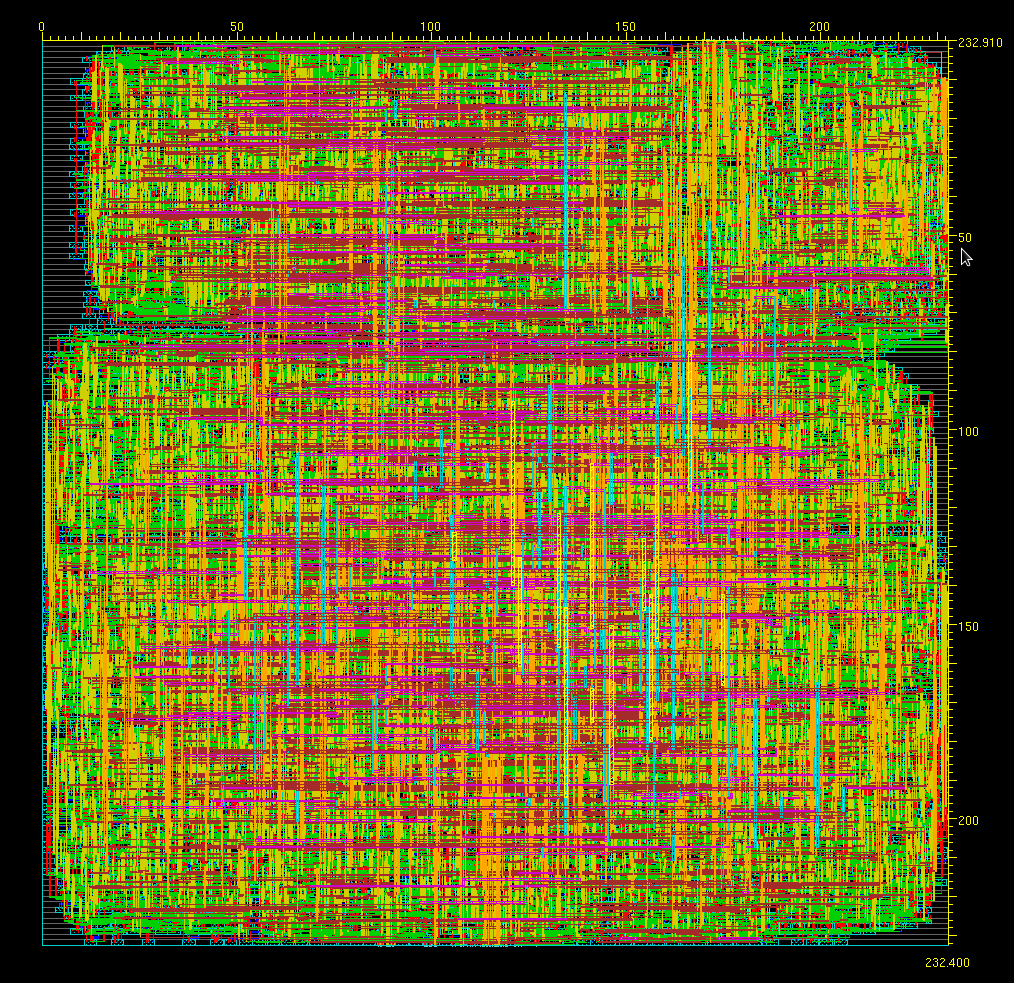
**MULTICYCLE CPU WITH APPROXIMATE ALU**

EC772 Project Report

Spring 2016



Darrin Ginoza

Kiran Vishal

Mahesh Bhutada

Omkar Nagendra

**Abstract**

The focus of our project was to construct a multi cycle CPU with a 32 bit input while incorporating an approximate multiplier within our ALU for 16 bit multiplications. The CPU accepts both I-type and R-type instructions and can perform operations such as addition, subtraction, multiplication, and logical operations. Among the operations specified, multiplication consumes the most computational power and time, so our focus was to improve this process. During the multiplication process, we implemented an approximation technique where we ignore a certain amount of bits depending on the size of the multiplicand. We perform the approximation with respect to a maximum calculated percent error with the exact value. The results of our implementation show a reduction in total cell area.

**Introduction**

Our project begins with a standard CPU construct with a 32-bit input, with 6-bit instructions accepting both I-type and R-type instructions. The standard CPU follows the flow diagram in figure 1. Data flows as follows:

0. The instruction is pre-fetched.

1. Instruction is decoded to determine R-type(01) or I-type(10) or I-type arithmetic/logical (11). These instructions are from the first two bits of op-code.

2. Determines address of register, and determines either load word (LW) or store word(SW).

3. Memory access for load word

4. Actually loading the word

5. Memory access for store word

6. Execution of logic computation

a. Op = 010000 : R1 = R2 -- MOVE

b. Op = 010010 : R1 = R2 + R3 – ADD

c. Op = 010011 : R1 = R2 - R3 -- SUBTRACT

d. Op = 010100 : R1 = R2 | R3 -- OR

e. Op = 010101 : R1 = R2 & R3 -- AND

f. Op = 010110 : R1 = R2 \* R3 -- MULTIPLY

Same instructions for I-type (except first two bits = 11)

7. Write back answer to memory

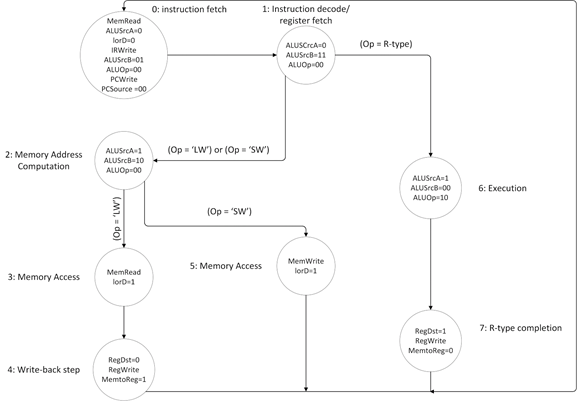


Figure 1: Standard CPU flow chart

**Design**

Approximation is implemented within the ALU block. First, calculations were done in LabView to map out how many bits of the LSB we can ignore for certain sized multiplicands. Calculations were done as shown in figure 2.

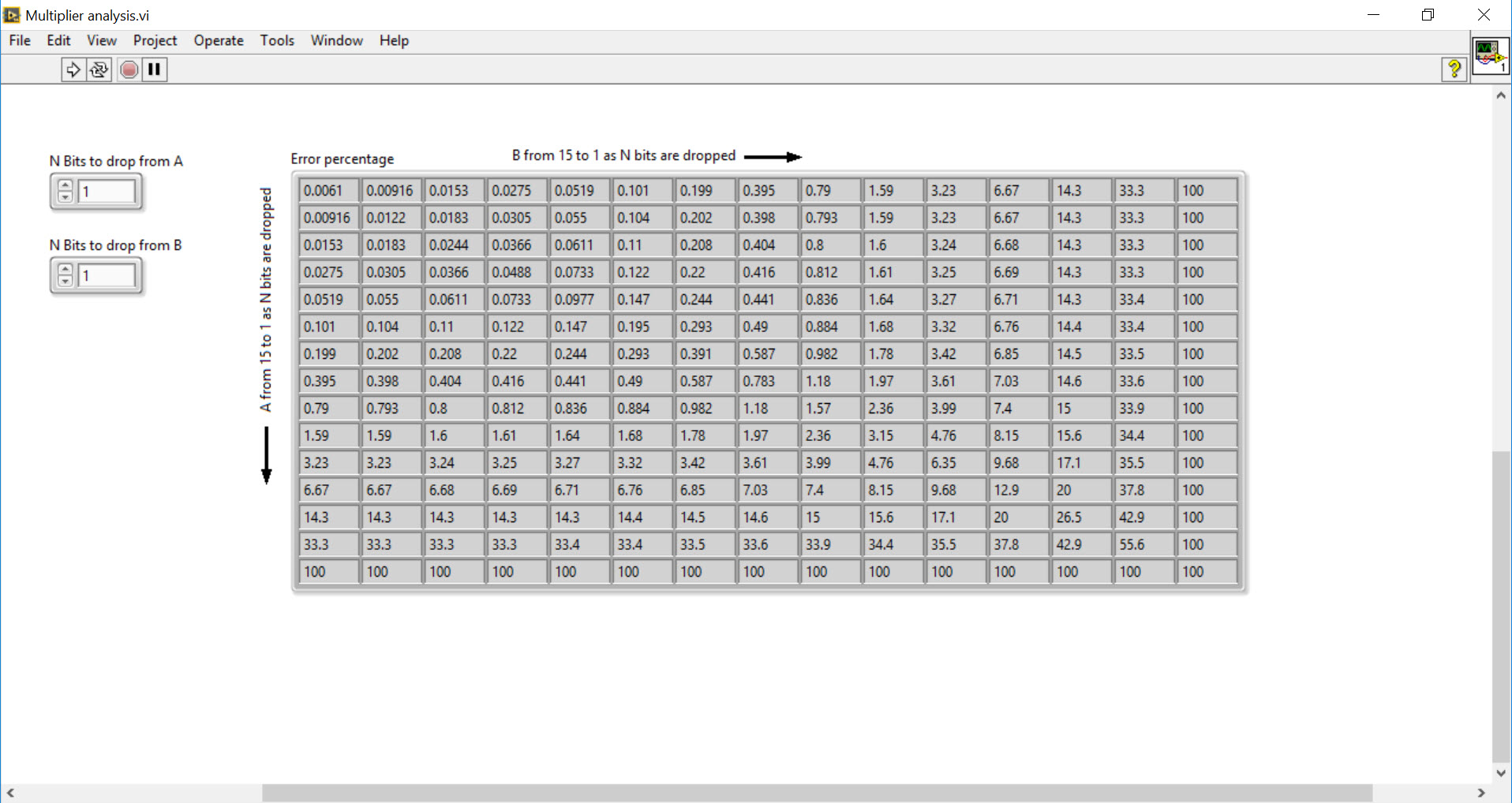
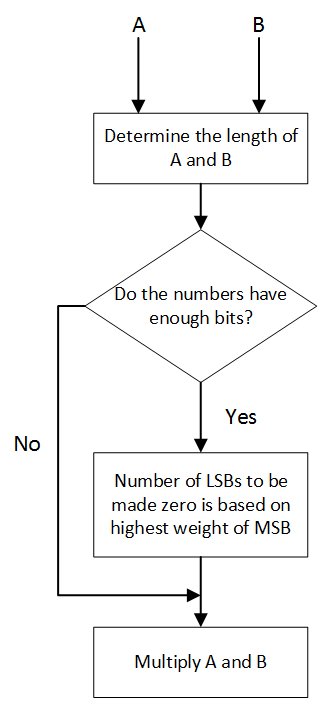


Figure 2: Calculations showing percent error

Figure 2 shows the percent error when two numbers of varying bits are multiplied together. We could adjust how many bits are eliminated, and see the resulting percent error. We predetermined how many bits were to be ignored, maximizing the number dropped, but keeping the percent error below 3%. We decided that for 16 bit numbers, 9 LSB bits can be ignored. For 15 bits, 8 bits can be ignored, and each consecutively smaller number, one less bit can be ignored. The flow of logic to process the approximation can be seen in figure 3.

  
Figure 3: Logic flow chart for approximation

Our approximation was implemented within ALU itself, and is described in figure 4.. Our op code will dictate to the ALU what logic function to execute, and if it the multiply operation, our approximation method will be implemented.

