EE1501 Assignment 2

Arjun Pavanje

April 2025

1 Convolution

- **Inputs:** Two 8-bit vectors
  - `[3:0] x [7:0]`
  - `[3:0] h [7:0]`.
- **Outputs:** Vector `[3:0] y [15:0]`, which contains the result of $x * h$

Approach:

Convolution of two vectors is basically polynomial multiplication where coeffecients of the polynomials are elements of the vector. Output vector will be given by,

$$y[n] = \sum_{k=0}^{7} x[k] \cdot h[n-k]$$

n varies from 0 to 15 (vectors are indexed from 0). All elements of y are also 4−bit numbers, overflow is ignored.

Code

```
module convolvolution (
  input  [3:0] x [7:0],
  input  [3:0] h [7:0],
  output reg [3:0] y [14:0]
);

integer i, k, j;
always @(*) begin
  for (i = 0; i < 15; i = i + 1) begin
    y[i] = 4'd0; // initalizing each element to 0
  end
  for (j=0; j < 8; j = j + 1) begin
    for (k = 0; k < 8; k = k + 1) begin
      y[j+k] += x[j]*h[k];
    end
  end
```

```
end
endmodule
```

Testbench

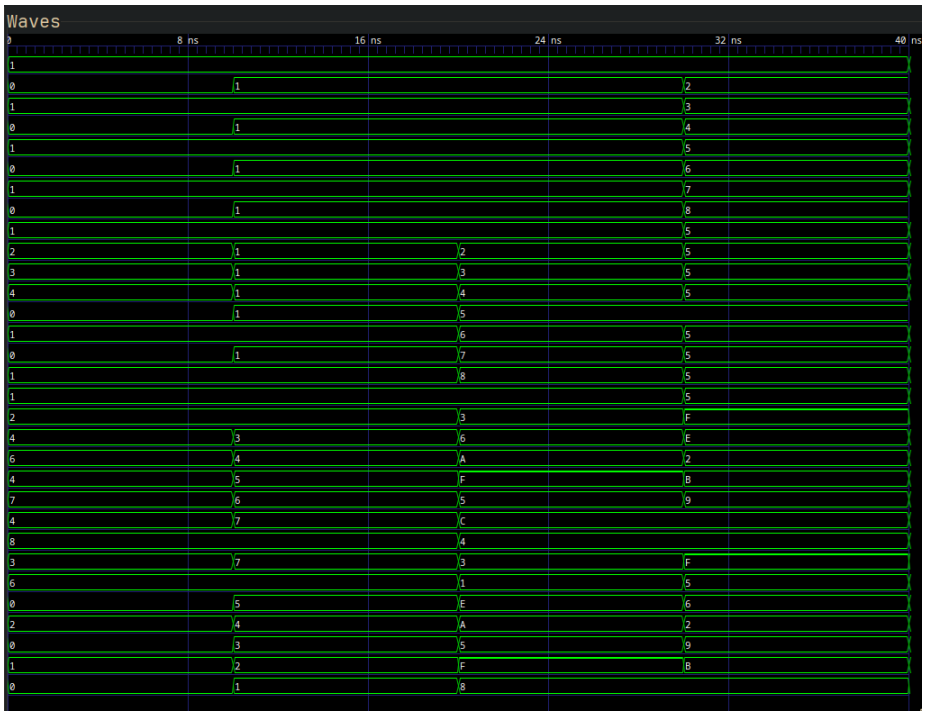
```
module tb_convolvolution;
reg [3:0] x[7:0];
reg [3:0] h[7:0];
wire [3:0] y[14:0];

convolvolution dut(.x(x), .h(h), .y(y));

initial begin
  $dumpfile("convolvolution.vcd");
  $dumpvars(0, tb_convolvolution);

  for (integer i = 0; i < 8; i++) begin
    $dumpvars(0, tb_convolvolution.x[i]);
    $dumpvars(0, tb_convolvolution.h[i]);
  end

  for (integer i = 0; i < 15; i++) begin
    $dumpvars(0, tb_convolvolution.y[i]);
  end

  $monitor("x = [%d, %d, %d, %d, %d, %d, %d, %d]\nh = [%d, %d, %d,
    %d, %d, %d, %d, %d]\nx*h = [%d, %d, %d, %d, %d, %d, %d, %d, %d,
     %d, %d, %d, %d, %d, %d]\n", x[0], x[1], x[2], x[3], x[4], x
    [5], x[6], x[7], h[0], h[1], h[2], h[3], h[4], h[5], h[6], h
    [7], y[0], y[1], y[2], y[3], y[4], y[5], y[6], y[7], y[8], y
    [9], y[10], y[11], y[12], y[13], y[14]);

  // Test Case 1
  x[0] = 4'd1; x[1] = 4'd2; x[2] = 4'd3; x[3] = 4'd4;
  x[4] = 4'd0; x[5] = 4'd1; x[6] = 4'd0; x[7] = 4'd1;
  h[0] = 4'd1; h[1] = 4'd0; h[2] = 4'd1; h[3] = 4'd0;
  h[4] = 4'd1; h[5] = 4'd0; h[6] = 4'd1; h[7] = 4'd0;
  #10

  // Test Case 2
  x[0] = 4'd1; x[1] = 4'd1; x[2] = 4'd1; x[3] = 4'd1;
  x[4] = 4'd1; x[5] = 4'd1; x[6] = 4'd1; x[7] = 4'd1;
  h[0] = 4'd1; h[1] = 4'd1; h[2] = 4'd1; h[3] = 4'd1;
  h[4] = 4'd1; h[5] = 4'd1; h[6] = 4'd1; h[7] = 4'd1;
  #10

  // Test Case 3
  x[0] = 4'd1; x[1] = 4'd2; x[2] = 4'd3; x[3] = 4'd4;
  x[4] = 4'd5; x[5] = 4'd6; x[6] = 4'd7; x[7] = 4'd8;
  h[0] = 4'd1; h[1] = 4'd1; h[2] = 4'd1; h[3] = 4'd1;
  h[4] = 4'd1; h[5] = 4'd1; h[6] = 4'd1; h[7] = 4'd1;
  #10

  // Test Case 4
  x[0] = 4'd5; x[1] = 4'd5; x[2] = 4'd5; x[3] = 4'd5;
  x[4] = 4'd5; x[5] = 4'd5; x[6] = 4'd5; x[7] = 4'd5;
```

```
  h[0] = 4'd1; h[1] = 4'd2; h[2] = 4'd3; h[3] = 4'd4;
  h[4] = 4'd5; h[5] = 4'd6; h[6] = 4'd7; h[7] = 4'd8;
  #10

  // Test Case 5
  x[0] = 4'd7; x[1] = 4'd3; x[2] = 4'd9; x[3] = 4'd2;
  x[4] = 4'd5; x[5] = 4'd1; x[6] = 4'd8; x[7] = 4'd4;
  h[0] = 4'd6; h[1] = 4'd2; h[2] = 4'd7; h[3] = 4'd1;
  h[4] = 4'd9; h[5] = 4'd3; h[6] = 4'd5; h[7] = 4'd8;
end
endmodule
```

