ECS 409: Computer Organization – Assignment 1

Adheesh Trivedi

2025-09-07

Problem 1: 3-Input AND and OR Gates

Structural versions built only from primitive gates (`AND`, `OR`) plus wires.

1.1 Verilog Code

Each module forms the 3-input function by first combining A & B, then feeding C.

Below is the verilog code for the 3-input `AND` and `OR` gates.

```
module AND(
  input A, input B, input C,
  output and_ABC
);

  wire t;

  and AB(t, A, B);
  and tC(and_ABC, t, C);

endmodule
```

3-input AND gate

```
module OR(
  input A, input B, input C,
  output or_ABC
);

  wire t;

  or AB(t, A, B);
  or tC(or_ABC, t, C);

endmodule
```

3-input OR gate

1.2 Testbench

Drives all 8 input combinations (every 10 ns), monitors outputs, and dumps VCD.

```
`timescale 1ns / 1ps

module TEST();

  reg A, B, C;
  wire and_ABC, or_ABC;

  AND op1(A, B, C, and_ABC);
  OR op2(A, B, C, or_ABC);


  initial begin
    $monitor(
      {
        " > A=%b, B=%b, C=%b\n",
        " | | AND=%b, OR=%b"
      },
      A, B, C, and_ABC, or_ABC
    );
```

```
    $dumpfile("q1.vcd");
    $dumpvars(0, TEST);

    A = 0; B = 0; C = 0; #10;
    A = 1; B = 0; C = 0; #10;
    A = 0; B = 1; C = 0; #10;
    A = 1; B = 1; C = 0; #10;
    A = 0; B = 0; C = 1; #10;
    A = 1; B = 0; C = 1; #10;
    A = 0; B = 1; C = 1; #10;
    A = 1; B = 1; C = 1; #10;

  end

endmodule
```

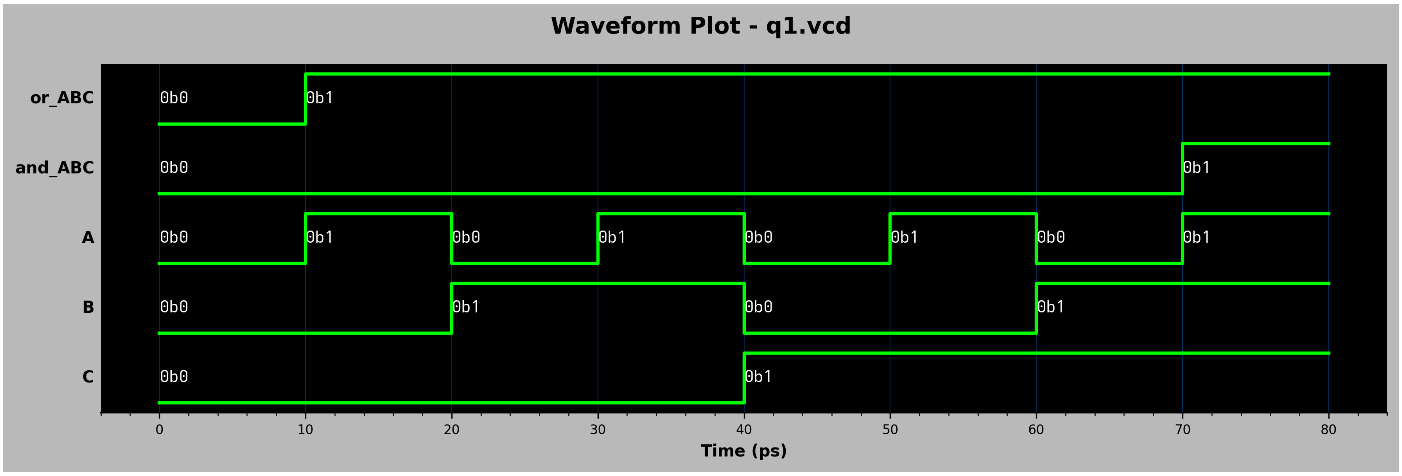
Testbench code

1.3 Simulation Result

Outputs match truth tables: AND is 1 only at A=B=C=1; OR is 1 for any high input. Sequence below confirms every combination exactly once (ascending by binary value).

```
➜ vvp q1.vvp
VCD info: dumpfile q1.vcd opened for output.
 > A=0, B=0, C=0
 | | AND=0, OR=0
 > A=1, B=0, C=0
 | | AND=0, OR=1
 > A=0, B=1, C=0
 | | AND=0, OR=1
 > A=1, B=1, C=0
 | | AND=0, OR=1
 > A=0, B=0, C=1
 | | AND=0, OR=1
 > A=1, B=0, C=1
 | | AND=0, OR=1
 > A=0, B=1, C=1
 | | AND=0, OR=1
 > A=1, B=1, C=1
 | | AND=1, OR=1
```

Terminal output of the simulation



Waveform (timing + annotated values)

Problem 2: 4-Bit Magnitude Comparator

Structural comparator built from four bitwise XNORs feeding a cascade of AND gates to assert equality only when every bit matches.

