Senior Capstone Project Proposal

Reconfigurable FPGA Implementation Of Digital Communication System

Project Members
Steve Koziol
Josh Romans

Project Advisor
Dr T.L. Stewart

Bradley University
Department of Electrical & Computer Engineering
EE 451 – Senior Laboratory
December 8, 2005

Table of Contents

Table of Images

Project Summary

This project is an attempt to develop various reconfigurable digital communication systems for use on a FPGA programmed in VHDL. The primary goal of this project is to determine whether or not the communication system's transmitter and receiver can be contained on a single FPGA. If not, the goal becomes the identification of what advances are necessary for the FPGA to have this kind of reconfigurable functionality. The initial communication system developed will be an amplitude modulated (AM) system. Based on the results of the AM system, a frequency modulated (FM) system is tentatively scheduled to follow. The Altera UP2 development board containing the FLEX10K series FPGA will be used to implement the communication system design.

Functional Description and System Block Diagram

Background

Improvements in the performance and density of FPGAs over the past decade have caused firmware designers to reexamine the role of FPGAs in their end products, specifically relating to the use of reconfigurable FPGAs. Reconfigurable logic design involves manipulation of the logic within the FPGA at run-time. This allows the hardware to change in response to the demands placed upon the system while it is running. Benefits of using a FPGA hardware design over more traditional application specific integrated circuits (ASIC) include greater functionality with a simpler hardware design, lower system cost, and reduced time-to-market. Benefits of using a reconfigurable FPGA in a communication system include reducing the size of the product by including the transmitter and receiver on a single chip, which in turn reduces cost. [1]

.

Objectives

The objective of this project is to develop various digital communication systems to be implemented on a FPGA programmed in VHDL. The communication system's transmitter and receiver will be contained on a single FPGA, making it a reconfigurable system. The initial communication system developed will be an AM system, with a FM communication system to follow on the same FPGA. The Altera UP2 development board contains the FPGA that will be used to implement the communication system design.

Description of Inputs and Outputs

The inputs and outputs are one in the same for the Reconfigurable FPGA Implementation of Digital Communication System. This means that the receiver and transmitter signals will be read into the reconfigurable FPGA. The system will switch functionality in run-time, so receiver and transmitter signals are treated as both inputs and outputs. The inputs and outputs initially tested will be voice signals, but other communication signals can be implemented using this system. A possible use of this system would be a walky-talky device contained on a single IC.

Overall System Flow Diagram

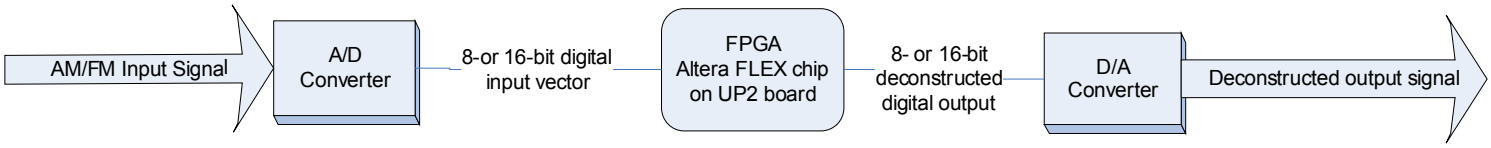

Figure 1 High-level RFIDCS complete system view

Description of Input and Output Connections among Subsystems

For the initial system being developed, an AM signal will act as the overall system input. After undergoing an A/D conversion on an external A/D chip, the digital input signal will

be read by the FPGA in its receiver configuration. After processing and deconstruction, the signal is then outputted by the FPGA configured as a transmitter to the D/A converter. The output of the D/A converter is the overall system output; a deconstructed equivalent of the AM input signal.

FPGA

Figure 2-1 focuses on the flow of data inside the FPGA. A control logic module will be necessary to switch the FPGA between its receiver and transmitter modes. The control logic will also be instrumental during the multiplication process because of the use of the Booth multiplication algorithm, which will be explained later. For this reason, it may be beneficial to have two separately dedicated control modules, one for mode control and one for multiplication data arrangement control.

