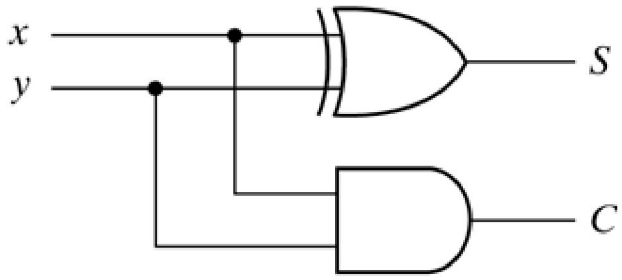
# Verilog

# Hierarchical Design

- Will show the hierarchical design approach using an example:
- A 4-bit adder in terms of full-adders
- Full-adders in terms of half-adders
- Half-adders in terms of gates

```
//HDL Example 4-2
//-----------------------------------------------
//Gate-level hierarchical description of 4-bit adder
// Description of half adder (see Fig 4-5b)
module halfadder (S,C,x,y);
   input x,y;
   output S,C;
//Instantiate primitive gates
   xor (S,x,y);
   and (C,x,y);
endmodule
```



# Hierarchical Design

```
//Description of full adder (see Fig 4-8)
module fulladder (S,C,x,y,z);
   input x,y,z;
   output S,C;
   wire S1,D1,D2; //Outputs of first XOR and two AND gates
//Instantiate the halfadder
    halfadder HA1 (S1,D1,x,y),
              HA2 (S,D2,S1,z);
    or g1(C,D2,D1);
endmodule
```

```
//HDL Example 4-2
//-----------------------------
//Gate-level hierarchical descri
// Description of half adder (se
module halfadder (S,C,x,y);
   input x,y;
   output S,C;
//Instantiate primitive gates
   xor (S,x,y);
   and (C,x,y);
endmodule
```

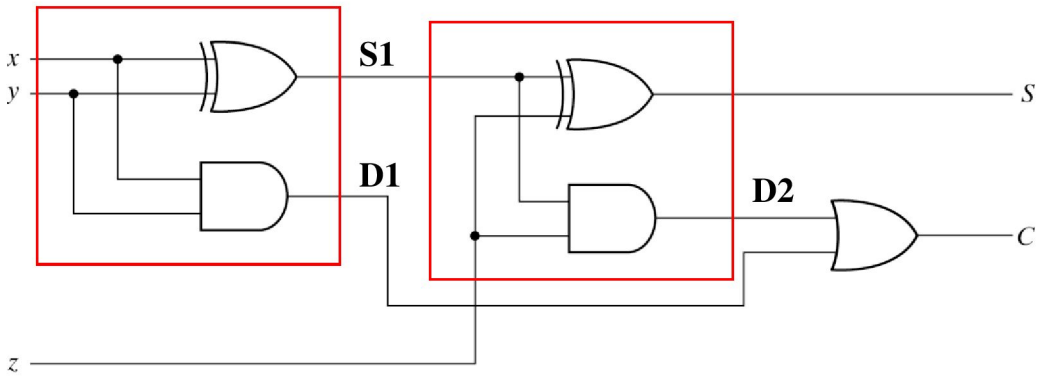



Fig. 4-8 Implementation of Full Adder with Two Half Adders and an OR Gate

# Hierarchical Design

```
//Description of 4-bit adder (see Fig 4-9)
module _4bit_adder (S,C4,A,B,C0);
   input [3:0] A,B;
   input C0;
   output [3:0] S;
   output C4;
   wire C1,C2,C3;  //Intermediate carries
//Instantiate the fulladder
   fulladder  FA0 (S[0],C1,A[0],B[0],C0),
              FA1 (S[1],C2,A[1],B[1],C1),
              FA2 (S[2],C3,A[2],B[2],C2),
              FA3 (S[3],C4,A[3],B[3],C3);
endmodule
```

```
//Description of full adder (see Fig
module fulladder (S,C,x,y,z);
   input x,y,z;
   output S,C;
   wire S1,D1,D2; //Outputs of first
//Instantiate the halfadder
    halfadder HA1 (S1,D1,x,y),
              HA2 (S,D2,S1,z);
    or g1(C,D2,D1);
endmodule
```

