Hugo Trivino
ECE 429-01.19S: VLSI Design
Due Date: May 6, 2019
Experiment Date: April 5-May 6, 2019

Final Project: Design and Synthesis of Central Processing Units

I Abstract

This project consist in the expansion of a CPU that perform 8-bit operations into a CPU that supports 32-bit operations. operations include addition, subtraction, shift right logical, shift left logical, and, exclusive nor, load immediate, jump if zero, and jump if not zero. The carry-ripple architecture of the adder had to be modified since the overhead was excessive for 32-bits. I implemented a look-ahead adder architecture. I also implemented an integer divider with a 32-bit dividend and 16-bit divisor since it was a client requirement.

II Introduction

During the first stage of this project I first synthesize an 8-bit CPU that supported 16-bit instructions, and during the second stage I modify the original design in order to work with 32-bits registers instead of 8-bit. The operations had to be modified to support 32-bits as well. The architecture of the carry-ripple adder was modified since the delay produced by the serialization of carry-in to carry-out along a 32 full adders was inefficient. For this process as I show later, I implemented a look-ahead adder architecture that was significantly faster. The subtraction operation is deeply intertwined with the addition operation since it is only an addition between a complementary representation of the sustained. The implementation of the shift right and left consisted basically of the inclusion of a 8-bits shift and 16-bits in cascade from the 1,2,4 bits that were already implemented in the 8-bit CPU. The exclusive NOR and the AND implementations were fairly straightforward since it was required only to extend the code bit by bit to 32. In order to implement the jump if zero, I added all bits from the register to evaluate within a OR statement in order to set the **jz** bit. The jump if not zero was not modified since it is calculated as a negation of the **jz** bit. Finally, the most challenging implementation was to create an integer divider using only the functions that were already implemented. I will go into details about the overall implementation in the sections to follow.

III Background

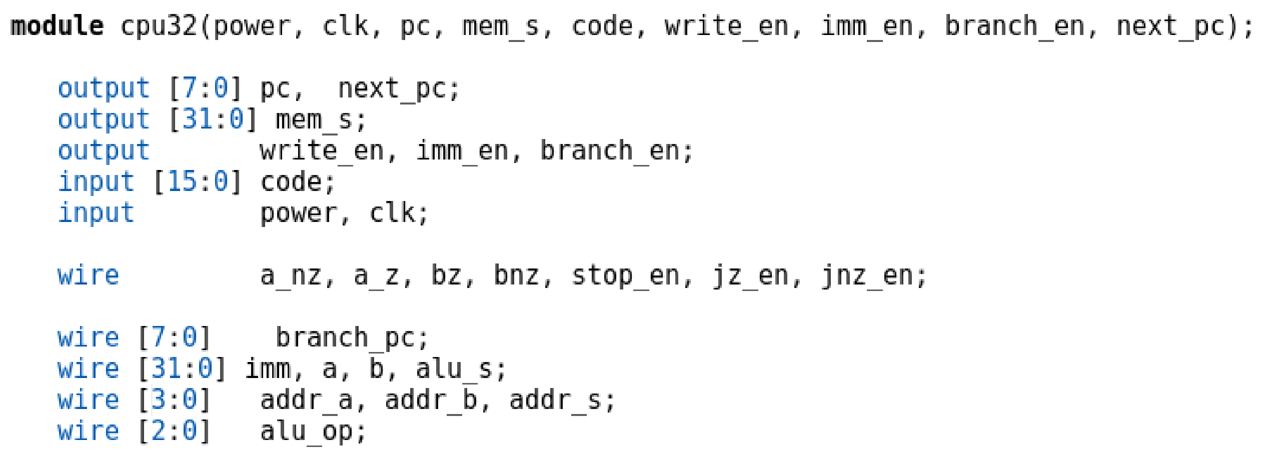
A central processing unit is probably the most important component of a computer because is the component that provides computability to the system. There are some registers to store some data or addresses from which computation would be stored or load into, an arithmetic logic unit to perform the operations on the operands , a control unit that decodes the instruction into the selection of operations performed by the ALU together with the decode of other control signals to correctly select the destination of the output from the ALU or further outcomes from the instruction such as branching, and a program counter to keep the address of the instruction being executed or to be executed. For this particular project, the architecture that we are provided perform operations on 8-bit registers [1]. The operations allowed are shift left logic, shift right logic, add, subtract, and, xnor, load immediate, jump if zero, jump if not zero, and stop execution. The extension of these instructions to 32 bits was not particularly difficult for the most part. However, the implementation of a different adder architecture was required since the delay of a carry-ripple. For this reason, I found that the easiest to understand and implement in my case was the look-ahead adder architecture as shown in 11.2 of the textbook [2]. Finally, as a bonus I implemented an integer divider that performs a division that uses a 32-bit dividend and a 16-bit divisor. The results are return in a 32 bit register that will contain the quotient in the lower 16-bits and the remainder in the upper 16-bits in the style of the Motorola 68k architecture [3]. I will use the standard cell based synthesis approach to transform the Verilog RTL description into a netlist of the gates and combinational logic required for the implementation of the 32 bit CPU. The logical equivalence of the synthesis will be checked at every step. After that, I will compile the RTL into a code that uses standard cells which are building blocks with previously known delays, power, and area of the cells. Following the previous step, I will connect the standard cells by using Encounter. I will test this equivalence by using Synopsys formal verification between the final interconnected design of the CPU and the original code. Finally, I will create a physical layout implementation of the CPU by using virtuoso. At the end I will test the cpu by running test programs to show how to how to perform several operations, how to multiply two numbers, and how to divide.

IV 32-bit CPU Implementation

In order to extend the 8-bit CPU into a 32-bit CPU my approach was as follows:

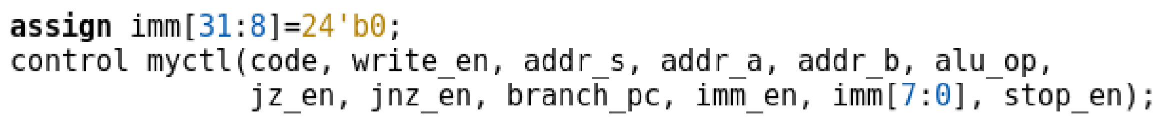
- I started from the top level, the last module of the file named cpu_8. I decided to modify the module entirely before changing its name, so although the cpu_8 was the first module to start modify, it was the last module to which I changed the name.

- For each line of that code if I saw an element of the data-path, I changed to 32 bits as seen in this case the **mem_s , imm, a,b ,alu_s**. Without modifying anything from the PC logic.



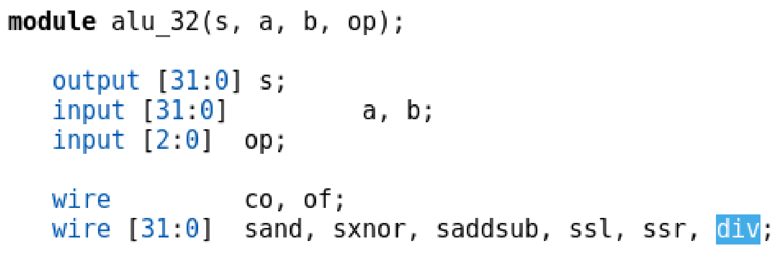
```
module cpu32(power, clk, pc, mem_s, code, write_en, imm_en, branch_en, next_pc);

   output [7:0] pc,  next_pc;
   output [31:0] mem_s;
   output        write_en, imm_en, branch_en;
   input [15:0] code;
   input         power, clk;

   wire          a_nz, a_z, bz, bnz, stop_en, jz_en, jnz_en;

   wire [7:0]     branch_pc;
   wire [31:0] imm, a, b, alu_s;
   wire [3:0]    addr_a, addr_b, addr_s;
   wire [2:0]    alu_op;
```

- The first module that had some contact with the data-path was the control unit. I decided not to modify the control unit but rather to only pass the lower byte of the **imm**. Note that I grounded the upper 24-bits.



```
assign imm[31:8]=24'b0;
control myctl(code, write_en, addr_s, addr_a, addr_b, alu_op,
             jz_en, jnz_en, branch_pc, imm_en, imm[7:0], stop_en);
```

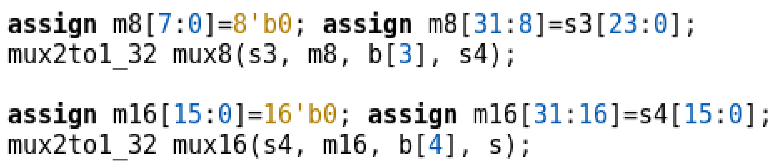
- The next module that had contact with the data-path was **alu_8**. I had to modify this module. The first modification was to change the input and output of the data path to 32-bits (not the control signals). Note that in this image I also include the **div** which will be described in more detail later.



```
module alu_32(s, a, b, op);

   output [31:0] s;
   input [31:0]        a, b;
   input [2:0]  op;

   wire          co, of;
   wire [31:0]  sand, sxnor, saddsub, ssl, ssr, div;
```

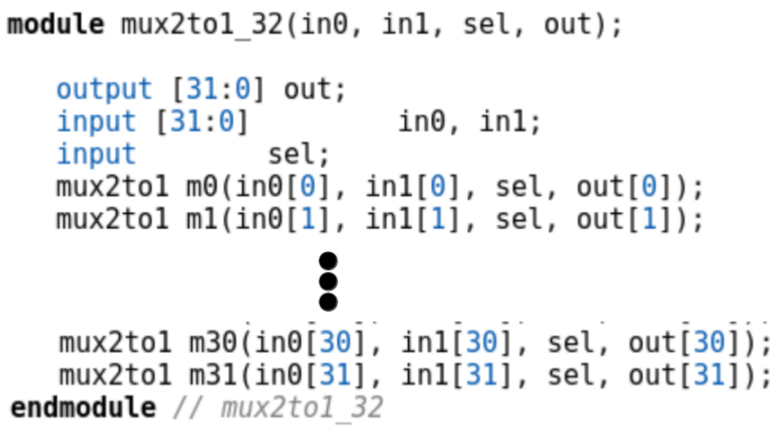
The ALU contained the definition of the operations and I went to each of the modules changing them one by one: **shift_left_logic_8** , **shift_right_logic_8** , **addsub_8** , **and_8, xnor_8** . After I modified these modules I change their names to 32 instead of 8 one by one.

- **shift_left_logic_8** was implemented by shifting 1, then 2, then 4 bits to the left to the input and only selecting the shifted input sequentially by using a mux controlled by the **b[0],b[1],b[2]** respectively. For example if **b[2:0]** = **'101'**, then the **a** will be shifted 1 bit and that output would be pass without alterations to the 2 bit shifter and it will be shifted again by the 4 bit shifter in order to shift **a** a total of 5 bits. Once I understood this logic, I added an 8 bit shifter and 16 bits shifter to be added in series with the original ones as shown in the following code:



```
assign m8[7:0]=8'b0; assign m8[31:8]=s3[23:0];
mux2to1_32 mux8(s3, m8, b[3], s4);

assign m16[15:0]=16'b0; assign m16[31:16]=s4[15:0];
mux2to1_32 mux16(s4, m16, b[4], s);
```

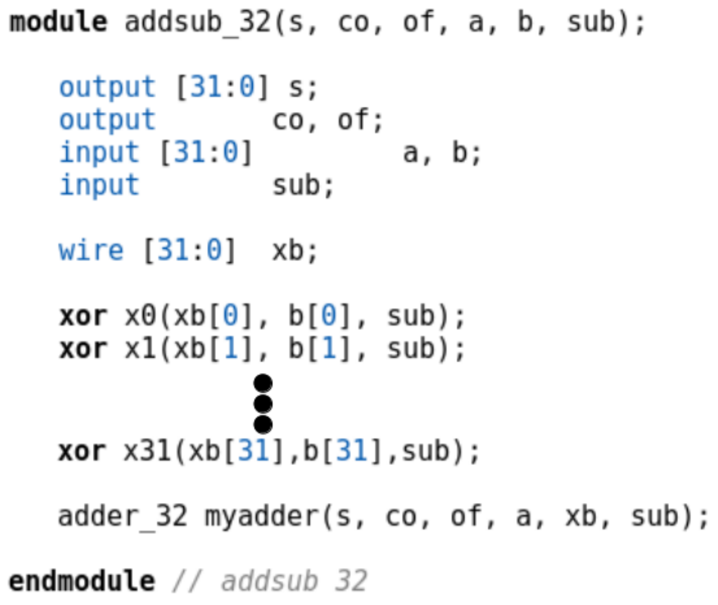
It might be important to mention that the mux itself had to be expanded to 32 bits This implementation of the mux is trivial. I just added more lines for each bit as shown in the following image.



```
module mux2to1_32(in0, in1, sel, out);

   output [31:0] out;
   input [31:0]        in0, in1;
   input        sel;
   mux2to1 m0(in0[0], in1[0], sel, out[0]);
   mux2to1 m1(in0[1], in1[1], sel, out[1]);
                      ●
                      ●
                      ●
   mux2to1 m30(in0[30], in1[30], sel, out[30]);
   mux2to1 m31(in0[31], in1[31], sel, out[31]);
endmodule // mux2to1_32
```

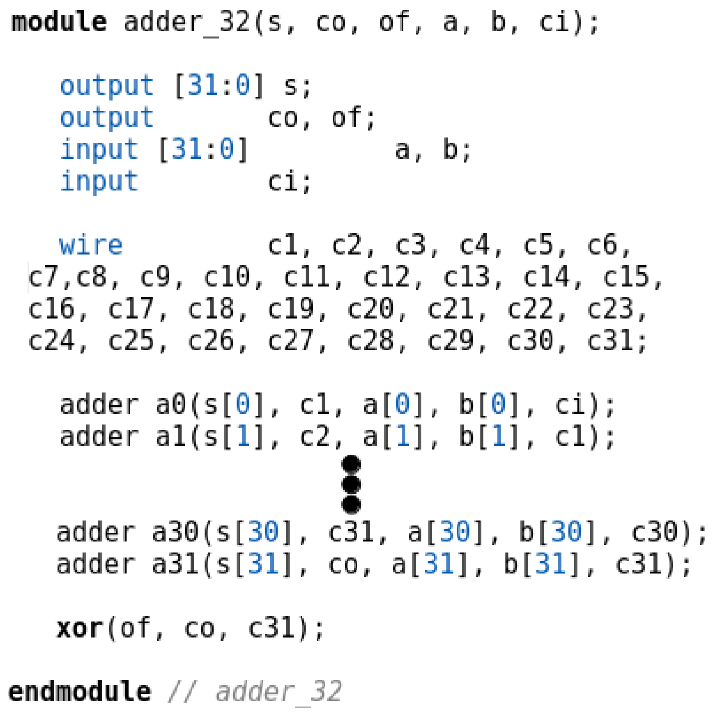
- After modifying the **shift_left_logic_8** into a **shift_left_logic_32**, the modification of the **shift_right_logic_8** into **shift_right_logic_32** was trivial.

