIC_VECTOR (3 downto 0)) ;
end;
architecture synchronous of flopr is
begin
  process (clk) begin
    if clk'event and clk = '1' then
      if reset = '1' then
        q <= "0000";
      else q <= d;
      end if;
    end if;
  end process;
end;
```

VHDL – S. Logic: Enabled Registers

- Recall, enabled registers respond to the clk only when the enable is asserted.
- The following example is for an asynchronously resettable enabled register




```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity flopenr is
  port (clk, reset, en: in STD_LOGIC;
        d: in STD_LOGIC_VECTOR (3 downto 0);
        q: out STD_LOGIC_VECTOR (3 downto 0));
  end;
architecture asynchronous of flopenr is
— — asynchronous reset
begin
  process (clk, reset) begin
    if reset = '1' then
      q <= "0000";
    elsif clk'event and clk = '1' then
      if en = '1' then
        q <= d;
      end if;
    end if;
  end process;
end;
```

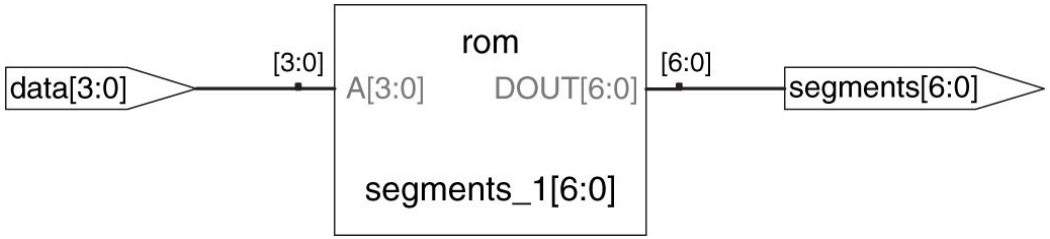
VHDL – More Combinational Logic

- Process can also be used to describe combinational circuits if all input combinations are defined (otherwise, will be sequential).
- Good practice for more complex code and for reusability
- Certain code can only appear inside a process such as case statement

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity inv is
  port (a: in STD_LOGIC_VECTOR (3 downto 0);
        y: out STD_LOGIC_VECTOR (3 downto 0));
end;
architecture proc of inv is
begin
  process (a) begin
    y <= not a;
  end process;
end;
```

VHDL – case Statement

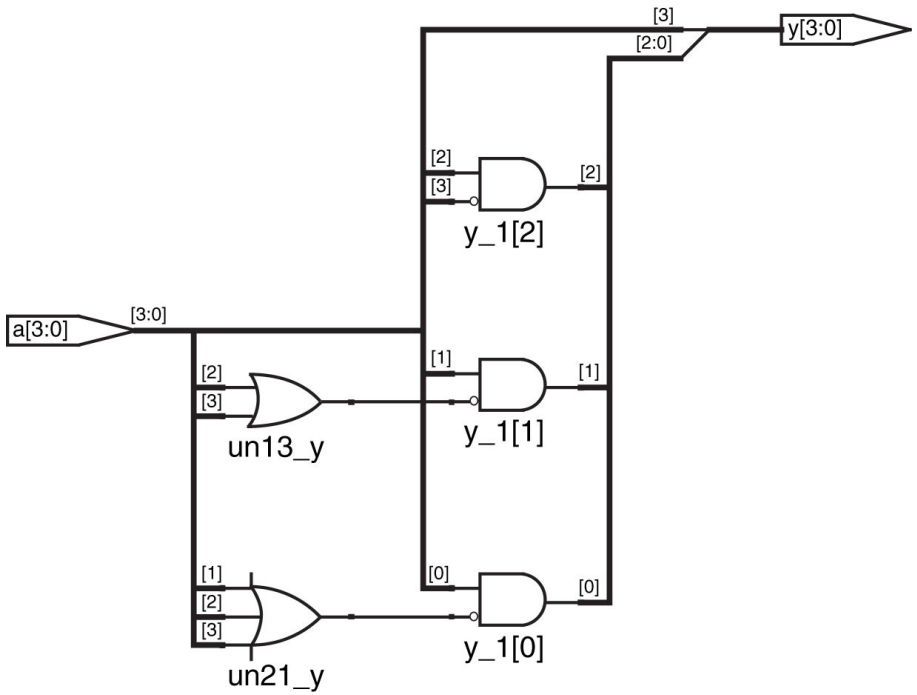
- when & select statements can also be used to implement the same function without the need of a process statement
- What happens if we do not use ”others”?




```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity seven_seg_decoder is
  port(data: in STD_LOGIC_VECTOR(3 downto 0);
       segments: out STD_LOGIC_VECTOR(6 downto 0));
end;
architecture synth of seven_seg_decoder is
begin
  process (data) begin
    case data is
      — —                                  abcdefg
      when X“0” => segments <= “1111110”;
      when X“1” => segments <= “0110000”;
      when X“2” => segments <= “1101101”;
      when X“3” => segments <= “1111001”;
      when X“4” => segments <= “0110011”;
      when X“5” => segments <= “1011011”;
      when X“6” => segments <= “1011111”;
      when X“7” => segments <= “1110000”;
      when X“8” => segments <= “1111111”;
      when X“9” => segments <= “1111011”;
      when others => segments <= “0000000”;
    end case;
  end process;
end;
```

