Design and Implementation of a RISC-like Microcontroller

Ayush Yadav

November 4, 2025

Contents

Listings

List of Figures

1 Introduction

This project presents the design and implementation of a custom 32-bit RISC-style microcontroller developed using **Verilog** in the **Vivado** environment. The microcontroller features a simplified yet versatile instruction set architecture (ISA) designed to balance computational flexibility with hardware efficiency. It incorporates a rich set of binary and arithmetic operations, flexible operand encoding including immediate values, and a combination of register-to-register and memory access instructions. The design prioritizes modularity and hardware resource reuse, evidenced by shared functional units for arithmetic and comparison operations. To facilitate programming and testing, a custom Python assembler can be developed that translates assembly code written in the ISA into hexadecimal instruction files (.hex), used to load programs directly into the microcontroller's instruction cache.

2 Project Overview

The microcontroller consists of 16 general-purpose 32-bit registers and supports a comprehensive instruction set, including binary logic operations, arithmetic computations (add, subtract, multiply, divide), bitwise shifts and rotations, and comparison instructions that update flag registers for conditional branching. It features a 256-instruction cache (can be extended via `ramDepth` parameter) that functions as program memory (RAM), enabling efficient on-chip instruction storage without reliance on external memory.

Operands may be specified either as register values, immediate 16-bit constants, or pairs of 8-bit values embedded within instructions. Multiplication and division instructions yield 64-bit results split across two registers to maintain precision. The control flow supports a wide range of conditional jumps, governed by status flags updated during arithmetic and logical operations.

The entire system is implemented in modular Verilog, supporting straightforward expansion and verification. One of the key things is to use gate level modeling wherever possible to gain a deeper understanding of digital design principles.

- Architecture type: RISC-like (32 bit)
- Data width: 32 bits
- Register count: 16 general-purpose registers
- Instruction set: Custom minimal ISA
- Modules: ALU, Register File, Control Unit, Program Counter, Memory
- Tools used: Verilog, Vivado, VSCode

3 Design Specifications

3.1 Instruction Set Architecture (ISA)

The 16 general-purpose registers are named `R0` to `R15`. The program counter (`PC`) is a separate register and is modified using control flow instructions such as `JUMP`. The micro-controller does not support segmentation, stack operations, or interrupts.

- Binary Operations: `AND, OR, XOR, NAND, NOR, XNOR, NOT`
- Bitwise Operations: `SHL` (shift left), `SHRL` (shift right logical), `SHRA` (shift right arithmetic), `ROL` (rotate left), `ROR` (rotate right)
- Arithmetic Operations: signed operations `ADD, SUB, MUL, DIV` and unsigned operations `ADDU, SUBU, MULU, DIVU` and also `INC, DEC, NEGATE`
- Data Movement: `MOV` (register to register), `WRITEU, WRITEL` (immediate to register)
- Memory Access: `LOAD` (load from memory to register), `STORE` (store from register to memory)
- Comparison Operations: `CMP` (sets flags based on comparison)
- Control Flow: `JUMP, JZ, JNZ, JLE, JL, JGE, JG, JE, JNE` (conditional and unconditional jumps)

A few of these instructions `INC, DEC, MOV` do not exist on machine code level the assembler uses other instructions to implement these. For example, `INC R1` is implemented as `ADD R1 R1 1`.

All instructions are 32 bits wide and are encoded as follows:

- Opcode: 3 bits
- Function Code: 3 bits (to specify the exact operation within a category)
- input type: 2 bits (to specify whether operands are registers, immediate values, or a combination)
- Destination Register: 4 bits
- Source Register 1: 4 bits
- Source Register 2 / Immediate Value: 16 bits (can be two 8-bit values)

The user only needs to write the instruction and the assembler determines the opcode, function code and input type based on the instruction and operands.

3.2 Assembly Instructions

The assembly codes are written in a simple text format, one instruction per line. The keywords are seperated via spaces, no need of commas (`,`). Comments can be added using the semicolon (`;`). Labels can be defined for jump instructions by adding a colon (`:`) after the label name.

3.2.1 Bitwise Logical Operations

`AND, OR, XOR, NAND, NOR, XNOR`

These instructions perform bitwise logical operations between two operands and store the result in a destination register. Operands may include:

- Two registers
- One register and a 16-bit immediate value
- Two 8-bit immediate values

Syntax

```
<Operation> <dest_reg> <src_reg_1> <src_reg_2 | 16bit | 8bit 8bit>
```

Notes

- For two 8-bit immediate values, provide both values separated by a space.
- In this case, `<src_reg_1>` is omitted.

Listing 1: Logical Bitwise Operations

```
1 AND   R3  R1  R2      ; R3 = R1 AND R2
2 OR    R4  R1  15f     ; R4 = R1 OR 0x15F
3 NAND  R5  2a  R11     ; R5 = 0x2A NAND R11
4 XOR   R6  12  34      ; R6 = 0x12 XOR 0x34
```

3.2.2 Shift and Rotate Operations

`SHL, SHRL, SHRA, ROL, ROR`

These instructions perform bitwise shift or rotate operations. The first operand is the value to be shifted or rotated, and the second operand specifies the amount. Operands may include:

- Two registers
- One register and a 16-bit immediate value
- Two 8-bit immediate values

Syntax

```
<Operation> <dest_reg> <src_reg_1> <src_reg_2 | 16bit | 8bit 8bit>
```

Notes

- For two 8-bit immediate values, provide both values separated by a space.
- In this case, `<src_reg_1>` is omitted.
- Shift/rotate amounts are computed modulo 32.

Listing 2: Shift and Rotate Operations

```
SHL   R3  R1  R2     ; R3 = R1 << R2
SHRL  R4  R1  15f    ; R4 = R1 logical right shift by 0x15F
SHRA  R5  2a  R11    ; R5 = 0x2A arithmetic right shift by R11
ROL   R6  12  34     ; R6 = 0x12 rotated left by 0x34
```

3.2.3 Single input binary operations

`NOT, NEGATE`

These instructions perform `NOT` or `NEGATE` operation on a single operand and store the result in a destination register. Operands may include:

- Register
- 16-bit immediate value
- 8-bit immediate value

Syntax

```
<Operation> <dest_reg> <src_reg_1 | 16bit | 8bit>
```

Notes

- Here if the operands value is immediate and can be contained in 8 bits then it is not possible for the assembler to know if it is 8 bit or 16 bit so it will always consider it as 8bit unless zeroes are added in front of it to make it 16 bit.
- For example, `NOT R1 0F` will be considered as 8 bit immediate but `NOT R1 00F` will be considered as 16 bit immediate.

Listing 3: Single Input Binary Operations

```
NEGATE R1 R1   ; R1 = -R1
NOT    R2 0F   ; R2 = NOT 0x0F here 0x0F is considered as 8 bit
NOT    R3 00F  ; R3 = NOT 0x00F here 0x00F is considered as 16
    bit
```

3.2.4 Arithmetic Operations

`ADD, SUB, MUL, DIV, ADDU, SUBU, MULU, DIVU`

These instructions perform arithmetic operations between two operands and store the result in a destination register. Operands may include:

- Two registers
- One register and a 16-bit immediate value
- Two 8-bit immediate values

Syntax

```
<Operation> <dest_reg> <src_reg_1> <src_reg_2 | 16bit | 8bit 8bit>
```

Notes

- For two 8-bit immediate values, provide both values separated by a space.
- In this case, `<src_reg_1>` is omitted.
- Shift/rotate amounts are computed modulo 32.

Listing 4: Arithmetic Operations

```
1 ADD   R3  R1  R2      ; R3 = R1 + R2
2 SUB   R4  R1  15f     ; R4 = R1 - 0x15F
3 MUL   R5  2a  R11     ; R5 = 0x2A * R11
4 DIV   R6  12  34      ; R6 = 0x12 / 0x34
5 ADDU  R7  R1  R2      ; R7 = R1 +  R2 (unsigned)
6 SUBU  R8  R1  15f     ; R8 = R1 - 0x15F (unsigned)
7 MULU  R9  2a  R11     ; R9 = 0x2A * R11 (unsigned)
8 DIVU  R10 12  34      ; R10 = 0x12 / 0x34 (unsigned)
```

3.2.5 Data Movement Instructions

`MOV, WRITEU, WRITEL`

These instructions move data between registers, or load immediate values into registers. Operands may include:

- Two registers: to move data between
- One register and a 16-bit immediate value: to load immediate value into register

Syntax

```
<Operation> <dest_reg> <src_reg_1> <src_reg_2 | 16bit>
```

Notes

- in case of `MOV` both operands are registers `<src_reg_1>` and `<src_reg_2>`
- in case of `WRITEU` and `WRITEL` the second operand is a 16 bit immediate value and `<src_reg_1>` is omitted.

Listing 5: Data Movement Instructions

```
MOV    R1   R2       ; R1 = R2 (equivalent to ADD R1 R2 0)
WRITEU R3   15f      ; R3 = 0x15F (upper 16 bits set to 0x015F
    while the lower 16bit retain their previous value)
WRITEL R4   2a       ; R4 = 0x2A0000 (lower 16 bits set to 0x002A
    while the upper 16bit retain their previous value)
```

3.2.6 Memory Access Instructions

`LOAD, STORE`

These instructions facilitate data transfer between registers and memory. `LOAD` retrieves data from a specified memory address into a register, while `STORE` writes data from a register to a specified memory address. Operands may include:

- Two registers: one for the data and one for the memory address
- One register and a 16-bit immediate value: to specify the memory address directly

Syntax

`<Operation> <dest_reg> <src_reg_1> <src_reg_2 | 16bit>`

Notes

- in case of `LOAD` the first operand is the destination register and the second operand is the memory address (register or immediate)
- in case of `STORE` the first operand is the source register 1 and the second operand is the memory address (register or immediate)
- Thus in any case one of the registers is omitted, making the instructions have only two operands.