2.1 Verilog Code

The equality vector is formed first, then reduced via AND tree for the single compare output.

```verilog
module COMPARE(
  input [3:0] A, input [3:0] B,
  output [3:0] comp_AB, output comp
);

  wire t1, t2;

  xnor (comp_AB[3], A[3], B[3]);
  xnor (comp_AB[2], A[2], B[2]);
  xnor (comp_AB[1], A[1], B[1]);
  xnor (comp_AB[0], A[0], B[0]);

  and (t1, comp_AB[3], comp_AB[2]);
  and (t2, t1, comp_AB[1]);
  and (comp, t2, comp_AB[0]);

endmodule
```

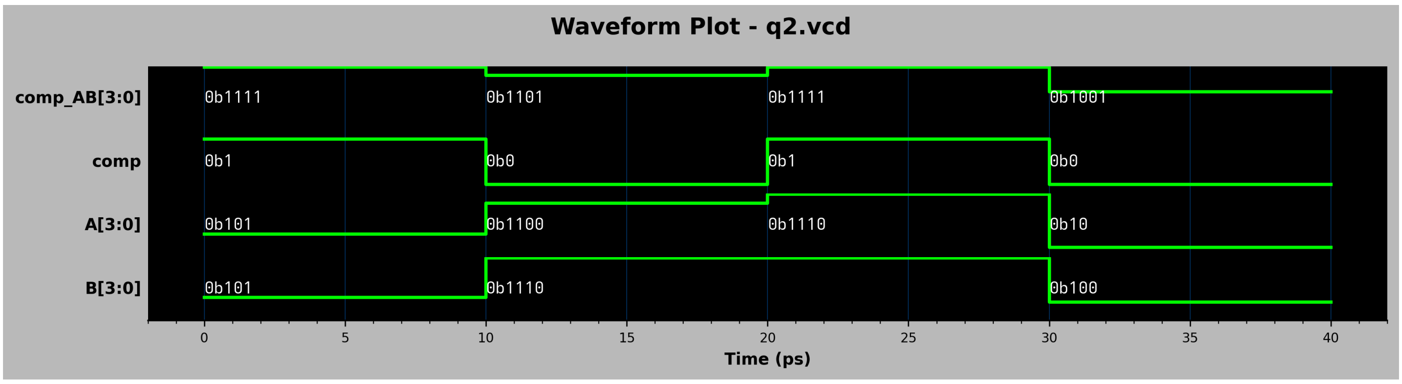
COMPARE module

```verilog
`timescale 1ns / 1ps

module TEST();

  reg [3:0] A, B;
  wire [3:0] comp_AB;
  wire comp;

  COMPARE op1(A, B, comp_AB, comp);


  initial begin

    $monitor(
      {
        " > A=%b, B=%b\n",
        " | | XNOR=%b, COMPARE=%b"
      },
      A, B, comp_AB, comp
    );
    $dumpfile("q2.vcd");
    $dumpvars(0, TEST);

    A = 4'b0101; B = 4'b0101; #10;
    A = 4'b1100; B = 4'b1110; #10;
    A = 4'b1110; B = 4'b1110; #10;
    A = 4'b0010; B = 4'b0100; #10;

  end

endmodule
```

Testbench

2.2 Simulation Result

Each stimulus pair matches the expected XNOR bit mask; COMPARE=1 only when all four bits are equal (0101==0101 and 1110==1110 cases).



Waveform (bitwise matches and final compare)

```
➜ vvp q2.vvp
VCD info: dumpfile q2.vcd opened for output.
 > A=0101, B=0101
 | | XNOR=1111, COMPARE=1
 > A=1100, B=1110
 | | XNOR=1101, COMPARE=0
 > A=1110, B=1110
 | | XNOR=1111, COMPARE=1
 > A=0010, B=0100
 | | XNOR=1001, COMPARE=0
```

Terminal output of the simulation

Problem 3: Half/Full Adder & Half/Full Subtractor

Structural ripple-style constructions: FULL_ADD uses two XORs plus AND/OR network; HALF_ADD is a FULL_ADD with Cin=0. Subtractor is implemented the other way, with borrow logic using two HALF_SUB stages and OR for final borrow in FULL_SUB.

3.1 Verilog Code (Addition)

```
module FULL_ADD (
  input A, input B, input C,
  output sum_ABC, output carry_ABC
);

  wire xor_AB;

  // sum = A ^ B ^ C
  xor (xor_AB, A, B);
  xor (sum_ABC, xor_AB, C);

  wire and_AB, and_C_xor_AB;

  and (and_AB, A, B);
  and (and_C_xor_AB, C, xor_AB);

  // carry = (A & B) | C & (A ^ B)
  or (carry_ABC, and_AB, and_C_xor_AB);

endmodule
```

FULL_ADD

```
module HALF_ADD (
  input A, input B,
  output sum_AB, output carry_AB
);

  FULL_ADD g1 (A, B, 0, sum_AB, carry_AB);

endmodule
```

HALF_ADD

3.2 Testbench (Addition)

```
`timescale 1ns / 1ps

module TEST();

  reg A, B, C;
  wire sum_ABC, carry_ABC;
  wire sum_AB, carry_AB;

  HALF_ADD g1 (A, B, sum_AB, carry_AB);
  FULL_ADD g2 (A, B, C, sum_ABC, carry_ABC);


  initial begin
    $monitor(
      {
        " > A=%b, B=%b, C=%b\n",
        " | > Half Add (A, B)\n",
        " | | | s=%b, c=%b\n",
        " | > Full Add (A, B, C)\n",
        " | | | s=%b, c=%b"
      },
```

```
      A, B, C, sum_AB, carry_AB,
      sum_ABC, carry_ABC
    );
    $dumpfile("q3_add.vcd");
    $dumpvars(0, TEST);

    A = 0; B = 0; C = 0; #10;
    A = 1; B = 0; C = 0; #10;
    A = 0; B = 1; C = 0; #10;
    A = 1; B = 1; C = 0; #10;
    A = 0; B = 0; C = 1; #10;
    A = 1; B = 0; C = 1; #10;
    A = 0; B = 1; C = 1; #10;
    A = 1; B = 1; C = 1; #10;

  end

endmodule
```