Output of test bench,

```
x = [ 1,  2,  3,  4,  0,  1,  0,  1]
h = [ 1,  0,  1,  0,  1,  0,  1,  0]
x*h = [ 1,  2,  4,  6,  4,  7,  4,  8,  3,  6,  0,  2,  0,  1,  0]

x = [ 1,  1,  1,  1,  1,  1,  1,  1]
h = [ 1,  1,  1,  1,  1,  1,  1,  1]
x*h = [ 1,  2,  3,  4,  5,  6,  7,  8,  7,  6,  5,  4,  3,  2,  1]

x = [ 1,  2,  3,  4,  5,  6,  7,  8]
h = [ 1,  1,  1,  1,  1,  1,  1,  1]
x*h = [ 1,  3,  6, 10, 15,  5, 12,  4,  3,  1, 14, 10,  5, 15,  8]

x = [ 5,  5,  5,  5,  5,  5,  5,  5]
h = [ 1,  2,  3,  4,  5,  6,  7,  8]
x*h = [ 5, 15, 14,  2, 11,  9, 12,  4, 15,  5,  6,  2,  9, 11,  8]

x = [ 7,  3,  9,  2,  5,  1,  8,  4]
h = [ 6,  2,  7,  1,  9,  3,  5,  8]
x*h = [10,  0, 13, 10,  3,  7,  4,  8,  9, 14,  8,  9, 12,  4,  0]
```

Waveforms



2 8-bit Full Adder

- **Inputs:**
  - `[7:0] a` – First 8-bit number
  - `[7:0] b` – Second 8-bit number
  - `carry_in` – Input carry (1-bit)
- **Outputs:**
  - `[7:0] Sum` – 8-bit sum
  - `carry_out` – Output carry (1-bit)

