DIGITAL SYSTEM DESIGN LABORATORY

REPORT LAB 3

IMPLEMENTATION OF ADDERS, SUBTRACTORS, AND MULTIPLIERS USING VERILOG HDL

NAME: Bùi Gia Bảo

ID: ITITIU22019

I. PURPOSE OF THE EXPERIMENT

The purpose of this laboratory exercise is to examine **arithmetic circuits** that add, subtract, and multiply numbers. Each type of circuit shall be implemented in **two ways**: first by writing **Verilog code** that describes the required functionality, and second by making use of **predefined subcircuits** from Altera's library of parameterized modules. The results produced for various implementations will be compared, both in terms of the **circuit structure** and its **speed of operation**. All circuits must be implemented using **pure Verilog structural modeling** with **primitive gates**, **full adders**, and **hierarchical instantiation**.

II. THEORETICAL BACKGROUND

II.1 Ripple-Carry Adder (RCA)

A **ripple-carry adder** is a combinational circuit that performs binary addition by cascading **full adders (FAs)**. For an 8-bit adder:

$$S_i = A_i \oplus B_i \oplus C_i, C_{i+1} = (A_i \cdot B_i) + (A_i \cdot C_i) + (B_i \cdot C_i)$$

- **Carry propagation delay**: The carry signal "ripples" from the least significant bit (LSB) to the most significant bit (MSB).
- **Critical path**: $C_0 \rightarrow C_8$through 8 full adders.
- **Maximum delay**: $T_{\text{total}} = T_{\text{XOR}} + 8 \times T_{\text{FA}}$
- **Advantage**: Simple, regular structure, easy to model structurally.
- **Disadvantage**: Poor scalability due to $O(n)$delay.

II.2 Two's Complement Representation and Overflow

In **8-bit two's complement**:

- Range: $[-128, +127]$
- Sign bit: A_7(0 = positive, 1 = negative)

Overflow occurs when the result of an addition exceeds the representable range:

- Two positive numbers → negative result
- Two negative numbers → positive result

Overflow detection logic: Overflow $= (A_7 \cdot B_7 \cdot \neg S_7) + (\neg A_7 \cdot \neg B_7 \cdot S_7)$

This is implemented using **AND**, **NOT**, and **OR** gates.

II.3 Subtraction Using Two's Complement

Subtraction $A - B$is implemented as: $A - B = A + (\neg B + 1)$

- **Conditional inversion**: $B \oplus \{\text{sub}, \ldots, \text{sub}\}$
- **Carry-in** = sub signal
- **Unified add/sub circuit** controlled by a single bit.

II.4 Binary Multiplication and Array Multiplier

Binary multiplication follows the **shift-and-add** algorithm:

$$P = A \times B = \sum_{i=0}^{7} \quad (A \times b_i) \ll i$$

- **Partial products (PPs)**: Generated using **AND gates**: $pp_i[j] = a_j \cdot b_i$
- **Summation**: Performed using **full adders** in a **2D array**
- **Array multiplier structure**:
  - Regular, scalable, synthesizable
  - Critical path grows diagonally → $O(n)$delay per dimension

II.5 Registered Datapath and Pipelining

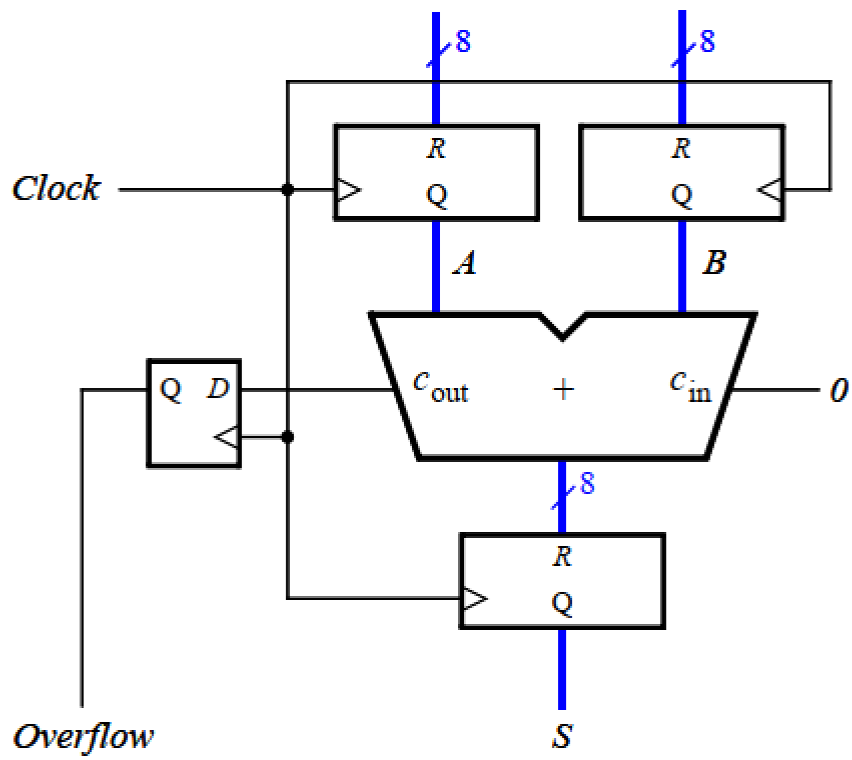
- **Input registers**: Sample inputs on clock edge → eliminate setup/hold violations
- **Output registers**: Stabilize outputs → improve observability
- **Write Enable (WE)**: Controls register loading
- **Pipelining**: Inserting registers between combinational stages reduces **critical path delay**, increasing **fmax**