In receiver mode, the FPGA will read and store the input data from the A/D converter. This data is then multiplied by an internally generated carrier frequency signal. The ideal method for producing this carrier frequency signal (e.g. cosine wave) within the FPGA has yet to be determined and is the subject of upcoming laboratory research. The control logic then switches the FPGA to transmitter mode based on a signal generated signifying the end of the current multiplication process. The product is then transmitted out of the FPGA to the external D/A converter.

FPGA Data Flow

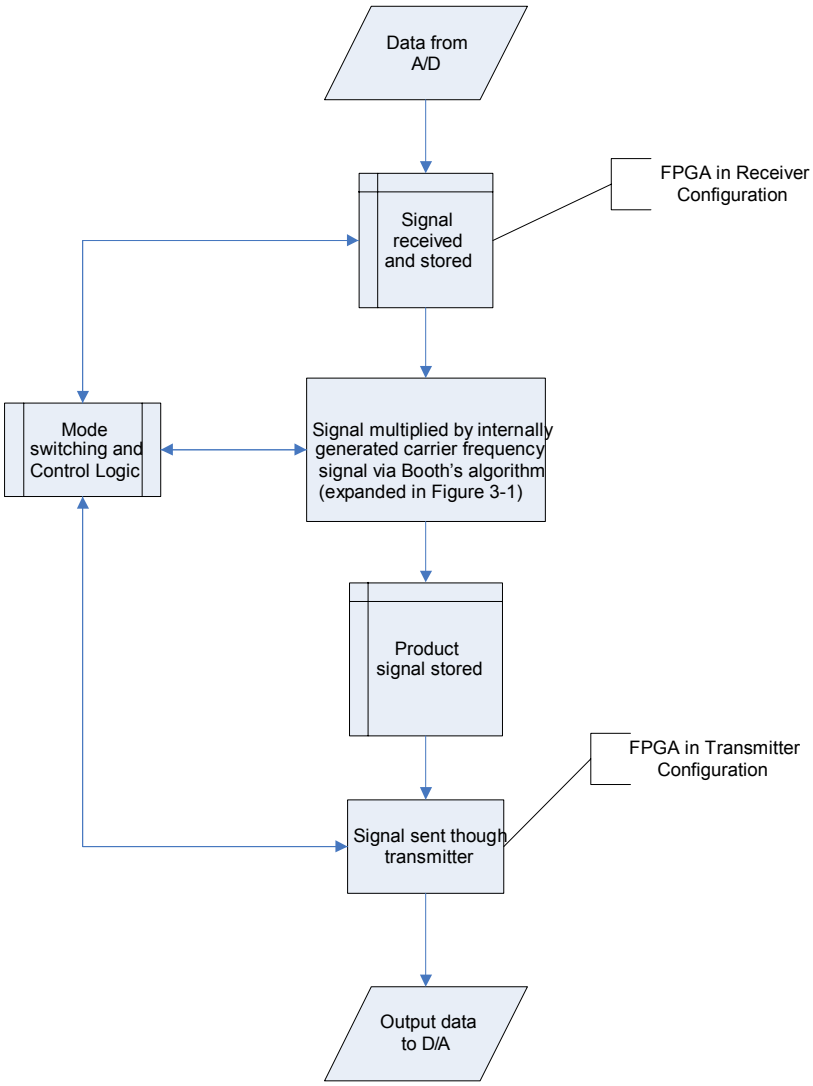

Figure 2 Showing the data flow and necessary code blocks internal to the FPGA

Booth Multiplication Flowchart

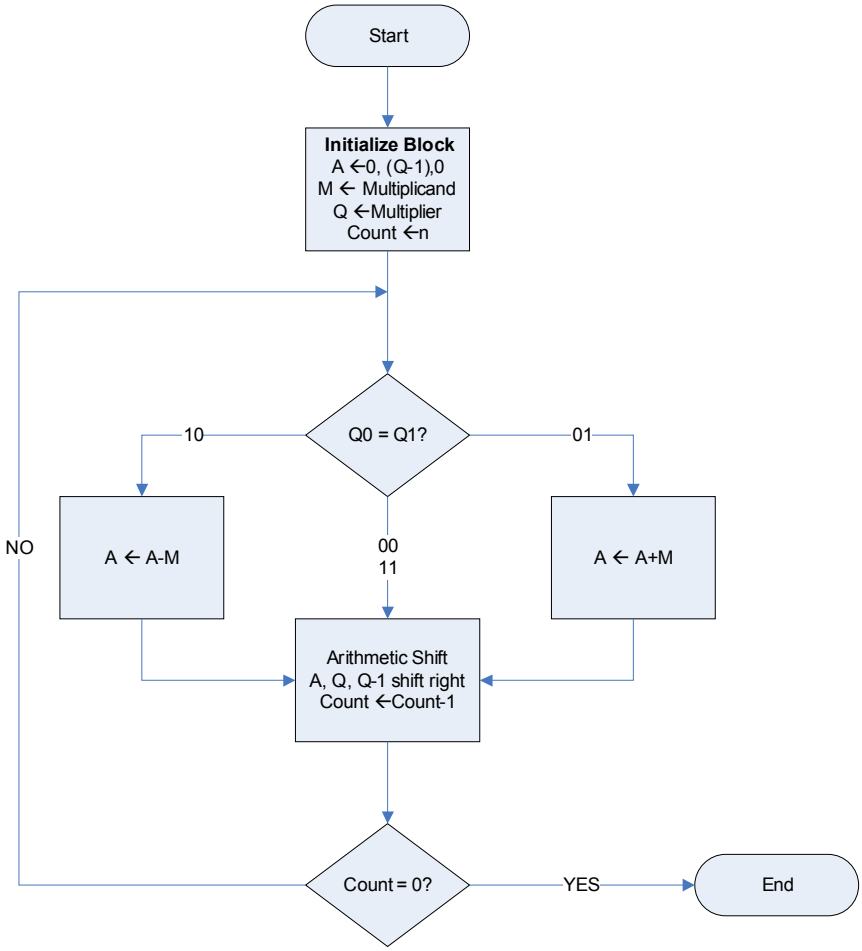

Figure 3 Data flow of the Booth multiplication algorithm

Figure 3-1 expands upon the Booth algorithm multiplication block show in Figure 2-1. The Booth algorithm is a fast and efficient method of two signed two's compliment variables. It is faster than conventional multiplication algorithms because it utilizes the fact that in multiplication, not all bits of the partial product need to be propagated to the next stage of summation. By eliminating the unnecessary bits, the size of the partial products generated is reduced significantly, allowing the next summation to be reached sooner and saving valuable hardware space. A partial schematic of a parallel two's compliment Booth multiplication is shown in Figure 4-1. Variable 'a' is the multiplier and 'b' is the multiplicand. Both are 32-bit two's compliment numbers. The add32csa block performs the addition of the variables 'a' and 'b' and a partial schematic of this block is shown in Figure 4-2. 'a' and 'b' are summed by the fadd block and then transmitted to the next cascaded add32csa segment.

