\

Digital Design Lab
EEN 316

Project #1
ALU

Group 4
Nathan Paternoster
(Partners: Andrew O’neil-Smith,
Garrett Clausen)

Sayan Maity, TA

University of Miami
4/14/14

Abstract

This project was to construct an Arithmetic Logic Unit (ALU) programmatically using Altera’s ModelSim program. The ALU is one of the fundamental components of virtually every processor. It is a digital circuit responsible for performing integer arithmetic and logic operations. The ALU circuit consists of the ALU kernel and several sub-components that must be programmed individually beforehand. We first implemented these circuits using the behavioral method. Then we combined the sub-components along with the ALU kernel in a top-level file using the structural method. We performed a simulation of several arithmetic operations to verify that it works correctly.

Table of Contents

Overview

The sub-components that make up the ALU are the register, multiplexer, and tri-state buffer. For this project we will work with only 16-bit inputs. Therefore all of these components will be 16-bit. We will only need 2-to-1 MUXs.

The register will take a load input and a clock input. It will take data from its input and store it in its memory when the load input is set to '1' on the rising edge of the clock and put the data in its memory to the output on the falling edge of the clock. It is a sequential element so it will be run synchronously with the clock.

The 2-to-1 multiplexer takes two inputs and a select line. It will output data from one of its inputs depending on the value of the select line. Because it is a combinational element it will not depend on the clock.

The tri-state buffer controls access to a data bus. It takes an enable input that, when set to '1', will put the input to the output, and, when set to '0', will output nothing (ZZZs). It is also a combinational element and will not depend on the clock.

The ALU kernel will accept two 16-bit inputs, X and Y, as well as the carry-in input CI. It also takes a 5-bit operation code (OPC) that will specify which arithmetic operation it should perform. It will output the 16-bit result in R and six status bits: AZ (zero), AN (negative), AC (carry-out), AV (overflow), AS (sign of X-operand), and AQ (sign of quotient).

Objectives

To understand the function and implementation of the ALU and how it relates to modern processors.

