CARDIAC: Basys2 Edition

User Manual

Eric W. Mann

5/10/2014

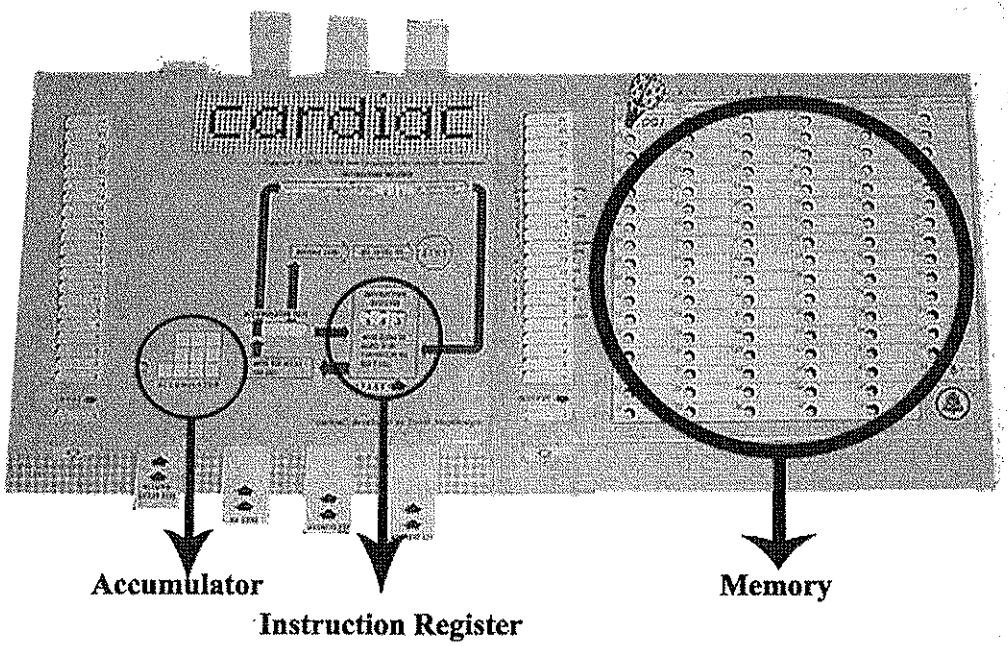
Learn to input, step through, run, and modify programs using the architecture of the CARDIAC computer developed by Bell Telephone Laboratory in 1968 which has been implemented using a Basys2 FPGA board.

Table of Contents:

Introduction to the CARDIAC

The CARDIAC computer was a "toy" computer that was created by Bell Telephone Laboratories in 1968 to aid students in learning how a computer operated. CARDIAC is an acronym derived from the following: **CARD**board **I**llustrative **A**id to **C**omputing. As the name implies, the CARDIAC was a learning aid not meant to be used for calculations or computations.

There are four components to the CARDIAC. The first is the memory which can store up to 100 signed three digit numbers ranging from -999 to +999. The memory holds both data and instructions which are stored as numbers. Next, is the Instruction Register which holds the current instruction being executed. Since a program is a sequence of instructions, the Program Counter is a register that hold the memory location of the next instruction (the user keeps track of the next instruction). The last of the four components, the Accumulator, is a register that holds the results of calculations for later use. Below is a picture of the CARDIAC and its components:



To use the CARDIAC, a user must first enter a program into memory before executing the program. By convention, the first instruction of the program is placed at location "00" followed sequentially by the remaining instructions. A sample program to add two numbers is listed below:

Memory Location	Instruction	Notes:
00	104	Load contents of 04 (A) into the Accumulator
01	205	Add the contents of 05 (B) to the contents of the Accumulator
02	606	Store the contents of the Accumulator to 06 (Sum)
03	900	Halt the program and set the Program Counter to 00
04	002	A
05	003	B
06	000	Sum

As seen in the above program, each instruction has two parts. The first digit is the Operation Code[1] (or op-code for short) which tells the CARDIAC what to do. The two digits of the instruction is a memory address; most instructions need to access the data stored in memory. Below is the list of ten different op-codes and what each of them does:

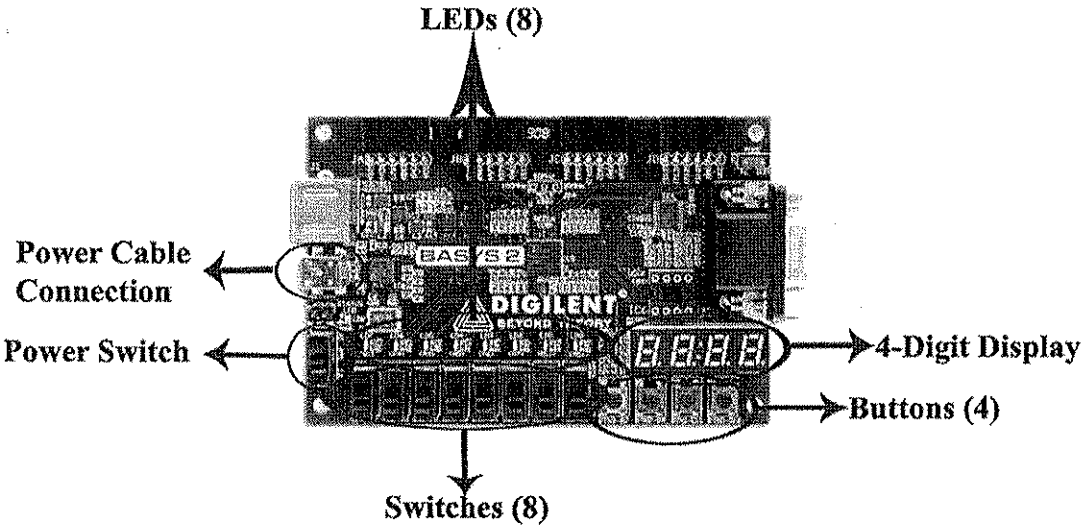
Op-Code	Description
0	Input: Load the address with input from the Basys2 board
1	Load: Load the Accumulator with the contents of the given memory location
2	Add: Add the contents of the given memory location with the Accumulator
3	Jump on Minus: Load the Program Counter with the address if the Accumulator is negative
4	Shift: Shift the Accumulator in the following manner 400 = Does not shift the Accumulator 410 = Shifts the Accumulator left one digit 401 = Shifts the Accumulator right one digit 411 = Shifts the Accumulator left one digit, then right one digit resulting in no shift 4XY (where X and Y are neither 1 nor 0) = Not supported. Results are unpredictable
5	Output: Displays the contents of the given memory location
6	Store: Saves the contents of the Accumulator at the given memory location
7	Subtract: Subtracts the contents of the given memory location from the contents of the Accumulator
8	Jump: Loads the Program Counter with the given memory address
9	Halt: Halts the program and jumps to the given memory address

The first instruction of the simple addition program given above is "104". This is composed of "1", the op-code, and "04", the memory address. The op-code "1" in the instruction tells the CARDIAC to load the contents of memory at location "04" (A, or "002" in the case of the sample program) into the Accumulator. After the first instruction is executed, the next instruction in memory gets executed. In this case, the next instruction is "205". This instruction will add (op-code "2") the contents of memory at location "05" (B, or "003") to the contents of the Accumulator which is currently set to "002" from the previous instruction. After the addition, the Accumulator will contain "005" which is the result of "002 + 003". The program will continue in this manner executing each instruction sequentially until it reaches the halt command (op-code "9") where the calculations will stop and be reset to given memory location.

[1] For more detailed information on the op-codes and their functions, see the technical manual

The Basys2 FGPA Board

The Basys2 board is an FPGA (field programmable gate array) board that can be used to simulate the functions and implements the CARDIAC. The picture below displays the parts of the board that are being utilized for the CARDIAC:



The use of the board is not as straight forward as using a pencil and eraser to put data into memory or even slide cards to do the operations (as was the case for the CARDIAC). The Basys2 board uses switches, LEDs, buttons, and a four digit display for input, output, and operation of the implemented version of the CARDIAC. The numbering scheme used for the switches, buttons, LEDs, and display start at the far right with 0, and going up by one towards the left. This is the numbering scheme (referred to as zero-based indexing) which will be used throughout the rest of the manual and is as follows:

NOTE: The numbering of components is from right to left on the Basys2 board.