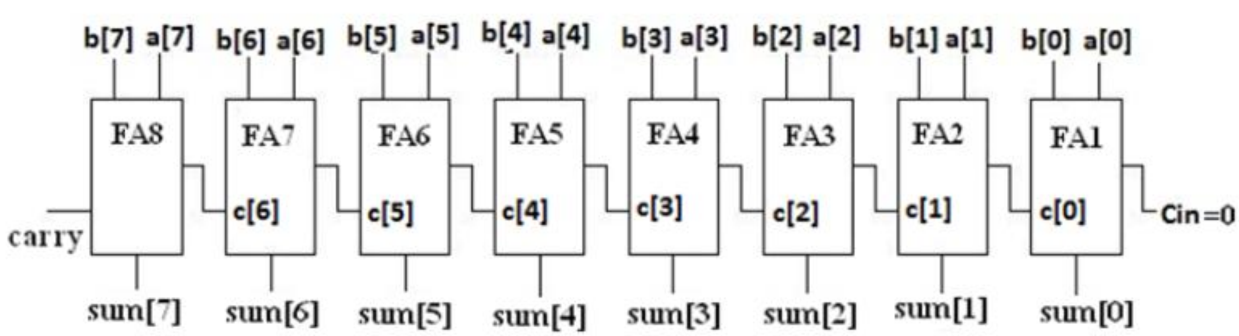
III. EXPERIMENTAL PROCEDURE

PART I: 8-BIT REGISTERED RIPPLE-CARRY ADDER WITH OVERFLOW

Objective

Design an 8-bit ripple-carry adder with **registered inputs and outputs**, and **overflow detection** (carry out) using **structural Verilog**.





Step-by-Step Design

1. Gate-Level Full Adder

```
module full_adder (
    input  wire a, b, cin,
    output wire sum, cout
);
    wire p, g, c1;

    // Propagate and Generate
    xor (p, a, b);
    and (g, a, b);

    // Sum
    xor (sum, p, cin);

    // Carry-out
    and (c1, p, cin);
    or  (cout, g, c1);
endmodule
```

Explanation:

- p = carry propagate
- g = carry generate
- Structural instantiation of primitive gates ensures **no LPM usage**

---

2. 8-Bit Ripple-Carry Adder (Structural)

```
module eight_bit_adder_stru (
    input  [7:0] a, b,
    input        cin,
    output [7:0] sum,
    output       cout
);
    wire [7:1] carry;
```

```
    full_adder fa0 (a[0], b[0], cin,      sum[0], carry[1]);
    full_adder fa1 (a[1], b[1], carry[1], sum[1], carry[2]);
    full_adder fa2 (a[2], b[2], carry[2], sum[2], carry[3]);
    full_adder fa3 (a[3], b[3], carry[3], sum[3], carry[4]);
    full_adder fa4 (a[4], b[4], carry[4], sum[4], carry[5]);
    full_adder fa5 (a[5], b[5], carry[5], sum[5], carry[6]);
    full_adder fa6 (a[6], b[6], carry[6], sum[6], carry[7]);
    full_adder fa7 (a[7], b[7], carry[7], sum[7], cout);
endmodule
```

