CS M152B Lab 1 Report

Introduction

This lab focuses on designing and implementing a 16-bit asynchronous ALU and a 32 16-bit register file using structural and behavioral Verilog. It emphasizes understanding hierarchical design, difference between structural and behavioral Verilog, arithmetic and logical shifts, and ALU operations. The lab also explores engineering constraints and standards in digital circuit design.

When designing an asynchronous 16-bit ALU, engineering standards and constraints must be considered that impact its design, implementation, and performance. The ALU must follow a standard instruction set architecture (ISA), defining how it executes operations (e.g. RISC-V). Hardware standards ensure that the ALU can run on FPGA platforms. A common ALU standard is being able to handle signed/unsigned integers correctly. Its asynchronous design means that the ALU is not timing dependent, so it must manage critical path delays independently. Power usage is another factor as well as minimizing area/complexity (i.e., number of logic gates).

Implementation

Part 1: 1-bit ALU

- 1-bit ALU

  `module ALU1`

  ALU1 is a standalone 1-bit ALU implemented with no behavioral code that supports the following operations: NOT, AND, OR, and FULL ADDER. We also check overflow for the case of addition.

```
module ALU1 (
    input A, B,
    input [1:0] ALUCtrl,
    output S,
    output Overflow
);

    wire D_not;
    wire D_and;
    wire D_or;
    wire D_add;

    not(D_not, A);
    and(D_and, A, B);
    or(D_or, A, B);
    wire add_of;
    addbit add (.a(A), .b(B), .cin(0), .s(D_add), .cout(add_of));

    m41 mux_out(
        .D0(D_not),
        .D1(D_and),
        .D2(D_or),
```

```
        .D3(D_add),
        .S1(ALUCtrl[1]),
        .S0(ALUCtrl[0]),
        .Y(S)
    );

    m41 mux_of(
        .D0(0),
        .D1(0),
        .D2(0),
        .D3(add_of),
        .S1(ALUCtrl[1]),
        .S0(ALUCtrl[0]),
        .Y(Overflow)
    );
endmodule
```

- 2-to-1, 4-to-1, 16-to-1 multiplexer

  ### `module m21`

  This module implements a 2-to-1 multiplexer that supports 16-bit inputs. I.e., there is a 1-bit selector that chooses between 2 16-bit inputs. The commented out code is for a 2-to-1 multiplexer with only a 1-bit granularity.

```
module m21(
    output [15:0] Y,
    input [15:0] D0, D1, // input D0, D1,
    input S
);

    wire [15:0] T1, T2;
    wire Sbar;

    not (Sbar, S);

    //and (T1, D1, S);
    and (T1[0], D1[0], S);
    and (T1[1], D1[1], S);
    and (T1[2], D1[2], S);
    and (T1[3], D1[3], S);
    and (T1[4], D1[4], S);
    and (T1[5], D1[5], S);
    and (T1[6], D1[6], S);
    and (T1[7], D1[7], S);
    and (T1[8], D1[8], S);
    and (T1[9], D1[9], S);
    and (T1[10], D1[10], S);
    and (T1[11], D1[11], S);
    and (T1[12], D1[12], S);
    and (T1[13], D1[13], S);
    and (T1[14], D1[14], S);
    and (T1[15], D1[15], S);

    //and (T2, D0, Sbar);
```

```
    and (T2[0], D0[0], Sbar);
    and (T2[1], D0[1], Sbar);
    and (T2[2], D0[2], Sbar);
    and (T2[3], D0[3], Sbar);
    and (T2[4], D0[4], Sbar);
    and (T2[5], D0[5], Sbar);
    and (T2[6], D0[6], Sbar);
    and (T2[7], D0[7], Sbar);
    and (T2[8], D0[8], Sbar);
    and (T2[9], D0[9], Sbar);
    and (T2[10], D0[10], Sbar);
    and (T2[11], D0[11], Sbar);
    and (T2[12], D0[12], Sbar);
    and (T2[13], D0[13], Sbar);
    and (T2[14], D0[14], Sbar);
    and (T2[15], D0[15], Sbar);

    //or (Y, T1, T2);
    or (Y[0], T1[0], T2[0]);
    or (Y[1], T1[1], T2[1]);
    or (Y[2], T1[2], T2[2]);
    or (Y[3], T1[3], T2[3]);
    or (Y[4], T1[4], T2[4]);
    or (Y[5], T1[5], T2[5]);
    or (Y[6], T1[6], T2[6]);
    or (Y[7], T1[7], T2[7]);
    or (Y[8], T1[8], T2[8]);
    or (Y[9], T1[9], T2[9]);
    or (Y[10], T1[10], T2[10]);
    or (Y[11], T1[11], T2[11]);
    or (Y[12], T1[12], T2[12]);
    or (Y[13], T1[13], T2[13]);
    or (Y[14], T1[14], T2[14]);
    or (Y[15], T1[15], T2[15]);
endmodule
```

We can easily extend this to a 4-to-1 multiplexer, and then to a 16-to-1 multiplexer that supports 16-bit inputs:

`module m41`

The 4-to-1 multiplexer essentially connects the outputs of two 2-to-1 muxes to the inputs of a third 2-to-1 mux; it requires two select lines `S0` and `S1`.

```
module m41(
    output [15:0] Y,
    input [15:0] D0, D1, D2, D3,
    input S1, S0
);
    wire [15:0] T1, T2;
    m21 mux1(T1, D0, D1, S0);
    m21 mux2(T2, D2, D3, S0);

    m21 mux3(Y, T1, T2, S1);
endmodule
```

`module m161`

The 16-to-1 multiplexer is similar to the 4-to-1 mux as it connects the outputs of four 4-to-1 muxes to the inputs of a fifth 4-to-1 mux; it requires a 4-bit selector `S`.

```
module m161(
    output [15:0] Y,
    input [15:0] D0, D1, D2, D3, D4, D5, D6, D7,
    D8, D9, D10, D11, D12, D13, D14, D15,            // 16 data inputs
    input [3:0] S                                    // 4 bit selector
);
    wire [15:0] T0, T1, T2, T3;
    m41 mux1(T0, D0, D1, D2, D3, S[1], S[0]);
    m41 mux2(T1, D4, D5, D6, D7, S[1], S[0]);
    m41 mux3(T2, D8, D9, D10, D11, S[1], S[0]);
    m41 mux4(T3, D12, D13, D14, D15, S[1], S[0]);

    m41 mux5(Y, T0, T1, T2, T3, S[3], S[2]);
endmodule
```

- 1-bit, 16-bit full adder

`module addbit`

This module implements a 1-bit full adder using custom XOR gates along with basic AND, OR logic gates. `s` (sum bit) is the result of `a + b + cin` (carry-in from previous bit addition) and `cout` is the carry-out bit.

```
module addbit(
    input a, b, cin,
    output s, cout
);
    wire atemp, btemp, ctemp, anot, bnot;

    my_xor xor1 (                               // xor(atemp, a, b);
        .a(a),
        .b(b),
        .out(atemp)
    );

    and (btemp, a, b);

    my_xor xor2 (                               // xor(s, cin, atemp);
        .a(cin),
        .b(atemp),
        .out(s)
    );

    and (ctemp, cin, atemp);
    or (cout, ctemp, btemp);
endmodule
```

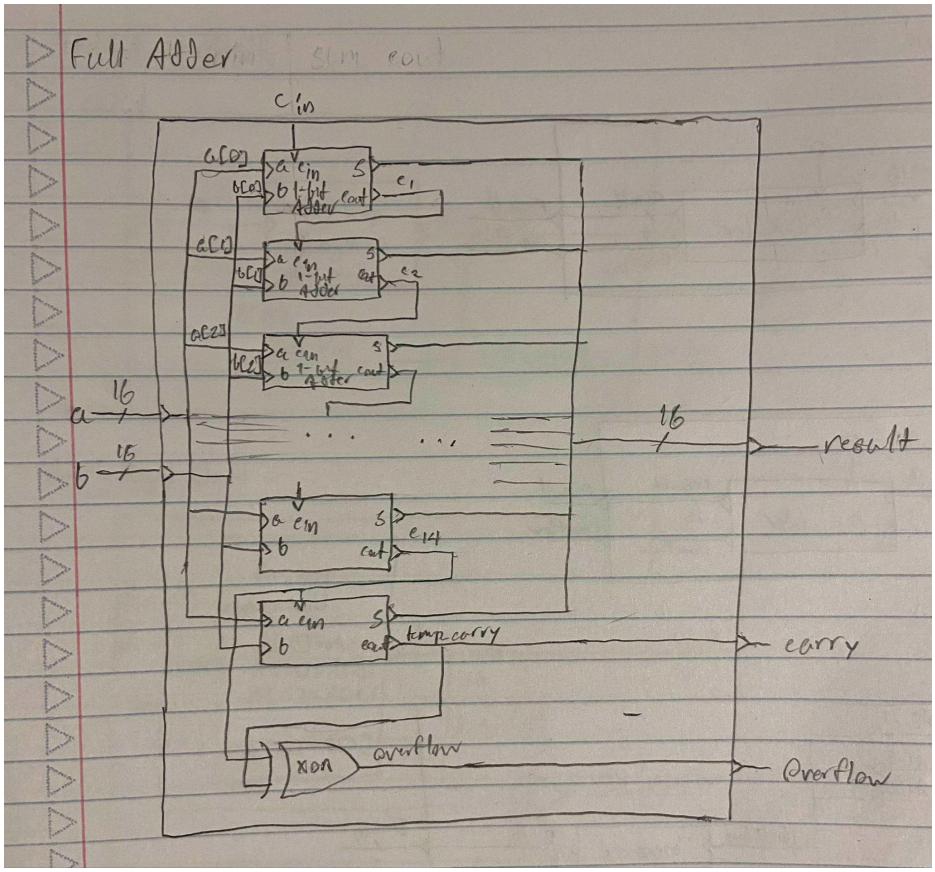
- 16-bit full adder (schematic required)