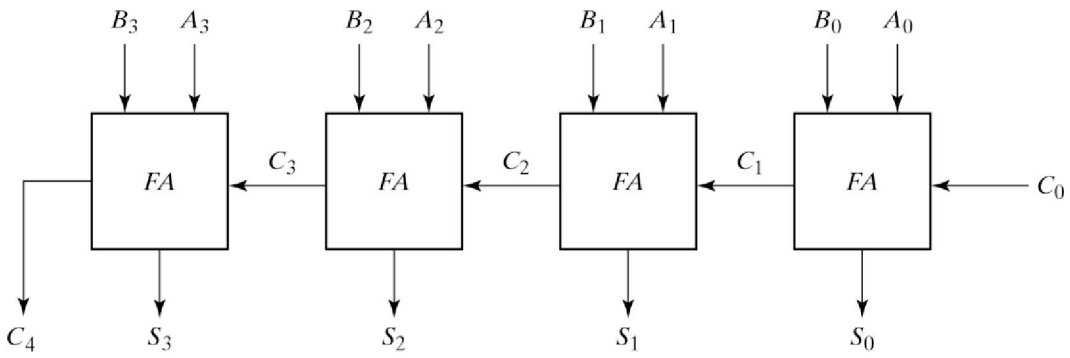



Fig. 4-9 4-Bit Adder

# Dataflow Description

- **Uses some operators and signal assignments**
  - "concat" appends two operands and makes a larger one

| operator | Operation |
|---|---|
| + | Binary addition |
| - | Binary subtraction |
| & | Bitwise and |
| \| | Bitwise or |
| ^ | Bitwise xor |
| ~ | Bitwise not |
| == | Equality |
| > | Greater |
| < | Less |
| { } | Concatenation |
| ?: | Conditional |

# Decoder

```
//HDL Example 4-3
//-----------------------------------
//Dataflow description of a 2-to-4-line decoder
//See Fig.4-19
module decoder_df (A,B,E,D);
   input A,B,E;
   output [0:3] D;
   assign D[0] = ~(~A & ~B & ~E),
          D[1] = ~(~A & B & ~E),
          D[2] = ~(A & ~B & ~E),
          D[3] = ~(A & B & ~E);
endmodule
```

- continuous assignment:
  - assigns a value to a net (`wire`, `output`)

# MUX

```
assign Y = (A & S) | (B & ~S);
```

```
//HDL Example 4-6
//---------------------------------------
//Dataflow description of 2-to-1-line multiplexer
module mux2x1_df (A,B,select,OUT);
   input A,B,select;
   output OUT;
   assign OUT = select ? A : B;
endmodule
```

# 4-Bit Adder

```
//HDL Example 4-4
//-------------------------------------
//Dataflow description of 4-bit adder
module binary_adder (A,B,Cin,SUM,Cout);
   input [3:0] A,B;
   input Cin;
   output [3:0] SUM;
   output Cout;
   assign {Cout,SUM} = A + B + Cin;
endmodule
```

- + can be used to add binary numbers
- Cout and SUM are concatenated

# 4-Bit Comparator

```
//HDL Example 4-5
//---------------------------------
//Dataflow description of a 4-bit comparator.
module magcomp (A,B,ALTB,AGTB,AEQB);
   input [3:0] A,B;
   output ALTB,AGTB,AEQB;
   assign ALTB = (A < B),
          AGTB = (A > B),
          AEQB = (A == B);
endmodule
```

- Comparison operators can be used to compare n-bit numbers

# Behavioral Modeling

```
//HDL Example 4-7
//--------------------------------
//Behavioral description of 2-to-1-line multiplexer
module mux2x1_bh(A,B,select,OUT);
   input A,B,select;
   output OUT;
   reg OUT;
   always @ (select or A or B)
         if (select == 1) OUT = A;
         else OUT = B;
endmodule
```

- Uses **`always`** and an expression:
  - Change in any variable after `@` → runs again
  - Statements within **`always`**: sequential
    - e.g., if then else
  - Output of the construct must be **`reg`**
    - Reminder: A **`reg`** net keeps its value until a new value is assigned