3. Top-Level Module (DE2-115)

```
module lab6_part1 (
    input  [17:0] SW,
    input  [1:0]  KEY,
    output [7:0]  LEDR,
    output [8:0]  LEDG,
    output [6:0]  HEX7, HEX6, HEX5, HEX4, HEX1, HEX0
);
    wire clk = KEY[1], reset_n = KEY[0];
    wire [7:0] A_reg, B_reg, S_reg;
    wire cout, overflow;

    // Input Registers
    dff8 regA (.d(SW[15:8]), .clk(clk), .clrn(reset_n), .q(A_reg));
    dff8 regB (.d(SW[7:0]),  .clk(clk), .clrn(reset_n), .q(B_reg));

    // 8-bit Adder
    eight_bit_adder_stru adder (.a(A_reg), .b(B_reg), .cin(1'b0), .sum(S_reg), .cout(cout));

    // Output Register
    dff8 regS (.d(S_reg), .clk(clk), .clrn(reset_n), .q(LEDR[7:0]));

    // Overflow Detection
```

```
    assign overflow = (A_reg[7] & B_reg[7] & ~S_reg[7]) | (~A_reg[7] & ~B_reg[7] & S_reg[7]);
    assign LEDG[8] = overflow;

    // 7-Segment Displays
    hex7seg h7(SW[15:12], HEX7); hex7seg h6(SW[11:8], HEX6);
    hex7seg h5(SW[7:4],   HEX5); hex7seg h4(SW[3:0],  HEX4);
    hex7seg h1(LEDR[7:4], HEX1); hex7seg h0(LEDR[3:0], HEX0);
endmodule
```

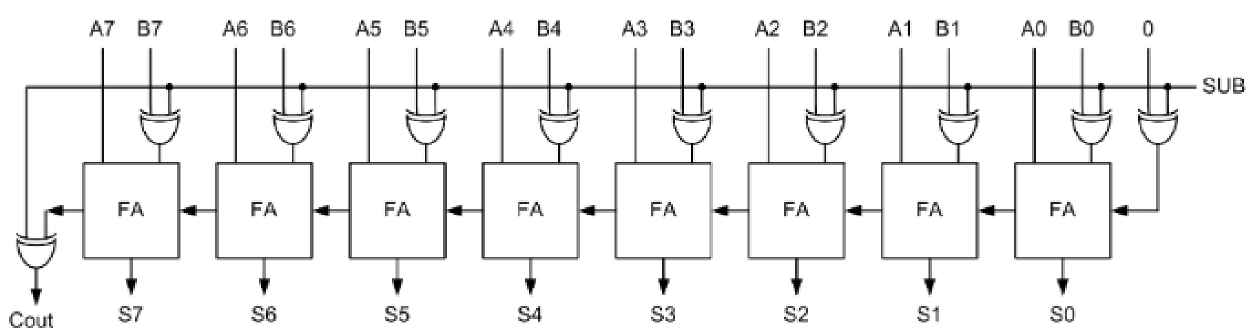
Tasks

1. **Create Quartus Prime project** and write the above Verilog code.
2. **Assign pins**:
   - SW[15:8] → A, SW[7:0] → B
   - KEY[1] → Clock, KEY[0] → Reset
   - LEDR[7:0] → Sum, LEDG[8] → Overflow
3. **Compile and simulate** using ModelSim. Verify overflow cases.
4. **Download to DE2-115** and test with switches.
5. **Open Timing Analyzer** → Report:
   - **fmax**
   - **Longest path delay**
   - **Logic Elements (LEs)**

PART II: 8-BIT REGISTERED ADDER/SUBTRACTOR

Objective

Modify Part I to support **addition and subtraction** using a **control signal SUB** .



Theory Recap

- $A - B = A + (\neg B + 1)$
- Use **XOR** to conditionally invert B
- Carry-in = sub

Structural Add/Sub Module

```
module eight_bit_addsub_stru (
    input  [7:0] a, b,
    input        sub,
    output [7:0] sum,
    output       cout, overflow
);
    wire [7:0] b_comp;
    wire [7:1] carry;

    assign b_comp = b ^ {8{sub}};  // Invert B if sub=1

    full_adder fa0 (a[0], b_comp[0], sub, sum[0], carry[1]);
    full_adder fa1 (a[1], b_comp[1], carry[1], sum[1], carry[2]);
    full_adder fa2 (a[2], b_comp[2], carry[2], sum[2], carry[3]);
    full_adder fa3 (a[3], b_comp[3], carry[3], sum[3], carry[4]);
    full_adder fa4 (a[4], b_comp[4], carry[4], sum[4], carry[5]);
    full_adder fa5 (a[5], b_comp[5], carry[5], sum[5], carry[6]);
    full_adder fa6 (a[6], b_comp[6], carry[6], sum[6], carry[7]);
    full_adder fa7 (a[7], b_comp[7], carry[7], sum[7], cout);

    assign overflow = (a[7] & b_comp[7] & ~sum[7]) | (~a[7] & ~b_comp[7] & sum[7]);
endmodule
```