`module fulladder`

This module extends the 1-bit full adder into a 16-bit full adder with a ripple-carry. It performs binary addition of two 16-bit inputs `r1` and `r2` along with a carry-in bit `ci` to produce a 16-bit sum `result`, carry-out bit `carry`, and `Overflow` flag. The first adder take in `ci` as the initial `cin`, each subsequent adder takes `cout` of the previous adder as its `cin`, and the final adder produces the final `carry`. Overflow occurs when the carry into the MSB (`c14`) is different from the carry out of the MSB (`temp_carry`). We check this condition using a custom XOR function.

```
module fulladder(
    input [15:0] r1,
    input [15:0] r2,
    input ci,
    output carry,
    output [15:0] result,
    output Overflow
);
    wire c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15;
    addbit u0 (.a(r1[0]), .b(r2[0]), .cin(ci), .s(result[0]), .cout(c1));
    addbit u1 (.a(r1[1]), .b(r2[1]), .cin(c1), .s(result[1]), .cout(c2));
    addbit u2 (.a(r1[2]), .b(r2[2]), .cin(c2), .s(result[2]), .cout(c3));
    addbit u3 (.a(r1[3]), .b(r2[3]), .cin(c3), .s(result[3]), .cout(c4));
    addbit u4 (.a(r1[4]), .b(r2[4]), .cin(c4), .s(result[4]), .cout(c5));
    addbit u5 (.a(r1[5]), .b(r2[5]), .cin(c5), .s(result[5]), .cout(c6));
    addbit u6 (.a(r1[6]), .b(r2[6]), .cin(c6), .s(result[6]), .cout(c7));
    addbit u7 (.a(r1[7]), .b(r2[7]), .cin(c7), .s(result[7]), .cout(c8));
    addbit u8 (.a(r1[8]), .b(r2[8]), .cin(c8), .s(result[8]), .cout(c9));
    addbit u9 (.a(r1[9]), .b(r2[9]), .cin(c9), .s(result[9]), .cout(c10));
    addbit u10 (.a(r1[10]), .b(r2[10]), .cin(c10), .s(result[10]), .cout(c11));
    addbit u11 (.a(r1[11]), .b(r2[11]), .cin(c11), .s(result[11]), .cout(c12));
    addbit u12 (.a(r1[12]), .b(r2[12]), .cin(c12), .s(result[12]), .cout(c13));
    addbit u13 (.a(r1[13]), .b(r2[13]), .cin(c13), .s(result[13]), .cout(c14));
    addbit u14 (.a(r1[14]), .b(r2[14]), .cin(c14), .s(result[14]), .cout(c15));

    wire temp_carry;
    addbit u15 (.a(r1[15]), .b(r2[15]), .cin(c15), .s(result[15]),
     .cout(temp_carry));

    xor_op xor_ab(.A(c14), .B(temp_carry), .D(Overflow)); // Check for overflow
    assign carry = temp_carry;
endmodule
```

- `16-bit full adder schematic`



Part 2: 16-bit ALU

`module ALU16`

A 16-bit asynchronous ALU that performs the following operations: subtraction, addition, bitwise OR, bitwise AND, decrement, increment, invert, arithmetic shift left, arithmetic shift right, logical shift left, logical shift right, and set on less than or equal. `A`, `B`, and `S` are 16-bit signed two's complement. We have 2 output signals `Zero` and `Overflow` that check if the output is 0 and if the ALU operation results in an overflow. Our implementation is essentially 2 16-to-1 muxes: an output mux that computes the result for each operation and chooses one based on `ALUCtrl` and an overflow mux that outputs the overflow bit for the given operation.

```
module ALU16 (
    input [15:0] A, B,
    input [3:0] ALUCtrl,
    output [15:0] S,
    output Zero,
    output Overflow
);
    wire [15:0] D_add, D_sub, D_or, D_and, D_dec, D_inc, D_inv, D_arith_left,
    D_arith_right, D_log_left, D_log_right, D_sle;
    wire [15:0] of;

    fullsubber sub(
        .r1(A),
```

```
    .r2(B),
    .ci(0),
    //.carry(carry1),               // Final carry output is omitted for sub/add
    .result(D_sub),
    .Overflow(of[0])
);

fulladder add(
    .r1(A),
    .r2(B),
    .ci(0),
    //.carry(carry2),               // Final carry output is omitted for sub/add
    .result(D_add),
    .Overflow(of[1])
);

bitwise_or or_op(
    .D(D_or),
    .A(A),
    .B(B)
);
assign of[2] = 0;

bitwise_and and_op(
    .D(D_and),
    .A(A),
    .B(B)
);
assign of[3] = 0;

decrement dec_op(
    .D(D_dec),
    .A(A),
    .Overflow(of[4])
);

increment inc_op(
    .D(D_inc),
    .A(A),
    .Overflow(of[5])
);

invert inv_op(
    .D(D_inv),
    .A(A),
    .Overflow(of[6])
);

arith_shift_left arith_left(
    .D(D_arith_left),
    .A(A),
    .B(B),
    .Overflow(of[7])
);
```

```
arith_shift_right arith_right(
    .D(D_arith_right),
    .A(A),
    .B(B),
    .Overflow(of[8])
);

log_shift_left log_left(
    .D(D_log_left),
    .A(A),
    .B(B),
    .Overflow(of[9])
);

log_shift_right log_right(
    .D(D_log_right),
    .A(A),
    .B(B),
    .Overflow(of[10])
);

set_less_equal sle(
    .D(D_sle),
    .A(A),
    .B(B)
);
assign of[11] = 0;

m161 outmux(
    .Y(S),
    .D0(D_sub),
    .D1(D_add),
    .D2(D_or),
    .D3(D_and),
    .D4(D_dec),
    .D5(D_inc),
    .D6(D_inv),
    .D7(D_arith_left),
    .D8(D_arith_right),
    .D9(D_log_left),
    .D10(D_log_right),
    .D11(D_sle),
    .D12(16'd0),
    .D13(16'd0),
    .D14(16'd0),
    .D15(16'd0),
    .S(ALUCtrl)
);

m161 ofmux(
    .Y(Overflow),
    .D0(of[0]),
    .D1(of[1]),
```

```
        .D2(of[2]),
        .D3(of[3]),
        .D4(of[4]),
        .D5(of[5]),
        .D6(of[6]),
        .D7(of[7]),
        .D8(of[8]),
        .D9(of[9]),
        .D10(of[10]),
        .D11(of[11]),
        .D12(16'd0),
        .D13(16'd0),
        .D14(16'd0),
        .D15(16'd0),
        .S(ALUCtrl)
    );

    // Set Zero flag
    wire [15:0] xor_out;
    bitwise_xor xor_ab(.A(S), .B(16'd0), .D(xor_out)); // If A == B, then xor_out = 0.
    all_zero zero_xor(.A(xor_out), .D(Zero));
endmodule
```

12 ALU operations:

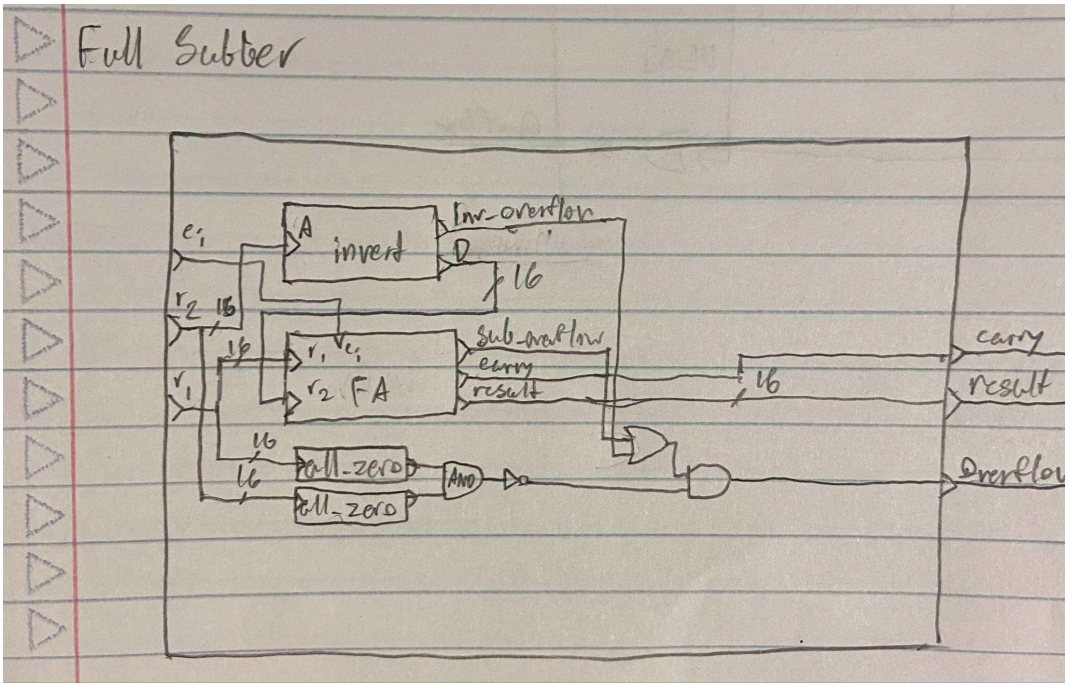
1. `module fullsubber`

   This module is derived from our 16-bit full adder. Instead of `r1 + r2`, it performs `r1 + invert(r2)`, which is essentially `r1 - r2`. Overflow occurs if inverting `r2` overflows or `r1 - r2` overflows. We explicitly check an edge case for when `r1` and `r2` are both zero. This is because `0 + (inv(0))` results in a fake overflow since `inv(0) = bitwiseNot(0) + 1 = 1…1 + 1 = 1…0`.

   ```
   module fullsubber(
       input [15:0] r1,
       input [15:0] r2,
       input ci,
       output carry,
       output [15:0] result,
       output Overflow
   );
       wire [15:0] r2_neg;
       wire inv_overflow;
       wire sub_overflow;

       invert inv_op(.D(r2_neg), .A(r2),. Overflow(inv_overflow));

       fulladder fa(
           .r1(r1),
           .r2(r2_neg),
           .ci(ci),
           .carry(carry),
           .result(result),
           .Overflow(sub_overflow)
   ```

```
    );

    // Check if r1 and r2 are zero using all_zero module
    wire r1_zero, r2_zero, both_zero;
    all_zero check_r1(.A(r1), .D(r1_zero));
    all_zero check_r2(.A(r2), .D(r2_zero));
    and both_zero_check(both_zero, r1_zero, r2_zero); // Both are zero

    wire override;
    not(override, both_zero);

    wire overflow_signal;
    or(overflow_signal, inv_overflow, sub_overflow);

    // Corrected overflow
    and(Overflow, overflow_signal, override);
endmodule
```

2. We used the `fulladder` module to implement the ALU add operation.
    a. Fullsubber schematic



3. `module bitwise_or`

Bitwise OR operation between two 16-bit inputs `A` and `B` to produce 16-bit output `D`.

```
module bitwise_or(
    input [15:0] A, B,
    output [15:0] D
);
    or (D[0], A[0], B[0]);
    or (D[1], A[1], B[1]);
    or (D[2], A[2], B[2]);
    or (D[3], A[3], B[3]);
```

```
    or (D[4], A[4], B[4]);
    or (D[5], A[5], B[5]);
    or (D[6], A[6], B[6]);
    or (D[7], A[7], B[7]);
    or (D[8], A[8], B[8]);
    or (D[9], A[9], B[9]);
    or (D[10], A[10], B[10]);
    or (D[11], A[11], B[11]);
    or (D[12], A[12], B[12]);
    or (D[13], A[13], B[13]);
    or (D[14], A[14], B[14]);
    or (D[15], A[15], B[15]);
endmodule
```

