A Binary Adder/Subtractor

Final Project Report
December 9, 1999
E157

Christopher Kalima and Joshua Kihm

Abstract:

A simple Four Function calculator is actually a fairly complex hardware design. Using two Xilinx Spartan XCS10PC84 FPGA's, this project is able to use familiar keypad inputs to calculate the sum or difference of two integers. The user can type in an input on a keyboard in familiar decimal form, press the control keys like a standard calculator, and then see the result of the operation displayed in hexadecimal on a eight number hexidecimal display. The design consists of control of the keyboard input, arithmetic logic, control and number storage units, the interface between the FPGA's, and output logic.

Introduction

A common four-function calculator is actually a fairly complex piece of hardware to implement. For a seven number display, one must find a way to perform arithmetic functions on numbers as large as ten to the seventh minus one (9,999,999) as small as ten to the negative seventh (.0000001). Although speed is not particularly vital for such a specialized system, chip area and systems resources are vitally important. As this project progressed, difficulty in implementing various system components and fitting our design onto the Xilinx Spartan XCS10PC84 field programmable gate array (FPGA) forced us to simplify our design. What started as a four function calculator using IEEE 754 floating point numbers ended up as a simpler binary integer adder/subtractor. We were, however able to at least implement each sub system in software and completed enough of the design to show proof of concept of our original design.

Schematics

The design of the circuit which we used is fairly simple. The first FPGA evaluation board handles the input and the arithmetic logic. It is hooked up to the key pad and to the other FPGA EVB. The keypad is a standard 4x4 matrix keypad as was used in lab 4. The keypad is set up so that the rows are tied high through 1kΩ resistors and are monitored while the columns are powered such that three are high and one is low at any given time. When a key is pressed, the row pin will be shorted to ground when the appropriate column pin is low. By deciphering what row and column are low, the logic can determine what key is being pressed.

The interface between the boards consists of seven data wires. The first five are used as a five bit wide serial data interface which sends data to the second FPGA EVB. One pin sends a signal which synchronizes the input and output of the two boards so that data on both ends matches. The last pin sends as error signal if there is overflow or another arithmetic error.

The second FPGA controls the output of the system. Its pins are taken up be the interface with the first FPGA and with pins used to display data on the seven segment displays. By using multiplexed output, that is switching the data back and forth between two displays and only powering the appropriate display, we cut down the number of necessary pins in half. Two PNP transistors are powered through 330Ω resistors to the base. They are powered one at a time. The collectors are connected to 5v and the emitters to the power pins of the seven segment displays. In this manner, only one of the two numbers on the display is powered at a given time. The seven segment displays use a common anode set up so each LED or number segment is lit when is pulled low through a 330Ω resistor. Figure one shows the setup of our system.

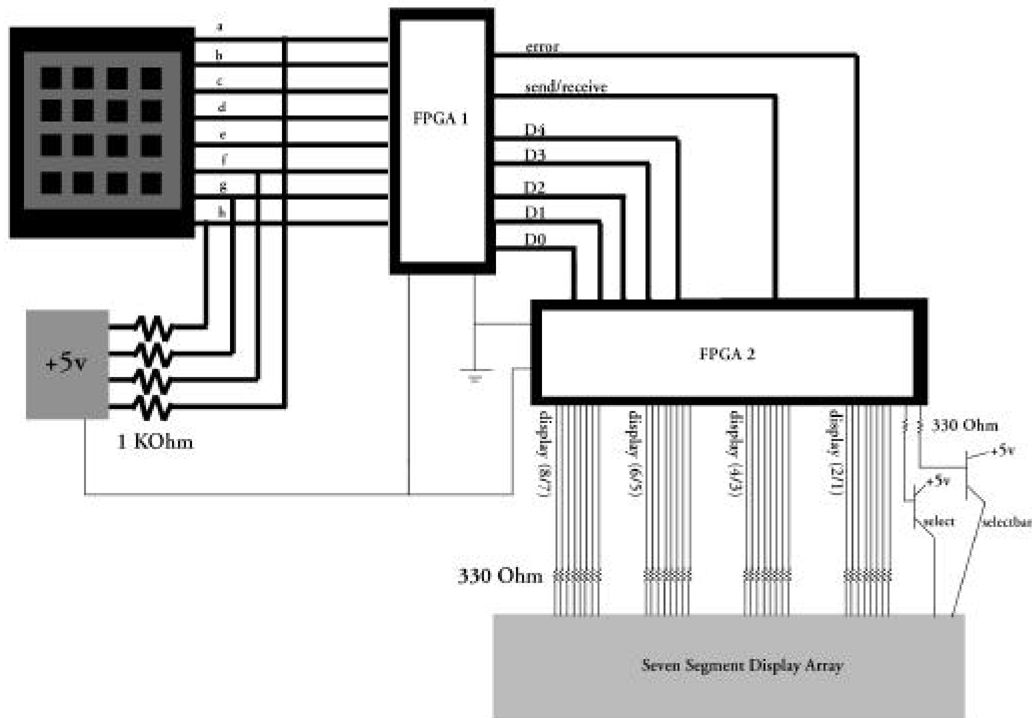

Figure 1:system schematic

FPGA Design

The majority of this project was spent in designing and trouble shooting the logic on our two FPGA's. The system is divided such that the first FPGA EVB is used to interpret the input and perform the arithmetic operation. The second FPGA EVB is used to convert the data from binary to the data used by the actual seven segment displays. Each FPGA also contains the appropriate logic for interfacing with each other.

The number foramt which we used consists of twenty five bits. The top bit is a sign bit and the bottom twenty five bits are the significand. This format was chosen because of its simplicity for multiplication and decoding. There is no shifting or exponent bits so the first bit always corresponds to one, the second to two, the third to four and so on.

The top level layout of the first FPGA can be seen in figure two. Basically, it controls the three registers and the inputs to the adder logic so that the correct data is displayed at the correct time. The first register (Rslt) is used to store the data from the adder and feeds it back into the adder. The next register (disp) also stores data from the adder and it feeds the output logic. The reason why two registers are needed is because when a function or equals key is pressed, the data in the result register clears but the display register keeps the data. The display register stores the data so that the user can still see the result. The result register is cleared so that on the next number press, a new number is started rather than appending it to the end of the old number. The shift left logic is used to compute the number in the register times ten by adding eight times the number plus two times the number. This is done when ever a number is pressed so that the user interface is in a decimal format. For example, if the user pressed two followed by one, the number would be interpreted as twenty-one. The final register (arg) stores the first argument of the main operation being performed. For example if the user input "45 + 90," the argument register would hold the data "45" while the user input the second number.

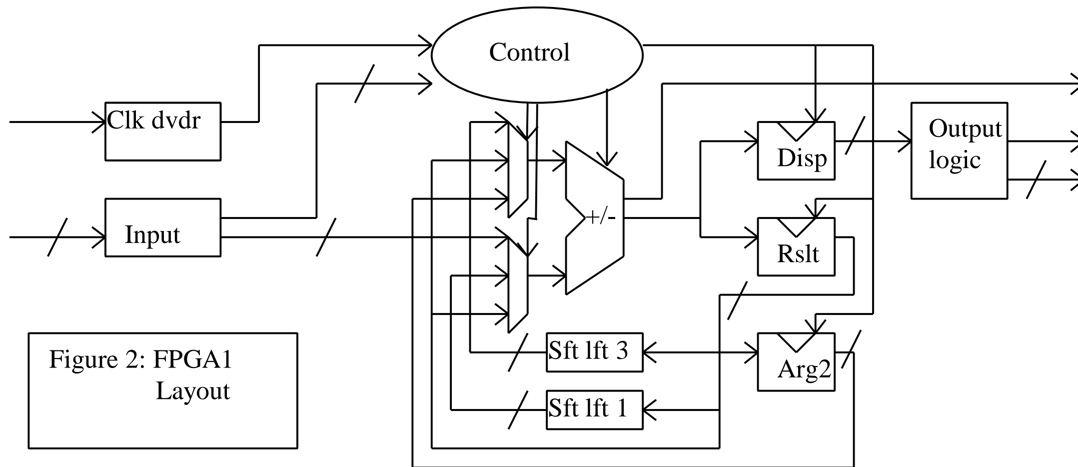

Figure 2: FPGA1 Layout

The logic which controls the input is one of the more complicated subsystems of this design. Fortunately, it was already completed as part of lab 4. The only change that had to be made was to change the decoder module and to remove the end so that it out put a number and a function code rather than signals for a seven-segment display. The basic layout can be seen in figure three. The clock counter slows the clock frequency so that the debouncer is looking at the signal in terms of milliseconds rather than microseconds. The keypad driver outputs to the keypad a one low signal which is used to determine what column a pressed key is in. The Three-Cycle delay block simply delays the data from the keypad driver so that it arrives at the same time as the data from the debouncer. The mux separates out the data which is debouncing form the three stable signals coming from the keypad. The debouncer debouces that one bit. Finally the decoder takes the driver and debouncer data together to determine what key was pressed.

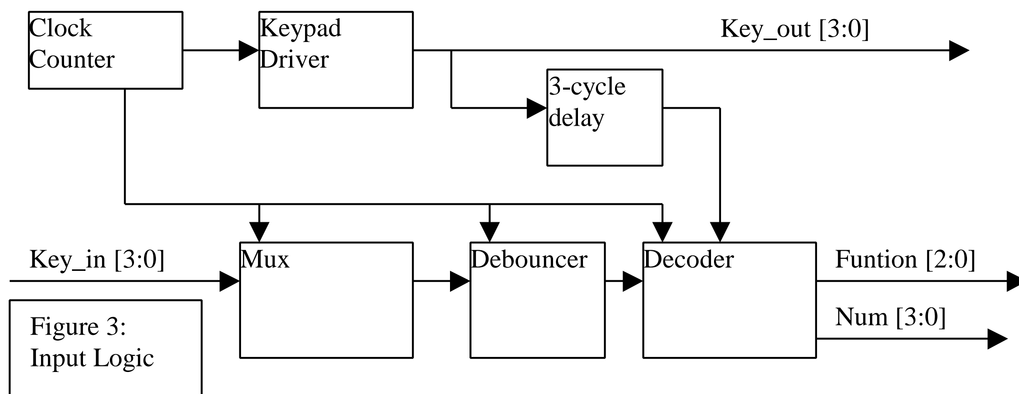

Figure 3:
Input Logic

The Adder logic is based on a finite state machine as shown in figure four. Basically it is just a twos compliment adder set up so that the data which comes out is positive and that the sign bit of the result and the sign bit of the result is adjusted accordingly. The add step simply adds the inputs. The invert state works by subtracting the data from the number zero. The sign of the answer is changed here too. The send state writes an output register so that the data is steady. There is also overflow detection in this software. Because the data to the adder is always positive, it simply states overflow if the sign bit is high on the ALU output and the operation was positive. The sign bit of the output is adjusted also if the first argument is negative. If this is the case, the result has to be arithmetically inverted.

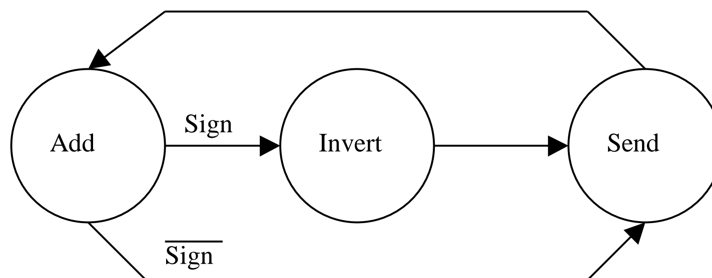

Figure Four: Finite State Machine for adder

The output logic is fairly simple. It is a six-state finite state machine. In the first state, the "send" command is high and the data wires are low. This state is used to synchronize the states of the input and output. The data is then sent five bits at a time with bits zero through four being sent first. Following that, the data is sent five bits at a time until the end is reached. The cycle then repeats.

The finite state machine of the control logic can be seen in figure five. The logic basically waits for a key to be pressed then goes through a few states according to the function code from the input decoder. If a number is pressed, the data in the result register is multiplied by ten then added to the number which was pressed. If the number in the result register was negative, then the number from the decoder is subtracted. If a function key is pressed, plus or minus, the operation code is stored, the number in the result register is written to the argument register, and the result register is cleared. This is where the display register is needed. If it were not there, then the old number would disappear when the function key was pressed or the next number pressed would simply be appended onto the end of the first. A clear command makes the result and display registers go to zero. The argument register is not cleared. This is so that if a mistake is made by the user, they can clear the number which they are inputting without losing the first argument. The plus minus key inverts the sign bit of the result and display registers. The equals key, performs the operation which is stored in the opcode register, writes the data to the argument register and then clears the result register.

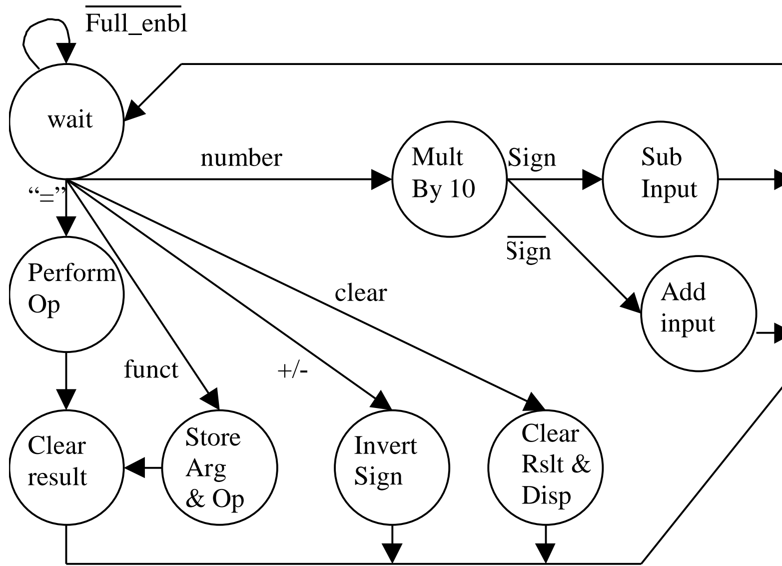

Figure Five: Finite state Machine for Control logic

The second FPGA handles the output of the system. It receives the 5-bit wide serial input from the other FPGA and then drives the seven segment displays. The input block of this FPGA is basically a shift register with a FSM control. The states simply count the clock cycles since the send signal from the output was high. Once that count reaches five, the data is done transferring and so the data in the shift register is written to the out put register. The out put register is then read by the output encoder (called out_reg). This module uses a multiplexor to read the input bits four at a time. It runs those four bits through a decoder which convert a four bit number to the appropriate signal for the seven segment display. On the other side of the decoder, eight 8-bit registers store the data from the decoder. The registers are written when the corresponding eight bits are being read. For example, if the four least significant bits are being read, then the first register is written with that data. The registers are eight bits so that the decimal point can be used, even though are system does not use it. The display is multiplexed using PNP transistors as in lab 3. When the display clock is high, the first, third fifth and seventh numbers on the display are driven and the even ones are driven when the clock is low.

Results

In the end, this system can add and subtract integers which are input through the key pad. Unfortunately, we were unable to get the system to work in floating point as originally planned. Also, we were unable to integrate the multiplication and division due to space restraints. We did prove that our original design would work if we had a slightly larger FPGA or a little more time to squeeze the needed logic on our chips.