Description

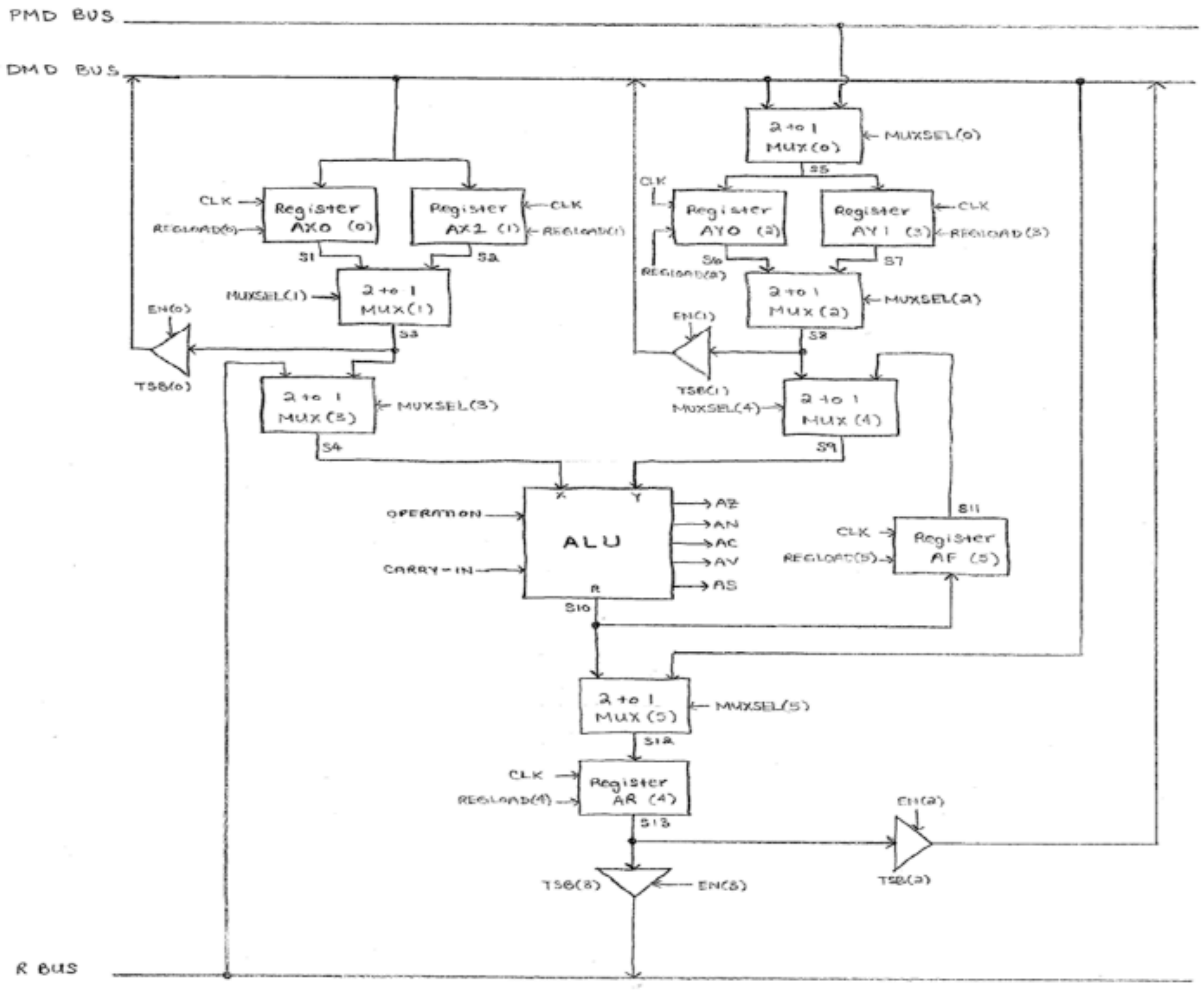
The first step to designing the ALU is to design its sub-components. We will design these entities in ModelSim using VHDL. Each sub-component will be designed to operate on 16-bits. The components will be programmed based on their ports specification and will all be implemented in the behavioral method. This means that the architecture of each sub-component will contain a process that will include a sensitivity list containing specific inputs. The register's process will be dependent on the clock; the multiplexer's process will be dependent on the inputs and select line; and the TSB's process will be dependent on only the control line (because the TSB is so simple it will only contain a conditional statement rather than a process).

The second step will be to design the ALU kernel. It will also be programmed behaviorally, and its process will depend upon the inputs (X, Y, CI) and the op-code (OPC). Its architecture will contain an internal signal (carry) that will act as the carry-in. Inside the process there will three local variables inX, inY, and outR. They will be 17 bits to allow for the analysis of any potential carry-out value. inX and inY will be set to X and Y and outR will be set to its correct value depending on the operation code. Finally the values of the status bits will be set accordingly and the result in outR will be copied over to R.

The third step to designing the ALU is to implement the top level coding. This step will require a structural entity to be programmed to represent the entire ALU circuit. It will contain the previous components and several internal signals. The architecture will consist of instances of all the components of the ALU circuit and the connections between them. The full circuit will require 2 X-input registers, 2 Y-input registers, 2 X-input MUXs, 3 Y-input MUXs, the ALU kernel, one feedback register, 1 output MUX, 1 output register, and 4 tri-state buffers.

Design Synthesis

The circuit will be designed using the following instances and internal signals:



ALU Function Code (AMF)

1	0	0	0	0	Y
1	0	0	0	1	Y + 1
1	0	0	1	0	X + Y + C
1	0	0	1	1	X + Y
1	0	1	0	0	NOT Y
1	0	1	0	1	– Y
1	0	1	1	0	X – Y + C – 1
1	0	1	1	1	X – Y
1	1	0	0	0	Y – 1
1	1	0	0	1	Y – X
1	1	0	1	0	Y – X + C – 1
1	1	0	1	1	NOT X
1	1	1	0	0	X AND Y
1	1	1	0	1	X OR Y
1	1	1	1	0	X XOR Y
1	1	1	1	1	X