4. `module bitwise_and`

   Bitwise AND operation between two 16-bit inputs `A` and `B` to produce 16-bit output `D`.

```
module bitwise_and(
    input [15:0] A, B,
    output [15:0] D
);
    and (D[0], A[0], B[0]);
    and (D[1], A[1], B[1]);
    and (D[2], A[2], B[2]);
    and (D[3], A[3], B[3]);
    and (D[4], A[4], B[4]);
    and (D[5], A[5], B[5]);
    and (D[6], A[6], B[6]);
    and (D[7], A[7], B[7]);
    and (D[8], A[8], B[8]);
    and (D[9], A[9], B[9]);
    and (D[10], A[10], B[10]);
    and (D[11], A[11], B[11]);
    and (D[12], A[12], B[12]);
    and (D[13], A[13], B[13]);
    and (D[14], A[14], B[14]);
    and (D[15], A[15], B[15]);
endmodule
```

5. `module decrement`

   This module subtracts 1 from a 16-bit input `A`. It is implemented using the `fullsubber` module where `r2 = 1`. There is only one possible overflow case where decrementing `A = INT_MIN` overflows to `INT_MAX`.

```
module decrement(
    input [15:0] A,
    output [15:0] D,
    output Overflow
);
    fullsubber fs (.r1(A), .r2(1), .ci(0), .carry(co), .result(D),
    .Overflow(Overflow));
endmodule
```

6. `module increment`

This module adds 1 to a 16-bit input `A`. It is implemented using the `fulladder` module where `r2 = 1`. There is only one possible overflow case where incrementing `A = INT_MAX` overflows to `INT_MIN`.

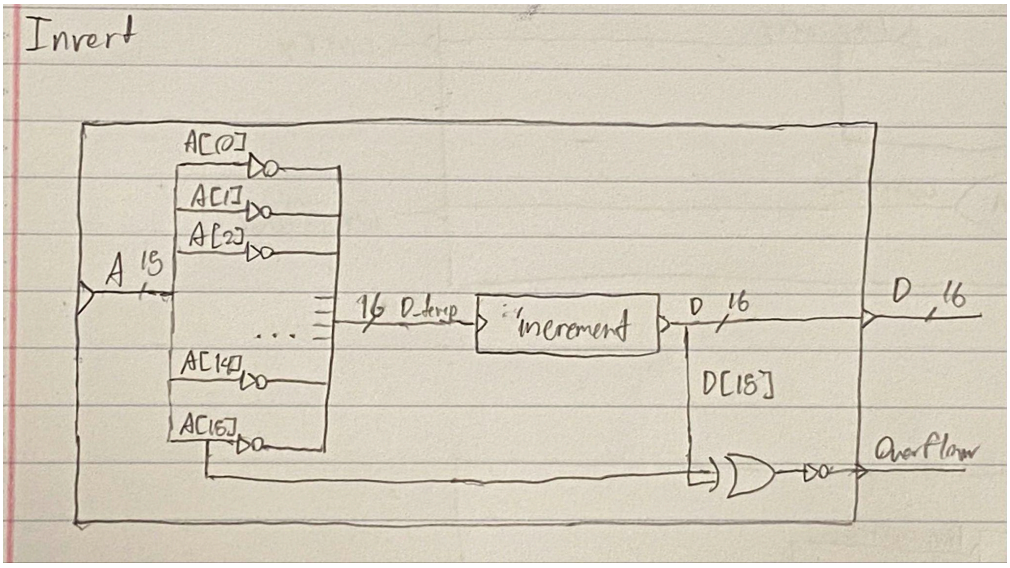
```
module increment(
    input [15:0] A,
    output [15:0] D,
    output Overflow
);
    fulladder fa (.r1(A), .r2(1), .ci(0), .carry(co), .result(D),
    .Overflow(Overflow));
endmodule
```

7. `module invert`

This module performs the invert in two's complement by computing `bitwiseNot(A) + 1`. Overflow occurs in two's complement inversion when the MSB stays the same (i.e., `A[15] = D[15]`). We check equality using XOR, then negate the result.

```
module invert(
    input [15:0] A,
    output [15:0] D,
    output Overflow
);
    wire orig_sign = A[15];
    wire [15:0] D_temp;

    // bitwiseNot(A)
    not (D_temp[0], A[0]);
    not (D_temp[1], A[1]);
    not (D_temp[2], A[2]);
    not (D_temp[3], A[3]);
    not (D_temp[4], A[4]);
    not (D_temp[5], A[5]);
    not (D_temp[6], A[6]);
    not (D_temp[7], A[7]);
    not (D_temp[8], A[8]);
    not (D_temp[9], A[9]);
    not (D_temp[10], A[10]);
    not (D_temp[11], A[11]);
    not (D_temp[12], A[12]);
    not (D_temp[13], A[13]);
    not (D_temp[14], A[14]);
    not (D_temp[15], A[15]);

    // bitwiseNot(A) + 1
    increment inc_op(.A(D_temp), .D(D));

    wire of_temp;
    xor_op xor_ab (.A(D[15]), .B(orig_sign), .D(of_temp));

    not(Overflow, of_temp);
endmodule
```

a. invert schematic



8. `module arith_shift_left`

Performs arithmetic left shift on a 16-bit input `A` by shift amount `B`. Arithmetic left shifting is equivalent to multiplying `A` by 2 `B` times. Overflow occurs whenever MSB changes. We check this using XOR.

```
module arith_shift_left(
    input [15:0] A, B,
    output [15:0] D,
    output Overflow
);
    wire orig_sign = A[15];
    reg [15:0] A_reg;
    integer i;

    always @(*) begin
        A_reg = A;
        for (i = 0; i < B; i = i + 1) begin
            A_reg = A_reg * 2;
        end
    end

    xor_op xor_ab(.A(A_reg[15]), .B(orig_sign), .D(Overflow));
    assign D = A_reg;
endmodule
```

9. `module arith_shift_right`

Performs arithmetic right shift on a 16-bit input `A` by shift amount `B`. Note that arithmetic right shifting preserves original sign. On each iteration, the old LSB is discarded and each (i)-th bit moves to the (i-1)-th position; the original sign bit is propagated down starting from the MSB. Overflow does not occur in arithmetic right shift since the sign bit never changes.

```
module arith_shift_right(
    input [15:0] A, B,
```

```
    output [15:0] D,
    output Overflow
);

    wire orig_sign = A[15];
    reg [15:0] A_new;
    reg [15:0] A_old;
    integer i;

    always @(*) begin
        A_new = A;
        A_old = A;
        for (i = 0; i < B; i = i + 1) begin
            A_new[15] = orig_sign;
            A_new[14] = A_old[15];
            A_new[13] = A_old[14];
            A_new[12] = A_old[13];
            A_new[11] = A_old[12];
            A_new[10] = A_old[11];
            A_new[9] = A_old[10];
            A_new[8]= A_old[9];
            A_new[7] = A_old[8];
            A_new[6] = A_old[7];
            A_new[5] = A_old[6];
            A_new[4] = A_old[5];
            A_new[3] = A_old[4];
            A_new[2] = A_old[3];
            A_new[1] = A_old[2];
            A_new[0] = A_old[1];
            A_old = A_new;                      // Old becomes new
        end
    end

    Assign Overflow = 0;                        // Overflow is hardcoded
    assign D = A_new;
endmodule
```

10. module log_shift_left

    Both logical left shift and arithmetic left shift have the same behavior. No sign bit preservation is required, making them identical in implementation and overflow detection. Logical shifting never results in an overflow.

```
module log_shift_left(
    input [15:0] A, B,
    output [15:0] D,
    output Overflow
);
    /* ... same as module arith_shift_left ... */
    assign Overflow = 0;                        // Overflow is hardcoded
endmodule
```

11. module log_shift_right

Logical right shift is essentially the same as arithmetic right shift, but it propagates 0 down starting from MSB instead of the original sign bit. Logical shifting never results in an overflow.

```
module log_shift_right(
    input [15:0] A, B,
    output [15:0] D,
    output Overflow
);
    wire orig_sign = A[15];
    reg [15:0] A_new;
    reg [15:0] A_old;
    integer i;

    always @(*) begin
        A_new = A;
        A_old = A;
        for (i = 0; i < B; i = i + 1) begin
            A_new[15] = 0;
            A_new[14] = A_old[15];
            A_new[13] = A_old[14];
            A_new[12] = A_old[13];
            A_new[11] = A_old[12];
            A_new[10] = A_old[11];
            A_new[9] = A_old[10];
            A_new[8]= A_old[9];
            A_new[7] = A_old[8];
            A_new[6] = A_old[7];
            A_new[5] = A_old[6];
            A_new[4] = A_old[5];
            A_new[3] = A_old[4];
            A_new[2] = A_old[3];
            A_new[1] = A_old[2];
            A_new[0] = A_old[1];
            A_old = A_new;                      // Old becomes new
        end
    end

    assign Overflow = 0;
    assign D = A_new;
endmodule
```