Booth Multiplication Schematics

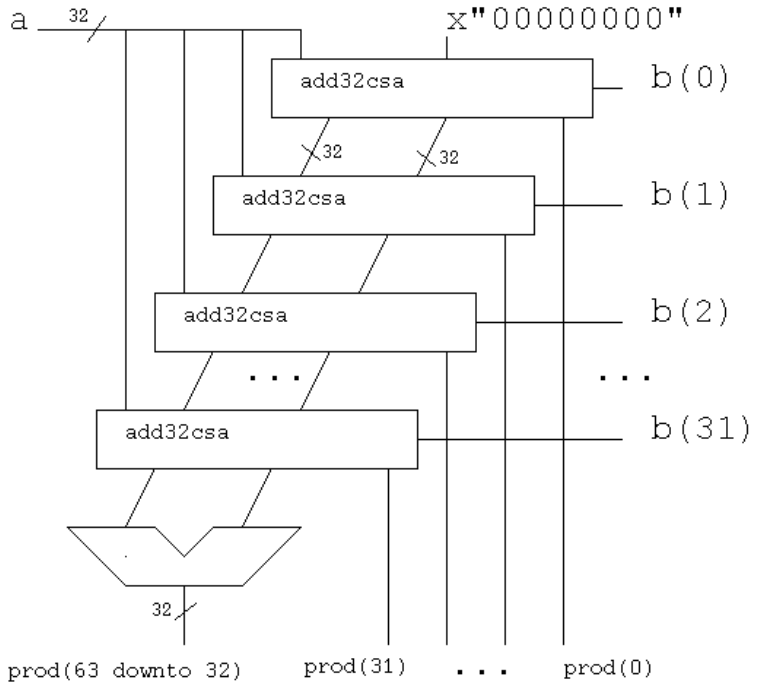

Figure 4 Booth multiplication partial schematic

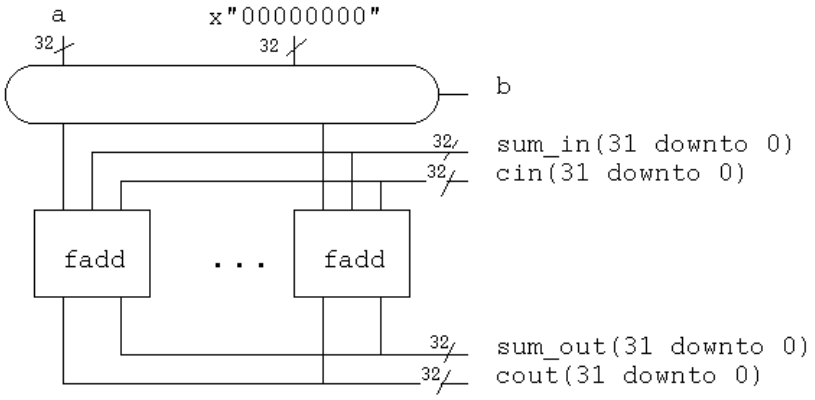

Figure 5 Partial schematic of the add32csa block in shown in figure 4-1

Simulation Results

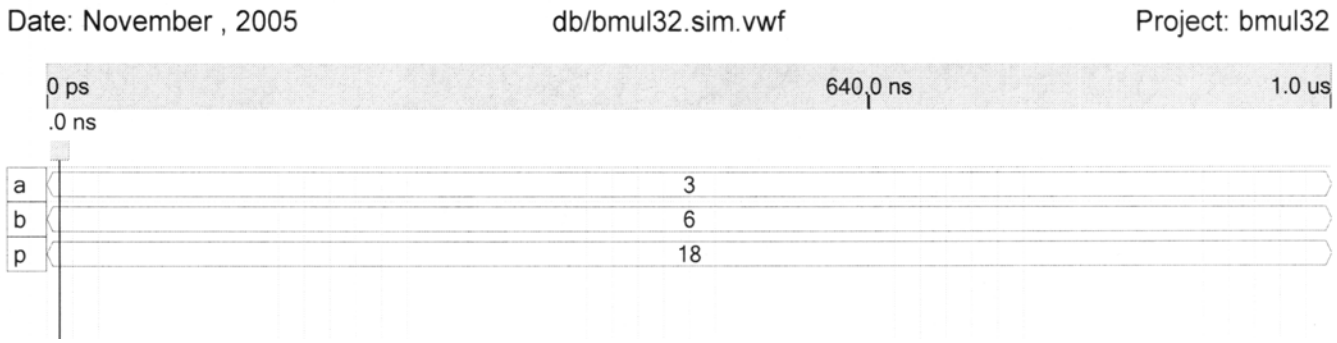

Figure 6 Initial multiplier results

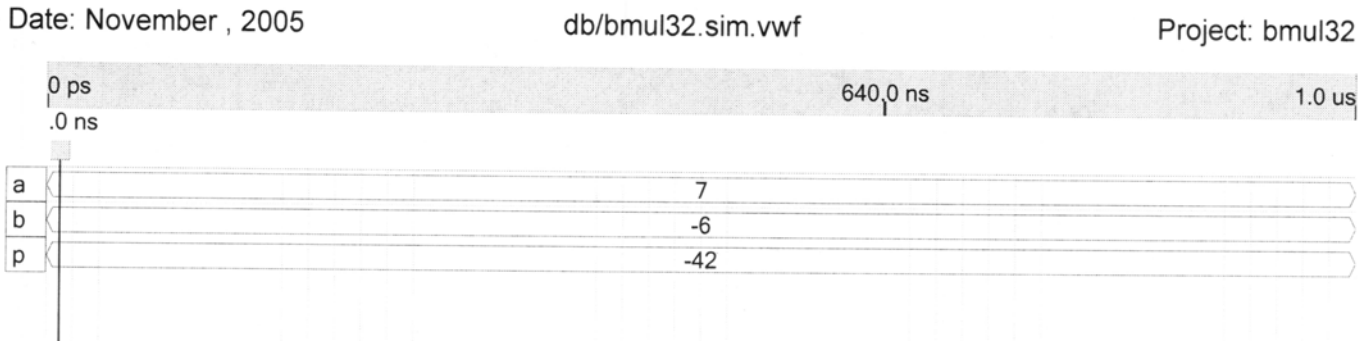

Figure 7 Initial multiplier results showing signed multiplication

Discussion of Results

Figures six and seven show initial functional simulation results performed on the multiplier. Variables 'a' and 'b' are the 32-bit multiplier and multiplicands respectively. In the completed system, 'a' would represent the input signal to the system and 'b' would be the internally generated carrier frequency signal. 'p' represents the modulated 64-bit product.

Relevant Patents

20050066156 Telecommunication device with software components

20050261797 Programmable radio transceiver

20050227627 Programmable radio transceiver

20020156998 Virtual computer of plural FPG's successively reconfigured in response to a succession of inputs

20050235173 Reconfigurable integrated circuit

20050235070 Systems and methods for reconfigurable computing

20040242261 Software-defined radio

Schedule

Week	Task	Team Members
Winter Break	Research reconfigurable communication systems Research VHDL adders and filters	Josh Steve
Week 1 Jan 23 – 27	Verify operation of multiplier with external signals	Both
Week 2 – 4 Jan 30 – Feb 17	Design and test reciever for AM system Design and test rectifier and filter for modulation	Josh Steve
Week 5 Feb 20 – Feb 24	Integrate Multiplier into reciever and verify results	Both
Week 6 – 9 Feb 27 – Mar 24	Design and test transmitter for AM system Design and test de-modulation smoothing filter	Steve Josh
Week 10 – 11 Mar 27 – Apr 7	Integrate reciever and transmitter	Both