# 4:1 MUX (Behavioral)

```
//HDL Example 4-8
//-----------------------------------
//Behavioral description of 4-to-1- line multiplexer
//Describes the function table of  Fig. 4-25(b).
module mux4x1_bh (i0,i1,i2,i3,select,y);
   input i0,i1,i2,i3;
   input [1:0] select;
   output y;
   reg y;
   always @ (i0 or i1 or i2 or i3 or select)
            case (select)
               2'b00: y = i0;
               2'b01: y = i1;
               2'b10: y = i2;
               2'b11: y = i3;
            endcase
endmodule
```

- If “always” used for combinational circuits, then:
  1. All input variables must be used in the sensitivity list
  2. For all combinations of the input, the output has to be specified
  3. Another condition which will be explained later

# Testbench

- **Testbench:**
  - Applies inputs and is used to see the outputs
  - Uses **`initial`** to drive the values
    - runs once
  - **`always`** runs many times

```
initial
        begin
                    A = 1; B = 0;
           #10      A = 0;
           #20      A = 1; B = 1;
        end
```

# Testbench

- 001 is added to D seven times with 10 time units in between

```
initial
        begin
                    D = 3'b000;
                    repeat (7)
           #10      D = D + 3'b001;
         end
```

# Testbench Structure

- Testbench items:
  1. test module name
  2. reg and wire declarations
  3. instantiation of circuit under test
  4. initial statement
  5. always statement
  6. outputs display
- No input/output ports:
  - Signals are applied through local **`reg`**
  - Outputs to be displayed are declared as **`wire`**

# Testbench

- You can see waveforms
- Values can be displayed by Verilog system tasks:
  - $display: displays variable value once (with newline)
    - $display (format, argument list);
  - $write: as $display without newline
  - $monitor: displays any variable that is changed during simulation
  - $time: shows simulation time
  - $finish: finishes the simulation

```
$display ("%d %b %b", C, A, B);
$display ("time = %0d A = %b B = %b", $time, A, B)
```

# Testbench: Example

- Testbench items:
  1. test module name,
  2. reg and wire declarations,
  3. instantiation of circuit under test,
  4. initial statement,
  5. always statement,
  6. outputs display,

```
//HDL Example 4-9
//--------------------------
//Stimulus for mux2x1_df.
module testmux;
  reg TA,TB,TS;  //inputs for mux
  wire Y;        //output from mux
  mux2x1_df mx (TA,TB,TS,Y); // inst'te mux
     initial
        begin
              TS = 1; TA = 0; TB = 1;
          #10 TA = 1; TB = 0;
          #10 TS = 0;
          #10 TA = 0; TB = 1;
        end
     initial
      $monitor("select = %b A = %b B = %b OUT = %b time = %0d",
               TS, TA, TB, Y, $time);
endmodule

//Dataflow description of 2-to-1-line multiplexer
//from Example 4-6
module mux2x1_df (A,B,select,OUT);
   input A,B,select;
   output OUT;
   assign OUT = select ? A : B;
endmodule
```

# Testbench

- `%0d` is better for `time` because `%d` reserves 10 positions
- MUX inputs: `reg`
- MUX output: `wire`