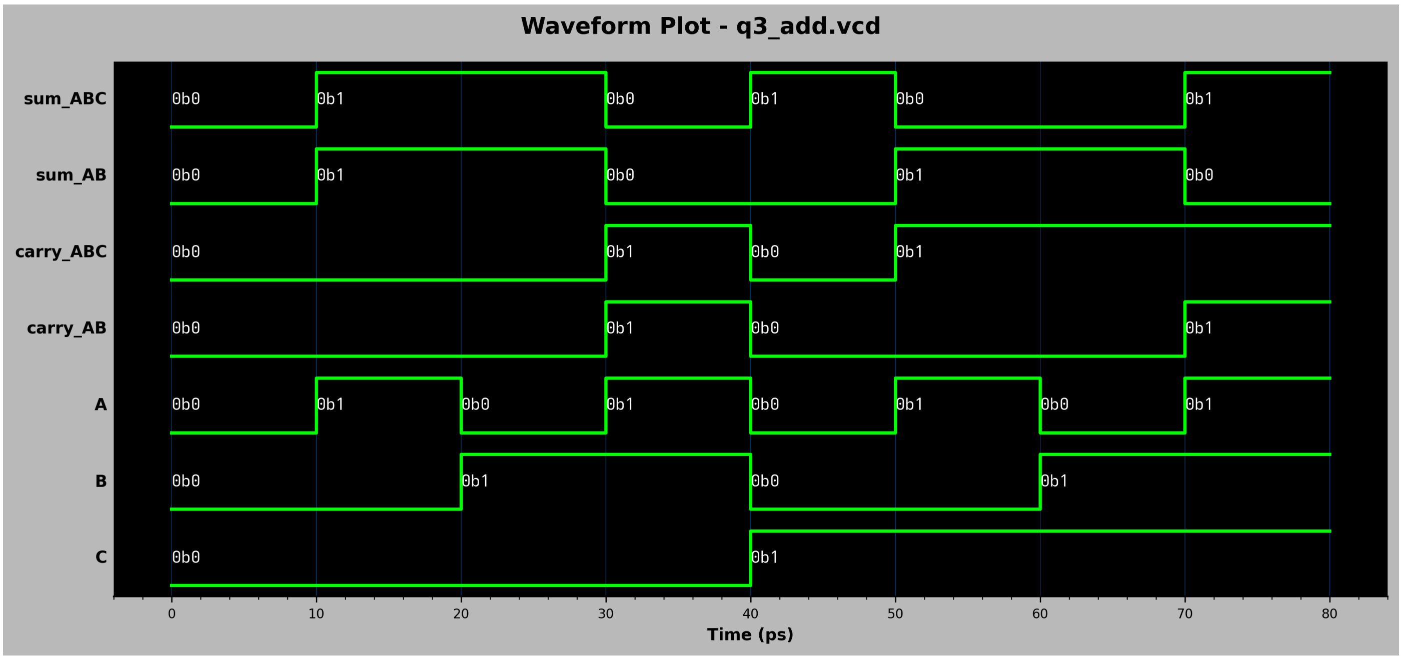
Adder testbench code

3.3 Simulation Result (Addition)

All 8 input combinations confirm HALF_ADD and FULL_ADD sums and carries match truth table (carry only when ≥2 inputs high).

```
→ vvp q3_add.vvp
VCD info: dumpfile q3_add.vcd opened for output.
 > A=0, B=0, C=0
 | > Half Add (A, B)
 | | | s=0, c=0
 | > Full Add (A, B, C)
 | | | s=0, c=0
 > A=1, B=0, C=0
 | > Half Add (A, B)
 | | | s=1, c=0
 | > Full Add (A, B, C)
 | | | s=1, c=0
 > A=0, B=1, C=0
 | > Half Add (A, B)
 | | | s=1, c=0
 | > Full Add (A, B, C)
 | | | s=1, c=0
 > A=1, B=1, C=0
 | > Half Add (A, B)
 | | | s=0, c=1
 | > Full Add (A, B, C)
 | | | s=0, c=1
 > A=0, B=0, C=1
 | > Half Add (A, B)
 | | | s=0, c=0
 | > Full Add (A, B, C)
 | | | s=1, c=0
 > A=1, B=0, C=1
 | > Half Add (A, B)
 | | | s=1, c=0
 | > Full Add (A, B, C)
 | | | s=0, c=1
 > A=0, B=1, C=1
 | > Half Add (A, B)
 | | | s=1, c=0
 | > Full Add (A, B, C)
 | | | s=0, c=1
 > A=1, B=1, C=1
 | > Half Add (A, B)
 | | | s=0, c=1
 | > Full Add (A, B, C)
 | | | s=1, c=1
```

Adder terminal output



Adder waveform

3.4 Verilog Code (Subtraction)

```
module HALF_SUB (
  input A, input B,
  output dif_AB, output brrw_AB
);

  // diff = A ^ B
  xor (dif_AB, A, B);

  wire not_A, and_AB, and_C_xor_AB;

  // borrow = ~A | B
  not (not_A, A);
  and (brrw_AB, not_A, B);

endmodule
```