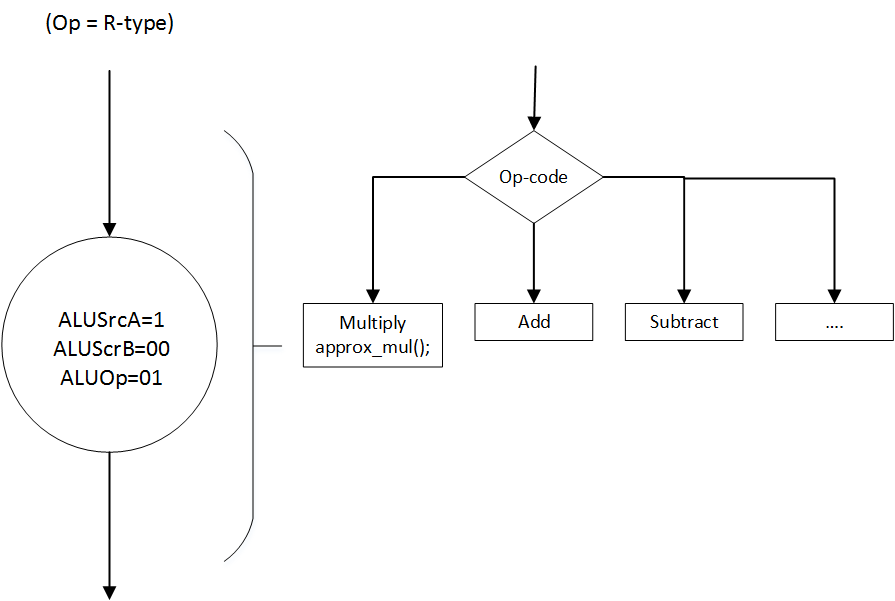


Figure 4: Approximation module embedded within ALU

**Design Implementation in Cadence Encounter**

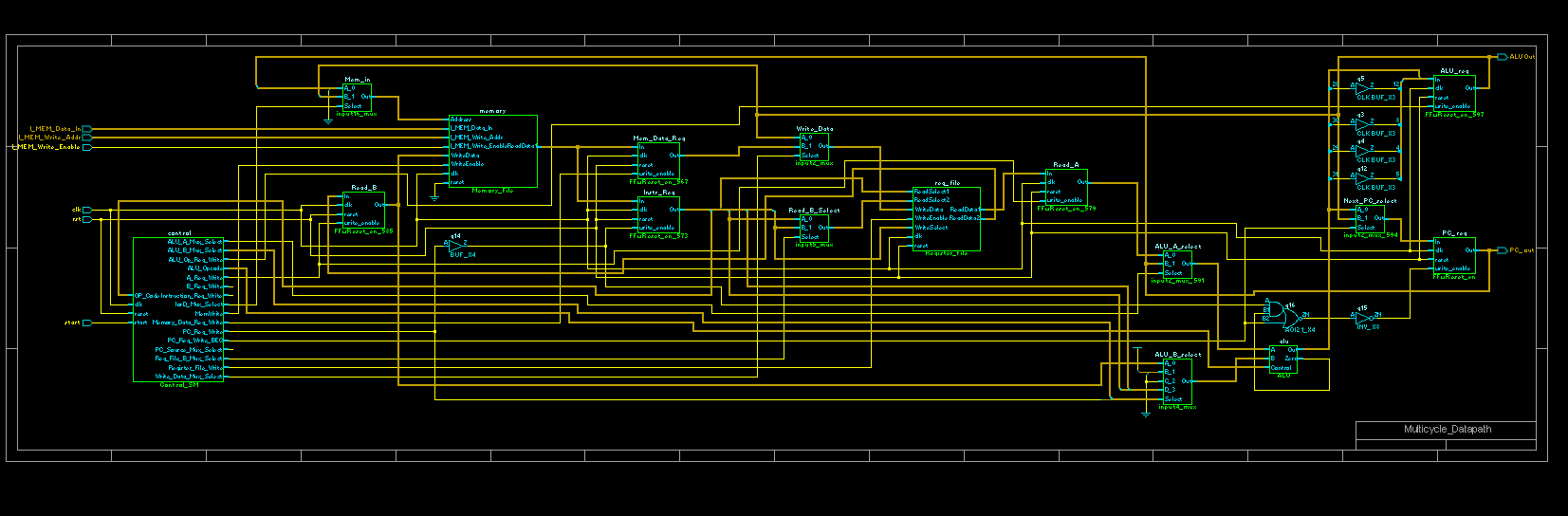


Figure 5: RTL schematic for regular CPU

Figure 5 above shows the RC schematic diagram of the overall standard CPU layout. As you can notice, it has striking similarities with the block diagram presented in figure 1. Each module represented here can be investigated further to expose the gate circuitry which drives each module.

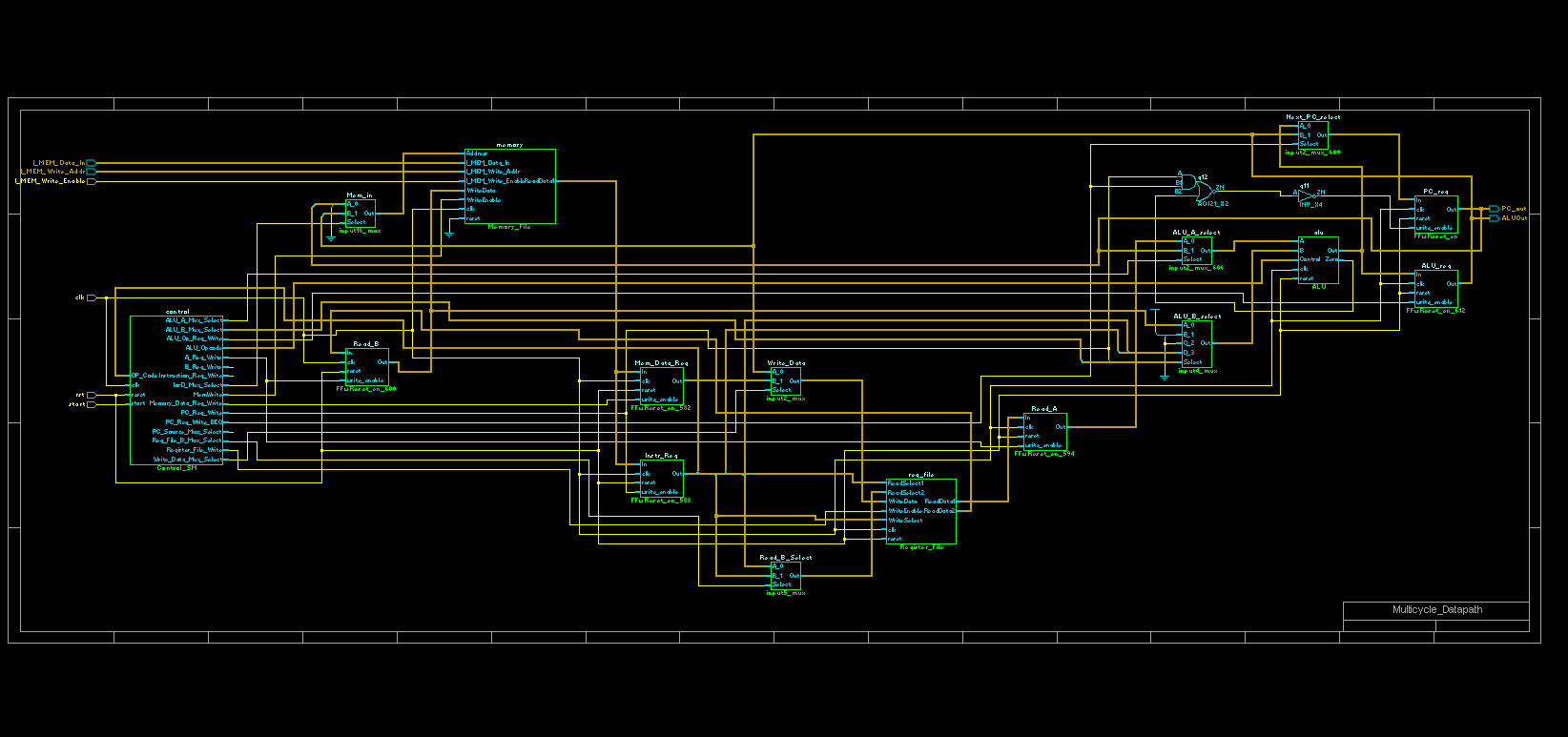


Figure 6: RTL schematic of approximate computing CPU

Figure 6 above shows the RC schematic diagram of the CPU with approximation module implemented. As you can probably notice, the layout is a bit different, but the number of modules are the same. This is due to the fact that our approximation module is embedded within the ALU, so overall, the schematic diagram will be the same.

Figure 7 on the right compares the memory blocks of the two implementations of the CPU, the left showing the standard CPU memory block, and the right showing the approximation CPU memory block. There are hardly any differences between the two, aside from moderate alterations in implementations due to differences in synthesizing the RC design.

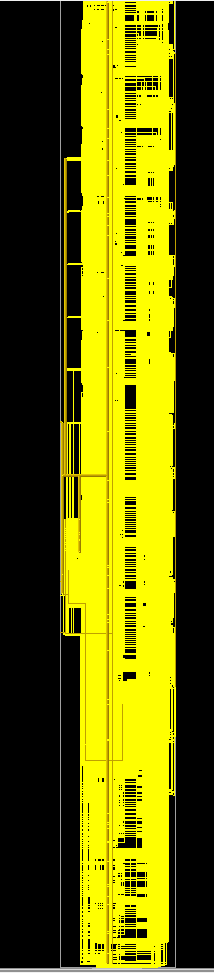


Figure 7: RTL memory block implementation of regular (left) and approximate computing CPU (right)

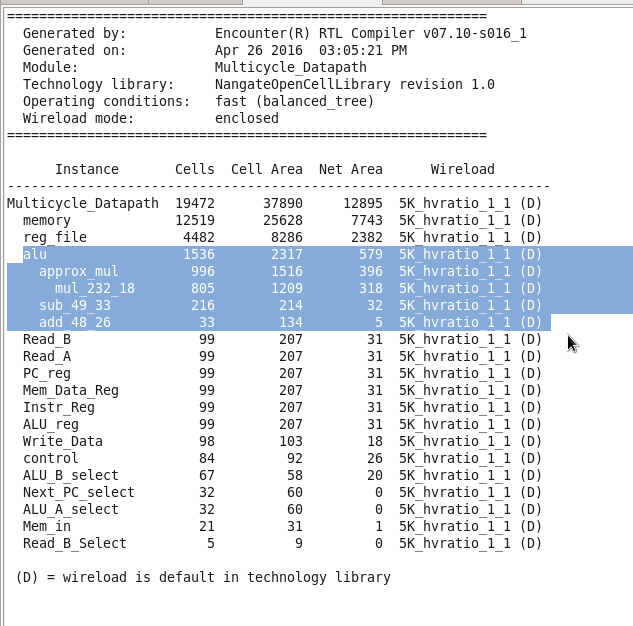
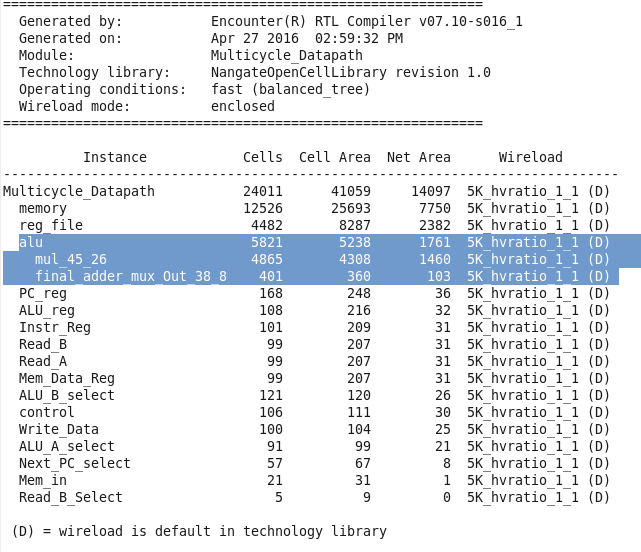


Figure 8: Regular CPU and approximate CPU cell area logs.