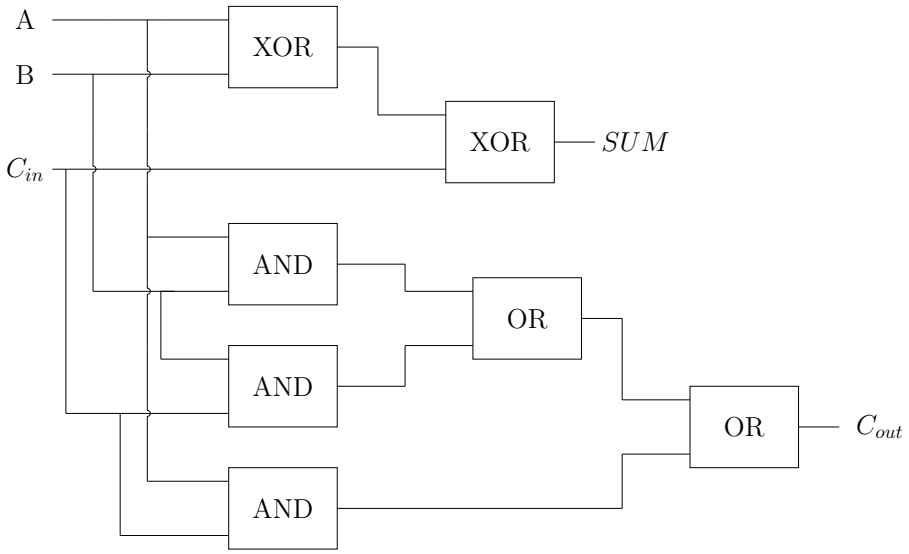
Approach:

An 8-bit full adder is constructed using eight 1-bit full adders in a ripple-carry configuration. FUll adder is implemented at each bit. A for-loop is used to

implement this ripple-carry behavior behaviorally inside an `always` block. For each bit, the sum and carry are calculated as,

$$\texttt{sum[i]} = A[i] \oplus B[i] \oplus \texttt{carry\_in}$$
$$\texttt{carry\_out} = (A[i] \wedge B[i]) \vee (A[i] \wedge \texttt{carry\_in}) \vee (B[i] \wedge \texttt{carry\_in})$$

The final carry becomes `carry_out`.



code

```
module bitadder(
  input [7:0] a,
  input [7:0] b,
  input carry_in,
  output reg [7:0] sum,
  output reg carry_out
);
reg carry;
integer i;

always @(*) begin
  carry = carry_in;
  for (i = 0; i < 8; i = i + 1) begin
    sum[i] = a[i] ^ b[i] ^ carry;
    carry = (a[i] & b[i]) | (a[i] & carry) | (b[i] & carry);
  end
  carry_out = carry;
end
endmodule
```

Testbench

```
module tb_adder;
reg[7:0] a, b;
reg carry_in;
wire [7:0] sum;
wire carry_out;

bitadder uut(
  .a(a),
  .b(b),
  .carry_in(carry_in),
  .sum(sum),
  .carry_out(carry_out)
);
initial begin
  $dumpfile("8bitadder.vcd");
  $dumpvars(0, tb_adder);

  $display("Test Cases");
  // Test Case 1
  a = 8'b01101101; b = 8'b10110010; carry_in = 0;
  $monitor("%0t. %b  %b (%b)  %b (%b)",  $time, a, b, carry_in, sum
    , carry_out);
  #10

  // Test Case 2
  a = 8'b11010110; b = 8'b01010101; carry_in = 0;
  #10

  // Test Case 3
  a = 8'b11111111; b = 8'b00000001; carry_in = 0;
  #10

  // Test Case 4
  a = 8'b10011001; b = 8'b01101100; carry_in = 1;
  #10

  // Test Case 5
  a = 8'b01010101; b = 8'b11101010; carry_in = 1;
  #10

end
endmodule
```

Output of test bench,

```
Test Cases
0. 01101101  10110010 (0)  00011111 (1)
10. 11010110  01010101 (0)  00101011 (1)
20. 11111111  00000001 (0)  00000000 (1)
30. 10011001  01101100 (1)  00000110 (1)
40. 01010101  11101010 (1)  01000000 (1)
```

Waveforms

3 4-bit Adder using NAND Gates

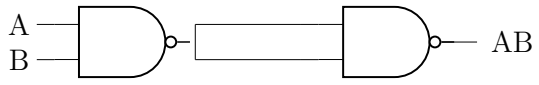
- **Inputs:**
  - `[3:0] A` – First 4-bit operand
  - `[3:0] B` – Second 4-bit operand
  - `carry_in` – Initial carry-in (1-bit)
- **Outputs:**
  - `[3:0] Sum` – 4-bit sum of inputs
  - `carry_out` – Final carry-out (1-bit)