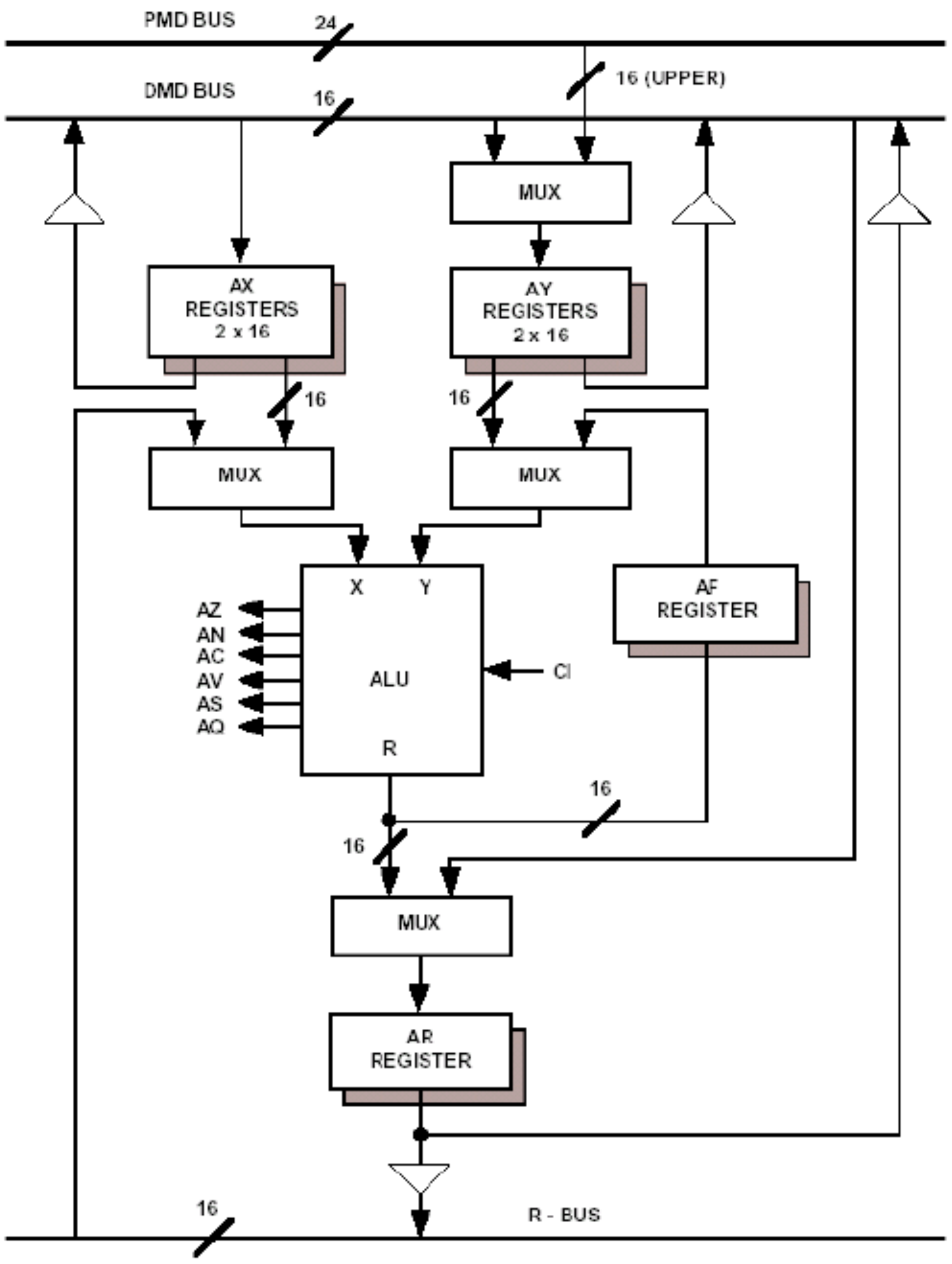
The operations our ALU will support are listed on the left. Our ALU will not be capable of multiplication or division. The top level circuit's ports are specified in the bottom picture.