Listing 6: Data Movement Instructions

```
LOAD   R1   R2       ; R1 = MEM[R2]
LOAD   R2   15f      ; R2 = MEM[0x15F]
STORE  R3   15f      ; MEM[0x15F] = R3
STORE  R4   R5       ; MEM[R5] = R4
```

3.2.7 Compare Instruction

`CMP`

This instruction compares two values and stores the result in flag register. Which are then used by the conditional jump instructions. Operands may include:

- Two registers
- One register and a 16-bit immediate value
- Two 8-bit immediate values

Syntax

```
<Operation> <src_reg_1> <src_reg_2 | 16bit>
```

Notes

- in case of `CMP` both operands are the values to be compared `<src_reg_1>` and `<src_reg_2 | 16bit>` thus the destination register is omitted.
- For two 8-bit immediate values, provide both values separated by a space.

Listing 7: Compare Instruction

```
1 CMP    R1   R2       ; Compare R1 and R2
2 CMP    R1   15f      ; Compare R1 and 0x15F
3 CMP    2a   R11      ; Compare 0x2A and R11
4 CMP    12   34       ; Compare 0x12 and 0x34
```

3.2.8 Jump Instructions

`JUMP, JZ, JNZ, JLE, JL, JGE, JG, JE, JNE`

These instructions modify the program counter (`PC`) to alter the flow of execution. `JUMP` is an unconditional jump, while the others are conditional jumps based on the status flags set by previous `CMP` instructions. Operands may include:

- A 16-bit immediate value representing the target address
- A label defined in the assembly code (replaced by the assembler with the corresponding address)

Syntax

```
<Operation> <16bit>
```

Notes

- in case of jump instructions the only operand is the target address (immediate or label) thus both source registers and the destination register are omitted.

Listing 8: Compare Instruction

```
JUMP    15f      ; Jump to address 0x15F
JZ      myLabel  ; Jump to label if zero flag is set
JNZ     20       ; Jump to address 0x20 if zero flag is not set
JLE     end      ; Jump to label if less than or equal flag is set
JL      30       ; Jump to address 0x30 if less than flag is set
JE      equal    ; Jump to label if equal flag is set
JNE     50       ; Jump to address 0x50 if not equal flag is set
JG      40       ; Jump to address 0x40 if greater than flag is set
JGE     loop     ; Jump to label if greater than or equal flag is
    set
```

4 Implementation

4.1 Registers

4.1.1 D Flip-Flop (32 bit) : `d_FF`

It forms the basic building block of all the registers in the design. The flip flop captures the input data on the rising edge of the clock and holds it until the next rising edge. The vivado schematic of the 32 bit D flip flop is shown in figure 1.

The 4 bit flip flop has a similar implementation, just the input and output data width is 4 bits instead of 32 bits.

```
module d_FF(
  input [31:0] D,
  input clk,
  output reg [31:0] Q
);

  always @(posedge clk) begin
      Q <= D;
  end

endmodule
```

Listing 9: dFF Module

4.1.2 Register Unit (32 bit) : `reg32`

The register unit is a 32 bit register that uses the D flip-flop to store data. It has an enable signal that allows data to be written to the register only when the enable signal is high. The vivado schematic of the 32 bit register is shown in figure 2. It uses a MUX to select between the current value of the register and the new value to be written. The code for the MUX is shown below also all other MUX used in the design have a similar implementation. Yet again the 4 bit register has a similar implementation, just the input and output data width is 4 bits instead of 32 bits.

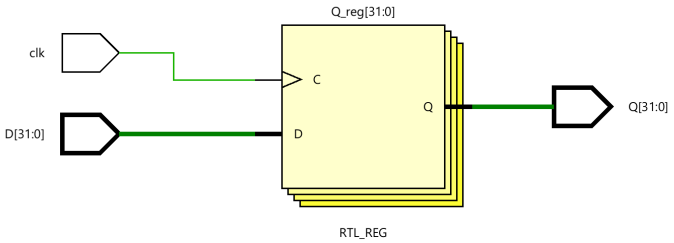

Figure 1: D Flip Flop (32 bit) Vivado Schematic

```
module mux2_1_32(
    input [31:0] X0, X1,
    input select,
    output reg [31:0] Y
);

    always @(*) begin
        case (select)
            1'b0 : Y = X0;
            1'b1 : Y = X1;
        endcase
    end

endmodule
```

Listing 10: mux 2:1 32 bit module

```
module reg32(
    input [31:0] D,
    input clk, enable,
    output [31:0] Q
);

    wire [31:0] out, in;

    mux2_1_32 M1 (
        .X0(out),
        .X1(D),
        .select(enable),
        .Y(in)
    );

    d_FF R_main (
        .D(in),
        .clk(clk),
        .Q(out)
    );

    assign Q = out;

```

```
endmodule
```

Listing 11: reg32 module

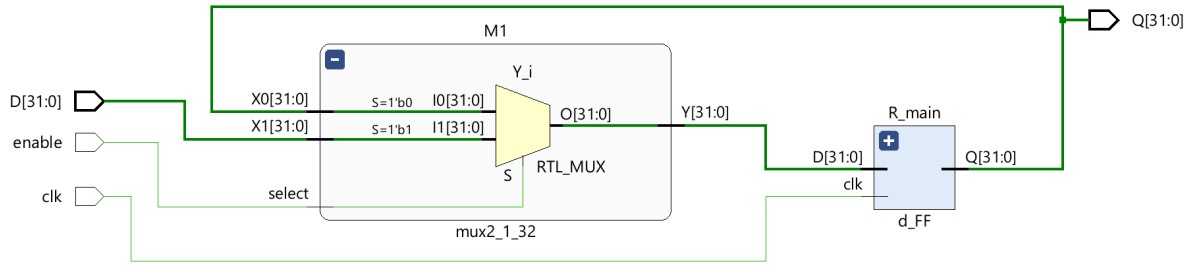

Figure 2: 32 bit Register Vivado Schematic

4.1.3 Dual write register unit (32 bit) : `regCell`

It is a 32 bit register that supports two write ports. It includes a small circuit to prevent race conditions when both write enables are high for the same register. In order to do this we use a MUX to select between the two data inputs based on the write enables. If only one of the enables is high the register sets the value else it ignores the write (XOR). The vivado schematic of the dual write 32 bit register is shown in figure 3.

The data is selected based on the value of `S1` pin as shown in the truth table below:

S1	data read from
0	D0
1	D1

```
module regCell(
    input [31:0] D0, D1,
    input S0, S1, clk,
    output [31:0] Q
);

    wire enable;
    wire [31:0] writeData;

    assign enable = S0 ^ S1;

    mux2_1_32 M1(
        .X0(D0), .X1(D1),
        .select(S1),
        .Y(writeData)
    );

    reg32 R32 (
        .D(writeData),
        .clk(clk), .enable(enable),
```

```
21          .Q(Q)
22      );
23
24  endmodule
```

Listing 12: regCell module

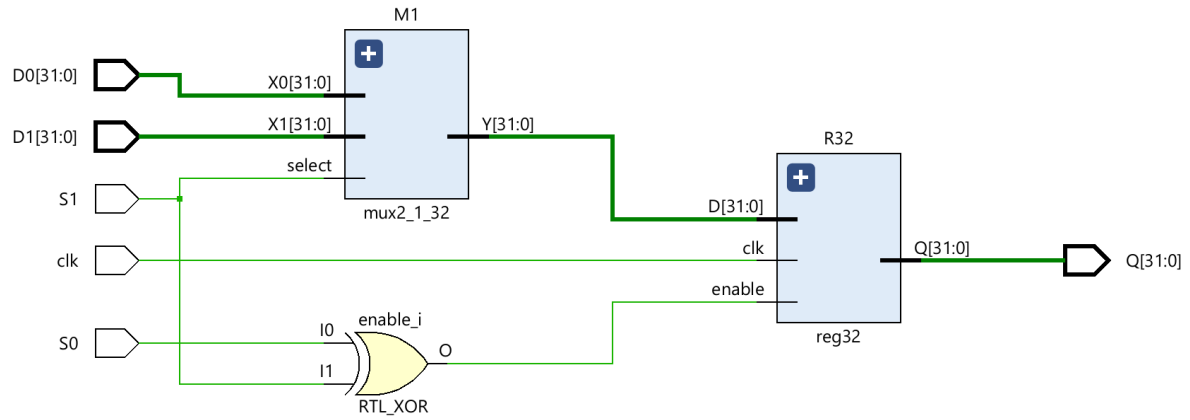

Figure 3: Dual write 32 bit Register Vivado Schematic

4.1.4 Register File 16x32 : `regFile`

The design incorporates 16 general-purpose 32-bit registers. It supports three read ports and two write ports, enabling simultaneous read and write operations. Potential race conditions arising from concurrent writes are resolved by the `regCell` module.

Figure 5 illustrates a portion of the Vivado-generated schematic for the register file. Due to its size, the complete schematic cannot be accommodated on a single page; hence, only a representative segment is shown. A hand-drawn conceptual diagram of the register file is also provided in Figure 5 for reference.

