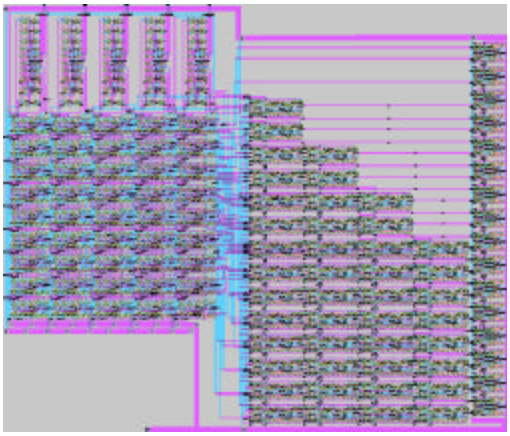
Introduction to CMOS VLSI Design (E158)

Final Project

8-bit Booth Recoded Multiplier



Alfred Chuang
Peter Grossmann

11 April 2000

FUNCTIONAL OVERVIEW

This project is a design for an 8-bit multiplication of unsigned numbers. The algorithm used is Radix-4 Booth encoding for generation of five partial products, a Wallace tree adder to perform carry-save-style addition on the partial products, and a 16-bit carry propagate adder to add the sum and carry bits output by the Wallace tree. This technique is similar to, albeit in a scaled-down form, techniques used in fast multipliers today.

Our particular implementation is as follows: a booth encoder generates five nine-bit partial products, each depending on three bits of the input B, or multiplier (abbreviated mier) and all bits of A, or the multiplicand (abbreviated mcand). Five are required since for each partial product, the middle bit of the multiplier is B[2*I], so order to , it is thus necessary to have the 0^{th} partial product consider B[1], B[0] B[-1] = 0, the 1^{st} to look at b[3:1], and so on until the 5^{th} partial product looks at B[9] = B[8] = 0 and B[7]. Each partial product PP is either 0, A, -A, 2A or –2A according to the following table of three-bit mier values:

Mier	PP
000	0
001	A
010	A
011	2A
100	-2A
101	-A
110	-A
111	0

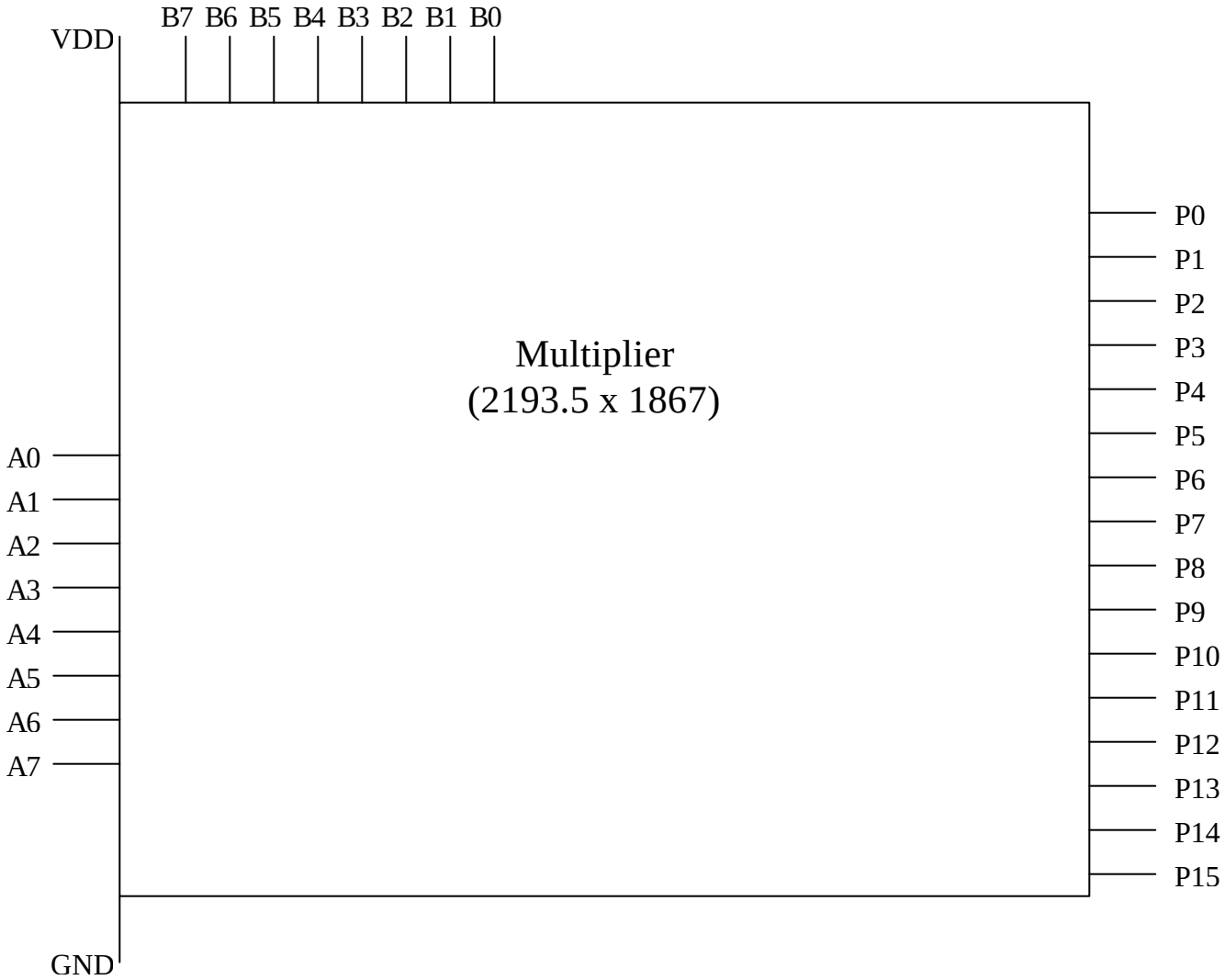
By looking at multiple bits of the multiplier and allowing the partial product to have this range of values, it is possible to multiply two 8-bit numbers summing five partial products instead of the usual eight. One slight tradeoff for this technique, at least for unsigned multiplication as implemented here, is that negation requires not only an inversion of bits, but an addition of one to preserve twos complement form, as well as sign extension. Notice above that the sign bit can be taken to be Mier[2] if negation is performed (albeit with no ultimate effect) for Mier = 111. This bit can be output to a product addition scheme as both a “carry” (addition of 0 for positive and 1 for negative to ensure 2s complement form) and a sign extension. This forms a 10^{th} bit of output for each partial product.

Once generated, the partial products must be summed. In this design, the Wallace tree compresses the bits of each column to be summed into two bits, a sum and a carry into the next column, using a series of full adders as 3:2 compressors. One bit of the resulting compression (the sum bit) remains in the current column while the other, as a carry bit, must be sent to the next most significant column if a carry for that column is already

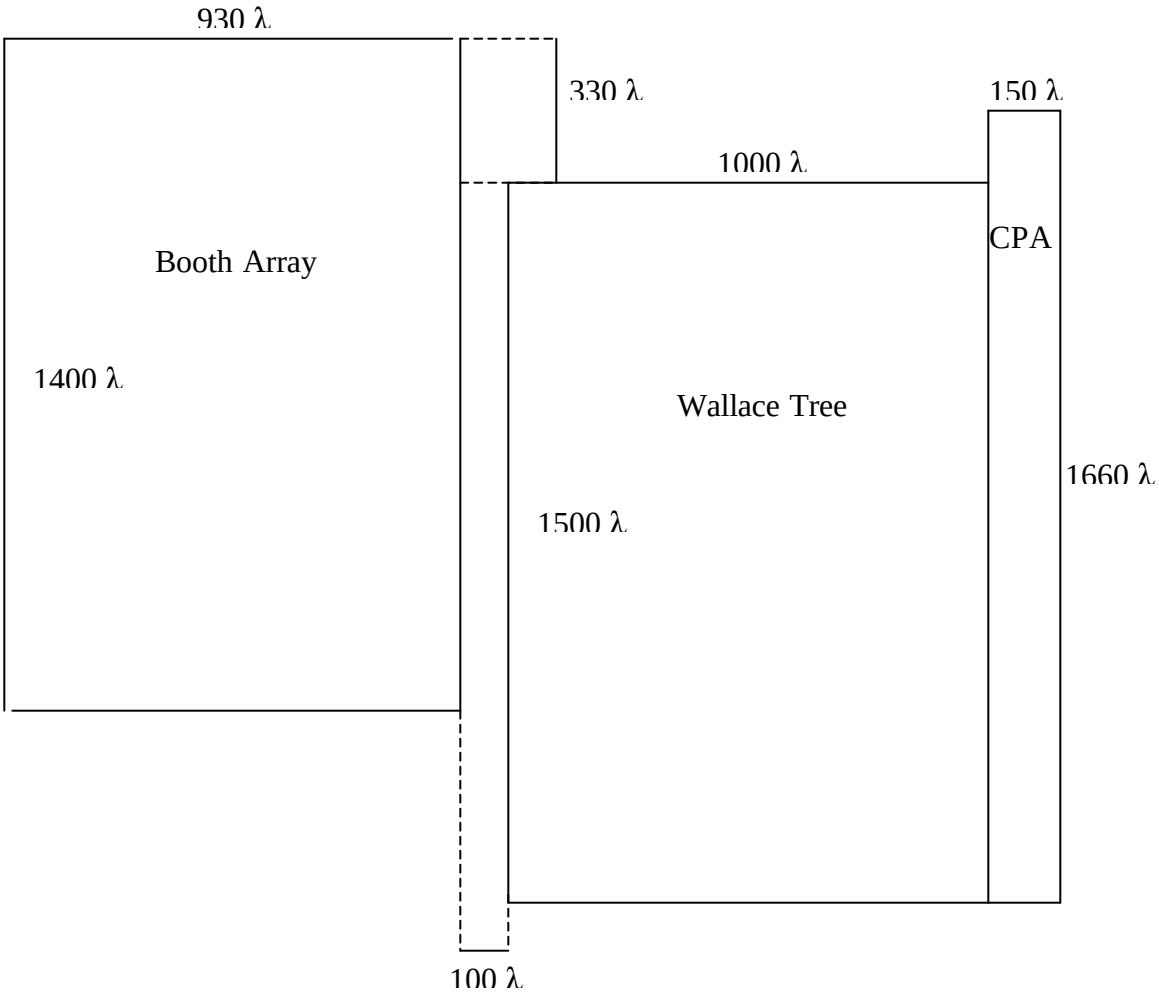
slated for the CPA. With five partial products plus one extra bit each, it is possible to accumulate up to three extra carries, so that up to nine bits may be added in a single column. This therefore requires no more than two levels of 3:2 compression for any one column. Once each column is compressed to two bits, the final add may occur according to any carry-propagate adder technique. To keep the project more manageable, a basic ripple-carry adder was used in this design.

CHIP PINOUT

Inputs	Outputs
A<7:0>	P<15:0>
B<7:0>	
VDD	
GND	



CHIP FLOORPLAN



AREA AND DESIGN TIME REPORT

The following summarizes the area of blocks designed specifically for the final project (standard gates designed in labs are not represented here). All areas represent the smallest rectangle that can be drawn around the entire layout.

Cell	Area (Width x Height)	Design Time (hrs)	
		Sch.	Layout
pass-gate	70.5 x 97.5	2	3
test	111.25 x 105.25	2	3
booth-pp	171.875 x 105.25	2	3
booth-decode	400.25 x 98.25	2	4
booth-cell = booth-pp x 9 + booth-decode x 1	185.875 x 1350.25	3	3
fa_cpa	139 x 99	2	6
fa_wal	246 x 99	2	12
booth-array = booth-cell x 5	930.625 x 1405	3	3
cpa = fa_cpa x 16	153 x 1695.5	2	3
waltree = fa_wal x 44	1005.5 x 1501	4	10
multiplier = booth-array + cpa + waltree	**2193.5 x 1867**	**24**	**50**
TOTAL AVAILABLE AREA	**2200 x 2200**		

SIMULATION

All simulation for this design was carried out using IRSIM. Waveforms of leaf cell output for all possible inputs are attached as Appendix A.

While leaf cells were readily simulated manually, some automation was needed to effectively simulate the three higher level blocks. In order to efficiently process a sufficient number of tests to ensure correct functionality, we wrote a Java program to generate .cmd files for scripting the simulation. Simulation results were processed by a second Java program. By putting a watch ("w" command in IRSIM) on all inputs and outputs, and piping simulation console output into a text file, we could import inputs and received outputs into the Java program, which then compared received outputs to expected outputs and printed appropriate messages when errors were found. The source code for these two Java programs are attached as Appendix B. The documentation contained within them describes their functionality in greater detail.

Java-based testing was conducted for the booth array, Wallace tree, and the multiplier as a whole. For the booth array, three corner cases (A = B = 0x00; A = B = 0xFF, and A = B = 0xAA), and 100 randomly generated pairs of inputs were used. For the Wallace tree and the multiplier, the same tests (albeit with different random cases) plus 256 additional corner cases (comprising all permutations of inputs where both contain exactly one 1) were used. Testing for the CPA was conducted through testing of the multiplier since its functionality was relatively straightforward. Final simulation results are as follows:

Booth-array: 0 errors
Wallace tree: 99 errors consisting of output containing Xs. In light of the correct behavior of the multiplier this is believed to be due to a software bug.
Multiplier: 0 errors.