Top-Level Module

```
module lab6_part2 (
    input  [17:0] SW,
    input  [1:0]  KEY,
    output [7:0]  LEDR,
```

```
    output [8:0]  LEDG,
    output [6:0]  HEX7, HEX6, HEX5, HEX4, HEX1, HEX0
);
    wire clk = KEY[1], reset_n = KEY[0], sub = SW[16];
    wire [7:0] A_reg, B_reg, S_reg;
    wire cout, overflow;

    dff8 regA (.d(SW[15:8]), .clk(clk), .clrn(reset_n), .q(A_reg));
    dff8 regB (.d(SW[7:0]),  .clk(clk), .clrn(reset_n), .q(B_reg));

    eight_bit_addsub_stru addsub (.a(A_reg), .b(B_reg), .sub(sub), .sum(S_reg), .cout(cout), .overflow(overflow));

    dff8 regS (.d(S_reg), .clk(clk), .clrn(reset_n), .q(LEDR[7:0]));
    assign LEDG[8] = overflow;

    // 7-segment displays
    hex7seg h7(SW[15:12], HEX7); hex7seg h6(SW[11:8], HEX6);
    hex7seg h5(SW[7:4],   HEX5); hex7seg h4(SW[3:0],  HEX4);
    hex7seg h1(LEDR[7:4], HEX1); hex7seg h0(LEDR[3:0], HEX0);
endmodule
```