HALF_SUB

```
module FULL_SUB (
  input A, input B, input Din,
  output dif_ABC, output brrw_ABC
);

  wire dif_AB, brrw_AB, brrw2;

  HALF_SUB g1 (A, B, dif_AB, brrw_AB);
  HALF_SUB g2 (dif_AB, Din, dif_ABC, brrw2);
  or (brrw_ABC, brrw_AB, brrw2);

endmodule
```

FULL_SUB

3.5 Testbench (Subtraction)

```
`timescale 1ns / 1ps

module TEST();

  reg A, B, C;
  wire dif_ABC, brrw_ABC;
  wire dif_AB, brrw_AB;

  HALF_SUB g1 (A, B, dif_AB, brrw_AB);
  FULL_SUB g2 (A, B, C, dif_ABC, brrw_ABC);


  initial begin
    $monitor(
      {
        " > A=%b, B=%b, C=%b\n",
        " | > Half Add (A, B)\n",
```

```
      A, B, C, dif_AB, brrw_AB,
      dif_ABC, brrw_ABC
    );
    $dumpfile("q3_sub.vcd");
    $dumpvars(0, TEST);

    A = 0; B = 0; C = 0; #10;
    A = 1; B = 0; C = 0; #10;
    A = 0; B = 1; C = 0; #10;
    A = 1; B = 1; C = 0; #10;
    A = 0; B = 0; C = 1; #10;
    A = 1; B = 0; C = 1; #10;
    A = 0; B = 1; C = 1; #10;
    A = 1; B = 1; C = 1; #10;

  end
```

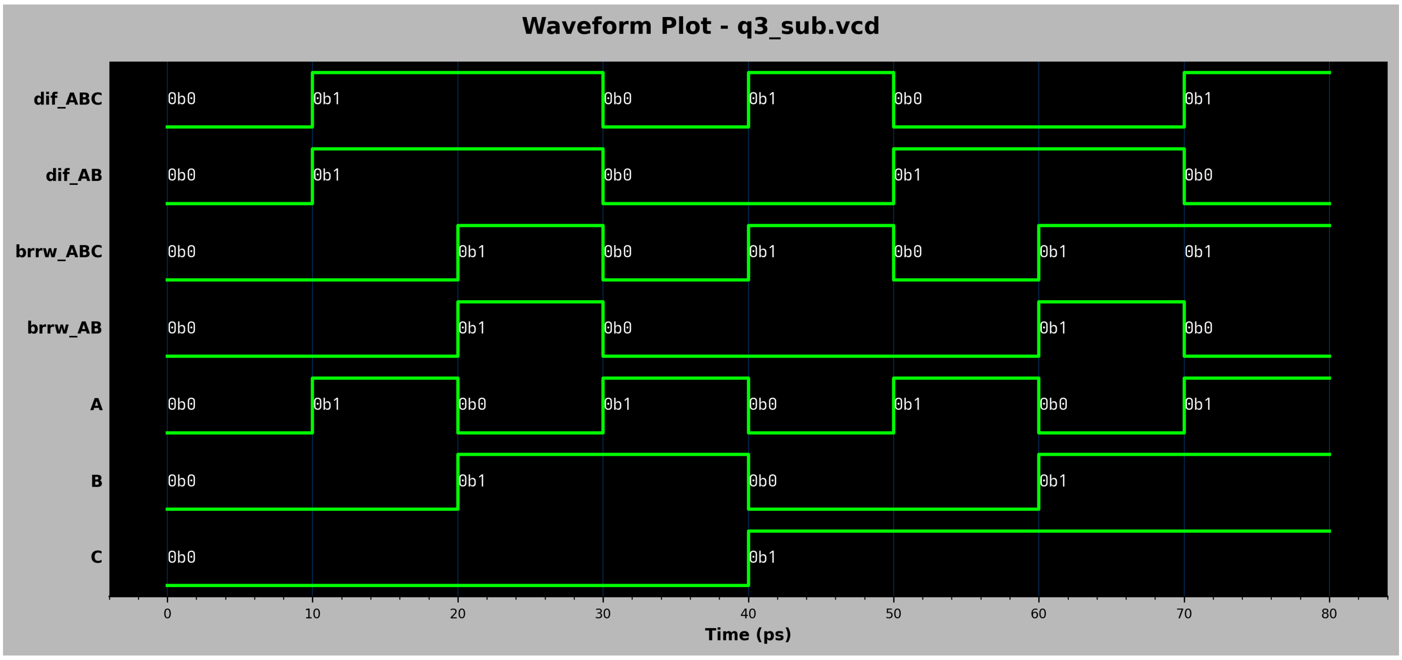
```
        " | | | s=%b, c=%b\n",
        " | > Full Add (A, B, C)\n",
        " | | | s=%b, c=%b"
      },
```

```
endmodule
```

Subtractor testbench code

3.6 Simulation Result (Subtraction)

Borrow asserted exactly when needed for A−B−C.