- Switches (sw): 7, 6, 5, 4, 3, 2, 1, 0
  - Ex: The second switch from the right will be referred to as sw(1)
- LEDs (ld): 7, 6, 5, 4, 3, 2, 1, 0
  - Ex: The second LED from the left will be referred to as ld(6)
- Buttons (btn): 3, 2, 1, 0
  - Ex: The button to the farthest right will be referred to as btn(0)
- Four Digit Display (dsp): 3, 2, 1, 0
  - Ex: The leftmost digit of the display will be referred to as dsp(3)

Implementing the CARDIAC on the Basys2 Board

The most *challenging* aspect of implementing the CARDIAC on the Basys2 board was working with the limited I/O. In order to circumvent the issue, different modes were added to allow for expanded functionality of the switches, buttons, LEDs, and the small display that is on the board. There are four modes that are used to operate the CARDIAC to its fullest: Mode 0, Mode 1, Mode 2, and Mode 3. The first two switches from the left, sw(7) and sw(6), are used to select the active mode. This information is summarized below:

(NOTE: when a switch is in the off position, it means the switch is facing downwards, away from the LEDs. The switch is on or active when it is flipped up towards the LEDs)

- Mode 0: Clears the Control Unit
  - Active when both sw(7) and sw(6) are off
- Mode 1: Sets the address of the memory
  - Active when sw(7) is off and sw(6) is on
- Mode 2: Sets the contents of the memory
  - Active when sw(7) is on and sw(6) is off
- Mode 3: Execution mode where the program is ran
  - Active when both sw(7) and sw(6) are on

Clear/Reset		
Mode 0	btn(0)	Clears the Instruction Register, Program Counter, and Accumulator
	btn(1)	Not used in this mode
	btn(2)	Clears the control unit
	btn(3)	Not used in this mode
	switches	Not used in this mode
Memory Address		
Mode 1	btn(0)	Sets the lower digit of the address
	btn(1)	Sets the upper digit of the address
	btn(2)	Not used in this mode
	btn(3)	Not used in this mode
	switches	sw(3 – 0) are used as the digit to be entered *NOTE: all digits entered are in binary*
Memory Contents		
Mode 2	btn(0)	Sets the lower digit of memory
	btn(1)	Sets the middle digit of memory
	btn(2)	Sets the upper digit of memory as well as the sign of the number (+ or -)
	btn(3)	Advances the memory to the next address
	switches	sw(3 – 0) are used for the digit to be assigned to memory. sw(4) is used when btn(2) is pressed to set the sign of the number along with the upper digit. *NOTE: all digits entered are in binary*

Execution		
Mode 3	btn(0)	Used for the input to set the lower digit of input
	btn(1)	Used for the input to set the middle digit of input
	btn(2)	Used for the input to set the upper digit of input
	btn(3)	Resumes execution
	switches	sw(5) is used to run the program instead of stepping through it. sw(1 – 0) are used to change what is displayed on the Four Digit Display

As mentioned in the table for Mode 3, the display changes the Four Digit Display depending on sw(1 – 0). When both sw(1) and sw(0) are off, dsp(3 – 2) shows the content of the Instruction Register and dsp (1 – 0) shows the contents of the Program Counter. When sw(1) is off but sw(0) is on, dsp(3 – 2) shows a debug mode and dsp(1 – 0) shows the contents of the Program Counter. The debug mode is used for only for technical purposes. When sw(1) is on and sw(0) is off, the display shows the contents of the Accumulator. Finally, when both sw(1) and sw(0) are on, the display shows the content of memory at the current location. A concise chart of the display options is shown below:

What is Displayed				**Switch settings**	
Dsp(3)	**Dsp(2)**	**Dsp(1)**	**Dsp(0)**	**Sw(1)**	**Sw(0)**
Instruction Register		Program Counter		Off	Off
Debug Display[2]		Program Counter		Off	On
Contents of the Accumulator				On	Off
Current Contents of Memory[3]				On	On

Mode 0 (Clear/Reset) uses two buttons: btn(0) to clear the three registers (Accumulator, Instruction Register, and Program Counter) and btn(2) to reset/initialize the control unit (the hardare device that synchronizes program execution).