VHDL – if Statement

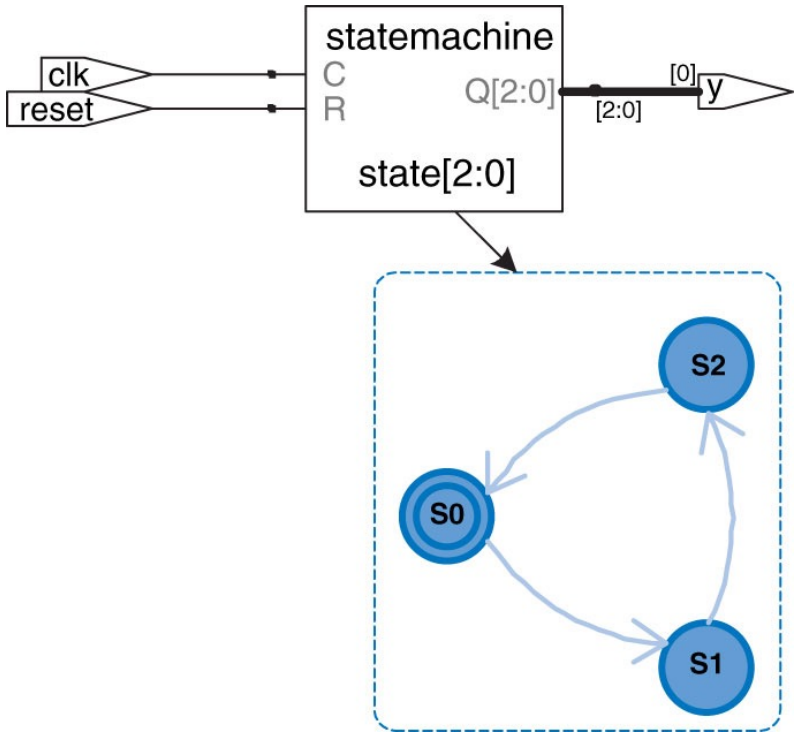
- Another way to implement selections inside the process module is the if/else statements.




```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity priority is
  port(a: in STD_LOGIC_VECTOR(3 downto 0);
       y: out STD_LOGIC_VECTOR(3 downto 0));
end;
architecture synth of priority is
begin
  process (a) begin
    if      a(3) = '1' then y <= "1000";
    elsif a(2) = '1' then y <= "0100";
    elsif a(1) = '1' then y <= "0010";
    elsif a(0) = '1' then y <= "0001";
    else                          y <= "0000";
    end if;
  end process;
end;
```

VHDL – Finite State Machine

- FSM VHDL modeling requires defining the next state and the output. The state logic, which is updated at the clk rising edge will need to be inside a process.




```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity divideby3FSM is
port (clk, reset: in STD_LOGIC;
        y: out STD_LOGIC);
end;
architecture synth of divideby3FSM is
  type statetype is (S0, S1, S2);
  signal state, nextstate: statetype;
begin
— — state register
  process (clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;
— — next state logic
  nextstate <= S1 when state = S0 else
                S2 when state = S1 else
                S0;
— — output logic
  y <= '1' when state = S0 else '0';
end;
```