Week 12 Apr 10 – Apr 14	Verify overall system functionality	Both
Week 13 – 14 Apr 17 – Apr 28	Prepare for final report and presentation	Both
Week 15 May 2	Presentation	Both

Equipment List

- **Altera UP2 Development Board using FLEX10K: EPF10K70RC240-4 FPGA**
- **LM ADC080X 8-bit A/D Converter**
- **LM DAC080X 8-bit D/A Converter**

References

[1] A Single-Chip Supervised Partial Self-Reconfigurable Architecture for Software Defined Radio – *IEEE Computer Society http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/ipdps/& toc=comp/proceedings/ipdps/2003/1926/00/1926toc.xml& DOI=10.1109/IPDPS.2003.1213354*

[2] University of Maryland, Department of Computer Science and Electrical Engineering, http://www.csee.umbc.edu/help/VHDL/samples/samples.shtml#mul32

[3] Khurram Muhammad, Robert Bogdan Staszewski, and Dirk Leipold, Texas Instruments, Digital RF Processing: Toward Low-Cost Reconfigurable Radios (IEEE Communication Magazine, August 2005)[1]

Appendix 1 – Reference Abstracts

1 Digital RF Processing: Toward Low-Cost Reconfigurable Radios

RF circuits for multi-gigahertz frequencies have recently migrated to state-of-the-art low cost digital CMOS processes. This article visits fundamental techniques recently developed that migrate RF and analog design complexity to the digital domain for a wireless RF transceiver. All digital phase locked loop and direct RF sampling techniques allow great flexibility in reconfigurable radio design. Digital signal processing concepts are used to help relieve analog design complexity, allowing one to reduce cost and power consumption in a reconfigurable design environment. Software layers are defined to enable these architectures to develop an efficient software-defined radio. The ideas presented have been used to develop two generations of commercial digital RF processors: a single-chip Bluetooth radio and a single-chip GSM radio.