ALU Complete Circuit/ Top level coding

Interface Specification

Name	Bit-Length	Mode	Description
DMD	16	Input&Output	DMD bus
PMD	24	Input	PMD bus
R	16	Input&Output	R bus
Load	6	Input	Load signals for registers
Sel	6	Input	Select signals for mux
En	4	Input	Control bits for tri-state buffer
OPC	5	Input	AMF function code
CI	1	Input	Carry in
Clk	1	Input	Clock signal
AZ	1	Output	Indicate if result is zero or not
AN	1	Output	Indicate if result is negative or not
AC	1	Output	Indicate there is carry out or not
AV	1	Output	Indicate there is overflow or not
AS	1	Output	Indicate if operand X is negative
AQ	1	Output	

Complete Logic Diagram



ALU block diagram

Code

Top-Level ALU circuit

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity ALUsixteen is
	port(DMD, R : inout std_logic_vector(15 downto 0);
		PMD : in std_logic_vector(23 downto 0);
		load, sel : in std_logic_vector(5 downto 0);
		en : in std_logic_vector(3 downto 0);
		opc : in std_logic_vector(4 downto 0);
		CI, clk : in std_logic;
		AZ, AN, AC, AV ,AS, AQ : out std_logic);
end entity;

architecture struct of ALUsixteen is
	component sixteenbitregister is
		port(inp : in  std_logic_vector(15 downto 0);
			load : in std_logic;
			clk : in std_logic;
			outp : out std_logic_vector(15 downto 0));
	end component;
	component sixteenbitTSB is
		port(data : in std_logic_vector(15 downto 0);
			ctr : in std_logic;
			outp : out std_logic_vector(15 downto 0));
	end component;
	 component sixteenbitMUX21 is
		port(inp1,inp2 : in std_logic_vector(15 downto 0);
			sel : in std_logic;
			outp : out std_logic_vector(15 downto 0));
	end component;
	component sixteenbitKernel is
		port(X, Y : in std_logic_vector(15 downto 0);
			CI : in std_logic;
			opc : in std_logic_vector(4 downto 0);
			R : out std_logic_vector(15 downto 0);
			AZ, AN, AC, AV, AS, AQ : out std_logic);
	end component;
	signal s1, s2, s3, X : std_logic_vector(15 downto 0);
	signal s5, s6, s7, s8, Y : std_logic_vector(15 downto 0);
	signal result, s11 : std_logic_vector(15 downto 0);
	signal s12, s13 : std_logic_vector(15 downto 0);
begin
	-- X input
	ax0 : sixteenbitregister port map (DMD, load(0), clk, s1);
	ax1 : sixteenbitregister port map (DMD, load(1), clk, s2);
	muxX1 : sixteenbitMUX21 port map (s1, s2, sel(1), s3);
```

```
		muxX2 : sixteenbitMUX21 port map (R, s3, sel(3), X);
		tsbX : sixteenbitTSB port map (s3, en(0), DMD);

		-- Y input
		muxY0 : sixteenbitMUX21 port map (DMD, PMD(23 downto 8), sel(0), s5);
		aY0 : sixteenbitregister port map (s5, load(2), clk, s6);
		aY1 : sixteenbitregister port map (s5, load(3), clk, s7);
		muxY1 : sixteenbitMUX21 port map (s6, s7, sel(2), s8);
		muxY2 : sixteenbitMUX21 port map (s8, s11, sel(4), Y);
		tsbY : sixteenbitTSB port map (s8, en(1), DMD);

		-- ALU
		ALU : sixteenbitKernel port map (X, Y, CI, opc, result, AZ, AN, AC, AV, AS, AQ);
		FBreg : sixteenbitregister port map (result, load(5), clk, s11);