Figure 8 above shows the cell are logs generated by the standard and approximate CPU logs, respectively. The left is the logs from the approximate implementation, and the right is the logs from the standard CPU implementation. As you can see, the area of the ALU with the approximation module is actually lower than the area of the standard ALU implementation









Figure 9: Regular CPU and Approximate CPU Amoeba

Figure 9 above shows the amoeba representations of our chip post encounter synthesis. As you can see, the memory takes up the majority of the chip. This is only with 100 memory locations. A full implementation of the memory would require about 65000 memory locations, so if that was implemented, not only would our entire chip be mostly memory cells, but the synthesis time would be much longer, as the chip size would increase as well. The left chip is our approximation chip, with the approximation module embedded, and the right chip is our standard CPU module. There is no noticeable difference in size between the two.

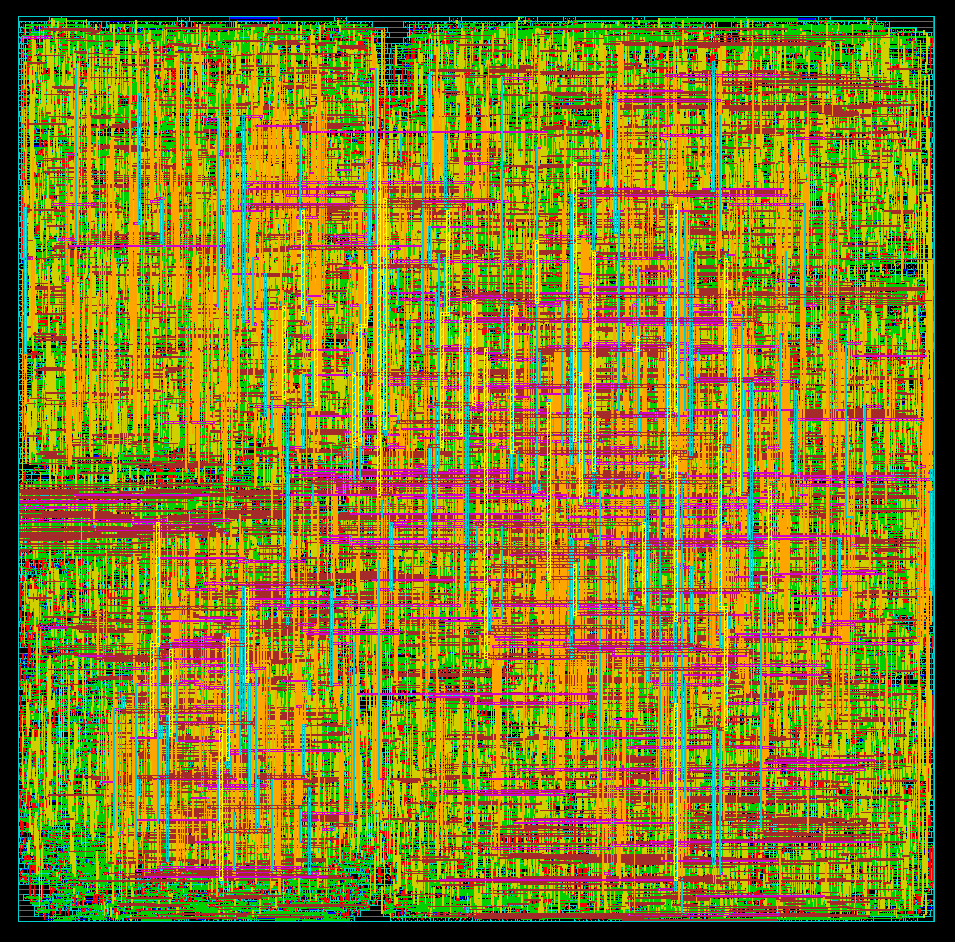
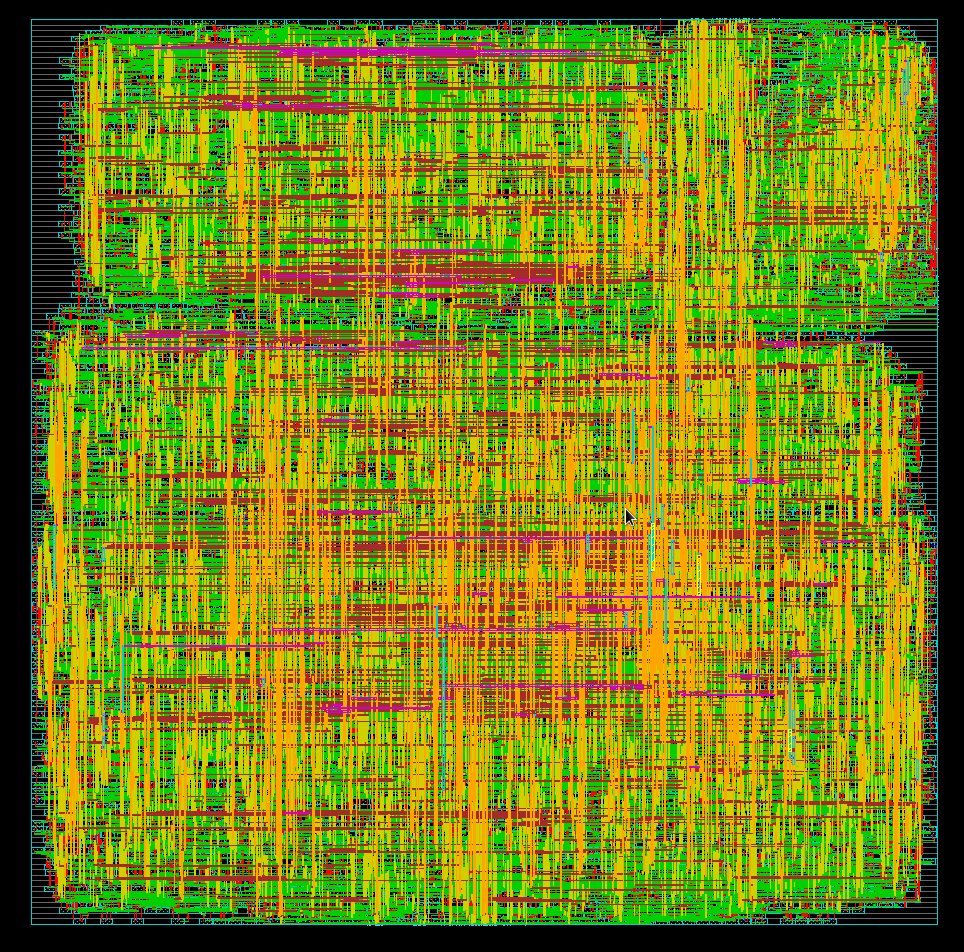
Figure 10: Regular CPU and Approximate CPU Globaldetailroute

Figure 10 above shows the metal layers between the the approximate CPU, right, and the standard CPU, left. There are some differences in how Cadence performed the layout for the place and route, but 10 layers of metal was still utilized in both cases.

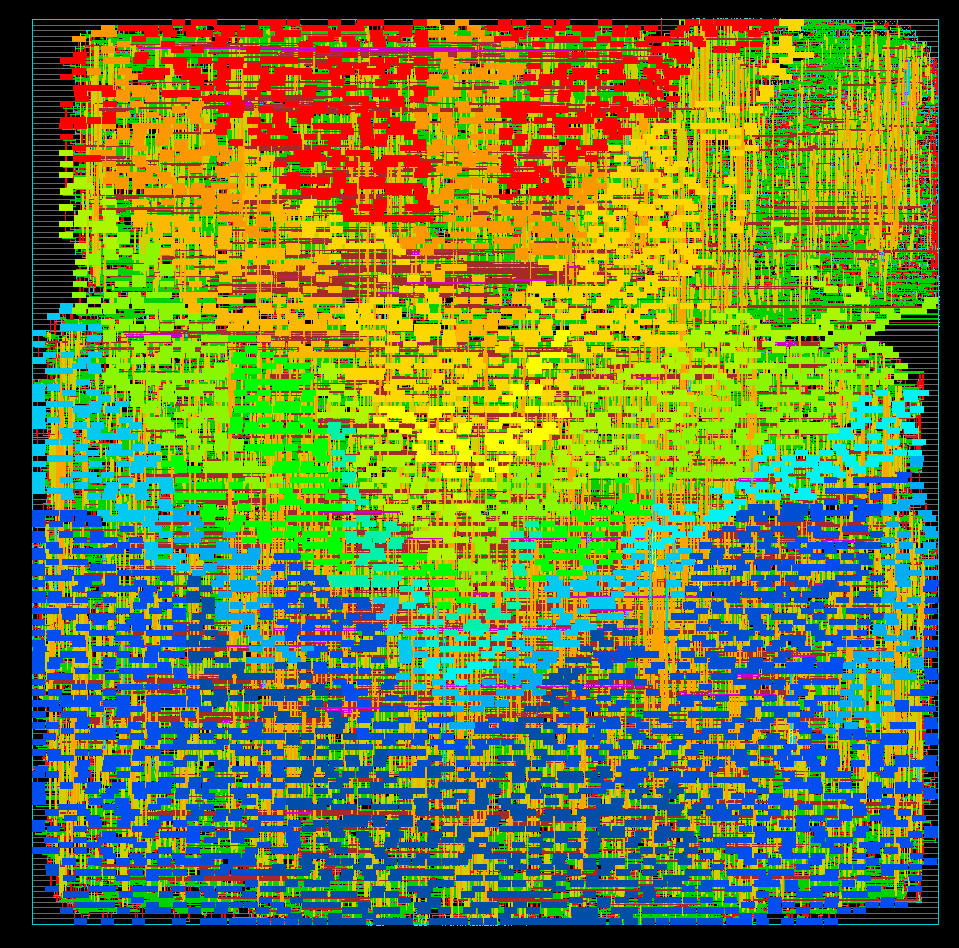
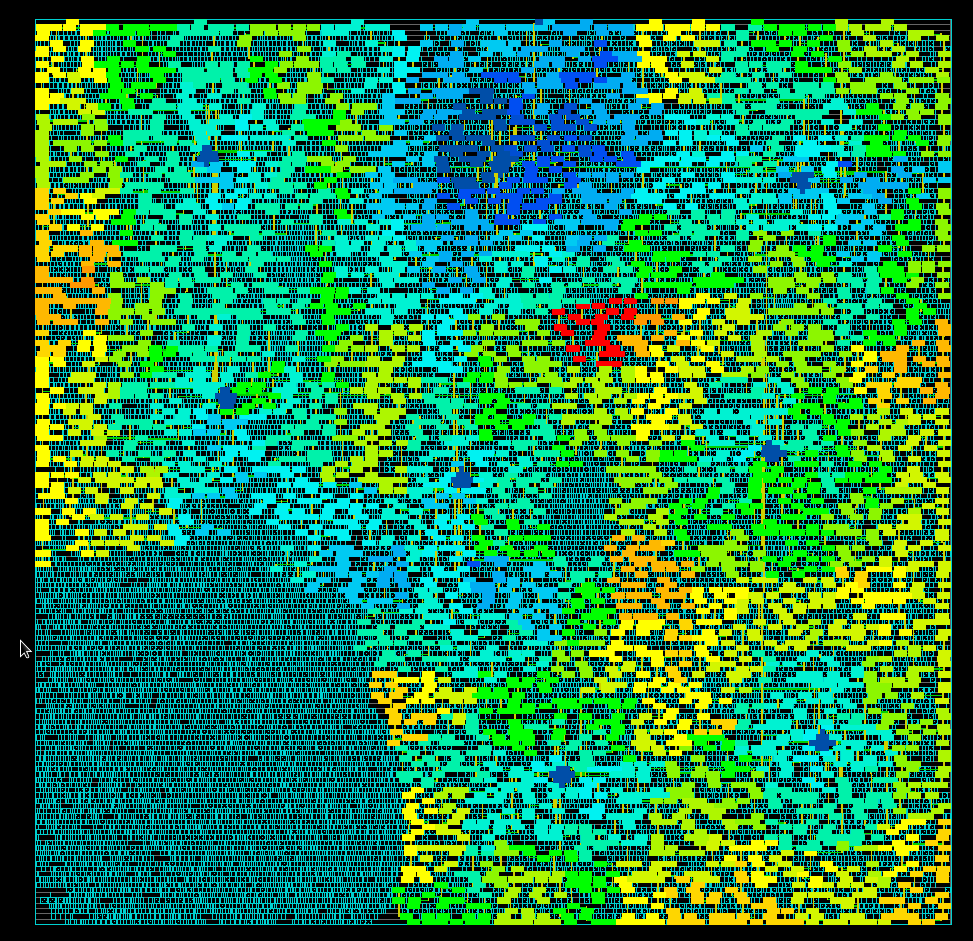