Originally, we planned to implement our design in IEEE 754 floating point. We were successful in implementing all of the needed code to perform the operations. However, when we tried to program our chip, the system required far more CLBs then were available. Our first attempts at optimization were basically in cleaning up our code so that we implied less hardware, mainly ALUs.. This was far from enough. The major progress we did make in reducing the size of our logic was that we changed the adder from a single cycle design to a multicycle design. The modules which performed variable shift were very large as they implied often twenty three way, twenty-three bit multiplexors. These were replaced in the multicycle approach by single step shifts and control logic which could repeatedly perform such steps. This design uses only one ALU to perform all of the various addition and subtraction needed in the floating point process. This reduced the adder itself to about sixty percent usage of the CLB's or about 120 CLB. This was a vast improvement over our early single cycle designs which were in the range of two hundred percent. The problem was that the Xilinx software would not right the eight bit exponent from the twenty-six bit ALU. Despite trying a large number of ways around this problem we could not find a way to get it to work. Also, the multiplier hardware would not perform multiplication on larger numbers after being recorded. Smaller numbers would come out correctly but large ones would cause problems. Due to the impending deadline and the troubles we were having with the arithmetic logic, we decided to go with a simpler solution so that we could meet our deadline. Looking back, it would have been a good idea to go to binary coded decimal due to the ease of decoding. However, our twenty five bit format required us to change only a small amount of hardware, mainly in the adder to get the system working. With the deadline looming, we were unable to fit the adder, input software, and the multiplier on one chip. The design, with all of this hardware only required ten more CLB than we had available. Given just another day, we probably could have further optimized the code to the point here it would have fit. The easiest way to do this would have been to use the same ALU for the adder and the multiplier which probably would be enough to fit the multiplier on the chip (a twenty five bit ALU usually takes up about 12 CLBs in implementation with the foundation software). So in the end, we ended up with the adder, control logic and registers, the interface with the second ALU, and the input system on the first chip. This implemented using 113 CLBs of the available 196 or about 57%.

The main reason we decided to use two chips was that we could fit the decimal decoder on the second chip. Since the system uses a simple binary number format, the decoder does not require any complicated hard ware for performing arithmetic. We were able to design the decoder which worked perfectly in simulation and only used sixty eight percent of the CLBs with the input system and output registers. However, when we actually tried the system, the outputs were only zero and error. This was probably due to a timing error. We implemented the final system with the second FPGA with only the interface system and the output registers so that the result would come out in hexadecimal. This system works perfectly so the problem is in the actual decimal decoder. Again, with the impending deadline, we were unable to fully track down this problem, but it is probably only a relatively minor mistake.

One final note on our implementation is that the system only works in the one megahertz mode on the EVBs. The output becomes blurry when we use the two mega hertz mode probably because the data does not have enough time to stabilize and fully stabilize the seven-segment displays before it changes to the other number.

Over all , the designs worked, at least in simulation, it was fitting them on the chip and worrying about timing in the real world that was the greatest challenge. This is probably a very good lesson to learn as this is exactly the sorts of things practicing engineers worry about. The system is at least minimally functional and contains enough functions to prove our conceptual design. With a day or two more time, multiplication could have been squeezed in and decimal decoding could have been worked out. Practicing engineers also have to work with deadlines though.

References

[1]Patterson, David and Hennesy, John, *Computer Organization and Design*, Palo Alto, CA: Morgan Kaufman, 1997.

Parts List

All parts used in this project are readily available in the Micro Systems lab or the engineering stock room.

Appendices

Appendix A: Binary Multiplication and Floating Point Operations

Appendix B: Verilog Code used in the final design

Appendix C: Verilog Code not used in the final design

Appendix A: Hardware not included in Final System

Through the course development of this design, several radically different approaches were taken to the arithmetic logic. Also, the conversion of numbers to decimal was tried. Unfortunately, size and timing constraints forced the omission of these systems from the final system. This appendix will address multiplication, division, decimal conversion and floating point addition logic which was omitted from the final design.

Binary Multiplier

When implementing our original design, the multiplier inferred in Verilog was optimized for operational speed, but took up too more space than any other component. Because of this, there was a need to design our own binary multiplier, which operated at a much, much slower rate but took up considerably less area on the chip.

The multiplication algorithm used is the same as the one that is presented in *Patterson and Hennesy*, which was chosen due to its relatively small area and simplicity of components. It consists of two registers, one being 24-bits in length (given a 23-bit significand plus an appended leading one) and an enabled 48-bit register (product register). There also exists a 24-bit adder, a three to one multiplexer, and a finite state controller, which sends the appropriate control signals to the hardware. The hardware looks like the following:

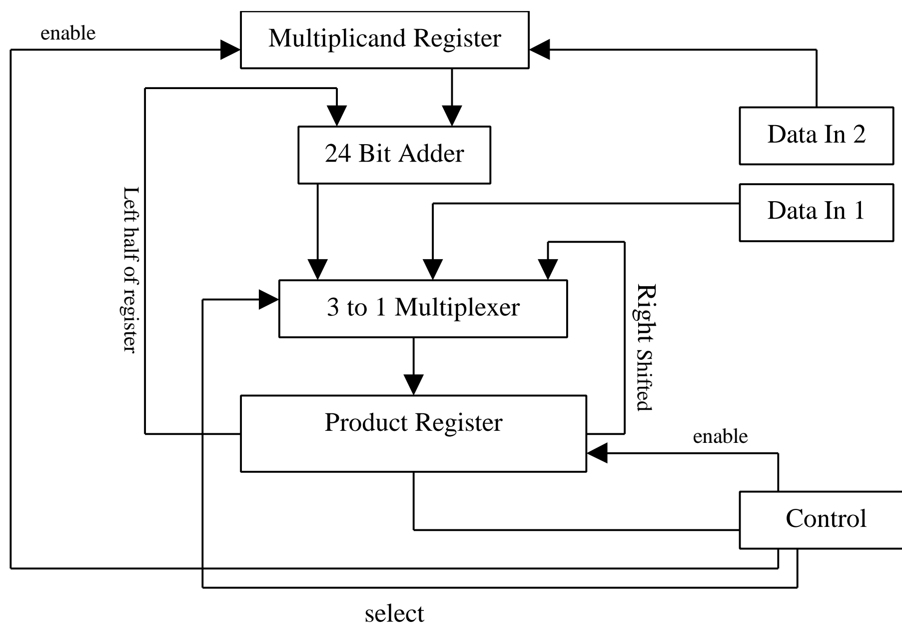

Figure A.1

Data 1 and Data 2 represent the numbers that are being multiplied. The multiplication process takes 24 cycles, but this sacrifice in speed allows for a reduction of area, which was crucial for our project. The actual control algorithm is implemented in a finite state machine, which is outlined below:

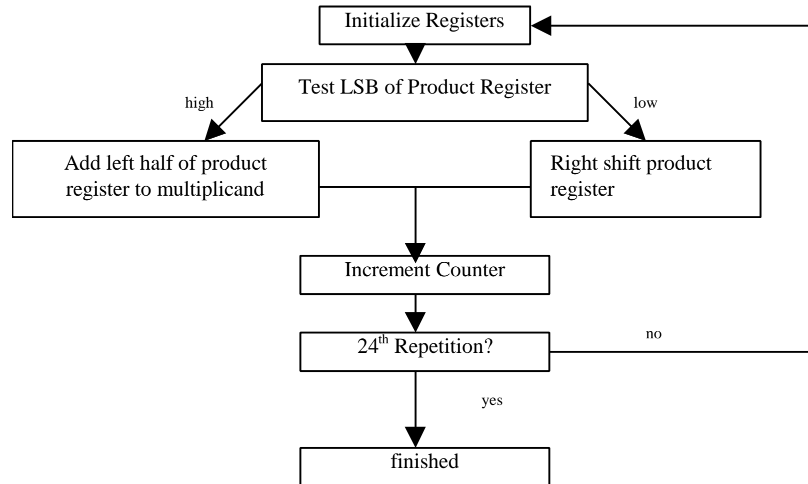

Figure A.2

In the initial state the registers are initialized with the two values that are to be multiplied, and the counter is reset to zero. Then the least significant bit of the product register is tested to see if it is high or low. If the bit is high, the left half of the product is added to the multiplicand then written to the left half of the product register. If the bit is low, then the product register is shifted right by one bit. Next, the counter is incremented to denote the passing of a cycle. This step is followed by a test of the counter, to see if all the necessary cycles have been performed. Since we are multiplying two 24-bit numbers, the counter must equal 24 before we are finished. Once the 24th repetition has been performed, the multiplier is finished and exerts a valid signal for two clock cycles before beginning the process again.

Floating-Point Multiplier

The floating-point multiplier was the largest single element in our design, eating up 86% of our available configurable logic blocks on the chip. The hardware for the multiplier is outlined below:

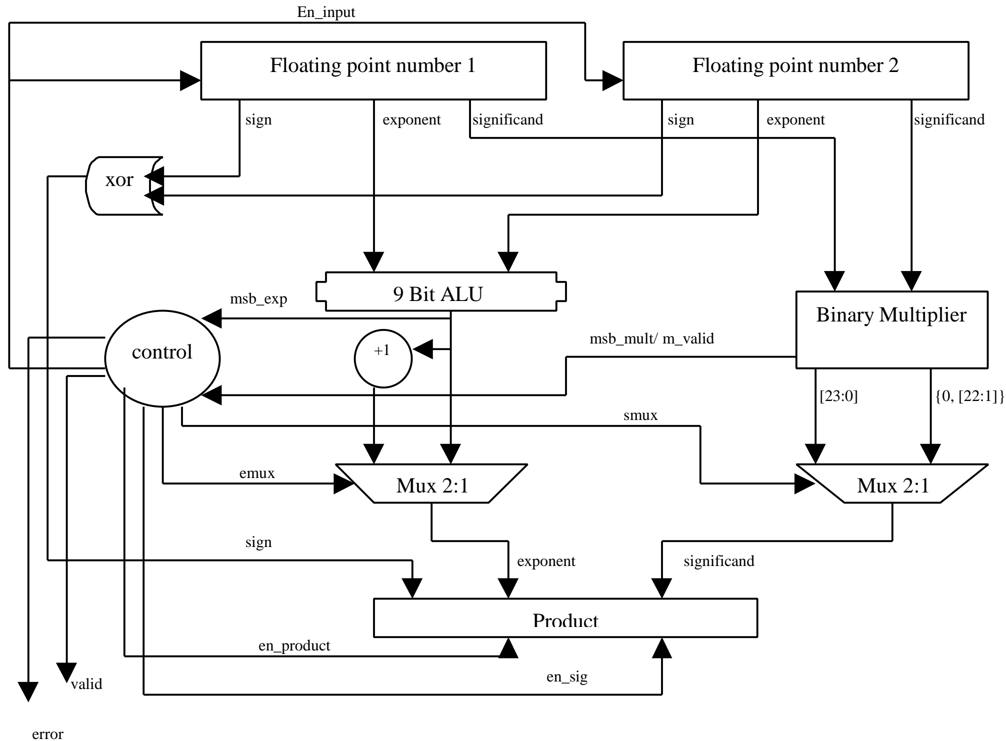

Figure A.3

The two floating-point numbers being multiplied are held in two 32-bit enabled registers. The sign bit of the product is found by simply performing an exclusive-or on the two sign bits. This result is then written directly to the product register. The exponents are sent to a 9-bit ALU which adds the two exponents and then either passes on the sum (minus the 127 bias), or the sum plus one if overflow occurs in the multiplier. This is controlled using a 2 to 1 multiplexer, which receives a select signal from the control logic. The significand are multiplied using the binary multiplier discussed earlier. The result from this block is also sent to a multiplexer, which is sent a control signal denoting whether overflow has occurred or not. If it has, the right shifted product is sent to the product register significand, if not then the unaltered product is sent. The control signal detects overflow in the binary multiplier by examining the top bit of the significand product, which is one bit larger than necessary. Overflow and underflow in the product register are detected by looking at the most significant bit of the exponent, which is also allowed one extra bit for this detection. If the most significant bit of the exponent is high after normalization, then there is an error and the product register is invalid. Setting an error signal high denotes this occurrence, which can be seen by top level control.

The control logic was again implemented in a finite state machine. An outline of the algorithm used is shown below. The initial state is the reset state, in which the two floating-point numbers are loaded into the appropriate registers, the exponent result register is enabled, and the valid signal is set low. The FSM then enters a delay state, in which it waits for the binary multiplier to finish, since through it lies the critical path of the design. Once the binary multiplier is finished, the result is written to the product significand. The control logic then checks the most significant bit of the binary multiplication product to see if it is normalized. If it is, then the exponent product register is written. However, if it is not, then the right shifted binary multiplication product is sent to the significand product register, and the exponent is incremented by one. Once this is complete, overflow and underflow are checked for and the error bit is set accordingly. The final state sets the valid signal high, and sends the FSM back to its initialization state.

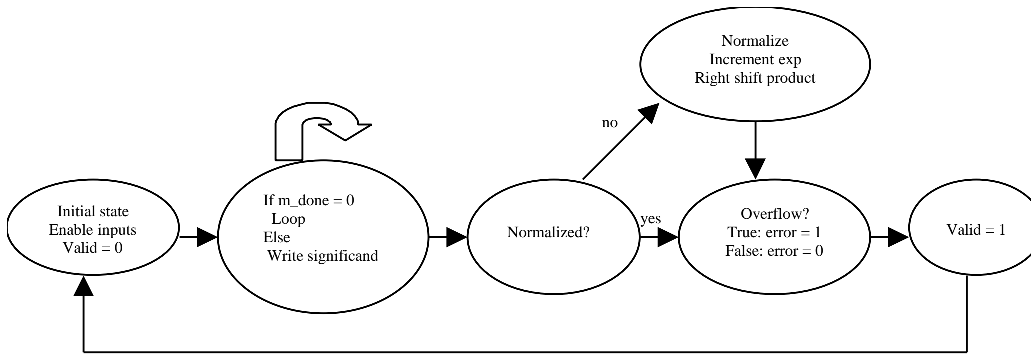

Figure A.4

Floating-Point Division

Floating-Point division resembles floating-point multiplication, however the dilemma it presented was much more immediate because Verilog does not support the division of non-constant numbers. Immediately it was apparent that we would need to design our own binary division hardware, then use it to implement our floating-point division. Unfortunately, once we decided that floating-point division was outside the scope of our project, the Verilog code disappeared. However the basic algorithm that was used still holds true and is outlined below.

This is very similar to floating-point multiplication, however now we are taking the difference of the exponents and dividing the two significands. First to find the sign we can again exclusive-or the sign bits. Next, to solve for the exponent, take the difference of the two input exponents. The significands are then divided, making sure to append the leading one for each. This result may not be normalized and must be checked for normalization. If it is the case that it is not, then simply shift the result right one bit and increment the exponent by one. Finally, we need to check for overflow and underflow. If either of these cases occur we have an exception that must be set.

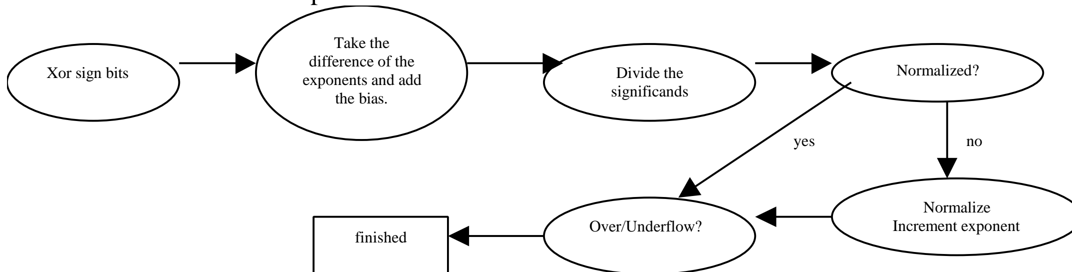

Figure A.5

The binary divider algorithm is analogously similar to the binary multiplication algorithm, again instead of additions, now we are performing subtractions. The algorithm that was implemented in our design is outlined below:

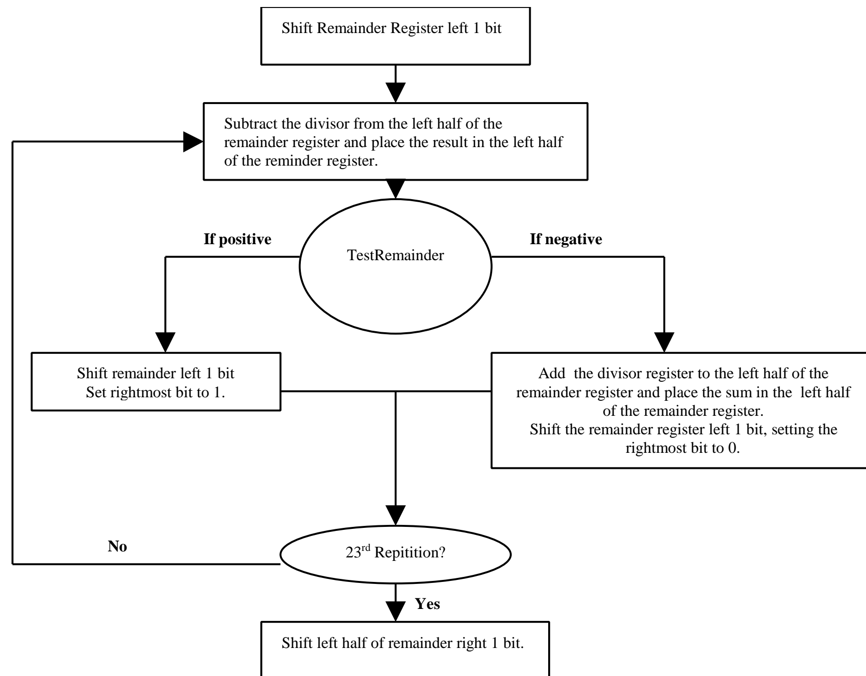

Figure A.6

This design required one 25-bit divisor register and one 50-bit remainder register. These are one and two bits larger than necessary respectively, however sign becomes an issue in this procedure and must be accounted for in the design. The easiest way to do so is to add a sign bit to the numbers being manipulated. The design also requires a 25-bit ALU that performs the necessary functions upon the two registers, and some multiplexers to the appropriate outputs at the appropriate times. The control logic is modeled after the algorithm above, and must count repetitions in order to write the output at the appropriate time.