- The next function to modify was the **addsub** function. This function just complements the value of **b** in the case of a **sub** operation and keep it as it is in an **add** operation. I just extended this step to 32 bits as follows:



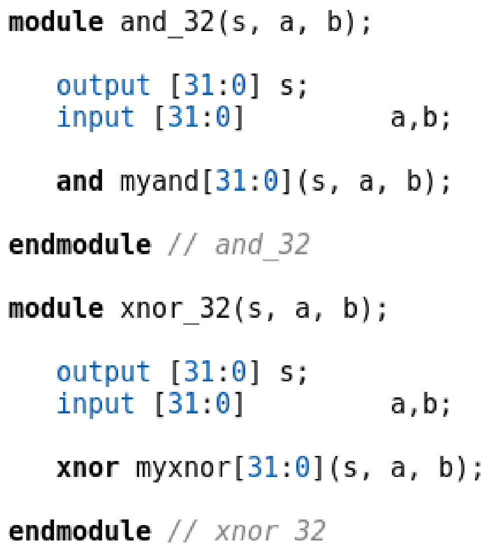
```
module addsub_32(s, co, of, a, b, sub);

   output [31:0] s;
   output        co, of;
   input [31:0]        a, b;
   input         sub;

   wire [31:0]  xb;

   xor x0(xb[0], b[0], sub);
   xor x1(xb[1], b[1], sub);
             ⋮
   xor x31(xb[31],b[31],sub);

   adder_32 myadder(s, co, of, a, xb, sub);

endmodule // addsub 32
```

- The next step was to extend the **adder_8** to 32 bits. This is the naive carry-ripple implementation, the look-ahead implementation will be treated in a separate chapter. The implementation of the adder is trivial as it was only an extension bit by bit of the 8 bits implementation. The only caveat is the usage of the **ci** in the carry-in from the full-adder, to use the carry out of the last full-adder as **co**, and finally, the use of the **c31** for the **xor** that decided the overflow bit. These can be shown in the following figure:



```
module adder_32(s, co, of, a, b, ci);

   output [31:0] s;
   output        co, of;
   input [31:0]        a, b;
   input         ci;

   wire          c1, c2, c3, c4, c5, c6,
 c7,c8, c9, c10, c11, c12, c13, c14, c15,
 c16, c17, c18, c19, c20, c21, c22, c23,
 c24, c25, c26, c27, c28, c29, c30, c31;

   adder a0(s[0], c1, a[0], b[0], ci);
   adder a1(s[1], c2, a[1], b[1], c1);
                    ⋮
   adder a30(s[30], c31, a[30], b[30], c30);
   adder a31(s[31], co, a[31], b[31], c31);

   xor(of, co, c31);

endmodule // adder_32
```

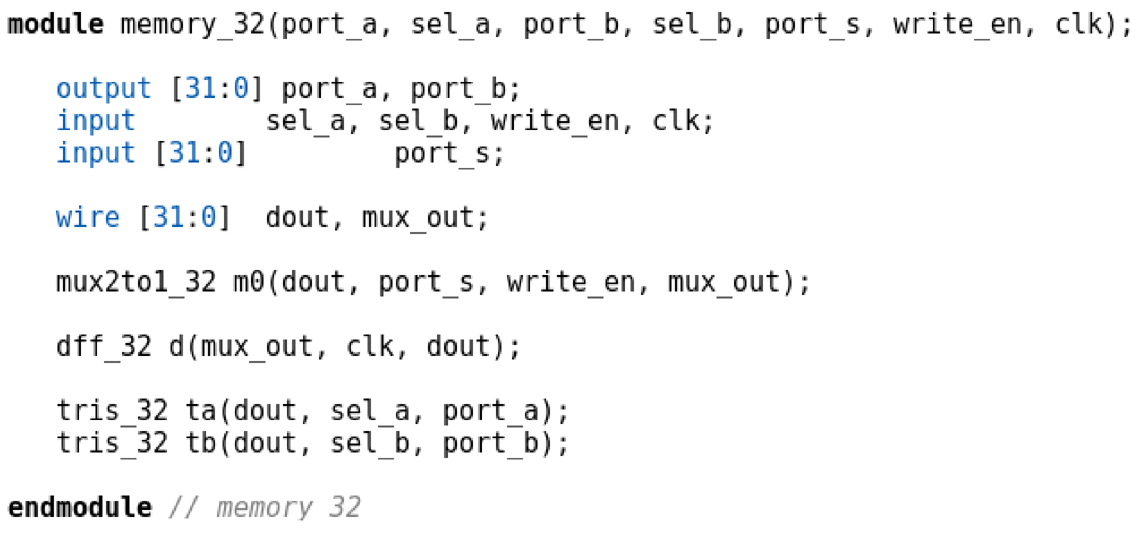
- The implementation of the **xnor** and **and** was probably the easiest to be extended since these instructions are supported natively within Verilog and don't require a separate implementation.



```
module and_32(s, a, b);

   output [31:0] s;
   input [31:0]         a,b;

   and myand[31:0](s, a, b);

endmodule // and_32

module xnor_32(s, a, b);

   output [31:0] s;
   input [31:0]         a,b;

   xnor myxnor[31:0](s, a, b);

endmodule // xnor_32
```

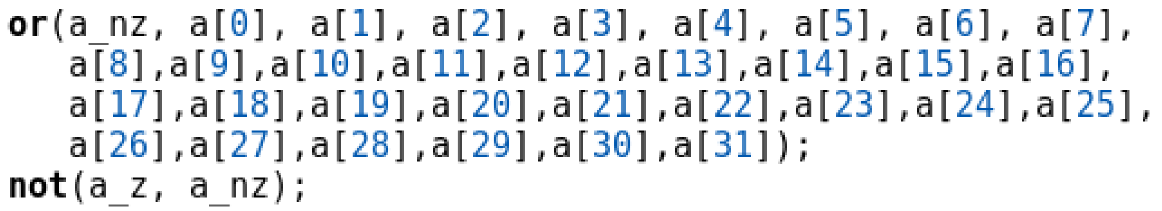
- After modifying **shift_left_logic_8** , **shift_right_logic_8** , **addsub_8** , **and_8, xnor_8** , I changed the names to **shift_left_logic_32** , **shift_right_logic_32** , **addsub_32** , **and_32, xnor_32** . And I changed the name of the **alu_8** to **alu_32**.

- However, the CPU was not entirely done since the memory was still an 8 bit memory. Therefore the next step was modifying **memory16_8**. In the **memory_8** itself I only modify the input and output ports to be 32 bits. However, there was a module called **memory_8** that was being instantiated. I modified **memory_8** creating **memory_32**. The only thing that I had to do was to extend a simple D-flip-flop and the tris function to 32 bits. Both of which did not represent a challenge at all. As we know by now, I have already created a **mux2to1_32**.



```
module memory_32(port_a, sel_a, port_b, sel_b, port_s, write_en, clk);

   output [31:0] port_a, port_b;
   input         sel_a, sel_b, write_en, clk;
   input [31:0]          port_s;

   wire [31:0]  dout, mux_out;

   mux2to1_32 m0(dout, port_s, write_en, mux_out);

   dff_32 d(mux_out, clk, dout);

   tris_32 ta(dout, sel_a, port_a);
   tris_32 tb(dout, sel_b, port_b);

endmodule // memory_32
```

- The last instruction that had to be modify in order to get the 32-bit cpu working was the jump if not zero. This function was implemented by checking 8 bits of the memory register and having an **or** between all of them. I used the same logic but extended to 32 bits as shown below:



```
or(a_nz, a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7],
   a[8],a[9],a[10],a[11],a[12],a[13],a[14],a[15],a[16],
   a[17],a[18],a[19],a[20],a[21],a[22],a[23],a[24],a[25],
   a[26],a[27],a[28],a[29],a[30],a[31]);
not(a_z, a_nz);
```

  Note that the jump if zero is just the complement of the jump if not zero bit as shown in the figure above.

- Now when all these instructions have been modified, I changed the **cpu_8** to **cpu_32**. I deleted all modules that were modified with the exceptions of those that were referenced in an unmodified portion of the code, like in the case of **tris_8, mux2to1_8** that were referenced in the PC logic.

V Architectural Exploration of Adders

In this section I will describe my implementation for the adder module. Since the carry-ripple adder took far too much time since the carry bit is completely serialized through the 32 bits, I decided to implement a carry look-ahead adder since all the other architectures of adders were too complex. Most architectures divided the bits into groups (four or eight) and added many gates and/or mux in order to obtain comparatively similar results that the ones obtained by the carry look-ahead adder. However, the carry look-ahead adder works with far simpler principles both to understand and to implement.

a) Propagation

$$P_i = A_i \oplus B_i$$

b) Generation

$$G_i = A_i B_i$$

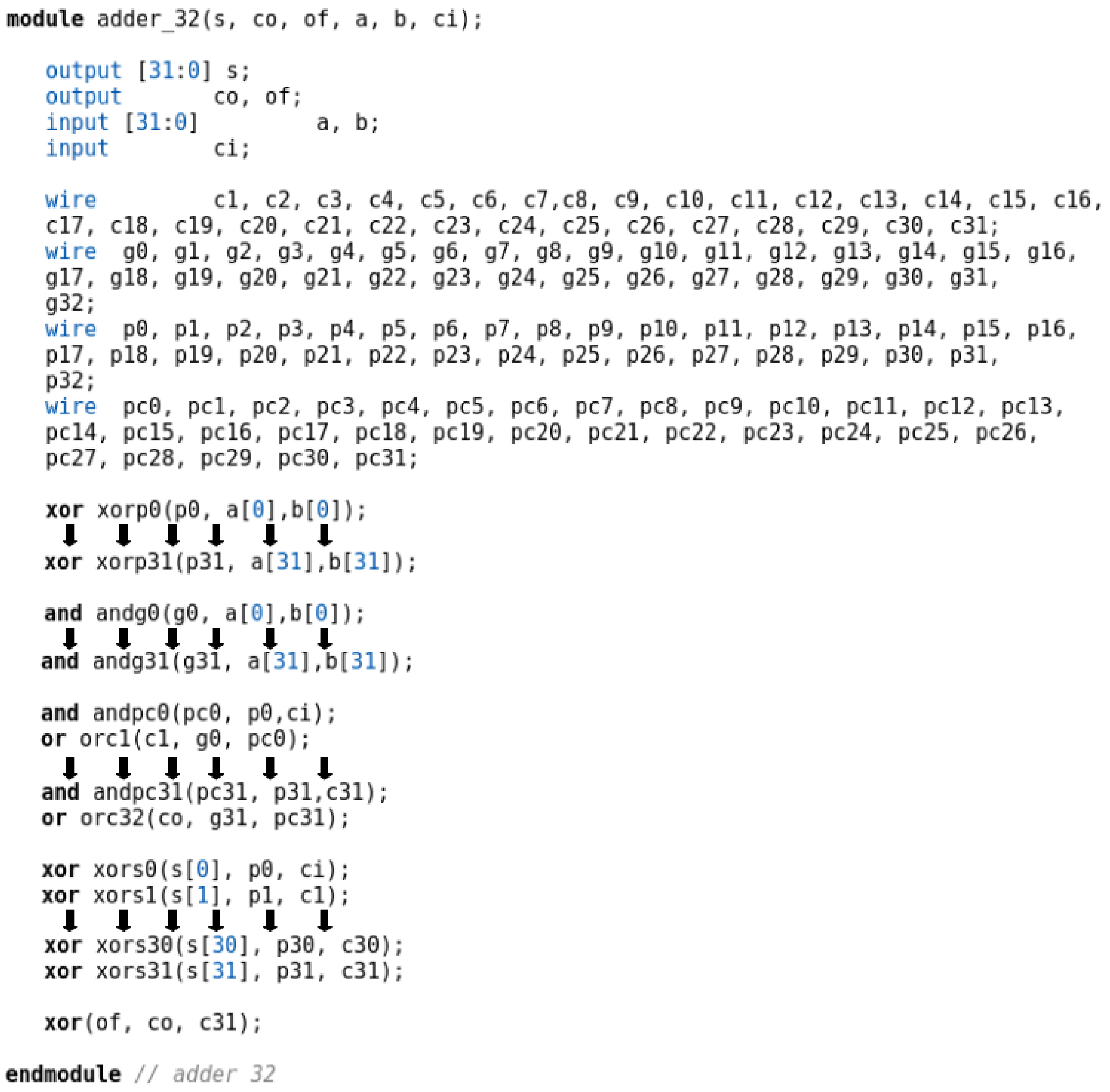
c) Carry

$$C_{i+1} = G_i + P_i C_i$$

d) Sum

$$S_i = P_i \oplus C_i$$

I implement each component in the same order as I listed them above. In the following code, the arrows indicate that the lines in between are the same but changing the index from 0 to 31. Note that each carry bit requires two lines because the P_iC_i has to be calculated separately from the G_i.



```
module adder_32(s, co, of, a, b, ci);

   output [31:0] s;
   output        co, of;
   input [31:0]         a, b;
   input         ci;

   wire         c1, c2, c3, c4, c5, c6, c7,c8, c9, c10, c11, c12, c13, c14, c15, c16,
   c17, c18, c19, c20, c21, c22, c23, c24, c25, c26, c27, c28, c29, c30, c31;
   wire  g0, g1, g2, g3, g4, g5, g6, g7, g8, g9, g10, g11, g12, g13, g14, g15, g16,
   g17, g18, g19, g20, g21, g22, g23, g24, g25, g26, g27, g28, g29, g30, g31,
   g32;
   wire  p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11, p12, p13, p14, p15, p16,
   p17, p18, p19, p20, p21, p22, p23, p24, p25, p26, p27, p28, p29, p30, p31,
   p32;
   wire  pc0, pc1, pc2, pc3, pc4, pc5, pc6, pc7, pc8, pc9, pc10, pc11, pc12, pc13,
   pc14, pc15, pc16, pc17, pc18, pc19, pc20, pc21, pc22, pc23, pc24, pc25, pc26,
   pc27, pc28, pc29, pc30, pc31;

   xor xorp0(p0, a[0],b[0]);
    ⬇   ⬇   ⬇  ⬇   ⬇   ⬇
   xor xorp31(p31, a[31],b[31]);

   and andg0(g0, a[0],b[0]);
    ⬇   ⬇   ⬇  ⬇   ⬇   ⬇
   and andg31(g31, a[31],b[31]);

   and andpc0(pc0, p0,ci);
   or orc1(c1, g0, pc0);
    ⬇   ⬇   ⬇  ⬇   ⬇   ⬇
   and andpc31(pc31, p31,c31);
   or orc32(co, g31, pc31);

   xor xors0(s[0], p0, ci);
   xor xors1(s[1], p1, c1);
    ⬇   ⬇   ⬇  ⬇   ⬇   ⬇
   xor xors30(s[30], p30, c30);
   xor xors31(s[31], p31, c31);

   xor(of, co, c31);

endmodule // adder 32
```

The slowest operation of all was the subtraction since the input had to be complemented before the actual addition. The performance of this operation was almost doubled by the implementation of the carry look-ahead architecture from 20 ns to about 10 ns.