Subtractor waveform

```
➜ vvp q3_sub.vvp
VCD info: dumpfile q3_sub.vcd opened for output.
 > A=0, B=0, C=0
 | > Half Add (A, B)
 | | | s=0, c=0
 | > Full Add (A, B, C)
 | | | s=0, c=0
 > A=1, B=0, C=0
 | > Half Add (A, B)
 | | | s=1, c=0
 | > Full Add (A, B, C)
 | | | s=1, c=0
 > A=0, B=1, C=0
 | > Half Add (A, B)
 | | | s=1, c=1
 | > Full Add (A, B, C)
 | | | s=1, c=1
 > A=1, B=1, C=0
 | > Half Add (A, B)
 | | | s=0, c=0
 | > Full Add (A, B, C)
 | | | s=0, c=0
 > A=0, B=0, C=1
 | > Half Add (A, B)
 | | | s=0, c=0
 | > Full Add (A, B, C)
 | | | s=1, c=1
 > A=1, B=0, C=1
 | > Half Add (A, B)
 | | | s=1, c=0
 | > Full Add (A, B, C)
 | | | s=0, c=0
 > A=0, B=1, C=1
 | > Half Add (A, B)
 | | | s=1, c=1
 | > Full Add (A, B, C)
 | | | s=0, c=1
 > A=1, B=1, C=1
 | > Half Add (A, B)
 | | | s=0, c=0
 | > Full Add (A, B, C)
 | | | s=1, c=1
```

Subtractor terminal output

Problem 4: 3-to-8 Decoder & 4-to-1 Multiplexer

Decoder uses input and their complements to drive one-hot outputs. MUX selects among four inputs using minterms and OR reduction.

4.1 Verilog Code (Decoder)

```
module DEC (
  input A, input B, input C,
  output [7:0] D
);

  wire not_A, not_B, not_C;
  not (not_A, A);
  not (not_B, B);
  not (not_C, C);

  and (D[0], not_A, not_B, not_C);
  and (D[1], not_A, not_B, C);
  and (D[2], not_A, B, not_C);
  and (D[3], not_A, B, C);
  and (D[4], A, not_B, not_C);
  and (D[5], A, not_B, C);
  and (D[6], A, B, not_C);
  and (D[7], A, B, C);

endmodule
```

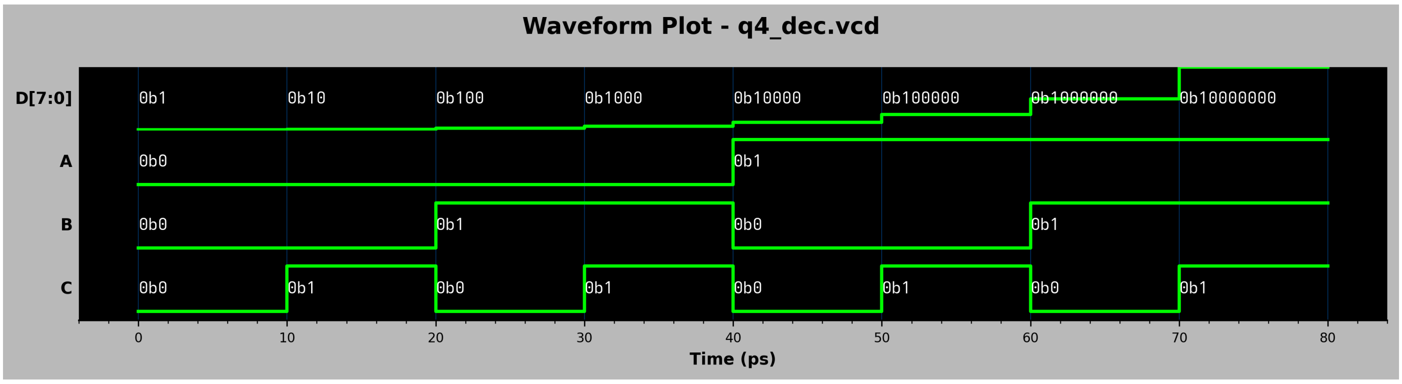
3-to-8 Decoder

```
`timescale 1ns / 1ps

module TEST();

  reg A, B, C;
  wire [7:0] D;

  DEC dut (A, B, C, D);

  initial begin
    $monitor(
      {
        " > A=%b, B=%b, C=%b\n",
        " | | D=%b"
      },
      A, B, C, D
    );
    $dumpfile("q4_dec.vcd");
    $dumpvars(0, TEST);

    A = 0; B = 0; C = 0; #10;
    A = 0; B = 0; C = 1; #10;
    A = 0; B = 1; C = 0; #10;
    A = 0; B = 1; C = 1; #10;
    A = 1; B = 0; C = 0; #10;
    A = 1; B = 0; C = 1; #10;
    A = 1; B = 1; C = 0; #10;
    A = 1; B = 1; C = 1; #10;

  end

endmodule
```

Decoder Testbench

4.2 Simulation Result (Decoder)

Output word cycles through single high bit in binary order confirming one-hot behavior for all input combinations. The graph for the output shows exponential growth as each successive bit is set.



