W12. Sequential Multiplier for Sign-Magnitude numbers

Hardware design of a Sign-Magnitude sequential multiplier

1. Laboratory objective

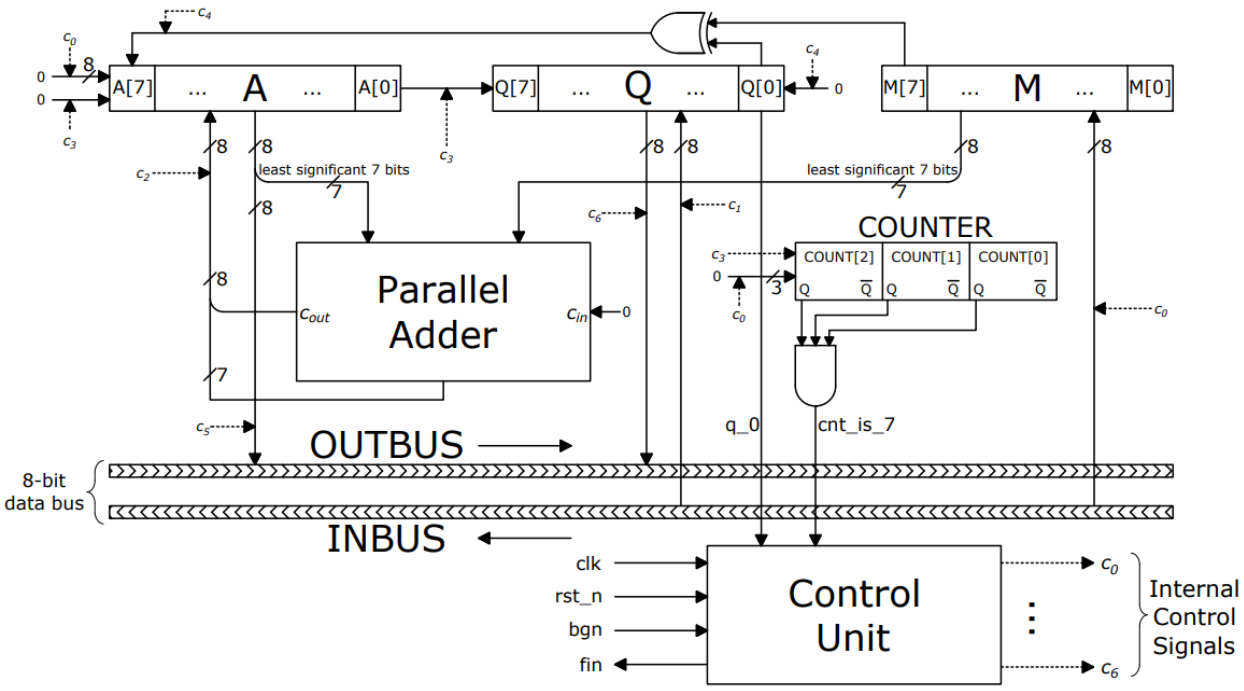
Construct a sequential architecture for Sign-Magnitude multiplication

2. Theoretical Background

2.1 Architecture's characteristics

- Operate fractional numbers on 8 bits
- Design and customize multiplier's components
- Design the control path of the architecture

2.2 Sign-Magnitude Sequential multiplier architecture



Note: Except **rst_b**, all other signals are *synchronous*.

2.2.1 Register M (solved)

Stores the multiplicand, having the following interface:

- **ld_ibus**: loads multiplicand's value from INBUS
  - **i_bus**: the INBUS line
- **q**: register's content

```
module reg_m (

input clk, rst_b, ld_ibus, [7:0] ibus,

output reg [7:0] q

             );

always @(posedge clk, negedge rst_b) begin

if (!rst_b)           q <= 7'd0; // Reset the register's content

else if (ld_ibus)     q <= ibus; // Load the value from INBUS lines

end

endmodule
```

2.2.2 Register Q (solved)

Stores the multiplier, having the following interface:

- **clr_lsb**: clear register's LSB (Least Significant Bit)
- **ld_ibus**: loads multiplier's value from INBUS
  - **ibus**: the INBUS lines
- **sh_r**: right shifts the register's content
  - **sh_i**: bit to be loaded into MSB (Most Signficiant Bit) during right shift
- **ld_obus**: deliver accumulator's content onto OUTBUS
  - **obus**: the OUTBUS lines
- **q**: register's content

```
module reg_q (

input clk, rst_b, clr_lsb, ld_ibus, ld_obus, sh_r,

input sh_i, [7:0] ibus,

output reg [7:0] obus, [7:0] q

             );

always @(posedge clk, negedge rst_b) begin

if (!rst_b)           q <= 8'b0; // Reset the register's content

else if (clr_lsb)     q[0] <= 1'b0; // Clear the least significant bit

else if (ld_ibus)     q <= ibus; // Load the value from INBUS lines

else if (shr)         q <= {sh_i, q[7:1]}; // Rshift and load sh_i

end

// Write content to OUTBUS lines

always @(*) begin

if (ld_obus)          obus = q; // Load register content onto OUTBUS

else                  obus = 8'bz; // High impedance state

end

endmodule
```

2.2.3 Register A (solved)

Multiplier's accumulator, it has the following interface:

- **clr**: synchronous, active high reset
- **ld_sum**: loads adder's result into accumulator

    - **sum**: adder's result to be loaded into accumulator
- **ld_sgn**: loads accumulator's sign
    - **sgn**: accumulator's sign
- **ld_obus**: unload accumulator's content onto OUTBUS
    - **obus**: the OUTBUS lines
- **sh_r**: right shift register's content
    - **sh_i**: bit to be loaded into MSB during right shift
- **q**: register's content

```
module reg_a (

input clk, rst_b, clr, sh_r, ld_sgn, ld_obus, ld_sum,

input sh_i, sgn, [7:0] sum,

output reg [7:0] obus, [7:0] q                    );

always @(posedge clk, negedge rst_b) begin

if (!rst_b)           q <= 8'b0; // Reset the register's content

else if (clr)         q <= 8'b0; // Clear the register's content

else if (ld_sum)      q <= sum; // Load the sum into the accumulator

else if (shr)         q <= {sh_i, q[7:1]}; // Rshift and load sh_i

else if (ld_sgn)      q[7] <= sgn ; // Load the sign into the MSB

end

// Write content to OUTBUS lines

always @(*) begin

obus = (ld_obus) ? q : 8'bz; // Write content only when ld_obus == 1

end

endmodule
```

2.2.4 Parallel Adder (solved)

```
module parallel_adder (
input [7:0] a, b, // Operands A and B
input cin, // Carry-in
output reg [7:0] sum, // Sum result
output reg cout // Carry-out      );
always @(*) begin
{cout,sum} = a + b + cin; // Perform addition with carry
end
endmodule
```