VERIFICATION STATUS

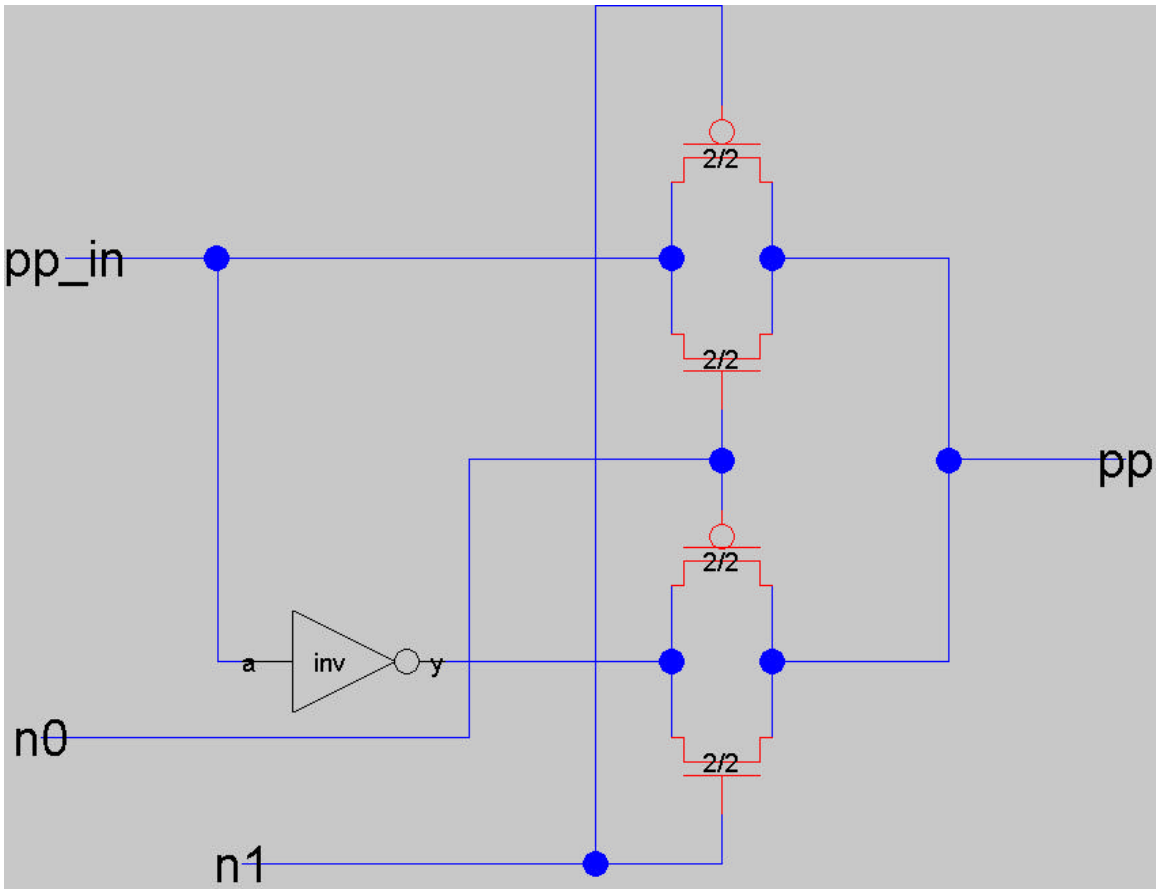
Leaf Cell	CELL	DRC	ERC	Network Compare (tool used)
√	pass-gate	Pass	Pass	Pass (NCC)
√	test	Pass	Pass	Pass (NCC)
√	booth-decode	Pass	Pass	Pass (NCC)
	booth-pp	Pass	Pass	Pass (NCC, Gemini)
	booth-cell	Pass	Pass	Pass (NCC, Gemini)
√	fa_cpa	Pass	Pass	Pass (Gemini)
√	fa_wal	Pass	Pass	Pass (Gemini)
	booth-array	Pass	Pass	Pass (NCC, Gemini)
	cpa	Pass	Pass	Pass (Gemini)
	waltree	Pass	Pass	Pass (Gemini)
	multiplier	Pass	Pass	Pass (Gemini)

POST-FABRICATION TEST PLAN

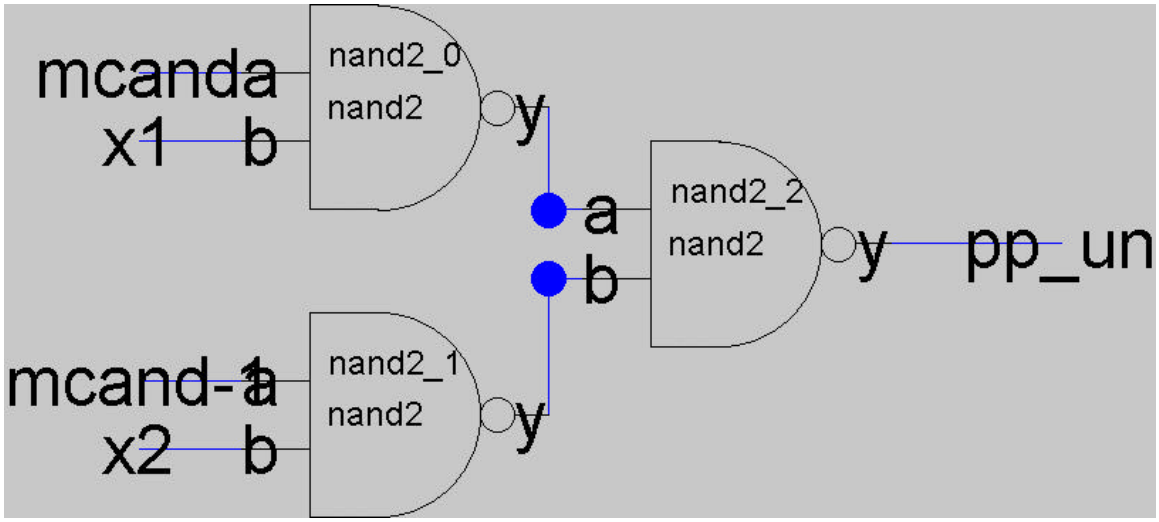
Testing of the multiplier chip could be done with an E155 FPGA board. The breadboard schematic would consist of the FPGA board, appropriate setup for installing the chip onto the breadboard, 16 bits worth of DIP switches, and 8 LEDs to supplement the 8 on-board LEDs. The FPGA may then be programmed with an 8-bit multiplier and 16-bit comparator. The person testing could control the two inputs via the DIP switches. These inputs would be sent to both the chip and the FPGA. The FPGA would compute the output and compare it to the output of the chip, turning on LEDs for bits with errors.

SCHEMATICS

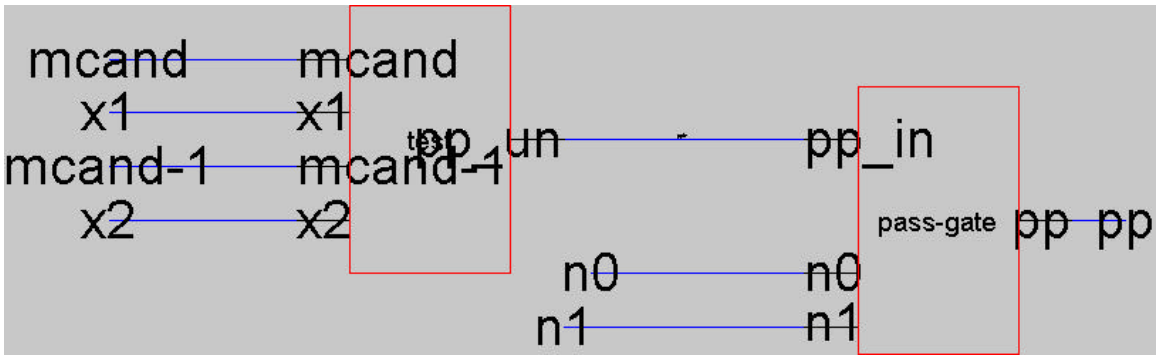
Pass-gate:



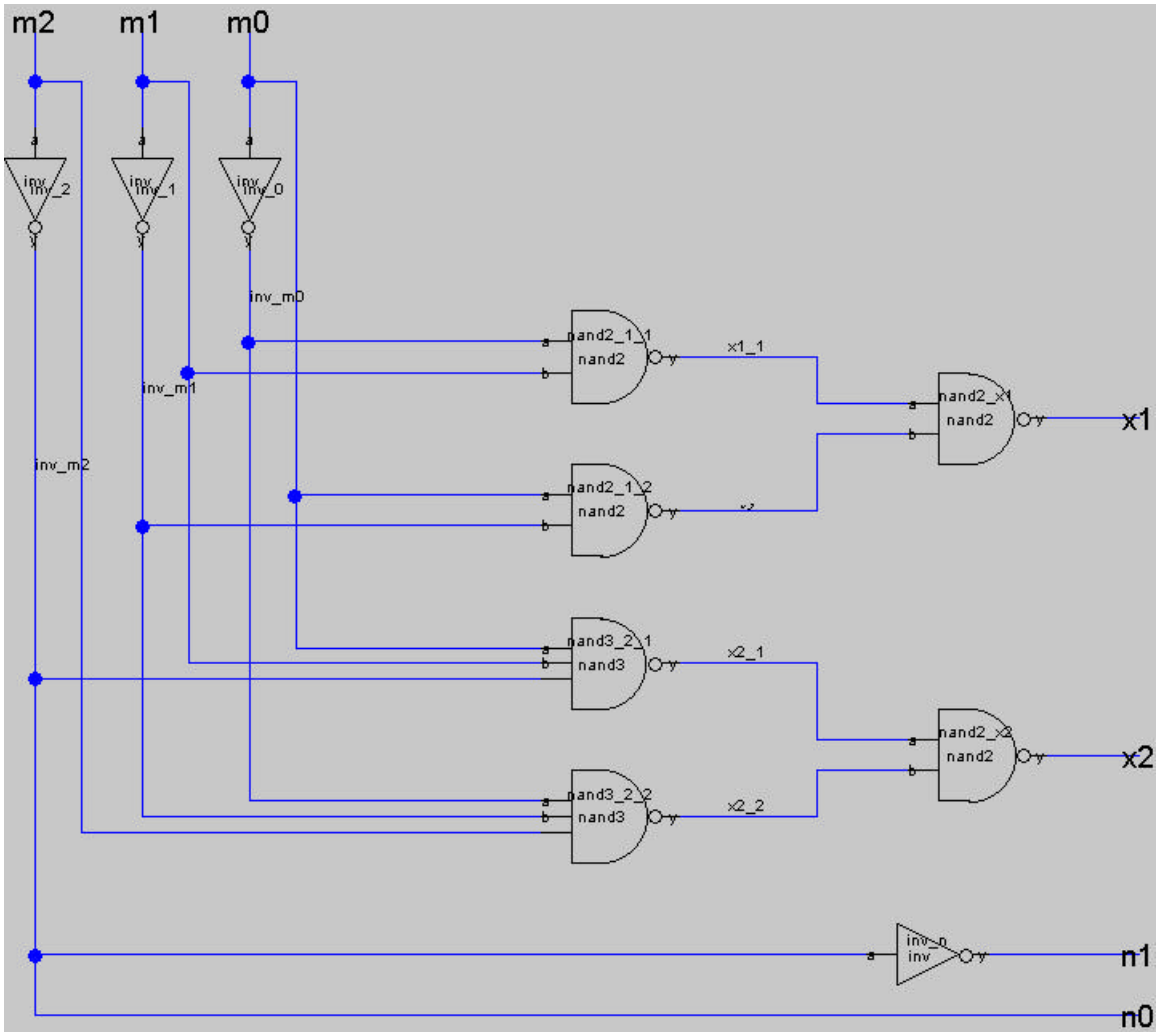
Test:



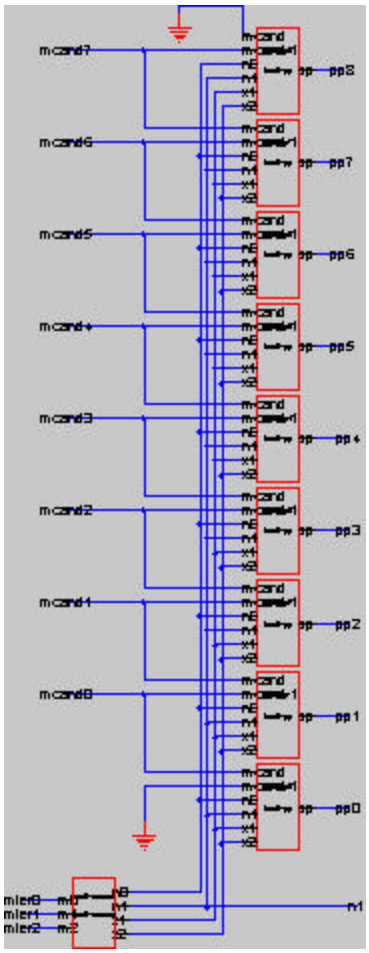
Booth-pp:



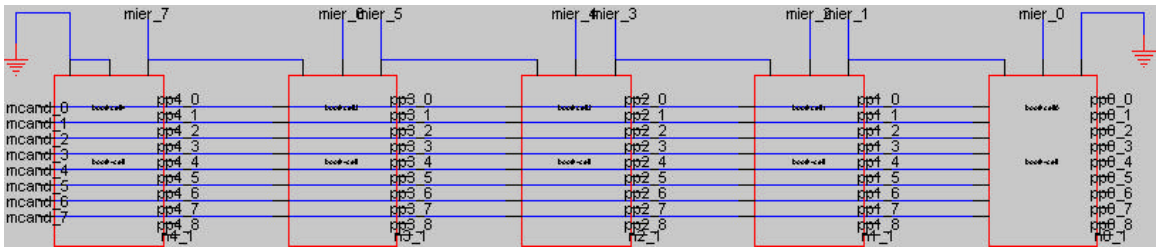
Booth-decode:



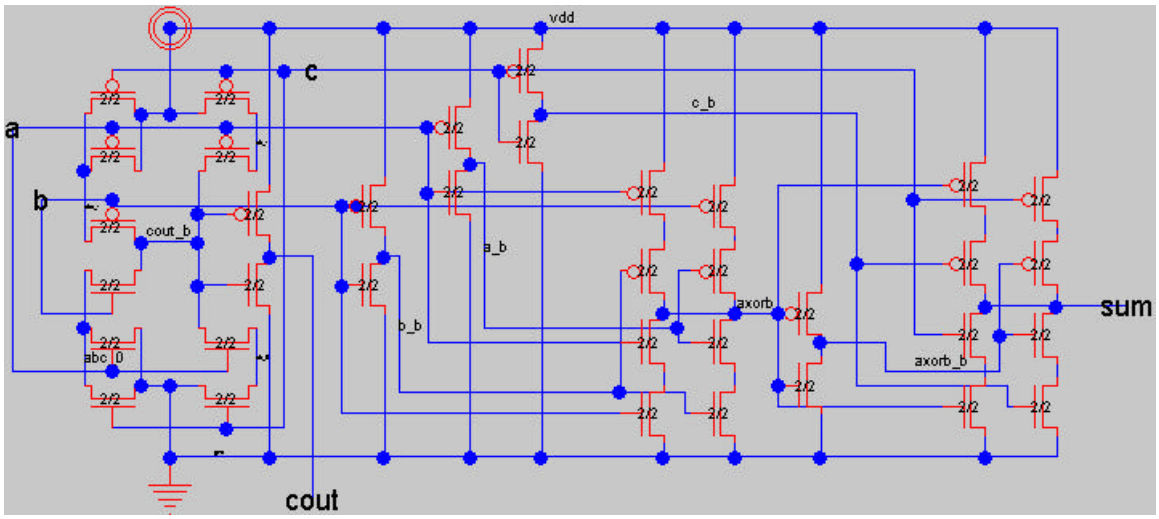
Booth-cell:



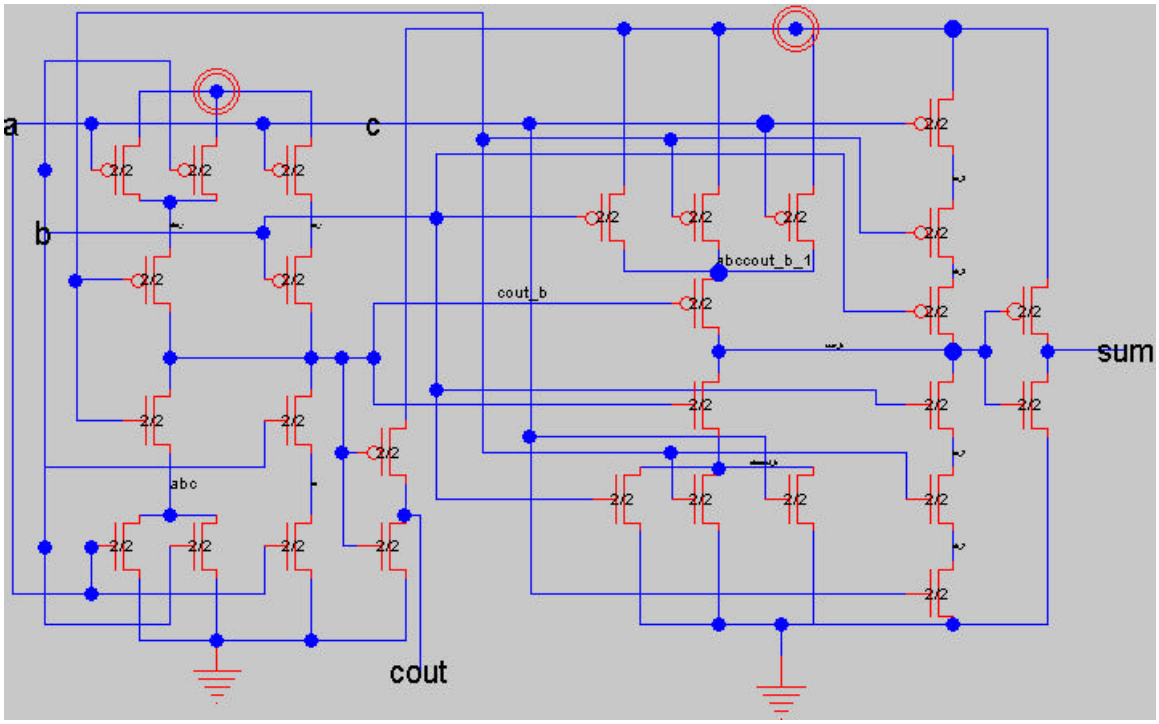
Booth-array:



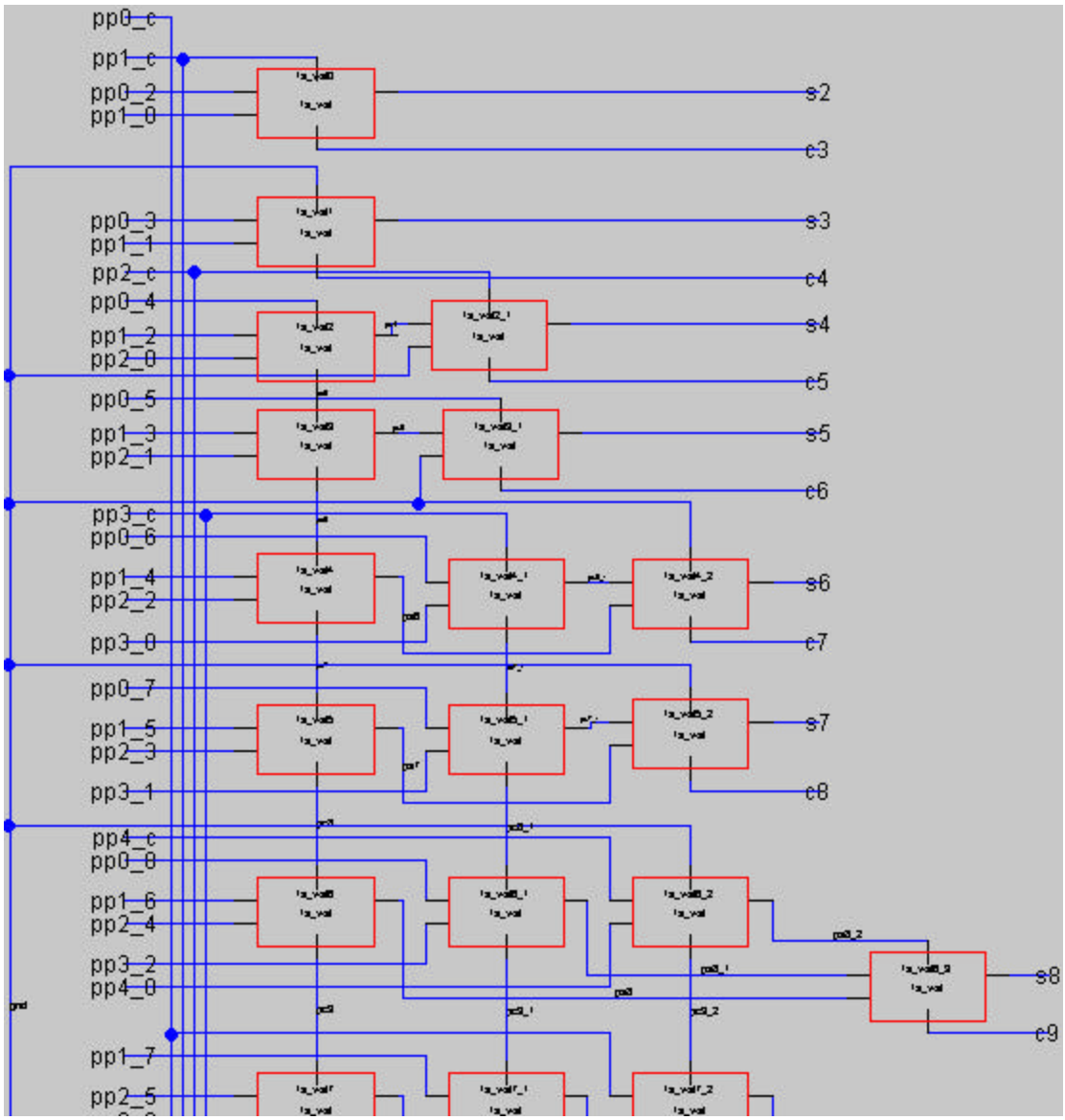
Full Adder: Wallace Tree



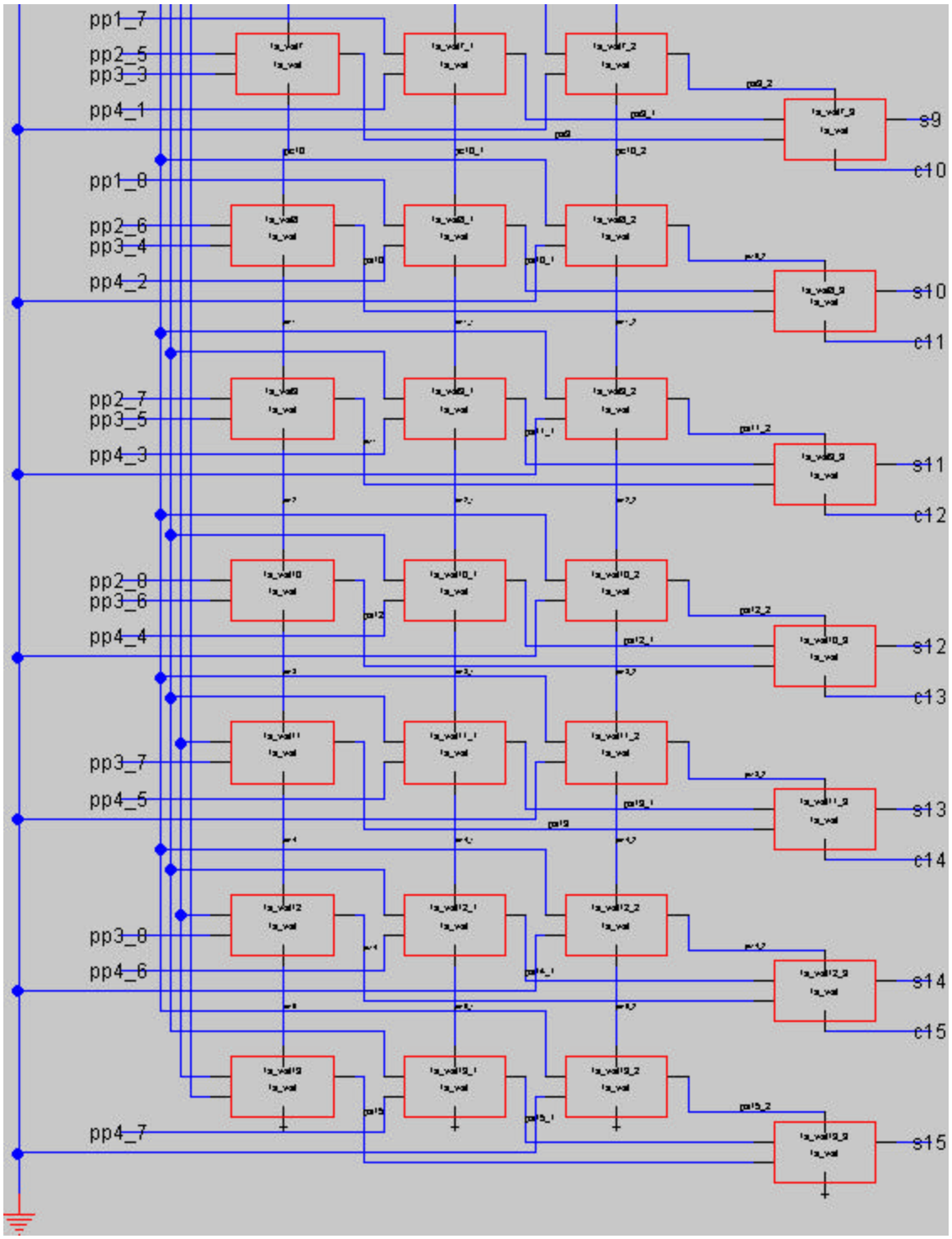
Full Adder: Ripple Carry Adder



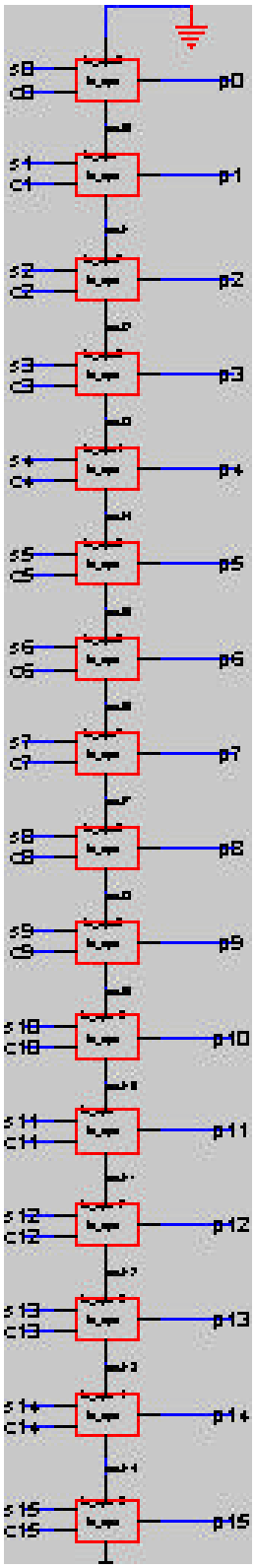
Wallace Tree (top half)



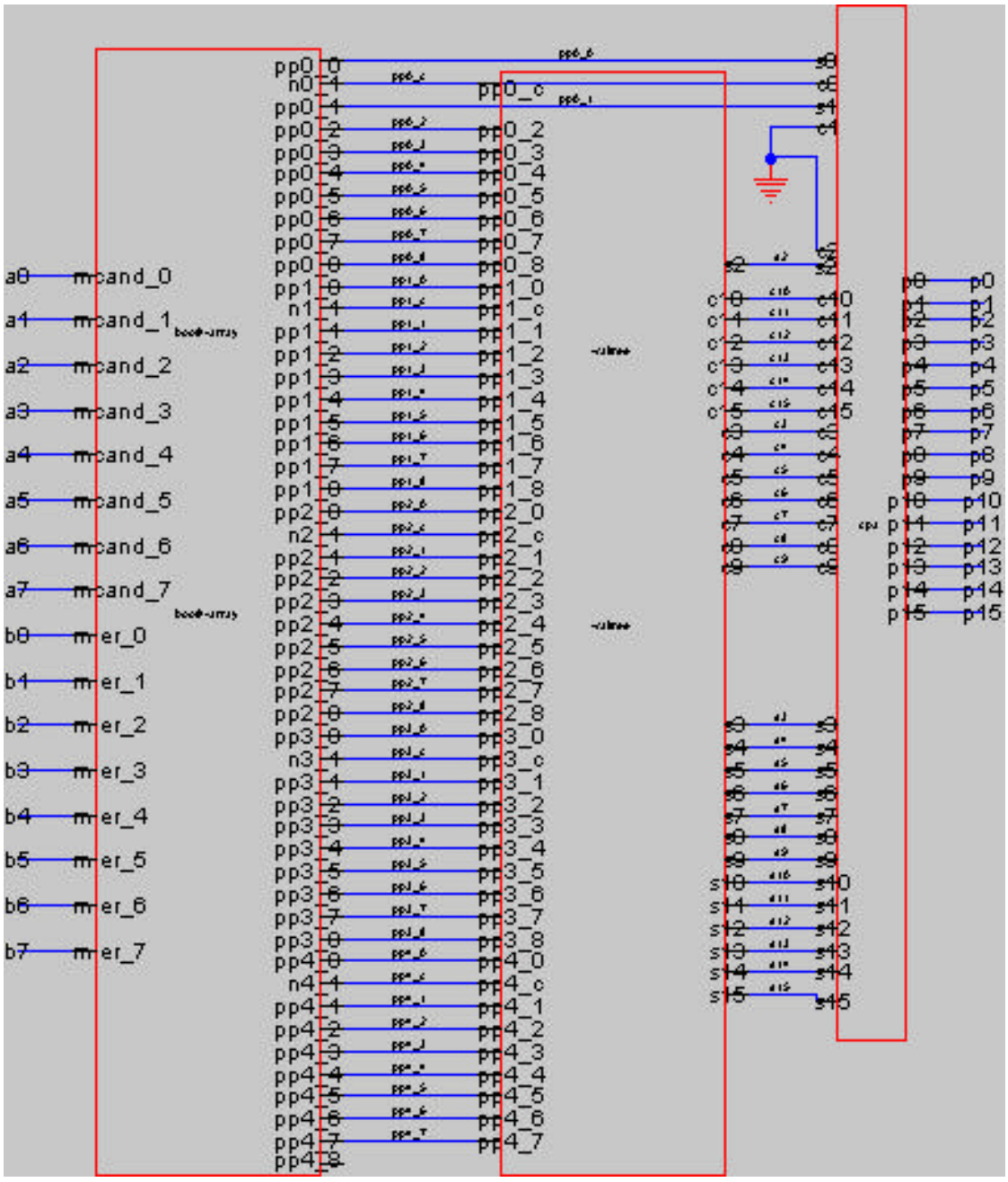
Wallace Tree (bottom half)



CPA

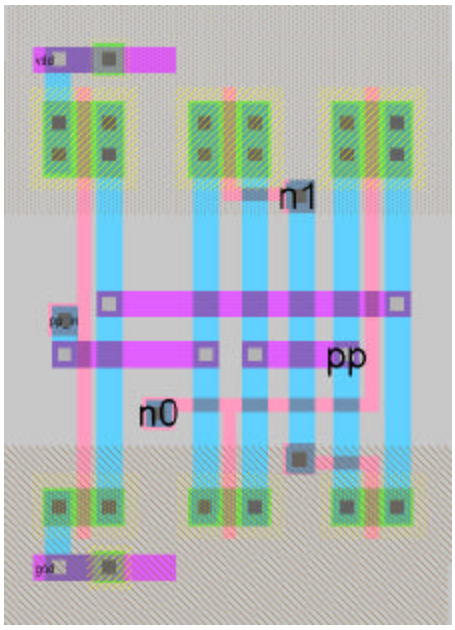

s0
c0
p0
s1
c1
p1
s2
c2
p2
s3
c3
p3
s4
c4
p4
s5
c5
p5
s6
c6
p6
s7
c7
p7
s8
c8
p8
s9
c9
p9
s10
c10
p10
s11
c11
p11
s12
c12
p12
s13
c13
p13
s14
c14
p14
s15
c15
p15