Decoder waveform

```
→ vvp q4_dec.vvp
VCD info: dumpfile q4_dec.vcd opened for output.
 > A=0, B=0, C=0
 | | D=00000001
 > A=0, B=0, C=1
 | | D=00000010
 > A=0, B=1, C=0
 | | D=00000100
 > A=0, B=1, C=1
 | | D=00001000
 > A=1, B=0, C=0
 | | D=00010000
 > A=1, B=0, C=1
 | | D=00100000
 > A=1, B=1, C=0
 | | D=01000000
 > A=1, B=1, C=1
 | | D=10000000
```

Decoder terminal output

4.3 Verilog Code (Multiplexer)

```verilog
module MUX4x1 (
  input  [3:0] I,
  input  [1:0] S,
  output Y
);

  wire nS0, nS1;
  not (nS0, S[0]);
  not (nS1, S[1]);

  wire w0, w1, w2, w3;
  and (w0, I[0], nS1, nS0);
  and (w1, I[1], nS1, S[0]);
  and (w2, I[2], S[1], nS0);
  and (w3, I[3], S[1], S[0]);

  wire t1, t2;
  or (t1, w0, w1);
  or (t2, w2, w3);
  or (Y, t1, t2);
```

```verilog
`timescale 1ns / 1ps

module TEST();

  reg [3:0] I;
  reg [1:0] S;
  wire Y;

  MUX4x1 dut (I, S, Y);

  initial begin
    $monitor(
      {
        " > I=%b, S=%d\n",
        " | | Y=%b"
      },
      I, S, Y
    );
    $dumpfile("q4_mux.vcd");
    $dumpvars(0, TEST);
```

```
endmodule
```

4-to-1 MUX

```
    I = 4'b1010;
    S = 2'b00; #10;
    S = 2'b01; #10;
    S = 2'b10; #10;
    S = 2'b11; #10;

    I = 4'b0111;
    S = 2'b00; #10;
    S = 2'b01; #10;
    S = 2'b10; #10;
    S = 2'b11; #10;

  end

endmodule
```

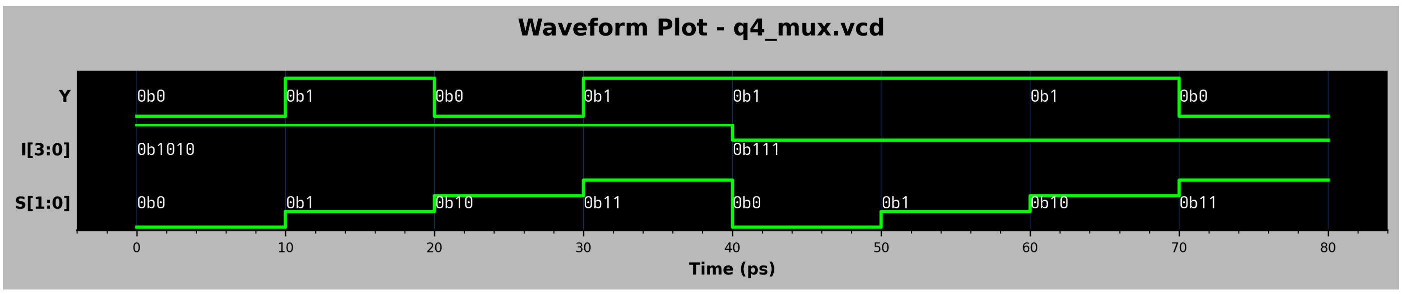
MUX Testbench

4.4 Simulation Result (Multiplexer)

For two input patterns, each select value routes the correct bit to Y; waveform shows gating signals and OR tree result.

```
➜ vvp q4_mux.vvp
VCD info: dumpfile q4_mux.vcd opened for output.
 > I=1010, S=0
 | | Y=0
 > I=1010, S=1
 | | Y=1
 > I=1010, S=2
 | | Y=0
 > I=1010, S=3
 | | Y=1
 > I=0111, S=0
 | | Y=1
 > I=0111, S=1
 | | Y=1
 > I=0111, S=2
 | | Y=1
 > I=0111, S=3
 | | Y=0
```

MUX terminal output



MUX waveform

Problem 5: 4-Bit Barrel Shifter (1- & 2-bit Rotational Shifts)

Two independent shifters are built from the same `MUX2x1` primitive: `SHIFTER1B` performs a 1-bit circular rotate, `SHIFTER2B` performs a 2-bit circular rotate. Direction (`dir`) selects left (0) or right (1) rotation; wiring of multiplexer inputs determines which original bit appears at each output position after the rotate distance.

5.1 Verilog Code

```verilog
module MUX2x1 (
  input A,
  input B,
  input S,
  output O
);

  wire not_S;

  not (not_S, S);
  and (and_1, not_S, A);
  and (and_2, S, B);
  or (O, and_1, and_2);

endmodule
```

MUX2x1

```verilog
module SHIFTER1B (
  input [3:0] I,
  // direction: 0 left, 1
right
  input dir,
  output [3:0] O
);

  MUX2x1 mux0 (
    I[2], I[0],
    dir, O[3]
  );
  MUX2x1 mux1 (
    I[1], I[3],
    dir, O[2]
  );
  MUX2x1 mux2 (
    I[0], I[2],
    dir, O[1]
  );
  MUX2x1 mux3 (
    I[3], I[1],
    dir, O[0]
  );

endmodule
```

1-bit Shifter

```verilog
module SHIFTER2B (
  input [3:0] I,
  // direction: 0 left, 1
right
  input dir,
  output [3:0] O
);

  MUX2x1 mux0 (
    I[1], I[1],
    dir, O[3]
  );
  MUX2x1 mux1 (
    I[0], I[0],
    dir, O[2]
  );
  MUX2x1 mux2 (
    I[3], I[3],
    dir, O[1]
  );
  MUX2x1 mux3 (
    I[2], I[2],
    dir, O[0]
  );

endmodule
```