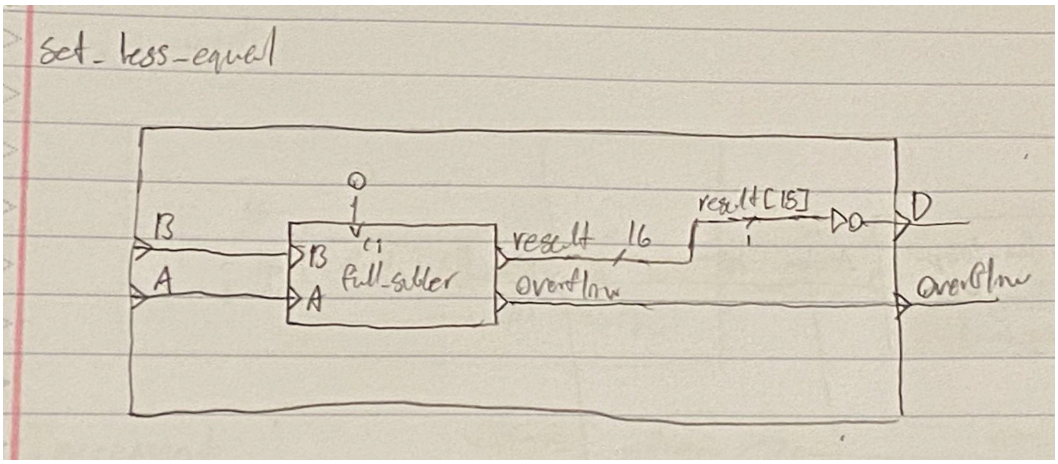
12. `module set_less_equal`

Outputs 1 if `A` $\leqslant$ `B`. The module computes the different `B - A` and checks the sign bit. A positive result satisfies this condition, so we negate the sign bit to get the output. **We added extra robustness to our implementation by adding an `Overflow` flag.** This is because there are some cases where `B - A` results in an overflow, thus an incorrect sign bit.

```
module set_less_equal(
    input [15:0] A,B,
    output D,
    output Overflow
);
    wire [15:0] diff_out;
```

```
        wire sign_out;
        wire co;
        fullsubber fs(.r1(B), .r2(A), .ci(0), .carry(co), .result(diff_out),
        .Overflow(Overflow));

        // fullsubber checks if B > A. Negate this to get A ⩽ B.
        not (D, diff_out[15]);
    endmodule
```

a. Set less equal schematic



Part 3: Register file

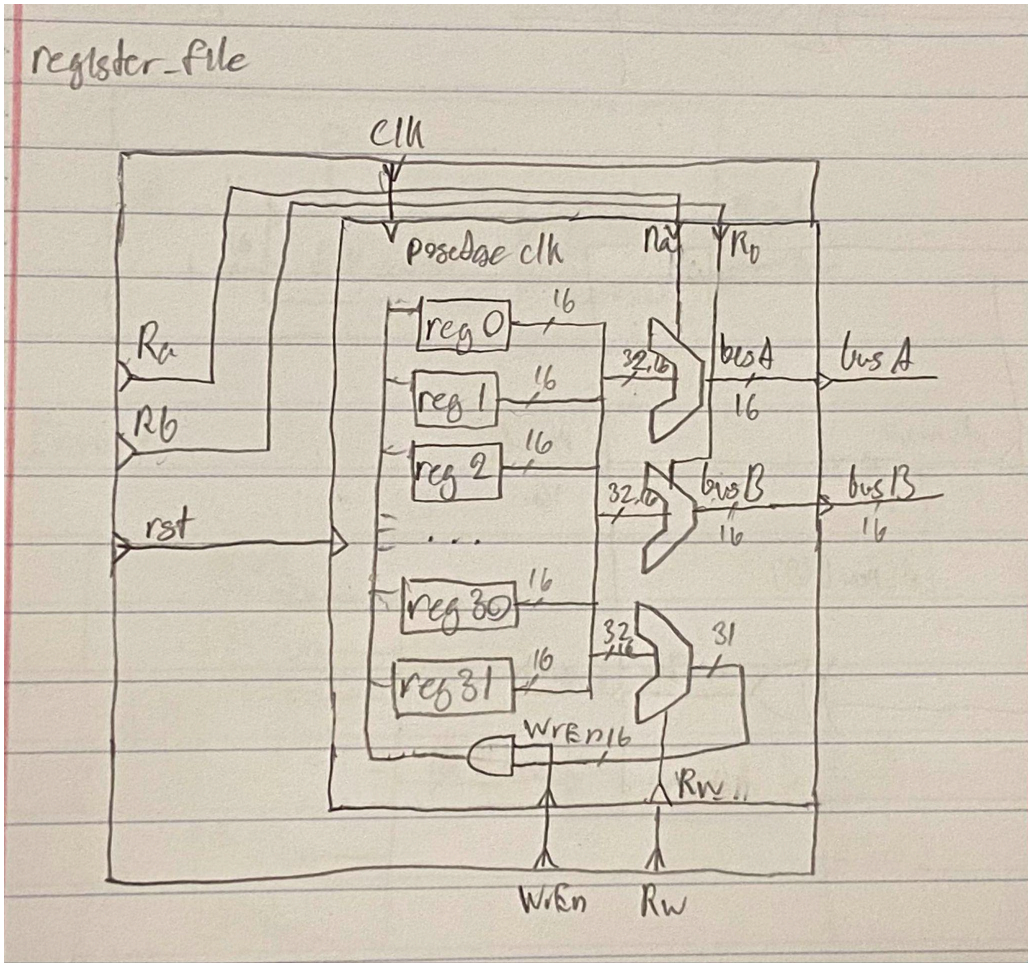
`module register_file`

Register file with 32 16-bit registers, two read ports, and one write port. The register file supports two concurrent read operations and one write operation. It follows that we can read and write to a register at the same time. The input `clk` allows for synchronous reset, write, and reads. Our implementation is structured such that we can only write to a register when `WrEn` is high.

```
module register_file(
    input wire clk,
    input wire rst,
    input wire [4:0] Ra,
    input wire [4:0] Rb,
    input wire [4:0] Rw,
    input wire [15:0] busW,
    input wire WrEn,
    output wire [15:0] busA,
    output wire [15:0] busB
);

    reg [15:0] registers [31:0]; // 32 16-bit registers
    assign busA = registers[Ra]; // Data in register Ra
    assign busB = registers[Rb]; // Data in register Rb

    // Write operation with synchronous reset
    always @(posedge clk) begin
        if (rst) begin
            // Clear all registers
```

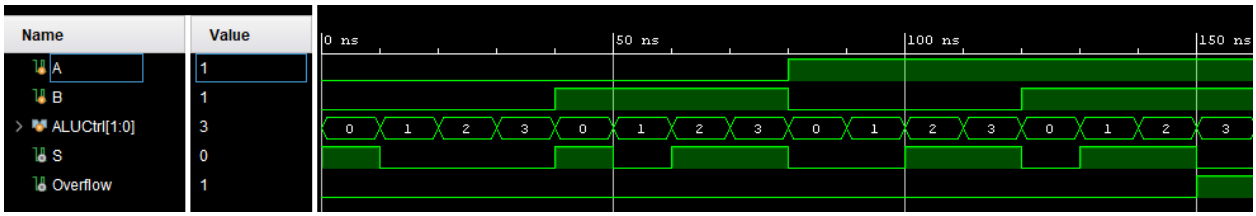
```
            registers[0] <= 16'b0;
            registers[1] <= 16'b0;
            registers[2] <= 16'b0;
            registers[3] <= 16'b0;
            registers[4] <= 16'b0;
            registers[5] <= 16'b0;
            registers[6] <= 16'b0;
            registers[7] <= 16'b0;
            registers[8] <= 16'b0;
            registers[9] <= 16'b0;
            registers[10] <= 16'b0;
            registers[11] <= 16'b0;
            registers[12] <= 16'b0;
            registers[13] <= 16'b0;
            registers[14] <= 16'b0;
            registers[15] <= 16'b0;
            registers[16] <= 16'b0;
            registers[17] <= 16'b0;
            registers[18] <= 16'b0;
            registers[19] <= 16'b0;
            registers[20] <= 16'b0;
            registers[21] <= 16'b0;
            registers[22] <= 16'b0;
            registers[23] <= 16'b0;
            registers[24] <= 16'b0;
            registers[25] <= 16'b0;
            registers[26] <= 16'b0;
            registers[27] <= 16'b0;
            registers[28] <= 16'b0;
            registers[29] <= 16'b0;
            registers[30] <= 16'b0;
            registers[31] <= 16'b0;
        end else if (WrEn) begin
            // Write operation
            registers[Rw] <= busW;
        end
    end
endmodule
```

1. register file schematic



Testing

1-bit ALU: `module ALU1`



This waveform demonstrates every ALU operation with every pair of A-B inputs (00, 01, 10, 11).

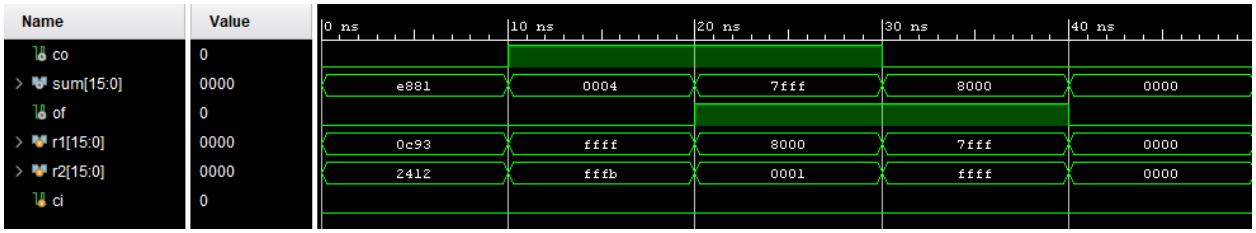
00	01	10	11
NOT 0 = 1 0 & 0 = 0 0 | 0 = 0 0 + 0 = 0	NOT 0 = 1 0 & 1 = 0 0 | 1 = 1 0 + 1 = 1	NOT 1 = 0 1 & 0 = 0 1 | 0 = 1 1 + 0 = 1	NOT 1 = 0 1 & 1 = 1 1 | 1 = 1 1 + 1 = 0, overflow = 1

```
module ALU1_tb;
```

```
    reg A;
    reg B;
    reg [1:0] ALUCtrl;
    wire S;
    wire Overflow;

    ALU1 my_alu(
       .A(A),
       .B(B),
       .S(S),
       .ALUCtrl(ALUCtrl),
       .Overflow(Overflow)
    );

    initial begin
        A = 0; B = 0; ALUCtrl = 0; #10;
        ALUCtrl = 1; #10;
        ALUCtrl = 2; #10;
        ALUCtrl = 3; #10;

        A = 0; B = 1; ALUCtrl = 0; #10;
        ALUCtrl = 1; #10;
        ALUCtrl = 2; #10;
        ALUCtrl = 3; #10;

        A = 1; B = 0; ALUCtrl = 0; #10;
        ALUCtrl = 1; #10;
        ALUCtrl = 2; #10;
        ALUCtrl = 3; #10;

        A = 1; B = 1; ALUCtrl = 0; #10;
        ALUCtrl = 1; #10;
        ALUCtrl = 2; #10;
        ALUCtrl = 3; #10;
        $finish;
    end
endmodule
```