Unfortunately we were never able to successfully implement this design, although steps had been made in the right direction. Ultimately time constraints forced us to eliminate division from the scope of our project, and area restrictions may have forced it out also, considering what little space remained after our successful implementation of just the addition and subtraction modules.

Binary to Binary Coded Decimal Decoder

In order to be simple and easy to use, we wanted to display our output in decimal. It was an obvious choice because only a select few people know what 528 is in hexadecimal off the top of their heads. Thus we needed some way of decoding these numbers we were calculating in binary to a binary coded decimal representation. We eventually settled upon a method that goes as follows.

Since our output could be no larger than 9,999,999 and no smaller than 0, we proposed to subtract the first four powers of 2 (2^0 = 1, 2^1 = 2, 2^2 = 4, 2^3 = 8) scaled by the decimal unit range desired (10^0 through 10^6) to produce a binary coded decimal number for each unit displayed. We would begin by subtracting the largest amount from the encoded number, or 8,000,000 and work our way down in descending order to 1. If the result of a difference is negative, then the encoded register remains unchanged and the binary coded decimal register for that unit is sent a zero. However, if the difference yields a positive number, then that difference is written to the result register and the binary coded decimal register receives a one. This algorithm and a brief example are given on the next page.

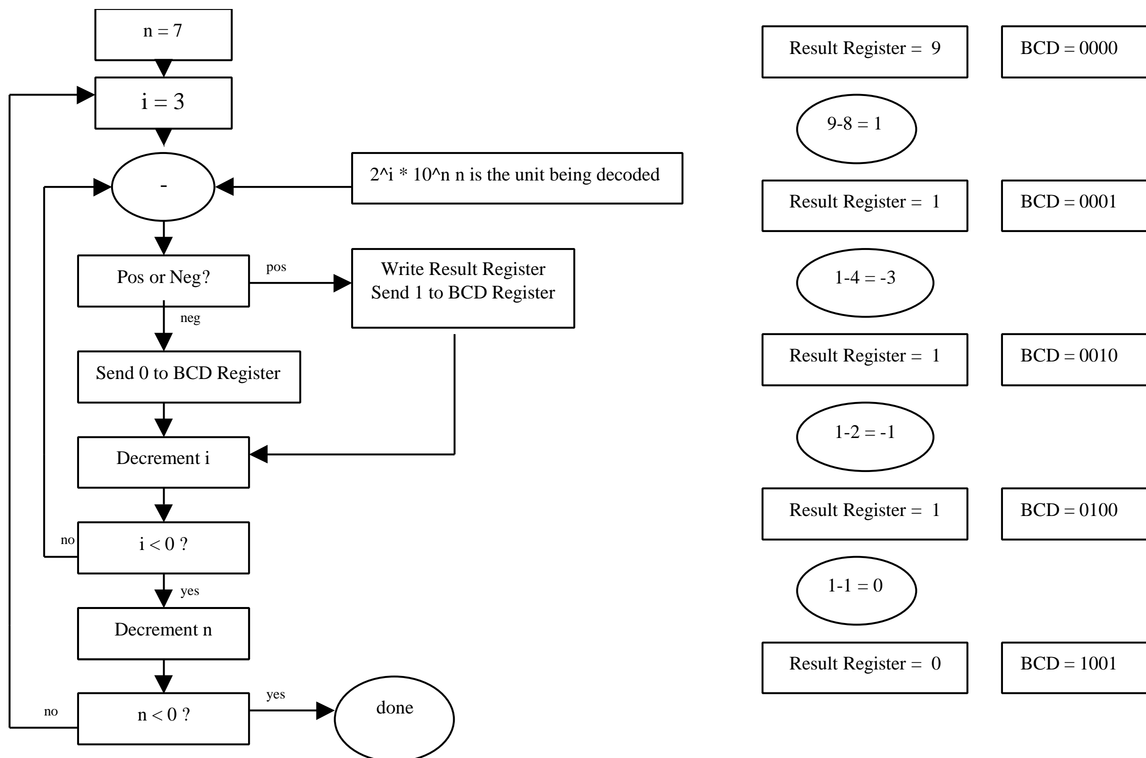

Figure A.7

The example above is simple, only taking into account the ones place, however this method can be expanded to all decimal places, and was done in our design. The BCD register consisted of a 28-bit shift register, that shifted one bit left and placed the incoming bit as the least significant bit. Furthermore, since we had a finite amount of numbers to subtract, we implemented a look up table that provided the necessary value. This design worked in simulation, however did not perform well in the actual implemented design. We believe that timing difficulties existed in the actual implementation that somehow erroneously reset the display to zero continuously.

Floating Point Addition

Floating point addition is a fairly complex process. First the difference between the two exponents must be found. This information is then used to shift the smaller numbers significand so that the two numbers line up correctly. The exponents are then added or subtracted appropriately. The result must then be arithmetically inverted if it is negative and the result sign has to then be changed. The new exponet has to be shifted and the exponent changed so that the result has the right exponent and the significand is in the right position with the implied leading one. The final step is to round the significand with taking into consideration that if the significand is all ones, then the exponent may have to be incremented. Originally, this was done in this project in a single cycle. This, unfortunately, was far too large to fit on a chip even after significant optimization. After that, a multicycle approach was tried using fewer ALU's and big muxes. This unfortunately, didn't work. We were unable to get the foundation software to use the ALU to increment and decrement the exponent so the answer would not come out right.

A single cycle implementation requires several different components the data is processed as described above on a continuous basis. This is understandably very large in implementation. The set-up of such a system can be seen in figure A.8. The original verilog code for this is called float_add and a more opimized version can be seen in float_add_op in appendix C.

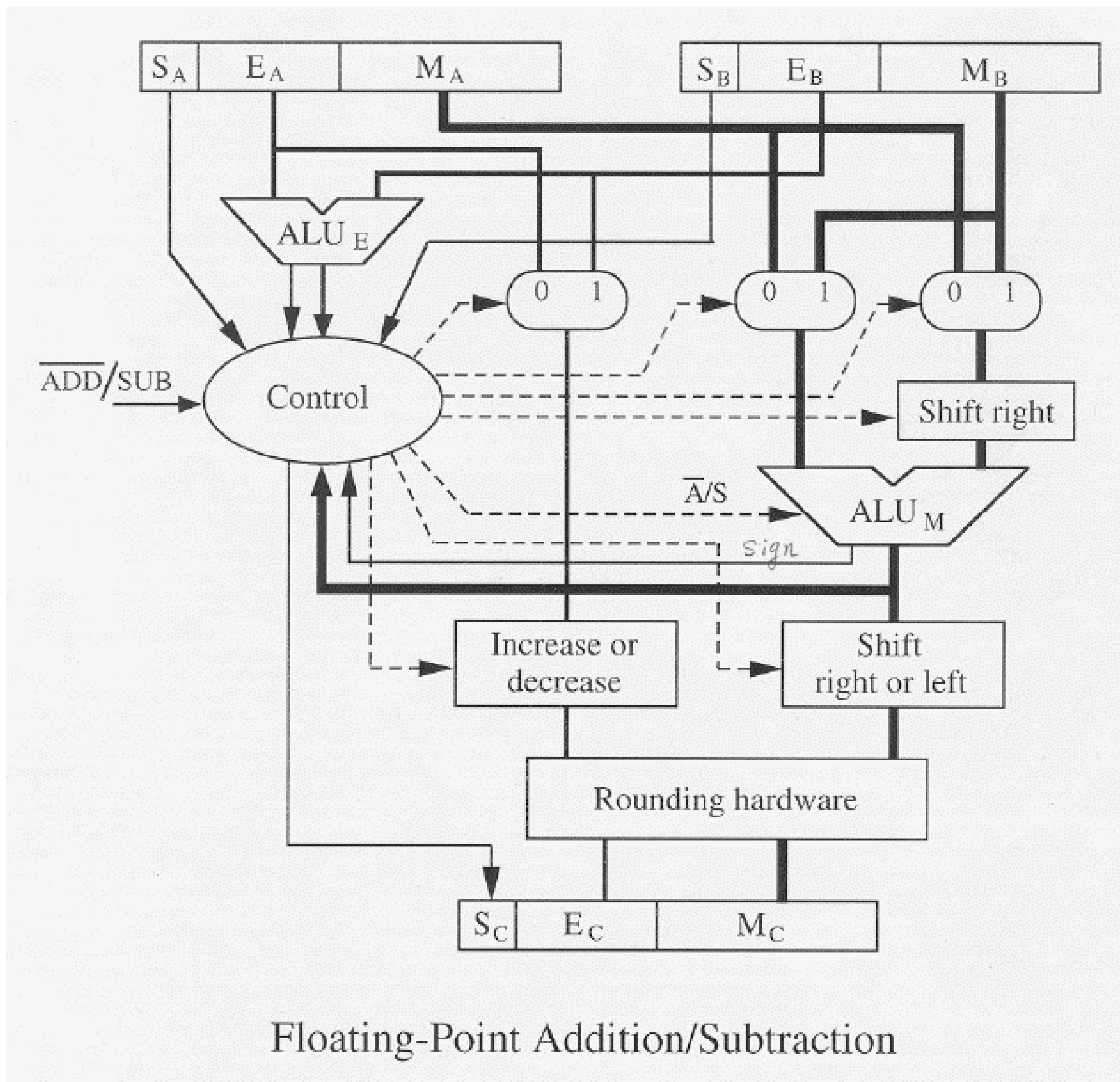

Figure A.8: Single Cycle Floating Point addtion/subration

As is true for most multicycle processes, floating point addition is far more coplicated in several steps rather than just one. The idea of this is that by using just one ALU and only shifting the data one bit at a time, one can save a great deal of space. The implementation of this system did, in fact take up far less space than the single step implementation. As stated earlier, the hardware had trouble incrementing and decrementing the exponent. It did however, correctly find the difference between two exponents. Due to time constraints, this problem was not solved. There is probably a way to fix this problem but it was not found in several hours of trouble shooting. A finite state machine of this system is shown in figure A.9.