VHDL – Pattern Recognition FSM Example

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity patternMoore is
  port (clk, reset: in STD_LOGIC;
        a: in STD_LOGIC;
        y: out STD_LOGIC);
end;
architecture synth of patternMoore is
  type statetype is (S0, S1, S2, S3, S4);
  signal state, nextstate: statetype;
begin
— — state register
  process (clk, reset) begin
    if reset = '1' then state <= S0;
    elsif clk'event and clk = '1' then
      state <= nextstate;
    end if;
  end process;
— — next state logic
process (state, a) begin
  case state is
    when S0 => if a = '1' then
                 nextstate <= S1;
               else nextstate <= S0;
               end if;
    when S1 =>if a = '1' then
                 nextstate <= S2;
               else nextstate <= S0;
               end if;
    when S2 =>if a = '1' then
                 nextstate <= S2;
               else nextstate <= S3;
               end if;
    when S3 => if a = '1' then
                 nextstate <= S4;
               else nextstate <= S0;
               end if;
    when S4 => if a = '1' then
                 nextstate <= S2;
               else nextstate <= S0;
               end if;
    when others => nextstate <= S0;
    end case;
  end process;
— — output logic
  y <= '1' when state = S4 else '0';
end;
```

VHDL – Testbenches

- A testbench is an HDL module that is used to test another module, called the device under test (DUT)
- The testbench contains statements to apply inputs to the DUT and, ideally, to check that the correct outputs are produced
- The input and desired output patterns are called test vectors

VHDL – Testbenches

- The following is a testbench for the function of $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
  entity simplefunction is
    port (a, b, c: in STD_LOGIC;
          y: out STD_LOGIC);
  end;
  architecture synth of simplefunction is
    begin
      y <= ((not a) and (not b) and (not c)) or
           (a and (not b) and (not c)) or
           (a and (not b) and c);
  end;
```

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity testbench1 is — — no inputs or outputs
end;
architecture sim of testbench1 is
  component simplefunction
    port (a, b, c: in STD_LOGIC;
          y: out STD_LOGIC);
    end component;
  signal a, b, c, y: STD_LOGIC;
begin
— — instantiate device under test
  dut: simplefunction port map (a, b, c, y);
— — apply inputs one at a time
  process begin
    a <= ‘0’; b <= ‘0’; c <= ‘0’;          wait for 10 ns;
    c <= ‘1’;                              wait for 10 ns;
    b <= ‘1’; c <= ‘0’;                    wait for 10 ns;
    c <= ‘1’;                              wait for 10 ns;
    a <= ‘1’; b <= ‘0’; c <= ‘0’;          wait for 10 ns;
    c <= ‘1’;                              wait for 10 ns;
    b <= ‘1’; c <= ‘0’;                    wait for 10 ns;
    c <= ‘1’;                              wait for 10 ns;
    wait; — — wait forever
  end process;
end;
```

VHDL – Self-Checking Testbenches

- VHDL support self-checking, such that you define the input test vectors and also the expected output.

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
  entity simplefunction is
    port (a, b, c: in STD_LOGIC;
            y: out STD_LOGIC);
  end;
  architecture synth of simplefunction is
    begin
      y <= ((not a) and (not b) and (not c)) or
              (a and (not b) and (not c)) or
              (a and (not b) and c);
  end;
```

```
library IEEE; use IEEE.STD_LOGIC_1164.all;
entity testbench1 is — — no inputs or outputs
end;
.
.
  process begin
    a <= ‘0’; b a <= ‘0’; c a <= ‘0’; wait for 10 ns;
      assert y = ‘1’ report “000 failed.”;
    c <= ‘1’; wait for 10 ns;
      assert  y = ‘0’ report “001 failed.”;
    b <= ‘1’; c <= ‘0’; wait for 10 ns;
      assert y = ‘0’ report “010 failed.”;
    c <= ‘1’; wait for 10 ns;
      assert y = ‘0’ report “011 failed.”;
    a <= ‘1’; b <= ‘0’; c <= ‘0’; wait for 10 ns;
      assert y = ‘1’ report “100 failed.”;
    c <= ‘1’; wait for 10 ns;
      assert y = ‘1’ report “101 failed.”;
    b <= ‘1’; c <= ‘0’; wait for 10 ns;
      assert y = ‘0’ report “110 failed.”;
    c <= ‘1’; wait for 10 ns;
      assert y = ‘0’ report “111 failed.”;
    wait; — — wait forever
  end process;
end;
```