Figure 11: Regular CPU and Approximate CPU Clock Phase Delay

Figure 11 above shows the phase delay of the who implementations of the chip. The left shows the phase delay of the standard CPU chip, and the right shows the phase delay of the chip with approximate multiplication. Notice how the phase delay is worse in the approximation implementations, as well as the lack of clock in the ALU. In the standard CPU implementation, the ALU does not require a clock, but the approximation block does require clock signal. On the right, you can see some of the clock penetrating the ALU.

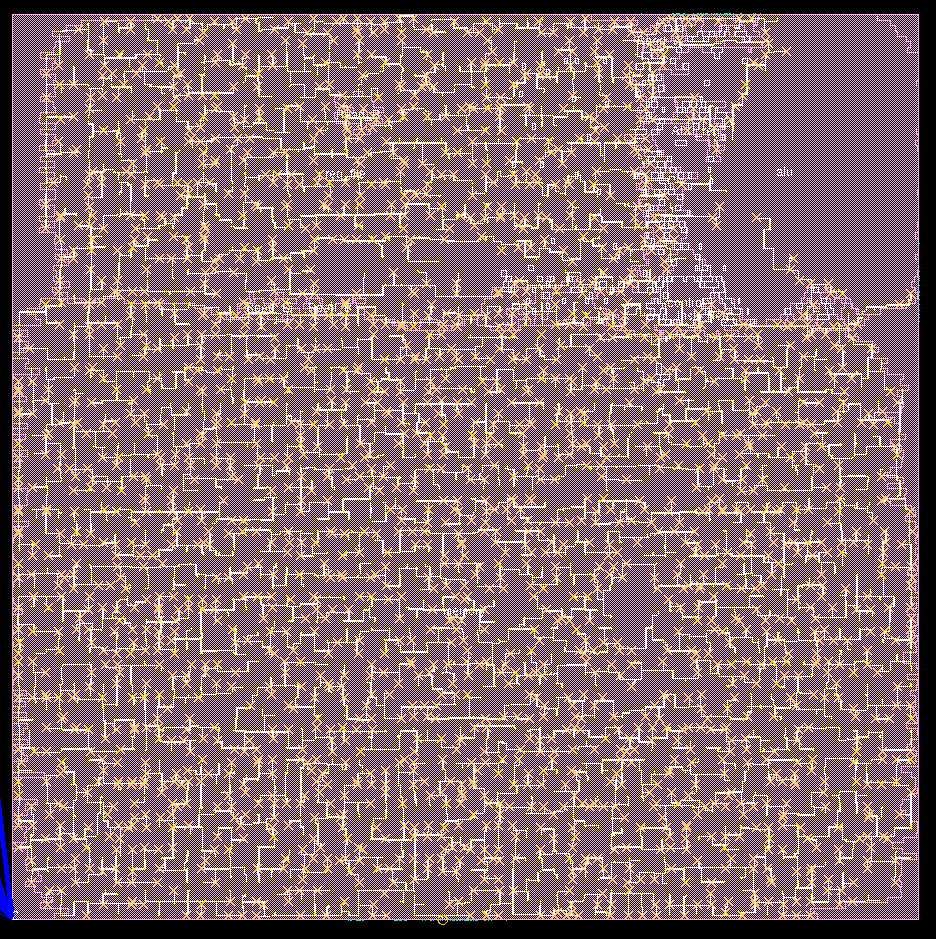
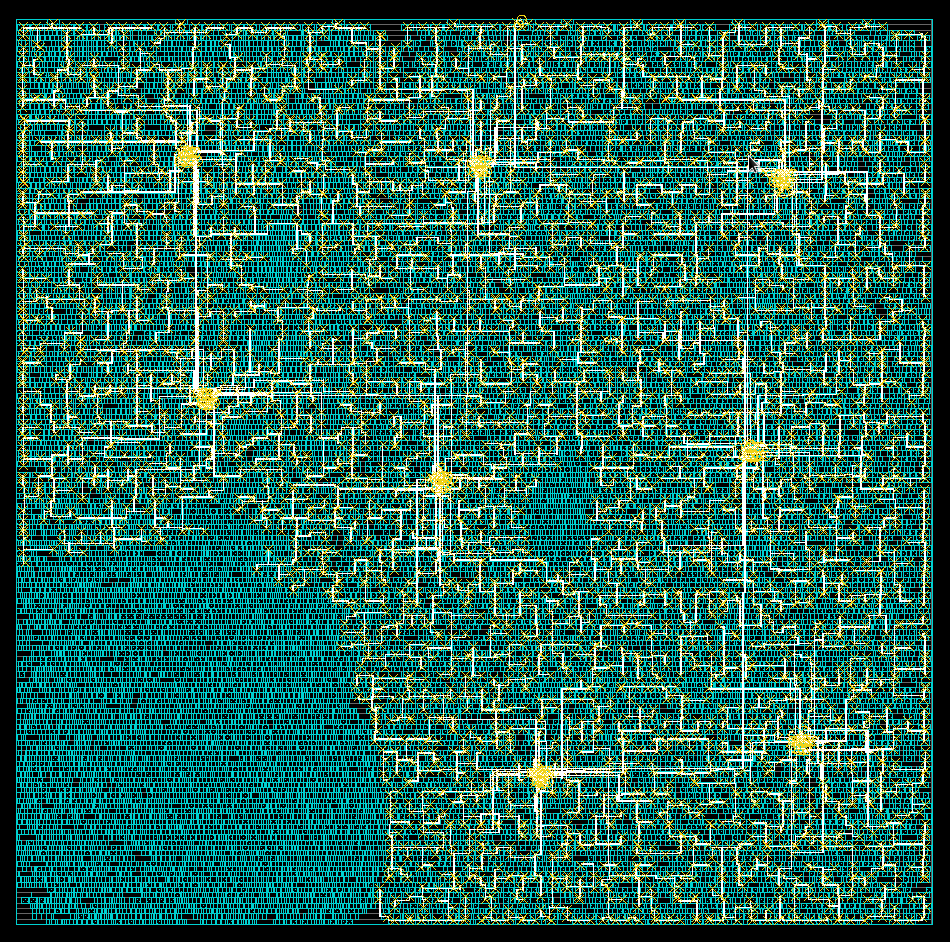


Figure 12: Regular CPU and Approximate CPU Clock Tree

Figure 12 above shows the clock trees of the approximate CPU (left) and the standard CPU (right). The clock tree has been distributed differently between both implementations, mainly due to the fact that our ALU now requires a clock signal for the approximation module.

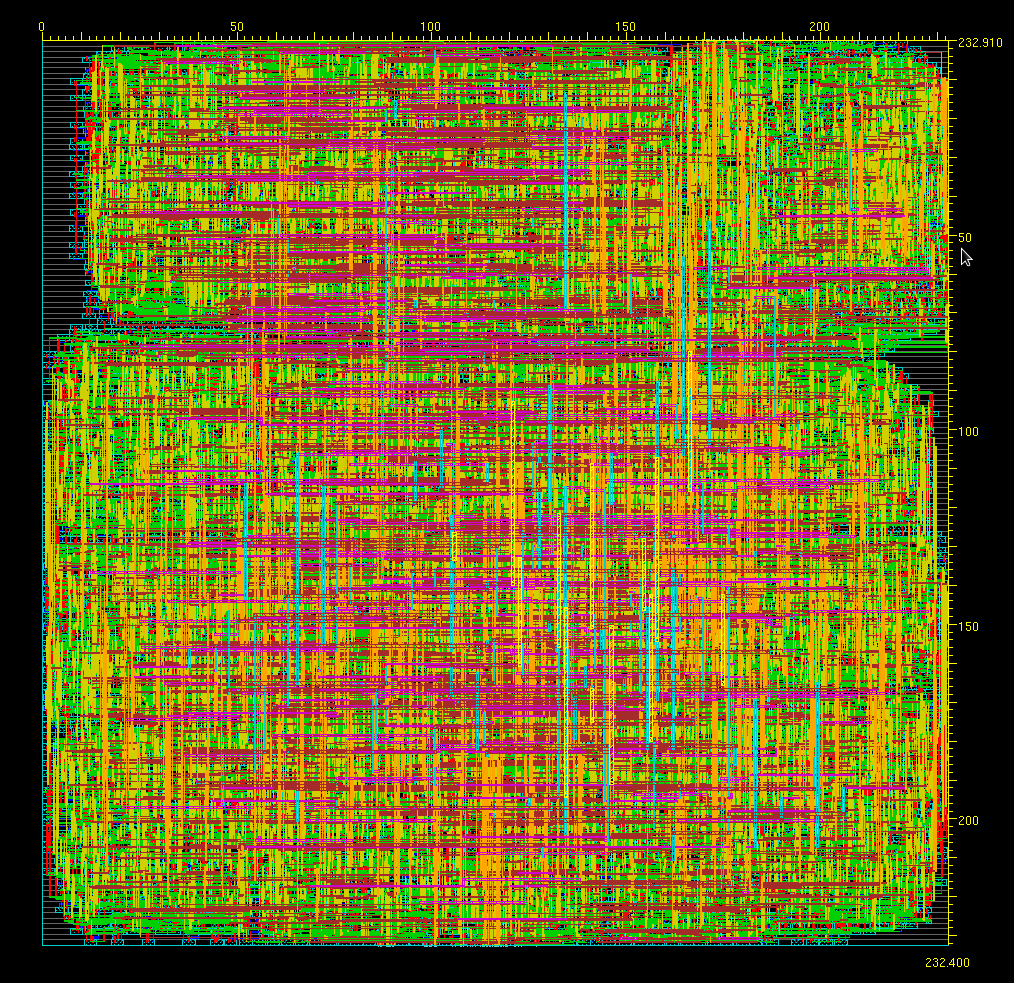


Figure 13: Approximate ALU with chip size

Figure 13 shows the synthesized chip design displaying the chip size of the entire chip. The chip size with approximation module is 232.910μm x 232.400μm, which is 0.54mm2 in area.