12 ALU operations:

1. `module fullsubber`

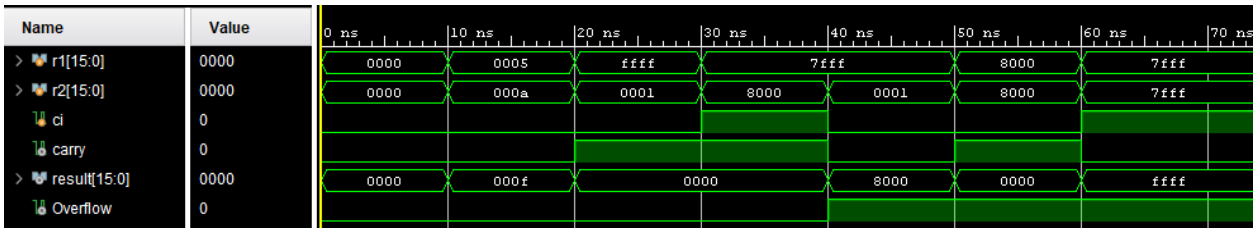


- Test 1: standard case, no overflow — 3219 - 9234 = -6015 **(e881)**
- Test 2: two negatives become positive — -1 - (-5) = 4 **(0004)**
- Test 3: underflow INT_MIN to INT_MAX — INT_MIN - 1 = INT_MAX **(7fff)**

- Test 4: overflow INT_MAX to INT_MIN — INT_MAX - (-1) = INT_MIN **(8000)**
  Test 5: no subtraction performed — 0 - 0 = 0 **(0000)**

```
module fullsubber_tb;
    wire co;
    wire [15:0] sum;
    wire of;
    reg [15:0] r1, r2;
    reg ci;

    fullsubber fa1(
        .r1(r1),
        .r2(r2),
        .ci(ci),
        .carry(co),
        .result(sum),
        .Overflow(of)
    );

    initial begin
        r1 = 3219; r2 = 9234; ci = 0; // 3219 - 9234 = -6015
        #10;
        r1 = -1; r2 = -5; ci = 0; // -1 - (-5) = 4
        #10;
        r1 = 16'h8000; r2 = 16'h0001; ci = 0;  // -32768 - 1 becomes 16'h7FFF
        #10;

        r1 = 16'h7fff; r2 = 16'hffff; ci = 0; // 32767 - (-1) becomes 16'h8000
        #10;
        r1 = 16'h0000; r2 = 16'h0000; ci = 0;  // 0 - 0 = 0
        #10;
        $finish;
     end
 endmodule
```

2. `module fulladder`

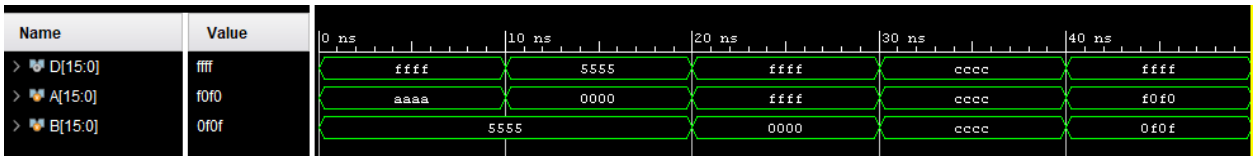


- Test 1: no addition performed — 0 + 0 = 0 **(0000)**
- Test 2: standard case, no overflow or carry — 5 + 10 = 15 **(000a)**
- Test 3: standard case, no overflow has carry — -1 + 1 = 0 **(0000)**
- Test 4: carry in and carry out — INT_MAX + INT_MIN+1 = 0 **(0000)**
- Test 5: overflow to INT_MIN — INT_MAX + 1 = INT_MIN **(8000)**
- Test 6: overflow to 0 — INT_MIN + INT_MIN = 0 **(0000)**
- Test 7: overflow with carry in — INT_MAX + INT_MAX+1 = -1 **(ffff)**

```
module tb_fulladder;
```

```
    reg [15:0] r1, r2;
    reg ci;
    wire carry;
    wire [15:0] result;
    wire Overflow;

    fulladder adder(
        .r1(r1),
        .r2(r2),
        .ci(ci),
        .carry(carry),
        .result(result),
        .Overflow(Overflow)
    );

    initial begin
        r1 = 16'd0; r2 = 16'd0; ci = 0;
        #10;
        r1 = 16'd5; r2 = 16'd10; ci = 0;
        #10;
        r1 = 16'd65535; r2 = 16'd1; ci = 0; // 0xFFFF + 0x0001
        #10;
        r1 = 16'd32767; r2 = 16'd32768; ci = 1; // INT_MAX + INT_MIN with
carry-in
        #10;
        r1 = 16'd32767; r2 = 16'd1; ci = 0; // INT_MAX + 1
        #10;
        r1 = 16'h8000; r2 = 16'h8000; ci = 0; // -32768 + -32768
        #10;
        r1 = 16'h7FFF; r2 = 16'h7FFF; ci = 1; // 32767 + 32767 + 1
        #10;
    end
endmodule
```

3. `module bitwise_or`

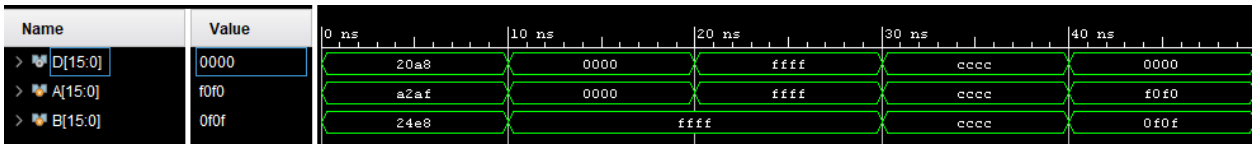


- Test 1: set to all 1s — aaaa | 5555 = **ffff**
- Test 2: OR with all 0s — 0000 | 5555 = **0x5555**
- Test 3: OR with all 0s — ffff | 0000 = **ffff**
- Test 4: OR with itself — cccc | cccc = **cccc**
- Test 5: OR with opposite bit values — f0f0 | 0f0f = **ffff**

```
module or_tb;
    wire [15:0] D;
    reg [15:0] A;
    reg [15:0] B;

    bitwise_or or_op(
        .D(D),
```

```
        .A(A),
        .B(B)
    );

    initial begin
        A = 16'b1010101010101010;
        B = 16'b0101010101010101;
        #10; // D = 16'b1111111111111111

        A = 16'b0000000000000000;
        B = 16'b0101010101010101;
        #10; // D = 16'b0101010101010101

        A = 16'b1111111111111111;
        B = 16'b0000000000000000;
        #10; // D = 16'b1111111111111111

        A = 16'b1100110011001100;
        B = 16'b1100110011001100;
        #10; // D = 16'b1100110011001100

        A = 16'b1111000011110000;
        B = 16'b0000111100001111;
        #10; // D = 16'b1111111111111111
        $finish;
    end
endmodule
```

4. `module bitwise_and`

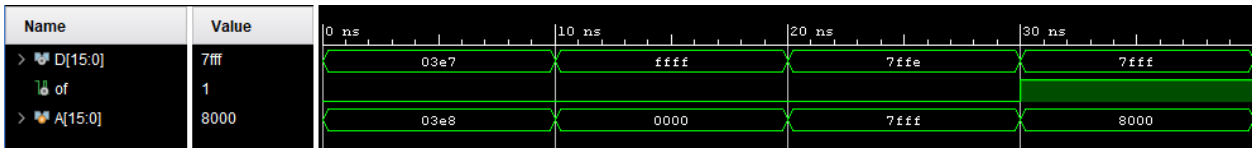


- Test 1: standard AND — a2af & 24e8 = **20a8**
- Test 2: AND with all 0s — 0000 & ffff = **0000**
- Test 3: AND with all 1s — ffff & ffff = **ffff**
- Test 4: AND with itself — cccc & cccc = **cccc**
- Test 5: AND with opposite bit values — f0f0 & 0f0f = **0000**

```
module and_tb;
    wire [15:0] D;
    reg [15:0] A;
    reg [15:0] B;

    bitwise_and and_op(
        .A(A),
        .B(B),
        .D(D)
      );

    initial begin
        A = 16'b1010001010101111;
```

```
        B = 16'b0010010011101000;
        #10; // D = 16'b0010000010101000

        A = 16'b0000000000000000;
        B = 16'b1111111111111111;
        #10; // D = 16'b0000000000000000

        A = 16'b1111111111111111;
        B = 16'b1111111111111111;
        #10; // D = 16'b1111111111111111

        A = 16'b1100110011001100;
        B = 16'b1100110011001100;
        #10; // D = 16'b1100110011001100

        A = 16'b1111000011110000;
        B = 16'b0000111100001111;
        #10; // D = 16'b0000000000000000
        $finish;
    end
endmodule
```

5. `module decrement`

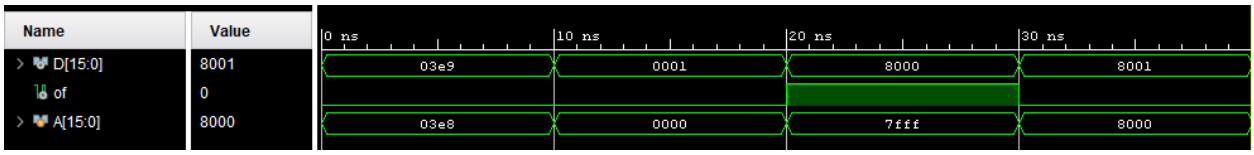


- Test 1: basic decrement — dec 1000 (03e8) = **03e7**
- Test 2: decrement 0 — dec 0 = -1 (**ffff**)
- Test 3: decrement INT_MAX — dec INT_MAX = **7ffe**
- Test 4: decrement INT_MIN; overflow — dec INT_MIN = INT_MAX **(7fff)**

```
module dec_tb;
    wire [15:0] D;
    wire of;
    reg [15:0] A;

    decrement dec_op(
        .A(A),
        .D(D),
        .Overflow(of)
    );

    initial begin
        A = 16'd1000;
        #10; // D = 999;
        A = 16'd0;
        #10;
        A = 16'h7FFF; // 32767
        #10; // D = 16'h8000 (-32768)
        A = 16'h8000;
        #10; // D = 16'7FFF (32767)
        $finish;
```

```
    end
endmodule
```

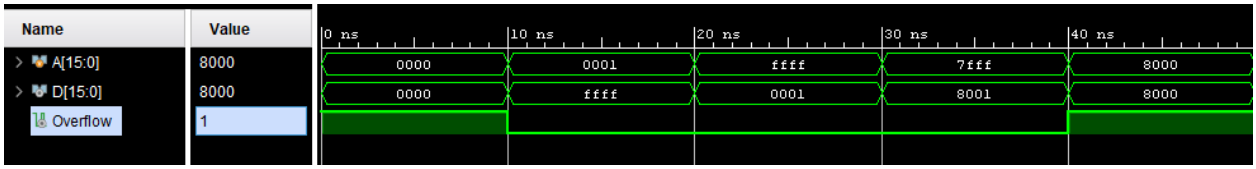
6. `module increment`



- Test 1: basic increment — inc 1000 (03e8) = **03e9**
- Test 2: increment 0 — inc 0 = **0001**
- Test 3: increment INT_MAX; overflow — inc INT_MAX = INT_MIN (**8000**)
- Test 4: increment INT_MIN — inc INT_MIN = **8001**

```
module inc_tb;
    wire [15:0] D;
    wire of;
    reg [15:0] A;

    increment inc_op(
        .A(A),
        .D(D),
        .Overflow(of)
    );

    initial begin
        A = 16'd1000;
        #10; // D = 1001;
        A = 16'd0;
        #10;
        A = 16'h7FFF; // 32767
        #10; // D = 16'h8000 (-32768)
        A = 16'h8000;
        #10; // D = 16'hFFF (-32767)
        $finish;
    end
endmodule
```