# Testbench: Example 2

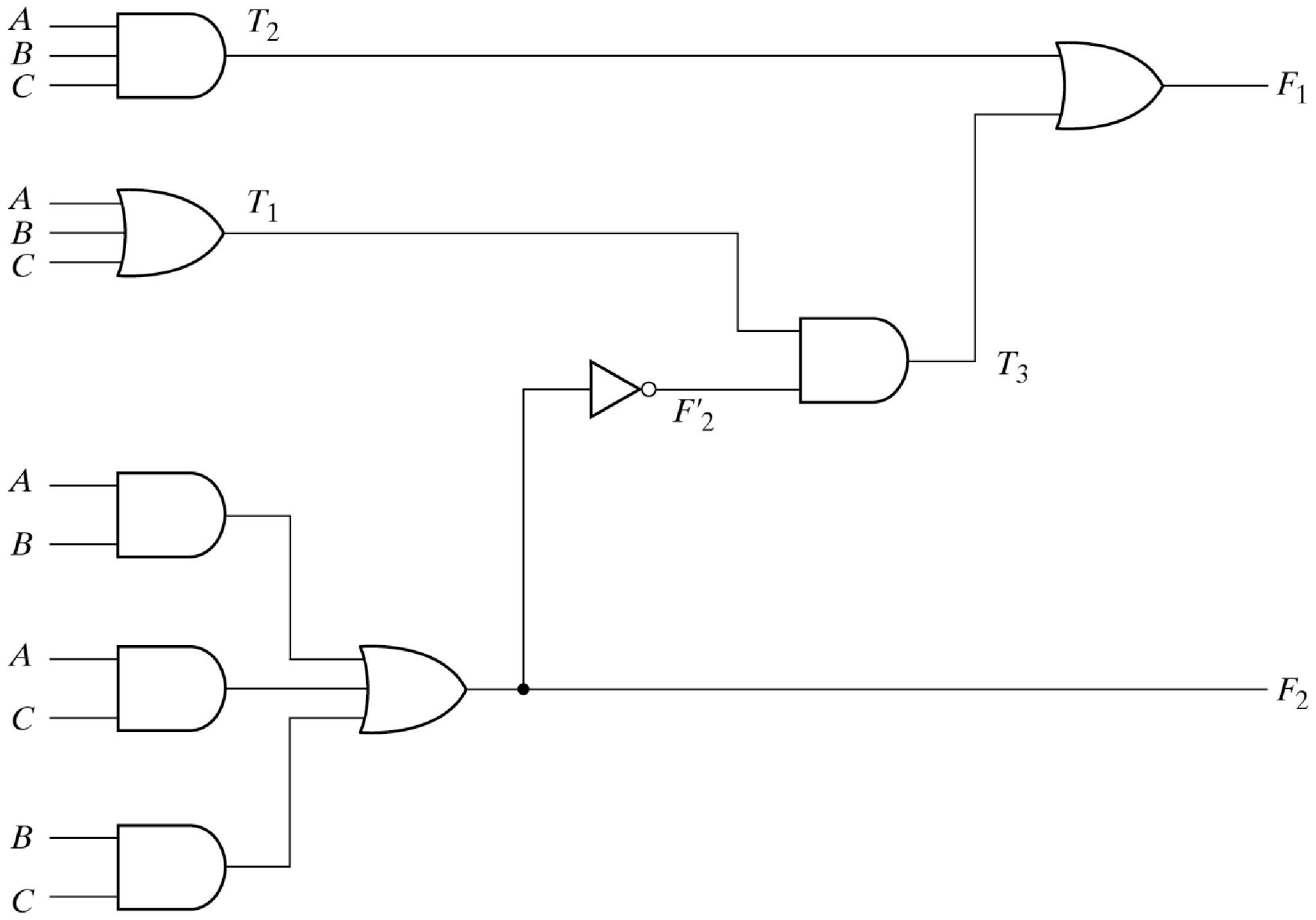



Fig. 4-2 Logic Diagram for Analysis Example

# Testbench: Example2

```
//HDL Example 4-10
//----------------------------------------
//Gate-level description of circuit of Fig. 4-2
module analysis (A,B,C,F1,F2);
   input   A,B,C;
   output  F1,F2;
   wire    T1,T2,T3,F2not,E1,E2,E3;
   or  g1 (T1,A,B,C);
   and g2 (T2,A,B,C);
   and g3 (E1,A,B);
   and g4 (E2,A,C);
   and g5 (E3,B,C);
   or  g6 (F2,E1,E2,E3);
   not g7 (F2not,F2);
```

```
//Stimulus to analyze the circuit
module test_circuit;
  reg [2:0]D;
  wire F1,F2;
  analysis fig42(D[2],D[1],D[0],F1,F2);
  initial
    begin
     D = 3'b000;
     repeat(7)
      #10 D = D + 1'b1;
    end
  initial
     $monitor ("ABC = %b F1 = %b F2 =%b ",D,
F1, F2);
endmodule
```

```
Simulation Log:

ABC = 000 F1 = 0 F2 =0
ABC = 001 F1 = 1 F2 =0
ABC = 010 F1 = 1 F2 =0
ABC = 011 F1 = 0 F2 =1
ABC = 100 F1 = 1 F2 =0
ABC = 101 F1 = 0 F2 =1
ABC = 110 F1 = 0 F2 =1
ABC = 111 F1 = 1 F2 =1
```