Multiplier

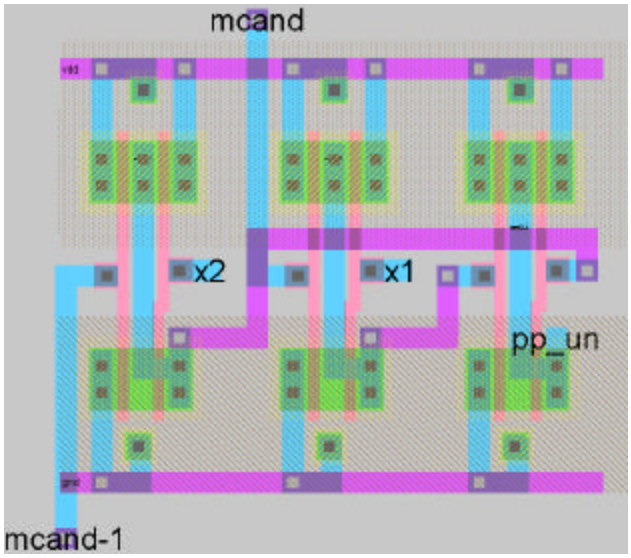

a0 mcand_0
a1 mcand_1
a2 mcand_2
a3 mcand_3
a4 mcand_4
a5 mcand_5
a6 mcand_6
a7 mcand_7
b0 mer_0
b1 mer_1
b2 mer_2
b3 mer_3
b4 mer_4
b5 mer_5
b6 mer_6
b7 mer_7
p0 p1 p2 p3 p4 p5 p6 p7 p8 p9 p10 p11 p12 p13 p14 p15

LAYOUT

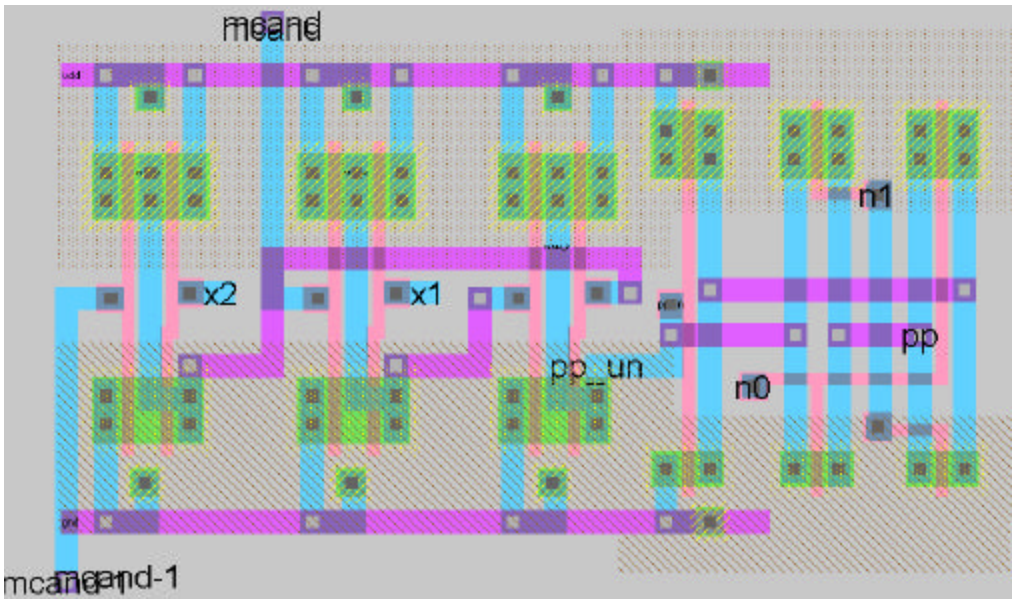
Pass-gate:



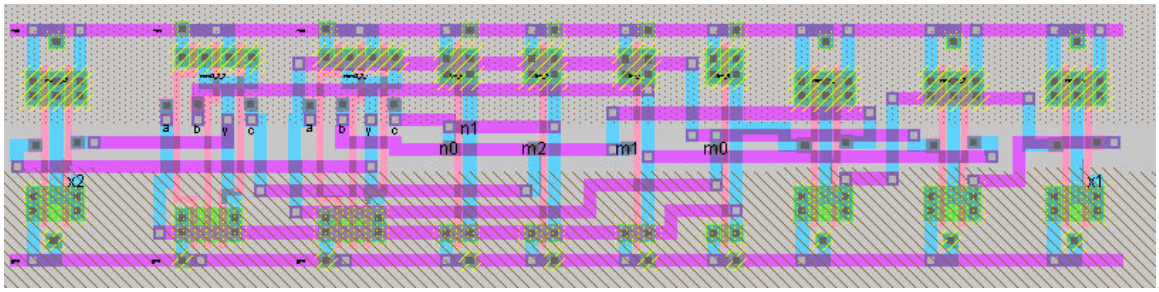
Test:



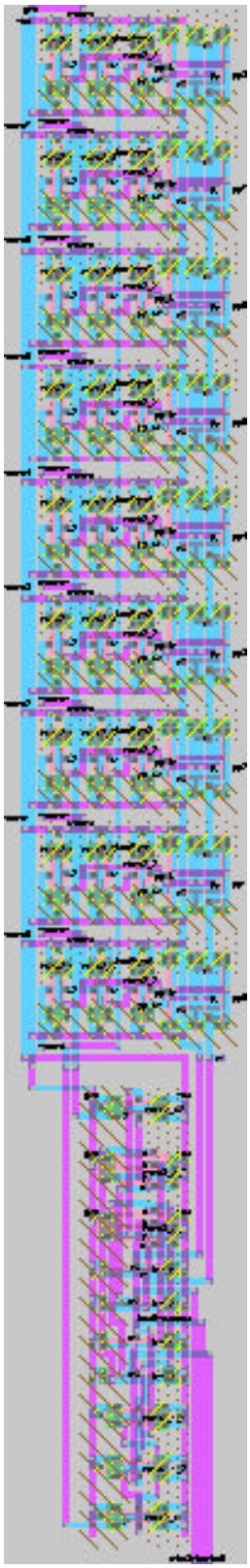
Booth-pp:



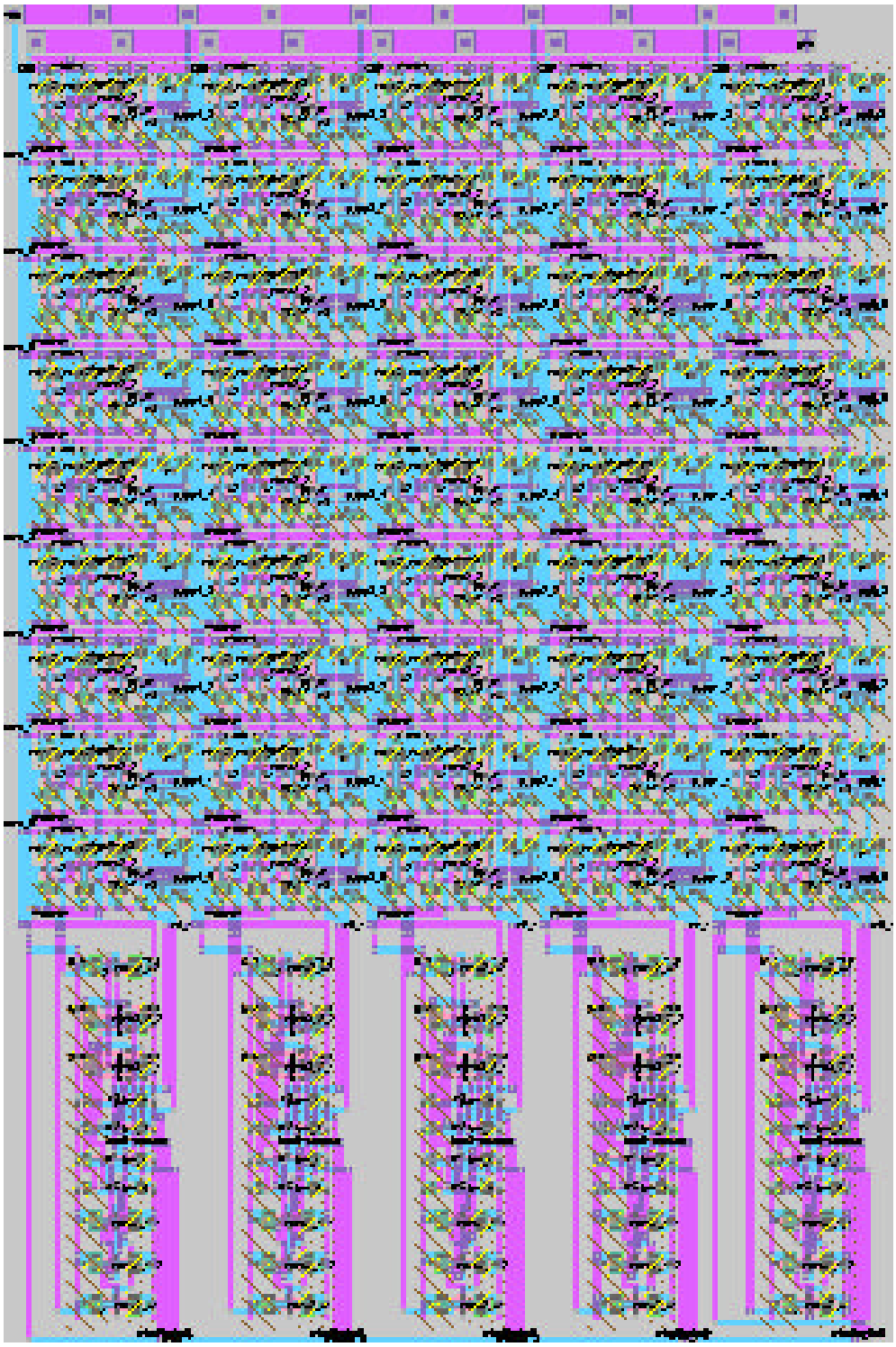
Booth-decode:



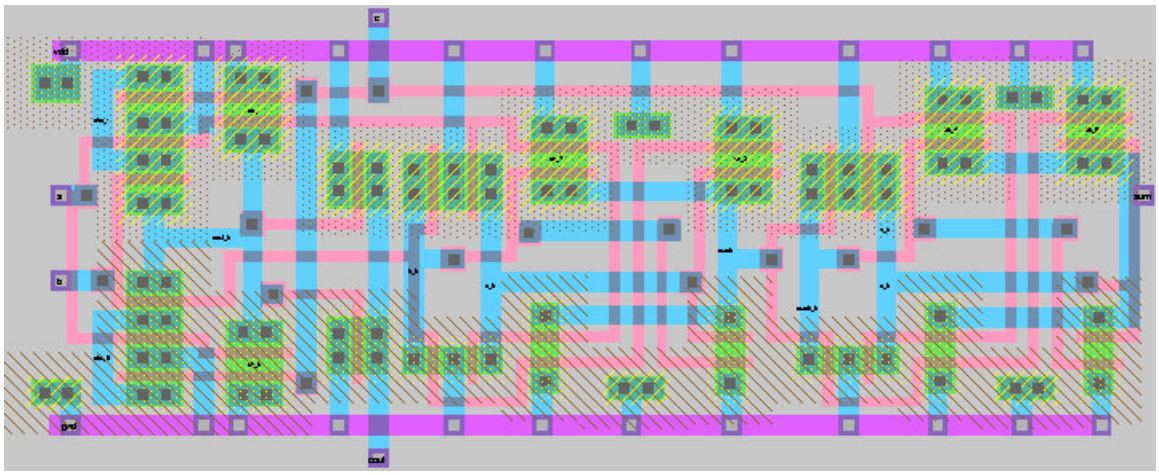
Booth-cell:



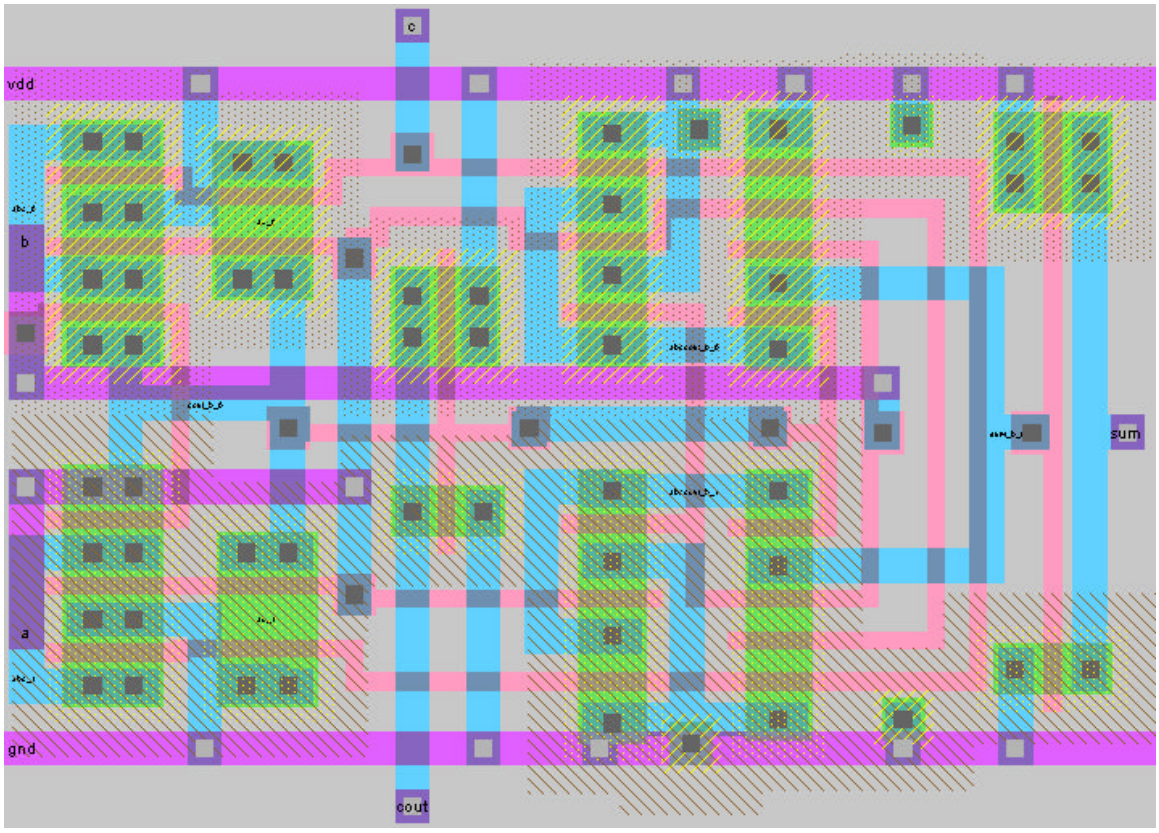
Booth-array:



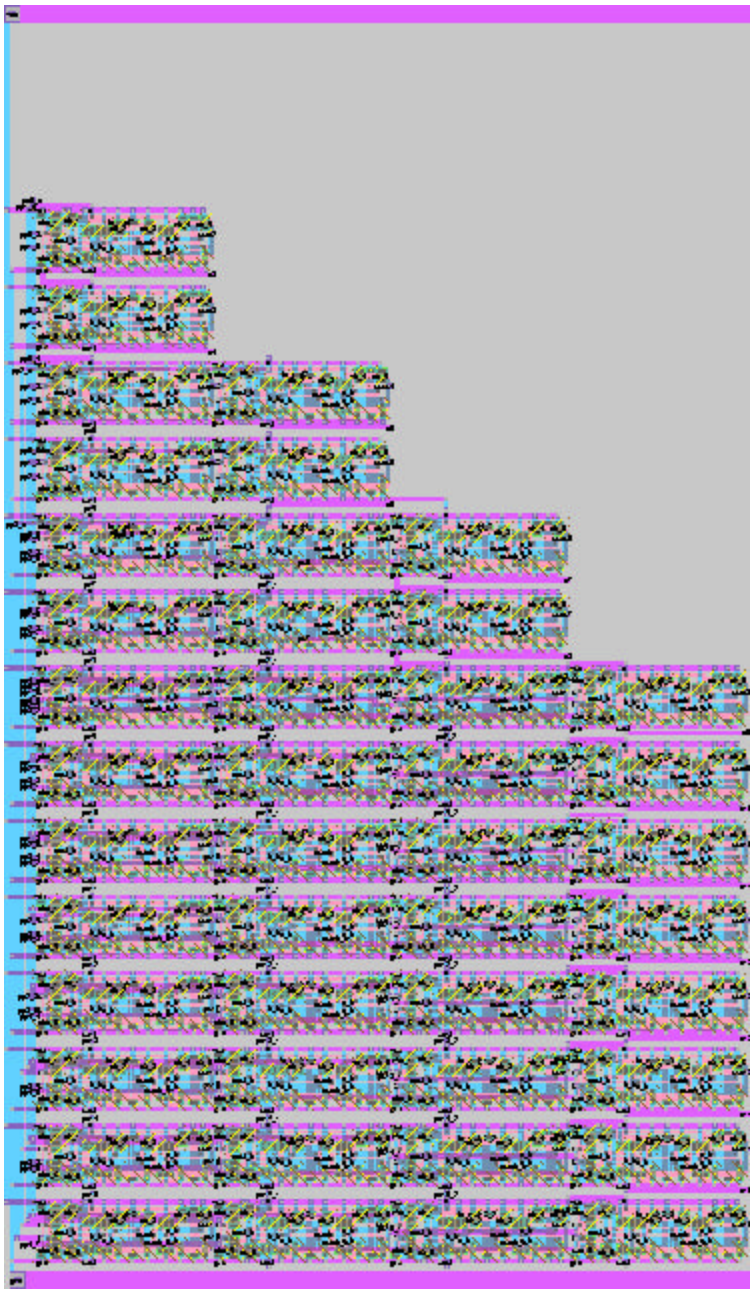
Full Adder: Wallace Tree



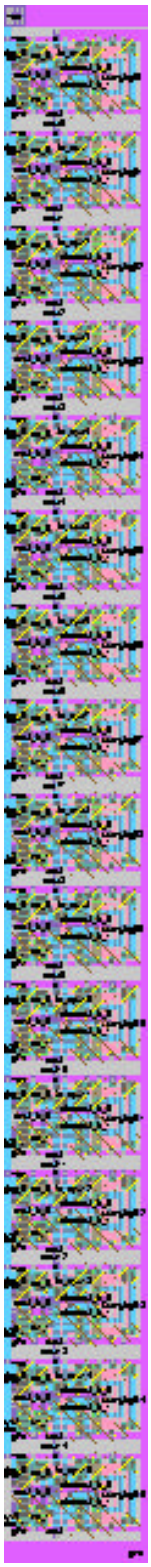
Full Adder: CPA



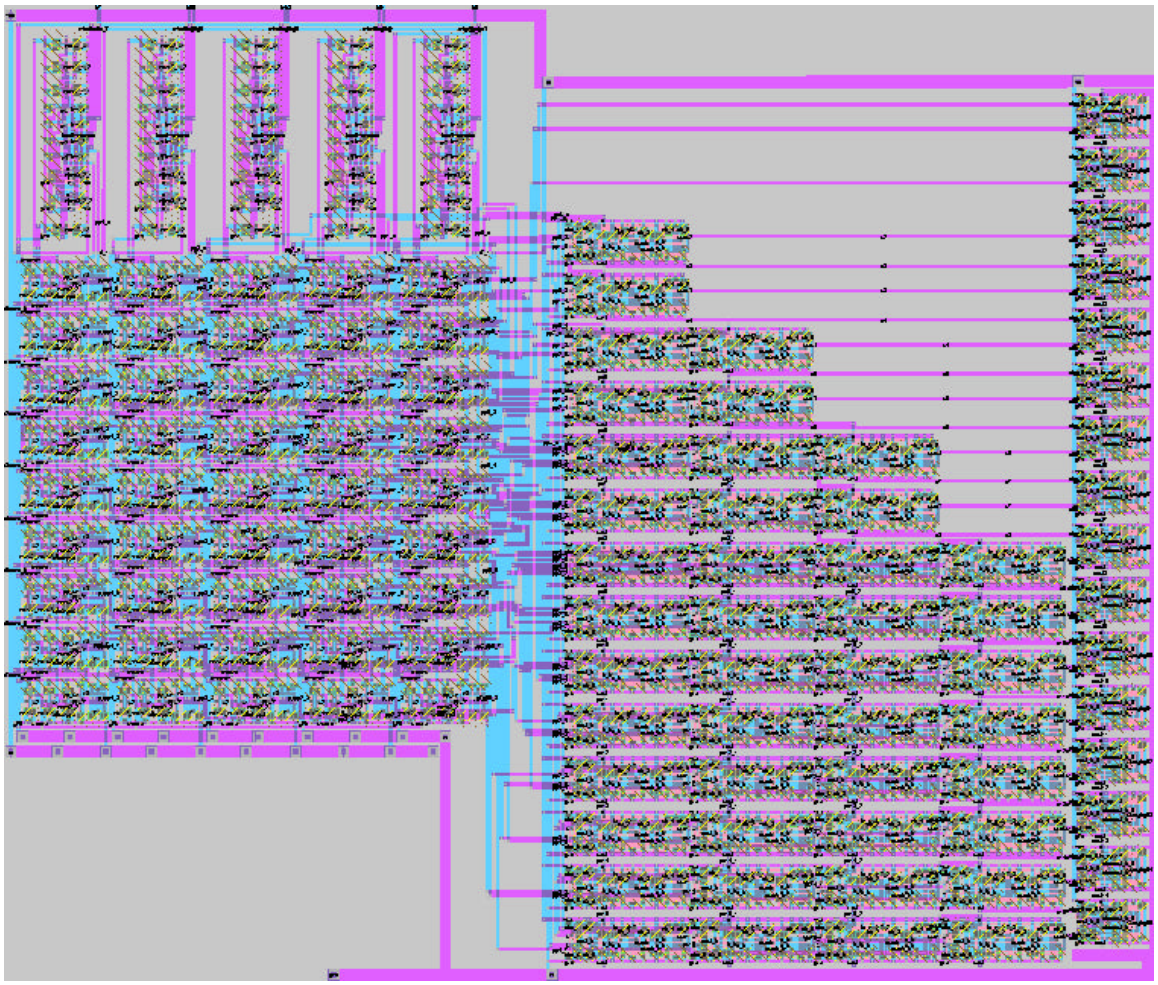
Wallace Tree



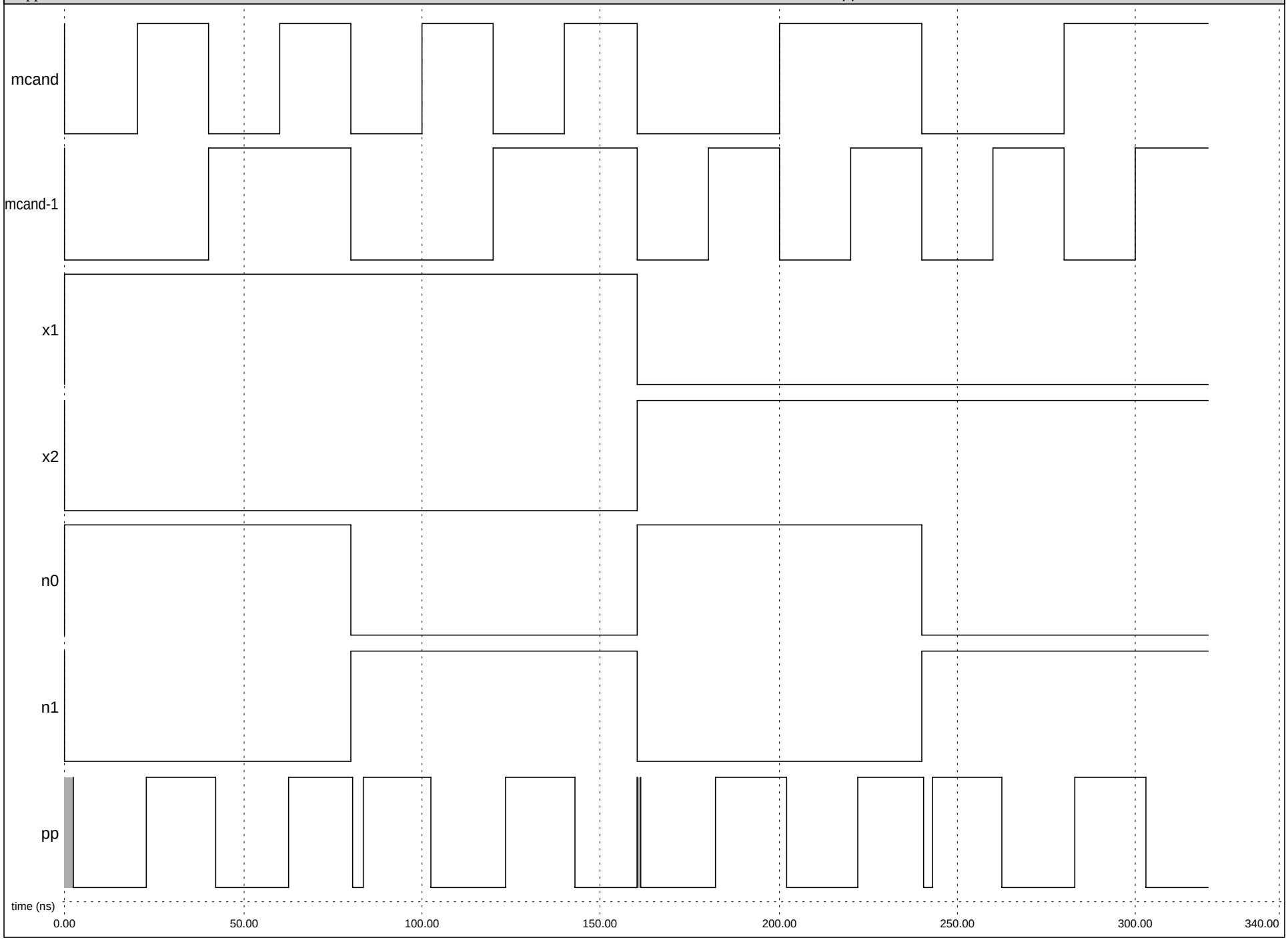
CPA

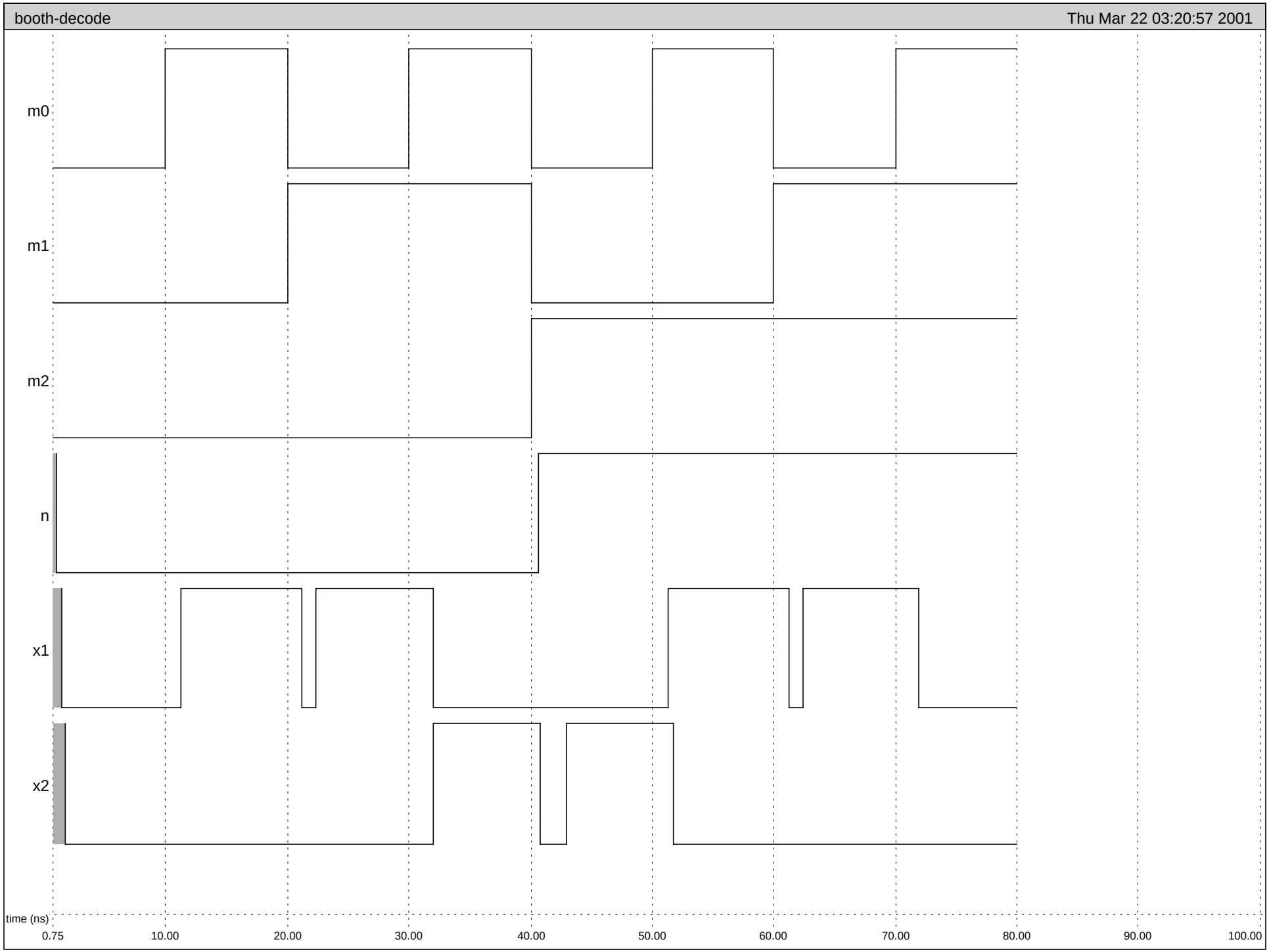


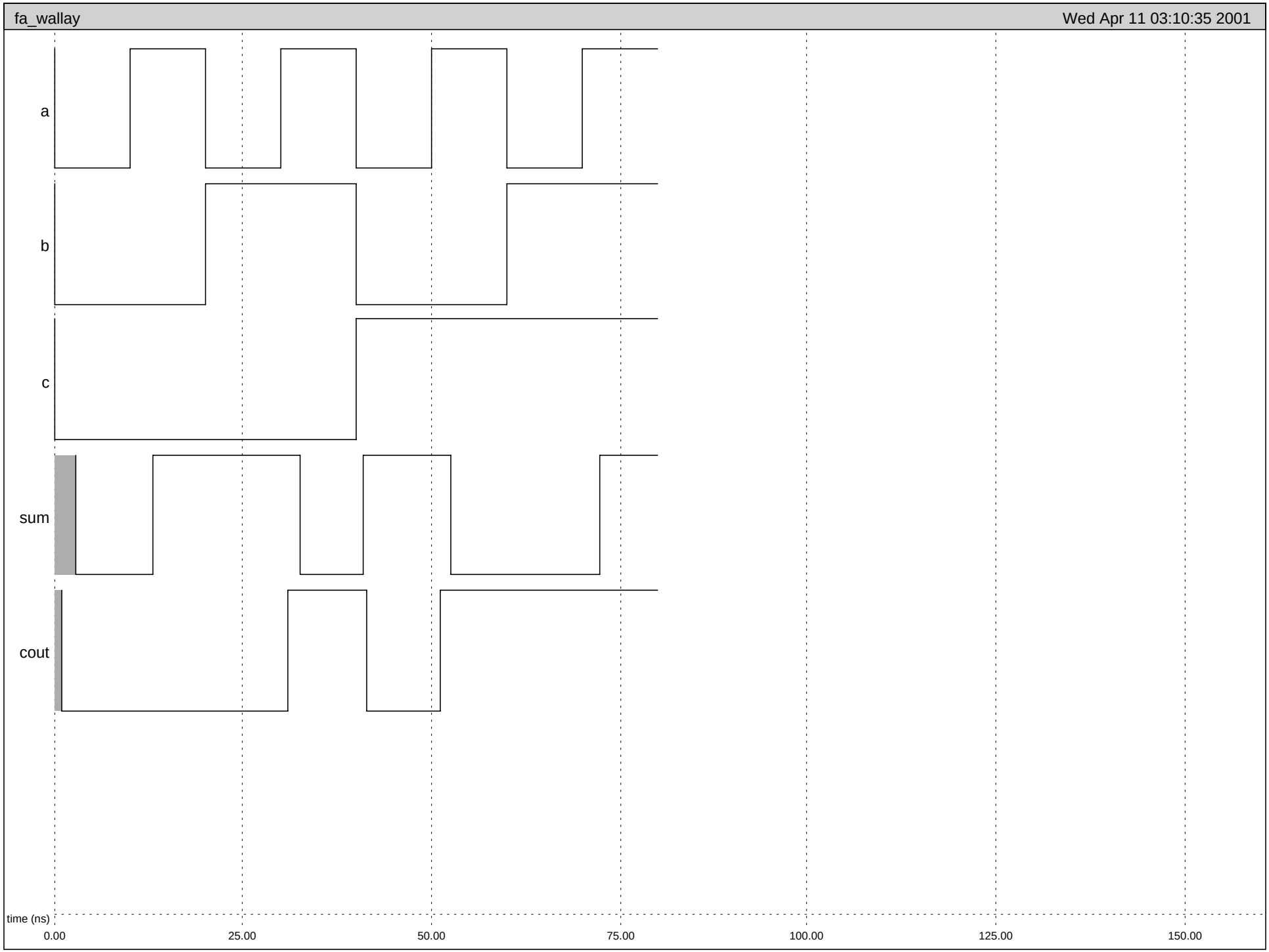
Multiplier

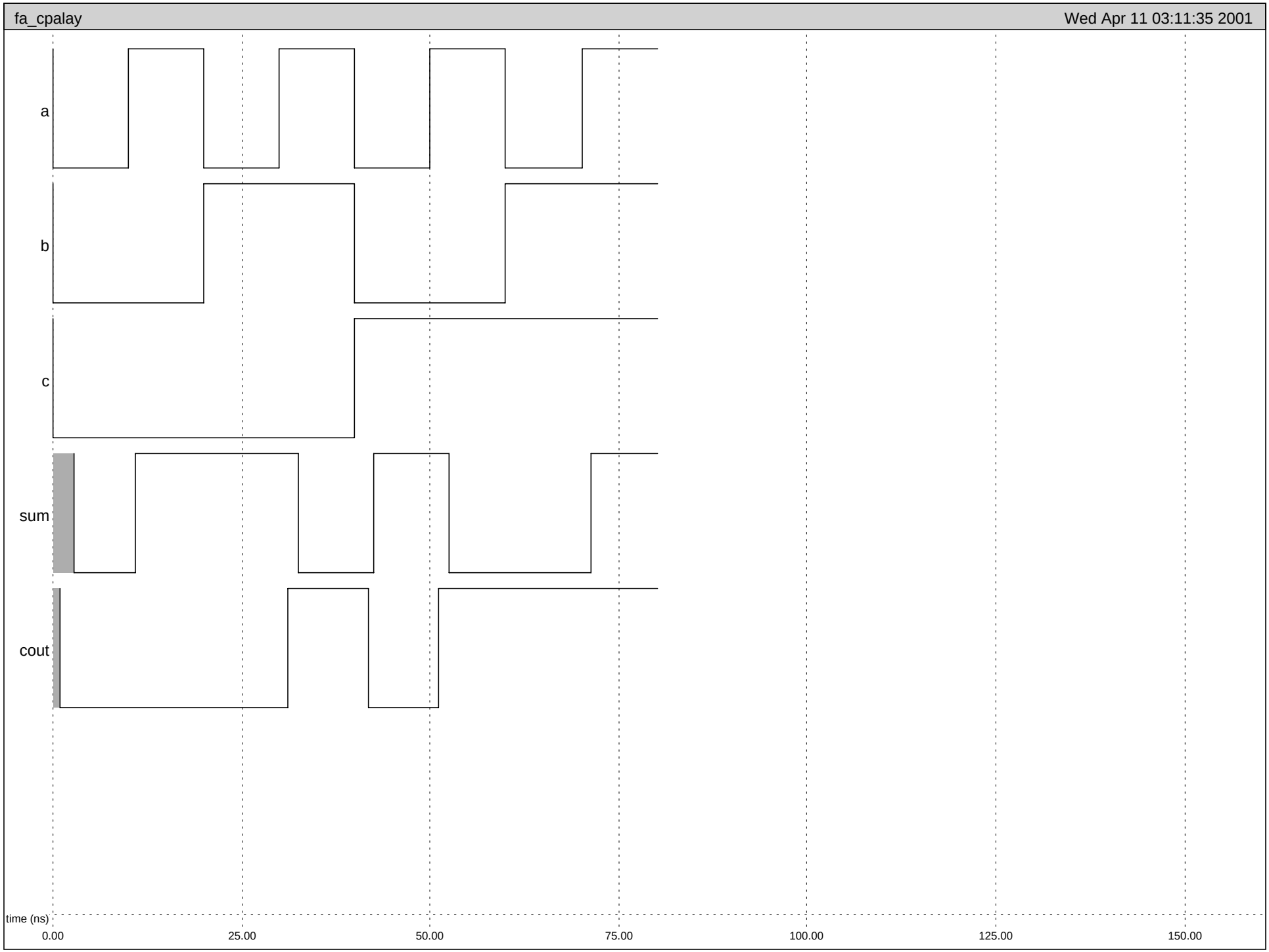


Appendix A: Simulation Waveforms




booth-decode
Thu Mar 22 03:20:57 2001
m0
m1
m2
n
x1
x2
time (ns)
0.75
10.00
20.00
30.00
40.00
50.00
60.00
70.00
80.00
90.00
100.00


fa_wallay
Wed Apr 11 03:10:35 2001
a
b
c
sum
cout
time (ns)
0.00
25.00
50.00
75.00
100.00
125.00
150.00


fa_cpalay
Wed Apr 11 03:11:35 2001
a
b
c
sum
cout
time (ns)
0.00
25.00
50.00
75.00
100.00
125.00
150.00