Figure 14 above shows the synthesized standard CPU chip with sizing. It is measured to be 243.590μm x 240.800μm, which is 0.58 mm2 in area. Compared to figure 13, the standard CPU is larger than the CPU with approximation module.

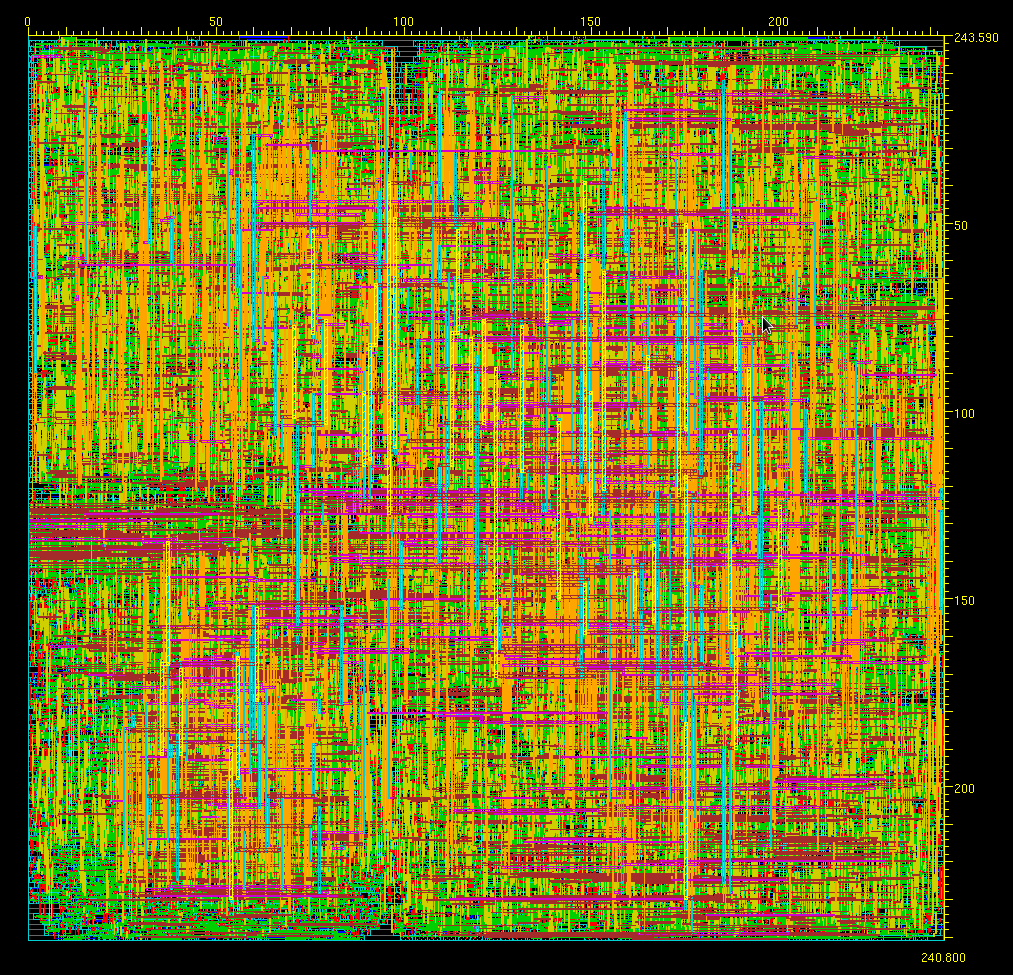


Figure 14: Standard CPU with chip size

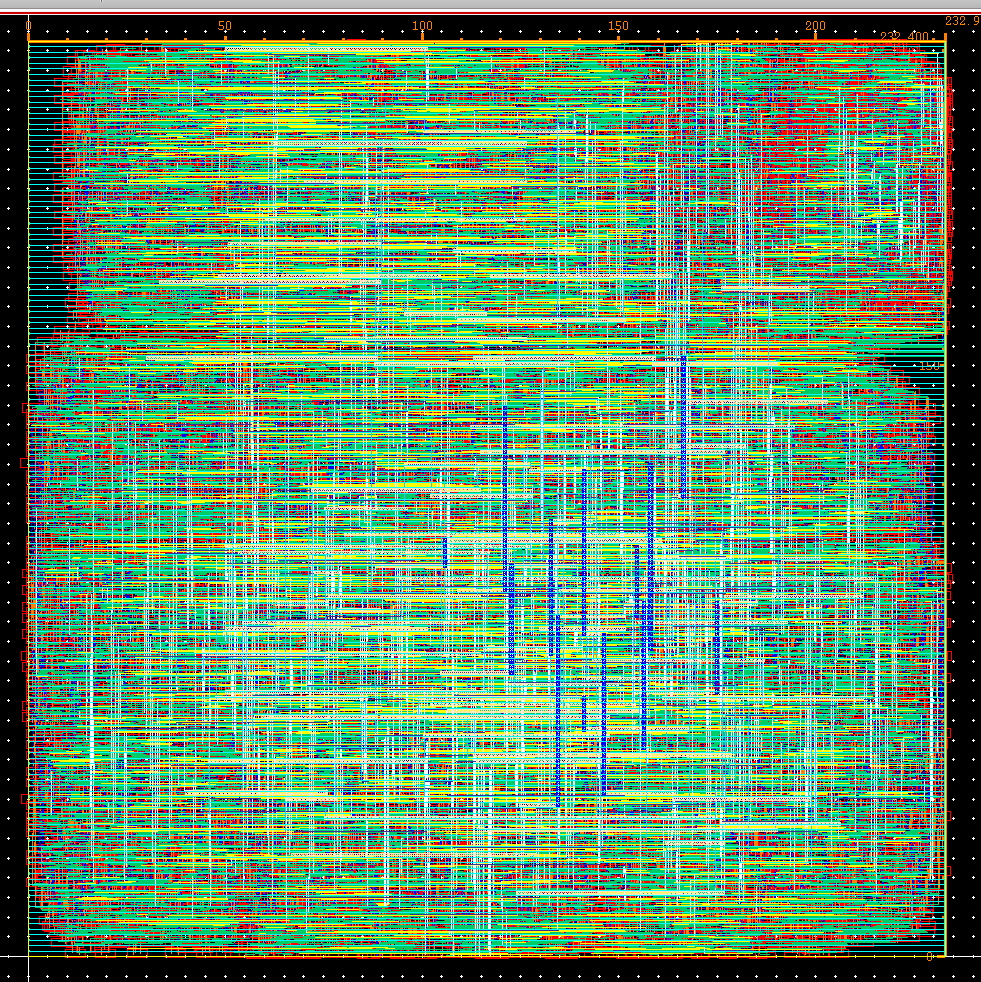
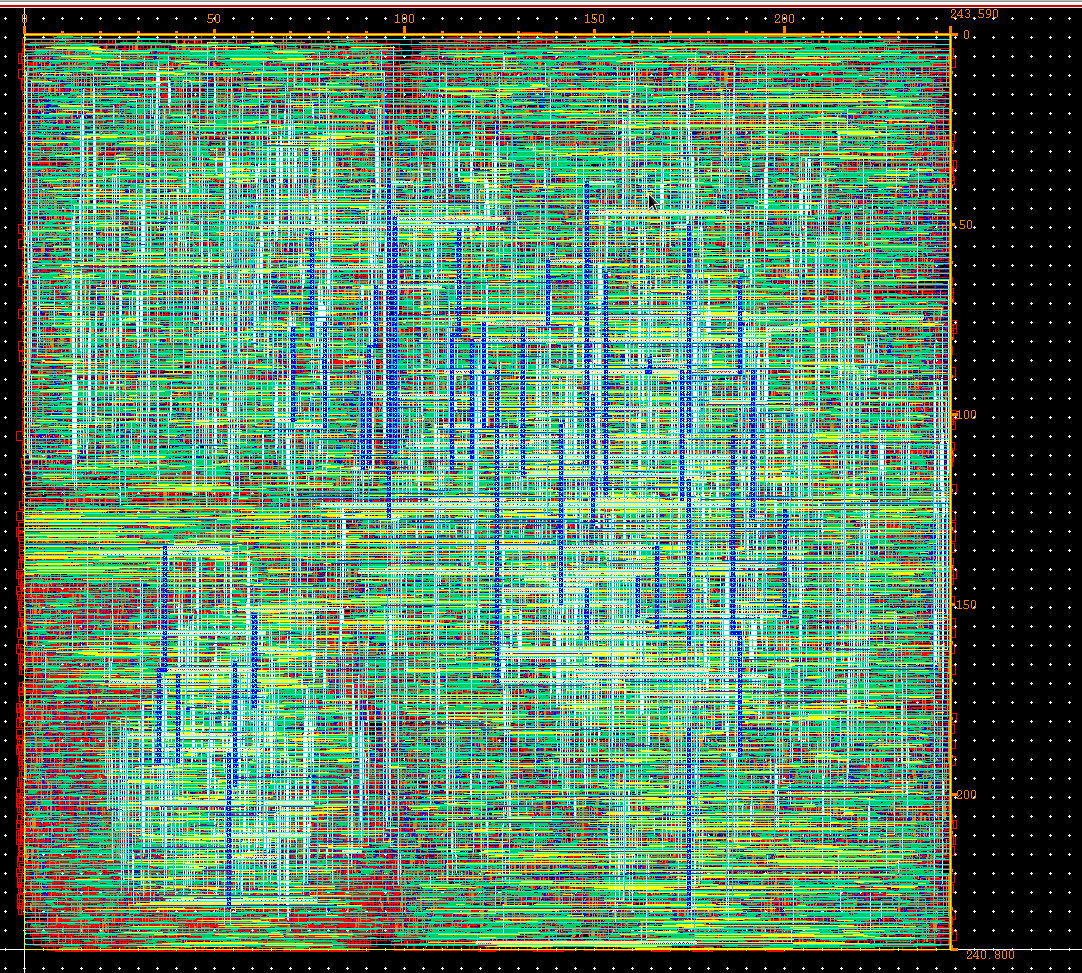


Figure 15: Regular CPU and Approximate CPU imported into Cadence Virtuoso Layout Editor

We have exported our chip design from encounter to try and dig deeper into how the cells are constructed by importing into virtuoso layout editor. The imported design in Cadence virtuoso can be seen above. We have confirmed the consistency in size between encounter and virtuoso, but unfortunately, not much else could be gained from importing into virtuoso. We had difficulty getting the wiring to display due to IBM 8hp process’ 7 metal layer limitation, while our chip design incorporated 10 metal layers from the NANgate045 techfile. Ultimately, we were able to see individual cell layouts that was incorporated with the Nangate library, but were unable to do any power analysis in virtuoso, since there was no netlist/schematic to verify and test design with.