The implementation includes a 4-to-16 decoder, whose code is presented below. All other decoders used in the design follow a similar structure and logic.

Module Description The `regFile` module comprises the following key signals:

- `inp0`, `inp1`: Two 32-bit input data lines used for writing data into the register file.
- `inSelect0`, `inSelect1`: 5-bit control signals specifying the target register for each write operation. The most significant bit (MSB, bit 4) serves as a write enable flag:
  - `1` = Write enabled
  - `0` = Write disabled
- `outSelect0`, `outSelect1`, `outSelect2`: 4-bit signals selecting the registers whose contents are to be output.
- `out0`, `out1`, `out2`: Corresponding 32-bit output data lines reflecting the values of the selected registers.

```
module decoder16(
    input [3:0] X,
    input enable,
    output [15:0] Y
);

    wire [3:0] n;

    not (n[0], X[0]);
    not (n[1], X[1]);
    not (n[2], X[2]);
    not (n[3], X[3]);

    and (Y[00], n[3], n[2], n[1], n[0], enable);
    and (Y[01], n[3], n[2], n[1], X[0], enable);
    and (Y[02], n[3], n[2], X[1], n[0], enable);
    and (Y[03], n[3], n[2], X[1], X[0], enable);
    and (Y[04], n[3], X[2], n[1], n[0], enable);
    and (Y[05], n[3], X[2], n[1], X[0], enable);
    and (Y[06], n[3], X[2], X[1], n[0], enable);
    and (Y[07], n[3], X[2], X[1], X[0], enable);
    and (Y[08], X[3], n[2], n[1], n[0], enable);
    and (Y[09], X[3], n[2], n[1], X[0], enable);
    and (Y[10], X[3], n[2], X[1], n[0], enable);
    and (Y[11], X[3], n[2], X[1], X[0], enable);
    and (Y[12], X[3], X[2], n[1], n[0], enable);
    and (Y[13], X[3], X[2], n[1], X[0], enable);
    and (Y[14], X[3], X[2], X[1], n[0], enable);
    and (Y[15], X[3], X[2], X[1], X[0], enable);

endmodule
```

Listing 13: decoder16 module

```
module regFile(
    input [31:0] inp0, inp1,
    input [4:0] inSelect0, inSelect1, // [4] is write enable : 1
       = WRITE , 0 = NO WRITE
    input [3:0] outSelect0, outSelect1, outSelect2,
    input clk,
    output [31:0] out0, out1, out2
);
    wire [15:0] S0Arr, S1Arr;
    wire [511:0] rFileStatus;

    decoder16 D1(
        .X(inSelect0[3:0]),
        .enable(inSelect0[4]),
        .Y(S0Arr)
    );
    decoder16 D2(
        .X(inSelect1[3:0]),
```

```
18          .enable(inSelect1[4]),
19          .Y(S1Arr)
20      );
21
22      generate
23          for (genvar i = 0; i < 16; i = i + 1) begin
24              regCell rC(
25                  .D0(inp0),
26                  .D1(inp1),
27                  .S0(S0Arr[i]),
28                  .S1(S1Arr[i]),
29                  .clk(clk),
30                  .Q(rFileStatus[(i+1)*32 - 1:i*32])
31              );
32          end
33      endgenerate
34
35      mux16_1_32 M0(
36          .X(rFileStatus),
37          .select(outSelect0),
38          .clk(clk),
39          .Y(out0)
40      );
41      mux16_1_32 M1(
42          .X(rFileStatus),
43          .select(outSelect1),
44          .clk(clk),
45          .Y(out1)
46      );
47      mux16_1_32 M2(
48          .X(rFileStatus),
49          .select(outSelect2),
50          .clk(clk),
51          .Y(out2)
52      );
53
54  endmodule
```

Listing 14: regFile module

4.2 Arithmetic Logic Unit (ALU)

4.2.1 Adders : `halfAdder`, `fullAdder`, `add4`, `add32`, `addSub32`

The adder architecture is constructed hierarchically by reusing modular components. Specifically, the full adder module is built upon the half adder module, both of which are shown below. Their corresponding Vivado schematics are presented in Figure 6 and Figure 7, respectively.

A 4-bit ripple-carry adder is implemented using four instances of the full adder module. Its schematic is shown in Figure 8. The 32-bit adder used within the ALU follows the same structural pattern, with the only difference being the increased data width (32 bits instead of 4 bits).

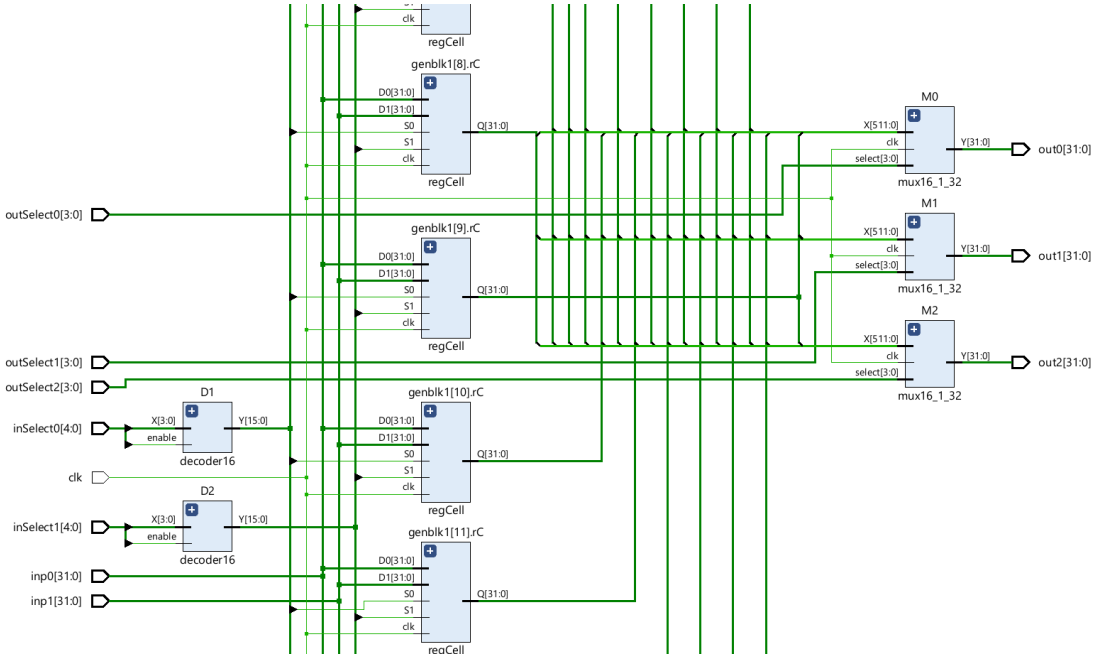

Figure 4: A portion of register file Vivado Schematic

The `addSub32` module extends the functionality of the `add32` adder by incorporating a conditional inversion mechanism using a XOR-based gate. This enables the module to perform both addition and subtraction based on a control signal:

- Control input = `0`: Performs addition
- Control input = `1`: Performs subtraction

The schematic of the `addSub32` module is shown in Figure 9, and its implementation code is provided below.

To facilitate conditional inversion, the `xor32A` module is used. It performs a bit-wise XOR between a 32-bit input and a single-bit control signal, effectively acting as a conditional NOT gate when the control signal is high.

```
1  module halfAdder(
2      input A, B,
3      output C, S
4  );
5
6      xor G1 (S, A, B);
7      and G2 (C, A, B);
8
9  endmodule
10
11 module fullAdder(
12     input Ci, A, B,
13     output Co, S
14 );
15
16     wire w1, w2, w3;
17
```