7. `module invert`

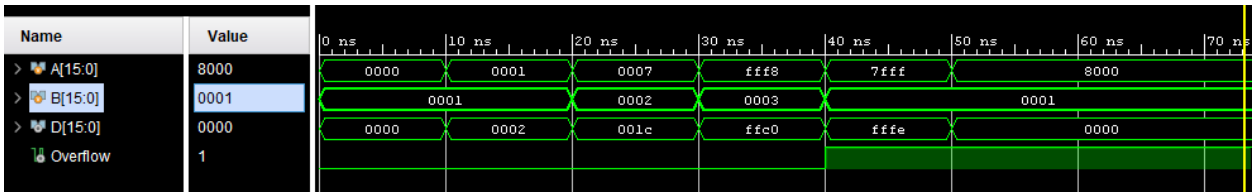


- Test 1: invert 0; overflow — inv 0 = 0
- Test 2: basic invert — inv 1 = -1 (**ffff**)
- Test 3: basic invert — inv -1 = 1 (**0001**)
- Test 4: invert INT_MAX — inv INT_MAX = INT_MIN + 1 (**8001**)
- Test 5: invert INT_MIN; overflow — inv INT_MIN = INT_MIN

```
module inv_tb;
    reg [15:0] A;
```

```
    wire [15:0] D;
    wire of;

    invert inv (
        .A(A),
        .D(D),
        .Overflow(of)
    );

    initial begin
        A = 16'b0; // Expected result: D = 0, Overflow = 0
        #10;
        A = 16'b0000_0000_0000_0001; // Expected result: D = -1, of = 0
        #10;
        A = 16'b1111_1111_1111_1111;  // Expected result: D = +1, of = 1
        #10;
        A = 16'b0111_1111_1111_1111; // Expected result: D = -32767, of = 1
        #10;
        A = 16'b1000_0000_0000_0000; // Expected result: D = -32768, of = 0
        #10;

        $finish;
    end
endmodule
```

8. `module arith_shift_left`

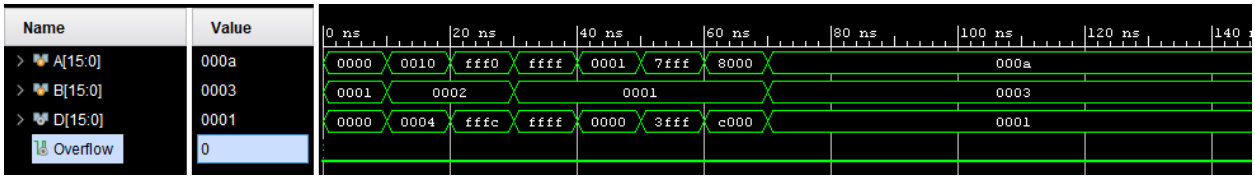


- Test 1: shift 0 by 1 bit; standard ALS — 0 <<< 1 = 0
- Test 2: shift 1 by 1 bit; standard ALS — 1 <<< 1 = 2
- Test 3: shift 7 by 2 bits; standard ALS — 7 <<< 2 = 28 (**001c**)
- Test 4: shift -8 by 3 bits; ALS negative number — -8 <<< 3 = -64 (**ffc0**)
- Test 5: shift INT_MAX by 1 bit; ALS INT_MAX, sign change corresponding to overflow — INT_MAX <<< 1 = -2 (**fffe**)
- Test 6: shift INT_MIN by 1 bit; ALS INT_MIN, sign change corresponding to overflow — INT_MIN <<< 1 = 0

```
module arith_left_tb;
    reg [15:0] A;
    reg [15:0] B;
    wire [15:0] D;
    wire Overflow;

    arith_shift_left arith_left (
        .A(A),
        .B(B),
        .D(D),
        .Overflow(Overflow)
```

```
    );

    initial begin
        A = 16'd0;
        B = 16'd1;
        #10; // Expected result: D = 0, Overflow = 0
        A = 16'd1;
        B = 16'd1;
        #10; // Expected result: D = 2, Overflow = 0
        A = 16'd7;
        B = 16'd2;
        #10; // Expected result: D = 28, Overflow = 0
        A = -8;
        B = 16'd3;
        #10; // Expected result: D = -64, Overflow = 0
        A = 16'd32767;
        B = 16'd1;
        #10; // Expected result: Overflow due to exceeding 16-bit limit
        A = -32768;
        B = 16'd1;
        #10; // Expected result: Overflow = 1
    end
endmodule
```

9. `module arith_shift_right`

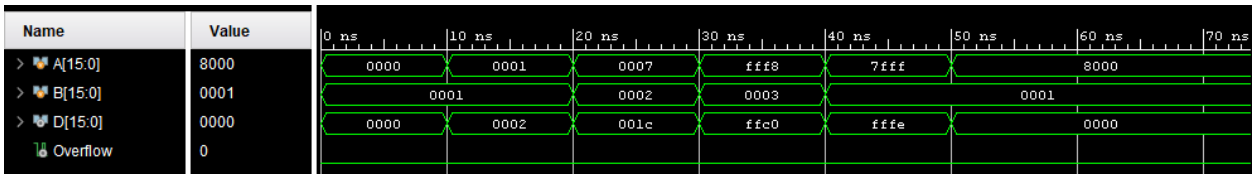


- Test 1: shift 0 by 1 bit; standard ARS 0 >>> 1 = 0
- Test 2: shift 16 by 2 bits; standard ARS 16 >>> 2 = 4
- Test 3: shift -16 by 2 bits; standard ARS with negative number, sign preserved
  -16 >>> 2 = -4 (**fffc**)
- Test 4: shift -1 by 2 bits; shift -1, sign preserved -1 >>> 2 = -1 (**ffff**)
- Test 5: shift 1 by 1 bit; standard ARS 1 >>> 1 = 0
- Test 6: shift INT_MAX by 1; standard ARS INT_MAX >>>1 = 16383 (**3fff**)
- Test 7: shift INT_MIN by 1; shift INT_MIN, sign preserved
  INT_MIN >>> =-16384 (**c000**)
- Test 8: shift 10 by 3 bits; standard ARS 10 >>> 3 = 0

```
module arith_right_tb;
    reg [15:0] A;
    reg [15:0] B;
    wire [15:0] D;
    wire Overflow;

    arith_shift_right arith_right (
        .A(A),
        .B(B),
        .D(D),
```

```
        .Overflow(Overflow)
    );

    initial begin
        A = 16'd0;
        B = 16'd1;
        #10; // Expected result: D = 0, Overflow = 0
        A = 16'd16;
        B = 16'd2;
        #10; // Expected result: D = 4, Overflow = 0
        A = -16;
        B = 16'd2;
        #10; // Expected result: D = -4, Overflow = 0
        A = -1;
        B = 16'd1;
        #10; // Expected result: D = -1, Overflow = 0
        A = 16'd1;
        B = 16'd1;
        #10; // Expected result: D = 0, Overflow = 0
        A = 16'd32767;
        B = 16'd1;
        #10; // Expected result: No overflow, D = 16383
        A = -32768;
        B = 16'd1;
        #10; // Expected result: No overflow, D = -16384
        A = 16'd10;
        B = 16'd3;
        #10; // Expected result: D = 1, Overflow = 0
    end
endmodule
```

10. `module log_shift_left`



- Test 1: shift 0 by 1; standard LLS — 0 << 1 = 0
- Test 2: shift 1 by 1; standard LLS — 1 << 1 = 1
- Test 3: shift 7 by 2; standard LLS — 7 << 2 = 28 (**001c**)
- Test 4: shift -8 by 3; standard LLS with negative number — -8 << 3 = -64 (**ffc0**)
- Test 5: shift INT_MAX by 1 — INT_MAX << 1 = -2 (**fffe**)
- Test 6: shift INT_MIN by 1 — INT_MIN << 1 = 0

```
module log_left_tb;
    reg [15:0] A;
    reg [15:0] B;
    wire [15:0] D;
    wire Overflow;

    log_shift_left log_left (
        .A(A),
```

```
        .B(B),
        .D(D),
        .Overflow(Overflow)
    );

    initial begin
        A = 16'd0;
        B = 16'd1;
        #10; // Expected result: D = 0, Overflow = 0
        A = 16'd1;
        B = 16'd1;
        #10; // Expected result: D = 2, Overflow = 0
        A = 16'd7;
        B = 16'd2;
        #10; // Expected result: D = 28, Overflow = 0
        A = -8;
        B = 16'd3;
        #10; // Expected result: D = -64, Overflow = 0
        A = 16'd32767;
        B = 16'd1;
        #10; // Expected result: Overflow due to exceeding 16-bit limit
        A = -32768;
        B = 16'd1;
        #10; // Expected result: Overflow = 1
    end
endmodule
```