VI Division Support

In order to support division, I implemented a long division algorithm as follows:

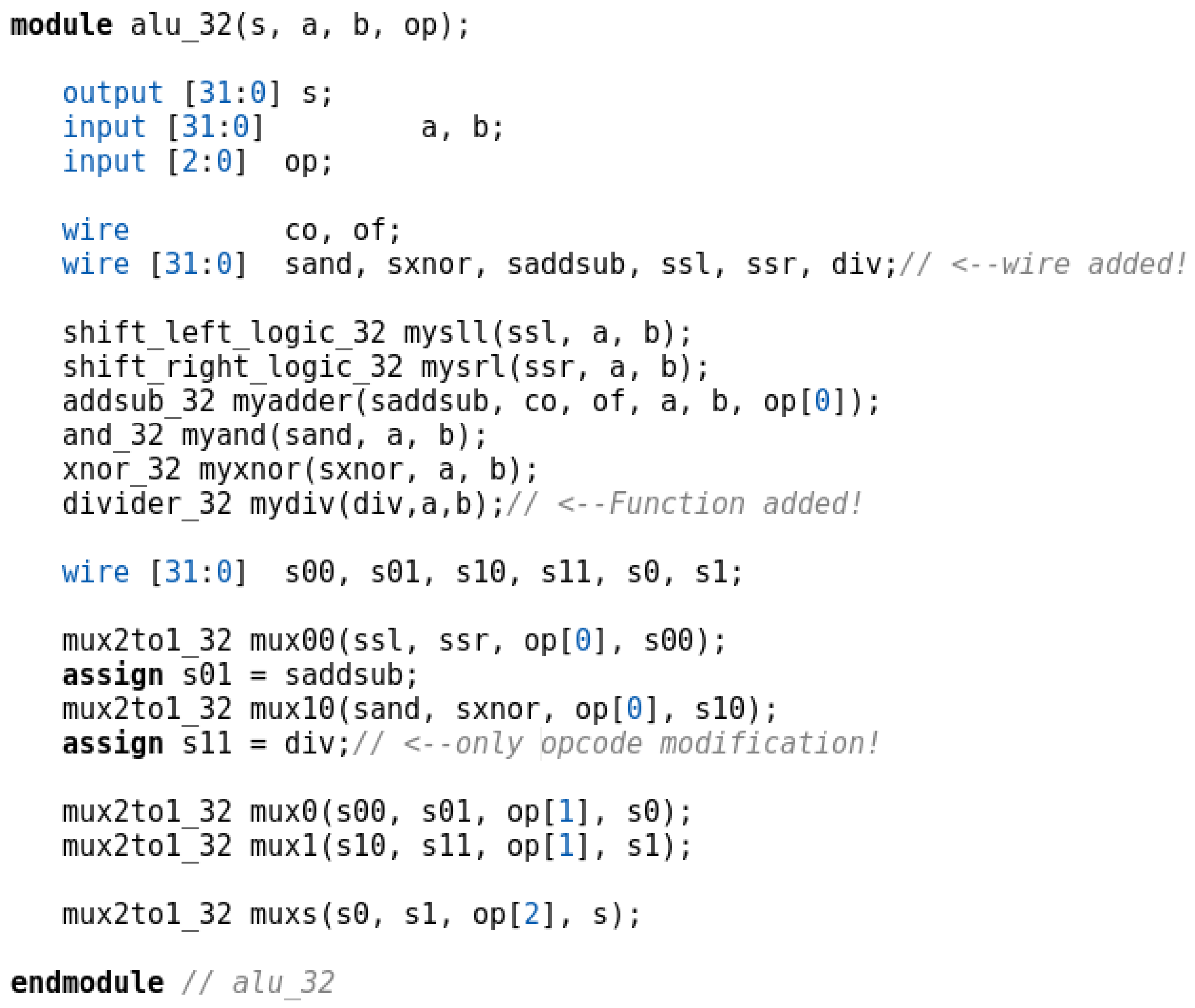
1 Subtract the divisor from the upper 16 bits of the dividend.

2 If the result is negative, the bit 15 of the quotient is set. Also, the new upper 16 bits of the dividend are replaced in place by the dividend - divisor.

3 Repeat using divisor and upper 17 bits, 18 ... to 32 bits setting the bits 15 to 0 of the quotient.

4 At the end of the previous steps, the result of the last dividend minus the divisor is the integer remainder and is stored in the upper 16 bits of the destination register.

The new instruction named **div** was introduced. The opcode was **0110** since it was now occupied by any other opcode. The modifications in the ALU are shown by an arrow <−. Fol

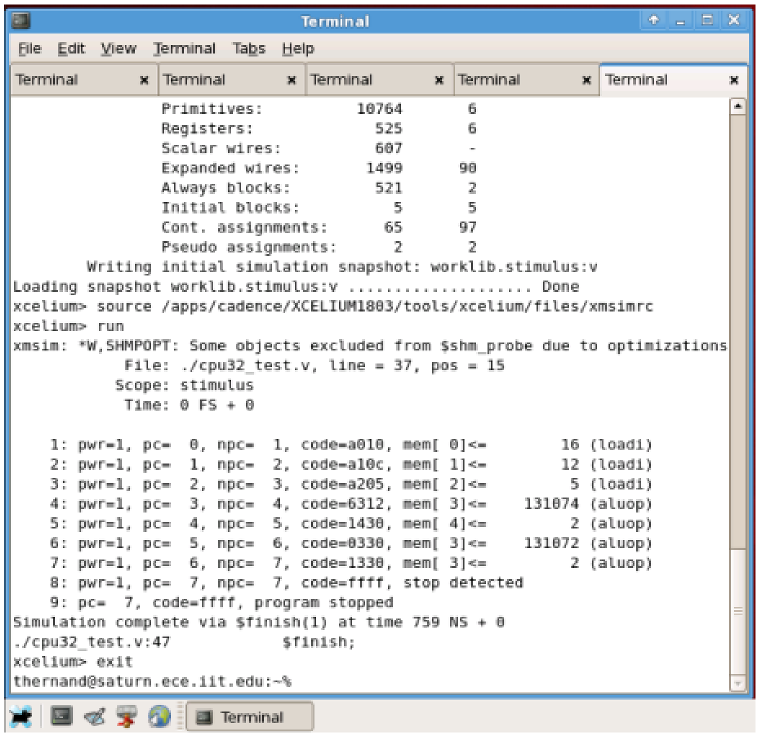


```
module alu_32(s, a, b, op);

   output [31:0] s;
   input [31:0]         a, b;
   input [2:0]  op;

   wire         co, of;
   wire [31:0]  sand, sxnor, saddsub, ssl, ssr, div;// <--wire added!

   shift_left_logic_32 mysll(ssl, a, b);
   shift_right_logic_32 mysrl(ssr, a, b);
   addsub_32 myadder(saddsub, co, of, a, b, op[0]);
   and_32 myand(sand, a, b);
   xnor_32 myxnor(sxnor, a, b);
   divider_32 mydiv(div,a,b);// <--Function added!

   wire [31:0]  s00, s01, s10, s11, s0, s1;

   mux2to1_32 mux00(ssl, ssr, op[0], s00);
   assign s01 = saddsub;
   mux2to1_32 mux10(sand, sxnor, op[0], s10);
   assign s11 = div;// <--only opcode modification!

   mux2to1_32 mux0(s00, s01, op[1], s0);
   mux2to1_32 mux1(s10, s11, op[1], s1);

   mux2to1_32 muxs(s0, s1, op[2], s);

endmodule // alu_32
```

Implementation of **divider_32** followed by Validation of 12/5, q=2, r=2

```
module divider_32(s, a, b);

   output [31:0] s;
   input [31:0]        a, b;

   wire [31:0] m0, m1, m2, m3, m4, m5, m6, m7, m8, m9, m10, m11, m12, m13, m14, m15;
   wire [31:0] s0,s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15 ;
   wire [31:0] r0,r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12, r13, r14, r15 ;
   wire c0,c1, c2, c3, c4, c5, c6, c7, c8, c9, c10, c11, c12, c13, c14, c15 ;
   wire [31:0] a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16;
   wire of0, of1, of2, of3, of4, of5, of6, of7, of8, of9, of10, of11, of12, of13, of14, of15;

   wire one,zero;
   wire sub;
   assign sub =1'b1;
   assign one = 1'b1;
   assign zero = 1'b0;


assign s0[16:0]=a[31:15]; assign s0[31:17]=15'b0;
addsub_32 sub17(r0, c0, of0, s0, b,  sub);
assign m0[31:15]=r0[16:0]; assign m0[14:0]=a[14:0];
mux2to1_32 mux17(a,  m0,c0, a1);
mux2to1 o16(zero, one, c0, s[15]);
   ⬇   ⬇   ⬇   ⬇   ⬇   ⬇   ⬇    ⬇     ⬇
assign s14[30:0]=a14[31:1]; assign s14[31]=1'b0;
addsub_32 sub14(r14, c14, of14, s14, b,  sub);
assign m14[31:1]=r14[30:0]; assign m14[0]=a14[0];
mux2to1_32 mux14(a14,  m14,c14, a15);
mux2to1 o14(zero, one, c14, s[1]);


   assign s15[31:0]=a15[31:0];
   addsub_32 sub15(r15, c15, of15, s15,b, sub);
   assign m15[31:0]=r15[31:0];
   mux2to1_32 mux15(a15, m15, c15, a16);
   mux2to1 o15(zero, one, c15, s[0]);
   assign s[31:16]=a16[15:0];


endmodule // divider_32
```

```
loadi @0 16   -> a010
loadi @1 12   -> a10c
loadi @2 5    -> a205
div @3 @1 @2  -> 6312
srl @4 @3 @0  -> 1430
sll @3 @3 @0  -> 0330
srl @3 @3 @0  -> 1330
stop          -> ffff
```



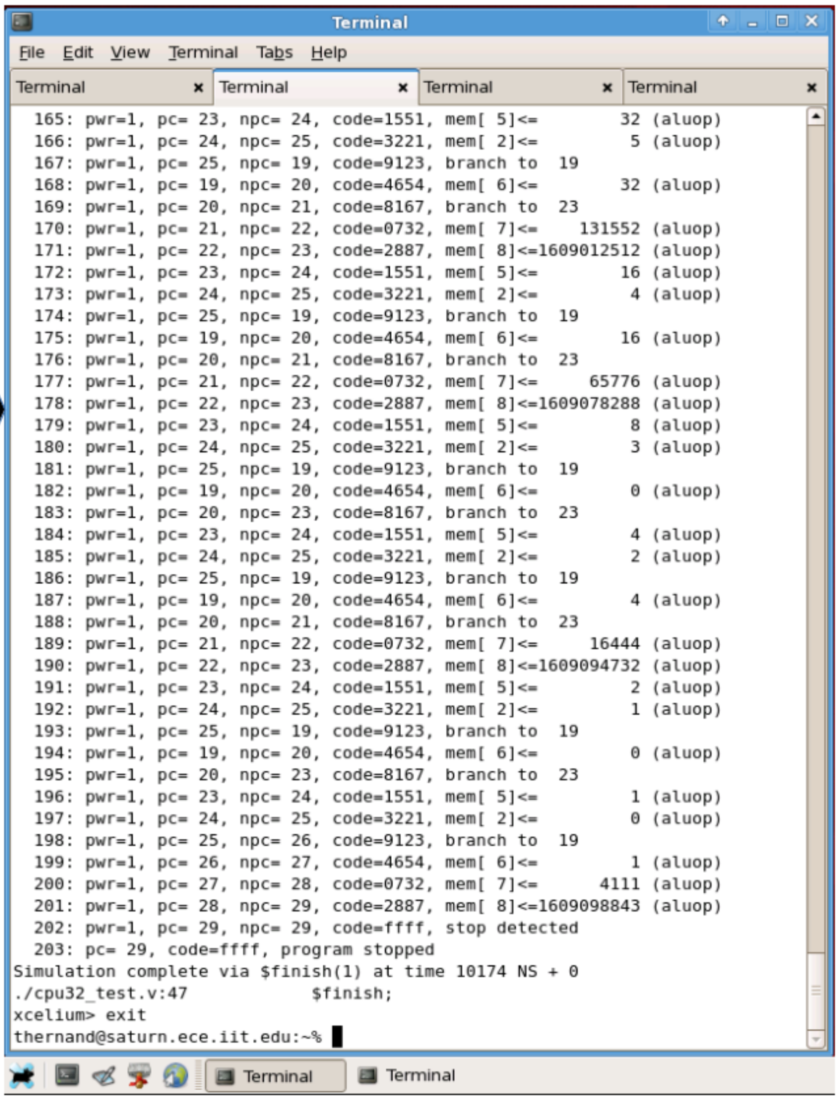
```
                   Primitives:          10764        6
                   Registers:             525        6
                   Scalar wires:          607        -
                   Expanded wires:       1499       90
                   Always blocks:         521        2
                   Initial blocks:          5        5
                   Cont. assignments:      65       97
                   Pseudo assignments:      2        2
           Writing initial simulation snapshot: worklib.stimulus:v
Loading snapshot worklib.stimulus:v .................... Done
xcelium> source /apps/cadence/XCELIUM1803/tools/xcelium/files/xmsimrc
xcelium> run
xmsim: *W,SHMPOPT: Some objects excluded from $shm_probe due to optimizations
            File: ./cpu32_test.v, line = 37, pos = 15
           Scope: stimulus
            Time: 0 FS + 0

    1: pwr=1, pc=  0, npc=  1, code=a010, mem[ 0]<=        16 (loadi)
    2: pwr=1, pc=  1, npc=  2, code=a10c, mem[ 1]<=        12 (loadi)
    3: pwr=1, pc=  2, npc=  3, code=a205, mem[ 2]<=         5 (loadi)
    4: pwr=1, pc=  3, npc=  4, code=6312, mem[ 3]<=    131074 (aluop)
    5: pwr=1, pc=  4, npc=  5, code=1430, mem[ 4]<=         2 (aluop)
    6: pwr=1, pc=  5, npc=  6, code=0330, mem[ 3]<=    131072 (aluop)
    7: pwr=1, pc=  6, npc=  7, code=1330, mem[ 3]<=         2 (aluop)
    8: pwr=1, pc=  7, npc=  7, code=ffff, stop detected
    9: pc=  7, code=ffff, program stopped
Simulation complete via $finish(1) at time 759 NS + 0
./cpu32_test.v:47             $finish;
xcelium> exit
thernand@saturn.ece.iit.edu:~%
```