Mode 1 (Memory Address) is used to set the address needed to access memory. Recall that to access a memory location (either to read from or write to memory), you must first provide an address. Since memory address are two digits long and given the limited I/O of the Basys2 board, address must be entered one digit at a time. To set the value of the lower (least significant) digit, set sw(3 – 0) to the binary[4] representation of the desired number and press btn(0). The binary value should then appear on the lower four LEDs, ld(3 – 0). To set the value of the upper (most significant) digit, set sw(3 – 0) to the binary representation of the digit and press btn(1). The value should appear on the upper four LEDs, ld(7 – 4).

For example, to set the address to "40", set the sw(3 – 0) to the binary representation of four (off, on, off, off in terms of switches, or 0100), and press btn(1). The LEDs on the board should then change to represent "4" on ld(7 – 4) and zero on ld(3 – 0). To set the lower digit, for example the "2" in "82", follow the same directions as setting the upper digit except hit btn(0) instead of btn(1). The LEDs of ld(3 – 0) should then change and represent the newest number.

[2] Exact display is delineated in the Technical Manual

[3] See Technical Manual for specific details on the output

[4] See page 15 for a brief "How To" on binary

Mode 2 (Memory Contents) is used to set the contents of memory at the current address (set using Mode 1). To set the contents of memory at the current address, a method similar to Mode 1 is used. Since each memory address can hold a signed three digit number, the digits must be entered sequentially using sw(3 – 0) for the value of each digit and btn(0) – btn(2) to insert the respective digits. To enter the least significant digit, set sw(3 – 0) to the desired digit (in binary) and press btn(0). To enter the next digit, set sw(3 – 0) to the binary value of the second digit and press btn(1). Repeat the same process for the most significant digit, using sw(4) to set the sign, and btn(2) to store to memory.

For example, to set a memory location to "948", set sw(3 – 0) to "8" and press btn(0). Next, set sw(3 – 0) to "4" and press btn(1). Finally, set sw(3 – 0) to "9" and press btn(2). To enter the negative number "-948", the same steps will be taken, but when setting the upper digit, set sw(4) to On as well as sw(3 – 0) for the digit and press btn(2). This will set both the upper digit and the sign bit of the three digit number. After the data has been set, it will remain set in memory unless the user changes the memory at that location, or an instruction is executed to change the specific location of memory.

Button 3, btn(3), is not used to enter in a number, but rather it is used to increment to the next memory location. Say for example the memory location "00" is to be set to "104" and the memory location "01" is to be set to "205". First, set 00 to "104" using the above method, then press btn(3) which will increment the memory address to 01 where "205" can be loaded.

Mode 3 (Execute) is for program execution. In Mode 3, all the buttons are used as well as all the switches. All programs should start at "00" (use Mode 0 to clear the Program Counter to "0"). The execution of one instruction is accomplished in three phases (or cycles): Fetch, Increment, and Execute. During the Fetch cycle, the next instruction, whose address is in the Program Counter, is fetched from memory and loaded into the Instruction Register. During the Increment cycle, the contents of the Program Counter is incremented to point to the address of the next instruction (to be used at the next Fetch Cycle). For example, "00" increments to "01", and continues each Increment cycle until "99" where it increments to "00". During the Execute cycle, the contents of the Instruction Register is *decoded*[5] and carried out by the hardware of the Basys2 board. This is the most complicated of the three cycles and may require a memory access to read the contents of memory (a Load instruction) or write to memory (a Store instruction).

The control unit then cycles back to the fetch state unless the op-code is "9", halt, where the control unit does not progress forward and no operations are executed. The halt cycle is effectively a waiting loop that con only be reset to the initial fetch cycle if the control unit is reset by going to Mode 0 and pressing btn(2).

To execute a program, two options are available. The first option is to **step** through each individual cycle and instruction. This is done by pressing btn(3) while in Mode 3. This use of stepping between cycles and instructions is to allow for examining the display at every cycle to see what is being done by the computer. The second option is to **run** the program, allowing the computer to execute each instruction automatically. This automates the execution of the program removing the need for the user to change states manually.