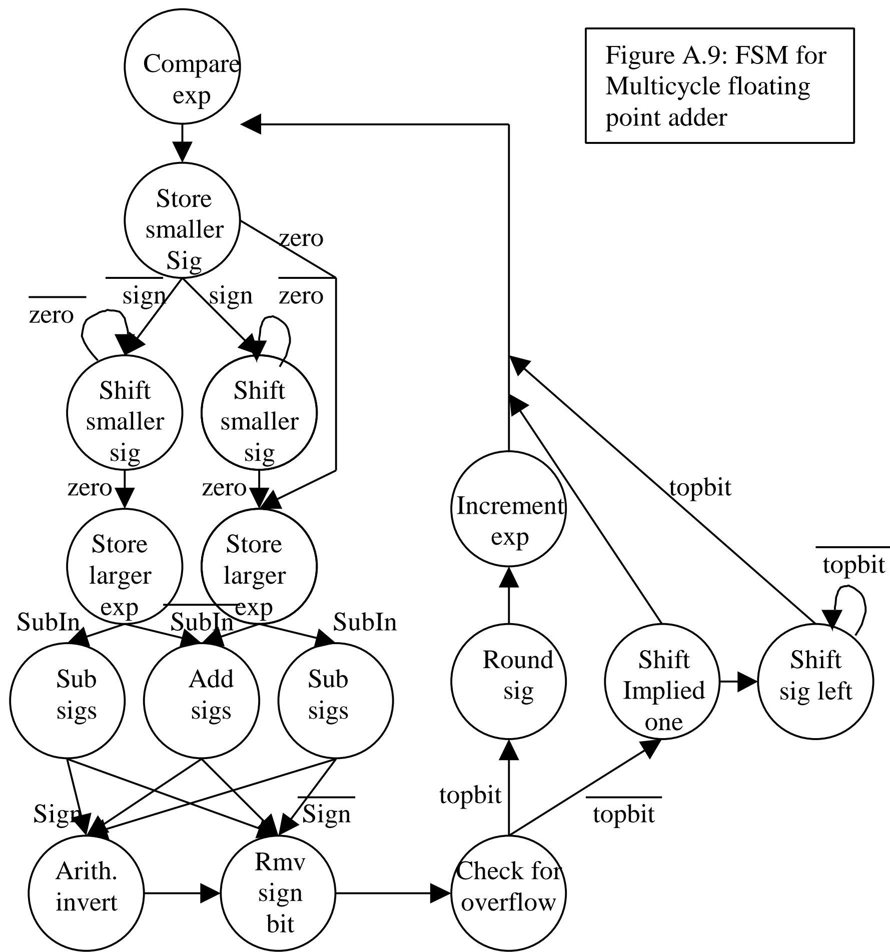

Figure A.9: FSM for Multicycle floating point adder

Appendix B: Verilog Code in Final System

FPGA 1

```
module binary_main3 (key_in, key_out, clk, reset, out, midclk, midclkb, send, error);
// this is the main module used on the first FPGA


//input and output declarations
input [3:0] key_in ;
input clk ;
input reset ;
output midclk, midclkb, send, error;
output [3:0] key_out ;
output [4:0] out;

//wire declarations
wire sclk, mux2db, db2dcdr, enbl, full_enbl, ovrflw, rslt_wrt, adderOp;
wire rslt_clr, disp_clr, disp_wrt;
wire [2:0] funct, opcode;
wire [3:0] M, S, n, num, state3;
wire [24:0] encdr_out, addr_out, addr2aluA, addr2aluB, aluOut, result;
wire [24:0] display, answer, mult_out, arg2;

wire arg_wrt, Inv, rslt_src, sub, m_error;
wire [1:0] addSrcB, addSrcA;
wire [12:0] control;

reg error;
reg [24:0] addrArgB, addrArgA;

assign addSrcA = control[12:11];
assign addSrcB = control[10:9];
assign rslt_clr = control[8];
assign disp_clr = control[7];
assign rslt_wrt = control[6];
assign disp_wrt = control[5];
assign arg_wrt = control[4];
assign Inv = control[2];
assign rslt_src = control[1];
assign sub = control[0];


assign answer = rslt_src?mult_out: addr_out;

always @(posedge clk or posedge reset)
        if (reset) error <= 0;
        else if (ovrflw || m_error) error <= 1;

//the keypad is dirven by the current state
assign key_out = S;

//midclkb is used to run the multiplexed display
assign midclkb = ~midclk;
```

```
//figure the inputs to the input module (calulate input plus result or result times ten)

always @(addSrcA or result or arg2)
        case(addSrcA)
          2'b00: addrArgA <= {result[24], result[20:0], 3'b0};
          2'b01: addrArgA <= result;
          2'b10: addrArgA <= arg2;
          default: addrArgA <= result;
        endcase
always @(addSrcB or result or encdr_out)
        case(addSrcB)
          2'b00: addrArgB <= {result[24], result[22:0], 1'b0};
          2'b01: addrArgB <= encdr_out;
          2'b10: addrArgB <= result;
          default: addrArgB <= result;
        endcase

//calls to the submodules
//clock control
cntr cntr1(clk, midclk, sclk, reset);

//input debounce and decode
driver driver1 (reset, sclk, S);
MUX mux1(key_in, midclk, reset, M, mux2db);
dbncr dbncr1(mux2db, midclk, db2dcdr, reset);
enblr enblr1(enbl, midclk, sclk, full_enbl, reset);
binary_encoder ncdr1(encdr_out, funct, enbl, state3, M, db2dcdr);
three_cyc_dly dly1(midclk, reset, S, state3);

//adder module
binary_adder adder(addrArgA, addrArgB, addr_out, sub, clk, reset, ovrflw, addr2aluA, addr2aluB,
adderOp, aluOut);

//dummy multiplier
pmultplr multplr1(result, arg2, mult_out, m_error, clk, reset);

//Alu
alu_25b ALU(addr2aluA, addr2aluB, aluOut, adderOp);

//result register
ERreg_25b rslt(result, addr_out, midclk, rslt_wrt, reset, rslt_clr, Inv);

//display register
ERreg_25b disp(display, addr_out, midclk, disp_wrt, reset, disp_clr, Inv);

//argument2 register
ERreg_25b_simp arg(arg2, result, midclk, arg_wrt, reset);

//control module
control cont(full_enbl, reset, midclk, funct, result[24], control);

//output module
serial_out_put_5x5 outp(display, out, send, clk, reset);

endmodule
```

```
module binary_adder (A, B, C, sub, clk, reset, ovrflw, argA, argB, alu_op, Alu_res);
//binary adder contains the logic used to perform addition and subracation on a
//25-bit signed binary number. It uses an outside alu so that the same ALU can be used
//for other operations

//declarations
input [24:0] A ;
input [24:0] B ;
output [24:0] C ;
input sub ;
input clk ;
input reset ;
output ovrflw, alu_op ;
output [24:0] argA, argB;
input [24:0] Alu_res;

//wires
wire op, signA, signB, opSel;

//registers
reg signC;
reg [1:0] state, next_state;
reg [24:0] res_reg, C;

//the sign bits are the top bits
assign signA = A[24];
assign signB = B[24];

//determines the operation which the ALU will carry out
assign op = (signA ^ signB) ^ sub;
assign alu_op = opSel?1:op;

//determines the inputs to the ALU
assign argA = state[0]?25'b0:{1'b0, A[23:0]};
assign argB = state[0]?res_reg:{1'b0, B[23:0]};

//dermines if overflow occurs by checking sign bit versus operation
assign ovrflw = ~op & Alu_res[24];

//control the registers
always @(posedge clk or posedge reset)
        if (reset)
        //on reset all registers go to zero
          begin
                state <= 0;
                res_reg <= 0;
                C <= 0;
                signC <= 0;
          end
        //otherwise the ALU result is written to the result register,
        //the state advances,  and possibly the sign of the answer is inverted
        //and the output is put to zero.
        else
          begin
            res_reg <= Alu_res;
            state <= next_state;
```

```
          if (state == 2'b01)
            signC <= ~signA;
          else signC <= signA;
          if (state[1])
            C <= ovrflw?0:{signC, res_reg[23:0]};
        end

//next state logic
always @(state or res_reg)
  case (state)
    2'b00: next_state <= res_reg[24]?2'b01:2'b10;
    2'b01: next_state <= 2'b10;
    2'b10: next_state <= 2'b00;
  endcase

endmodule
```

```
module alu_25b (A, B, C, sub) ;
//this is a simple ALU which performs two's compliment addition
// and subtraction on two 25-bit numbers

//declarations
input [24:0] A ;
input [24:0] B ;
output [24:0] C ;
input sub ;

wire [24:0] B_arg;

//sets the B argument depending on the function
assign B_arg = sub?~B:B;

//performs addtion
assign C = B_arg + A + sub;

endmodule
```

```
//Josh Kihm
//this module is the control for the binary calculator

module control (full_enbl, reset, midclk, funct, sign, cntrl) ;

//declarations
input full_enbl, reset, midclk, sign;
output [12:0] cntrl;
input [2:0] funct ;

wire rslt_wrt, opwrt;
wire [1:0] strd_op;

reg [3:0] state, next_state;
reg [12:0] cntrl;

parameter S00 = 4'b0000;
parameter S01 = 4'b0001;
parameter S02 = 4'b0010;
parameter S03 = 4'b0011;
parameter S04 = 4'b0100;
parameter S05 = 4'b0101;
parameter S06 = 4'b0110;
parameter S07 = 4'b0111;
parameter S08 = 4'b1000;

//this is a enable register which stores the op code
ERreg_2b op_reg(strd_op, funct[1:0], midclk, opwrt, reset);

//control signal logic based on state and the stored operation
always @(state or strd_op)
  case(state)
        S00: cntrl <= 13'b0000000000000;
        S01: cntrl <= 13'b0000001100000;
        S02: cntrl <= 13'b0101001100000;
        S03: cntrl <= 13'b0000110000000;
        S04: cntrl <= 13'b0000000011000;
        S05: cntrl <= {11'b10100001100, strd_op[1:0]};
        S06: cntrl <= 13'b0000000000100;
        S07: cntrl <= 13'b0101001100001;
        S08: cntrl <= 13'b0000100000000;
  endcase

//write the op_reg when ever the appropriate signal is high
assign opwrt = cntrl[3];

//register control
always @(posedge midclk or posedge reset)
  if (reset)
        state <= S00;
  else
        state <= next_state;

//next state logic
always @(state or full_enbl or funct or sign)
  case (state)
```

```
  S00: if (full_enbl)
            case (funct)
              3'b000: next_state <= S01;//number
              3'b001: next_state <= S06;//+/-
              3'b010: next_state <= S05;//=
              3'b100: next_state <= S04;//add
              3'b101: next_state <= S04;//sub
              3'b110: next_state <= S04;//mult
              3'b111: next_state <= S03;//clear
           endcase
           else next_state <= S00;
  S01:  next_state <= sign?S07:S02;
  S02:  next_state <= S00;
  S03:  next_state <= S00;
  S04:  next_state <= S08;
  S05:  next_state <= S08;
  S06:  next_state <= S00;
  S07:  next_state <= S00;
  S08:  next_state <= S00;
  default next_state <= S00;
 endcase

endmodule
```

```
module serial_out_put_5x5 (in, out, send, clk, reset) ;
//this is the output module for the interface between the two chips
//it outputs the input five bits at a time.  It has a send signal which
//marks the beginning of the transfer so that the input and output are
//synchronized.

//declarations
input [24:0] in ;
output [4:0] out ;
output send;
input clk, reset ;

reg [2:0] state, next_state;
reg [4:0] out;

wire [4:0] in1, in2, in3, in4, in5;

parameter S0 = 3'b000;
parameter S1 = 3'b001;
parameter S2 = 3'b010;
parameter S3 = 3'b011;
parameter S4 = 3'b100;
parameter S5 = 3'b101;

//the send signal goes high in the first state
assign send = (state[2:0] == S0);

//give names to the different sections
assign in1 = in[4:0];
assign in2 = in[9:5];
assign in3 = in[14:10];
assign in4 = in[19:15];
assign in5 = in[24:20];


//advance the state with clock
always @(posedge clk or posedge reset)
              if (reset) state <= S0;
              else state <= next_state;

//next state logic and output control based on state
always @(state or in1 or in2 or in3 or in4 or in5)
 case(state)
   S0:begin
      next_state <= S1;
      out <= 5'b0;
     end
   S1:begin
      next_state <= S2;
      out <= in1;
     end
   S2:begin
      next_state <= S3;
      out <= in2;
     end
   S3:begin
```

```
      next_state <= S4;
      out <= in3;
     end
   S4:begin
      next_state <= S5;
      out <= in4;
     end
   S5:begin
      next_state <= S0;
      out <= in5;
     end
  endcase


endmodule
```

```
module dbncr (mux2db, midclk, db2dcdr, reset) ;
//debouncer module from lab four.

//declarations
input reset;
input mux2db ;
input midclk ;
output db2dcdr ;

reg data1, data2, data3, db2dcdr;
wire AllSame;

//if the siganl is steady for three cycles, Allsame goes high
assign AllSame = ((data1 == data2) && (data2 == data3));

//shift register determines the last three values of the input
always @(posedge midclk or posedge reset)
        if (reset)
          begin
            data1 <= 1;
            data2 <= 1;
            data3 <= 1;
            db2dcdr <= 1;
          end
        else
          begin
            data1 <= mux2db;
            data2 <= data1;
            data3 <= data2;
            db2dcdr <= AllSame?data3:db2dcdr;
          end

endmodule

module MUX (K, midclk, reset, M, mux2db) ;
//this module selects the appropriate signal to send to the
// debouncer if there is a change in change in any of the
// siganls.

//declarations
input [3:0] K ;
input midclk ;
input reset ;

output [3:0] M ;
output mux2db ;

reg [3:0] M;
reg mux2db ;

wire A, B, C, D;

//give names to signals
assign A = K[0];
assign B = K[1];
assign C = K[2];
```

```
assign D = K[3];

//output register sends on the opposite of the input
always @(posedge midclk or posedge reset)
            if (reset) M <= 4'b0000;
            else if (K != 4'b1111) M <= ~K;

//determines which signal goes to the debouncer
always @(M or A or B or C or D)
begin
  case (M)
    4'b0001 : mux2db = A;
    4'b0010 : mux2db = B;
    4'b0100 : mux2db = C;
    4'b1000 : mux2db = D;
          default : mux2db = 1;
  endcase
end


endmodule

module driver (reset, sclk, S) ;
//the driver circuit sends the four bits that drive the rows of the keypad

input reset ;
input sclk ;
output [3:0] S;

reg [3:0] S;


//on reset, the default value is given, otherwise it cycles through the
// 'one cold' encoding.
always @(posedge sclk or posedge reset)
          if (reset) S <= 4'b1110;
          else
            begin
              S[3] <= S[2];
              S[2] <= S[1];
              S[1] <= S[0];
              S[0] <= S[3];
            end


endmodule


module enblr (enbl, midclk, sclk, full_enbl, reset) ;
//enblr makes sure that a signal is only read once by sending a
// high signal when the signal only when the enabl signal from the
//decoder is low for the first time since it was last low in the current state.

// delcarations
input enbl ;
input midclk ;
```

```
input sclk ;
input reset ;
output full_enbl ;

reg lst_enbl1;
reg [3:0] lst_enbl2;

//shift register at slow slock frequency, lst_enbl2[3] hold the
//enble input form when the last time the locgic was in the current state
always @(posedge sclk or posedge reset)
        if (reset) lst_enbl2 <= 0;
        else
          begin
            lst_enbl2[0] <= enbl;
            lst_enbl2[1] <= lst_enbl2[0];
            lst_enbl2[2] <= lst_enbl2[1];
            lst_enbl2[3] <= lst_enbl2[2];
          end

//last_enbl1 holds the enbl signal from the last cycle of midclk
always @(posedge midclk or posedge reset)
        if (reset) lst_enbl1 <= 0;
        else lst_enbl1 <= enbl;

//only if enbl is high and it was low on the last cycle of midclk and
//four cycles of sclk ago, is full unble high.
assign full_enbl = enbl && ~lst_enbl1 &&  ~lst_enbl2[3];

endmodule

module cntr (clk, midclk, sclk, reset) ;
//counter is used to slow down the clock for use by the
// various system components.  Midclk runs at 1/1024
//clk frequency and sclk runs at 1/8196 clk frequency


input clk ;
input reset;
output midclk ;
output sclk ;


reg [11:0] cnt;

assign midclk = cnt[8];
assign sclk = cnt[11];

always @(posedge clk or posedge reset)
        if (reset) cnt <= 0;
        else cnt <= cnt +1;

endmodule

module binary_encoder (binary, funct, enbl, state, mux, db2dcdr) ;
//binary encoder is a simple, combinational logic module that
//converts the debounced data from the keypad to the appropriate
```

```
//number and function symbols.

output [24:0] binary ;
output enbl ;
output [2:0] funct;
input [3:0] state ;
input [3:0] mux ;
input db2dcdr ;

reg [2:0] funct;
reg [4:0] low_bits;

assign binary = {21'b0, low_bits};

assign enbl = ~db2dcdr;

//number logic
always @(state or mux)
 case ({state, mux})
  8'b01110100: low_bits <=4'b0000; //0
  8'b11101000: low_bits <=4'b0001; //1
  8'b11100100: low_bits <=4'b0010; //2
  8'b11100010: low_bits <=4'b0011; //3
  8'b11011000: low_bits <=4'b0100; //4
  8'b11010100: low_bits <=4'b0101; //5
  8'b11010010: low_bits <=4'b0110; //6
  8'b10111000: low_bits <=4'b0111; //7
  8'b10110100: low_bits <=4'b1000; //8
  8'b10110010: low_bits <=4'b1001; //9
  default: low_bits <=0;
 endcase

//function logic
always @(state or mux)
 case ({state, mux})
  8'b11100001: funct <= 3'b100; //add
  8'b11010001: funct <= 3'b101; //subtract
  8'b10110001: funct <= 3'b110; //mult
  8'b01110001: funct <= 3'b111; //clear
  8'b01111000: funct <= 3'b001; //plus/minus
  8'b01110010: funct <= 3'b010; //equal
  default: funct <= 0; //no funct
 endcase



endmodule


module three_cyc_dly (midclk, reset, S, state3) ;
//three cycle delay is basically a shift register used to delay the state data
// which runs to the decoder in order tocompenste for the delay through the
// debouncer.

input midclk ;
input [3:0] S ;
```

```
input reset ;
output [3:0] state3 ;

reg[3:0] state1, state2, state3;

always @(posedge midclk or posedge reset)
        if (reset)
          begin
            state1 <= 4'b1111;
            state2 <= 4'b1111;
            state3 <= 4'b1111;
          end
        else
          begin
            state1 <= S;
            state2 <= state1;
            state3 <= state2;
          end


endmodule


module pmultplr (A, B, C, error, clk, reset) ;
//pmultplr is a dummy multiplier module used so that
// the main module and the control module need not be
// changed when the multiplier is taken in and out.
// it actually performs a logic and on the inputs


input[24:0] A, B;
output[24:0]C;
output error;
input clk;
input reset;

reg [24:0] C;

assign error = 0;

always @ (posedge clk or posedge reset)
        if (reset) C <= 0;
        else C <= A & B;

endmodule



module ERreg_2b (out, in, clk, enbl, reset) ;
// this is a two bit enable register with asynchronous reset

input [1:0] in ;
output [1:0] out ;
input enbl ;
input reset ;
input clk ;
```

```
reg [1:0] out;

always @(posedge clk or posedge reset)
  if (reset) out <=0;
  else if (enbl) out <= in;
endmodule



module ERreg_25b_simp (out, in, clk, enbl, reset) ;
// this is a 25-bit enable register with asynchronous reset

input [24:0] in ;
output [24:0] out ;
input enbl ;
input reset ;
input clk ;

reg [24:0] out;

always @(posedge clk or posedge reset)
  if (reset) out <=0;
  else if (enbl) out <= in;
endmodule



module ERreg_25b (out, in, clk, enbl, reset, clr, inv) ;
// this is a 25 bit enable register with both synchronous and asynchronous reset
// as well as a inv input witch causes the top bit to be inverted

input [24:0] in ;
output [24:0] out ;
input enbl ;
input reset ;
input clk, clr, inv ;

reg [24:0] out;

always @(posedge clk or posedge reset)
  if (reset) out <=0;
  else if (clr) out <= 0;
       else if (enbl) out <= in;
                 else if (inv) out [24] <= ~out[24];
endmodule
```