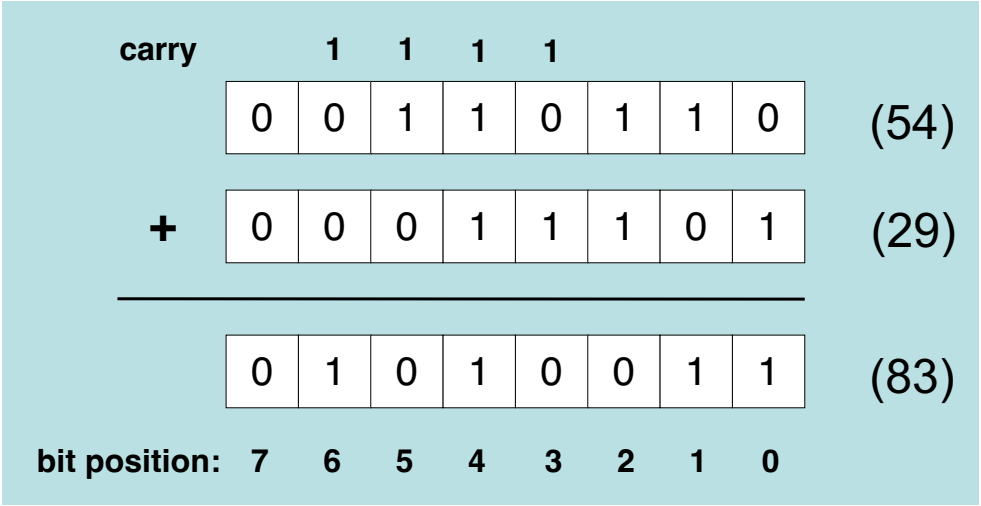
Digital Building Blocks

- Now it is time to go one-level up in our abstraction hierarchy to introduce more complex digital components, how they can be used and interconnected, rather than the internal build of each.
- These building blocks are the basis for the microprocessor design
- The scope of our study includes:
  - Arithmetic circuits: addition, subtraction, comparators, ALUs, shifters/rotators, multiplication and division
  - Sequential building blocks: counters and shift registers
  - Memory arrays: memory cells, DRAM. SRAM and ROM
  - Logic arrays: programmable logic arrays (PLAs) and Field Programmable Gate Array (FPGA)

Binary Addition

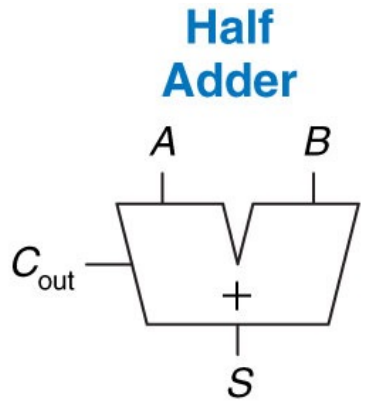
Start with the least significant bit (rightmost bit):

- Add each pair of bits
- Include the carry in the addition, if present
- Discard the carry if it exceeds the available maximum number size (overflow)



Binary Addition Circuits: Half Adder

- 1-bit half adder is the simplest addition building block. It has two 1-bit inputs A & B and two 1-bit outputs S and C_{out}.
- S is the sum of A & B, and if the addition is > 1, then C_{out} and S together represent the result of the addition (2).
- Careful study of the truth table shows that S is the A XOR B, and C_{out} is AB
- The half adder cannot be used for more than 1-bit numbers addition (why?)

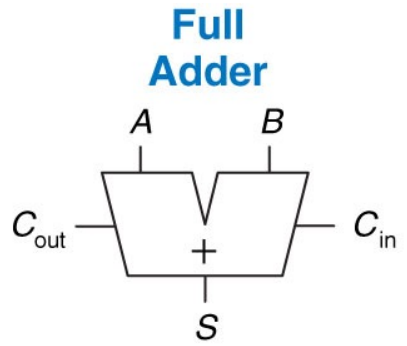


A	B	C_{out}	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$$S = A \oplus B$$
$$C_{out} = AB$$

Binary Addition Circuits: Full Adder

- A full adder introduces a 3rd input C_{in} to the half adder circuit
- This allows the circuit to accept carry from a previous bit-addition.
- Careful study of the truth table can verify the functional logic of S and C_{out}
- Recall XOR of N inputs produces a 1 if the input has odd number of 1, and 0 otherwise.
- $S = A \oplus B \oplus C$
- $C_{out} = AB + (A + B)\, C_{in}$