		-- Output
		muxOut : sixteenbitMUX21 port map (result, DMD, sel(5), s12);
		regOut : sixteenbitregister port map (s12, load(4), clk, s13);
		outBuff0 : sixteenbitTSB port map (s13, en(3), R);
		outBuff1 : sixteenbitTSB port map (s13, en(2), DMD);
end architecture;
```

16-bit ALU Kernel Component

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_1164.all;

entity sixteenbitKernel is
		port(X, Y : in std_logic_vector(15 downto 0);
			CI : in std_logic;
			opc : in std_logic_vector(4 downto 0);
			R : out std_logic_vector(15 downto 0);
			AZ, AN, AC, AV, AS, AQ : out std_logic);
end entity;

architecture behavioral of sixteenbitKernel is
		signal carry: std_logic_vector(16 downto 0);
begin
		carry <= "0000000000000000" & CI;
		process(X, Y, CI, opc)
			variable outR : std_logic_vector(16 downto 0); -- 17 bits for 16 input + carry
			variable inX : std_logic_vector(16 downto 0);
			variable inY : std_logic_vector(16 downto 0);
		 begin
			inX := '0' & X;			-- making room for the 17th carry bit
			inY := '0' & Y;			-- making room for the 17th carry bit
			if (opc = "10000")then
				outR := inY;
			elsif (opc = "10001") then
				outR := inY +"00000000000000001";
```

```
elsif (opc = "10010") then
        outR := inX + inY + carry;
elsif (opc = "10011") then
        outR := inX + inY;
elsif (opc = "10100") then
        outR := not(inY);
elsif (opc = "10101") then
        outR := "00000000000000000" - inY;
elsif (opc = "10110") then
        outR := inX - inY + carry - "00000000000000001";
elsif (opc = "10111") then
        outR := inX - inY;
elsif (opc = "11000") then
        outR := inY - "00000000000000001";
elsif (opc = "11001") then
        outR := inY - inX;
elsif (opc = "11010") then
        outR := inY - inX + carry - "00000000000000001";
elsif (opc = "11011") then
        outR := not(inX);
elsif (opc = "11100") then
        outR := inX and inY;
elsif (opc = "11101") then
        outR := inX or inY;
elsif (opc = "11110") then
        outR := inX xor inY;
elsif (opc = "11111") then
        outR := inX;
else
        outR := "ZZZZZZZZZZZZZZZZZ";
end if;

 if (outR(15 downto 0) = "0000000000000000") then          -- setting AZ
        AZ <= '1';
else
        AZ <= '0';
end if;
AN <= outR(15);                             -- setting AN
AC <= outR(16);                             -- setting AC
AS <= X(15);                                -- setting AS

if (opc = "10111") then                     -- setting AV
        if (X(15)='0' and (not Y(15))='0' and outR(15)='1') then
                AV <= '1';
        elsif (X(15)='1' and (not Y(15))='1' and outR(15)='0') then
                AV <= '1';
        else
                AV <= '0';
        end if;
end if;
if (opc = "11001") then
        if (X(15)='1' and Y(15)='0' and outR(15)='1') then
                AV <= '1';
                elsif (X(15)='0' and Y(15)='1' and outR(15)='0') then
                    AV <= '1';
                else
                    AV <= '0';
                end if;
            elsif (TRUE) then
                if (X(15)='0' and Y(15)='0' and outR(15)='1') then
                    AV <= '1';
                elsif (X(15)='1' and Y(15)='1' and outR(15)='0') then
                    AV <= '1';
                else
                    AV <= '0';
                end if;
            end if;
            R <= outR(15 downto 0);          -- setting the result R
        end process;
end architecture;
```

16-bit 2-to-1 Multiplexer Component

```
library ieee;
use ieee.std_logic_1164.all

entity sixteenbitMUX21 is
    port(inp1, inp2: in std_logic_vector(15 downto 0);
        sel : in std_logic;
        outp : out std_logic_vector(15 downto 0));
end entity;

architecture behave of sixteenbitMUX21 is
begin
    process(inp1, inp2, sel)
    begin
        if (sel = '1')then
            outp <= inp2;
        elsif (sel = '0')then
             outp <= inp1;
        end if;
    end process;
end architecture;
```

16-bit Register Component

```
library ieee;
use ieee.std_logic_1164.all;

entity sixteenbitregister is
    port(inp : in  std_logic_vector(15 downto 0);
        load : in std_logic;
        clk : in std_logic;
```

```
			outp : out std_logic_vector(15 downto 0));
end entity;

architecture behave of sixteenbitregister is
		signal store: std_logic_vector(15 downto 0);
begin
		fourbitprocess: process(clk)
		begin
			if (clk'event and clk = '1' and load = '1') then
				store <= inp;
			end if;
			if (clk'event and clk = '0') then
				outp <= store;
			end if;
		end process;
end architecture;
```