FPGA 2

```
module main (in, clk, reset, out1, out2, out3, out4, receive, midclk, midclkb) ;
//this is the main module of FPGA2, it calls the input module to interface with the
//other FPGA and the out_reg module to output to the seven segment displays
//it also uses the cntr module to produce slower clock signals

input [4:0] in ;
input clk ;
input reset, receive ;
output [7:0] out1 ;
output [7:0] out2 ;
output [7:0] out3 ;
output [7:0] out4 ;
output midclk, midclkb;

wire [24:0] num;
wire sclk;

assign midclkb = ~midclk;

cntr cnt(clk, midclk, sclk, reset);
serial_input_5x5 inpt(num, in, receive, clk, reset) ;
out_reg o_reg({7'b0,num}, out1, out2, out3, out4, clk, midclk, reset);

endmodule
```

```
module serial_input_5x5 (out, in, receive, clk, reset) ;
//this module is the receiving module for our chip to chip interface
// if is basically a shift register

//declarations
output [24:0] out ;
input [4:0] in ;
input receive ;
input clk, reset ;

wire [24:0] wire1;

reg [2:0] state, next_state;
reg [24:0] shft_out;
reg [24:0] out;

parameter S0 = 3'b000;
parameter S1 = 3'b001;
parameter S2 = 3'b010;
parameter S3 = 3'b011;
parameter S4 = 3'b100;
parameter S5 = 3'b101;

//on clk up, state gets the next state, the new data is shifted in and if
// the state is state 5, then the output is written
always @(posedge clk or posedge reset)
        if (reset)
          begin
            shft_out <= 0;
            out <= 0;
            state <= 0;
          end

        else
         begin
          state <= next_state;
          shft_out <= {in, shft_out[24:5]};
          if (state == S0) out <= shft_out;
        end

//next state calculation, if recive is high, the next state is S1
always @(state or receive)
  case(state)
    S0: next_state <= receive?S1:S0;
    S1: next_state <= receive?S1:S2;
    S2: next_state <= receive?S1:S3;
    S3: next_state <= receive?S1:S4;
    S4: next_state <= receive?S1:S5;
    S5: next_state <= receive?S1:S0;
  endcase


endmodule
```

```
module out_reg (in, out1, out2, out3, out4, clk, midclk, reset) ;
//this module uses one display decoder and several registers to diplay the output

// input and output declarations
input [31:0] in ;
output [7:0] out1 ;
output [7:0] out2 ;
output [7:0] out3 ;
output [7:0] out4 ;
input clk ;
input midclk ;
input reset ;

// reg and wire declarations
reg [2:0]cntr;
reg [6:0] decout, dec1, dec2, dec3, dec4, dec5, dec6, dec7, dec8;

wire [3:0] num1, num2, num3, num4, num5, num6, num7, num8, muxout;
wire [7:0] dec_plc, reg_enbl, ans1, ans2, ans3, ans4, ans5, ans6, ans7, ans8;

// the counter simply cycles through eight states
always @(posedge clk or posedge reset)
  if (reset)
   cntr <= 0;
  else
   cntr <= cntr + 1;

//these lines divde up the output, the decimal point is set to off
//replace these 9 lines with decode logic
assign num1 = in[3:0];
assign num2 = in[7:4];
assign num3 = in[11:8];
assign num4 = in[15:12];
assign num5 = in[19:16];
assign num6 = in[23:20];
assign num7 = in[27:24];
assign num8 = in[31:28];
assign dec_plc = 8'b11111111;

//this converts the counter into a one hot signal
one_hot_encdr encdr(cntr, reg_enbl);

//this mux deicdes th input to the decoder
mux8_4b mux1(num1, num2, num3, num4, num5, num6, num7, num8, muxout, cntr);

//this decodes the number for the 7-seg display
//replace with dec_dcdr
Hex_Dcdr dcdr(muxout, decout);

//these registers are written at the appropriate time as the data comes out
//of the 7-segment decoder.  The registers are written one at a time so that
//only one decoder is needed.
erreg_8b reg1(ans1, ({decout, dec_plc[0]}), clk, reset, reg_enbl[0]);
erreg_8b reg2(ans2, ({decout, dec_plc[1]}), clk, reset, reg_enbl[1]);
erreg_8b reg3(ans3, ({decout, dec_plc[2]}), clk, reset, reg_enbl[2]);
erreg_8b reg4(ans4, ({decout, dec_plc[3]}), clk, reset, reg_enbl[3]);
```

```
erreg_8b reg5(ans5, ({decout, dec_plc[4]}), clk, reset, reg_enbl[4]);
erreg_8b reg6(ans6, ({decout, dec_plc[5]}), clk, reset, reg_enbl[5]);
erreg_8b reg7(ans7, ({decout, dec_plc[6]}), clk, reset, reg_enbl[6]);
erreg_8b reg8(ans8, ({decout, dec_plc[7]}), clk, reset, reg_enbl[7]);

//the outputs are simply multiplexed so that fewer pins are needed
assign out1 = midclk?ans1:ans2;
assign out2 = midclk?ans3:ans4;
assign out3 = midclk?ans5:ans6;
assign out4 = midclk?ans7:ans8;


endmodule
```

```
module one_hot_encdr (in, out) ;
//one hot encoder is a combinational moduel that converts
// a binary number to a one hot signal


input [2:0] in ;
output [7:0] out ;

reg [2:0] out;

parameter ONE   = 8'b00000001;
parameter TWO   = 8'b00000010;
parameter THREE         = 8'b00000100;
parameter FOUR = 8'b00001000;
parameter FIVE  = 8'b00010000;
parameter SIX    = 8'b00100000;
parameter SEVEN         = 8'b01000000;
parameter EIGHT         = 8'b10000000;

always @(in)
 case (in)
  0: out <= ONE;
  1: out <= TWO;
  2: out <= THREE;
  3: out <= FOUR;
  4: out <= FIVE;
  5: out <= SIX;
  6: out <= SEVEN;
  7: out <= EIGHT;
 endcase

endmodule
```

```
module mux8_4b (d1, d2, d3, d4, d5, d6, d7, d8, out, sel) ;
// eight way, 4-bit mux

input [3:0] d1, d2, d3, d4, d5, d6, d7, d8 ;
output [3:0] out ;
input [2:0] sel ;

reg [3:0] out;

always @(d1 or d2 or d3 or d4 or d5 or d6 or d7 or d8 or sel)
  case (sel)
    0: out <= d1;
    1: out <= d2;
    2: out <= d3;
    3: out <= d4;
    4: out <= d5;
    5: out <= d6;
    6: out <= d7;
    7: out <= d8;
 endcase

endmodule



module erreg_8b (out, in, clk, reset, enbl) ;
//8 bit enable register w/ asynchronous reset

input [7:0] in ;
input reset ;
input enbl ;
input clk ;
output [7:0] out ;

reg [7:0] out;

always @(posedge clk or posedge reset)
  if (reset) out <= 0;
  else if (enbl) out <= in;

endmodule



module cntr (clk, midclk, sclk, reset) ;
//counter is used to slow down the clock for use by the
// various system components.  Midclk runs at 1/1024
//clk frequency and sclk runs at 1/8196 clk frequency


input clk ;
input reset;
output midclk ;
output sclk ;
```

```
reg [11:0] cnt;

assign midclk = cnt[8];
assign sclk = cnt[11];

always @(posedge clk or posedge reset)
        if (reset) cnt <= 0;
        else cnt <= cnt +1;

endmodule
```