VII Functional Validation and Verification

VII.I Code simulation

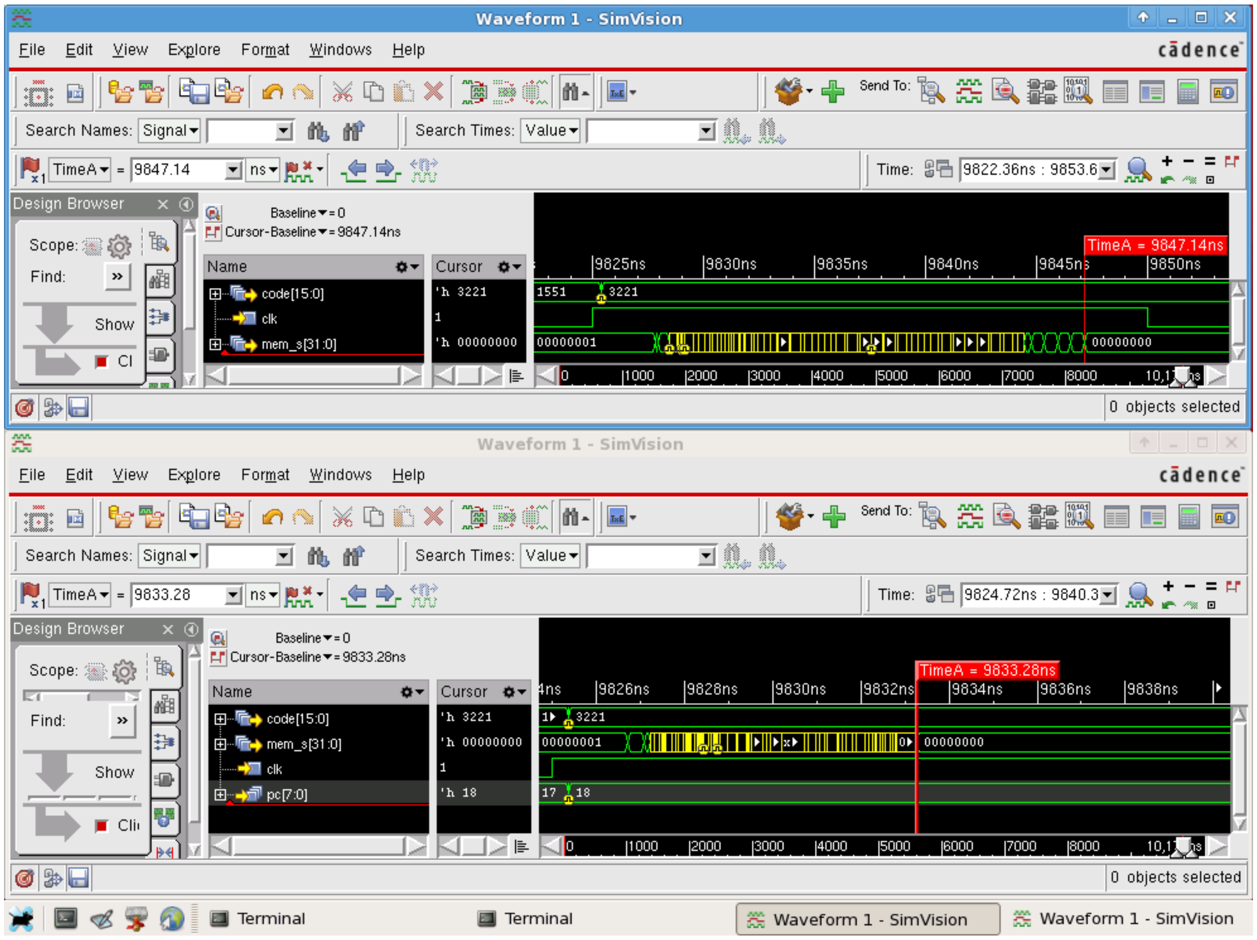
The RTL design of the 32-bit CPU once finish could be tested functionally by using the **cpu32_test.v** file. This file contain a test bench that would attempt to run the hexadecimal code in the file **code.hex** and show the results to the screen. In the case that we see below, I wrote this assembly code in order to test the instruction set. In this code I multiply 4111 times the last six digits of my CWID number (391413). The result is as expected 1,609,098,843. In this case I used shift left logical for multiplying in factors of two and add the results into @8.

```
loadi @0, 0;
loadi @1, 1;
loadi @2, 31;
loadi @3, 16; 0x10
loadi @4, 15; 0x0F
loadi @5, 8;
sll @3, @3, @5;
add @3, @3, @4;
loadi @4, 5; 0x05
loadi @6, 248; 0xF8
loadi @7, 245; 0xF5
sll @4, @4, @5;
sll @4, @4, @5;
sll @6, @6, @5;
add @4, @4, @6;
add @4, @4, @7;
loadi @5, 1;
loadi @8, 0;
sll @5, @5, @2;
and @6, @5, @4;
jz @6, 23;
sll @7, @3, @2;
add @8, @8, @7;
srl @5, @5, @1;
sub @2, @2, @1;
jnz @2, 19
and @6, @5, @4;
sll @7, @3, @2;
add @8, @8, @7;
stop ;
```



```
  165: pwr=1, pc= 23, npc= 24, code=1551, mem[ 5]<=        32 (aluop)
  166: pwr=1, pc= 24, npc= 25, code=3221, mem[ 2]<=         5 (aluop)
  167: pwr=1, pc= 25, npc= 19, code=9123, branch to  19
  168: pwr=1, pc= 19, npc= 20, code=4654, mem[ 6]<=        32 (aluop)
  169: pwr=1, pc= 20, npc= 21, code=8167, branch to  23
  170: pwr=1, pc= 21, npc= 22, code=0732, mem[ 7]<=    131552 (aluop)
  171: pwr=1, pc= 22, npc= 23, code=2887, mem[ 8]<=1609012512 (aluop)
  172: pwr=1, pc= 23, npc= 24, code=1551, mem[ 5]<=        16 (aluop)
  173: pwr=1, pc= 24, npc= 25, code=3221, mem[ 2]<=         4 (aluop)
  174: pwr=1, pc= 25, npc= 19, code=9123, branch to  19
  175: pwr=1, pc= 19, npc= 20, code=4654, mem[ 6]<=        16 (aluop)
  176: pwr=1, pc= 20, npc= 21, code=8167, branch to  23
  177: pwr=1, pc= 21, npc= 22, code=0732, mem[ 7]<=     65776 (aluop)
  178: pwr=1, pc= 22, npc= 23, code=2887, mem[ 8]<=1609078288 (aluop)
  179: pwr=1, pc= 23, npc= 24, code=1551, mem[ 5]<=         8 (aluop)
  180: pwr=1, pc= 24, npc= 25, code=3221, mem[ 2]<=         3 (aluop)
  181: pwr=1, pc= 25, npc= 19, code=9123, branch to  19
  182: pwr=1, pc= 19, npc= 20, code=4654, mem[ 6]<=         0 (aluop)
  183: pwr=1, pc= 20, npc= 23, code=8167, branch to  23
  184: pwr=1, pc= 23, npc= 24, code=1551, mem[ 5]<=         4 (aluop)
  185: pwr=1, pc= 24, npc= 25, code=3221, mem[ 2]<=         2 (aluop)
  186: pwr=1, pc= 25, npc= 19, code=9123, branch to  19
  187: pwr=1, pc= 19, npc= 20, code=4654, mem[ 6]<=         4 (aluop)
  188: pwr=1, pc= 20, npc= 21, code=8167, branch to  23
  189: pwr=1, pc= 21, npc= 22, code=0732, mem[ 7]<=     16444 (aluop)
  190: pwr=1, pc= 22, npc= 23, code=2887, mem[ 8]<=1609094732 (aluop)
  191: pwr=1, pc= 23, npc= 24, code=1551, mem[ 5]<=         2 (aluop)
  192: pwr=1, pc= 24, npc= 25, code=3221, mem[ 2]<=         1 (aluop)
  193: pwr=1, pc= 25, npc= 19, code=9123, branch to  19
  194: pwr=1, pc= 19, npc= 20, code=4654, mem[ 6]<=         0 (aluop)
  195: pwr=1, pc= 20, npc= 23, code=8167, branch to  23
  196: pwr=1, pc= 23, npc= 24, code=1551, mem[ 5]<=         1 (aluop)
  197: pwr=1, pc= 24, npc= 25, code=3221, mem[ 2]<=         0 (aluop)
  198: pwr=1, pc= 25, npc= 26, code=9123, branch to  19
  199: pwr=1, pc= 26, npc= 27, code=4654, mem[ 6]<=         1 (aluop)
  200: pwr=1, pc= 27, npc= 28, code=0732, mem[ 7]<=      4111 (aluop)
  201: pwr=1, pc= 28, npc= 29, code=2887, mem[ 8]<=1609098843 (aluop)
  202: pwr=1, pc= 29, npc= 29, code=ffff, stop detected
  203: pc= 29, code=ffff, program stopped
Simulation complete via $finish(1) at time 10174 NS + 0
./cpu32_test.v:47            $finish;
xcelium> exit
thernand@saturn.ece.iit.edu:~%
```

The mapping into hexadecimal was performed using a python script that I wrote. It takes an assembly file and converts it into hexadecimal as shown in the appendix A-1. The difference between the carry-ripple adder and the carry look-ahead adder can be seen in the Simvision simulation once both standards cells and routing have been mapped to the Verilog code. Also the simulation shows a close look at the line 197 from the figure above with the instruction sub @2, @2, @1 –¿ 3221.



We can see in this case that the difference in performance is almost 14ns. The simulation below is performed by the carry look-ahead.

VII.II Formality Check

Once the netlist of the connected components in the standard cells have been generated, an important test is to check the logical equivalence between this netlist and the Verilog code. In this particular case, the amount of ports that have to be checked are pretty extensive. However, as we see in the equivalence check, the equivalence is verified without problems. The verification is a very important step considering that the map between standard cells and the original Verilog circuit with the interconnections are done only by manipulating template files that might contain errors, in this particular case the verification was successful but it is not guarantee that this would be the case every time for different projects. A screen-shot of the equivalence checking can be seen in the appendix A-2. This equivalence checking includes an equivalence checking between the carry look-ahead and ripple adders and one between the final.v and the cpu32.v files.

VIII Synthesis Results

VIII.I Chip Area

During the mapping from the original Verilog file to the Verilog header file with the standard cells, an approximation of the total area of the chip was created. However, this approximation did not contain the interconnects between the cells and therefore could not be completely accurate.
The area of the CPU chip was $15338.6 \mu m^2$ for the carry look-ahead adder and slightly different for the CPU with the carry-ripple implementation ($15368.6 \mu m^2$)
After the interconnection of the cells, the final area was further reduced until $13769.3 \mu m^2$, I was expecting that the area would increase from the estimation without the interconnection since we are introducing further constrains into the optimization. I am glad I was wrong about the final size. For a screenshot of the area files please take a look at A-3 in the appendix.

VIII.II Power Usage

The power was incredibly different for the two CPU, since the carry-ripple adder used almost one fourth of power extra. The power usage before the interconnection was fairly incomplete with 1.077mW for the look-ahead architecture and $1.324 mW$. Being used mainly to maintain the latches within the registers and some combinational logic.
However, once all the interconnection was done, the power went up to $3.22 mW$ for the look-ahead implementation and $4.006 mW$ for the carry ripple implementation. With most of the power consumed within the internal power of the sequential logic and the switching power of the combinational logic (including the clock).
In order to look at the power files, we can see the appendix A-4.

VIII.III Timing

The most important difference in timing was at the subtraction operation in which the look-ahead adder architecture worked up to 14 ns faster than the carry ripple adder.

VIII.IV Layout

The cells layout information is not included within the files created during all previous stages. We only have their placement and interconnections. For this reason we have to use virtuoso to retrieve the layout information from our libraries in order to create the final physical layout that is shown in the appendix A-5.

IX Conclusion and Future Work

During this project, I extended the data path and registers to 32 bits. In this way, all the operations worked without a problem with a 32-bit size. However, since the critical path was the addition with its 32 serial full adders, I implemented a look-ahead adder to solve partially the problem. The look-ahead adder is not free of propagation delay, but it has a far more efficient way of calculating the carry bit. This improvement in the adder architecture also produced system-wide changes in the timing of operations, area of the chip, and the most striking of all almost one forth of savings in power consumption. I also added a divider functionality to the CPU. This project so far has been incredibly challenging, specially the implementation of the divider. However, I have learn so much that with this set of tools I feel like I could implement far more functionalities if needed. I have put into practice most of the knowledge gathered in the labs in this project. The overall project has been finished and the semester too. However, there are some important pieces that we could have implemented in the future. An important aspect of a computer is to have memory (besides the registers). In the next iteration we could have memory to load and store values and programs. Another important aspects of a CPU is the implementation of exceptions handling. For example my divider does not give an error when the user tries to divide by zero, what would happen if there is a wrongly coded instruction? There are many important aspects to implement in this CPU. Most CPU's nowadays use pipelines and are therefore, multi-cycle implementations. This is probably far harder to code, but the tools are already there for us to use them.