**Conclusion:**

Hence, we have observed that our approximation algorithm can be implemented into a CPU chip design. Also it is observed that this method allows us to save on area.  
  
  
**Future work:**

Online tutorials from other colleges show that there is actually a method to implement a testbench to test the real power consumption, and hence savings of this circuit, but due to lack of a .vcd file we could not test that.

Although the savings of this method are relatively less because of the overall chip-size being large, it can be said that this will allow to have a lesser power consumption and area, and a much greater savings in a multi-core 64 bit implementation. Unfortunately, due to lack of hardware capable of simulating a 64 bit system, this cannot be verified, but initial results lead to the conclusion that it is true.

Also, if this can be implemented with a floating point operations algorithm, this could potentially be a huge improvement on computational processor manufacturing.

**Code**

Verliog Code

Top Module: Multicycle\_Datapath.v (No Approximation

`timescale 1ns / 1ps

`include "Control\_SM.v"

`include "General\_Modules.v"

`include "ALU.v"

`include "Memory\_File.v"

`include "Register\_File.v"

module Multicycle\_Datapath(clk, rst, start, I\_MEM\_Write\_Enable, I\_MEM\_Data\_In, I\_MEM\_Write\_Addr, ALUOut, PC\_out);

input clk, rst, start;

input I\_MEM\_Write\_Enable;

input [31:0] I\_MEM\_Data\_In;

input [15:0] I\_MEM\_Write\_Addr;

output [31:0] ALUOut, PC\_out;

wire MemRead, MemWrite, PC\_Reg\_Write, PC\_Reg\_Write\_BEQ, Instruction\_Reg\_Write, Memory\_Data\_Reg\_Write;

wire A\_Reg\_Write, B\_Reg\_Write, ALU\_Op\_Reg\_Write, Register\_File\_Write;

wire IorD\_Mux\_Select, Reg\_File\_B\_Mux\_Select, Write\_Data\_Mux\_Select, ALU\_A\_Mux\_Select, PC\_Source\_Mux\_Select;

wire [1:0] ALU\_B\_Mux\_Select;

wire [5:0] ALU\_Opcode;

wire [31:0] PC\_reg\_out, PC\_Source;

wire PC\_reg\_write\_enable, Zero;

wire [15:0] Mem\_Addr;

wire [31:0] Mem\_Out, Mem\_Data, Instr\_Out;

wire [31:0] Register\_Write\_Data;

wire [4:0] Reg\_File\_B\_Addr;

wire [31:0] A\_D\_in, A\_D\_out, B\_D\_in, B\_D\_out;

wire [31:0] ALU\_A, ALU\_B, ALU\_out, ALU\_reg\_out;

Control\_SM control(.OP\_Code(Instr\_Out[31:26]),

.MemWrite(MemWrite),

.PC\_Reg\_Write(PC\_Reg\_Write),

.PC\_Reg\_Write\_BEQ(PC\_Reg\_Write\_BEQ),

.Instruction\_Reg\_Write(Instruction\_Reg\_Write),

.Memory\_Data\_Reg\_Write(Memory\_Data\_Reg\_Write),

.A\_Reg\_Write(A\_Reg\_Write),

.B\_Reg\_Write(B\_Reg\_Write),

.ALU\_Op\_Reg\_Write(ALU\_Op\_Reg\_Write),

.Register\_File\_Write(Register\_File\_Write),

.IorD\_Mux\_Select(IorD\_Mux\_Select),

.Reg\_File\_B\_Mux\_Select(Reg\_File\_B\_Mux\_Select),

.Write\_Data\_Mux\_Select(Write\_Data\_Mux\_Select),

.ALU\_A\_Mux\_Select(ALU\_A\_Mux\_Select),

.ALU\_B\_Mux\_Select(ALU\_B\_Mux\_Select),

.PC\_Source\_Mux\_Select(PC\_Source\_Mux\_Select),

.ALU\_Opcode(ALU\_Opcode),

.clk(clk), .reset(rst), .start(start));

assign PC\_out = PC\_reg\_out;

assign ALUOut = ALU\_reg\_out;

assign PC\_reg\_write\_enable = PC\_Reg\_Write | (PC\_Reg\_Write\_BEQ & Zero);

FFwReset\_en PC\_reg(.In(PC\_Source), .Out(PC\_reg\_out), .clk(clk), .reset(rst),

.write\_enable(PC\_reg\_write\_enable));

input16\_mux Mem\_in(.A\_0(PC\_reg\_out), .B\_1(ALU\_reg\_out), .Select(IorD\_Mux\_Select), .Out(Mem\_Addr));

Memory\_File memory(.Address(Mem\_Addr), .WriteData(B\_D\_out),

.WriteEnable(MemWrite), .ReadData1(Mem\_Out), .clk(clk), .reset(rst),

.I\_MEM\_Write\_Enable(I\_MEM\_Write\_Enable), .I\_MEM\_Data\_In(I\_MEM\_Data\_In),

.I\_MEM\_Write\_Addr(I\_MEM\_Write\_Addr));

FFwReset\_en Mem\_Data\_Reg(.In(Mem\_Out), .Out(Mem\_Data), .clk(clk), .reset(rst),

.write\_enable(Memory\_Data\_Reg\_Write));

FFwReset\_en Instr\_Reg(.In(Mem\_Out), .Out(Instr\_Out), .clk(clk), .reset(rst),

.write\_enable(Instruction\_Reg\_Write));

input5\_mux Read\_B\_Select(.A\_0(Instr\_Out[15:11]), .B\_1(Instr\_Out[25:21]),

.Select(Reg\_File\_B\_Mux\_Select), .Out(Reg\_File\_B\_Addr));

input2\_mux Write\_Data(.A\_0(ALU\_reg\_out), .B\_1(Mem\_Data), .Select(Write\_Data\_Mux\_Select),

.Out(Register\_Write\_Data));

Register\_File reg\_file(.ReadSelect1(Instr\_Out[20:16]), .ReadSelect2(Reg\_File\_B\_Addr),

.WriteSelect(Instr\_Out[25:21]), .WriteData(Register\_Write\_Data),

.WriteEnable(Register\_File\_Write), .ReadData1(A\_D\_in), .ReadData2(B\_D\_in),

.clk(clk), .reset(rst));

FFwReset\_en Read\_A(.In(A\_D\_in), .Out(A\_D\_out), .clk(clk), .reset(rst), .write\_enable(A\_Reg\_Write));

FFwReset\_en Read\_B(.In(B\_D\_in), .Out(B\_D\_out), .clk(clk), .reset(rst), .write\_enable(B\_Reg\_Write));

input2\_mux ALU\_A\_select(.A\_0(A\_D\_out), .B\_1(PC\_reg\_out), .Select(ALU\_A\_Mux\_Select), .Out(ALU\_A));

input4\_mux ALU\_B\_select(.A\_0(B\_D\_out), .B\_1(1), .C\_2({16'h0, Instr\_Out[15:0]}),

.D\_3({{16{Instr\_Out[15]}},Instr\_Out[15:0]}), .Select(ALU\_B\_Mux\_Select), .Out(ALU\_B));

ALU alu(.A(ALU\_A), .B(ALU\_B), .Control(ALU\_Opcode), .Out(ALU\_out), .Zero(Zero));

input2\_mux Next\_PC\_select(.A\_0(ALU\_out), .B\_1(ALU\_reg\_out), .Select(PC\_Source\_Mux\_Select), .Out(PC\_Source));

FFwReset\_en ALU\_reg(.In(ALU\_out), .Out(ALU\_reg\_out), .clk(clk), .reset(rst),

.write\_enable(ALU\_Op\_Reg\_Write));

endmodule

General Modules.v

`timescale 1ns / 1ns

module input2\_mux(A\_0, B\_1, Select, Out);

parameter SIZE = 32;

input [SIZE-1:0] A\_0, B\_1;

input Select;

output [SIZE-1:0] Out;

// Select = 0: Out = A\_0

// Select = 1: Out = B\_1

assign Out = (Select) ? B\_1 : A\_0;

endmodule

module input16\_mux(A\_0, B\_1, Select, Out);

parameter SIZE = 32;

input [SIZE-1:0] A\_0, B\_1;

input Select;

output [15:0] Out;

// Select = 0: Out = A\_0

// Select = 1: Out = B\_1

assign Out = (Select) ? B\_1 : A\_0;

endmodule

module input5\_mux(A\_0, B\_1, Select, Out);

parameter SIZE = 5;

input [SIZE-1:0] A\_0, B\_1;

input Select;

output [SIZE-1:0] Out;

// Select = 0: Out = A\_0

// Select = 1: Out = B\_1

assign Out = (Select) ? B\_1 : A\_0;

endmodule

module input4\_mux(A\_0, B\_1, C\_2, D\_3, Select, Out);

parameter SIZE = 32;

input [SIZE-1:0] A\_0, B\_1, C\_2, D\_3;

input [1:0] Select;

output [SIZE-1:0] Out;

reg [SIZE-1:0] Out;

// Select = 0: Out = A\_0

// Select = 1: Out = B\_1

// Select = 2: Out = C\_2

// Select = 3: Out = D\_3

always @(A\_0 or B\_1 or C\_2 or D\_3 or Select)

begin

case (Select)

0 : Out = A\_0;

1 : Out = B\_1;

2 : Out = C\_2;

3 : Out = D\_3;

default : Out = 'bx;

endcase

end

endmodule

module FFwReset\_en(In, Out, clk, reset, write\_enable);

parameter SIZE = 32;

input [SIZE-1:0] In;

output [SIZE-1:0] Out;

reg [SIZE-1:0] Out;

input clk, reset, write\_enable;

always @(posedge clk)

begin

if (reset)

Out <= 0; // Clear output on reset

else

begin

if (write\_enable)

Out <= In; // Latch the input on clk

else

Out <= Out;

end

end

endmodule

Control\_SM.v

`timescale 1ns / 1ns

module Control\_SM(OP\_Code, MemWrite, PC\_Reg\_Write, PC\_Reg\_Write\_BEQ, Instruction\_Reg\_Write,

Memory\_Data\_Reg\_Write, A\_Reg\_Write, B\_Reg\_Write, ALU\_Op\_Reg\_Write,

Register\_File\_Write, IorD\_Mux\_Select, Reg\_File\_B\_Mux\_Select, Write\_Data\_Mux\_Select,

ALU\_A\_Mux\_Select, ALU\_B\_Mux\_Select, ALU\_Opcode, PC\_Source\_Mux\_Select, clk, reset, start);

input [5:0] OP\_Code;

input clk, reset, start;

output MemWrite, PC\_Reg\_Write, PC\_Reg\_Write\_BEQ, Instruction\_Reg\_Write, Memory\_Data\_Reg\_Write;

reg MemWrite, PC\_Reg\_Write, PC\_Reg\_Write\_BEQ, Instruction\_Reg\_Write, Memory\_Data\_Reg\_Write;

output A\_Reg\_Write, B\_Reg\_Write, ALU\_Op\_Reg\_Write, Register\_File\_Write;

reg A\_Reg\_Write, B\_Reg\_Write, ALU\_Op\_Reg\_Write, Register\_File\_Write;

output IorD\_Mux\_Select, Reg\_File\_B\_Mux\_Select, Write\_Data\_Mux\_Select, ALU\_A\_Mux\_Select, PC\_Source\_Mux\_Select;

reg IorD\_Mux\_Select, Reg\_File\_B\_Mux\_Select, Write\_Data\_Mux\_Select, ALU\_A\_Mux\_Select, PC\_Source\_Mux\_Select;

output [1:0] ALU\_B\_Mux\_Select;

reg [1:0] ALU\_B\_Mux\_Select;

output [5:0] ALU\_Opcode;

reg [5:0] ALU\_Opcode;

reg [3:0] State, NextState;