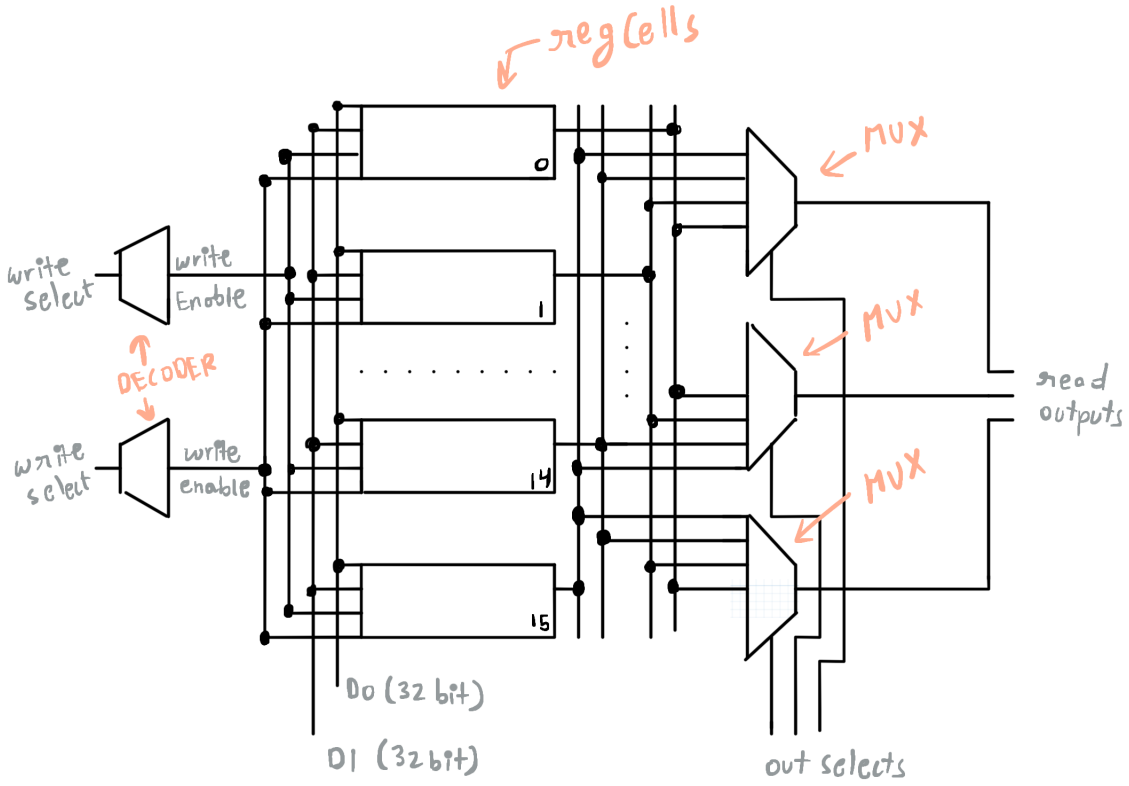

Figure 5: Basic Design of register file - hand drawn

```
18      or (Co, w1, w3);
19
20      halfAdder HA1 (
21          .A(Ci),
22          .B(A),
23          .C(w1),
24          .S(w2)
25      );
26
27      halfAdder HA2 (
28          .A(w2),
29          .B(B),
30          .C(w3),
31          .S(S)
32      );
33
34  endmodule
```

Listing 15: half adder and full adder modules

```
1  module add4(      // unsigned 4 bit adder
2      input [3:0] A, B,
3      input Ci,
4      output [3:0] S,
5      output Co
6  );
```

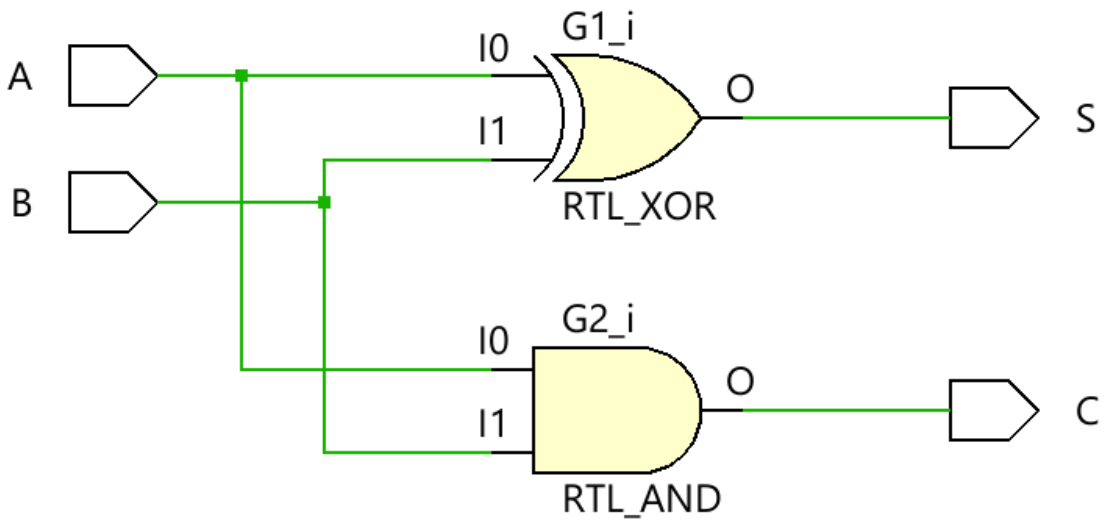

Figure 6: half adder Vivado Schematic

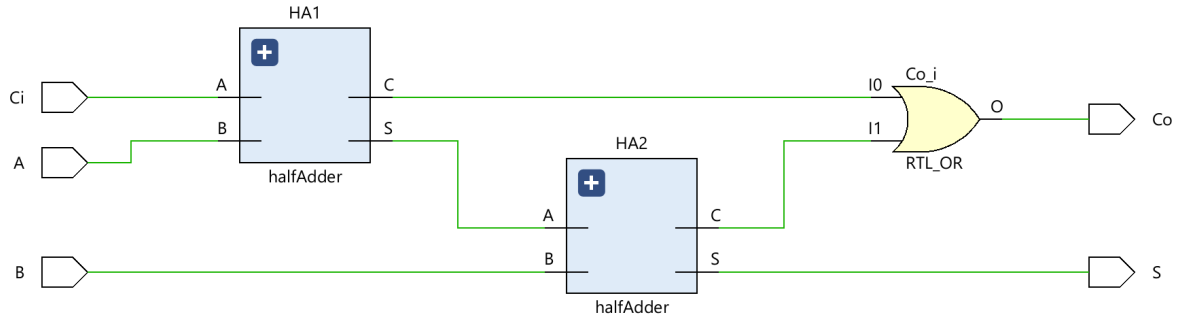

Figure 7: full adder Vivado Schematic

```
7
8       wire [3:0] cBus;
9
10      assign Co = cBus[3];
11
12      generate
13          for (genvar i = 0; i < 4; i = i + 1) begin
14              if (i == 0) begin
15                  fullAdder FAi (
16                      .A(A[i]),
17                      .B(B[i]),
18                      .Ci(Ci),
19                      .S(S[i]),
20                      .Co(cBus[i])
21                  );
22              end else begin
23                  fullAdder FAi (
```

```
                        .A(A[i]),
                        .B(B[i]),
                        .Ci(cBus[i-1]),
                        .S(S[i]),
                        .Co(cBus[i])
                    );
                end
            end
        endgenerate

endmodule
```

Listing 16: 4 bit adder

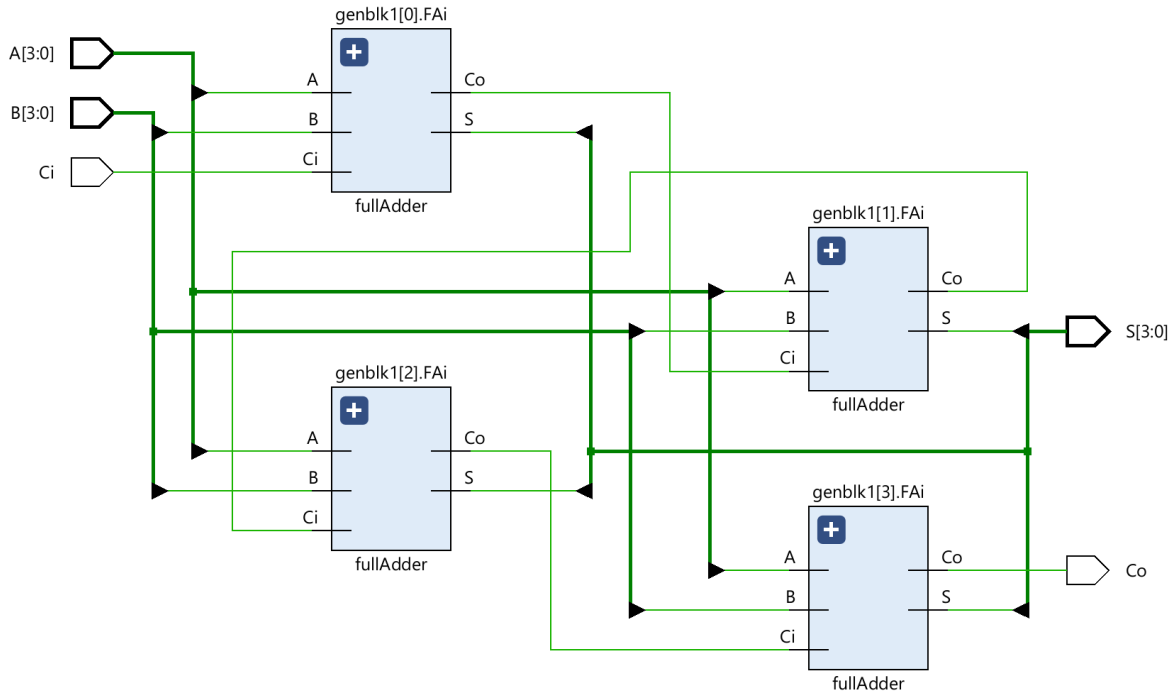

Figure 8: 4 bit adder Vivado Schematic

```
module addSub32(       // unsigned adder subtractor
    input [31:0] A, B,
    input control, // 1 = SUB 0 = ADD
    output [31:0] S,
    output Co
);

    wire [31:0] fB; // filtered B

    xor32A XA(
        .A(B),
        .B(control),
        .Y(fB)
    );

```

```
16      add32 a32 (
17          .Ci(control),
18          .A(A),
19          .B(fB),
20          .S(S),
21          .Co(Co)
22      );
23
24  endmodule
25
26  module xor32A(
27      input [31:0] A,
28      input B,
29      output [31:0] Y
30      );
31
32      generate
33          for (genvar i = 0; i < 32; i = i + 1) begin
34              xor (Y[i], A[i], B);
35          end
36      endgenerate
37
38  endmodule
```

Listing 17: 32bit adder subtractor, xor32A

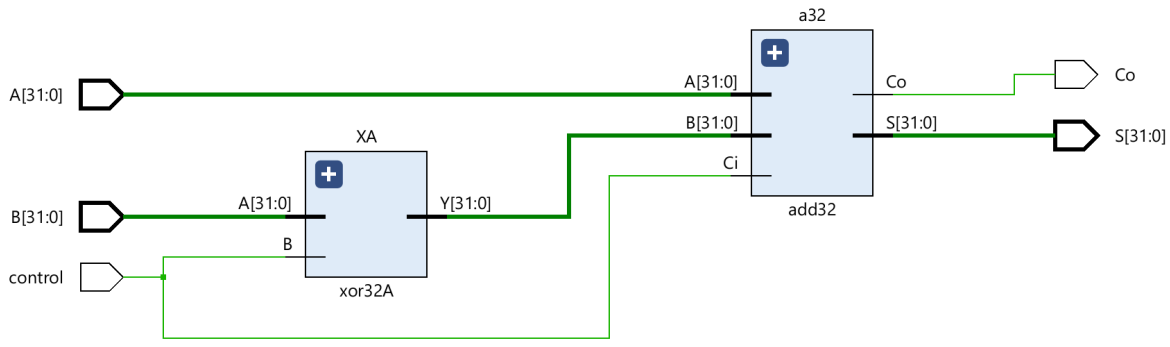

Figure 9: 32 bit adder subtractor Vivado Schematic

4.2.2 Magnitude Multiplier : `multiplierBase`

The Magnitude Multiplier is an internal module within the Arithmetic Logic Unit (ALU) designed to compute the product of two 32-bit unsigned integers, yielding a 64-bit result. The multiplication is performed in a single clock cycle using the shift-and-add algorithm.