2-bit Shifter

5.2 Testbench

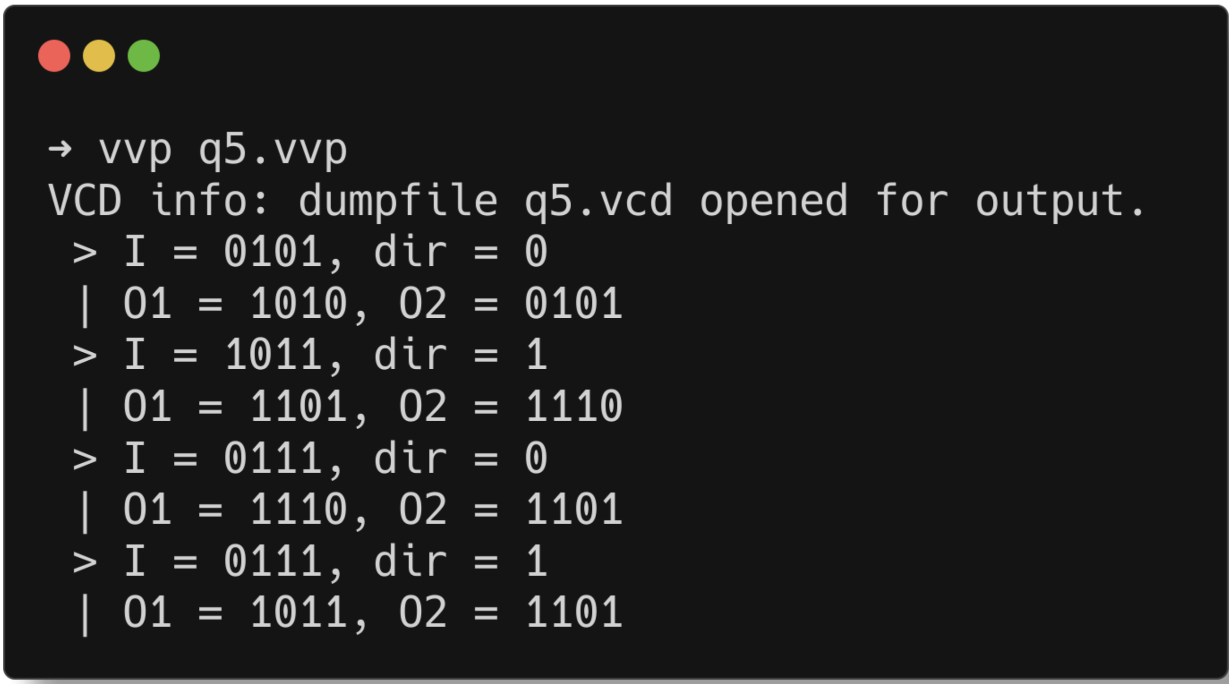
```verilog
`timescale 1ns / 1ps

module TEST();

  reg [3:0] I;
  reg dir;
  wire [3:0] O1, O2;

  SHIFTER1B sft1 (I, dir, O1);
  SHIFTER2B sft2 (I, dir, O2);

  initial begin
    $monitor(
      {
        " > I = %b, dir = %b\n",
        " | O1 = %b, O2 = %b"
      },
      I, dir, O1, O2
    );
```

```verilog
    $dumpfile("q5.vcd");
    $dumpvars(0, TEST);

    I = 4'b0101; dir = 0; #10;
    I = 4'b1011; dir = 1; #10;
    I = 4'b0111; dir = 0; #10;
    I = 4'b0111; dir = 1; #10;

  end

endmodule
```

Shifter testbench code (O1=1-bit rotate, O2=2-bit rotate)

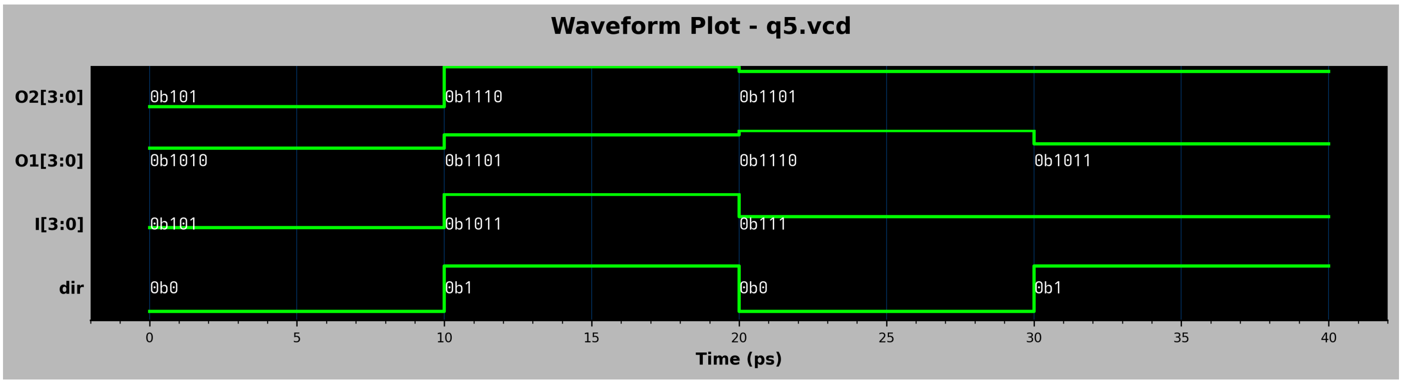
5.3 Simulation Result

Results match expected circular permutations. Separate modules clarify distinct wiring for 1- vs 2-bit paths.



```
➜ vvp q5.vvp
VCD info: dumpfile q5.vcd opened for output.
 > I = 0101, dir = 0
 | O1 = 1010, O2 = 0101
 > I = 1011, dir = 1
 | O1 = 1101, O2 = 1110
 > I = 0111, dir = 0
 | O1 = 1110, O2 = 1101
 > I = 0111, dir = 1
 | O1 = 1011, O2 = 1101
```

Shifter terminal output (includes both distances)



Shifter waveform (shows O1 & O2 evolution)

Problem 6: Simple 1-Bit ALU

ALU composes bitwise primitives and two 8:1 multiplexers to select outputs per opcode. Y0 carries primary result; Y1 provides carry/borrow for arithmetic cases; undefined opcodes yield don't care (x).