Approach:

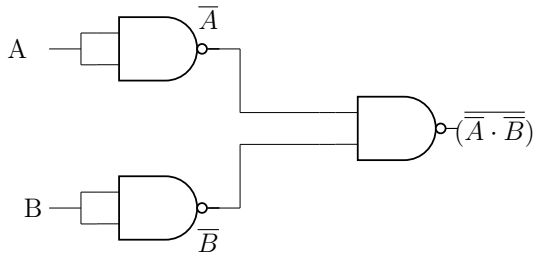
A 4-bit ripple carry adder is constructed by chaining four 1-bit full adders. Each full adder is implemented using only 2-input NAND gates. The logic gates for XOR, AND, and OR are derived using NAND-only constructs. The same expressions for bitwise sum and carry are used as in the previous question, except instead of using XOR and OR gates directly, we have derived them using only fundamental NAND gates.

Logical Gates using NAND

AND gate using NAND



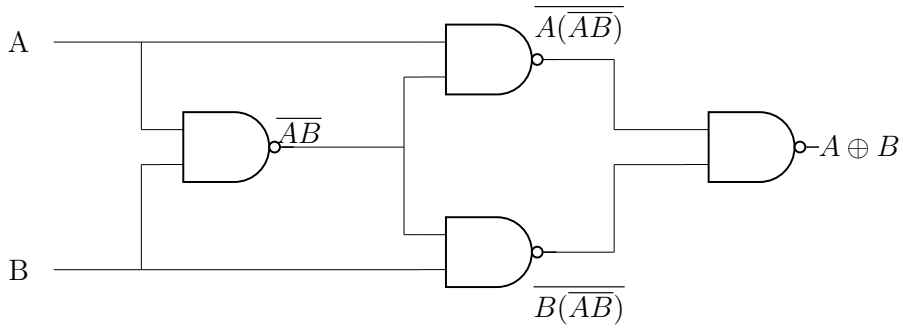
Output of first NAND Gate is $\overline{A \cdot B}$. And passing it through both terminals of a two input NAND gate (which basically acts like a NOT gate) returns $A \cdot B$



OR gate using NAND

Inputs A, B are duplicated and each (seperately) passed to a NAND gate, where they are inverted. The resultants are passed to another NAND gate where the final result is, $(\overline{\overline{A} \cdot \overline{B}})$ which simplifies (using De-Morgans Laws) to be, $A + B$

XOR using NAND



Worst Case Delay

There are 6 levels of combinational logics that occur sequentially for calculation of each bit of "sum", there will be the same number of levels in case of "carry_out". The maximum delay will occur regardless of a, b as long as one of them are $HIGH$, and c_{in} toggles. This causes a propagation delay in each and every level of the circuit, and since we have taken delay as $1ns$ at each gate, total delay will be equal to number of levels, as delay propagates parallelly through all the gates in each level. So here, total delay will be $6ns$. This is for a singular bit, since we are dealing with 4-bit numbers, total delay comes out to be $24ns$ which is what we observe through the timing diagram. We have taken the consecutive test cases, $a = 0, b = 0, c_{in} = 0$ and $a = 15, b = 0, c_{in} = 1$. here, c_{in} toggles for every bit from the former to the later test case, so observe the waveform for maximum delay.
In case of "sum", there are 6 levels again, between the two test cases since the