X Appendix

A-1: Python Parser

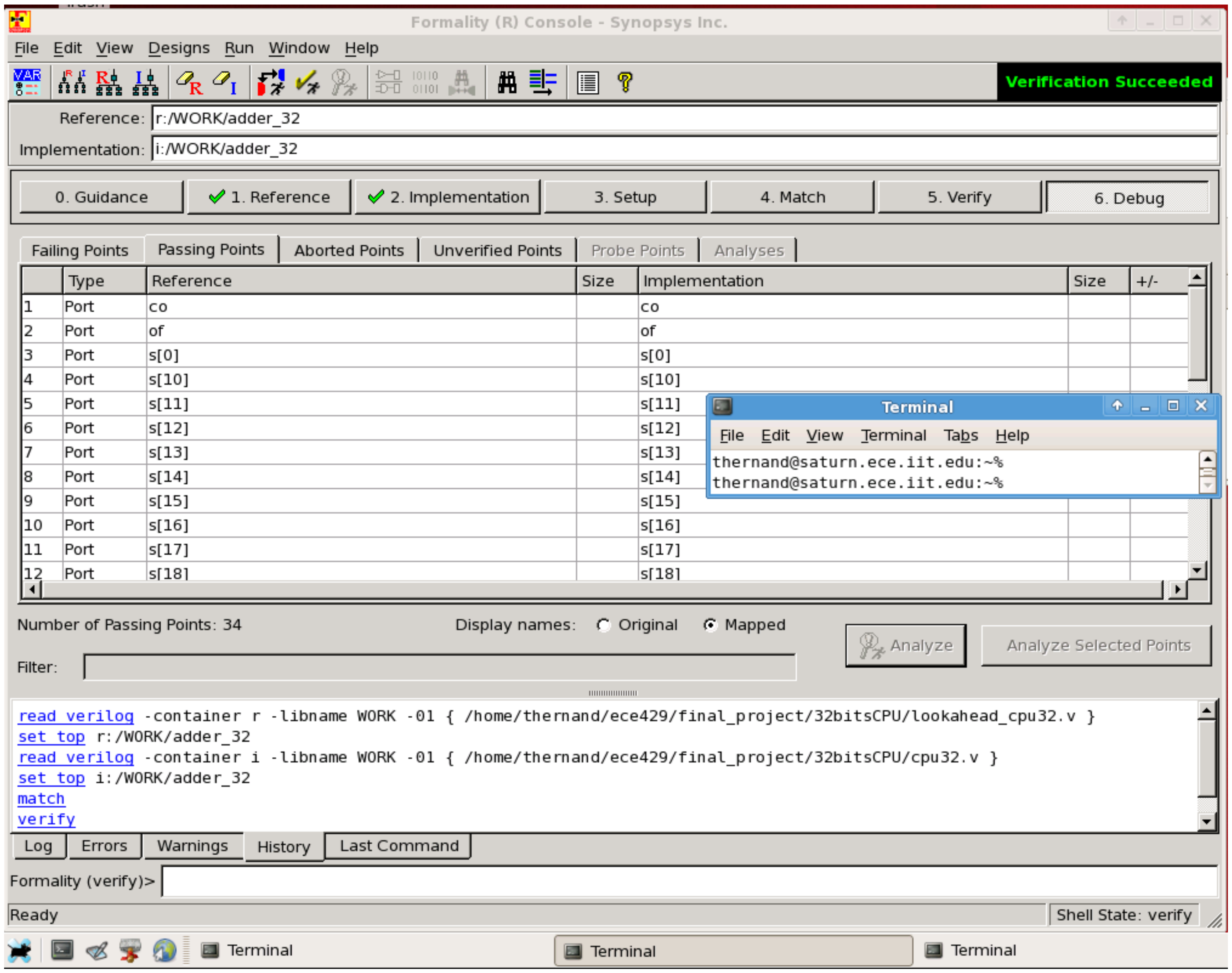
This code is to be run from the terminal. it's usage is ”convert.py -i ¡inputfile¿ -o ¡outputfile¿” where the input file contains assembly code for this CPU and the output file is a file in which the parser will write the hexadecimal translation of the code.

```
#!/usr/bin/python
g1={'sll':0,'srl':1,'add':2,'sub':3,'and':4,'xnor':5}
g2={'loadi':10}
g3={'jz':8,'jnz':9,'stop':15}
import sys, getopt,re
def translate(inputfile,outputfile):
    with open(outputfile,'w') as ofile:
        with open(inputfile,'r') as ifile:
            for line in ifile.readlines():
                line=line.split()
                if len(line)>0:
                    if line[0] in g1:
                        binary = group1(line)
                    if line[0] in g2:
                        binary = group2(line)
                    if line[0] in g3:
                        binary = group3(line)
                    binary = '%0*X' % ((len(binary) + 3) // 4, int(binary, 2))
                    ofile.write(binary.lower()+"\n")

def group1(instr):
    s=instr[1]
    a=instr[2]
    b=instr[3]
    if s[0] == '@' and a[0] == '@' and b[0] == '@':
        import re
        s=int(re.sub("\D", "", s))
        b=int(re.sub("\D", "", b))
        a=int(re.sub("\D", "", a))
    else:
        print('Wrong Format',instr)
    binary = "0000{0:b}".format(g1[instr[0]])[-4:]+ \
        "000{0:b}".format(s)[-4:]+"000{0:b}".format(a)[-4:]+"000{0:b}".format(b)[-4:]
    print(instr[0],"@"+str(s),"@"+str(a),"@"+str(b),
    '\t -> ',binary[0:4],binary[4:8],binary[8:12],binary[12:])
    return binary
def group2(instr):
    s=instr[1]
    imm=instr[2]
    if s[0] == '@':
        s=int(re.sub("\D", "", s))
        imm=int(re.sub("\D", "", imm))
    else:
        print('Wrong Format',instr)
    binary = "0000{0:b}".format(g2[instr[0]])[-4:]+ \
        "000{0:b}".format(s)[-4:]+"0000000{0:b}".format(imm)[-8:]
    print(instr[0],"@"+str(s),imm,'\t -> ',
        binary[0:4],binary[4:8],binary[8:12],binary[12:])
    return binary
```

```
def group3(instr):
    if instr[0]=='stop':
        binary = '1'*16
        print(instr[0],'\t\t -> ',binary[0:4],
            binary[4:8],binary[8:12],binary[12:])
        return binary
    a=instr[1]
    bpc=instr[2]
    if a[0] == '@':
        a=int(re.sub("\D", "", a))
        bpc=int(re.sub("\D", "", bpc))
    else:
        print('Wrong Format',instr)
    bpcBin="0000000{0:b}".format(bpc)[-8:]
    binary = "0000{0:b}".format(g3[instr[0]])[-4:] +\
         bpcBin[:-4] + "000{0:b}".format(a)[-4:]+bpcBin[-4:]
    print(instr[0],"@"+str(a),bpc,'\t -> ',
        binary[0:4],binary[4:8],binary[8:12],binary[12:])
    return binary

def main(argv):
   inputfile = ''
   outputfile = ''
   try:
      opts, args = getopt.getopt(argv,"hi:o:",["ifile=","ofile="])
   except getopt.GetoptError:
      print('convert.py -i <inputfile> -o <outputfile>')
      sys.exit(2)
   for opt, arg in opts:
      if opt == '-h':
         print('convert.py -i <inputfile> -o <outputfile>')
         sys.exit()
      elif opt in ("-i", "--ifile"):
         inputfile = arg
      elif opt in ("-o", "--ofile"):
         outputfile = arg
   print('Input file is "', inputfile,'"')
   print('Output file is "', outputfile,'"')
   translate(inputfile,outputfile)
if __name__ == "__main__":
   main(sys.argv[1:])
```

A-2.1: Logical Equivalence Verification Between two Adders

In this case I performed an Equivalence Verification between the carry ripple adder and the carry look-ahead adder



Reference: r:/WORK/adder_32

Implementation: i:/WORK/adder_32

	Type	Reference	Size	Implementation	Size	+/-
1	Port	co		co		
2	Port	of		of		
3	Port	s[0]		s[0]		
4	Port	s[10]		s[10]		
5	Port	s[11]		s[11]		
6	Port	s[12]		s[12]		
7	Port	s[13]		s[13]		
8	Port	s[14]		s[14]		
9	Port	s[15]		s[15]		
10	Port	s[16]		s[16]		
11	Port	s[17]		s[17]		
12	Port	s[18]		s[18]		

```
thernand@saturn.ece.iit.edu:~%
thernand@saturn.ece.iit.edu:~%
```

Number of Passing Points: 34   Display names: Original [✓ Mapped]

Filter:

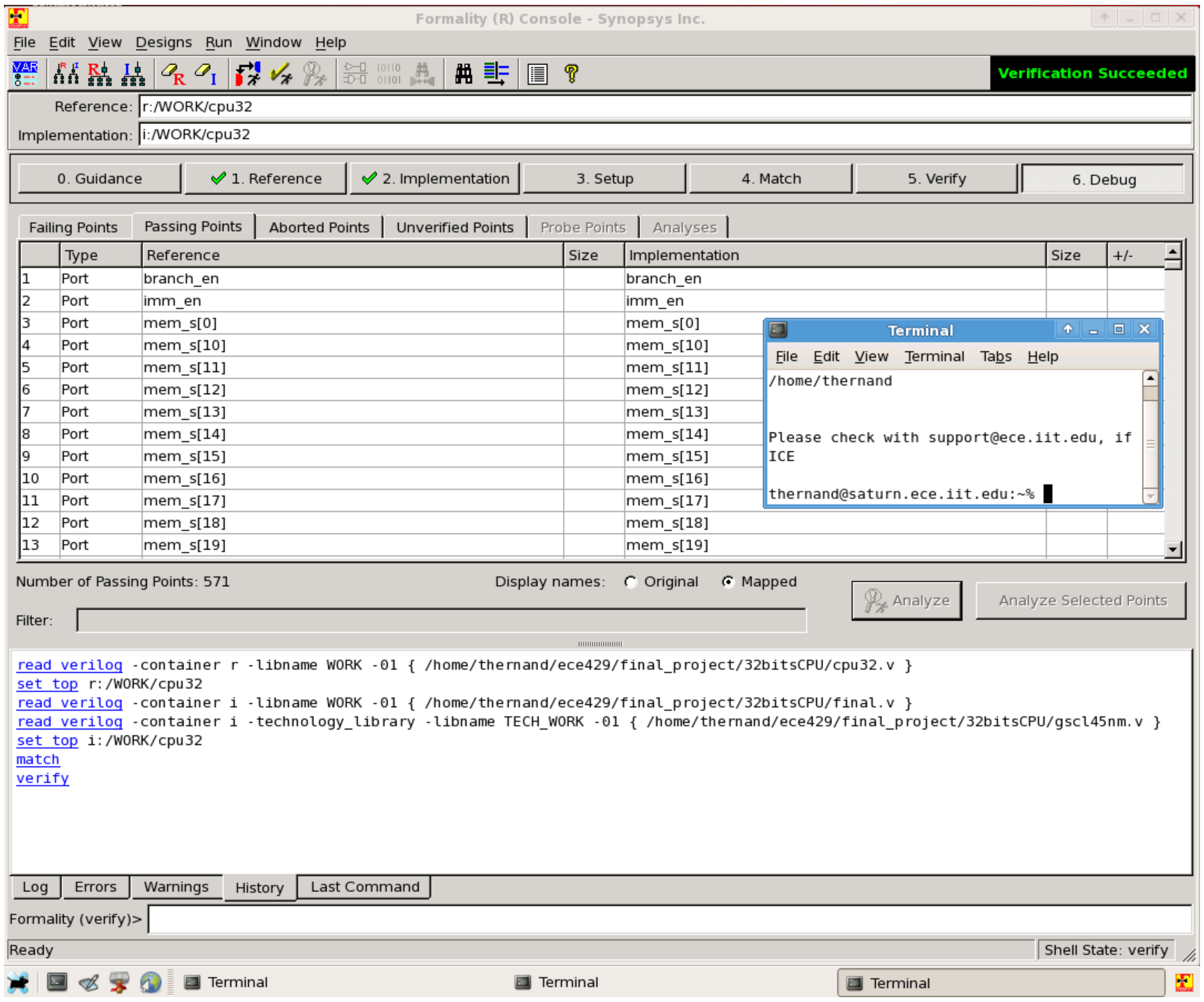
```
read_verilog -container r -libname WORK -01 { /home/thernand/ece429/final_project/32bitsCPU/lookahead_cpu32.v }
set_top r:/WORK/adder_32
read_verilog -container i -libname WORK -01 { /home/thernand/ece429/final_project/32bitsCPU/cpu32.v }
set_top i:/WORK/adder_32
match
verify
```

Formality (verify)>

Ready   Shell State: verify

A-2.2: Logical Equivalence Verification Between cpu32.v final.v

The process of creating a Verilog header with the standard cells and finalizing their interconnections is likely to produce a logical equivalent circuit to the original Verilog file. However, it must always be verified. For this reason we perform an equivalence check between them.