At the core of this module is the specialized `and32M` unit, which multiplies a 32-bit operand by a single-bit value, producing a 32-bit partial product. This operation is repeated for each bit of the multiplier, resulting in 32 intermediate values. These partial products are then appropriately shifted and summed to produce the final 64-bit output.

To facilitate the accumulation of these 32 shifted partial products, the design employs 31 parallel 32-bit adders, with each adder's output aligned according to its corresponding

shift offset. Figure 10 illustrates the shift-and-add methodology, where the rows labeled `I1, I2, I3, I4` represent sample intermediate results.

Figure 11 further demonstrates the addition process using a simplified example involving three 4-bit adders to generate an 8-bit result. It is evident from the figure that, except for the first row, all subsequent additions follow a consistent structural pattern. Consequently, a `generate` block was utilized to instantiate the required adder logic programmatically.

While the complete 32-bit implementation is not depicted due to its size, the accompanying Verilog code provides a comprehensive representation of the underlying architecture and logic.

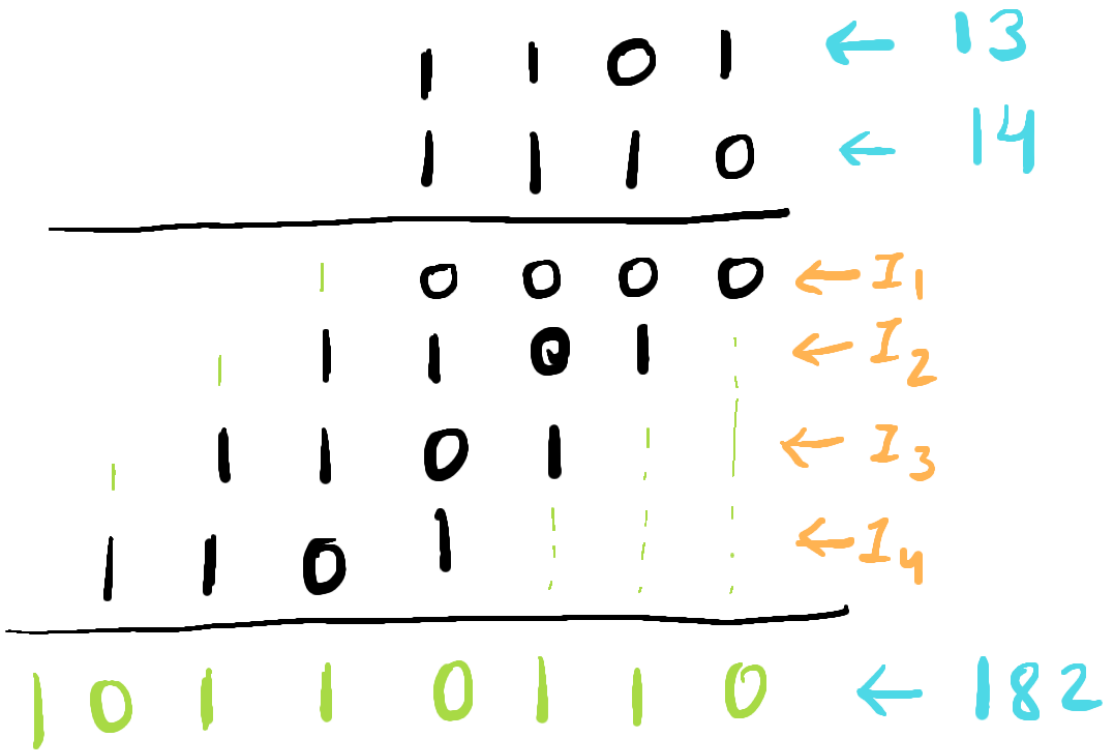

Figure 10: multiplication example on 2 4 bit numbers using shift and add method (13 x 14 = 182)

```
module and32M(
    input [31:0] A,
    input B,
    output [31:0] Y
    );

    generate
        for (genvar i = 0; i < 32; i = i + 1) begin
            and (Y[i], A[i], B);
        end
    endgenerate

endmodule

module multiplierBase(
```

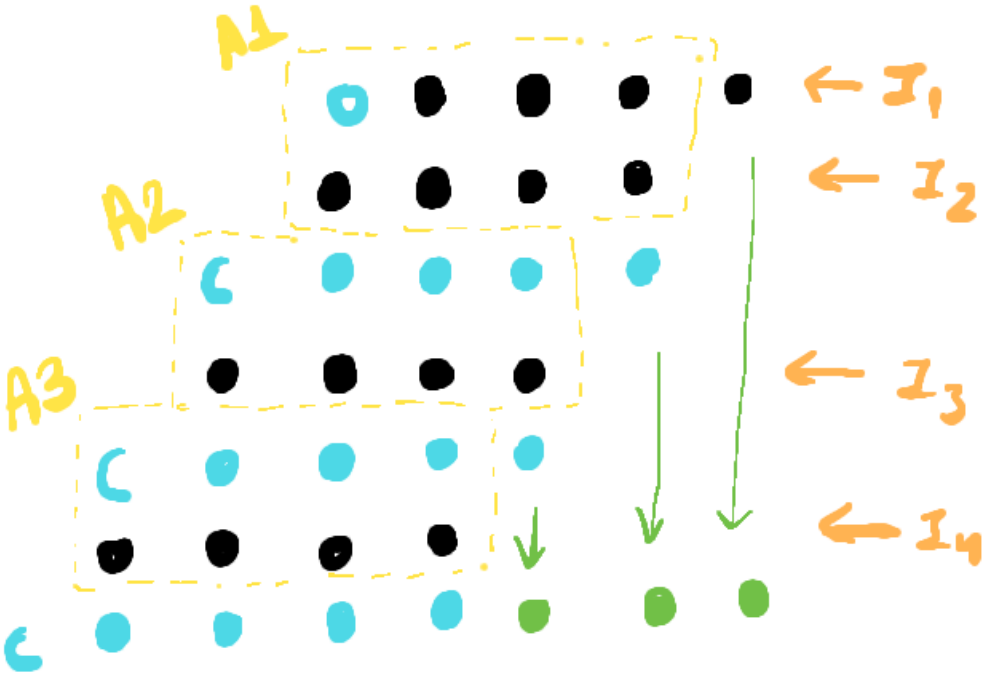

Figure 11: Addition of intermediates of 4 bit multiplication using 3 4bit adders to produce a 8 bit result

```
16      input [31:0] A, B,
17      output [31:0] OH, OL
18      );
19
20      wire [31:0] prodArr [31:0];
21      wire [30:0] carryBus;
22      wire [31:0] sumArr [30:0];
23      wire [64:0] finalProd;
24
25      generate
26          for (genvar i = 0; i < 32; i = i + 1) begin
27              and32M G1 (
28                  .A(A),
29                  .B(B[i]),
30                  .Y(prodArr[i])
31              );
32          end
33      endgenerate
34
35      generate
36          for (genvar i = 0; i < 31; i = i + 1) begin
37              if (i == 0) begin
38                  add32 G2 (
39                      .A({0, prodArr[0][31:1]}),
40                      .B(prodArr[1]),
41                      .Ci(1'b0),
```

```
42                      .S(sumArr[0]),
43                      .Co(carryBus[0])
44                  );
45              end else begin
46                  add32 G3 (
47                      .A({carryBus[i-1], sumArr[i-1][31:1]}),
48                      .B(prodArr[i+1]),
49                      .Ci(1'b0),
50                      .S(sumArr[i]),
51                      .Co(carryBus[i])
52                  );
53              end
54          end
55      endgenerate
56
57      assign OL[0] = prodArr[0][0];
58      assign OH = {carryBus[30] ,sumArr[30][31:1]};
59
60      generate
61          for (genvar i = 0; i < 31; i = i + 1) begin
62              assign OL[i+1] = sumArr[i][0];
63          end
64      endgenerate
65
66  endmodule
```

Listing 18: multiplier base and and32M modules

4.2.3 Magnitude Division : `dividerCell, division`

This module performs division between two 32-bit unsigned integers, yielding a 32-bit quotient and a 32-bit remainder. The operation is completed in a single clock cycle using a hardware-based long division algorithm.