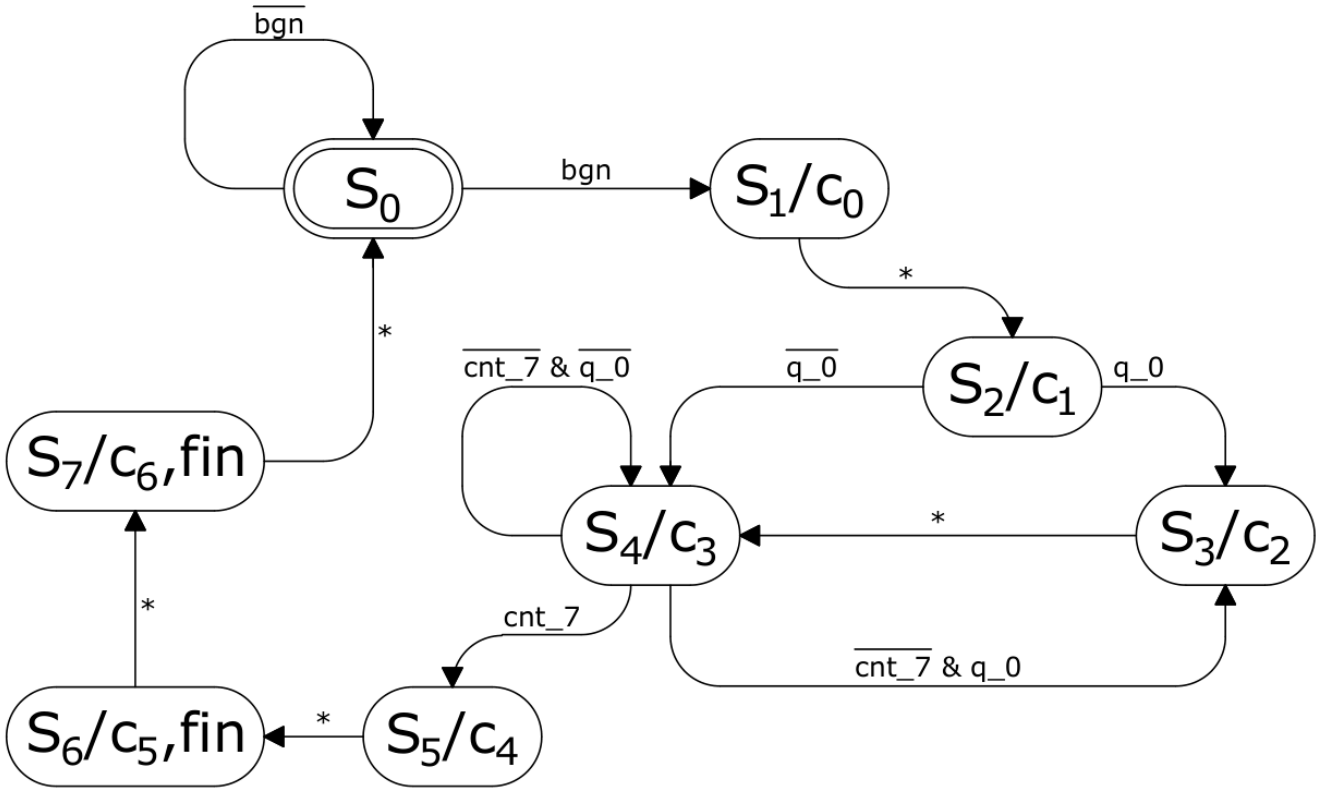
2.2.5 Counter (solved)

```
module counter (
input clk, rst_b, clr, c_up, // Clock, reset, clear, and count up
output reg [2:0] q // 3-bit counter output
);
always @(posedge clk, negedge rst_b) begin
if (!rst_b)           q <= 3'b0; // Reset the counter's content
else if (clr)         q <= 8'b0; // Clear the counter's content
else if (c_up)        q <= q + 1; // Increment the counter value
end
endmodule
```

2.2.6 Control Unit (solved)

The Control Unit has the following interface:

- **bgn**: starts the multiplication
- **q_0**: register Q's LSB
- **cnt_is_7**: active if COUNTER's value is 7
- **c0**: clears registers A and COUNTER, loads M from INBUS
- **c1**: loads Q from INBUS
- **c2**: loads adder's result back into A
- **c3**: right-shifts A concatenated with Q (introduce a bit of 0 in register A's MSB), increments COUNTER
- **c4**: sets sign of register A, clear register Q's LSB
- **c5/c6**: deliver register A/Q to OUTBUS
- **fin**: marks multiplication completion, activated along c5, c6



Important: Control unit must be triggered by the opposite edge of the clock, compared to all the other sequential elements in the architecture!

```
module control_unit (
input clk, rst_b, bgn, q_0, cnt_is_7, // Inputs
output c0, c1, c2, c3, c4, c5, c6, fin // Outputs
);          //Five-step implementation method
//Step 1 – Defining the states of the machine
localparam S0 = 3'b000; // Initial state
localparam S1 = 3'b001; // State for c0
localparam S2 = 3'b010; // State for c1
localparam S3 = 3'b011; // State for c2
localparam S4 = 3'b100; // State for c3
localparam S5 = 3'b101; // State for c4
localparam S6 = 3'b110; // State for c5
localparam S7 = 3'b111; // State for c6
//Step 2 – Defining the current state and next state of the machine
reg [2:0] state, next_state;
//Step 3 – Constructing the next state of the machine (always block)
always @(*) begin
case (state)
S0: if (bgn) next_state = S1; // Start multiplication
    else next_state = S0;
S1: next_state = S2;          // Load Q from INBUS
```

```
S2: if (q_0) next_state = S3;       else next_state = S4;

S3: next_state = S4;

S4: if (!cnt_is_7 && q_0) next_state = S3;

else if (!cnt_is_7 && !q_0)   next_state = S4; else next_state = S5;

S5: next_state = S6;          // Finalize multiplication

S6: next_state = S7;          // Finalize multiplication

S7: next_state = S0;          // Return to initial state

default: next_state = S0;     // Default state

endcase

end

//Step 4 – Constructing the outputs of the machine (always block)

always @(*) begin

{c0, c1, c2, c3, c4, c5, c6, fin} = 8'b0; // All outputs to 0

case (state)

S0: ;                   // No control signals

S1: c0 = 1;             /* Clear registers A and COUNTER, load M from
INBUS*/

S2: c1 = 1;             // Load Q from INBUS

S3: c2 = 1;             // Load adder result back into A

S4: c3 = 1;             // Right-shift A/Q, increment COUNTER

S5: c4 = 1;       // Set sign of A, clear Q's LSB, and signal finish
```

```
S6: {c5, fin} = 2'b11;    // Deliver A/Q to OUTBUS and signal finish
S7: {c6, fin} = 2'b11;    // Deliver A/Q to OUTBUS and signal finish
endcase
end
//Step 5 – Updating the current state in a sequential always block
always @(posedge clk, negedge rst_b) begin
        if (!rst_b)
            state <= S0; // Reset to initial state
        else
            state <= next_state; // Transition to the next state
end
```