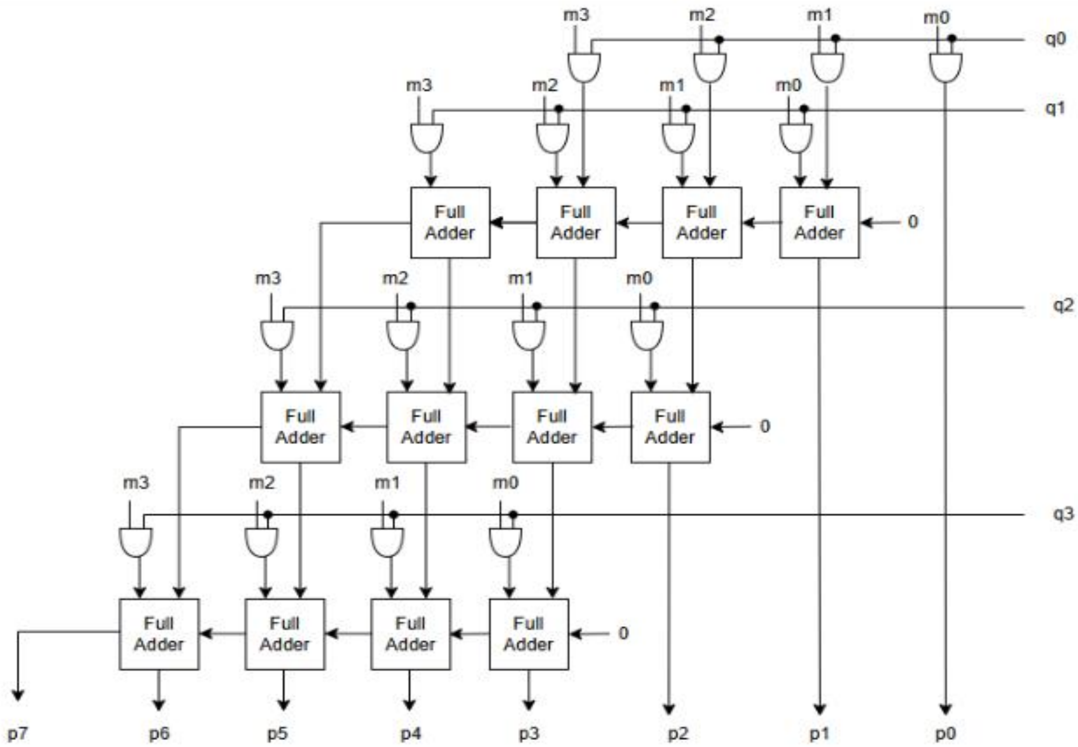
Tasks

1. Simulate **add** and **subtract** modes.
2. Download to FPGA.
3. Compare **fmax** and **longest path** with Part I.

PART III: 4-BIT ARRAY MULTIPLIER

Objective

Implement a **4×4 array multiplier** using **AND gates** and **full adders**.




```
module array_multiplier_4x4 (
    input  [3:0] a, b,
    output [7:0] p
);
    wire [3:0] pp0, pp1, pp2, pp3;
    wire [6:0] c;

    assign pp0 = a & {4{b[0]}};
    assign pp1 = a & {4{b[1]}};
    assign pp2 = a & {4{b[2]}};
    assign pp3 = a & {4{b[3]}};

    assign p[0] = pp0[0];
    full_adder fa00 (pp0[1], pp1[0], 1'b0, p[1], c[0]);

    full_adder fa10 (pp0[2], pp1[1], c[0], p[2], c[1]);
    full_adder fa11 (pp0[3], pp1[2], c[1], p[3], c[2]);
    full_adder fa12 (pp1[3], 1'b0,   c[2], p[4], c[3]);

    full_adder fa20 (pp2[1], c[3], 1'b0, p[5], c[4]);
```

```
    full_adder fa21 (pp2[2], c[4], 1'b0, p[6], c[5]);
    full_adder fa22 (pp2[3], c[5], 1'b0, p[7], c[6]);

    assign p[7] = pp3[3] ^ c[6];
endmodule
```

Top-Level 4-Bit Multiplier

```
module lab6_part3 (
    input  [11:0] SW,
    output [6:0]  HEX6, HEX4, HEX1, HEX0
);
    wire [7:0] p;

    array_multiplier_4x4 mult (.a(SW[11:8]), .b(SW[3:0]), .p(p));

    hex7seg h6(SW[11:8], HEX6); hex7seg h4(SW[3:0], HEX4);
    hex7seg h1(p[7:4], HEX1);   hex7seg h0(p[3:0], HEX0);
endmodule
```