The core of the division logic is implemented via the `dividerCell` unit. Each `dividerCell` receives the intermediate remainder from the previous stage, performs a left shift, appends the next input bit, and compares the resulting value against the divisor. Based on this comparison, it determines the next bit of the quotient: `0` if the value is less than the divisor, or `1` otherwise. If the value is greater than or equal to the divisor, the new remainder is computed as the difference between the value and the divisor; otherwise, the remainder remains unchanged.

This process is repeated iteratively for 32 stages to compute the final quotient and remainder.

The comparison is efficiently handled using a subtractor implemented via the existing adder module. If the subtraction yields a carry, the value is less than the divisor; otherwise, it is greater than or equal. This approach eliminates the need for separate subtraction and comparison logic.

Due to the complexity of the schematic, it is not included here. However, the corresponding Verilog implementation is provided below, offering insight into the architectural details.

```
1  module dividerCell(
```

```
    input nxtBit,
    input [31:0] prevAns, invDivider,
    output Q,
    output [31:0] R
    );

    wire [31:0] subRes;
    wire muxCont;
    wire temp;

    add32 S1 (
        .A({prevAns[30:0], nxtBit}),
        .B(invDivider),
        .Ci(1'b1),
        .S(subRes),
        .Co(temp)
    );

    mux2_1_32 M1 (
        .X0(subRes),
        .X1({prevAns[30:0], nxtBit}),
        .select(subRes[31]),
        .Y(R)
    );

    assign Q = ~subRes[31];

endmodule

module division(
    input [31:0] A, B, prevR,
    output [31:0] Q, R
);

    wire [31:0] invB;
    wire t;
    wire [31:0] subArr [31:0];

    generate
        for (genvar i = 0; i < 32; i = i + 1) begin
            not N (invB[i], B[i]);
        end
    endgenerate

    generate
        for (genvar i = 31; i >= 0; i = i - 1) begin
            if (i == 31) begin
                dividerCell dC1 (
                    .nxtBit(A[31]),
                    .prevAns(prevR),
                    .invDivider(invB),
```

```
                    .Q(Q[31]),
                    .R(subArr[31])
                );
            end else begin
                dividerCell dC2 (
                    .nxtBit(A[i]),
                    .prevAns(subArr[i+1]),
                    .invDivider(invB),
                    .Q(Q[i]),
                    .R(subArr[i])
                );
            end
        end
    endgenerate

    assign R = subArr[31];

endmodule
```

Listing 19: division base and divisionCell modules

4.2.4 Signed Arithmetic: `intMul`, `intDiv`, `negate32`

These modules extend the functionality of the magnitude-based arithmetic units to support signed operations. Each follows a three-stage process:

- **Stage 1: Sign and Magnitude Extraction** The output sign is computed by XOR-ing the sign bits of both inputs and AND-ing the result with the `us` control signal (`us` = 1 for unsigned, 0 for signed). If the operation is signed, negative inputs are converted to their magnitudes.
- **Stage 2: Core Arithmetic** The processed magnitudes are passed to the magnitude multiplication or division modules.
- **Stage 3: Output Adjustment** The result is conditionally negated based on the previously computed sign. For division, the `negate32` module is used. For multiplication, a custom 64-bit negation is implemented directly using the `add32` and `not` modules, as `negate32` supports only 32-bit inputs.

Schematics for `negate32`, `intMul`, and `intDiv` are shown in Figures 12, 13, and 14, respectively. The corresponding Verilog code is provided below for reference.

```
module negate32(
    input [31:0] X,
    output [31:0] Y
);

    wire [31:0] nx;

    not32 N32 (
        .X(X),
        .Y(nx)
```

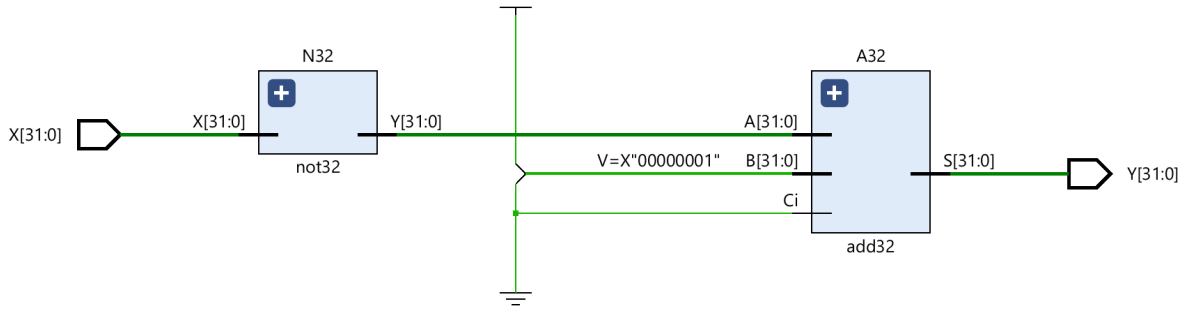

Figure 12: 32 bit negate : negate32 module schematic

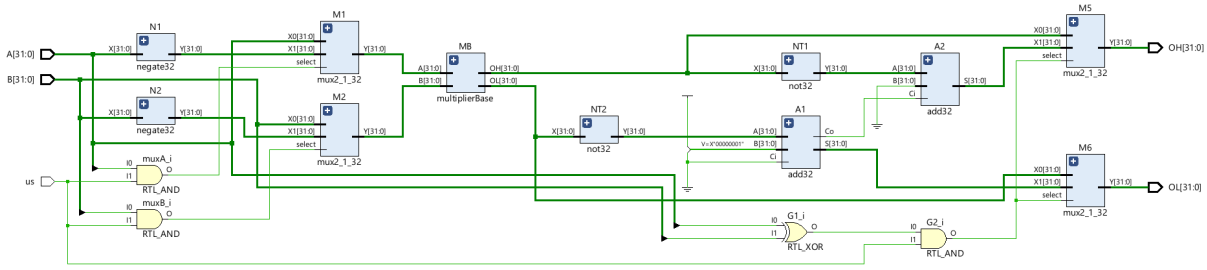

Figure 13: signed / unsigned multiplication module structure

```
11      );
12
13      add32 A32 (
14          .A(nx),
15          .B(32'h0000_0001),
16          .Ci(32'h0000_0000),
17          .S(Y)
18      );
19
20  endmodule
21
22  module intMul(
23      input [31:0] A, B,
24      input us, // 0 if input = unsigned, 1 if input = signed
25      output [31:0] OH, OL
26  );
27
28      wire xBit, fMux;
29
30      xor G1 (xBit, A[31], B[31]);
31      and G2 (fMux, xBit, us);
32
33      wire [31:0] mag1, mag2;
34      wire [31:0] oM1, oM2;
35      wire muxA, muxB;
36
```