Appendix – 2 Multiplier Code

```
--Steve Koziol and Josh Romans
--Bradley University
--Department of Electrical and Computer Engineering
--Version 1.1, last update 11/23/2005

-- bmul32   full combinatorial 32 X 32 = 64 bit two's complement
multiplier
-- Booth two's complement multiplication using badd4 component

library IEEE;
use IEEE.std_logic_1164.all;

entity bmul32 is   -- 32-bit by 32-bit two's complement multiplier
  port (a : in  std_logic_vector(31 downto 0);   -- multiplier
        b : in  std_logic_vector(31 downto 0);   -- multiplicand
        p : out std_logic_vector(63 downto 0));  -- product
end entity bmul32;

architecture circuits of bmul32 is
  signal zer : std_logic_vector(31 downto 0) := x"00000000";     --
zeros
  signal mul0: std_logic_vector(2 downto 0);
  subtype word is std_logic_vector(31 downto 0);
  type ary is array(0 to 15) of word;
  signal s : ary;                        -- temp sums
begin  -- circuits of bmul32
  mul0 <= a(1 downto 0) & '0';
  a0:  entity WORK.badd32 port map(
              mul0,            b, zer,   s( 0), p( 1 downto  0));
  a1:  entity WORK.badd32 port map(
              a(3  downto  1), b, s( 0), s( 1), p( 3 downto  2));
  a2:  entity WORK.badd32 port map(
              a(5  downto  3), b, s( 1), s( 2), p( 5 downto  4));
  a3:  entity WORK.badd32 port map(
              a(7  downto  5), b, s( 2), s( 3), p( 7 downto  6));
  a4:  entity WORK.badd32 port map(
              a(9  downto  7), b, s( 3), s( 4), p( 9 downto  8));
  a5:  entity WORK.badd32 port map(
              a(11 downto  9), b, s( 4), s( 5), p(11 downto 10));
  a6:  entity WORK.badd32 port map(
              a(13 downto 11), b, s( 5), s( 6), p(13 downto 12));
  a7:  entity WORK.badd32 port map(
              a(15 downto 13), b, s( 6), s( 7), p(15 downto 14));
  a8:  entity WORK.badd32 port map(
              a(17 downto 15), b, s( 7), s( 8), p(17 downto 16));
  a9:  entity WORK.badd32 port map(
```

```
              a(19 downto 17), b, s( 8), s( 9), p(19 downto 18));
  a10: entity WORK.badd32 port map(
              a(21 downto 19), b, s( 9), s(10), p(21 downto 20));
  a11: entity WORK.badd32 port map(
              a(23 downto 21), b, s(10), s(11), p(23 downto 22));
  a12: entity WORK.badd32 port map(
              a(25 downto 23), b, s(11), s(12), p(25 downto 24));
  a13: entity WORK.badd32 port map(
              a(27 downto 25), b, s(12), s(13), p(27 downto 26));
  a14: entity WORK.badd32 port map(
              a(29 downto 27), b, s(13), s(14), p(29 downto 28));
  a15: entity WORK.badd32 port map(
              a(31 downto 29), b, s(14), p(63 downto 32) , p(31 downto
30));
end architecture circuits;  -- of bmul32

--Bradley University
--Department of Electrical and Computer Engineering
--Steve Koziol and Josh Romans
--Version 1.1, last update 11/23/2005

--badd23 is the heart of the multiplier algorithm

library IEEE;
use IEEE.std_logic_1164.all;

entity badd32 is
  port (a       : in  std_logic_vector(2 downto 0);  -- Booth
multiplier
        b       : in  std_logic_vector(31 downto 0); -- multiplicand
        sum_in  : in  std_logic_vector(31 downto 0); -- sum input
        sum_out : out std_logic_vector(31 downto 0); -- sum output
        prod    : out std_logic_vector(1 downto 0)); -- 2 bits of
product
end entity badd32;

architecture circuits of badd32 is
  -- Note: Most of the multiply algorithm is performed in here.
  -- multiplier action
  --     a             bb
  -- i+1 i i-1         multiplier,  shift partial result two places