There are two special extended cycles in Mode 3 to handle input/output. These two immediately follow the execute cycle if the op-code is "0" for input or "5" for output. (Note: these special cycles are needed to synchronize the slower human reaction-time needed for input

[5] See Technical Manual to see the decode chart for op-codes.

and output with the much faster rate that a computer executes a program). For input (op-code "0"), the program waits for the human to enter a three digit number using a manner similar to that used in Mode 2 (Memory Contents) to enter a value into memory. Pressing btn(3) will resume the execution of the program. For output (op-code "5"), the program pauses and displays the contents of the specified memory address on the display. Pressing btn(3) will also resume the execution of the program in this special cycle as well.

Running a Program

Understanding the modes may be the hardest part of using the CARDIAC on the Basys2 board. Once there is a firm grasp on each mode and its functions, the next step is to input a program into memory and then execute it. Below is a step by step process of how to enter and run the sample program of adding two numbers together.

Steps:

1) Clear the Control Unit and the Registers
    i. Go to Mode 0 (sw(7) = Off, sw(6) = Off), and press btn(0) and btn(2) to clear the registers and the control unit to insure the program starts at the first memory location, as well as initializes the control unit to the fetch state.

2) Enter Program into Memory
    i. Go to the first memory location.
        1. Use Mode 1 (sw(7) = Off, sw(6) = On) to set the address to 00.
    ii. Enter in the first instruction into memory.
        1. This is done by using Mode 2 (sw(7) = On, sw(6) = Off) and setting the switches to the binary representation of the digit you wish to enter, then press the corresponding button to load it into memory. Repeat this process for each digit until memory location "00" has "104" displayed on the 4-Digit display.
    iii. Press btn(3) to advance to the next memory location
        1. Press btn(3) to increment the memory location by one to "01". From "01", the memory location will progress to "02", then "03", then "04", all the way up to 99, where it will wrap around to "00" if btn(3) is pressed again.
    iv. Repeat Step 2 and Step 3.
        1. Input each instruction into its respective memory address then progress to the next memory location. Repeat this process until the final instruction is saved into memory. In the example case, this would be when memory location "05" contains "003" (since "06" is "000" by default and will be stored to later, it does not need to be set to "000".)

3) Execute the Program
    i. First set the mode to Mode 3 (sw(7) = On, sw(6) = On).
    ii. Next, set the display to show the desired output (the easiest to read is when he display is set to show the Accumulator. This will show how the numbers are loaded and added together and is done by setting sw(1) to On and sw(0) to Off).
    iii. Run or step through the program.
        1. To step through the program, press btn(3) continuously to progress from one state to the next.
        2. To run the program, set sw(5) to On and let the program run on its own until it halts.

4) View Results
    i. First make sure the program has finished running. To tell if the program is done, switch the display to view the contents of memory (sw(7) and sw(6) are turned on). If the display shows the first instruction at location 00, then the program has finished running (in the case of the addition program, the display should show "104"). Then turn off the run switch (sw(5)) to insure the program doesn't run continue to run
    ii. To view the results, go to the location that the sum of A and B was stored (memory location 06). To do this, go to Mode 1 (sw(7) = Off, sw(6) = On) and follow the steps to set the address to "06" (six in binary for the switches would be off, on, on, off, in regards to sw(3 – 0) respectively).
    iii. Once the address has been set to "06", the memory location should no longer show "0", but should instead show "5", the result of adding "002" and "003".

5) Clear the Control Unit and the Registers
    i. Go to Mode 0 (sw(7) = Off, sw(6) = Off) and clear both the control unit and the registers. This is done to allow for another program to be executed and prevents unexpected results when entering in a new program.

6) Celebrate!
    i. Congratulations! Now that the first program has been successfully ran, there are many more programs that can be executed to get different results such as multiplication, division, shifting numbers, a countdown loop; the sky is the limit!

Running a Program Using Input and Output

Running a program that utilizes the input and output instructions are not much different than running any other program. The only difference is how the control unit progresses from state to state as well as how the user interacts with the program. When an input command is encountered, even during run mode, the control unit goes into a special input state. The control unit will remain in this state until the user presses btn(3) showing the computer that the data has been input and the user is ready to continue with calculations. However, this can become tricky while stepping through the program and not using the run mode. When the display switches to showing just "0" the user has entered the input mode. The number should be entered before pressing btn(3) and progressing to the next instruction in the program. It may be hard to tell, but knowing what instruction to expect will make it easier to know when an input mode has been reached.