APPENDIX B: Java Test Bench

```
//cmdmaker.java
//Peter Grossmann
//Devised 30 March 2001
//Last modified 11 April 2001

//This file generates .cmd files for testing the multiplier.
//Admittedly, the techniques employed are crude and the code would
//definitely get marked down for style in any Mudd CS class.  However,
//it got the job done and didn't take inordinate amounts of time to
//write, which was good.

//The program works by typing the following sequence into the command
line
//
//java cmdmaker <test type> <filename>
//
//It then looks for a block it's generating a test for as either the
//booth array (booth), the wallace tree (waltree) or the full
multiplier
//(mult).  Based on which is selected, it generates an array of Strings
//where each String is a command line for a .cmd file.  The array of
//Strings is converted to a plain text file using some file I/O
routines I
//stole from old code of mine.  Feel free to steal them yourself if you
want;
//there's nothing magic, special or copywritten about them.

//There are three groups of commands generated:  "header" which
consists
//of defining input vectors and other setup stuff, "corner cases" for
//performing directed tests (A = B = 0; A = B = 11111111, A = B =
10101010
//for all blocks, plus all permutations of a single bit high in each
input
//for the booth array and the multiplier), and random tests where
inputs
//are allowed to vary over their full range.


import java.io.*;

class cmdmaker {

    public static void main(String args[]) {

      //Step 1 of the program is to determine how many lines long
      //the .cmd file will be to allocate space for it in the form
      //of a String array.  There are four parameters used here:

      //headerlines:  number of lines of pre-input toggling/simulation
      //commands

      //linespercase:  number of commands needed to perform each test
      //this is just the number of inputs plus one.
```

```
      //cornercases:  the number of corner cases

      //randomcases:  the number of random cases

      //endlines:  the number of endlines.  In this code, the only
      //necessary endline is "exit" so that the .cmd file also
      //terminates irsim (handy for automated test suites)

      if (args.length == 2) {
          boolean booth = false;
          boolean waltree = false;

          int headerlines = 7;
          int linespercase = 3;
          int cornercases = 259;

          if (args[0].equals("booth")) {
            headerlines = 15;
            booth = true;
          }
          else {
            if (args[0].equals("waltree")) {
                headerlines = 10;
                linespercase = 7;
                cornercases = 2;
                waltree = true;
            }
          }

          int randomcases = 100;
          int endlines = 1;

          String[] cmdlines = new String[headerlines +
                                linespercase * (cornercases +
                                            randomcases)
                                + endlines];


          //Header required for the booth test:  definition of
          //input vectors A[7:0], B[7:0] and output vectors
          //PP0[8:0], PP1[8:0], PP2[8:0], PP3[8:0] and PP4[8:0],
          //the five partial products, plus five sign/carry bits.
          //If the partial product is to be subtracted, then
          //the multiplier negates them conforming to twos complement
          //form.  Thus the partial products need to be sign extended
          //and have one added to them if they are negative (following
          //their inversion).  Since the sign is always equal to a
negation
          //signal, and the one is added only if the negation signal is
          //high, it makes sense to simply feed this negation signal to
          //the wallace tree and add it in whatever columns it is
needed.
          //This is what the multiplier does so the test needs to
account
          //for that.
```

```
          //One additional header line (very important to the overall
test
          //plan) is the watch ("w") command applied to all i/o
vectors.
          //This causes IRSIM to display on screen the value of each
vector
          //being watched.  It's then no trick to pipe the screen
output
          //into a text file and process the file with a parser (see
          //multester.java; that's what it does).

          //note here that the code hard-writes the signal names that
we
          //used, so if you were to adapt this code for your own use,
you'd
          //need to rewrite the header to suit your own project.

          cmdlines[0] = "stepsize 100\n";
          if (booth) {
            cmdlines[1] = "vector PP0 pp0_8 pp0_7 pp0_6 pp0_5 pp0_4
pp0_3 pp0_2 pp0_1 pp0_0\n";
            cmdlines[2] = "vector PP1 pp1_8 pp1_7 pp1_6 pp1_5 pp1_4
pp1_3 pp1_2 pp1_1 pp1_0\n";
            cmdlines[3] = "vector PP2 pp2_8 pp2_7 pp2_6 pp2_5 pp2_4
pp2_3 pp2_2 pp2_1 pp2_0\n";
            cmdlines[4] = "vector PP3 pp3_8 pp3_7 pp3_6 pp3_5 pp3_4
pp3_3 pp3_2 pp3_1 pp3_0\n";
            cmdlines[5] = "vector PP4 pp4_8 pp4_7 pp4_6 pp4_5 pp4_4
pp4_3 pp4_2 pp4_1 pp4_0\n";
            cmdlines[6] = "vector PPC n4_1 n3_1 n2_1 n1_1 n0_1\n";

            cmdlines[7] = "vector B4 mier_7\n";
            cmdlines[8] = "vector B3 mier_7 mier_6 mier_5 \n";
            cmdlines[9] = "vector B2 mier_5 mier_4 mier_3\n";
            cmdlines[10] = "vector B1 mier_3 mier_2 mier_1\n";
            cmdlines[11] = "vector B0 mier_1 mier_0 gnd\n";

            cmdlines[12] = "vector B mier_7 mier_6 mier_5 mier_4 mier_3
mier_2 mier_1 mier_0\n";
            cmdlines[13] = "vector A mcand_7 mcand_6 mcand_5 mcand_4
mcand_3 mcand_2 mcand_1 mcand_0\n";
            cmdlines[14] = "w PPC PP4 PP3 PP2 PP1 PP0 B4 B3 B2 B1 B0 B
A\n";
          }

          else {


            //Wallace tree test header:  Input vectors are the five
            //partial products; outputs are a sum vector and a carry
            //vector.  Note that the handful of lower bits that don't
need
            //to be sent through the Wallace tree aren't represented
here.

            if (waltree) {
```

```
                cmdlines[1] = "vector PP0 pp0_8 pp0_7 pp0_6 pp0_5 pp0_4
pp0_3 pp0_2\n";
                cmdlines[2] = "vector PP1 pp1_8 pp1_7 pp1_6 pp1_5 pp1_4
pp1_3 pp1_2 pp1_1 pp1_0\n";
                cmdlines[3] = "vector PP2 pp2_8 pp2_7 pp2_6 pp2_5 pp2_4
pp2_3 pp2_2 pp2_1 pp2_0\n";
                cmdlines[4] = "vector PP3 pp3_8 pp3_7 pp3_6 pp3_5 pp3_4
pp3_3 pp3_2 pp3_1 pp3_0\n";
                cmdlines[5] = "vector PP4 pp4_7 pp4_6 pp4_5 pp4_4 pp4_3
pp4_2 pp4_1 pp4_0\n";
                cmdlines[6] = "vector PPC pp4_c pp3_c pp2_c pp1_c\n";
                cmdlines[7] = "vector S s15 s14 s13 s12 s11 s10 s9 s8
s7 s6 s5 s4 s3 s2\n";
                cmdlines[8] = "vector C c15 c14 c13 c12 c11 c10 c9 c8
c7 c6 c5 c4 c3\n";
                cmdlines[9] = "w C S PPC PP4 PP3 PP2 PP1 PP0\n";
            }

            //multiplier test header is simple:  Inputs A and B, and
            //output P the product.  Since the Wallace tree is the
            //toughest part of this to get right, it's also nice to
            //have the sum and carry values it generates displayed

            else { //multiplier test

                cmdlines[1] = "vector A a7 a6 a5 a4 a3 a2 a1 a0\n";
                cmdlines[2] = "vector B b7 b6 b5 b4 b3 b2 b1 b0\n";
                cmdlines[3] = "vector P p15 p14 p13 p12 p11 p10 p9 p8
p7 p6 p5 p4 p3 p2 p1 p0\n";
                cmdlines[4] = "vector s s15 s14 s13 s12 s11 s10 s9 s8
s7 s6 s5 s4 s3 s2\n";
                cmdlines[5] = "vector c c15 c14 c13 c12 c11 c10 c9 c8
c7 c6 c5 c4 c3\n";
                cmdlines[6] = "w C S P B A\n";
            }
          }

          //corner cases:
          //-----------------------------------------------

          //Wallace Tree corners:  all zeros and all ones:
          if (waltree) {
            cmdlines[headerlines] = "set PP0 0000000\n";
            cmdlines[headerlines + 1] = "set PP1 000000000\n";
            cmdlines[headerlines + 2] = "set PP2 000000000\n";
            cmdlines[headerlines + 3] = "set PP3 000000000\n";
            cmdlines[headerlines + 4] = "set PP4 00000000\n";
            cmdlines[headerlines + 5] = "set PPC 0000\n";
            cmdlines[headerlines + 6] = "s\n";
            cmdlines[headerlines + 7] = "set PP0 1111111\n";
            cmdlines[headerlines + 8] = "set PP1 111111111\n";
            cmdlines[headerlines + 9] = "set PP2 111111111\n";
            cmdlines[headerlines + 10] = "set PP3 111111111\n";
            cmdlines[headerlines + 11] = "set PP4 11111111\n";
            cmdlines[headerlines + 12] = "set PPC 1111\n";
            cmdlines[headerlines + 13] = "s\n";
          }
```

```
//booth and multiplier corner cases:
else {

  cmdlines[headerlines] = "set A 00000000\n";
  cmdlines[headerlines + 1] = "set B 00000000\n";
  cmdlines[headerlines + 2] = "s\n";
  cmdlines[headerlines + 3] = "set A 11111111\n";
  cmdlines[headerlines + 4] = "set B 11111111\n";
  cmdlines[headerlines + 5] = "s\n";
  cmdlines[headerlines + 6] = "set A 10101010\n";
  cmdlines[headerlines + 7] = "set B 10101010\n";
  cmdlines[headerlines + 8] = "s\n";
  char[] AString = {'0', '0', '0', '0', '0', '0', '0', '0'};
  char[] BString = {'0', '0', '0', '0', '0', '0', '0', '0'};
  int indexstart = headerlines + 9;

  //place a single 1 in A and in B and then shift them
  //through each permutation:
  for (int index = 7; index >= 0; index--) {
      AString[index] = '1';

      for (int indexb = 7; indexb >= 0; indexb--) {
        BString[indexb] = '1';

        int i = linespercase * (8 * (7 - index) +
                          (7 - indexb)) + indexstart;
        cmdlines[i] = "set A "+new String(AString)+"\n";
        cmdlines[i+1] = "set B "+new String(BString)+"\n";
        cmdlines[i+2] = "s\n";
        BString[indexb] = '0';
      }
      AString[index] = '0';
  }

}

//random cases:
//-----------------------------------------------

for (int index = headerlines + linespercase * cornercases;
   index < cmdlines.length - endlines;
   index = index + linespercase) {

  //for each n-bit input, select a random bit value
  //between 0 and 2^n - 1.  Then convert it from an
  //integer into a binary String

  if (waltree) {
      long pp0 = Math.round(127.0 * Math.random());
      long pp1 = Math.round(511.0 * Math.random());
      long pp2 = Math.round(511.0 * Math.random());
      long pp3 = Math.round(511.0 * Math.random());
      long pp4 = Math.round(255.0 * Math.random());
      long ppc = Math.round(15.0 * Math.random());
      String pp0String = longtobin(pp0, 7);
      String pp1String = longtobin(pp1, 9);
```

```
                String pp2String = longtobin(pp2, 9);
                String pp3String = longtobin(pp3, 9);
                String pp4String = longtobin(pp4, 8);
                String ppcString = longtobin(ppc, 4);
                cmdlines[index] = "set PP0 " + pp0String + "\n";
                cmdlines[index + 1] = "set PP1 " + pp1String + "\n";
                cmdlines[index + 2] = "set PP2 " + pp2String + "\n";
                cmdlines[index + 3] = "set PP3 " + pp3String + "\n";
                cmdlines[index + 4] = "set PP4 " + pp4String + "\n";
                cmdlines[index + 5] = "set PPC " + ppcString + "\n";
                cmdlines[index + 6] = "s\n";
            }
            else {
                long a = Math.round(255.0 * Math.random());
                long b = Math.round(255.0 * Math.random());
                String aString = longtobin(a, 8);
                String bString = longtobin(b, 8);
                cmdlines[index] = "set A " + aString + "\n";
                cmdlines[index + 1] = "set B " + bString + "\n";
                cmdlines[index + 2] = "s\n";
            }
        }

        cmdlines[cmdlines.length - 1] = "exit\n";

        //The commands are set:  time to write the file:
        writeFile(args[1], cmdlines);
    }

    else {
        System.out.println("Error, wrong number of arguments\n");
    }
  }

  //The rest of the code is helper methods:

  //longtobin accepts an integer and a number of bits N and returns
an
  //N-bit representation of that integer:
  public static String longtobin(long longval, int numdigits) {

    //System.out.println(longval);

    char[] binarray = new char[numdigits];
    numdigits = numdigits - 1;

    //build the character array from LSB to MSB:
    while (longval > 0) {
        long curdigit = longval % 2;
        //System.out.println(curdigit);
        if (curdigit == 1) {
          binarray[numdigits] = '1';
        }
        else {
          binarray[numdigits] = '0';
        }
```

```
            longval = (longval - curdigit) / 2;
            //System.out.println(longval);
            numdigits = numdigits - 1;
        }

        //pad the value with zeros so that it's always an 8-bit value
        while (numdigits >= 0) {

            binarray[numdigits] = '0';
            numdigits = numdigits - 1;
        }

        return new String(binarray);
    }

    //This function converts a String array into a text file with
    //one string per line.  As can be seen, most of the work is
    //done by standard Java packages.
    public static void writeFile(String filename, String[] fileData) {

        try {
            BufferedWriter out = new BufferedWriter(
                                        new FileWriter(filename));


            for (int index = 0; index < fileData.length; index++) {

                  if (fileData[index] != null) {
                  out.write(fileData[index], 0,
fileData[index].length());
                  }
            }
            out.close();

            System.out.println(filename + " successfully written.");
        }

        catch (IOException toss) {
            System.out.println("IOException at writeFile; can't build
BufferedWriter.");
        }
    }
}
```

```
//multester.java
//Peter Grossmann
//Devised 30 March 2001
//Last Modified 11 April 2001

//This program is a test utility for an 8-bit booth-encoded multiplier

//The program works by typing the following sequence into the command
line
//
//java multester <test block> <filename>
//
//where test block is either the booth array, the wallace tree, or the
//multiplier, and filename is the name of a text file containing
//piped IRSIM screen output.  The program reads the file and, based on
//which block is being tested, looks for lines in the text file that
//correspond to test results.  It translates those lines into inputs
and
//received outputs, generates expected values of outputs for those
inputs,
//compares expected to received outputs, and prints error messages
//when results disagree.  I will note up front that the implementation
is
//overly-directive, makes a lot of assumptions about text file format.
//Programming power and style was sacrificed significantly in the
interest
//of rapid development.  However, when used in conjunction with
simulations
//run using cmdmaker.java, it works, and therefore does exactly what it
needs
//to do.

import java.io.*;

class multester {

    public static void main(String args[]) {

      //see which test is being requested and run it:

      if (args.length == 2) {

          if (args[0].equals("mult")) {

            multtest(args[1]);
          }