6.1 Verilog Code

```
module MUX8x1 (
  input [7:0] I,
  input [2:0] S,
  output Y
);
  wire [7:0] w;
  wire nS2, nS1, nS0;

  not (nS0, S[0]);
  not (nS1, S[1]);
  not (nS2, S[2]);

  and (w[0], I[0], nS2, nS1, nS0);
  and (w[1], I[1], nS2, nS1, S[0]);
  and (w[2], I[2], nS2, S[1], nS0);
  and (w[3], I[3], nS2, S[1], S[0]);
  and (w[4], I[4], S[2], nS1, nS0);
  and (w[5], I[5], S[2], nS1, S[0]);
  and (w[6], I[6], S[2], S[1], nS0);
  and (w[7], I[7], S[2], S[1], S[0]);

  or (Y, w[0], w[1], w[2], w[3], w[4], w[5],
w[6], w[7]);

endmodule
```

MUX8x1

```
module ALU (
  input A, B,
  input [2:0] OP,
  output Y0,
  output Y1
);

  wire and_AB, or_AB, xor_AB, not_A, brrw_AB;

  and (and_AB, A, B);
  or (or_AB, A, B);
  xor (xor_AB, A, B);
  not (not_A, A);

  // borrow = ~A | B
  or (brrw_AB, not_A, B);

  MUX8x1 mux (
    {1'bz, 1'bz, and_AB, xor_AB, xor_AB,
xor_AB, or_AB, and_AB},
    OP,
    Y0
  );

  MUX8x1 mux2 (
    {1'bz, 1'bz, 1'bz, brrw_AB, carry_AB,
1'bz, 1'bz, 1'bz},
    OP,
    Y1
  );

endmodule
```

ALU

6.2 Testbench

```
`timescale 1ns / 1ps

module TEST();

  reg A, B;
  reg [2:0] OP;
  wire Y0, Y1;

  ALU alu (A, B, OP, Y0, Y1);

  initial begin
    $monitor(
      {
        " > A=%b, B=%b, OP=%b\n",
        " | Y0=%b, Y1=%b"
      },
      A, B, OP, Y0, Y1
    );
```

```
    // 2 example for each operation
    A = 0; B = 0; OP = 3'b000; #10;
    A = 1; B = 1; OP = 3'b000; #10;
    A = 0; B = 1; OP = 3'b001; #10;
    A = 1; B = 0; OP = 3'b001; #10;
    A = 0; B = 1; OP = 3'b010; #10;
    A = 1; B = 1; OP = 3'b010; #10;
    A = 0; B = 0; OP = 3'b011; #10;
    A = 1; B = 1; OP = 3'b011; #10;
    A = 0; B = 1; OP = 3'b100; #10;
    A = 1; B = 0; OP = 3'b100; #10;
    A = 0; B = 0; OP = 3'b101; #10;
    A = 1; B = 1; OP = 3'b101; #10;

    // No operation defined
    A = 0; B = 1; OP = 3'b110; #10;
    A = 1; B = 0; OP = 3'b111; #10;
```

```
    $dumpfile("q6.vcd");
    $dumpvars(0, TEST);
  end

endmodule
```

ALU testbench code

6.3 Simulation Result

All defined opcodes produce correct logical/arithmetic outputs (AND, OR, XOR, ADD, SUB, MULT alias) with carry/borrow (or borrow) reflected on Y1; undefined opcodes show don't-care (x) outputs as routed via unused MUX inputs.

Why no waveform figure: The waveform would be cluttered and hard to read due to the many signals and rapid changes and don't cares (x). The terminal output fully characterizes the ALU's single-bit combinational behavior across some input combinations and opcodes, making it a more effective summary of functionality.

```
→ vvp q6.vvp
VCD info: dumpfile q6.vcd opened for output.
 > A=0, B=0, OP=000
 | Y0=0, Y1=x
 > A=1, B=1, OP=000
 | Y0=1, Y1=x
 > A=0, B=1, OP=001
 | Y0=1, Y1=x
 > A=1, B=0, OP=001
 | Y0=1, Y1=x
 > A=0, B=1, OP=010
 | Y0=1, Y1=x
 > A=1, B=1, OP=010
 | Y0=0, Y1=x
 > A=0, B=0, OP=011
 | Y0=0, Y1=x
 > A=1, B=1, OP=011
 | Y0=0, Y1=x
 > A=0, B=1, OP=100
 | Y0=1, Y1=1
 > A=1, B=0, OP=100
 | Y0=1, Y1=0
 > A=0, B=0, OP=101
 | Y0=0, Y1=x
 > A=1, B=1, OP=101
 | Y0=1, Y1=x
 > A=0, B=1, OP=110
 | Y0=x, Y1=x
 > A=1, B=0, OP=111
 | Y0=x, Y1=x
```

ALU terminal output