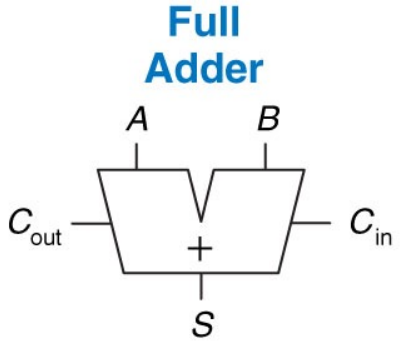


C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$

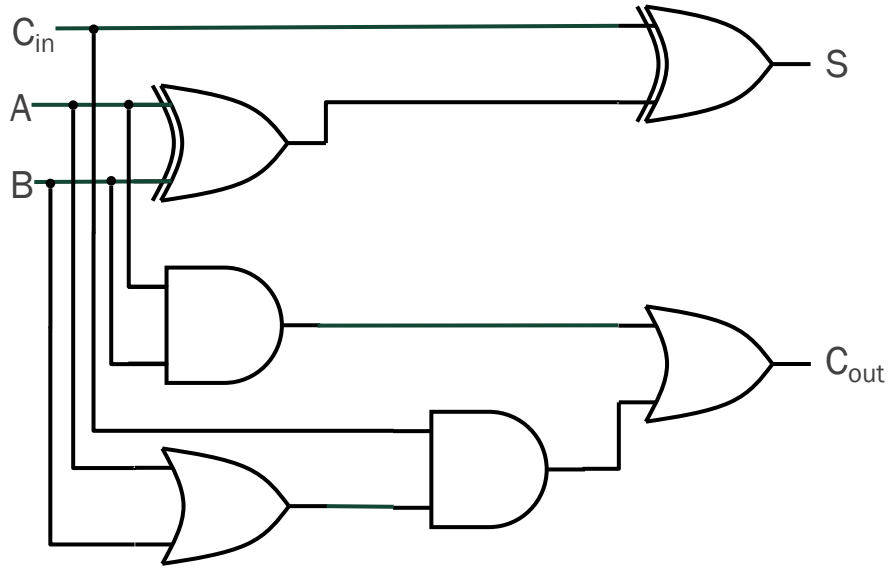
Full Adders Circuit Implementation



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

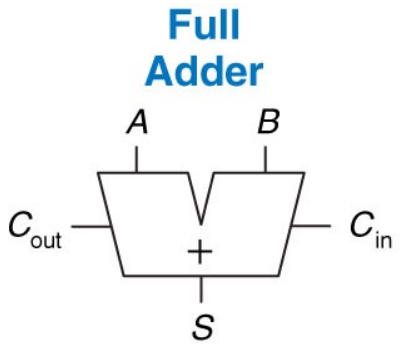
$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$



The full adder can be simplified further by noticing that C_{out} can be expressed as $AB + (A \oplus B)C_{in}$ (why?)

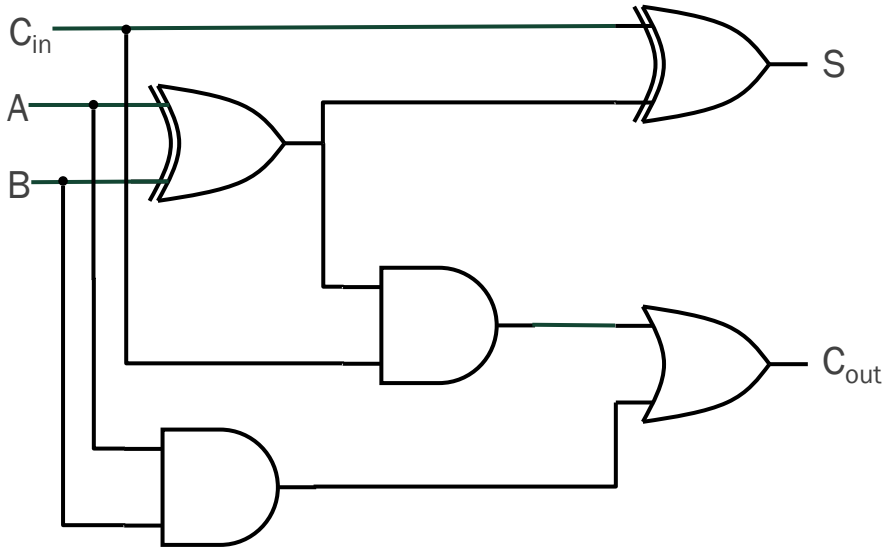
Full Adders - Simplified



C_{in}	A	B	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = AB + AC_{in} + BC_{in}$$



To Do List

- Review lecture notes
- Study Chapters 4, 5.1-3