Reference: r:/WORK/cpu32

Implementation: i:/WORK/cpu32

	Type	Reference	Size	Implementation	Size	+/-
1	Port	branch_en		branch_en		
2	Port	imm_en		imm_en		
3	Port	mem_s[0]		mem_s[0]		
4	Port	mem_s[10]		mem_s[10]		
5	Port	mem_s[11]		mem_s[11]		
6	Port	mem_s[12]		mem_s[12]		
7	Port	mem_s[13]		mem_s[13]		
8	Port	mem_s[14]		mem_s[14]		
9	Port	mem_s[15]		mem_s[15]		
10	Port	mem_s[16]		mem_s[16]		
11	Port	mem_s[17]		mem_s[17]		
12	Port	mem_s[18]		mem_s[18]		
13	Port	mem_s[19]		mem_s[19]		

Number of Passing Points: 571

Display names: Original [✓ Mapped]

Filter:

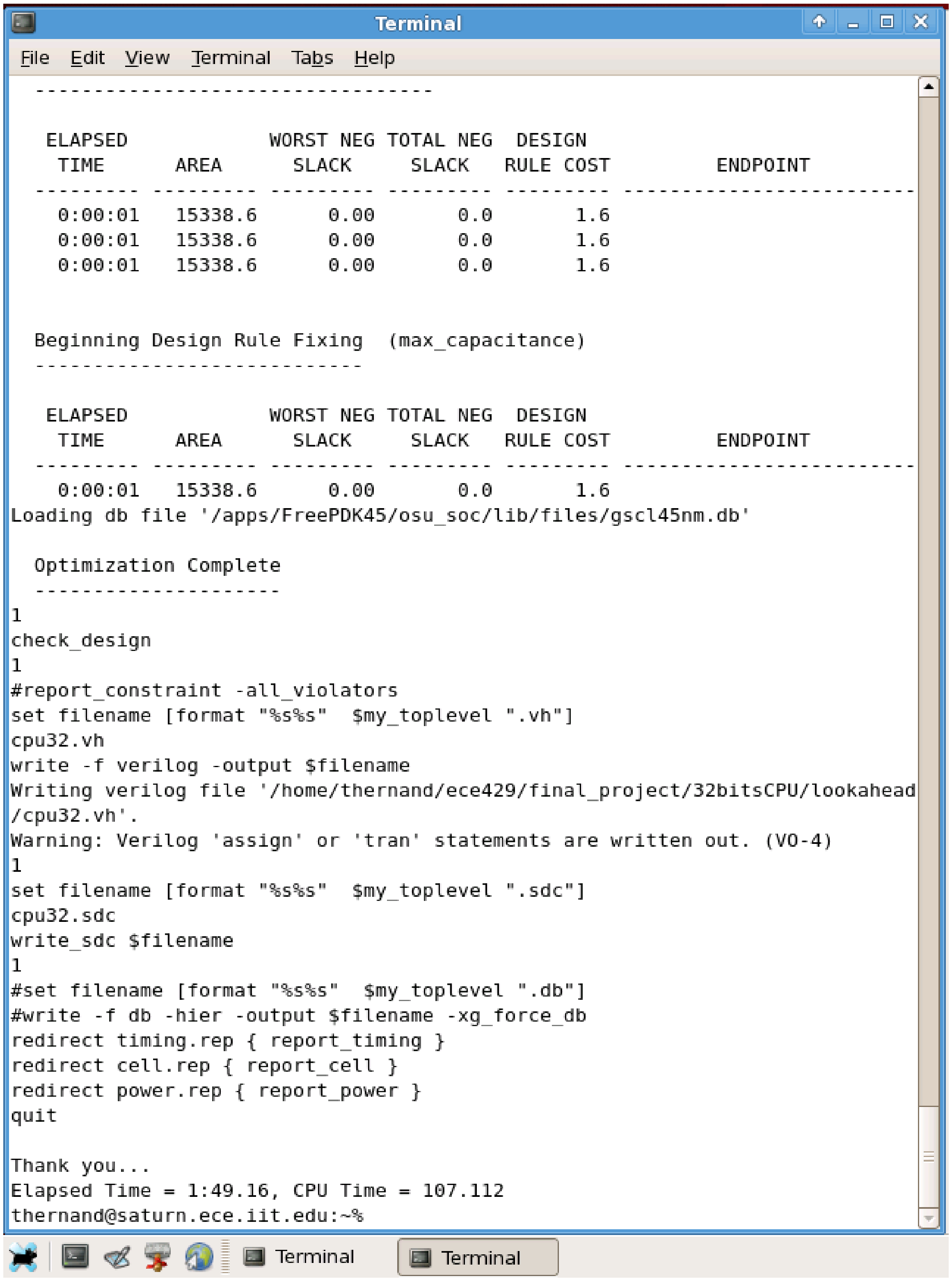
```
/home/thernand


Please check with support@ece.iit.edu, if
ICE

thernand@saturn.ece.iit.edu:~%
```

```
read_verilog -container r -libname WORK -01 { /home/thernand/ece429/final_project/32bitsCPU/cpu32.v }
set_top r:/WORK/cpu32
read_verilog -container i -libname WORK -01 { /home/thernand/ece429/final_project/32bitsCPU/final.v }
read_verilog -container i -technology_library -libname TECH_WORK -01 { /home/thernand/ece429/final_project/32bitsCPU/gscl45nm.v }
set_top i:/WORK/cpu32
match
verify
```

Formality (verify)>

A-3.1: Chip Area partial Look-Ahead

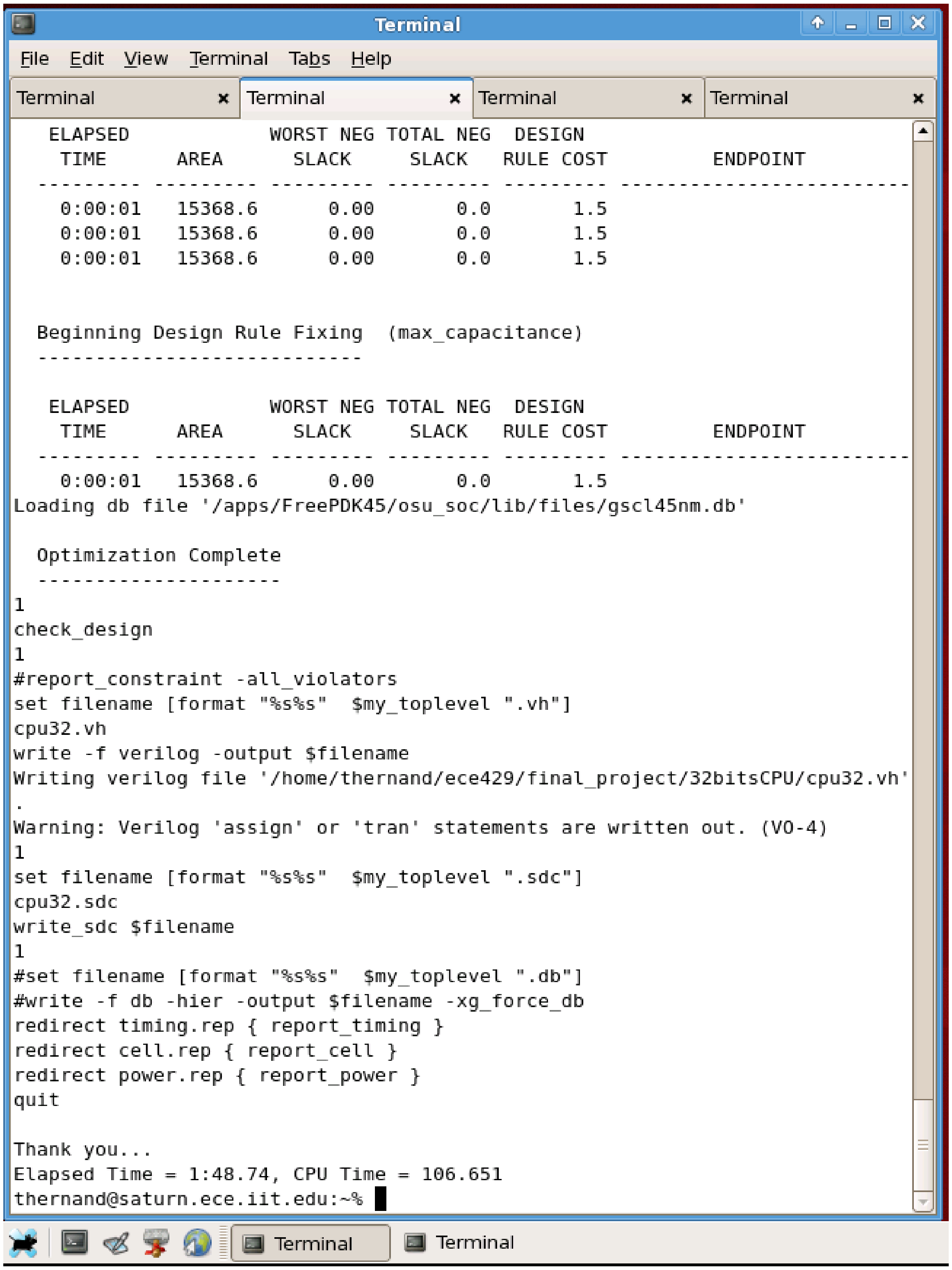
The top terminal is showing the area without the interconnect in several iterations ending in to $15338.6 \mu m^2$.



```
  ----------------------------------

   ELAPSED            WORST NEG TOTAL NEG  DESIGN
    TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
  --------- --------- --------- --------- --------- -------------------------
    0:00:01   15338.6      0.00       0.0       1.6
    0:00:01   15338.6      0.00       0.0       1.6
    0:00:01   15338.6      0.00       0.0       1.6


  Beginning Design Rule Fixing  (max_capacitance)
  ----------------------------

   ELAPSED            WORST NEG TOTAL NEG  DESIGN
    TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
  --------- --------- --------- --------- --------- -------------------------
    0:00:01   15338.6      0.00       0.0       1.6
Loading db file '/apps/FreePDK45/osu_soc/lib/files/gscl45nm.db'

  Optimization Complete
  ---------------------
1
check_design
1
#report_constraint -all_violators
set filename [format "%s%s"  $my_toplevel ".vh"]
cpu32.vh
write -f verilog -output $filename
Writing verilog file '/home/thernand/ece429/final_project/32bitsCPU/lookahead
/cpu32.vh'.
Warning: Verilog 'assign' or 'tran' statements are written out. (VO-4)
1
set filename [format "%s%s"  $my_toplevel ".sdc"]
cpu32.sdc
write_sdc $filename
1
#set filename [format "%s%s"  $my_toplevel ".db"]
#write -f db -hier -output $filename -xg_force_db
redirect timing.rep { report_timing }
redirect cell.rep { report_cell }
redirect power.rep { report_power }
quit

Thank you...
Elapsed Time = 1:49.16, CPU Time = 107.112
thernand@saturn.ece.iit.edu:~%
```

A-3.2: Chip Area partial ripple

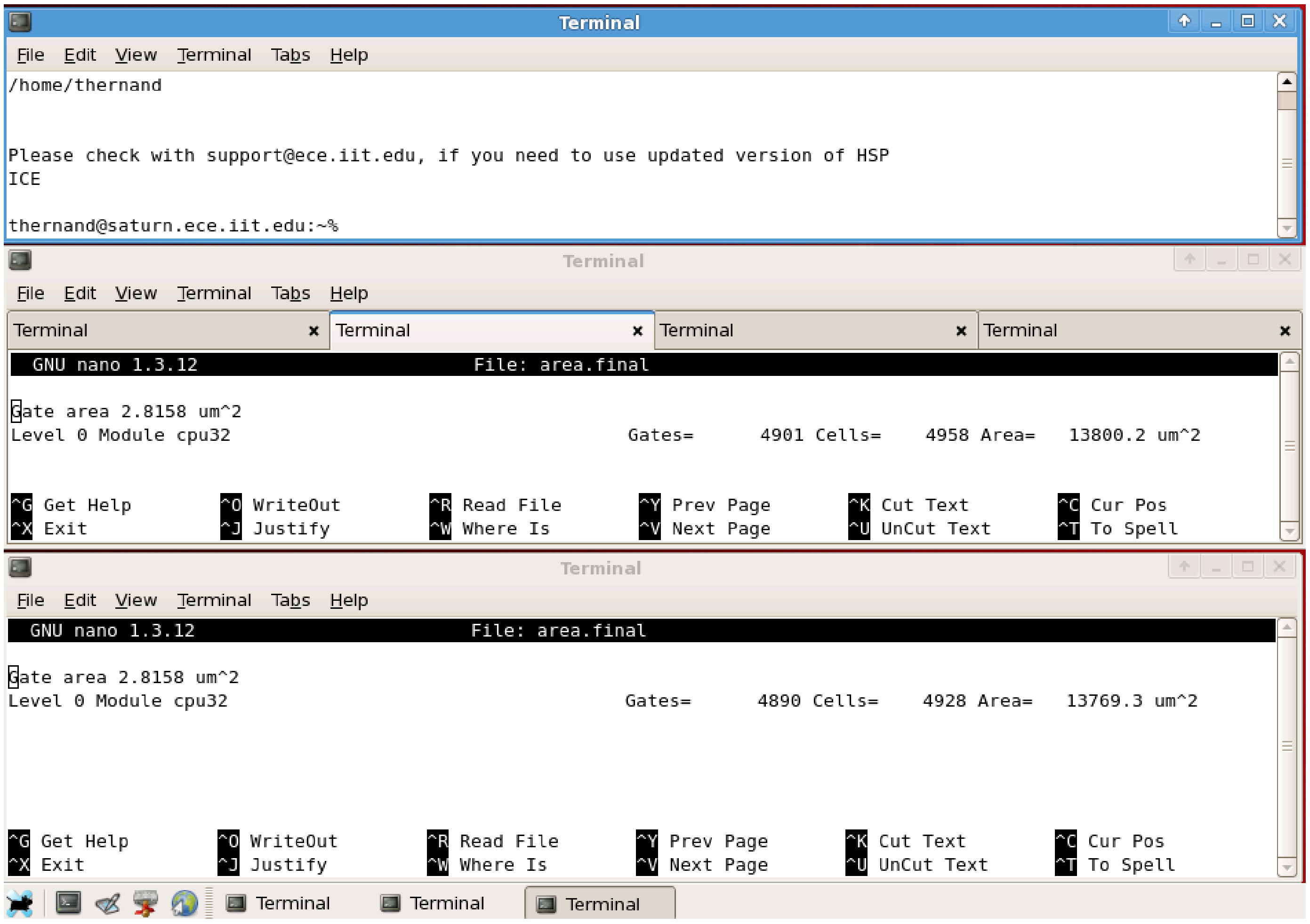
The top terminal is showing the area without the interconnect in several iterations ending in to $15368.6\mu m^2$.



```
  ELAPSED           WORST NEG TOTAL NEG  DESIGN
    TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
  --------- --------- --------- --------- --------- -------------------------
    0:00:01   15368.6      0.00       0.0       1.5
    0:00:01   15368.6      0.00       0.0       1.5
    0:00:01   15368.6      0.00       0.0       1.5


  Beginning Design Rule Fixing  (max_capacitance)
  ----------------------------


  ELAPSED           WORST NEG TOTAL NEG  DESIGN
    TIME      AREA      SLACK     SLACK   RULE COST         ENDPOINT
  --------- --------- --------- --------- --------- -------------------------
    0:00:01   15368.6      0.00       0.0       1.5
Loading db file '/apps/FreePDK45/osu_soc/lib/files/gscl45nm.db'

  Optimization Complete
  ---------------------
1
check_design
1
#report_constraint -all_violators
set filename [format "%s%s"  $my_toplevel ".vh"]
cpu32.vh
write -f verilog -output $filename
Writing verilog file '/home/thernand/ece429/final_project/32bitsCPU/cpu32.vh'
.
Warning: Verilog 'assign' or 'tran' statements are written out. (VO-4)
1
set filename [format "%s%s"  $my_toplevel ".sdc"]
cpu32.sdc
write_sdc $filename
1
#set filename [format "%s%s"  $my_toplevel ".db"]
#write -f db -hier -output $filename -xg_force_db
redirect timing.rep { report_timing }
redirect cell.rep { report_cell }
redirect power.rep { report_power }
quit

Thank you...
Elapsed Time = 1:48.74, CPU Time = 106.651
thernand@saturn.ece.iit.edu:~%
```