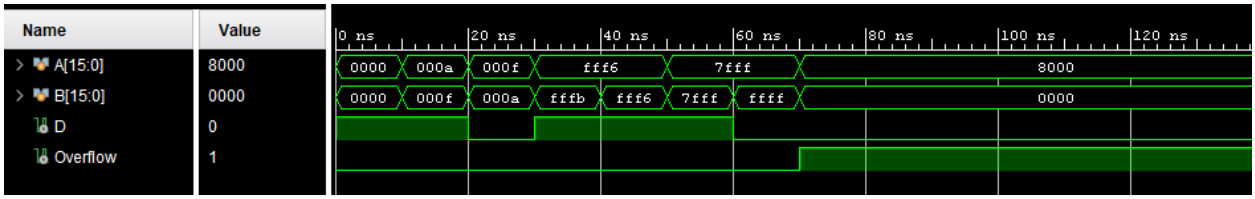
11. `module log_shift_right`

- Test 1: shift 0 by 1 bit; standard LRS 0 >> 1 = 0
- Test 2: shift 8 by 1 bit; standard LRS 8 >> 1 = 4
- Test 3: shift 16 by 2; standard LRS 16 >> 2 = 4
- Test 4: shift INT_MIN by 3; shift INT_MIN, no preservation of sign
  INT_MIN >> 3 = 4096 (**1000**)
- Test 5: shift -1 by 4; shift -1, no preservation of sign -1 >> 4 = 4095 (**0fff**)
- Test 6: shift 1024 by 0; no shift 1024 >> 0 = 0
- Test 7: shift 1 by 15; shift full width - 1 1 >> 15 = 0
- Test 8: shift INT_MAX by 2; standard ARS INT_MAX >> 2 = 8191 (**1fff**)

```
module log_right_tb;
    reg [15:0] A;
    reg [15:0] B;
    wire [15:0] D;
    wire Overflow;

    log_shift_right log_right (
        .A(A),
```

```
        .B(B),
        .D(D),
        .Overflow(Overflow)
    );

    initial begin
        A = 16'd0;
        B = 16'd1;
        #10; // Expected result: D = 0
        A = 16'd8;  // Binary: 0000_0000_0000_1000
        B = 16'd1;
        #10; // Expected result: Binary: 0000_0000_0000_0100
        A = 16'd16; // Binary: 0000_0000_0001_0000
        B = 16'd2;
        #10; // Expected result: Binary: 0000_0000_0000_0100
        A = 16'd32768; // Binary: 1000_0000_0000_0000
        B = 16'd3;
        #10; // Expected result: Binary: 0000_1000_0000_0000
        A = 16'd65535; // Binary: 1111_1111_1111_1111
        B = 16'd4;
        #10; // Expected result: Binary: 0000_1111_1111_1111
        A = 16'd1024; // Binary: 0000_0100_0000_0000
        B = 16'd0;
        #10; // Expected result: D = 1024
        A = 16'd1; // Binary: 0000_0000_0000_0001
        B = 16'd15;
        #10; // Expected result: D = 0
        A = 16'd32767; // Binary: 0111_1111_1111_1111
        B = 16'd2;
        #10; // Expected result: Binary: 0001_1111_1111_1111
    end
endmodule
```

12. `module set_less_equal`

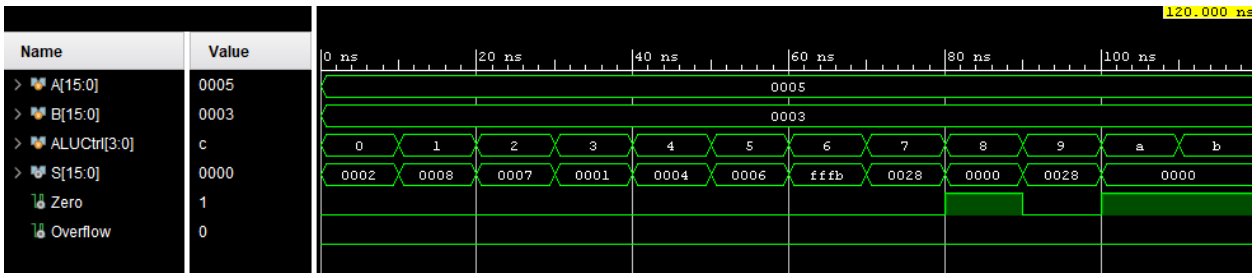


- Test 1: A = 0, B = 0; standard SLE — 0 <= 0 = 1
- Test 2: A = 10, B = 15; standard SLE — 10 <= 15 = 1
- Test 3: A = 15, B = 10; standard SLE — 15 is not <= 10 = 0
- Test 4: A = -10, B = -5; standard SLE with negative numbers — -10 <= -5 = 1
- Test 5: A = -10, B = -10; standard SLE with negative numbers — -10 <= -10 = 1
- Test 6: A = INT_MAX, B = INT_MAX; standard SLE with INT_MAX
  INT_MAX <= INT_MAX = 1
- Test 7: A = INT_MAX, B = -1; compare INT_MAX with negative number
  INT_MAX is not <= -1 = 0
- Test 8: A = INT_MIN, B = 0; compare INT_MIN with positive number, overflow

```
module sle_tb;
```

```
    reg [15:0] A, B;
    wire D;
    wire Overflow;

    set_less_equal sle (
        .A(A),
        .B(B),
        .D(D),
        .Overflow(Overflow)
    );

    initial begin
        A = 16'd0;
        B = 16'd0;
        #10; // Expected D = 1 (A ⩽ B)
        A = 16'd10;
        B = 16'd15;
        #10; // Expected D = 1 (A ⩽ B)
        A = 16'd15;
        B = 16'd10;
        #10; // Expected D = 0 (A > B)
        A = 16'hFFF6; // -10 in 16-bit 2's complement
        B = 16'hFFFB; // -5 in 16-bit 2's complement
        #10; // Expected D = 1 (A ⩽ B)
        A = 16'hFFF6; // -10 in 16-bit 2's complement
        B = 16'hFFF6; // -10 in 16-bit 2's complement
        #10; // Expected D = 1 (A ⩽ B)
        A = 16'h7FFF; // 32767
        B = 16'h7FFF; // 32767
        #10; // Expected D = 1 (A ⩽ B)
        A = 16'h7FFF; // 32767
        B = 16'hFFFF; // -1
        #10;  // Expected D = 0 (A > B)
        A = 16'h8000; // -32768
        B = 16'd0;
        #10;  // Expected D = 1 (A ⩽ B)
    end
endmodule
```

16-bit ALU: `module ALU16`



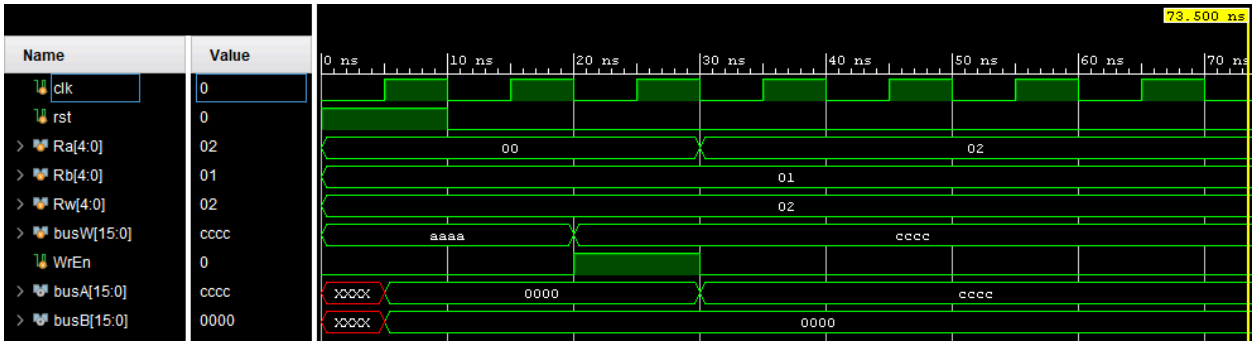
All tests are performed with A = 5 and B = 3. This testbench primarily shows the correct behavior with ALUCtrl. Extensive test casing for each operation is in the previous section.

- Test 1: SUB 5 - 3 = 2

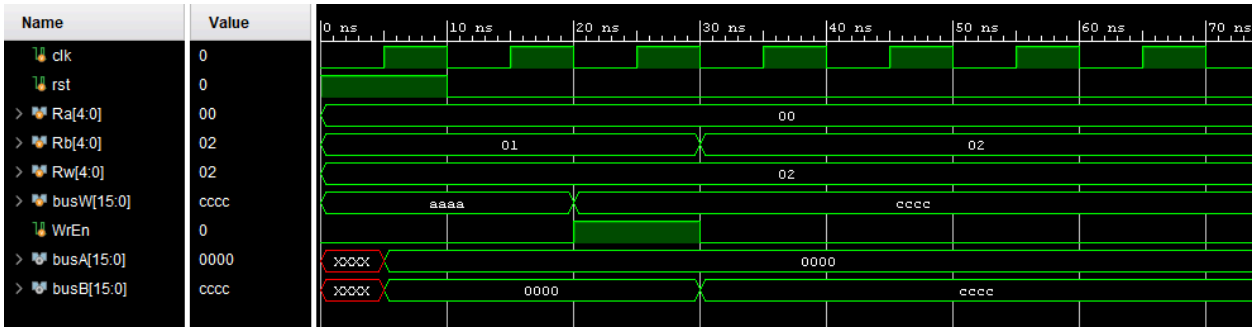
- Test 2: ADD 5 + 3 = 8
- Test 3: OR 5 | 3 = 7
- Test 4: AND 5 & 3 = 1
- Test 5: DEC dec 5 = 4
- Test 6: INC inc 5 = 6
- Test 7: INV inv 5 = -5 (fffb)
- Test 8: Arith Left Shift 5 <<< 3 = 40 (0028)
- Test 9: Arith Right Shift; Zero = 1 5 >>> 3 = 0
- Test 10: Log Left Shift 5 << 3 = 40 (0028)
- Test 11: Log Right Shift , Zero = 1 5 >> 3 = 0
- Test 12: SLE, Zero = 1 5 is not <= 3 = 0

```
module ALU16_tb;
    reg [15:0] A;
    reg [15:0] B;
    reg [3:0] ALUCtrl;
    wire [15:0] S; // Output
    wire Zero;
    wire Overflow;

    ALU16 alu16(
        .A(A),
        .B(B),
        .ALUCtrl(ALUCtrl),
        .S(S),
        .Zero(Zero),
        .Overflow(Overflow)
    );

    initial begin
        A = 16'd5;
        B = 16'd3;
        for (ALUCtrl = 0; ALUCtrl ⩽ 11; ALUCtrl = ALUCtrl + 1) begin
            #10;
        end
        $finish;
    end
endmodule
```