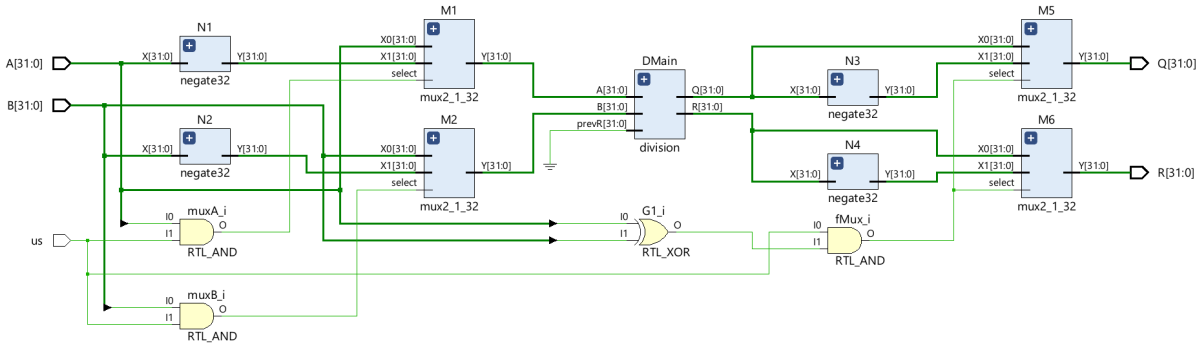

Figure 14: signed / unsigned division module structure

```
37      assign muxA = A[31] & us;
38      assign muxB = B[31] & us;
39
40      negate32 N1 (
41          .X(A),
42          .Y(mag1)
43      );
44
45      negate32 N2 (
46          .X(B),
47          .Y(mag2)
48      );
49
50      mux2_1_32 M1 (
51          .X0(A),
52          .X1(mag1),
53          .select(muxA),
54          .Y(oM1)
55      );
56
57      mux2_1_32 M2 (
58          .X0(B),
59          .X1(mag2),
60          .select(muxB),
61          .Y(oM2)
62      );
63
64      wire [31:0] OHm, OLm;
65
66      multiplierBase MB (
67          .A(oM1),
68          .B(oM2),
69          .OH(OHm),
70          .OL(OLm)
71      );
72
73      wire [31:0] OHn, OLn;
```

```
74
75      not32 NT1 (
76          .X(OHm),
77          .Y(OHn)
78      );
79
80      not32 NT2 (
81          .X(OLm),
82          .Y(OLn)
83      );
84
85      wire [31:0] OHn2, OLn2;
86      wire t;
87
88      add32 A1 (
89          .A(OLn),
90          .B(32'h0000_0001),
91          .Ci(1'b0),
92          .S(OLn2),
93          .Co(t)
94      );
95
96      add32 A2 (
97          .A(OHn),
98          .B(32'h0000_0000),
99          .Ci(t),
100         .S(OHn2)
101     );
102
103     mux2_1_32 M5 (
104         .X0(OHm),
105         .X1(OHn2),
106         .select(fMux),
107         .Y(OH)
108     );
109
110     mux2_1_32 M6 (
111         .X0(OLm),
112         .X1(OLn2),
113         .select(fMux),
114         .Y(OL)
115     );
116 endmodule
117
118 module intDiv(
119     input [31:0] A, B,
120     input us,
121     output [31:0] Q, R
122 );
123
124     wire fSign;
```

```
125
126     xor G1 (fSign, A[31], B[31]);
127
128     wire [31:0] absA, absB, negA, negB, baseQ, baseR;
129     wire muxA, muxB;
130
131     negate32 N1 (
132         .X(A),
133         .Y(negA)
134     );
135
136     negate32 N2 (
137         .X(B),
138         .Y(negB)
139     );
140
141     assign muxA = A[31] & us;
142     assign muxB = B[31] & us;
143
144     mux2_1_32 M1 (
145         .X0(A),
146         .X1(negA),
147         .select(muxA),
148         .Y(absA)
149     );
150
151     mux2_1_32 M2 (
152         .X0(B),
153         .X1(negB),
154         .select(muxB),
155         .Y(absB)
156     );
157
158     division DMain (
159         .A(absA),
160         .B(absB),
161         .prevR(32'b0000_0000),
162         .Q(baseQ),
163         .R(baseR)
164     );
165
166     wire [31:0] negQ, negR;
167
168     negate32 N3 (
169         .X(baseQ),
170         .Y(negQ)
171     );
172
173     negate32 N4 (
174         .X(baseR),
175         .Y(negR)
```

```
176     );
177
178     wire fMux;
179
180     and (fMux, us, fSign);
181
182     mux2_1_32 M5 (
183         .X0(baseQ),
184         .X1(negQ),
185         .select(fMux),
186         .Y(Q)
187     );
188
189     mux2_1_32 M6 (
190         .X0(baseR),
191         .X1(negR),
192         .select(fMux),
193         .Y(R)
194     );
195
196 endmodule
```

Listing 20: negate32, intMul, intDiv

4.2.5 Arithmetic comparator architecture : `arithCmp`

The Arithmetic comparator Unit is designed to perform four fundamental operations: addition, subtraction, multiplication, and division—on both signed and unsigned 32-bit inputs, denoted as `R0` and `R1`. In addition to arithmetic operations, the ALU supports comparison functionality by exposing two flags:

- `sMSB`: The signed most significant bit, extracted as bit 31 of the result from `R0 - R1`. It is set to `1` when the result is negative, i.e., `R0 < R1` (signed comparison).
- `usMSB`: The unsigned comparison flag, derived as the logical NOT of the carry-out from the `addSub32` module. It is set to `1` when `R0 < R1` (unsigned comparison).

These flags are consumed by comparator modules to update flag registers, which are subsequently used for conditional jump instructions.

A 3-bit control signal, `spCode`, determines the operation performed by the ALU according to the following encoding:

Both inputs (`R0`, `R1`) are simultaneously fed into three functional modules: `intMul`, `intDiv`, and `addSub32`. A multiplexer selects the appropriate output based on the value of `spCode`.

The control signal for the `addSub32` module is computed as:

```
spCode[0] | spCode[1] | cmp
```

This ensures that subtraction is enforced when the comparison module is active, while addition is selected only for `spCode` values corresponding to unsigned and signed addition.

The ALU provides two outputs:

spCode	Operation
000	Unsigned Addition
001	Unsigned Subtraction
010	Unsigned Multiplication
011	Unsigned Division
100	Signed Addition
101	Signed Subtraction
110	Signed Multiplication
111	Signed Division

Table 1: arithCmp Instruction decoding based on `spCode` values

- `Rd1`: Primary 32-bit result of the selected operation.
- `Rd2`: Secondary 32-bit output, relevant only for multiplication and division operations. It carries either the upper 32 bits of the product or the remainder, depending on the operation.

The selection between "Remainder" and "MSB 32 bits of product" for `Rd2` is governed by `spCode[0]`. In all other cases, `Rd2` is considered a "don't care" signal by external circuitry.

A schematic of the ALU architecture is shown in Figure 15, and the corresponding implementation code is provided below.

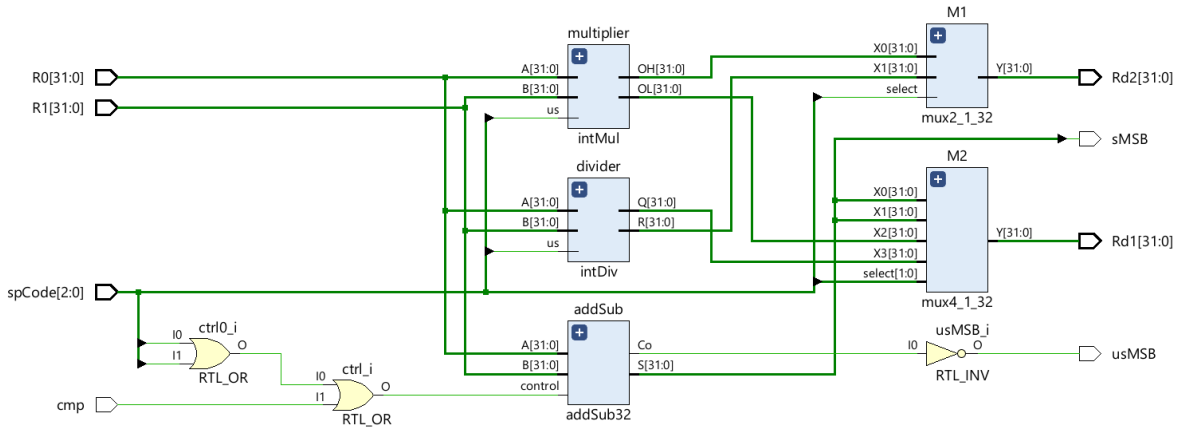

Figure 15: ALU architecture - Vivado Schematic

```
module arithCmp (
    input [31:0] R0, R1,
    input [2:0] spCode,
    input cmp,
    output [31:0] Rd1, Rd2,
    output usMSB, sMSB
);

    wire ctrl, bit33;
```

```
10      wire [31:0] addSubOut, mul1, mul0, divQ, divR, filteredR1;
11
12      assign ctrl = spCode[1] | spCode[0] | cmp;
13
14      addSub32 addSub(       // unsigned adder subtractor
15          .A(R0), .B(R1),
16          .control(ctrl), // 1 = SUB 0 = ADD
17          .S(addSubOut),
18          .Co(bit33)
19      );
20
21      assign usMSB = ~bit33;
22      assign sMSB = addSubOut[31];
23
24      intMul multiplier(
25          .A(R0), .B(R1),
26          .us(spCode[2]), // 0 if input = unsigned, 1 if input =
                signed
27          .OH(mul1), .OL(mul0)
28      );
29
30      intDiv divider(
31          .A(R0), .B(R1),
32          .us(spCode[2]),
33          .Q(divQ), .R(divR)
34      );
35
36      mux2_1_32 M1 (
37          .X0(mul1), .X1(divR),
38          .select(spCode[0]),
39          .Y(Rd2)
40      );
41
42      mux4_1_32 M2(
43          .X0(addSubOut), .X1(addSubOut), .X2(mul0), .X3(divQ),
44          .select(spCode[1:0]),
45          .Y(Rd1)
46      );
47
48  endmodule
```

Listing 21: ALU structure

4.2.6 Binary Block 1 : `Bin1, and32, or32, xor32, not32`

This block implements a suite of binary operations, including: `AND`, `OR`, `XOR`, `NAND`, `NOR`, `XNOR`, `NOT`, and `NEGATE`.

Operations such as `NOT` and `NEGATE` are unary and operate solely on the 32-bit input `R0`. All other operations are binary and require both `R0` and `R1` as inputs.

The schematic is designed to first compute the `AND`, `OR`, and `XOR` results between `R0` and `R1`. These outputs are then routed through a multiplexer (MUX), which selects the

spCode	Operation
000	AND
100	NAND
001	OR
101	NOR
010	XOR
110	XNOR
011	NEGATE
111	NOT

Table 2: B1 Instruction decoding based on `spCode` values

desired operation based on control signals. For the case where `select = 11`, the MUX is configured to pass `R0` directly, enabling identity passthrough.

The selected output is subsequently processed through a `NOT` gate and a `NEGATE` block. Final output selection is handled by additional MUXes, which determine whether the result should reflect a logical operation, its negation or inputs negation.