          else {

            if (args[0].equals("booth")) {

                boothtest(args[1]);
            }
            else {
                if (args[0].equals("waltree")) {

                  waltreetest(args[1]);
```

```
                }
                else {
                  System.out.println("Error; unsupported feature");
                }
              }
            }

      }

      else {
          System.out.println("Error; wrong number of arguments");
      }

    }


    //The multiplier test reads in inputs A and B and output P and
verifies
    //whether A * B = P for A and B treated as unsigned numbers:
    public static void multtest(String filename) {
      try {
          FileInputStream results =
            new FileInputStream(filename);
          String[] tests = readFileLines(results);

          int errors = 0;
          int wirebugs = 0;

          for (int index = 0; index < tests.length; index++) {

            //read the file as follows:  each line of the text
            //file that begins with the sequence "A=" should be
            //taken as a line containing a set of input and output
            //vectors.  The vectors are assumed to be in the order
            //A B P s c where s and c are the sum and carry output
            //of the Wallace tree.  This works with the cmdmaker.java-
            //generated tests because it always puts the watch so that
            //vectors appear in that order, and they are sufficiently
            //small vectors that they fit on one line of text (note,
            //however, the lack of error checking here).  It's then
            //a matter of parsing the line into integers that can
            //be compared, and outputting error messages when
            //the comparison fails.

            if (tests[index].startsWith("A=")) {

                //parser decomposes line of text into an array of
                //words, which are then split at the equals sign
                //to extract the binary values of the i/o:
                String[] iovectors =
                  parseCommand(tests[index]);
                String[] Aval = split(iovectors[0], "=");
                String[] Bval = split(iovectors[1], "=");
                String[] Pval = split(iovectors[2], "=");
                String[] sval = split(iovectors[3], "=");
                String[] cval = split(iovectors[4], "=");
                int A = bintoint(Aval[1]);
```

```
                int B = bintoint(Bval[1]);
                int P = bintoint(Pval[1]);
                int s = bintoint(sval[1]);
                int c = bintoint(cval[1]);


                if (A * B != P) {
                  //log number of errors encountered.
                  errors++;
                  System.out.print("Case: ");
                  System.out.println(tests[index]);
                  //print expected and actual values in hex:
                  System.out.print("Expected: ");
                  System.out.print(inttohex(A * B));
                  System.out.print(" Got: ");
                  System.out.println(inttohex(P));
                }

              }
              else {
                  //System.out.println(tests[index]);
              }
            }
            //report a total number of errors at the end
            System.out.println("Total Errors: "+errors);
      }

      catch(IOException toss) {
          System.out.println("Error; file not found");
      }

    }


    //the booth test performs a number of operations.  It naturally
reads
    //inputs A and B, computes partial products, and checks to see that
those
    //are as expected.  However, as an extra check, it also verifies
that
    //the sum of the partial products equals the product of the inputs
    //as unsigned numbers, both from expected and actual partial
products.
    //These errors are counted and displayed separately; thus for most
    //failed cases multiple errors will appear:

    public static void boothtest(String filename) {
      try {
          FileInputStream results =
            new FileInputStream(filename);
          String[] tests = readFileLines(results);

          int errors = 0;        //discrepancy between sum of computed
                                 //partial products
                                 //and sum of hardware partial
products.
          int adderrors = 0;     //discrepacncy between sum of partial
```

```
                        //products and product of A and B
            int pperrors = 0;      //discrepancy between expected and
                                   //computed individual partial products

            for (int index = 0; index < tests.length; index++) {

              if (tests[index].startsWith("A=")) {

                   //irsim puts in carriage returns, so the data
                   //for this set of tests comes in on two lines:
                   String[] iovectors =  parseCommand(tests[index]);
                   String[] iovectors2 = parseCommand(tests[index + 1]);

                   String[] Aval = split(iovectors[0], "=");
                   String[] Bval = split(iovectors[1], "=");
                   String[] B0val = split(iovectors[2], "=");
                   String[] B1val = split(iovectors[3], "=");
                   String[] B2val = split(iovectors[4], "=");
                   String[] B3val = split(iovectors[5], "=");
                   String[] B4val = split(iovectors[6], "=");
                   String[] PP0val = split(iovectors[7], "=");

                   //data from second line of displays:
                   String[] PP1val = split(iovectors2[0], "=");
                   String[] PP2val = split(iovectors2[1], "=");
                   String[] PP3val = split(iovectors2[2], "=");
                   String[] PP4val = split(iovectors2[3], "=");
                   String[] PPCval = split(iovectors2[4], "=");

                   //conversion of binary String to integer:
                   int A = bintoint(Aval[1]);
                   int B = bintoint(Bval[1]);
                   int B0 = bintoint(B0val[1]);
                   int B1 = bintoint(B1val[1]);
                   int B2 = bintoint(B2val[1]);
                   int B3 = bintoint(B3val[1]);
                   int B4 = bintoint(B4val[1]);
                   int PP0 = bintoint(PP0val[1]);
                   int PP1 = bintoint(PP1val[1]);
                   int PP2 = bintoint(PP2val[1]);
                   int PP3 = bintoint(PP3val[1]);
                   int PP4 = bintoint(PP4val[1]);

                   //Compute the partial products expected based
                   //on A and B
                   int PP0expected = computePP(A, B0);
                   int PP1expected = computePP(A, B1);
                   int PP2expected = computePP(A, B2);
                   int PP3expected = computePP(A, B3);
                   int PP4expected = computePP(A, B4);
                   int PPCexpected = computecarries(Bval[1]);


                   //Perform sign extension if necessary (meaning whenever
                   //the most significant of the three bits being checked
            for
                   //the partial product goes high).
```

```
                if (Bval[1].charAt(6) == '1') {
                  PP0 = PP0 + 65024;
                }
                if (Bval[1].charAt(4) == '1') {
                  PP1 = PP1 + 63488 / 4;
                }
                if (Bval[1].charAt(2) == '1') {
                  PP2 = PP2 + 57344 / 16;
                }
                if (Bval[1].charAt(0) == '1') {
                  PP3 = PP3 + 32768 / 64;
                }

                //compute three different products:  A * B
                //and sum of computed and received partial products.

                int Pexpected = A * B;

                int Pcomputed = PP0expected + 4 * PP1expected + 16
                  * PP2expected + 64 * PP3expected + 256 * PP4expected
                  + PPCexpected;

                //each successive partial product is shifted two bits
                //to the left from the previous one, corresponding in
                //integer form to successive factors of four:

                int P = PP0 + 4 * PP1 + 16 * PP2 + 64 * PP3 + 256 *
PP4;


                //add the sign bit as a carry to take care of
                //any "add one" steps in twos complement negation:
                if (PPCval[1].charAt(4) == '1') {
                  P = P + 1;
                }
                if (PPCval[1].charAt(3) == '1') {
                  P = P + 4;
                }
                if (PPCval[1].charAt(2) == '1') {
                  P = P + 16;
                }
                if (PPCval[1].charAt(1) == '1') {
                  P = P + 64;
                }
                if (PPCval[1].charAt(0) == '1') {
                  P = P + 256;
                }


                //Since extra bits have been added to the product
                //computation, do a modulo (2^16) operation to
guarantee
                //16-bit output (just like the chip does):

                P = P % 65536;
                Pcomputed = Pcomputed % 65536;
```

```
                //error checking begins here:

                if (P != Pexpected) {
                  adderrors++;
                  System.out.println("Failed Case: "+index);
                  System.out.println(tests[index]);
                  System.out.println(tests[index + 1]);
                  System.out.println("Addition Error");
                  System.out.println("Expected: "+inttohex(Pexpected));
                  System.out.println("Computed: "+inttohex(Pcomputed));
                  System.out.println("Got: "+inttohex(P));
                }
                if (Pexpected != Pcomputed) {
                  errors++;
                  System.out.println("Failed Case: "+index);
                  System.out.println(tests[index]);
                  System.out.println(tests[index + 1]);
                  System.out.println("Computation Error");
                  System.out.println("Expected: "+inttohex(Pexpected));
                  System.out.println("Computed: "+inttohex(Pcomputed));
                }

                if (PP0 != PP0expected) {
                  pperrors++;
                  System.out.println("Failed Case: "+index);
                  System.out.println(tests[index]);
                  System.out.println(tests[index + 1]);
                  System.out.println("Partial Product Error in PP0");
                  System.out.println("Expected:
"+inttohex(PP0expected));
                  System.out.println("Got: "+inttohex(PP0));
                }

                if (PP1 != PP1expected) {
                  pperrors++;
                  System.out.println("Failed Case: "+index);
                  System.out.println(tests[index]);
                  System.out.println(tests[index + 1]);
                  System.out.println("Partial Product Error in PP1");
                  System.out.println("Expected:
"+inttohex(PP1expected));
                  System.out.println("Got: "+inttohex(PP1));
                }

                if (PP2 != PP2expected) {
                  pperrors++;
                  System.out.println("Failed Case: "+index);
                  System.out.println(tests[index]);
                  System.out.println(tests[index + 1]);
                  System.out.println("Partial Product Error in PP2");
                  System.out.println("Expected:
"+inttohex(PP2expected));
                  System.out.println("Got: "+inttohex(PP2));
                }

                if (PP3 != PP3expected) {
```

```
                  pperrors++;
                  System.out.println("Failed Case: "+index);
                  System.out.println(tests[index]);
                  System.out.println(tests[index + 1]);
                  System.out.println("Partial Product Error in PP3");
                  System.out.println("Expected:
"+inttohex(PP3expected));
                  System.out.println("Got: "+inttohex(PP3));
                }

                if (PP4 != PP4expected) {
                  pperrors++;
                  System.out.println("Failed Case: "+index);
                  System.out.println(tests[index]);
                  System.out.println(tests[index + 1]);
                  System.out.println("Partial Product Error in PP4");
                  System.out.println("Expected:
"+inttohex(PP4expected));
                  System.out.println("Got: "+inttohex(PP4));
                }
            }
            else {
                //System.out.println(tests[index]);
            }
          }
          //Report errors:
          System.out.println("Total Computation Errors: "+errors);
          System.out.println("Partial Product Errors: "+pperrors);
          System.out.println("Addition Errors: "+adderrors);
      }

      catch(IOException toss) {
          System.out.println("Error; file not found");
      }
    }

    //computecarries looks at the bits of input B (like the booth
array) and
    //computes the integer sum of the additions generated by twos
complement
    //negation that gets added to the partial products:
    public static int computecarries(String Bval) {

      int carries = 0;
      if (Bval.charAt(6) == '1') {
          carries = carries + 1;
      }
      if (Bval.charAt(4) == '1') {
          carries = carries + 4;
      }
      if (Bval.charAt(2) == '1') {
          carries = carries + 16;
      }
      if (Bval.charAt(0) == '1') {
          carries = carries + 64;
      }
```

```
      return carries;
    }

    //This method mimics the booth-decoder by accepting a number from
    //zero to seven along with an integer and returning 0*, 1*, -1*, 2*
or
    //-2* that integer as the partial product:
    public static int computePP(int mcand, int mierbits) {

      int pp = 0;

      switch(mierbits) {
      case(0) :
          pp = 0;
          break;
      case(1) :
          pp = mcand;
          break;
      case(2) :
          pp = mcand;
          break;
      case(3) :
          pp = 2 * mcand;
          break;
      case(4) :
          pp = negate(2 * mcand);
          break;
      case(5) :
          pp = negate(mcand);
          break;
      case(6) :
          pp = negate(mcand);
          break;
      case(7) :
          pp = 65535;
          break;
      default :
          System.out.println(mierbits);
          System.out.println("Illegal mier value; assuming pp = 0");
          break;
      }


      return pp;
    }

    //This method accepts an integer, treats it as a sixteen-bit
number, and
    //inverts all its bits.  It does not, however, add one to make the
return
    //value a twos complement number, since the booth array does not do
    //this either.

    public static int negate(int binary) {

      int negated;
      if (binary == 0) {
```

```
            negated = 65535;
        }
        else {
            //twos complement-style negation, but don't add the one
because
            //the booth array makes the Wallace tree do that step
            negated = 65535 - binary;
        }
        return negated;

    }


    //The wallace tree test consists of reading in the partial products
    //as inputs and computing whether the sum and carry outputs sum to
the
    //sum of the partial products correctly:

    public static void waltreetest(String filename) {

      try {
          FileInputStream results =
            new FileInputStream(filename);
          String[] tests = readFileLines(results);

          int errors = 0;

          for (int index = 0; index < tests.length; index++) {

            if (tests[index].startsWith("PP0=")) {

                //irsim puts in carriage returns, so the data
                //for this set of tests comes in on two lines:
                String[] iovectors =  parseCommand(tests[index]);
                String[] iovectors2 = parseCommand(tests[index + 1]);

                String[] PP0val = split(iovectors[0], "=");
                String[] PP1val = split(iovectors[1], "=");
                String[] PP2val = split(iovectors[2], "=");
                String[] PP3val = split(iovectors[3], "=");
                String[] PP4val = split(iovectors[4], "=");
                String[] PPCval = split(iovectors[5], "=");

                String[] Sval = split(iovectors2[0], "=");
                String[] Cval = split(iovectors2[1], "=");

                //Perform sign extension as needed on the
                //partial products just like the wallace tree
                //does:
                int PP0 = bintoint(PP0val[1]);
                if (PP0 > 63) {
                  PP0 = PP0 + 65408;
                }
                int PP1 = bintoint(PP1val[1]);

                if (PP1 > 255) {
                  PP1 = PP1 + 65024;
```

```
        }
        if (PPCval[1].charAt(3) == '1') {
          PP1++;
        }

        int PP2 = bintoint(PP2val[1]);
        if (PP2 > 255) {
          PP2 = PP2 + 65024;
        }
        if (PPCval[1].charAt(2) == '1') {
          PP2++;
        }
        int PP3 = bintoint(PP3val[1]);
        if (PP3 > 255) {
          PP3 = PP3 + 65024;
        }
        if (PPCval[1].charAt(1) == '1') {
          PP3++;
        }

        int PP4 = bintoint(PP4val[1]);
        if (PP4 > 127) {
          PP4 = PP4 + 65280;
        }
        if (PPCval[1].charAt(0) == '1') {
          PP4++;
        }


        int S = bintoint(Sval[1]);
        int C = bintoint(Cval[1]);

        //the least significant bit of the carry is one
        //higher than the LSB of the sum, so shift it and
        //normalize the result to a 14-bit value since that's
        //how wide the wallace tree output is:
        int Scomputed = (S + 2 * C) % 16384;
        int Ssum = PP0 + PP1 + 4 * PP2 + 16 * PP3 + 64 * PP4;
        int Sexpected = Ssum % 16384;

        if (Scomputed != Sexpected) {
          errors++;
          System.out.print("Failed Case: "+index);
          System.out.println(tests[index]);
          System.out.println(tests[index + 1]);
          System.out.print("Expected: ");
          System.out.print(inttohex(Sexpected));
          System.out.print(" Got: ");
          System.out.println(inttohex(Scomputed));
        }

    }
    else {
        //System.out.println(tests[index]);
    }
}
System.out.println("Total Errors: "+errors);
```

```
  }

  catch(IOException toss) {
      System.out.println("Error; file not found");
  }
}

//this method accepts a String as input and converts it to binary,
//treating non-zeros and ones as zeros.  Note that this may mask
//a few types of errors if your output is X's.

public static int bintoint(String binval) {

  //System.out.println(binval);

  int intval = 0;
  int factor = 1;

  for (int index = binval.length() - 1; index >= 0; index--) {
      if (binval.charAt(index) == '1') {
        intval = intval + factor;
      }
      factor = factor * 2;
  }

  return intval;
}


//Accepts a String of hexidecimal characters and outputs
//the decimal represenation of the hex number
public static int hextoint(String hexval) {

  int intval = 0;
  int factor = 1;

  for (int index = hexval.length() - 1; index >= 0; index--) {
      intval = intval + factor * hextoint(hexval.charAt(index));
      factor = factor * 16;
  }
  return intval;
}

//decodes a single hex digit into an integer value
public static int hextoint(char hexval) {

  int intval = 0;
  boolean error = true;

  if (hexval == '0') {
      intval = 0;
      error = false;
  }
  if (hexval == '1') {
      intval = 1;
      error = false;
  }
```

```
if (hexval == '2') {
    intval = 2;
    error = false;
}
if (hexval == '3') {
    intval = 3;
    error = false;
}
if (hexval == '4') {
    intval = 4;
    error = false;
}
if (hexval == '5') {
    intval = 5;
    error = false;
}
if (hexval == '6') {
    intval = 6;
    error = false;
}
if (hexval == '7') {
    intval = 7;
    error = false;
}
if (hexval == '8') {
    intval = 8;
    error = false;
}
if (hexval == '9') {
    intval = 9;
    error = false;
}
if (hexval == 'a' || hexval == 'A') {
    intval = 10;
    error = false;
}
if (hexval == 'b' || hexval == 'B') {
    intval = 11;
    error = false;
}
if (hexval == 'c' || hexval == 'C') {
    intval = 12;
    error = false;
}
if (hexval == 'd' || hexval == 'D') {
    intval = 13;
    error = false;
}
if (hexval == 'e' || hexval == 'E') {
    intval = 14;
    error = false;
}
if (hexval == 'f' || hexval == 'F') {
    intval = 15;
    error = false;
}
```

```
      if (error == true) {
          System.out.println("Error; bad hex character, assuming 0");
      }

      return intval;
    }

    //converts an integer to a String:
    public static String inttohex(int intval) {

      //System.out.println(intval);

      int numdigits = 0;
      int temp = intval;

      do {
          temp = (temp - (temp % 16)) / 16;
          numdigits++;
      } while (temp > 0);

      char[] hexarray = new char[numdigits];
      numdigits = numdigits - 1;

      do {
          int curdigit = intval % 16;
          //System.out.println(curdigit);
          hexarray[numdigits] = inttohexchar(curdigit);

          intval = (intval - curdigit) / 16;
          //System.out.println(intval);
          numdigits = numdigits - 1;
      } while (intval > 0);

      return new String(hexarray);
    }

    //decodes a integer and translates it into a single character.
Note that
    public static char inttohexchar(int intval) {


      char hexval = '0';

      if (intval == 0) {
          hexval = '0';
      }
      if (intval == 1) {
          hexval = '1';
      }
      if (intval == 2) {
          hexval = '2';
      }
      if (intval == 3) {
          hexval = '3';
      }
      if (intval == 4) {
          hexval = '4';
```

```
      }
      if (intval == 5) {
          hexval = '5';
      }
      if (intval == 6) {
          hexval = '6';
      }
      if (intval == 7) {
          hexval = '7';
      }
      if (intval == 8) {
          hexval = '8';
      }
      if (intval == 9) {
          hexval = '9';
      }
      if (intval == 10) {
          hexval = 'a';
      }
      if (intval == 11) {
          hexval = 'b';
      }
      if (intval == 12) {
          hexval = 'c';
      }
      if (intval == 13) {
          hexval = 'd';
      }
      if (intval == 14) {
          hexval = 'e';
      }
      if (intval == 15) {
          hexval = 'f';
      }
      return hexval;
    }

    //split accepts a delimiter and returns a two-string array
containing a
    //split of the string right before the first instance of the
delimiter
    public static String[] split(String input, String delimiter) {

      String[] splitString = new String[2];

      for (int index = 0; index < input.length(); index++) {

          if (input.substring(index).startsWith(delimiter)) {
            splitString[0] = input.substring(0, index);
            splitString[1] = input.substring(index + 1);
            break;
          }
      }

      if (splitString[0] == null) {
          splitString[0] = "Error!";
          splitString[1] = "Error!";
```

```
      }

      return splitString;
    }

    //this method reads an entire file line-by-line and stores it as
    //a single array of Strings, so that the file may be manipulated
    //accordingly.

    public static String[] readFileLines(FileInputStream input)
      throws IOException {

      BufferedReader lineReader = new BufferedReader(
            new InputStreamReader(input));

      //mark beginning of file
      lineReader.mark(80);

      String[] lineList = new String[20];
      int lineNumber = 0;
      while (lineReader.read() != -1) {

          lineReader.reset();
          lineList[lineNumber] = lineReader.readLine();
          lineNumber++;
          if (lineNumber == lineList.length) {
            lineList = expandList(lineList);
          }
          lineReader.mark(80);
      }

      lineList = TrimArray(lineList);

      return lineList;
    }

    //this method reads a single line, where a line is
    //terminated by a carriage return, and parses the line into
    //its constituent words (whitespaceless strings)

    public static String[] parseCommand(String commandLine) {

          String[] commandList = new String[20];
          int listIndex = 0;
          int charNum = 0;
          StringBuffer temp = new StringBuffer(80);

          while (charNum < commandLine.length()) {

            Character testChar =
                new Character(commandLine.charAt(charNum));

            //permit words to be grouped into phrases by wrapping them
in
            //quotation marks

            if (isQuotationMark(commandLine.charAt(charNum)) == true) {
```

```
                    //skip the first quotation mark
                    charNum++;
                    while (isQuotationMark(commandLine.charAt(charNum))
                           == false) {

                          temp.append(commandLine.charAt(charNum));
                          charNum++;
                          if (charNum == commandLine.length()) {
                                break;
                          }
                    }
                    //skip the second quotation mark, unless the end of
                    //the string has been reached
                    if (charNum == commandLine.length()) {
                          System.out.println("Warning; didn't find
matching \"");
                          break;
                    }
                    charNum++;
                    //if the quotation mark is the last character,
                    //it's time to be done!
                    if (charNum == commandLine.length()) {
                          commandList[listIndex] = temp.toString();
                          break;
                    }
              }
              if (testChar.isWhitespace(commandLine.charAt(charNum)) ==
                  false && charNum < commandLine.length()) {

                  temp.append(commandLine.charAt(charNum));
                  charNum++;
              }
              else {
                  //store the word just made and set up
                  //for the next one
                  commandList[listIndex] = temp.toString();
                  listIndex++;
                  if (listIndex == commandList.length) {
                    commandList = expandList(commandList);
                  }
                  charNum++;
                  temp = new StringBuffer(80);
              }
            }
            //There's one more String to append when the loop breaks, so
            //add it here:
            commandList[listIndex] = temp.toString();

        commandList = TrimArray(commandList);
        return commandList;
      }

      //this method reads a single line, where a line is
      //terminated by a carriage return, and parses the line into
      //its constituent words (whitespaceless strings)
```

```
    public static String[] readCommandLine(InputStream input) {

      BufferedReader lineReader = new BufferedReader(
            new InputStreamReader(input));

      String[] commandList = new String[20];

      //read a single line

      try {
          String commandLine = lineReader.readLine();

          int listIndex = 0;
          int charNum = 0;
          StringBuffer temp = new StringBuffer(80);

          while (charNum < commandLine.length()) {

            Character testChar =
                new Character(commandLine.charAt(charNum));

            //permit words to be grouped into phrases by wrapping them
in
            //quotation marks

            if (isQuotationMark(commandLine.charAt(charNum)) == true) {

                  //skip the first quotation mark
                  charNum++;
                  while (isQuotationMark(commandLine.charAt(charNum))
                         == false) {

                        temp.append(commandLine.charAt(charNum));
                        charNum++;
                        if (charNum == commandLine.length()) {
                              break;
                        }
                  }
                  //skip the second quotation mark, unless there isn't
                  //one to be skipped
                  if (charNum == commandLine.length()) {
                        System.out.println("Warning; didn't find
matching \"");
                        break;
                  }
                  charNum++;
                  //if the quotation mark is the last character, it's
                  //time to be done!
                  if (charNum == commandLine.length()) {
                        commandList[listIndex] = temp.toString();
                        break;
                  }
            }

            if (testChar.isWhitespace(commandLine.charAt(charNum))
                   == false) {
```

```
            temp.append(commandLine.charAt(charNum));
            charNum++;
          }
          else {
            //store the word just made and set up
            //for the next one
            commandList[listIndex] = temp.toString();
            listIndex++;
            if (listIndex == commandList.length) {
              commandList = expandList(commandList);
            }
            charNum++;
            temp = new StringBuffer(80);
          }
        }
        //There's one more String to append when the loop breaks, so
        //add it here:
        commandList[listIndex] = temp.toString();
    }

    catch(IOException toss) {
        System.out.println("Error reading command line");
        commandList[0] = "error";
    }

    commandList = TrimArray(commandList);
    return commandList;
  }

  public static String[] expandList(String[] smaller) {

    String[] larger = new String[2 * smaller.length];

    for (int index = 0; index < smaller.length; index++) {

        larger[index] = smaller[index];
    }

    return larger;
  }

  //this method takes an array that's had more space allocated
  //than it needs and gets rid of all the extra space.
  //in doing so, it assumes that all null quantities appear
  //consecutively at the end of the array, i.e. the array of size M
  //has valid data from indices 0-N and nulls from (N+1)-M.

  public static String[] TrimArray(String[] tooBig) {

    int realLength = 0;

    while (tooBig[realLength] != null) {

        realLength++;
    }

    String notTooBig[] = new String[realLength];
```

```
      for (int index = 0; index < realLength; index++) {

          notTooBig[index] = tooBig[index];
      }

      return notTooBig;
    }

    public static boolean isQuotationMark(char testChar) {

      return (testChar == '\"' || testChar == '\223' || testChar ==
'\224');
    }
}
```