Output is similar to input except the user does not enter any data into memory. Instead, the output instruction exists solely to allow the user to see a given location in memory that may be a result of some calculation. This can be especially helpful when writing a program to see if the calculations are progressing correctly, or if there is a bug somewhere. Much like input mode, the control unit will stop at this state and waits for the user to press btn(3) to advance to the next state.

Below is an example input/output addition program step by step instructions on how it is run.

Memory Location	Instruction	Notes:
00	007	Input data into 07 (A)
01	008	Input data into 08 (B)
02	107	Load Contents of 07 (A) into the Accumulator
03	208	Add the contents of 08 (B) to the contents of the Accumulator
04	609	Store the contents of the Accumulator to 09 (Sum)
05	509	View the contents of 09 (Sum)
06	900	Halt the program and set the Program Counter to 00
07	000	A
08	000	B
09	000	Sum

Steps:

1) Clear control unit and the registers

2) Enter Program into Memory
   i. Go to the first memory location.
   ii. Enter in the first instruction into memory.
   iii. Press btn(3) to advance to the next memory location
   iv. Repeat Step 2 and Step 3, entering in each instruction of the program into memory.

NOTE: Up to this point, the directions have been nearly the same as if using the addition program. The main differences come in the next step, Execute the Program, where there will be more in depth instructions for input and output.

3) Execute the Program
   i. Set the mode to Mode 3 (sw(7) = On, sw(6) = On).
   ii. Set the display to show the desired output.
   iii. Run or step through the program
      i. While running or stepping through the program, the program will halt at each input/output instruction that is encounter and will wait for the user to advance to the next state.
      ii. For an input instruction:
         1. Insert the desired instruction/ data using sw(4 – 0) as if entering an instruction into memory normally.
         2. Once input has been entered, switch the display switches to the correct setting to show the desired output on the display.
         3. Press btn(3) to advance to the next state and continue on with the execution of the program
      iii. For an output instruction:
         1. View the display and see the contents of memory at the address in the instruction.
         2. Once the display has been viewed, press btn(3) to advance to the next state and continue on with the execution of the program.

NOTE: From this point on, viewing results of a program and clearing the device are the same as other programs.

4) View Results
   i. Make sure the program has finished running.
   ii. View results of the program at the given memory locations within the program. In the case of the example program for input and output, go to locations "07 – 09" to view the input data, and the resulting sum.

5) Clear the Control Unit and the Registers

Appendix

Sample Programs:

Below are some examples of sample programs that can be loaded and ran using the CARDIAC implemented on the Basys2 board. The programs are as follows and can be manipulated or combined together to make more advanced or powerful programs: multiplication, division, shifting a digit, and countdown summation program.

Multiplication Program:

Memory Location	Instruction	Notes:
00	112	Load 13 (Product) into the Accumulator
01	213	Add 13 (A) to the Accumulator
02	612	Store contents of Accumulator to 12 (Product)
03	114	Load 14 (B) into the Accumulator
04	715	Subtract 15 (One) from the Accumulator
05	614	Store contents of Accumulator to 14 (B)
06	308	Jump to 08 if Accumulator is negative
07	800	Repeat addition loop (Jump to 00)
08	112	Load 13 (Product) into the Accumulator
09	713	Subtract 13 (A) from Accumulator (Correct for over multiplying)
10	612	Store Product at 13
11	900	Halt
12	000	Product
13	004	A
14	003	B
15	001	One

Division Program:

Memory Location	Instruction	Notes:
00	114	Load 14 (Quotient) into the Accumulator
01	217	Add 17 (One) to the Accumulator
02	614	Store contents of Accumulator to 14 (Quotient)
03	115	Load 15 (Dividend) into the Accumulator
04	716	Subtract 16 (Divisor) from the Accumulator
05	615	Store contents of Accumulator to 15 (Dividend)
06	308	Jump to 08 if Accumulator is negative
07	800	Repeat subtraction loop (Jump to 00)
08	216	Add 16 (Divisor) to the Accumulator to make it positive
09	615	Store contents of Accumulator to 15 (Dividend)
10	114	Load 14 (Quotient) into the Accumulator
11	717	Subtract 17 (One) from the Accumulator (Correct the quotient)
12	614	Store contents of Accumulator to 14 (Quotient)
13	900	Halt
14	000	Quotient
15	035	Dividend (Remainder of division after program finishes)
16	006	Divisor
17	001	One