A-3.3: Chip Area Final Both Look-Ahead and Ripple Adders

As we can see in this two terminals, the top is the ripple adder CPU that finish with an area of 13800.2 μm^2 while the look-Ahead ended with an area of 13769.3 μm^2



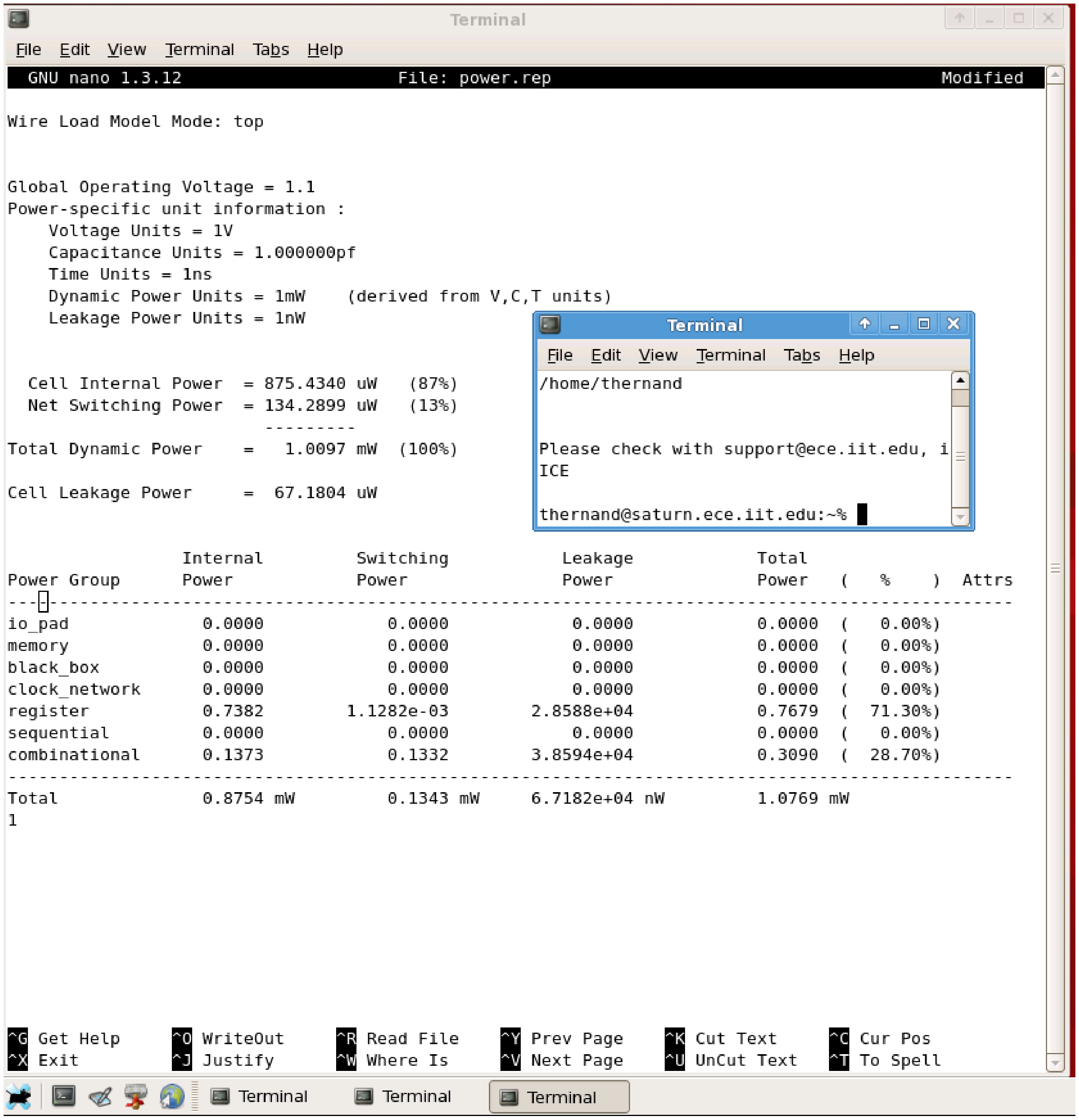
```
/home/thernand


Please check with support@ece.iit.edu, if you need to use updated version of HSP
ICE

thernand@saturn.ece.iit.edu:~%
```

```
  GNU nano 1.3.12                        File: area.final

Gate area 2.8158 um^2
Level 0 Module cpu32                                    Gates=      4901 Cells=    4958 Area=    13800.2 um^2


^G Get Help        ^O WriteOut        ^R Read File       ^Y Prev Page       ^K Cut Text        ^C Cur Pos
^X Exit            ^J Justify         ^W Where Is        ^V Next Page       ^U UnCut Text      ^T To Spell
```

```
  GNU nano 1.3.12                        File: area.final

Gate area 2.8158 um^2
Level 0 Module cpu32                                    Gates=      4890 Cells=    4928 Area=    13769.3 um^2


^G Get Help        ^O WriteOut        ^R Read File       ^Y Prev Page       ^K Cut Text        ^C Cur Pos
^X Exit            ^J Justify         ^W Where Is        ^V Next Page       ^U UnCut Text      ^T To Spell
```

A-4.1: Power W/O Interconnect Look-Ahead Adder

The look-ahead adder cpu spent a total power of 1.077 mW



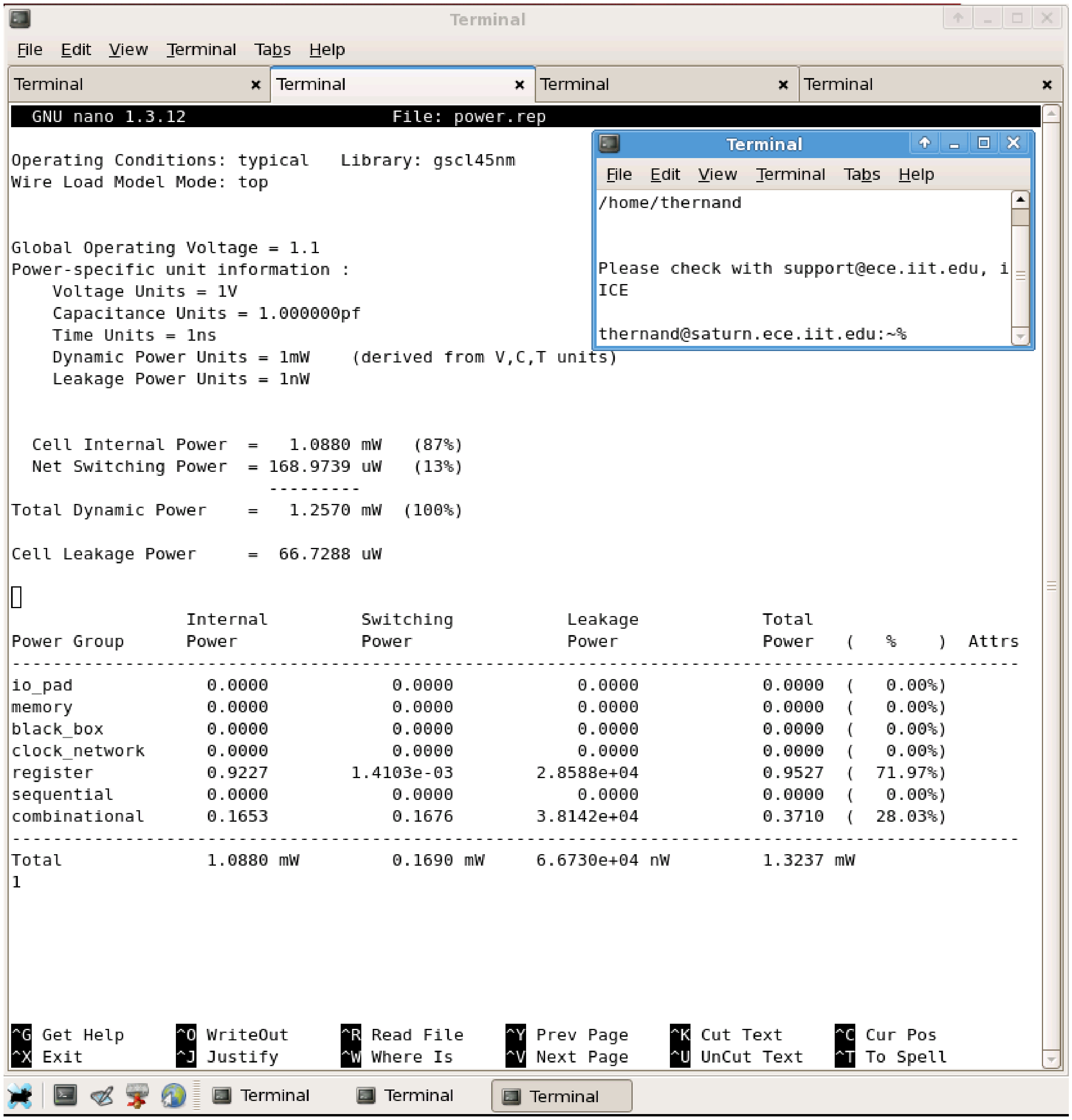
```
  GNU nano 1.3.12                    File: power.rep                    Modified

Wire Load Model Mode: top


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  = 875.4340 uW   (87%)
  Net Switching Power  = 134.2899 uW   (13%)
                         ---------
Total Dynamic Power    =   1.0097 mW  (100%)

Cell Leakage Power     =  67.1804 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
---------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)
register           0.7382        1.1282e-03         2.8588e+04             0.7679  (  71.30%)
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)
combinational      0.1373           0.1332          3.8594e+04             0.3090  (  28.70%)
---------------------------------------------------------------------------------------------------
Total              0.8754 mW        0.1343 mW       6.7182e+04 nW          1.0769 mW
1
```

```
/home/thernand


Please check with support@ece.iit.edu, i
ICE

thernand@saturn.ece.iit.edu:~%
```

Terminal
File Edit View Terminal Tabs Help
^G Get Help ^O WriteOut ^R Read File ^Y Prev Page ^K Cut Text ^C Cur Pos
^X Exit ^J Justify ^W Where Is ^V Next Page ^U UnCut Text ^T To Spell
Terminal
Terminal
Terminal

A-4.2: Power W/O Interconnect Ripple Adder

The ripple adder cpu has a total power of 1.324 mW

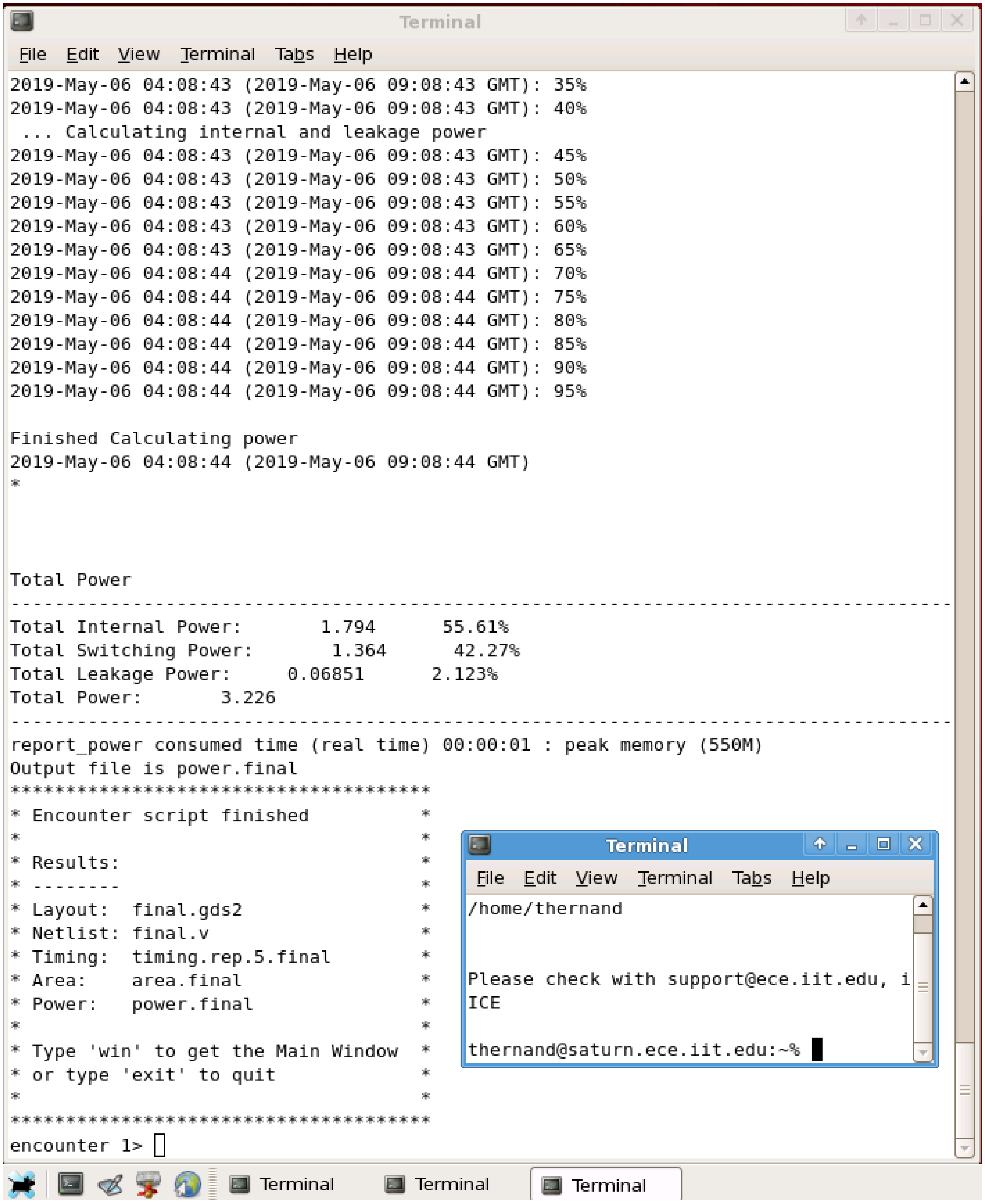


```
  GNU nano 1.3.12                    File: power.rep

Operating Conditions: typical   Library: gscl45nm
Wire Load Model Mode: top


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1nW


  Cell Internal Power  =   1.0880 mW   (87%)
  Net Switching Power  = 168.9739 uW   (13%)
                         ---------
Total Dynamic Power    =   1.2570 mW  (100%)

Cell Leakage Power     =  66.7288 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000             0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000             0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000             0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000             0.0000  (   0.00%)
register           0.9227        1.4103e-03         2.8588e+04             0.9527  (  71.97%)
sequential         0.0000           0.0000              0.0000             0.0000  (   0.00%)
combinational      0.1653           0.1676          3.8142e+04             0.3710  (  28.03%)
--------------------------------------------------------------------------------------------------
Total              1.0880 mW        0.1690 mW       6.6730e+04 nW          1.3237 mW
1
```

```
/home/thernand


Please check with support@ece.iit.edu, i
ICE

thernand@saturn.ece.iit.edu:~%
```