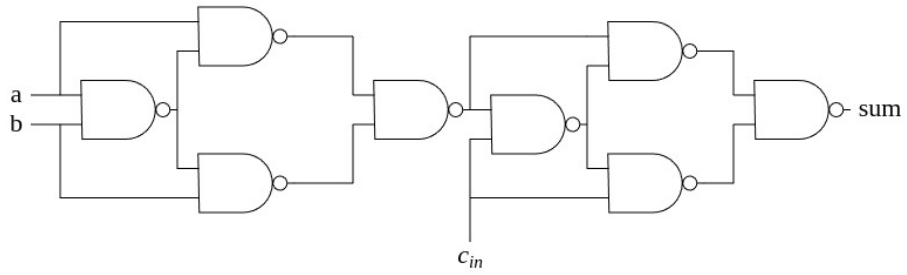


Figure 1: Sum

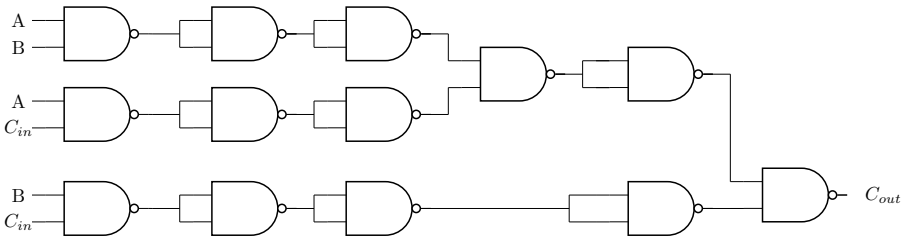


Figure 2: Carry

delay for each bit is $6ns$ which is also the maximum possible delay which can occur for "sum" bit, the sum calculated within $6ns$ itself. Since a 4-bit sum requires only 3 carrys to be calculated, before it can be computed, it requires $18ns$ to compute. As for the last bit, it only requires $3ns$ as $a \oplus b$ would have already been calculated. So a total delay of $21ns$ is observed for the sum.

Code

```
‘timescale 1ns/1ns

module and_nand(
  input a,
  input b,
  output out
);
wire temp;
nand #1(temp, a, b);
nand #1(out, temp, temp);
endmodule

module or_nand(
  input a,
  input b,
  output out
);
wire temp1, temp2;
```

```
nand #1(temp1, a, a);
nand #1(temp2, b, b);
nand #1(out, temp1, temp2);
endmodule

module xor_nand(
  input a,
  input b,
  output out
);
wire temp1, temp2, temp3;
nand #1(temp1, a, b);
nand #1(temp2, a, temp1);
nand #1(temp3, b, temp1);
nand #1(out, temp2, temp3);
endmodule

module bitadder(
  input [3:0] a,
  input [3:0] b,
  input carry_in,
  output [3:0] sum,
  output carry_out
);
wire [4:0] carry;
assign carry[0] = carry_in;


genvar i;
generate
  for (i = 0; i < 4; i = i + 1) begin
    wire temp1, temp2, temp3, temp4, temp5;

    xor_nand xor1 (.a(a[i]), .b(b[i]), .out(temp1));
    xor_nand xor2 (.a(temp1), .b(carry[i]), .out(sum[i]));

    and_nand and1 (.a(a[i]), .b(b[i]), .out(temp2));
    and_nand and2 (.a(a[i]), .b(carry[i]), .out(temp3));
    and_nand and3 (.a(b[i]), .b(carry[i]), .out(temp4));

    or_nand or1 (.a(temp2), .b(temp3), .out(temp5));
    or_nand or2 (.a(temp5), .b(temp4), .out(carry[i+1]));
  end
endgenerate

assign carry_out = carry[4];

endmodule
```

Testbench

```
module tb_adder;
reg [3:0] a, b;
reg carry_in;
wire [3:0] sum;
wire carry_out;
```

```
bitadder uut (
  .a(a),
  .b(b),
  .carry_in(carry_in),
  .sum(sum),
  .carry_out(carry_out)
);
initial begin
  $dumpfile("4bitadder.vcd");
  $dumpvars(0, tb_adder);

  // Test Case 1
  a = 4'b0000; b = 4'b0000; carry_in = 0;
  #50


  // Test Case 2
  $display("%0t %b  %b (%b)  %b (%b)", $time, a, b, carry_in, sum,
    carry_out);
  $display("done");

  a = 4'b1111; b = 4'b0000; carry_in = 1;
  #50

  // Test Case 3
  $display("%0t %b  %b (%b)  %b (%b)", $time, a, b, carry_in, sum,
    carry_out);
  $display("done");

  a = 4'b1111; b = 4'b1111; carry_in = 0;
  #50

  // Test Case 4
  $display("%0t %b  %b (%b)  %b (%b)", $time, a, b, carry_in, sum,
    carry_out);
  $display("done");

  a = 4'b0000; b = 4'b0000; carry_in = 1;
  #50

  // Test Case 5
  $display("%0t %b  %b (%b)  %b (%b)", $time, a, b, carry_in, sum,
    carry_out);
  $display("done");

  a = 4'b1010; b = 4'b0101; carry_in = 0;
  #50
  $display("%0t %b  %b (%b)  %b (%b)", $time, a, b, carry_in, sum,
    carry_out);

end
endmodule
```

Output of test bench,

```
    50 0000  0000 (0)  0000 (0)
done
```

```
100  1111   0000 (1)   0000 (1)
done
150  1111   1111 (0)   1110 (1)
done
200  0000   0000 (1)   0001 (0)
done
250  1010   0101 (0)   1111 (0)
```

Waveforms