// Go to next state every clock tick

always @(posedge clk)

begin

if (reset)

State <= 0;

else

if (start)

State <= NextState;

else

State <= State;

end

// NextState Logic

always @(OP\_Code or State or NextState or reset or start)

begin

case(State)

0:

begin

NextState = 4'b0001;

end

4'b0001:

begin

case (OP\_Code)

6'h10, 6'h12, 6'h13, 6'h14, 6'h15,

6'h32, 6'h33, 6'h34, 6'h35, 6'h39: NextState = 4'b0110;

6'h20: NextState = 4'b1000;

6'h3B, 6'h3C: NextState = 4'b0010;

default: NextState = 0;

endcase

end

4'b0010:

begin

case (OP\_Code)

6'h3B: NextState = 4'b0011;

6'h3C: NextState = 4'b0101;

default: NextState = 0;

endcase

end

4'b0011:

begin

NextState = 4'b0100;

end

4'b0100:

begin

NextState = 0;

end

4'b0101:

begin

NextState = 0;

end

4'b0110:

begin

NextState = 4'b0111;

end

4'b0111:

begin

NextState = 0;

end

4'b1000:

begin

NextState = 0;

end

default: NextState = 0;

endcase

end

// Moore state control logic

always @(OP\_Code or State or NextState or reset or start)

begin

MemWrite <= 0;

PC\_Reg\_Write <= 0;

PC\_Reg\_Write\_BEQ <= 0;

Instruction\_Reg\_Write <= 0;

Memory\_Data\_Reg\_Write <= 0;

A\_Reg\_Write <= 0;

B\_Reg\_Write <= 0;

ALU\_Op\_Reg\_Write <= 0;

Register\_File\_Write <= 0;

IorD\_Mux\_Select <= 0;

Reg\_File\_B\_Mux\_Select <= 0;

Write\_Data\_Mux\_Select <= 0;

ALU\_A\_Mux\_Select <= 0;

ALU\_B\_Mux\_Select <= 0;

PC\_Source\_Mux\_Select <= 0;

ALU\_Opcode <= OP\_Code;

if (start)

begin

case(State)

0:

begin

// Get the next instruction and increment PC

Instruction\_Reg\_Write <= 1;

ALU\_A\_Mux\_Select <= 1;

ALU\_B\_Mux\_Select <= 1;

ALU\_Opcode <= 6'h12;

PC\_Reg\_Write <= 1;

end

4'b0001:

begin

// Send instruction register to register file and increment PC

// with immediate for branch instructions

if (OP\_Code[5] == 1)

Reg\_File\_B\_Mux\_Select <= 1;

A\_Reg\_Write <= 1;

B\_Reg\_Write <= 1;

ALU\_A\_Mux\_Select <= 1;

ALU\_B\_Mux\_Select <= 2;

ALU\_Opcode <= 6'h12;

ALU\_Op\_Reg\_Write <= 1;

end

4'b0010:

begin

// Store Memory location in ALU Register

ALU\_B\_Mux\_Select <= 2;

ALU\_Op\_Reg\_Write <= 1;

end

4'b0011:

begin

// Read memory and store in data register

IorD\_Mux\_Select <= 1;

Memory\_Data\_Reg\_Write <= 1;

end

4'b0100:

begin

// Write Memory into register file

Write\_Data\_Mux\_Select <= 1;

Register\_File\_Write <= 1;

end

4'b0101:

begin

// Write register into memory address

IorD\_Mux\_Select <= 1;

MemWrite <= 1;

end

4'b0110:

begin

// Perform ALU operation save into ALU reg

if (OP\_Code[5] == 1)

begin

if (OP\_Code == 6'h32 || OP\_Code == 6'h33)

ALU\_B\_Mux\_Select <= 3;

ALU\_B\_Mux\_Select <= 2;

end

ALU\_Op\_Reg\_Write <= 1;

end

4'b0111:

begin

// Write back ALU reg into register file

Register\_File\_Write <= 1;

end

4'b1000:

begin

// Check branch condition and change pc if true

ALU\_Opcode <= 6'h20;

PC\_Reg\_Write\_BEQ <= 1;

PC\_Source\_Mux\_Select <= 1;

end

default:

begin

end

endcase

end

end

endmodule

ALU.v (no Approximation)

`timescale 1ns / 1ns

module ALU(A, B, Control, Out, Zero);

parameter SIZE = 32;

input [SIZE-1:0] A, B;

input [5:0] Control;

output [SIZE-1:0] Out;

reg [SIZE-1:0] Out;

output Zero;

reg Zero;

always @(A or B or Control)

begin

case (Control)

6'h0: Out = 0; //NOOP do nothing

6'h10 : Out = A; //MOV : R1 = R2

6'h39, 6'h3B, 6'h3C : Out = B; //LI : R1 = IMM

6'h12, 6'h32 : Out = A + B; //ADD : R1 = R2 + R3

6'h13, 6'h20, 6'h33 : Out = A - B; //SUB : R1 = R2 - R3

6'h14, 6'h34 : Out = A | B; //OR : R1 = R2 | R3

6'h15, 6'h35 : Out = A \* B; //MUL : R1 = R2 & R3

default : Out = 'bx; //Default : R1 = x

endcase

end

// New Zero signal which activates if output is equal to zero

always @(A or B or Control)

begin

if (Out == 0)

Zero <= 1;

else

Zero <= 0;

end

endmodule

Memory\_File.v

`timescale 1ns / 1ns

module Memory\_File(Address, WriteData, WriteEnable, ReadData1, clk, reset,

I\_MEM\_Write\_Enable, I\_MEM\_Data\_In, I\_MEM\_Write\_Addr);

parameter BITSIZE = 32;

parameter REGSIZE = 16;

input [REGSIZE-1:0] Address;

input [BITSIZE-1:0] WriteData;

input WriteEnable;

output [BITSIZE-1:0] ReadData1;

reg [BITSIZE-1:0] ReadData1;

input clk, reset;

//#(.BITSIZE(32), .REGSIZE(65536))

//ADDED for testbench from TA

input I\_MEM\_Write\_Enable;

input [31:0] I\_MEM\_Data\_In;

input [15:0] I\_MEM\_Write\_Addr;

reg [BITSIZE-1:0] memory\_file [0:100-1]; // Entire list of memory

// Asyncronous read of memory. Was not specified.

always @(Address or memory\_file[Address])

begin

ReadData1 = memory\_file[Address];

end

// Write back to memory on clk edge.

always @(posedge clk)

begin

if (WriteEnable)

memory\_file[Address] <= WriteData; //If writeback is enabled and not 0 register.

if (I\_MEM\_Write\_Enable)

memory\_file[I\_MEM\_Write\_Addr] <= I\_MEM\_Data\_In;

end

//MY method of filling the memory instead of through a test bench

//initial

// begin

// $readmemb("memory.txt", memory\_file);

// end

endmodule

Register\_File.v

`timescale 1ns / 1ns

module Register\_File(ReadSelect1, ReadSelect2, WriteSelect, WriteData, WriteEnable, ReadData1, ReadData2, clk, reset);

parameter BITSIZE = 32;

parameter REGSIZE = 32;

input [4:0] ReadSelect1, ReadSelect2, WriteSelect;

input [BITSIZE-1:0] WriteData;

input WriteEnable;

output [BITSIZE-1:0] ReadData1, ReadData2;

reg [BITSIZE-1:0] ReadData1, ReadData2;

input clk, reset;

reg [BITSIZE-1:0] reg\_file [REGSIZE-1:0]; // Entire list of registers

integer i; // Used below to reset all registers

// Asyncronous read of register file.

always @(ReadSelect1 or reg\_file[ReadSelect1])

begin

ReadData1 = reg\_file[ReadSelect1];

end

// Asyncronous read of register file.

always @(ReadSelect2 or reg\_file[ReadSelect2])

begin

ReadData2 = reg\_file[ReadSelect2];

end

// Write back to register file on clk edge.

always @(posedge clk)

begin

if (reset)

for (i=0; i<REGSIZE; i=i+1) reg\_file[i] <= 'b0; // Reset all registers

else

begin

if (WriteEnable && WriteSelect != 0)

reg\_file[WriteSelect] <= WriteData; //If writeback is enabled and not 0 register.

end

end

endmodule

With Approximation Module, only changes to top module and ALU code.

Multicycle\_Datapath.v (with Approx)

`timescale 1ns / 1ps

`include "Control\_SM.v"

`include "General\_Modules.v"

`include "ALU.v"

`include "Memory\_File.v"

`include "Register\_File.v"

module Multicycle\_Datapath(clk, rst, start, I\_MEM\_Write\_Enable, I\_MEM\_Data\_In, I\_MEM\_Write\_Addr, ALUOut, PC\_out);

input clk, rst, start;

input I\_MEM\_Write\_Enable;

input [31:0] I\_MEM\_Data\_In;

input [15:0] I\_MEM\_Write\_Addr;

output [31:0] ALUOut, PC\_out;

wire MemRead, MemWrite, PC\_Reg\_Write, PC\_Reg\_Write\_BEQ, Instruction\_Reg\_Write, Memory\_Data\_Reg\_Write;

wire A\_Reg\_Write, B\_Reg\_Write, ALU\_Op\_Reg\_Write, Register\_File\_Write;

wire IorD\_Mux\_Select, Reg\_File\_B\_Mux\_Select, Write\_Data\_Mux\_Select, ALU\_A\_Mux\_Select, PC\_Source\_Mux\_Select;

wire [1:0] ALU\_B\_Mux\_Select;

wire [5:0] ALU\_Opcode;

wire [31:0] PC\_reg\_out, PC\_Source;

wire PC\_reg\_write\_enable, Zero;

wire [15:0] Mem\_Addr;

wire [31:0] Mem\_Out, Mem\_Data, Instr\_Out;

wire [31:0] Register\_Write\_Data;

wire [4:0] Reg\_File\_B\_Addr;

wire [31:0] A\_D\_in, A\_D\_out, B\_D\_in, B\_D\_out;

wire [31:0] ALU\_A, ALU\_B, ALU\_out, ALU\_reg\_out;

Control\_SM control(.OP\_Code(Instr\_Out[31:26]),

.MemWrite(MemWrite),

.PC\_Reg\_Write(PC\_Reg\_Write),

.PC\_Reg\_Write\_BEQ(PC\_Reg\_Write\_BEQ),

.Instruction\_Reg\_Write(Instruction\_Reg\_Write),

.Memory\_Data\_Reg\_Write(Memory\_Data\_Reg\_Write),

.A\_Reg\_Write(A\_Reg\_Write),

.B\_Reg\_Write(B\_Reg\_Write),

.ALU\_Op\_Reg\_Write(ALU\_Op\_Reg\_Write),

.Register\_File\_Write(Register\_File\_Write),

.IorD\_Mux\_Select(IorD\_Mux\_Select),

.Reg\_File\_B\_Mux\_Select(Reg\_File\_B\_Mux\_Select),

.Write\_Data\_Mux\_Select(Write\_Data\_Mux\_Select),

.ALU\_A\_Mux\_Select(ALU\_A\_Mux\_Select),

.ALU\_B\_Mux\_Select(ALU\_B\_Mux\_Select),

.PC\_Source\_Mux\_Select(PC\_Source\_Mux\_Select),

.ALU\_Opcode(ALU\_Opcode),

.clk(clk), .reset(rst), .start(start));

assign PC\_out = PC\_reg\_out;

assign ALUOut = ALU\_reg\_out;

assign PC\_reg\_write\_enable = PC\_Reg\_Write | (PC\_Reg\_Write\_BEQ & Zero);