A-4.3: Power Final Look-Ahead Adder

The look-ahead adder cpu spent a total power of 3.226 mW

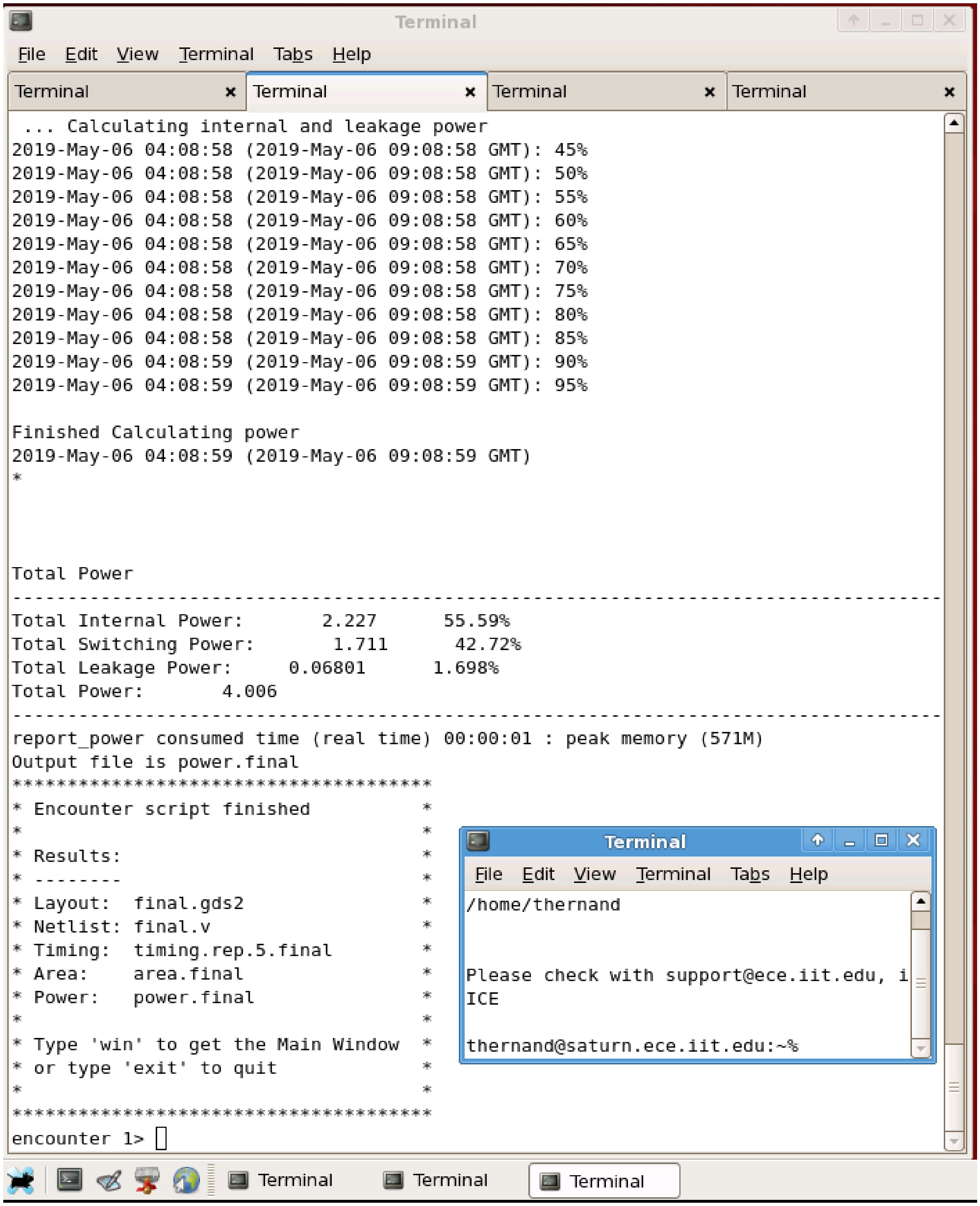


```
2019-May-06 04:08:43 (2019-May-06 09:08:43 GMT): 35%
2019-May-06 04:08:43 (2019-May-06 09:08:43 GMT): 40%
 ... Calculating internal and leakage power
2019-May-06 04:08:43 (2019-May-06 09:08:43 GMT): 45%
2019-May-06 04:08:43 (2019-May-06 09:08:43 GMT): 50%
2019-May-06 04:08:43 (2019-May-06 09:08:43 GMT): 55%
2019-May-06 04:08:43 (2019-May-06 09:08:43 GMT): 60%
2019-May-06 04:08:43 (2019-May-06 09:08:43 GMT): 65%
2019-May-06 04:08:44 (2019-May-06 09:08:44 GMT): 70%
2019-May-06 04:08:44 (2019-May-06 09:08:44 GMT): 75%
2019-May-06 04:08:44 (2019-May-06 09:08:44 GMT): 80%
2019-May-06 04:08:44 (2019-May-06 09:08:44 GMT): 85%
2019-May-06 04:08:44 (2019-May-06 09:08:44 GMT): 90%
2019-May-06 04:08:44 (2019-May-06 09:08:44 GMT): 95%

Finished Calculating power
2019-May-06 04:08:44 (2019-May-06 09:08:44 GMT)
*




Total Power
-----------------------------------------------------------------------------------------
Total Internal Power:       1.794      55.61%
Total Switching Power:       1.364      42.27%
Total Leakage Power:     0.06851      2.123%
Total Power:       3.226
-----------------------------------------------------------------------------------------
report_power consumed time (real time) 00:00:01 : peak memory (550M)
Output file is power.final
**************************************
* Encounter script finished          *
*                                    *
* Results:                           *
* --------                           *
* Layout:  final.gds2                *
* Netlist: final.v                   *
* Timing:  timing.rep.5.final        *
* Area:    area.final                *
* Power:   power.final               *
*                                    *
* Type 'win' to get the Main Window  *
* or type 'exit' to quit             *
*                                    *
**************************************
encounter 1>
```

```
/home/thernand


Please check with support@ece.iit.edu, i
ICE

thernand@saturn.ece.iit.edu:~%
```

A-4.4: Power Final Ripple Adder

The ripple adder cpu has a total power of 4.006 mW

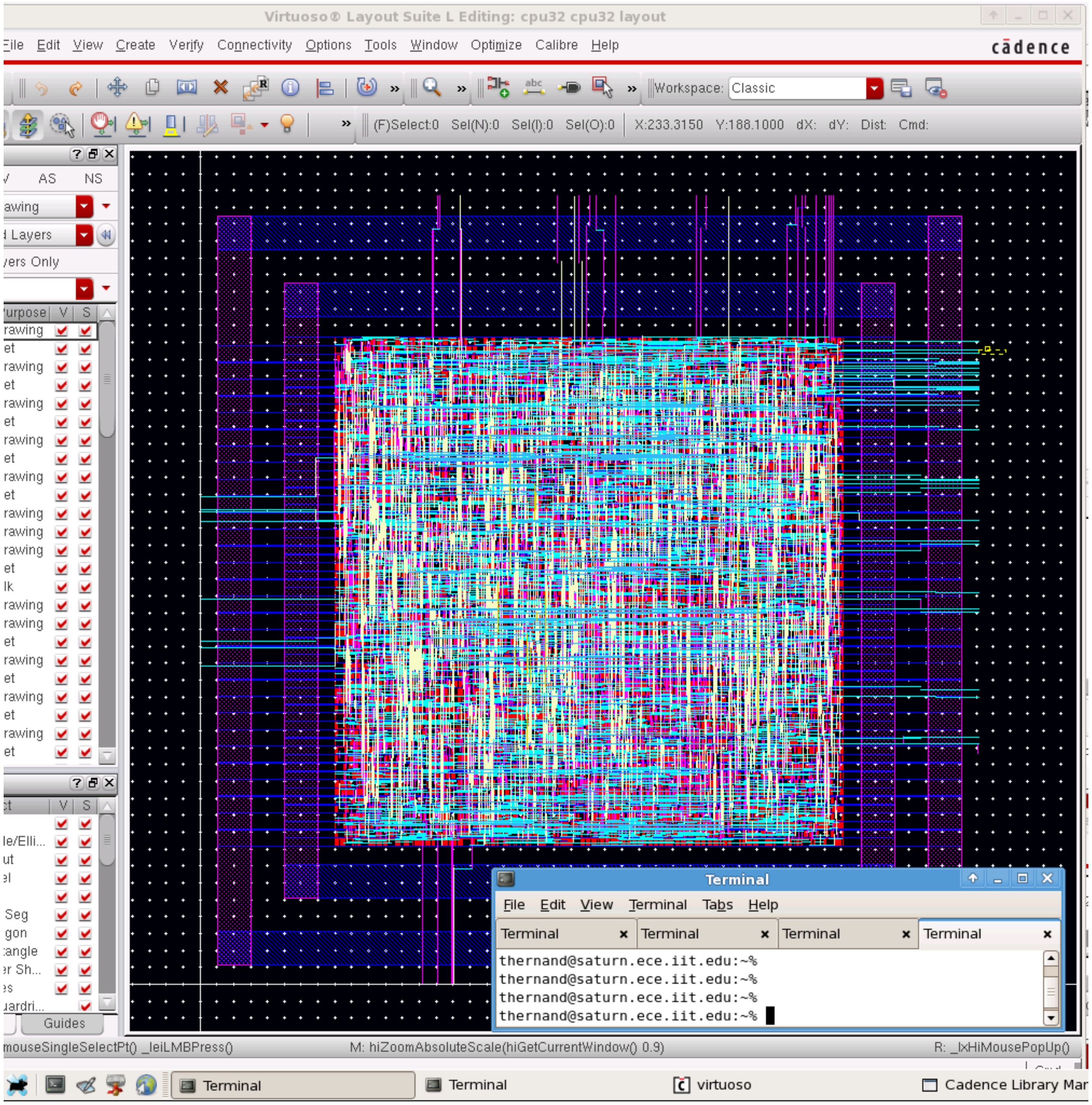


```
 ... Calculating internal and leakage power
2019-May-06 04:08:58 (2019-May-06 09:08:58 GMT): 45%
2019-May-06 04:08:58 (2019-May-06 09:08:58 GMT): 50%
2019-May-06 04:08:58 (2019-May-06 09:08:58 GMT): 55%
2019-May-06 04:08:58 (2019-May-06 09:08:58 GMT): 60%
2019-May-06 04:08:58 (2019-May-06 09:08:58 GMT): 65%
2019-May-06 04:08:58 (2019-May-06 09:08:58 GMT): 70%
2019-May-06 04:08:58 (2019-May-06 09:08:58 GMT): 75%
2019-May-06 04:08:58 (2019-May-06 09:08:58 GMT): 80%
2019-May-06 04:08:58 (2019-May-06 09:08:58 GMT): 85%
2019-May-06 04:08:59 (2019-May-06 09:08:59 GMT): 90%
2019-May-06 04:08:59 (2019-May-06 09:08:59 GMT): 95%

Finished Calculating power
2019-May-06 04:08:59 (2019-May-06 09:08:59 GMT)
*



Total Power
-----------------------------------------------------------------------------------
Total Internal Power:       2.227      55.59%
Total Switching Power:       1.711      42.72%
Total Leakage Power:     0.06801      1.698%
Total Power:       4.006
-----------------------------------------------------------------------------------
report_power consumed time (real time) 00:00:01 : peak memory (571M)
Output file is power.final
**************************************
* Encounter script finished          *
*                                    *
* Results:                           *
* --------                           *
* Layout:  final.gds2                *
* Netlist: final.v                   *
* Timing:  timing.rep.5.final        *
* Area:    area.final                *
* Power:   power.final               *
*                                    *
* Type 'win' to get the Main Window  *
* or type 'exit' to quit             *
*                                    *
**************************************
encounter 1>
```

```
/home/thernand


Please check with support@ece.iit.edu, i
ICE

thernand@saturn.ece.iit.edu:~%
```

A-5: Final Layout of the circuit

The cells layout information is not included within the files created during all previous stages. We only have their placement and interconnections. For this reason we have to use virtuoso to retrieve the layout information from our libraries in order to create the final physical layout that is shown in the figure below.


Virtuoso® Layout Suite L Editing: cpu32 cpu32 layout
File Edit View Create Verify Connectivity Options Tools Window Optimize Calibre Help
cādence
Workspace: Classic
(F)Select:0 Sel(N):0 Sel(I):0 Sel(O):0 X:233.3150 Y:188.1000 dX: dY: Dist: Cmd:
mouseSingleSelectPt() _leiLMBPress()
M: hiZoomAbsoluteScale(hiGetCurrentWindow() 0.9)
R: _lxHiMousePopUp()
Terminal
File Edit View Terminal Tabs Help
Terminal
Terminal
Terminal
Terminal
virtuoso
Cadence Library Mar

```
thernand@saturn.ece.iit.edu:~%
thernand@saturn.ece.iit.edu:~%
thernand@saturn.ece.iit.edu:~%
thernand@saturn.ece.iit.edu:~%
```

XI References

References

[1] *Project Description,* `http://www.ece.iit.edu/~jwang/ece429-2019s/ece429-prj.pdf`

[2] *Weste, N. H. E., & Harris, D. M. (2011). CMOS VLSI design: A circuits and systems perspective. Boston: Addison Wesley.*

[3] *The 68000's Instruction Set (page 21)*
`http://www.ece.iit.edu/~jwang/ece429-2019s/ece429-prj.pdf`