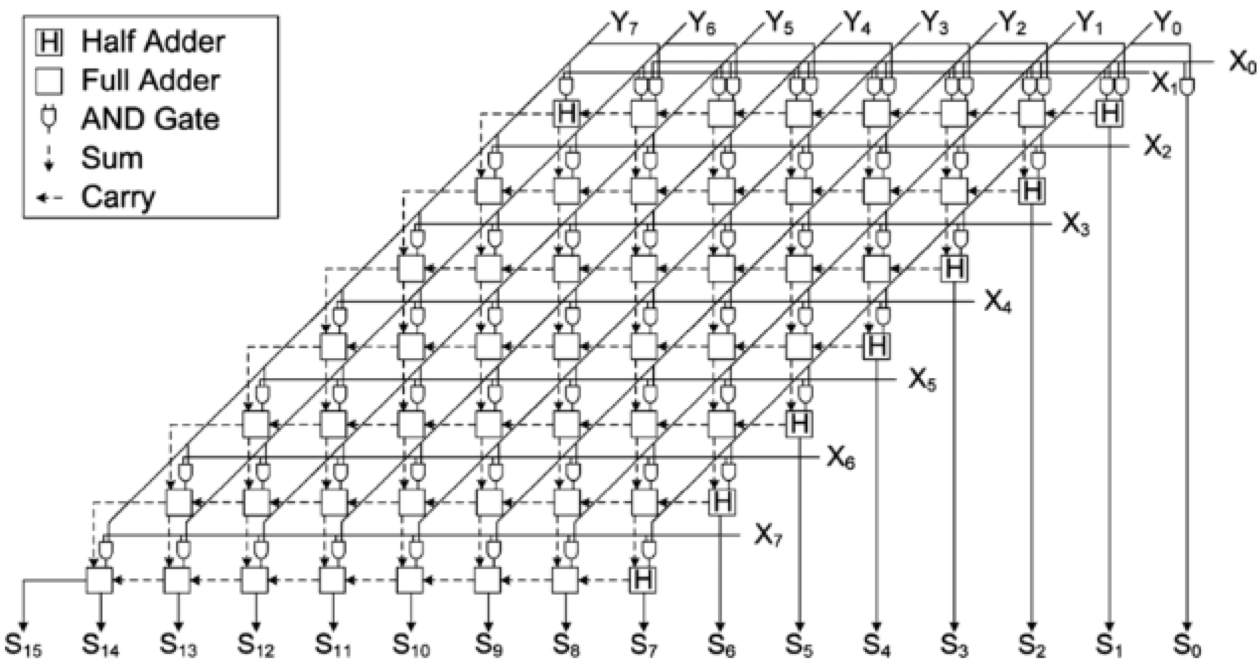
Tasks

1. Simulate and verify with testbench.
2. Display A, B, Pon 7-segment displays.
3. Test on DE2-115.

PART IV: 8-BIT REGISTERED ARRAY MULTIPLIER

Objective

Extend the 4-bit multiplier to **8×8 → 16-bit** with **input/output registers**.



8×8 Array Multiplier (Hierarchical)

```
module array_multiplier_8x8 (
   input  [7:0] a, b,
   input        clk, clrn,
   output [15:0] p
);
   wire [7:0] A_reg, B_reg;
   wire [15:0] p_comb;

   dff8 regA (.d(a), .clk(clk), .clrn(clrn), .q(A_reg));
   dff8 regB (.d(b), .clk(clk), .clrn(clrn), .q(B_reg));

   // Full 8x8 array (use generate or hierarchical 4x4 blocks)
   // ... (complex — implement using generate loops)
```

```
    dff16 regP (.d(p_comb), .clk(clk), .clrn(clrn), .q(p));
endmodule
```

PART V: S = (A×B) + (C×D) WITH WRITE ENABLE

Objective

Implement a **registered MAC unit** with **write enable** and **input selection**.

Top-Level MAC Unit

```
module lab6_part5 (
    input  [17:0] SW,
    input  [1:0]  KEY,
    output [8:0]  LEDG,
    output [6:0]  HEX7, HEX6, HEX5, HEX4, HEX3, HEX2, HEX1, HEX0
);
    wire clk = KEY[1], clrn = KEY[0], we = SW[17], sel = SW[16];
    wire [7:0] A, B, C, D, inA, inB;
    wire [15:0] P1, P2, Sum;
    wire cout;

    assign inA = sel ? SW[15:8] : SW[15:8];
    assign inB = sel ? SW[7:0]  : SW[7:0];

    dff8_we regA (.d(inA), .clk(clk), .we(we), .clrn(clrn), .q(A));
    dff8_we regB (.d(inB), .clk(clk), .we(we), .clrn(clrn), .q(B));
    // Repeat for C, D

    array_multiplier_8x8 mult1 (.a(A), .b(B), .clk(clk), .clrn(clrn), .p(P1));
    array_multiplier_8x8 mult2 (.a(C), .b(D), .clk(clk), .clrn(clrn), .p(P2));

    eight_bit_addsub_stru adder16 (.a(P1), .b(P2), .sub(1'b0), .sum(Sum), .cout(cout));
```

```
    dff16 regSum (.d(Sum), .clk(clk), .clrn(clrn), .q({HEX3,HEX2,HEX1,HEX0}));
    assign LEDG[8] = cout;
endmodule
```

Your code here

……….

……….

IV. SUPPORT MODULES

```
module dff8(input [7:0] d, input clk, clrn, output reg [7:0] q);
    always @(posedge clk or negedge clrn)
        if (!clrn) q <= 0; else q <= d;
endmodule

module dff8_we(input [7:0] d, input clk, we, clrn, output reg [7:0] q);
    always @(posedge clk or negedge clrn)
        if (!clrn) q <= 0; else if (we) q <= d;
endmodule

module dff16(input [15:0] d, input clk, clrn, output reg [15:0] q);
    always @(posedge clk or negedge clrn)
        if (!clrn) q <= 0; else q <= d;
endmodule

module hex7seg(input [3:0] hex, output reg [6:0] seg);
    always @(*) case(hex)
        0: seg = 7'b1000000; 1: seg = 7'b1111001;
        2: seg = 7'b0100100; 3: seg = 7'b0110000;
        4: seg = 7'b0011001; 5: seg = 7'b0010010;
        6: seg = 7'b0000010; 7: seg = 7'b1111000;
        8: seg = 7'b0000000; 9: seg = 7'b0010000;
        10: seg = 7'b0001000; 11: seg = 7'b0000011;
        12: seg = 7'b1000110; 13: seg = 7'b0100001;
```

```
        14: seg = 7'b0000110; 15: seg = 7'b0001110;
    endcase
endmodule
```