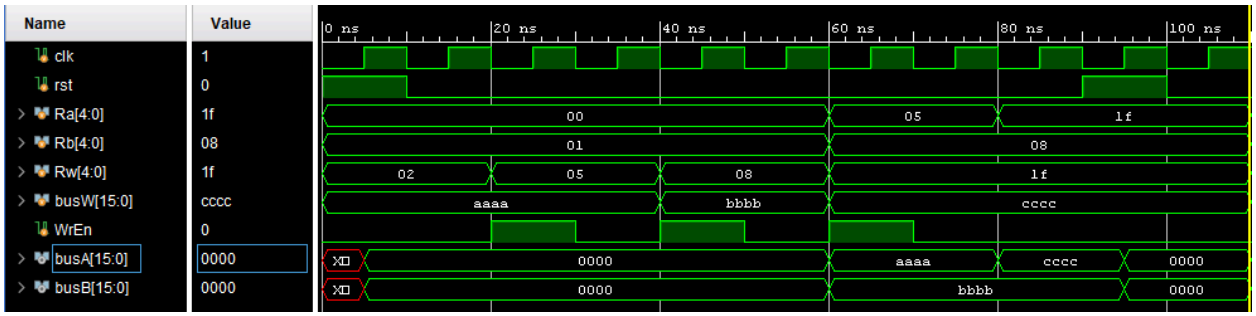
Register file: `module register_file`



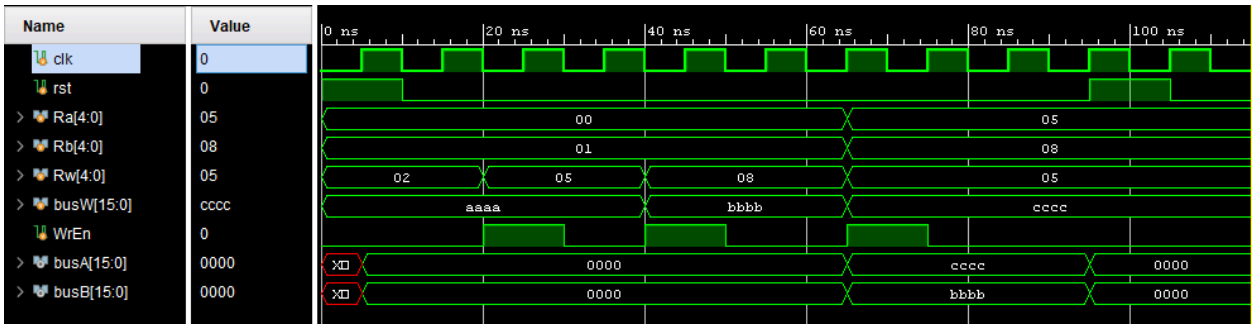
1. This waveform demonstrates the basic operations of the register file. Register A is set to 0, Register B is set to 1, and the file will write “cccc” to Register 2. Reading and writing do not occur concurrently for this waveform, as Register A only switches to 2 after the write occurs. Once this happens, Register A will display the “cccc” that was written to 2.



2. This waveform demonstrates that the output bus will update at the same time as the write, if one of the read registers points to the register being written to. Register A is set to 0, Register B is set to 2, and the file will write “cccc” to Register 2. Once the WriteEnable switches from on to off, the change will automatically propagate to Register 2. Thus, the busB line will read “cccc”. Note that we cannot write to register Rw unless WriteEnable is high.



3. This waveform demonstrates that the register file can handle concurrent reads and writes to different registers. “aaaa” is written to Register 2 and 5, and “bbbb” is written to Register 8. Then, Register A will read 5 and Register B will read 8, outputting “aaaa” and “bbbb” respectively. At the same time, the file will write “cccc” to Register 31. After two clock cycles, Register A will switch to 31 and will output “cccc”.



4. This waveform demonstrates that the register file can concurrently handle two reads and a write, where one of the registers read is the same one that it is written to. “aaaa” is written to Register 2

and 5, and “bbbb” is written to Register 8. Then, both Register A and the write register switches to 5, and the WriteEnable bit is set. The file will write “cccc” to Register 5, and at the same instant, Register A will output “cccc”.

```
module register_file_tb;
    reg clk;
    reg rst;
    reg [4:0] Ra, Rb, Rw;
    reg [15:0] busW;
    reg WrEn;
    wire [15:0] busA, busB;
    register_file uut (
        .clk(clk),
        .rst(rst),
        .Ra(Ra),
        .Rb(Rb),
        .Rw(Rw),
        .busW(busW),
        .WrEn(WrEn),
        .busA(busA),
        .busB(busB)
    );

    always begin
        #5 clk = ~clk; // Toggle clock every 5 time units
    end

    initial begin
        // Initialize signals
        clk = 0;
        rst = 0;
        Ra = 5'b00000;
        Rb = 5'b00001;
        Rw = 5'b00010;
        busW = 16'b1010101010101010;
        WrEn = 0;

        // Apply reset
        rst = 1;
        #10;
        rst = 0;
        #10;

        // Write
        /*Rw = 5'b00010; // Write address
        busW = 16'b1100110011001100; // Data to write
        WrEn = 1; // Enable write
        #10;
        WrEn = 0; // Disable write

        // Read
        Rb = 5'b00010; // Read address A; should be 1100110011001100*/
```

```
        // COMPLEX CASE //
        // Write to reg A
        Rw = 5'b00101;
        busW = 16'haaaa;
        WrEn = 1;
        #10;
        WrEn = 0;

        #10;

        // Write to reg B
        Rw = 5'b01000;
        busW = 16'hbbbb;
        WrEn = 1;
        #10;
        WrEn = 0;

        #15; // Changed delay from 10 to 15 because we want to sync the write enable with the positive edge of the clock.

        // 2 reads and a write
        Ra = 5'b00101;
        Rb = 5'b01000;
        // Rw = 5'b11111; // This is for two reads and one write with three separate registers
        Rw = 5'b00101; // This is for two reads and one write where we are reading and writing to Register 00101
        busW = 16'hcccc;
        WrEn = 1;
        #10;
        WrEn = 0;
        #10;

        // Check the value of reg we wrote to
        // Ra = 5'b11111;
        #10;

        // Reset
        rst = 1;
        #10;
        rst = 0;
        #10;
        $finish;
    end
endmodule
```

Challenges

- One challenge we faced was implementing the majority of our 16-bit ALU in structural Verilog. I.e., we replaced if/else statements with a custom XOR function, along with adding some clever gate logic to check for a certain condition.

- Another difficulty we faced was having to create other modules apart from the ones defined in the lab specification. We created a custom `XOR gate`, `bitwise XOR`, and `zero checker`. XOR was useful for determining equality and the zero checker largely helped in setting the `Zero` flag.
- Our subtraction module originally had an error where `0 - 0` created a fake overflow. This was due to our implementation of deriving a full subtractor from our full adder. We were computing `0 + inv(0)`, but inverting in signed two's complement is `inv(0) = bitwiseNot(0) + 1`. So `0 - 0 = 0` would be computationally correct, but the `Overflow` bit would be set to high. We fixed this by adding a zero checker for both inputs.
- Another challenge we experienced was writing syntactically correct code that could not be simulated in Vivado. In particular, we originally had the notion of `` `wire [15:0] D [15:0];` `` – essentially an array of length 16 that stores 16-bit strings. This stores the result for each ALU operation (12 ALU operations, but the array is extended to 16 so it can fit into a 16-to-1 mux). However, parsing this and passing `D[0], …, D[15]` to a 16-to-1 mux raised an error in Vivado, so we flattened it into individual 16-bit wires (`D_sub, D_add,` etc.). It is possible that this issue has been solved in newer versions of Vivado.
- We originally calculated the set on less than or equal operation by taking the result of `A - B` and checking the sign bit. `1` MSB and `0` MSB (for all zeroes) satisfies this condition, but we struggled to distinguish between `0` MSB when `A = B` and `0` MSB when `A > B`. Alternatively, we computed `B - A` so that a `0` MSB satisfies this condition for both `less than` and `equal`.

Questions

1. Structural Verilog mimics how hardware components are physically connected by using wires with wire assignments and combinatorial primitives like AND, OR, NOT, etc. Behavioral Verilog is used for top-level control, i.e. wrappers and testing, with behavioral assignments like =, <=, if-else, etc.

   Example 1: 4-to-1 mux in structural Verilog:

```
module m41_struc(

    output [15:0] Y,
    input [15:0] D0, D1, D2, D3,
    input S1, S0
);
    wire [15:0] T1, T2;                 // Direct connections with wires
    m21 mux1(T1, D0, D1, S0);           // Same m21 module used in Lab 1
    m21 mux2(T2, D2, D3, S0);                   // Implemented using structural

    m21 mux3(Y, T1, T2, S1);
endmodule
```

   Example 2: 4-to-1 mux in behavioral Verilog:

```
module m41_behav(
    output reg Y,                       // Notion of registers
    input [3:0] D,
    input [1:0] S,
);
    always @(*) begin                   // Monitor changes in signal
```

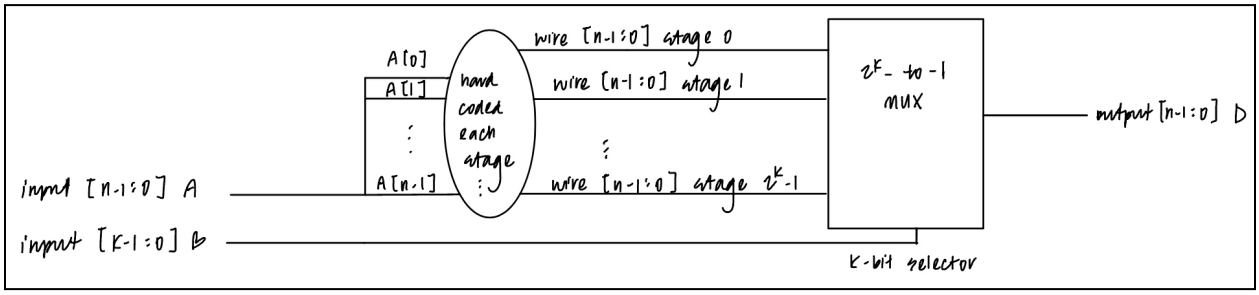
```
        case(S)
            2'b00: Y = D[0];            // Behavioral assignments
            2'b01: Y = D[1];
            2'b10: Y = D[2];
            2'b11: Y = D[3];
        endcase
    end
endmodule
```

2. The difference between asynchronous and synchronous multiplexers is when the output updates. Asynchronous multiplexer updates output immediately (`always @(*)`) whenever the inputs or select signal change. Synchronous multiplexer updates output on a clock edge (`always @(posedge clk)`) and uses flip-flops (<=) to store the value. To create a synchronous multiplexer in behavioral Verilog, we would add a clock input and modify the code such that `Y` updates only on a clock edge:

```
module m41_behav_sync(
    …
    input clk
);
    always @(posedge clk) begin
        case (S)
            2'b00: Y <= D[0];
            2'b01: Y <= D[1];
            2'b10: Y <= D[2];
            2'b11: Y <= D[3];
        endcase
    end
endmodule
```

3. Logical shifting fills in 0s regardless of sign and arithmetic shifting preserves the sign when shifting right.
4. To create a generalized n-bit arithmetic shifter with a k-bit shift amount, we can create 2^k n-bit wires that hold the shifted output for each stage (0 to 2^k-1). Then we would build a 2^k-to-1 multiplexer that selects the correct shifted output using a k-bit selector. The difference between implementing a right or left shifter is what value each intermediate n-bit wire holds.

   Simple diagram:



Responsibilities

105817312

- Lab: 6 ALU operations and testbenches, 1-bit ALU and testbench

- Report: Introduction, implementation, questions, challenges

905699244

- Lab: 6 ALU operations and testbenches, register file and testbench
- Report: Schematics, implementation, testing