Appendix C: Verilog code not in final System

```
module float_add (A, B, sub, C) ;

input [31:0] A ;
input [31:0] B ;
input sub ;
output [31:0] C ;


//declarations
wire Sa, Sb, Sc;
wire [7:0] Ea, Eb, Ec;
wire [22:0] Ma, Mb, Mc;

wire [8:0] Edif;
wire [8:0] NegEdif;
wire EdifS;

wire inc;
wire MnsPlsA, MnsPlsB;
wire [7:0] intE;

wire [8:0] ShftAmt_1;
wire [4:0] ShftAmt;
wire [4:0] shftLAmt;

wire arg1sgn, arg2sgn, ALUsub;
wire [25:0] arg1, arg2;
reg [25:0] arg1f, arg2f;
wire [25:0] arg2p, AluRes;
wire [25:0] AluResP;

wire [2:0] lead;

assign lead = 3'b001;


//break up the inputs
assign Sa = A[31];
assign Sb = B[31];
assign Ea = A[30:23];
assign Eb = B[30:23];
assign Ma = A[22:0];
assign Mb = B[22:0];

//assemble the output
assign C = {Sc, Ec, Mc};

//find the difference betweent the exponents
assign Edif = Ea-Eb;
assign EdifS = Edif[8];
assign NegEdif = -Edif;

//figure out whether the argumanets are added or subtracted
```

```
assign MnsPlsA = Sa;
assign MnsPlsB = Sb ^ sub;

assign arg1sgn = EdifS ?MnsPlsB:MnsPlsA;
assign arg2sgn = EdifS ?MnsPlsA:MnsPlsB;
assign ALUsub = arg1sgn ^ arg2sgn;

//shift the smaller argument right
shft_right shft1(arg2, arg2p, ShftAmt);

//figure out an expected output
assign intE = EdifS? Eb: Ea;

//figure out how far to shift the smaller argument
assign ShftAmt_1 = EdifS? NegEdif: Edif;
// if the sfift amount is greater than 31, then 31 is sent to the shifter
assign ShftAmt = ~|ShftAmt_1[8:5]?ShftAmt_1[4:0]:5'b11111;



//add leading ones, overflow bit, and sign bit to arguments
// and dicide which one goes to which port of the ALU
assign arg1 = EdifS? ({lead[2:0], Mb}): ({lead[2:0], Ma});
assign arg2 = EdifS? ({lead[2:0], Ma}): ({lead[2:0], Mb});



//do any necesary unary inversion on the arguments
always @(EdifS or MnsPlsA or MnsPlsB or arg1 or arg2p)
case ({EdifS, MnsPlsA, MnsPlsB})
3'b000:  begin
           arg1f <= arg1;
           arg2f <= arg2p;
         end
3'b001: begin
           arg1f <= arg1;
           arg2f <= -arg2p;
         end
3'b010: begin
           arg1f <= -arg1;
           arg2f <= arg2p;
         end
3'b011: begin
           arg1f <= -arg1;
           arg2f <= -arg2p;
         end
3'b100: begin
           arg1f <= arg1;
           arg2f <= arg2p;
         end
3'b101: begin
           arg1f <= -arg1;
           arg2f <= arg2p;
         end
3'b110: begin
           arg1f <= arg1;
```

```
                arg2f <= -arg2p;
              end
3'b111: begin
                arg1f <= -arg1;
                arg2f <= -arg2p;
              end
default: begin
                arg1f <= arg1;
                arg2f <= arg2p;
              end
endcase


//add the arguments and find the sign
assign AluRes = arg1f+arg2f;
assign Sc = AluRes[25];

//perform unary inversion if necissary
assign AluResP = Sc? -AluRes: AluRes;

//figure out how far to shift the result
shft_fndr one(AluResP, shftLAmt);

//shift the result that far
Shft_lft lftShft(shftLAmt, AluResP, Mc, inc);

//figure out the result exponent as the estimated exponent, minus the
// amount of shifting necissary, plus the overflow bit, plus two
// (two is the amount of shifting necissary if there was no overflow)
assign Ec = intE - shftLAmt + inc + 2;

endmodule

module float_add_op (A, B, sub, C) ;

input [31:0] A ;
input [31:0] B ;
input sub ;
output [31:0] C;

//declarations
wire Sa, Sb, Sc;
wire [7:0] Ea, Eb, Ec;
wire [22:0] Ma, Mb, Mc;


wire [8:0] Edif;
wire [8:0] NegEdif;
wire EdifS;

wire inc;
wire MnsPlsA, MnsPlsB;
wire [7:0] intE;

wire [8:0] ShftAmt_1;
wire [4:0] ShftAmt;
```

```
wire [4:0] shftLAmt;

wire arg1sgn, arg2sgn, ALUsub;
wire [25:0] arg1, arg2, arg2f, arg2p, AluRes;
wire [25:0] AluResP;

wire [2:0] lead;

assign lead = 3'b001;


//break up the inputs
assign Sa = A[31];
assign Sb = B[31];
assign Ea = A[30:23];
assign Eb = B[30:23];
assign Ma = A[22:0];
assign Mb = B[22:0];

//assemble the output
assign C = {Sc, Ec, Mc};

//find the difference betweent the exponents
assign Edif = Ea-Eb;
assign EdifS = Edif[8];
assign NegEdif = -Edif;

//figure out whether the argumanets are added or subtracted
assign MnsPlsA = Sa;
assign MnsPlsB = Sb ^ sub;

assign arg1sgn = EdifS ?MnsPlsB:MnsPlsA;
assign arg2sgn = EdifS ?MnsPlsA:MnsPlsB;
assign ALUsub = arg1sgn ^ arg2sgn;

//shift the smaller argument right
shft_right shft1(arg2, arg2p, ShftAmt);

//figure out an expected output
assign intE = EdifS? Eb: Ea;

//figure out how far to shift the smaller argument
assign ShftAmt_1 = EdifS? NegEdif: Edif;
// if the sfift amount is greater than 31, then 31 is sent to the shifter
assign ShftAmt = ~|ShftAmt_1[8:5]?ShftAmt_1[4:0]:5'b11111;

//add leading ones, overflow bit, and sign bit to arguments
// and dicide which one goes to which port of the ALU
assign arg1 = EdifS? {lead[2:0], Mb}: {lead[2:0], Ma};
assign arg2 = EdifS? {lead[2:0], Ma}: {lead[2:0], Mb};

assign arg2f = ALUsub?(~arg2p):arg2p;

//add the arguments and find the sign
assign AluRes = arg1 + arg2f + ALUsub;
assign Sc = AluRes[25] ^ arg1sgn;
```

```
//perform unary inversion if necissary
assign AluResP = arg1sgn? -AluRes: AluRes;

//figure out how far to shift the result
shft_fndr one(AluResP, shftLAmt);

//shift the result that far
Shft_lft lftShft(shftLAmt, AluResP, Mc, inc);

//figure out the result exponent as the estimated exponent, minus the
// amount of shifting necissary, plus the overflow bit, plus two
// (two is the amount of shifting necissary if there was no overflow)
assign Ec = intE - shftLAmt + ({1'b1, inc});

endmodule


//This code performs floating point multiplication on two IEEE 754
//floating point numbers

module f_mult(aclk, areset, d1, d2, result, aerror);

input           aclk, areset;
input   [31:0]  d1, d2;
output  [31:0]  result;
output          aerror;

reg     [31:0]  result;

wire    [31:0]  multiplier, multiplicand;
wire    [8:0]   exp_sum;                    //one bit larger than actual exponent to account for
overflow
wire    [8:0]   exp_inc;            //exponent incremented by 1
wire    [8:0]   emux_out;
wire    [8:0]   exponent;
wire    [47:0]  product;            //48 bits in length, 23 bits plus inferred leading one
wire    [22:0]  significand;                //23 bit significand
wire    [22:0]  smux_out;
wire            sign, aerror;
wire            smux, emux;                 //mux select signals
wire            en_input, en_sig, en_exp;   //enable signals
wire            complete;                   //goes high upon multiplication completion

parameter BIAS   = 7'b111_1111;             //bias of 127 decimal
parameter MAX_EXP = 8'b1111_1110;           //overflow limit = 254 decimal for given bias
parameter MIN_EXP = 1'b1;                   //underflow limit = 1 decimal for given bias

assign exp_inc = exponent + 1;
assign sign = multiplier[31] ^ multiplicand[31];     //exclusive-or the sign bits



f_mult_cont fcont(aclk, areset, significand[22], exponent[8], m_valid, smux, emux, en_input, en_sig,
en_exp, aerror, complete);
```

```
reg_32 d1_reg(aclk, areset, en_input, d1, multiplier);
reg_32 d2_reg(aclk, areset, en_input, d2, multiplicand);
reg_23 sig_reg(aclk, areset, en_sig, smux_out, significand);
reg_9  exp_reg(aclk, areset, en_exp, emux_out, exponent);


wire      [22:0]    rshift_sig;
assign rshift_sig = {0, significand[22:1]};     //right shifted significand, used if unnormalized

//mux used to select for unnormalized significand
//if unnormalized, set smux = 1
mux2_1_23b sig_mux(smux, product[47:25], rshift_sig, smux_out);

mux2_1_9b exp_mux(emux, exp_sum, exp_inc, emux_out);


wire      [23:0]    sig1, sig2;
assign    sig1 = {1, multiplier[22:0]};
assign    sig2 = {1, multiplicand[22:0]};

//multiplication is slow and takes upwards of 120 clock cycles
//must wait for m_valid signal before floating-point multiplication is correct
mult f_product(aclk, areset, sig1, sig2, product, m_valid);

//if bias is not 127, need to change alu_8 code
alu_8 f_exponent(multiplier[30:23], multiplicand[30:23], BIAS, exp_sum);

always @ (posedge aclk or posedge areset)
  if (areset) result = 0;
  else if (complete) result = {sign, exponent[7:0], significand};

endmodule


//a 24-bit unsigned multiplier controller

module f_mult_cont(clk, reset, msb_sig, msb_exp, m_valid, smux, emux, en_input, en_sig, en_exp, error,
complete);

input          clk, reset;
input          msb_sig;              //msb of product register, used to detect normalization
input          msb_exp;              //msb of exponent register, used to detect over/underflow
input          m_valid;       //indicates completion of mult module

output         smux;                 //selects significand mux output
output         emux;                 //selects exponent mux output
output         en_exp;               //enables exponent register
output         en_input, en_sig; //enables new inputs and significand register
output         complete;             //mulitplication is finished
output         error;                //over/underflow occurs

reg            smux, emux;
reg            en_input, en_sig, en_exp;
reg            complete, error;

parameter
```

```
        S0 = 3'b000,
        S1 = 3'b001,
        S1a = 3'b010,
        S2 = 3'b011,
        S3 = 3'b100,
        S4 = 3'b101,
        S5 = 3'b110;

reg     [3:0]   current_state;
reg     [3:0]   next_state;

always @ (posedge clk or posedge reset)
  begin
    if (reset) current_state <= S0;
    else  current_state <= next_state;
  end

always @ (current_state or msb_sig or msb_exp or m_valid)
  begin
    case (current_state)
        S0: begin
            en_input <= 1;              //inputs are enabled
            en_exp <= 1;        //exponent register is enabled
            en_sig <= 0;        //significand register is disabled
            complete <= 0;              //multiplication is not complete
            smux <= 0;          //set mux for normalized product
            emux <= 0;          //set for sum of exponents
            error <= 0;
            next_state <= S1;
          end
        S1: begin       //wait for mult product
            en_input <= 0;              //lock inputs into registers
            if (m_valid == 0)           //loop until mult module produces a valid result
              next_state <= S1;
            else
            begin
             en_sig <= 1;       //write product to significand register
             next_state <= S1a;
            end
          end
        S1a: begin
                en_sig <= 1;            //keep significand enabled to product is written
                next_state <= S2;
           end
        S2: begin       //check if normalized
           if (msb_sig == 1)            //product is not normalized
            begin
              smux <= 1;        //normalize by setting smux high
              en_sig <= 1;      //enable sig_reg
              next_state <= S3;
            end
           else                 //product is already normalized
        next_state <= S4;
          end
        S3: begin       //increment exponent
            en_sig <= 0;        //disable significand register
```

```
                emux <= 1;              //set emux for incremented exponent
                next_state <= S4;
              end
            S4: begin           //check for over/underflow
                en_sig <= 0;            //disable sig_reg
                en_exp <= 0;            //disable exponent register
                if (msb_exp == 1)                   //overflow occurs
                  error <= 1;
                next_state <= S5;
              end
            S5: begin           //multiplication is complete
                complete <= 1;
                next_state <= S0;
              end
     default: next_state <= S0;
   endcase
  end
endmodule
//a 24-bit multiplier
module mult(clk, reset, d1, d2, dout, done, error);

input               clk, reset;
input     [23:0]    d1, d2;
output    [23:0]    dout;
output              done, error;

reg       [23:0]    dout;
reg                 done, error;

wire      [23:0]    multiplicand;
wire      [47:0]    rshift_product;             //right shifted product register
wire      [47:0]    product;
wire      [47:0]    mux_out;
wire      [23:0]    add_out;
wire      [1:0]     smux;
wire                enable, en_multreg, valid;

parameter ZERO = 24'h00_0000;
parameter MAX_VALUE = 24'd9999999;

assign rshift_product = product >> 1;           //right shifted product register

//multiplication controller
mult_cont mcont(clk, reset, product[0], smux, enable, en_multreg, valid);

reg_48 product_reg(clk, reset, enable, mux_out, product);       //enabled product register
reg_24 mult_reg(clk, reset, en_multreg, d1, multiplicand);      //multiplicand register

//24-bit addition module
assign add_out = multiplicand + product[47:24];

//3 to 1 mux module
mux3_1 product_mux(smux, {add_out, product[23:0]}, rshift_product, {ZERO, d2}, mux_out);

always @ (posedge clk or posedge reset)
 begin
```

```
  if (reset)
  begin
    dout <= 0;
    error <= 0;
    done <= 0;
  end
  else if (valid == 1)
  begin
    done <= 1;
    dout <= product[23:0];
    if (product > MAX_VALUE)
      error <= 1;
  end
  else
  begin
    dout <= 0;
    done <= 0;
    error <= 0;
  end//a 24-bit unsigned multiplier controller

module mult_cont(clk, reset, din, smux, en1, en2, valid);

input             clk, reset;
input             din;                 //lsb of product register
output   [1:0]    smux;                //selects mux output
output            en1, en2, valid;     //indicates write to register enables, valid product

reg      [1:0]    smux;
reg               en1, en2, valid;

parameter
         S0 = 3'b000,
         S1 = 3'b001,
         S2 = 3'b010,
         S3 = 3'b011,
         S4 = 3'b100,
         S5 = 3'b101,
         S6 = 3'b110,
         S7 = 3'b111;

reg      [4:0]    rep;                 //repetition register
reg      [3:0]    current_state;
reg      [3:0]    next_state;

always @ (posedge clk or posedge reset)
  begin
    if (reset) current_state <= S0;
    else  current_state <= next_state;
  end

always @ (posedge clk)                 //increment counter at given states on posedge clk
  begin
    if (current_state == S0) rep <= 0;
    else if (current_state == S4) rep <= rep + 1;
  end
```

```
always @ (current_state or din or rep)
  begin
    case (current_state)
          S0: begin
                en1 <= 1;                          //write to register
                en2 <= 1;
                valid <= 0;
                smux <= 2;              //select initial input to multiplier
                next_state <= S1;
              end
          S1: begin                                //test lsb of product register
                en1 <= 0;                          //don't write product register
                en2 <= 0;                          //multiplication register never changes
                if (din == 0) next_state <= S3;
                else next_state <= S2;
              end
          S2: begin                                //add multiplicand to left half of product and place in left half
of product
                smux <= 0;              //set mux for addition output
                en1 <= 1;               //write to register
                next_state <= S3;
              end
          S3: begin                                //right shift the product register by 1 bit
                smux <= 1;              //set mux for right shift output
                en1 <= 1;               //write to register
                next_state <= S4;
              end
          S4: begin
                en1 <= 0;               //disable register
                if (rep == 23)
                begin
                  valid <= 1;
                  next_state <= S6;
                end
                else next_state <= S1;
              end
          S6: begin
                en1 <= 0;               //disable register
                next_state <= S7;
              end
          S7: next_state <= S0;                    //delay state so valid signal is read correctly
          default: next_state <= S0;
    endcase
  end
endmodule


//Chris Kalima
//This code performs floating point division on two IEEE 754
//floating point numbers

module div(clk, reset, num, den, result, error);

input               clk, reset;
input      [31:0]   num, den;
output     [31:0]   result;
```

```
output              error;

reg     [8:0]   exponent;                       //one bit larger than actual exponent to account for
overflow
reg     [47:0]  quotient;               //48 bits in length, 23 bits plus inferred leading one