3. Integrating the components into the Sign-magnitude design

```
module sm_unit (
input clk,
input rst_b,
input bgn,
input [7:0] ibus,
output fin,
output [7:0] obus );
```

```
// Internal signals
    wire c0, c1, c2, c3, c4, c5, c6;
    wire [7:0] reg_m_out, reg_q_out, reg_a_out;
    wire [7:0] sum;
    wire cout;
    wire q_0, cnt_is_7;
    wire [2:0] count;
// Instantiate Register M (stores multiplicand)
    reg_m u_reg_m       (
        .clk(clk),
        .rst_b(rst_b),
        .ld_ibus(c0),    // Load multiplicand when c0 is active
        .ibus(ibus),
        .q(reg_m_out)    );
// Instantiate Register Q (stores multiplier)
    reg_q u_reg_q       (
        .clk(clk),
        .rst_b(rst_b),
        .clr_lsb(c4),    // Clear LSB when c4 is active
        .ld_ibus(c1),    // Load multiplier when c1 is active
```

```
        .ld_obus(c6),    // Deliver content to OUTBUS when c6 is active
        .shr(c3),        // Right shift when c3 is active
        .sh_i(1'b0),     // Insert 0 during shift
        .ibus(ibus),
        .obus(obus),     // Connected to the output bus
        .q(reg_q_out)    );
// Instantiate Register A (accumulator)
    reg_a u_reg_a        (
        .clk(clk),
        .rst_b(rst_b),
        .clr(c0),         // Clear accumulator when c0 is active
        .shr(c3),         // Right shift when c3 is active
        .ld_sgn(c4),      // Load sign when c4 is active
        .ld_obus(c5),     // Deliver content to OUTBUS when c5 - active
        .ld_sum(c2),      // Load sum when c2 is active
        .sh_i(1'b0),      // Insert 0 during shift
        .sgn(reg_q_out[7]), // Use MSB of reg_q_out as sign
        .sum(sum),        // Input from the adder
        .obus(obus),      // Connected to the output bus
        .q(reg_a_out)
                          );
```

```
// Instantiate Parallel Adder
    parallel_adder u_adder (
        .a(reg_a_out),    // Accumulator output
        .b(reg_m_out),    // Multiplicand
        .cin(1'b0),       // No initial carry
        .sum(sum),        // Adder result
        .cout(cout)       // Carry out (not used here)
    );
// Instantiate Counter
    counter u_counter (
        .clk(clk),
        .rst_b(rst_b),
        .clr(c0),         // Clear counter when c0 is active
        .inc(c3),         // Increment counter when c3 is active
        .count(count)     // Counter output
    );
// Assign counter signals
    assign cnt_is_7 = (count == 3'b111); // Counter reaches 7
    assign q_0 = reg_q_out[0];           // LSB of Register Q
// To be continued on the next page
// ...
```

```
// Instantiate Control Unit

    ctrl_unit u_ctrl (

        .clk(clk),

        .rst_b(rst_b),

        .bgn(bgn),

        .q_0(q_0),

        .cnt_is_7(cnt_is_7),

        .c0(c0),

        .c1(c1),

        .c2(c2),

        .c3(c3),

        .c4(c4),

        .c5(c5),

        .c6(c6),

        .fin(fin)          // Finish signal

    );

endmodule
```

4. References

Vlăduţiu, M. (2012). *Computer Arithmetic*: *Algorithms and Hardware Implementations*. Springer Science & Business Media. ISBN: 978-3-642-18314-4.

Roth, C. H., Jr., & Kinney, L. L. (2013). *Fundamentals of Logic Design* (7th ed.). Cengage Learning. ISBN: 978-1133628477.