each stage
  --  0  0  0   0    pass along
  --  0  0  1   +b   add
  --  0  1  0   +b   add
  --  0  1  1   +2b  shift add
  --  1  0  0   -2b  shift subtract
  --  1  0  1   -b   subtract
  --  1  1  0   -b   subtract
  --  1  1  1   0    pass along
  subtype word is std_logic_vector(31 downto 0);
  signal bb        : word;
  signal psum      : word;
  signal b_bar     : word;
```

```
  signal two_b     : word;
  signal two_b_bar : word;
  signal cout      : std_logic;
  signal cin       : std_logic;
  signal topbit    : std_logic;
  signal topout    : std_logic;
  signal nc1       : std_logic;
begin  -- circuits of badd32
  two_b <= b(30 downto 0) & '0';
  b_bar <= not b;
  two_b_bar <= b_bar(30 downto 0) & '0';
  bb <= b when a="001" or a="010"           -- 5-input mux
        else two_b when a="011"
        else two_b_bar when a="100"         -- cin=1
        else b_bar when a="101" or a="110"  -- cin=1
        else x"00000000";
  cin <= '1' when a="100" or a="101" or a="110"
         else '0';
  topbit <= b(31) when a="001" or a="010" or a="011"
            else b_bar(31) when a="100" or a="101" or a="110"
            else '0';

  a1: entity WORK.add32 port map(sum_in, bb, cin, psum, cout);
  a2: entity WORK.fadd port map(sum_in(31), topbit, cout, topout, nc1);

  sum_out(29 downto 0) <= psum(31 downto 2);
  sum_out(31) <= topout;
  sum_out(30) <= topout;
  prod <= psum(1 downto 0);
end architecture circuits;  -- of badd32

--Bradley University
--Department of Electrical and Computer Engineering
--Steve Koziol and Josh Romans
--Version 1.1, last update 11/23/2005

--add32 is the 32 bit adder with carry

library IEEE;
use IEEE.std_logic_1164.all;
entity add32 is             -- simple 32 bit ripple carry adder
  port(a    : in  std_logic_vector(31 downto 0);
       b    : in  std_logic_vector(31 downto 0);
       cin  : in  std_logic;
       sum  : out std_logic_vector(31 downto 0);
       cout : out std_logic);
end entity add32;

architecture circuits of add32 is
  signal c : std_logic_vector(0 to 30); -- internal carry signals
  component fadd   -- duplicates entity port
  port(a    : in  std_logic;
       b    : in  std_logic;
       cin  : in  std_logic;
```

```
       s    : out std_logic;
       cout : out std_logic);
  end component fadd ;
begin  -- circuits of add32
  a0:            fadd port map(a(0), b(0), cin, sum(0), c(0));
  stage: for I in 1 to 30 generate
            as: fadd port map(a(I), b(I), c(I-1) , sum(I), c(I));
        end generate stage;
  a31:           fadd port map(a(31), b(31), c(30) , sum(31), cout);
end architecture circuits;  -- of add32

--Bradley University
--Department of Electrical and Computer Engineering
--Steve Koziol and Josh Romans
--Version 1.1, last update 11/23/2005

-- bmul32.vhdl parallel multiply 32 bit x 32 bit two's complement
-- the main components are bmul32, special Booth 32 x 32 -> 16 bit
multiplier
-- badd32  32 bit specialized  adder for Booth multiplier
-- needs add32 and fadd components in WORK

library IEEE;
use IEEE.std_logic_1164.all;

entity fadd is               -- full adder stage, interface
  port(a    : in  std_logic;
       b    : in  std_logic;
       cin  : in  std_logic;
       s    : out std_logic;
       cout : out std_logic);
end entity fadd;

architecture circuits of fadd is  -- full adder stage, body
begin  -- circuits of fadd
  s <= a xor b xor cin after 1 ns;
  cout <= (a and b) or (a and cin) or (b and cin) after 1 ns;
end architecture circuits; -- of fadd
```