reg     [22:0]  significand;                    //23 bit significand
reg             sign, error;

parameter BIAS   = 7'b111_1111;                 //bias of 127 decimal
parameter MAX_EXP = 8'b1111_1110;               //overflow limit = 254 decimal for given bias
parameter MIN_EXP = 1'b1;                       //underflow limit = 1 decimal for given bias

//=================
//ASSUMES num/den
//=================

always @ (posedge clk or posedge reset)
  if (reset)
    begin
    error = 0;
    sign = 0;                                   //set result to zero on reset
    exponent = 0;
    significand = 0; end
    else if (num[22:0] == 0)
    begin
    error = 0;
    sign = 0;                                   //set result to zero if num = 0
    exponent = 0;
    significand = 0; end
    else if (den[22:0] == 0)            //divide by zero
    begin                               //set result as indeterminate = FFC0_0000
    error = 1;                                  //set error as high to show invalid result
    sign = 1;
    exponent = 8'b1111_1111;
    significand = 23'b100_0000_0000_0000_0000_0000; end
    else begin
//compute resultant exponent
    exponent = num[30:23] - den[30:23] - BIAS;          //take the difference of exponents and subtract bias
//compute resultant significand
    quotient = {1,num[22:0]} / {1,den[22:0]};//append leading ones and divide
    if (quotient[47] == 1) begin        //check highest bit to see if normalized
            exponent = exponent + 1;    //increment exponent
            quotient = quotient >> 1; end       //shift quotient right one bit
    quotient = quotient << 2;                   //shift left by 2 to truncate leading one
    significand = quotient [47:25];     //place high 23 bits in significand
//check for overflow and underflow
    if (exponent[8] == 1)               //msb will go high if overflow or underflow occurs
            error = 1;
    else
            error = 0;
//compute resultant sign
    sign = num[31] ^ den[31];                   //exclusive-or the sign bits
  end

assign result = {sign, exponent[7:0], significand};
```

```
endmodule


module out_main (in, clk, reset, out1, out2, out3, out4, receive, midclk, midclkb) ;

input [4:0] in ;
input clk ;
input reset, receive ;
output midclk, midclkb;
output [7:0] out1 ;
output [7:0] out2 ;
output [7:0] out3 ;
output [7:0] out4 ;

wire [27:0] bcd_num;
wire [24:0] num;
wire sclk, sign;


parameter ERROR = 32'hffff_ff60;

assign midclkb = ~midclk;
assign sign = num[24];

cntr cnt(clk, midclk, sclk, reset);
serial_input_5x5 inpt(num, in, receive, clk, reset);
result_decoder decode(clk, reset, num[23:0], bcd_num);
out_reg o_reg(bcd_num, sign, out1, out2, out3, out4, clk, midclk, reset);
endmodule



module result_decoder(clk, reset, din, dout);

input           clk, reset;
input   [23:0]  din;            //unsigned 24-bit input
output  [27:0]  dout;           //28-bit BCD output

wire    [27:0]  bcd_num;
wire    [23:0]  mux_out, alu_in1;
wire    [23:0]  alu_in2;
wire    [23:0]  m1, m2, m3, m4, m5, m6, m7;
wire    [2:0]   l_mux;
wire    [1:0]   s_lut;
wire            s_mux, en, en_s, valid;

//testing purposes
wire    [24:0]  difference;
assign  difference = alu_in1 - alu_in2;
wire    sign;
assign  sign = difference[24];
wire    num;
assign  num = ~sign;            //BCD takes opposite of sign value for our method

decode_cont cont(clk, reset, sign, s_mux, l_mux, s_lut, en, en_s, valid);
mux2_1_24b d_mux(s_mux, din, difference[23:0], mux_out);
```

```
mux7_1_24b lut_mux(l_mux, m7, m6, m5, m4, m3, m2, m1, alu_in2);
reg_24 d_reg(clk, reset, en, mux_out, alu_in1);
lut_ones ones(s_lut, m1);
lut_tens tens(s_lut, m2);
lut_hund hund(s_lut, m3);
lut_thou thou(s_lut, m4);
lut_ttho ttho(s_lut, m5);
lut_htho htho(s_lut, m6);
lut_mil   mil(s_lut, m7);
shift_reg28 out_reg(clk, reset, en_s, num, bcd_num);
reg_28 oreg(clk, reset, valid, bcd_num, dout);

endmodule


//FSM to control the decoder
//works by cycling through all 7 powers of ten (7 digit display)
//and subtracting decreasing powers of two (3-0)

module decode_cont(clk, reset, din, s_mux, l_mux, s_lut, en, en_shift, complete);

input             clk, reset, din;
output            s_mux, en, en_shift, complete;
output   [1:0]    s_lut;
output   [2:0]    l_mux;
reg      [1:0]    s_lut;
reg               s_mux, en, en_shift, complete;

parameter
        S0 = 3'b000,
        S1 = 3'b001,
        S2 = 3'b010,
        S3 = 3'b011,
        S4 = 3'b100,
        S5 = 3'b101;

reg      [2:0]    current_state;
reg      [2:0]    next_state;
reg      [2:0]    rep;

assign l_mux = rep;                         //sets l_mux for proper power of ten

always @ (posedge clk or posedge reset)
  begin
    if (reset) current_state <= S0;
    else  current_state <= next_state;
  end

always @ (posedge clk or posedge reset)
  begin
    if (reset) rep <= 0;
    else if (current_state == S4)
      rep <= rep + 1;
    else if (complete)
      rep <= 0;
  end
```

```
always @ (current_state or din or rep)
  begin
    case (current_state)
            S0: begin
                  complete <= 0;
                  en <= 1;                                    //enable decode register
                  s_mux <= 0;               //set mux to result register
                  en_shift <= 0;
                  next_state <= S1;
                end
            S1: begin
                  s_mux <= 1;               //set mux to alu output
                  en_shift <= 1;            //enable shift register
                  if (din == 0) en <= 1;   //if positive, enable decode register
                  else en <= 0;             //if negative, maintain decode register
                  s_lut <= 0;               //look up first value
                  next_state <= S2;
                end
            S2: begin
                  if (din == 0) en <= 1;
                  else en <= 0;
                  s_lut <= 1;
                  next_state <= S3;
                end
            S3: begin
                  if (din == 0) en <= 1;
                  else en <= 0;
                  s_lut <= 2;
                  next_state <= S4;
                end
            S4: begin
                  if (din == 0) en <= 1;
                  else en <= 0;
                  s_lut <= 3;
                  next_state <= S5;
                end
            S5: begin
                  en_shift <= 0;
                  en <= 0;
                  s_lut <= 0;
                  if (rep == 7)
                  begin
                    complete <= 1;
                    next_state <= S0;
                  end
                  else
                    next_state <= S1;
                end
            default: next_state <= S0;
    endcase
  end
endmodule
```

```
module float_encoder (float, funct, enbl, state, mux, db2dcdr) ;

output [31:0] float ;
output enbl ;
output [2:0] funct;
input [3:0] state ;
input [3:0] mux ;
input db2dcdr ;

reg [2:0] funct;
reg [31:0] float;


assign enbl = ~db2dcdr;

always @(state or mux)
 case ({mux, state})
  8'b01000111: float <= (32'h00000000); //0
  8'b10001110: float <= (32'h3f800000); //1
  8'b01001110: float <= (32'h40000000); //2
  8'b00101110: float <= (32'h40400000); //3
  8'b10001101: float <= (32'h40800000); //4
  8'b01001101: float <= (32'h40A00000); //5
  8'b00101101: float <= (32'h40C00000); //6
  8'b10001011: float <= (32'h40E00000); //7
  8'b01001011: float <= (32'h41000000); //8
  8'b00101011: float <= (32'h41100000); //9
  default: float <=0;
 endcase

always @(state or mux)
 case ({state, mux})
  225: funct <= 3'b100; //add
  226: funct <= 3'b101; //subtract
  228: funct <= 3'b110; //mult
  232: funct <= 3'b111; //divide
  120: funct <= 3'b001; //plus/minus
  216: funct <= 3'b010; //decimal
  default: funct <= 0; //no funct
 endcase



endmodule

module float_encoder2 (float, funct, enbl, state, mux, db2dcdr) ;

output [31:0] float ;
output enbl ;
output [2:0] funct;
input [3:0] state ;
input [3:0] mux ;
input db2dcdr ;

reg [2:0] funct;
reg [31:0] float;
```

```
assign enbl = ~db2dcdr;

always @(state or mux)
 case ({state, mux})
  184: float <=32'h00000000; //0
  113: float <=32'h3f800000; //1
  177: float <=32'h40000000; //2
  209: float <=32'h40400000; //3
  114: float <=32'h40800000; //4
  178: float <=32'h40A00000; //5
  210: float <=32'h40C00000; //6
  116: float <=32'h40E00000; //7
  180: float <=32'h41000000; //8
  212: float <=32'h41100000; //9
  default: float <=0;
 endcase

always @(state or mux)
 case ({state, mux})
  225: funct <= 3'b100; //add
  226: funct <= 3'b101; //subtract
  228: funct <= 3'b110; //mult
  232: funct <= 3'b111; //divide
  120: funct <= 3'b001; //plus/minus
  216: funct <= 3'b010; //decimal
  default: funct <= 0; //no funct
 endcase
endmodule

module num_dcdr (enbl, n, s, m, db2dcdr) ;

output enbl ;
output [3:0] n ;
input [3:0] s ;
input [3:0] m ;
input db2dcdr ;

assign enbl = ~db2dcdr;

assign n[0] = ~s[0] || (s[3] && m[2]) || (s[2] && m[3]);
assign n[1] = ~s[0] || m[1] || (m[2] && ~s[3]);
assign n[2] = (m[3] && s[2]) || (~s[1] && ~m[2]) || (~s[2] && m[0]) || (m[2] && (~s[3] || ~s[0]));
assign n[3] = (~s[1] && ~m[1]) || (~s[2] && m[1]) || (~s[0] && (m[1] || m[3])) || (~s[3] && (m[0] || m[2]));

endmodule

module multplr (A, B, C, error, clk, reset) ;

input [24:0] A ;
input [24:0] B ;
output [24:0] C ;
output error ;
input clk, reset;
```

```
wire sign, done;
wire [23:0] ans;

assign sign = A[24] ^ B[24];

mult mult1(clk, reset, A[23:0], B[23:0], ans, done, error);

ERreg_25b_simp result_reg(C, {sign, ans[23:0]}, clk, done, reset);

endmodule


module dec_dcdr (in, out) ;

output [6:0] out;
input [3:0] in;

reg out;

// define a set of numbers representing the outputs
parameter BLNK = 7'b111_1111;
parameter ZERO = 7'b000_0001;
parameter ONE   = 7'b100_1111;
parameter TWO   = 7'b001_0010;
parameter THREE = 7'b000_0110;
parameter FOUR = 7'b100_1100;
parameter FIVE   = 7'b010_0100;
parameter SIX     = 7'b010_0000;
parameter SEVEN             = 7'b000_1111;
parameter EIGHT = 7'b000_0000;
parameter NINE  = 7'b000_0100;
parameter MINUS = 7'b111_1110;
parameter E         = 7'b011_0000;

always@(in)
        case (in)
                0 : out <= ZERO;
                1 : out <= ONE;
                2 : out <= TWO;
                3 : out <= THREE;
                4 : out <= FOUR;
                5 : out <= FIVE;
                6 : out <= SIX;
                7 : out <= SEVEN;
                8 : out <= EIGHT;
                9 : out <= NINE;
                10 : out <= MINUS;
                15 : out <= E;
                default : out <= BLNK;
        endcase

endmodule


module dly (s, sdlyd, midclk, reset) ;
```

```
input reset, midclk;
input [3:0] s ;
output [3:0] sdlyd ;

reg [3:0] s1, s2, sdlyd;

always @(posedge midclk or posedge reset)
        if (reset)
          begin
            s1 <= 4'b1111;
            s2 <= 4'b1111;
            sdlyd <= 4'b1111;
          end

        else
          begin
            s1 <= s;
            s2 <= s1;
            sdlyd <= s2;
          end

endmodule


module num_mem (full_enbl, midclk, n, num, reset) ;

input full_enbl ;
input midclk ;
input [3:0] n ;
output [3:0] num ;
input reset ;

reg [3:0] num1, num2;

assign num = midclk?num1:num2;

always @(posedge midclk or posedge reset)
        if (reset)
          begin
            num1 <= 0;
            num2 <= 0;
          end

        else
           if(full_enbl)
            begin
              num1 <= full_enbl?n:num1;
              num2 <= full_enbl?num1:num2;
            end

endmodule


module shft_fndr (In, out) ;
```

```
input [25:0] In ;
output [4:0] out ;

wire Nor1, Nor2, Nor3, Nor4, Nor5, Nor6, Nor7, Nor8, Nor9, Nor10, Nor11, Nor12, Nor13;

assign Nor1 = ~|(In[25:10]);
assign Nor2 = ~|(In[25:18]);
assign Nor3 = ~|(In[9:2]);
assign Nor4 = ~|(In[25:22]);
assign Nor5 = ~|(In[17:14]);
assign Nor6 = ~|(In[9:6]);
assign Nor7 = ~|(In[25:24]);
assign Nor8 = ~|(In[21:20]);
assign Nor9 = ~|(In[17:16]);
assign Nor10 = ~|(In[13:12]);
assign Nor11 = ~|(In[9:8]);
assign Nor12 = ~|(In[5:4]);
assign Nor13 = ~|(In[1:0]);

assign out[4] = Nor1;
assign out[3] = (~Nor1 & Nor2) | (Nor1 & Nor3);
assign out[2] = (~Nor1 & ~Nor2 & Nor4) | (~Nor1 & Nor2 & Nor5) | (Nor1 & ~Nor3 & Nor6);
assign out[1] = (~Nor1 & ~Nor2 & ~Nor4 & Nor7) | (~Nor1 & ~Nor2 & Nor4 & Nor8) | (~Nor1 & Nor2 &
~Nor5 & Nor9) | (~Nor1 & Nor2 & Nor5 & Nor10) | (Nor1 & ~Nor3 & ~Nor6 & Nor11) | (Nor1 & ~Nor3
& Nor6 & Nor12) | (Nor1 & Nor3 & Nor13);
assign out[0] = (~Nor1 & ~Nor2 & ~Nor4 & ~Nor7 & ~In[25]) | (~Nor1 & ~Nor2 & ~Nor4 & Nor7 &
~In[23]) | (~Nor1 & ~Nor2 & Nor4 & ~Nor8 & ~In[21]) | (~Nor1 & ~Nor2 & Nor4 & Nor8 & ~In[19]) |
(~Nor1 & Nor2 & ~Nor5 & ~Nor9 & ~In[17]) | (~Nor1 & Nor2 & ~Nor5 & Nor9 & ~In[15]) | (~Nor1 &
Nor2 & Nor5 & ~Nor10 & ~In[13]) | (~Nor1 & Nor2 & Nor5 & Nor10 & ~In[11]) | (Nor1 & ~Nor3 &
~Nor6 & ~Nor11 & ~In[9]) | (Nor1 & ~Nor3 & ~Nor6 & Nor11 & ~In[7]) | (Nor1 & ~Nor3 & Nor6 &
~Nor12 & ~In[5]) | (Nor1 & ~Nor3 & Nor6 & Nor12 & ~In[3]) | (Nor1 & Nor3 & ~Nor13 & ~In[1]);


endmodule


module Shft_lft (shft_amt, in,  out, inc) ;

input [25:0] in ;
input [4:0] shft_amt ;
output [22:0] out ;
output inc ;


reg [25:0] int;
wire [23:0] int2;

reg [22:0] out ;

always @(in or shft_amt)
case(shft_amt)
 1:        int <= {in[24:0], 1'b0};
 2:        int <= {in[23:0], 2'b0};
 3:        int <= {in[22:0], 3'b0};
 4:        int <= {in[21:0], 4'b0};
 5:        int <= {in[20:0], 5'b0};
```

```
 6:         int <= {in[19:0], 6'b0};
 7:         int <= {in[18:0], 7'b0};
 8:         int <= {in[17:0], 8'b0};
 9:         int <= {in[16:0], 9'b0};
 10:        int <= {in[15:0], 10'b0};
 11:        int <= {in[14:0], 11'b0};
 12:        int <= {in[13:0], 12'b0};
 13:        int <= {in[12:0], 13'b0};
 14:        int <= {in[11:0], 14'b0};
 15:        int <= {in[10:0], 15'b0};
 16:        int <= {in[9:0], 16'b0};
 17:        int <= {in[8:0], 17'b0};
 18:        int <= {in[7:0], 18'b0};
 19:        int <= {in[6:0], 19'b0};
 20:        int <= {in[5:0], 20'b0};
 21:        int <= {in[4:0], 21'b0};
 22:        int <= {in[3:0], 22'b0};
 23:        int <= {in[2:0], 23'b0};
 24:        int <= {in[1:0], 24'b0};
 25:        int <= {in[0], 25'b0};
 default: int <= 0;
endcase

//int2 rounds off the two LSB of int
assign int2 = int[25:2] + int[1];

//inc checks for overflow
assign inc = ~int2[23];

//out gets inc2 minus the implied leading one.  if there is overflow, then the
// answer is shifted one more place
always @(int2)
if(int2[23])
 out <= int2[22:0];
else
 out <= int2[23:1] + int2[0];

endmodule


module shft_right (Din, Dout, Shft_amt) ;

input [25:0] Din ;
output [25:0] Dout ;
input [4:0] Shft_amt ;

reg [47:0] int;

always @(Din or Shft_amt)
case(Shft_amt)
 0:         int <= {Din[25:0], 22'b0};
 1:         int <= {1'b0, Din[25:0], 21'b0};
 2:         int <= {2'b0, Din[25:0], 20'b0};
 3:         int <= {3'b0, Din[25:0], 19'b0};
 4:         int <= {4'b0, Din[25:0], 18'b0};
 5:         int <= {5'b0, Din[25:0], 17'b0};
```

```
 6:        int <= {6'b0, Din[25:0], 16'b0};
 7:        int <= {7'b0, Din[25:0], 15'b0};
 8:        int <= {8'b0, Din[25:0], 14'b0};
 9:        int <= {9'b0, Din[25:0], 13'b0};
 10:       int <= {10'b0, Din[25:0], 12'b0};
 11:       int <= {11'b0, Din[25:0], 11'b0};
 12:       int <= {12'b0, Din[25:0], 10'b0};
 13:       int <= {13'b0, Din[25:0], 9'b0};
 14:       int <= {14'b0, Din[25:0], 8'b0};
 15:       int <= {15'b0, Din[25:0], 7'b0};
 16:       int <= {16'b0, Din[25:0], 6'b0};
 17:       int <= {17'b0, Din[25:0], 5'b0};
 18:       int <= {18'b0, Din[25:0], 4'b0};
 19:       int <= {19'b0, Din[25:0], 3'b0};
 20:       int <= {20'b0, Din[25:0], 2'b0};
 21:       int <= {21'b0, Din[25:0], 1'b0};
 22:       int <= {22'b0, Din[25:0]};
 default int <=48'b0;
endcase

//round off
assign Dout = int[47:22] + int[21];

endmodule


module io_tester (in, out, clk, reset) ;

input [24:0] in ;
output [24:0] out ;
input clk ;
input reset ;

wire sr;
wire [4:0] int;

serial_out_put_5x5 outp(in, int, sr, clk, reset);
serial_input_5x5 inp(out, int, sr, clk, reset);

endmodule


module lab_4 (clk, reset, midclk, midclkb, S, out, K) ;


input clk ;
input reset ;
input [3:0] K;
output midclk ;
output midclkb ;
output [3:0] S;
output [6:0] out ;

reg [3:0] state1, state2, state3;
wire sclk, mux2db, db2dcdr, enbl, full_enbl;
wire [3:0] M, n, num;
```

```
assign midclkb = ~midclk;

cntr    cntr1 (clk, midclk, sclk, reset);
driver driver1 (reset, sclk, S);
Hex_Dcdr hex1(num, out);
MUX mux1(K, midclk, reset, M, mux2db);