FFwReset\_en PC\_reg(.In(PC\_Source), .Out(PC\_reg\_out), .clk(clk), .reset(rst),

.write\_enable(PC\_reg\_write\_enable));

input16\_mux Mem\_in(.A\_0(PC\_reg\_out), .B\_1(ALU\_reg\_out), .Select(IorD\_Mux\_Select), .Out(Mem\_Addr));

Memory\_File memory(.Address(Mem\_Addr), .WriteData(B\_D\_out),

.WriteEnable(MemWrite), .ReadData1(Mem\_Out), .clk(clk), .reset(rst),

.I\_MEM\_Write\_Enable(I\_MEM\_Write\_Enable), .I\_MEM\_Data\_In(I\_MEM\_Data\_In),

.I\_MEM\_Write\_Addr(I\_MEM\_Write\_Addr));

FFwReset\_en Mem\_Data\_Reg(.In(Mem\_Out), .Out(Mem\_Data), .clk(clk), .reset(rst),

.write\_enable(Memory\_Data\_Reg\_Write));

FFwReset\_en Instr\_Reg(.In(Mem\_Out), .Out(Instr\_Out), .clk(clk), .reset(rst),

.write\_enable(Instruction\_Reg\_Write));

input5\_mux Read\_B\_Select(.A\_0(Instr\_Out[15:11]), .B\_1(Instr\_Out[25:21]),

.Select(Reg\_File\_B\_Mux\_Select), .Out(Reg\_File\_B\_Addr));

input2\_mux Write\_Data(.A\_0(ALU\_reg\_out), .B\_1(Mem\_Data), .Select(Write\_Data\_Mux\_Select),

.Out(Register\_Write\_Data));

Register\_File reg\_file(.ReadSelect1(Instr\_Out[20:16]), .ReadSelect2(Reg\_File\_B\_Addr),

.WriteSelect(Instr\_Out[25:21]), .WriteData(Register\_Write\_Data),

.WriteEnable(Register\_File\_Write), .ReadData1(A\_D\_in), .ReadData2(B\_D\_in),

.clk(clk), .reset(rst));

FFwReset\_en Read\_A(.In(A\_D\_in), .Out(A\_D\_out), .clk(clk), .reset(rst), .write\_enable(A\_Reg\_Write));

FFwReset\_en Read\_B(.In(B\_D\_in), .Out(B\_D\_out), .clk(clk), .reset(rst), .write\_enable(B\_Reg\_Write));

input2\_mux ALU\_A\_select(.A\_0(A\_D\_out), .B\_1(PC\_reg\_out), .Select(ALU\_A\_Mux\_Select), .Out(ALU\_A));

input4\_mux ALU\_B\_select(.A\_0(B\_D\_out), .B\_1(1), .C\_2({16'h0, Instr\_Out[15:0]}),

.D\_3({{16{Instr\_Out[15]}},Instr\_Out[15:0]}), .Select(ALU\_B\_Mux\_Select), .Out(ALU\_B));

ALU alu(.A(ALU\_A), .B(ALU\_B), .Control(ALU\_Opcode), .Out(ALU\_out), .Zero(Zero), .clk(clk), .reset(rst));

input2\_mux Next\_PC\_select(.A\_0(ALU\_out), .B\_1(ALU\_reg\_out), .Select(PC\_Source\_Mux\_Select), .Out(PC\_Source));

FFwReset\_en ALU\_reg(.In(ALU\_out), .Out(ALU\_reg\_out), .clk(clk), .reset(rst),

.write\_enable(ALU\_Op\_Reg\_Write));

endmodule

ALU.v (with approximation)

`timescale 1ns / 1ns

`include "approx\_mul.v"

module ALU(A, B, Control, Out, Zero,clk, reset);

parameter SIZE = 32;

input [SIZE-1:0] A, B;

input [5:0] Control;

output [SIZE-1:0] Out;

reg [SIZE-1:0] Out;

output Zero;

reg Zero;

input clk,reset;

wire [31:0]out\_mul;

reg [15:0]A1,B1;

always @(A or B or Control)

begin

case (Control)

6'h0: Out = 0; //NOOP do nothing

6'h10 : Out = A; //MOV : R1 = R2

6'h39, 6'h3B, 6'h3C : Out = B; //LI : R1 = IMM

6'h12, 6'h32 : Out = A + B; //ADD : R1 = R2 + R3

6'h13, 6'h20, 6'h33 : Out = A - B; //SUB : R1 = R2 - R3

6'h14, 6'h34 : Out = A | B; //OR : R1 = R2 | R3

6'h15, 6'h35 : Out = A & B; //AND : R1 = R2 & R3

6'h16 : begin

A1 = A[15:0];

B1 = B[15:0];

end

default : Out = 'bx; //Default : R1 = x

endcase

end

approx\_mul approx\_mul(clk, reset, A1, B1, out\_mul);

always @ (A or B or Control)

begin

if (Control == 6'h16)

Out = out\_mul;

end

// New Zero signal which activates if output is equal to zero

always @(A or B or Control)

begin

if (Out == 0)

Zero <= 1;

else

Zero <= 0;

end

endmodule

Approx\_mul.v

`timescale 1ns / 1ps

module approx\_mul(clk, reset, in\_a, in\_b, out\_mul);

input clk, reset;

input [15:0] in\_a, in\_b;

reg [15:0] out\_A, out\_B;

reg [4:0] shift\_a,shift\_b;

reg [4:0] shift\_A, shift\_B;

reg [4:0] shifta,shiftb;

output [31:0] out\_mul;

reg [31:0] out\_mul;

reg [31:0] out\_nmul;

always@(posedge clk)

begin

//getting the length of input A

if (reset)

begin

out\_A = 0;

out\_B = 0;

end

else

begin

if(in\_a[15] == 1'b1)

shift\_a = 5'd16;

else if (in\_a[14] == 1'b1)

shift\_a = 5'd15;

else if (in\_a[13] == 1'b1)

shift\_a = 5'd14;

else if (in\_a[12] == 1'b1)

shift\_a = 5'd13;

else if (in\_a[11] == 1'b1)

shift\_a = 5'd12;

else if (in\_a[10] == 1'b1)

shift\_a = 5'd11;

else if (in\_a[9] == 1'b1)

shift\_a = 5'd10;

else if (in\_a[8] == 1'b1)

shift\_a = 5'd9;

else if (in\_a[7] == 1'b1)

shift\_a = 5'd8;

else if (in\_a[6] == 1'b1)

shift\_a = 5'd7;

else if (in\_a[5] == 1'b1)

shift\_a = 5'd6;

else if (in\_a[4] == 1'b1)

shift\_a = 5'd5;

else if (in\_a[3] == 1'b1)

shift\_a = 5'd4;

else if (in\_a[2] == 1'b1)

shift\_a = 5'd3;

else if (in\_a[1] == 1'b1)

shift\_a = 5'd2;

else if (in\_a[0] == 1'b1)

shift\_a = 5'd1;

else

shift\_a = 5'd0;

//getting the length of input B

if(in\_b[15] == 1'b1)

shift\_b = 5'd16;

else if (in\_b[14] == 1'b1)

shift\_b = 5'd15;

else if (in\_b[13] == 1'b1)

shift\_b = 5'd14;

else if (in\_b[12] == 1'b1)

shift\_b = 5'd13;

else if (in\_b[11] == 1'b1)

shift\_b = 5'd12;

else if (in\_b[10] == 1'b1)

shift\_b = 5'd11;

else if (in\_b[9] == 1'b1)

shift\_b = 5'd10;

else if (in\_b[8] == 1'b1)

shift\_b = 5'd9;

else if (in\_b[7] == 1'b1)

shift\_b = 5'd8;

else if (in\_b[6] == 1'b1)

shift\_b = 5'd7;

else if (in\_b[5] == 1'b1)

shift\_b = 5'd6;

else if (in\_b[4] == 1'b1)

shift\_b = 5'd5;

else if (in\_b[3] == 1'b1)

shift\_b = 5'd4;

else if (in\_a[2] == 1'b1)

shift\_b = 5'd3;

else if (in\_b[1] == 1'b1)

shift\_b = 5'd2;

else if (in\_b[0] == 1'b1)

shift\_b = 5'd1;

else

shift\_b = 5'd0;

// Shifting the bits to create new outputs

// shift\_A = 5'd16 - shift\_a;

// shift\_B = 5'd16 - shift\_b;

//shift\_a and shift\_b have the msb

//finding the bits to be shifted in inputA

case(shift\_a)

5'd16 : shift\_A = 5'd9; // if 16 bit number drop 8 bits from the lsb

5'd15 : shift\_A = 5'd8;

5'd14 : shift\_A = 5'd7;

5'd13 : shift\_A = 5'd6;

5'd12 : shift\_A = 5'd5;

5'd11 : shift\_A = 5'd4;

5'd10 : shift\_A = 5'd3;

5'd9 : shift\_A = 5'd2;

5'd8 : shift\_A = 5'd1;

default : shift\_A = 5'd0;

endcase

//finding the bits to be shifted in inputB

case(shift\_b)

5'd16 : shift\_B = 5'd9; // if 16 bit number drop 8 bits from the lsb

5'd15 : shift\_B = 5'd8;

5'd14 : shift\_B = 5'd7;

5'd13 : shift\_B = 5'd6;

5'd12 : shift\_B = 5'd5;

5'd11 : shift\_B = 5'd4;

5'd10 : shift\_B = 5'd3;

5'd9 : shift\_B = 5'd2;

5'd8 : shift\_B = 5'd1;

default : shift\_B = 5'd0;

endcase

end

out\_A = in\_a;

out\_B = in\_b;

case (shift\_A)

5'd9 : out\_A[8:0] = 0;

5'd8 : out\_A[7:0] = 0;

5'd7 : out\_A[6:0] = 0;

5'd6 : out\_A[5:0] = 0;

5'd5 : out\_A[4:0] = 0;

5'd4 : out\_A[3:0] = 0;

5'd3 : out\_A[2:0] = 0;

5'd2 : out\_A[1:0] = 0;

5'd1 : out\_A[0] = 0;

default : out\_A = in\_a;

endcase

case (shift\_B)

5'd9 : out\_B[8:0] = 0;

5'd8 : out\_B[7:0] = 0;

5'd7 : out\_B[6:0] = 0;

5'd6 : out\_B[5:0] = 0;

5'd5 : out\_B[4:0] = 0;

5'd4 : out\_B[3:0] = 0;

5'd3 : out\_B[2:0] = 0;

5'd2 : out\_B[1:0] = 0;

5'd1 : out\_B[0] = 0;

default : out\_B = in\_b;

endcase

out\_mul = out\_A \* out\_B;

out\_nmul = in\_a \* in\_b;

end

endmodule