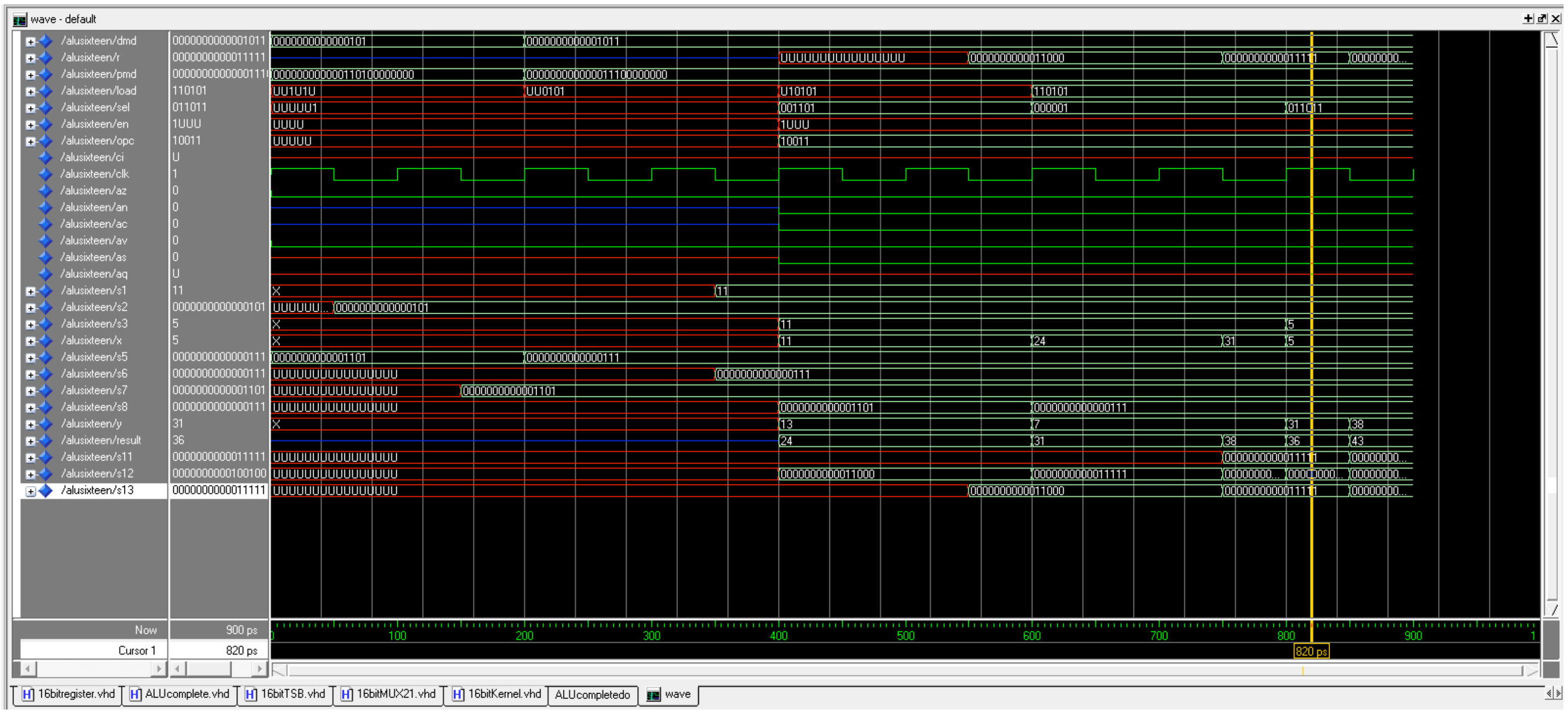
16-bit Tri-State Buffer Component

```
library ieee;
use ieee.std_logic_1164.all;

entity sixteenbitTSB is
		port(data : in std_logic_vector(15 downto 0);
			ctr : in std_logic;
			outp : out std_logic_vector(15 downto 0));
end entity;

architecture behave of sixteenbitTSB is
begin
		outp <= data when (ctr = '1') else "ZZZZZZZZZZZZZZZZ";
end architecture;
```