The schematic of the binary operation block is illustrated in Figure 16. Implementation details and code snippets for the `AND`, `OR`, `XOR`, and `NOT` operations are provided below.

```
module and32(
    input [31:0] A, B,
    output [31:0] Y
    );

    generate
        for (genvar i = 0; i < 32; i = i + 1) begin
            and N (Y[i], A[i], B[i]);
        end
    endgenerate

endmodule

module or32(
    input [31:0] A, B,
    output [31:0] Y
    );

    generate
        for (genvar i = 0; i < 32; i = i + 1) begin
            or N (Y[i], A[i], B[i]);
        end
    endgenerate

endmodule

module xor32(
    input [31:0] A, B,
    output [31:0] Y
```

```
30      );
31
32      generate
33          for (genvar i = 0; i < 32; i = i + 1) begin
34              xor N (Y[i], A[i], B[i]);
35          end
36      endgenerate
37
38  endmodule
39
40  module not32(
41      input [31:0] X,
42      output [31:0] Y
43      );
44
45      generate
46          for (genvar i = 0; i < 32; i = i + 1) begin
47              not N (Y[i], X[i]);
48          end
49      endgenerate
50
51  endmodule
52
53  module Bin1 (
54      input [31:0] R0, R1,
55      input [2:0] spCode,
56      output [31:0] res
57  );
58
59      wire [31:0] a32, o32, x32, aox, neAox, naox, lastMux0;
60
61      and32 A1 (
62          .A(R0), .B(R1),
63          .Y(a32)
64      );
65      or32 O1 (
66          .A(R0), .B(R1),
67          .Y(o32)
68      );
69      xor32 X1 (
70          .A(R0), .B(R1),
71          .Y(x32)
72      );
73
74      mux4_1_32 M1 (
75          .X0(a32), .X1(o32), .X2(x32), .X3(R0),
76          .select(spCode[1:0]),
77          .Y(aox)
78      );
79
80      negate32 N1(
```

```
81          .X(aox),
82          .Y(neAox)
83      );
84
85      not32 N2 (
86          .X(aox),
87          .Y(naox)
88      );
89
90      mux2_1_32 M2 (
91          .X0(aox), .X1(neAox),
92          .select(spCode[1] & spCode[0]),
93          .Y(lastMux0)
94      );
95
96      mux2_1_32 M3 (
97          .X0(lastMux0), .X1(naox),
98          .select(spCode[2]),
99          .Y(res)
100     );
101
102 endmodule
```

Listing 22: AND, OR, XOR, NOT, Bin1

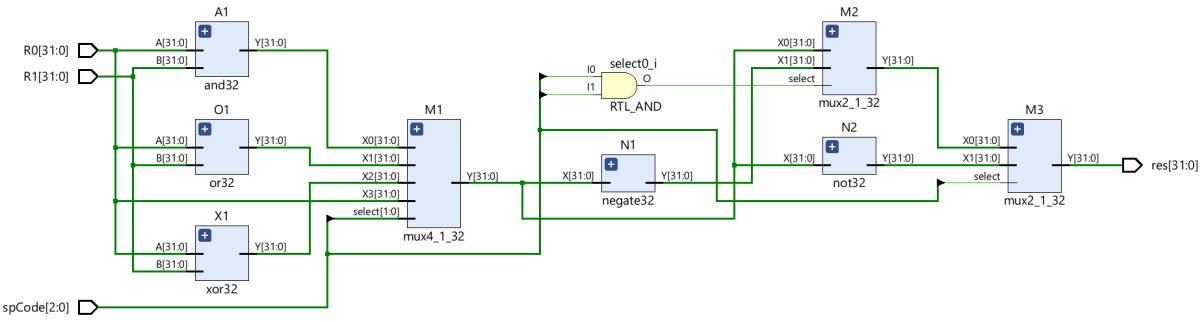

Figure 16: B1 Architecture - Vivado Schematic

4.2.7 Binary Block 2: `Bin2, lShift, rShift`

This module is responsible for performing binary shift operations. It utilizes the `lShift` submodule to execute left shifts and the `rShift` submodule to perform right shifts by specified amounts. Additionally, the module includes a control input to determine whether the shift is logical or arithmetic.

The `Bin2` block accepts three inputs:

- `spCode` — a control signal that specifies the operation to be performed.
- `R0` — a 32-bit input value on which the shift operation is applied.

spCode	Operation
00X	Left Shift
100	Logical Right Shift
101	Arithmetic Right Shift
01X	Rotate Left
11X	Rotate Right

Table 3: Instruction decoding for `Bin2` shift operations

- `R2` — a 32-bit input whose lower 5 bits determine the shift amount. Only the lower 5 bits are considered, as any shift beyond 32 bits is redundant for a 32-bit word.

The decoding of `spCode` for various operations is summarized in Table 3. The codes for `lShift, rShift, Bin2` are given below while the schematic for `Bin1` is shown in 17. All these modules use `tristate_buffer` whose code is also given below.

```
module tristate_buffer (
    input x,      // Data input
    input c,      // Control (enable)
    output y      // Output
);

assign y = c ? x : 1'bz;

endmodule

module lShift(
    input [31:0] X,
    input [4:0] shiftAmt,
    input SR, // shift = 0, ROTATE = 1
    output [31:0] Y
);

    wire [31:0] controlSig;

    decoder32 d32 (
        .X (shiftAmt),
        .Y(controlSig)
    );

    generate
        for (genvar i = 0; i < 32; i = i + 1) begin // shift amt
            for (genvar j = 0; j < 32; j = j + 1) begin // output
                if (j - i < 0) begin
                    wire muxConnect;

                    mux2_1_1 M (
                        .X0(1'b0),
                        .X1(X[j - i + 32]),
                        .select(SR),
                        .Y(muxConnect)
```

```
36                      );
37
38                      tristate_buffer tsb (
39                          .x(muxConnect),
40                          .y(Y[j]),
41                          .c(controlSig[i])
42                      );
43                  end else begin
44                      tristate_buffer tsb (
45                          .x(X[j - i]),
46                          .y(Y[j]),
47                          .c(controlSig[i])
48                      );
49                  end
50              end
51          end
52      endgenerate
53
54  endmodule
55
56  module rShift(
57      input [31:0] X,
58      input [4:0] shiftAmt,
59      input SR, // shift = 0, ROTATE = 1
60      input LA, // logical = 0, ARITHMETIC = 1
61      output [31:0] Y
62  );
63
64      wire [31:0] controlSig;
65
66      decoder32 d32 (
67          .X (shiftAmt),
68          .Y(controlSig)
69      );
70
71      generate
72          for (genvar i = 0; i < 32; i = i + 1) begin // shift amt
73              for (genvar j = 0; j < 32; j = j + 1) begin // output
74                  if (j + i >= 32) begin
75                      wire newBit, muxConnect;
76
77                      mux2_1_1 M_LA (
78                          .X0(1'b0),
79                          .X1(X[31]),
80                          .select(LA),
81                          .Y(newBit)
82                      );
83
84                      mux2_1_1 M_SR (
85                          .X0(newBit),
86                          .X1(X[j + i - 32]),
```

```
87                          .select(SR),
88                          .Y(muxConnect)
89                      );
90
91                      tristate_buffer tsb (
92                          .x(muxConnect),
93                          .y(Y[j]),
94                          .c(controlSig[i])
95                      );
96                  end else begin
97                      tristate_buffer tsb (
98                          .x(X[j + i]),
99                          .y(Y[j]),
100                         .c(controlSig[i])
101                     );
102                 end
103             end
104         end
105     endgenerate
106
107 endmodule
108
109 module Bin2 (
110     input [31:0] R0, R1, //R0 = num, R1 = amt to rotate with
111     input [2:0] spCode,
112     output [31:0] res
113 );
114
115     wire [31:0] L, R;
116
117     lShift LS(
118         .X(R0),
119         .shiftAmt(R1[4:0]),
120         .SR(spCode[1]), // shift = 0, ROTATE = 1
121         .Y(L)
122     );
123
124     rShift RS(
125         .X(R0),
126         .shiftAmt(R1[4:0]),
127         .SR(spCode[1]), // shift = 0, ROTATE = 1
128         .LA(spCode[0]), // logical = 0, ARITHMETIC = 1
129         .Y(R)
130     );
131
132     mux2_1_32 M1 (
133         .X0(L), .X1(R),
134         .select(spCode[2]),
135         .Y(res)
136    );
137
```

```
138 endmodule
```

Listing 23: lShift, rShift and B2 modules

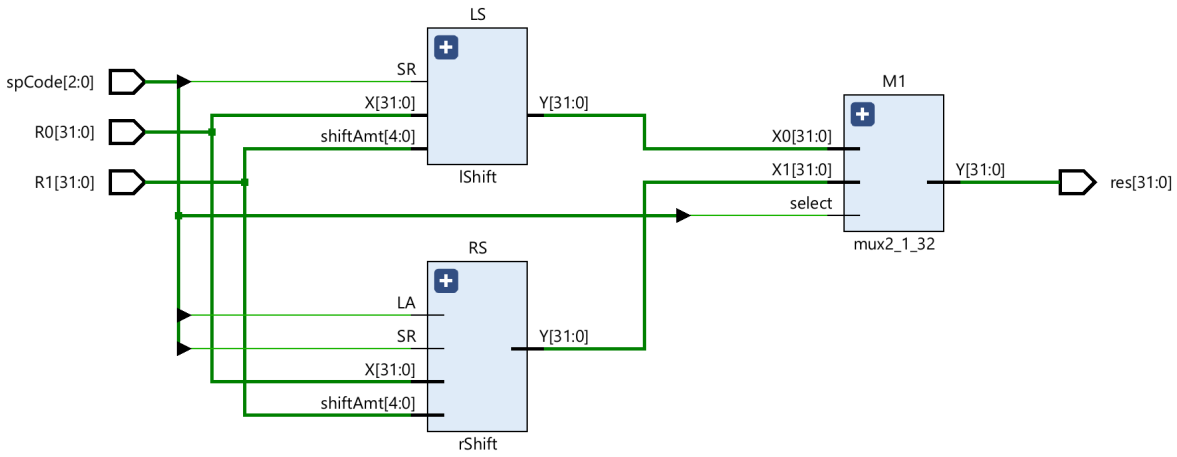

Figure 17: B2 Architecture - Vivado Schematic