dbncr dbncr1(mux2db, midclk, db2dcdr, reset);
enblr enblr1(enbl, midclk, sclk, full_enbl, reset);
num_mem num_mem1(full_enbl, midclk, n, num, reset);
num_dcdr num_dcdr1(enbl, n, state3, M, db2dcdr);


always @(posedge midclk or posedge reset)
        if (reset)
          begin
            state1 <= 4'b1111;
            state2 <= 4'b1111;
            state3 <= 4'b1111;
          end
        else
          begin
            state1 <= S;
            state2 <= state1;
            state3 <= state2;
          end

endmodule


module min_main (key_in, key_out, clk, reset, out1, out2, out3, out4, midclk, midclkb);

//input and output declarations
input [3:0] key_in ;
input clk ;
input reset ;
output midclk, midclkb;
output [3:0] key_out ;
output [7:0] out1, out2, out3, out4 ;

//wire declarations
wire sclk, mux2db, db2dcdr, enbl, full_enbl;
wire [2:0] funct;
wire [3:0] M, S, n, num, state3;
wire [31:0] ncdr_out, result;

//the keypad is dirven by the current state
assign key_out = S;

//midclkb is used to run the multiplexed display
assign midclkb = ~midclk;

//calls to the submodules
//clock control
cntr    cntr1 (clk, midclk, sclk, reset);
```

```
//input debounce and decode
driver driver1 (reset, sclk, S);
MUX mux1(key_in, midclk, reset, M, mux2db);
dbncr dbncr1(mux2db, midclk, db2dcdr, reset);
enblr enblr1(enbl, midclk, sclk, full_enbl, reset);
float_encoder ncdr1(ncdr_out, funct, enbl, S, M, db2dcdr);
three_cyc_dly dly1(midclk, reset, S, state3);

//result register
ERreg_32b rslt(result, ncdr_out, midclk, full_enbl, reset);

//output register
out_reg oput(result, out1, out2, out3, out4, clk, midclk, reset);


endmodule

module test_main (key_in, key_out, clk, reset, out1, out2, out3, out4, midclk, midclkb);//, result, ncdr_out,
sub, add_out) ;

//input and output declarations
input [3:0] key_in ;
input clk ;
input reset ;
output midclk, midclkb;
output [3:0] key_out ;
output [7:0] out1, out2, out3, out4 ;

//output [31:0] result, ncdr_out, add_out;
//output sub;

//wire declarations
wire sclk, mux2db, db2dcdr, enbl, full_enbl, M_error, sub;
wire [2:0] funct;
wire [3:0] M, S, n, num, state3;
wire [31:0] ncdr_out, add_out, result;

//the keypad is dirven by the current state
assign key_out = S;
assign sub = 0;
//assign the exponent output as the exponent of the answer

//midclkb is used to run the multiplexed display
assign midclkb = ~midclk;

//calls to the submodules
//clock control
cntr    cntr1 (clk, midclk, sclk, reset);

//input debounce and decode
driver driver1 (reset, sclk, S);
MUX mux1(key_in, midclk, reset, M, mux2db);
dbncr dbncr1(mux2db, midclk, db2dcdr, reset);
enblr enblr1(enbl, midclk, sclk, full_enbl, reset);
float_encoder ncdr1(ncdr_out, funct, enbl, S, M, db2dcdr);
three_cyc_dly dly1(midclk, reset, S, state3);
```

```
//arithmetic logic
float_add_op addr1(result, ncdr_out, sub, add_out);

//result register
ERreg_32b rslt(result, add_out, midclk, full_enbl, reset);

//output register
out_reg oput(result, out1, out2, out3, out4, clk, midclk, reset);


endmodule

module final_main (key_in, key_out, float_exp, clk, reset, out1, out2, out3, out4, midclk, midclkb) ;

//input and output declarations
input [3:0] key_in ;
input clk ;
input reset ;
output midclk, midclkb;
output [3:0] key_out ;
output [7:0] float_exp, out1, out2, out3, out4 ;

//wire declarations
wire sclk, mux2db, db2dcdr, enbl, full_enbl, M_error;
wire [2:0] funct;
wire [3:0] M, S, n, num, state3;
wire [31:0] ncdr_out, mult_out, add_out, result;

parameter TEN = 32'h41200000;

//the keypad is dirven by the current state
assign key_out = S;
//assign the exponent output as the exponent of the answer
assign float_exp = result [30:23];
//midclkb is used to run the multiplexed display
assign midclkb = ~midclk;

//calls to the submodules
//clock control
cntr    cntr1 (clk, midclk, sclk, reset);

//input debounce and decode
driver driver1 (reset, sclk, S);
MUX mux1(key_in, midclk, reset, M, mux2db);
dbncr dbncr1(mux2db, midclk, db2dcdr, reset);
enblr enblr1(enbl, midclk, sclk, full_enbl, reset);
float_encoder ncdr1(ncdr_out, funct, enbl, S, M, db2dcdr);
three_cyc_dly dly1(midclk, reset, S, state3);

//arithmetic logic
mult mltplr(clk, reset, TEN, result, mult_out, M_error);
float_add addr1(mult_out, ncdr_out, 1'b0, add_out);

//result register
ERreg_32b rslt(result, add_out, midclk, full_enbl, reset);
```

```
//output register
out_reg oput(result, out1, out2, out3, out4, clk, midclk, reset);


endmodule

//binary encoded decimal conversion look up table
module lut_ones(select, dout);

input   [1:0]   select;
output  [23:0]  dout;
reg     [23:0]  dout;

always @(select)
case (select)
   0: dout = 23'd8;//eight
   1: dout = 23'd4;//four
   2: dout = 23'd2;//two
   3: dout = 23'd1;//one
   default: dout = 0;
  endcase
endmodule


//binary encoded decimal conversion look up table
module lut_tens(select, dout);

input   [1:0]   select;
output  [23:0]  dout;
reg     [23:0]  dout;

always @(select)
case (select)
   0: dout = 23'd80;
   1: dout = 23'd40;
   2: dout = 23'd20;
   3: dout = 23'd10;
   default: dout = 0;
  endcase
endmodule


//binary encoded decimal conversion look up table
module lut_hund(select, dout);

input   [1:0]   select;
output  [23:0]  dout;
reg     [23:0]  dout;

always @(select)
case (select)
   0: dout = 23'd800;
   1: dout = 23'd400;
   2: dout = 23'd200;
   3: dout = 23'd100;
```

```
   default: dout = 0;
  endcase
endmodule


//binary encoded decimal conversion look up table
module lut_thou(select, dout);

input      [1:0]      select;
output    [23:0]    dout;
reg         [23:0]    dout;

always @(select)
case (select)
   0: dout = 23'd8000;
   1: dout = 23'd4000;
   2: dout = 23'd2000;
   3: dout = 23'd1000;
   default: dout = 0;
  endcase
endmodule


//binary encoded decimal conversion look up table
module lut_ttho(select, dout);

input      [1:0]      select;
output    [23:0]    dout;
reg         [23:0]    dout;

always @(select)
case (select)
   0: dout = 23'd80000;
   1: dout = 23'd40000;
   2: dout = 23'd20000;
   3: dout = 23'd10000;
   default: dout = 0;
  endcase
endmodule


//binary encoded decimal conversion look up table
module lut_htho(select, dout);

input      [1:0]      select;
output    [23:0]    dout;
reg         [23:0]    dout;

always @(select)
case (select)
   0: dout = 23'd800000;
   1: dout = 23'd400000;
   2: dout = 23'd200000;
   3: dout = 23'd100000;
   default: dout = 0;
  endcase
```

```
endmodule


//binary encoded decimal conversion look up table
module lut_mil(select, dout);

input    [1:0]    select;
output   [23:0]   dout;
reg      [23:0]   dout;

always @(select)
case (select)
   0: dout = 23'd8000000;
   1: dout = 23'd4000000;
   2: dout = 23'd2000000;
   3: dout = 23'd1000000;
   default: dout = 0;
  endcase
endmodule


module add_24(d1, d2, dout);

input    [23:0]   d1, d2;
output   [23:0]   dout;

assign dout = d1 + d2;

endmodule


//Addition: select = 1
//Subtraction: select = 0
//Performs d1 (+-) d2
module alu_24(clk, d1, d2, select, dout);

input             clk, select;
input    [23:0]   d1, d2;
output   [23:0]   dout;
reg      [23:0]   temp;              //temp used for subtraction

//25 bit mask of 1's
parameter MASK = 24'b1111_1111_1111_1111_1111_1111;

always @(posedge clk)
  begin
    if (select)
      temp = d2;
    else
      temp = (d2 ^ MASK) + 1;        //invert bits of d2 and add one for two's comp
  end

assign dout = d1 + temp;
endmodule
```

```
//Chris Kalima
//A 25-bit addition-subtraction unit [alu_25]
//Addition: select = 1
//Subtraction: select = 0
//Performs d1 (+-) d2
module alu_25(clk, d1, d2, select, dout);

input               clk, select;
input     [24:0]    d1, d2;
output    [24:0]    dout;
reg       [24:0]    temp;             //temp used for subtraction

//25 bit mask of 1's
parameter MASK = 25'b1_1111_1111_1111_1111_1111_1111;

always @(posedge clk)
  begin
    if (select)
      temp = d2;
    else
      temp = (d2 ^ MASK) + 1;         //invert bits of d2 and add one for two's comp
  end

assign dout = d1 + temp;
endmodule


//A specialized 8-bit addition-subtraction unit [alu_8]
//Performs d1 + d2 and outputs a nine-bit number to account for overflow
module alu_8(d1, d2, bias, dout);

input     [7:0]     d1, d2;
input     [6:0]     bias;
output    [8:0]     dout;

assign dout = (d1 + d2) - bias;

endmodule


//A 8-bit enabled register with asyncronous reset
module reg_9(clk, reset, enable, din, dout);

input               clk, reset, enable;
input     [8:0]     din;
output    [8:0]     dout;
reg       [8:0]     dout;

always @(posedge clk or posedge reset)
  begin
    if (reset) dout <= 0;
    else if (enable)
      dout <= din;
  end
endmodule
```

```
//A 50-bit register with asyncronous reset
module reg_50(clk, reset, din, dout);

input               clk, reset;
input     [49:0]    din;
output    [49:0]    dout;
reg       [49:0]    dout;

always @(posedge clk or posedge reset)
  begin
    if (reset) dout <= 0;
    else dout <= din;
  end
endmodule


//A 48-bit enabled register with asyncronous reset
module reg_48(clk, reset, enable, din, dout);

input               clk, reset, enable;
input     [47:0]    din;
output    [47:0]    dout;
reg       [47:0]    dout;

always @(posedge clk or posedge reset)
  begin
    if (reset) dout <= 0;
    else if (enable)
      dout <= din;
  end
endmodule


//A 32-bit enabled register with asyncronous reset
module reg_32(clk, reset, enable, din, dout);

input               clk, reset, enable;
input     [31:0]    din;
output    [31:0]    dout;
reg       [31:0]    dout;

always @(posedge clk or posedge reset)
  begin
    if (reset) dout <= 0;
    else if (enable)
      dout <= din;
  end
endmodule


//A 28-bit enabled register with asyncronous reset
module reg_28(clk, reset, enable, din, dout);

input               clk, reset, enable;
input     [27:0]    din;
```

```
output    [27:0]    dout;
reg       [27:0]    dout;

always @(posedge clk or posedge reset)
  begin
    if (reset) dout <= 0;                 //set to zero on reset
    else if (enable) dout <= din;
  end
endmodule


//A 25-bit enabled register with asyncronous reset
module reg_25_4(clk, reset, enable, din, dout);

input               clk, reset, enable;
input     [4:0]     din;
output    [4:0]     dout;
reg       [4:0]     dout;

always @(posedge clk or posedge reset)
  begin
    if (reset) dout <= 0;
    else if (enable) dout <= din;
  end
endmodule


//A 25-bit enabled register with asyncronous reset
module reg_25(clk, reset, enable, din, dout);

input               clk, reset, enable;
input     [24:0]    din;
output    [24:0]    dout;
reg       [24:0]    dout;

always @(posedge clk or posedge reset)
  begin
    if (reset) dout <= 0;
    else if (enable) dout <= din;
  end
endmodule


//A 24-bit enabled register with asyncronous reset
module reg_24(clk, reset, enable, din, dout);

input               clk, reset, enable;
input     [23:0]    din;
output    [23:0]    dout;
reg       [23:0]    dout;

always @(posedge clk or posedge reset)
  begin
    if (reset) dout <= 0;
    else if (enable) dout <= din;
  end
```

```
endmodule


//A 23-bit enabled register with asyncronous reset
module reg_23(clk, reset, enable, din, dout);

input               clk, reset, enable;
input     [22:0]    din;
output    [22:0]    dout;
reg       [22:0]    dout;

always @(posedge clk or posedge reset)
  begin
    if (reset) dout <= 0;
    else if (enable)
      dout <= din;
  end
endmodule


module ERreg_3b (out, in, clk, enbl, reset) ;

input [2:0] in ;
output [2:0] out ;
input enbl ;
input reset ;
input clk ;

reg [2:0] out;

always @(posedge clk or posedge reset)
  if (reset) out <=0;
  else if (enbl) out <= in;
endmodule


module erreg_8b (out, in, clk, reset, enbl) ;
//8 bit enable register w/ asynchronous reset

input [7:0] in ;
input reset ;
input enbl ;
input clk ;
output [7:0] out ;

reg [7:0] out;

always @(posedge clk or posedge reset)
  if (reset) out <= 0;
  else if (enbl) out <= in;

endmodule


module ERreg_32b (out, in, clk, enbl, reset) ;
```

```
input [31:0] in ;
output [31:0] out ;
input enbl ;
input reset ;
input clk ;

reg [31:0] out;

always @(posedge clk or posedge reset)
  if (reset) out <=0;
  else if (enbl) out <= in;
endmodule


module shift_reg28(clk, reset, en, din, dout);

input             clk, reset, din, en;
output   [27:0]   dout;
reg      [27:0]   dout;
reg               en;

always @(posedge clk or posedge reset)
  if (reset) dout = 0;
  else if (en) dout  = {dout[26:0], din};          //shifts left

endmodule


module flop(clk, din, dout);

input clk, din;
output dout;
reg dout;

always @(posedge clk)
     dout = din;

endmodule


module mux2_1_4(SEL, A, B, OUT);

input             SEL;
input    [4:0]    A, B;
output   [4:0]    OUT;
reg      [4:0]    OUT;

always @(SEL or A or B)
begin
  case (SEL)
     1'b0 : OUT = A;
     1'b1 : OUT = B;
   default : OUT = 0;
  endcase
end
```

```
endmodule


module mux2_1_9b(SEL, A, B, OUT);

input           SEL;
input   [8:0]   A, B;
output  [8:0]   OUT;
reg     [8:0]   OUT;

always @(SEL or A or B)
begin
  case (SEL)
     1'b0 : OUT = A;
     1'b1 : OUT = B;
   default : OUT = 0;
  endcase
end

endmodule


module mux2_1_24b(SEL, A, B, OUT);

input           SEL;
input   [23:0]  A, B;
output  [23:0]  OUT;
reg     [23:0]  OUT;

always @(SEL or A or B)
begin
  case (SEL)
     1'b0 : OUT = A;
     1'b1 : OUT = B;
   default : OUT = 0;
  endcase
end

endmodule


module mux2_1(SEL, A, B, OUT);

input           SEL;
input   [24:0]  A, B;
output  [24:0]  OUT;
reg     [24:0]  OUT;

always @(SEL or A or B)
begin
  case (SEL)
     1'b0 : OUT = A;
     1'b1 : OUT = B;
   default : OUT = 0;
  endcase
end
```

```
endmodule
module mux2_1_32b(SEL, A, B, OUT);

input            SEL;
input   [31:0]   A, B;
output  [31:0]   OUT;
reg     [31:0]   OUT;

always @(SEL or A or B)
begin
  case (SEL)
     1'b0 : OUT = A;
     1'b1 : OUT = B;
   default : OUT = 0;
  endcase
end

endmodule


module mux3_1(SEL, A, B, C, OUT);

input   [1:0]    SEL;
input   [47:0]   A, B, C;
output  [47:0]   OUT;
reg     [47:0]   OUT;

always @(SEL or A or B or C)
begin
  case (SEL)
     2'b00 : OUT = A;
     2'b01 : OUT = B;
     2'b10 : OUT = C;
    default : OUT = 0;
  endcase
end

endmodule


//A 5 to 1 multiplexer
//has a 50-bit width for inputs and outputs
//FAILS AT 50-BIT WIDTH!!!!
module mux5_1(select, a, b, c, d, e, dout);

input   [2:0]    select;
input   [40:0]   a, b, c, d, e;
output  [40:0]   dout;
reg     [40:0]   dout;

always @(select or a or b or c or d or e)
begin
   case(select)
     3'b000 :  dout = a;
     3'b001 :  dout = b;
```

```
      3'b010 :  dout = c;
      3'b011 :  dout = d;
      3'b100 :  dout = e;
     default :  dout = 0;
   endcase
end
endmodule


module mux7_1_24b(SEL, A, B, C, D, E, F, G, MUX_OUT);

input     [2:0]     SEL;
input     [23:0]    A, B, C, D, E, F, G;
output    [23:0]    MUX_OUT;
reg       [23:0]    MUX_OUT;

always @(SEL or A or B or C or D or E or F or G)
begin
   case (SEL)
      3'b000 : MUX_OUT = A;
      3'b001 : MUX_OUT = B;
      3'b010 : MUX_OUT = C;
      3'b011 : MUX_OUT = D;
      3'b100 : MUX_OUT = E;
      3'b101 : MUX_OUT = F;
      3'b110 : MUX_OUT = G;
            default : MUX_OUT = 0;
   endcase
end

endmodule


module mux8_4b (d1, d2, d3, d4, d5, d6, d7, d8, out, sel) ;

input [3:0] d1, d2, d3, d4, d5, d6, d7, d8 ;
output [3:0] out ;
input [2:0] sel ;

reg [3:0] out;

always @(d1 or d2 or d3 or d4 or d5 or d6 or d7 or d8 or sel)
  case (sel)
    0: out <= d1;
    1: out <= d2;
    2: out <= d3;
    3: out <= d4;
    4: out <= d5;
    5: out <= d6;
    6: out <= d7;
    7: out <= d8;
 endcase

endmodule
```

```
module mux2_4b (a, b, out, sel) ;

input [3:0] a ;
input [3:0] b ;
output [3:0] out ;
input sel ;

reg out;

always @(a or b or sel)
  if (sel)
   out <= b;
  else
   out <= a;

endmodule


module mux2_32b (out, a, b, sel) ;

output [31:0] out ;
input [31:0] a ;
input [31:0] b ;
input sel ;

reg [31:0] out;

always @(a or b or sel)
 if (sel)
  out <= b;
 else
  out <= a;

endmodule


module mux3_32b (out, a, b, c, sel) ;

output [31:0] out ;
input [31:0] a, b, c ;
input [1:0] sel ;

reg [31:0] out;

always @(a or b or c or sel)
 case (sel)
  00: out <= a;
  01: out <= b;
  10: out <= c;
  default: out <= 0;
 endcase

endmodule


module mux4_32b (out, a, b, c, d, sel) ;
```

```
output [31:0] out ;
input [31:0] a, b, c, d ;
input [1:0] sel ;

reg [31:0] out;

always @(a or b or c or d or sel)
 case (sel)
  00: out <= a;
  01: out <= b;
  10: out <= c;
  11: out <= d;
 endcase

endmodule
```