Results and Simulations

Simulation

```
AX1 = 5; AY1 = 13;
AX0 = 11; AY0 = 7;
AR = AX0 + AY1;
AF = AR + AY0;
AR = AF + AX1;
```



wave - default

/alusixteen/dmd	0000000000001011
/alusixteen/r	0000000000011111
/alusixteen/pmd	000000000000111
/alusixteen/load	110101
/alusixteen/sel	011011
/alusixteen/en	1UUU
/alusixteen/opc	10011
/alusixteen/ci	U
/alusixteen/clk	1
/alusixteen/az	0
/alusixteen/an	0
/alusixteen/ac	0
/alusixteen/av	0
/alusixteen/as	0
/alusixteen/aq	U
/alusixteen/s1	11
/alusixteen/s2	0000000000000101
/alusixteen/s3	5
/alusixteen/x	5
/alusixteen/s5	0000000000000111
/alusixteen/s6	0000000000000111
/alusixteen/s7	0000000000001101
/alusixteen/s8	0000000000000111
/alusixteen/y	31
/alusixteen/result	36
/alusixteen/s11	0000000000011111
/alusixteen/s12	0000000000100100
/alusixteen/s13	0000000000011111
Now	900 ps
Cursor 1	820 ps

The first four clock cycles loaded registers MX0, MX1, MY0, MY1 with 11, 5, 7, and 13 respectively and cleared the result register to 0. The next four clock cycles performed the first operation and placed the result (24) in our result register. The next operation (4 clock cycles) placed the result (31) in the result register. Finally we get our final calculated value of 36 placed in the result register.

Do script

```
vsim work.alusixteen(struct)
add wave sim:/alusixteen/*
# step 1 AX1=5, AY1=13
force -freeze sim:/alusixteen/clk 1 0, 0 {50 ps} -r 100
force -freeze sim:/alusixteen/dmd 0000000000000101 0
force -freeze sim:/alusixteen/pmd 000000000000110100000000 0
force -freeze sim:/alusixteen/sel(0) 1 0
force -freeze sim:/alusixteen/load(1) 1 0
force -freeze sim:/alusixteen/load(3) 1 0
run
run
# step 2 AX0=11, AY0 = 7
force -freeze sim:/alusixteen/dmd 0000000000001011 0
force -freeze sim:/alusixteen/pmd 000000000000011100000000 0
force -freeze sim:/alusixteen/load(3) 0 0
force -freeze sim:/alusixteen/load(1) 0 0
force -freeze sim:/alusixteen/load(0) 1 0
force -freeze sim:/alusixteen/load(2) 1 0
run
run
#step 3 AR= AX0 + AY1
force -freeze sim:/alusixteen/sel(1) 0 0
force -freeze sim:/alusixteen/sel(2) 1 0
force -freeze sim:/alusixteen/sel(3) 1 0
force -freeze sim:/alusixteen/sel(4) 0 0
# putting AR on to R bus
force -freeze sim:/alusixteen/sel(5) 0 0
force -freeze sim:/alusixteen/load(4) 1 0
force -freeze sim:/alusixteen/en(3) 1 0
force -freeze sim:/alusixteen/opc 10011 0
run
run
#step 4  AF = AR + AY0
force -freeze sim:/alusixteen/sel(3) 0 0
force -freeze sim:/alusixteen/sel(2) 0 0
force -freeze sim:/alusixteen/sel(4) 0 0
force -freeze sim:/alusixteen/load(5) 1 0
run
run
#step 5 AR = AF + AX1
force -freeze sim:/alusixteen/sel(4) 1 0
force -freeze sim:/alusixteen/sel(1) 1 0
force -freeze sim:/alusixteen/sel(3) 1 0
run
```

Conclusion

The ALU was successfully implemented and the simulation produced the correct result. We had to be careful to make sure that we were consistent with our internal signals. Our circuit can perform addition and subtraction but cannot perform multiplication or division.

Signed OFF

[Included]