Shifting Program (Shift Left):

Memory Location	Instruction	Notes:
00	104	Load contents of 04 (Un-Shifted) into the Accumulator
01	410	Shift the contents of the Accumulator left one digit
02	605	Store the contents of the Accumulator to 05 (Shifted)
03	900	Halt the program and set the Program Counter to 00
04	123	Un-Shifted
05	000	Shifted

Shifting Program (Shift Right):

Memory Location	Instruction	Notes:
00	104	Load contents of 04 (Un-Shifted) into the Accumulator
01	401	Shift the contents of the Accumulator right one digit
02	605	Store the contents of the Accumulator to 05 (Shifted)
03	900	Halt the program and set the Program Counter to 00
04	123	Un-Shifted
05	000	Shifted

Countdown Summation Loop (adds N + (N – 1) + (N – 2) + … + 0):

Memory Location	Instruction	Notes:
00	109	Load contents of 09 (Sum) into the Accumulator
01	210	Add 10 (I) to the Accumulator
02	609	Store the contents of the Accumulator to 09 (Sum)
03	110	Load contents of 10 (I) into the Accumulator
04	711	Subtract 11 (One) from the Accumulator
05	610	Store the contents of the Accumulator to 10 (I)
06	308	Jump to 08 if Accumulator is negative
07	800	Repeat the Countdown Loop
08	900	Halt the program and set the Program Counter to 00
09	000	Sum
10	005	I
11	001	One

More complicated programs can be used that combine some or all of the op-codes into a single program to perform complex operations. Below is an example of such a program that takes input from the switches on the Basys2 board, calculates the square of the given number that was input, displays the result of the squaring process on the display, then terminates. The special quality of this program is that it not only utilizes all but one of the op-codes, but can be run over and over again without ever having to change memory. The data that is given as input will overwrite any previous data stored in the same location in memory allowing for continuous runs of the program using only Mode 3 to execute, and Mode 0 to clear the control.

Number Squaring Program (Squares a number given as input):

Memory Location	Instruction	Notes:
00	016	Receives Input from external device
01	116	Load contents of 16 (Number) into the Accumulator
02	216	Add 16 (Number) to the Accumulator
03	717	Subtract 17 (One) from the Accumulator
04	616	Store the contents of the Accumulator to 16 (Number)
05	615	Store the contents of the Accumulator to 15 (Square)
06	718	Subtract 18 (Two) from the Accumulator
07	313	Jump to 13 if the Accumulator is negative
08	616	Store the contents of the Accumulator to 16 (Number)
09	215	Add 15 (Square) to the Accumulator
10	615	Store the contents of the Accumulator to 15 (Square)
11	116	Load contents of 15 (Number) into the Accumulator
12	806	Repeat the Squaring Loop
13	515	Output contents of 15 (Square) to the display
14	900	Halt the program and set the Program Counter to 00
15	000	Square
16	000	Number
17	001	One
18	002	Two

How to Read Binary Numbers:

Binary is used in Computer Science as a way to represent numbers in the Base 2 format. What that means, it that each digit has only two options, a 0 or a 1. Think about the normal counting system. It is called Base 10. That means there are 10 possible options for each digit in a number: 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9. To represent the number 1, for example, the first digit of the number is simply 1. Done! If the number 14 was desired, however, then simply put 1 for the first digit, followed by a 4. How is this possible? Well, take the number 14 again. The farthest digit to the right is the ones place, and the next digit to the left is the tens place. How this terminology came to be is from simple multiplication, addition, and exponents. The farthest digit to the right in Base 10 is 10^0, or 1. The next digit is 10^1, or 10, then 10^2, or 100, and so on and so forth. We then take each digit, multiply it by the power of 10 that it is associated with and add each multiplication step together:

$$14 : 1(10^1) + 4(10^0) = 1(10) + 4(1) = 10 + 4 = 14$$
$$143 : 1(10^2) + 4(10^1) + 3(10^0) = 1(100) + 4(10) + 3(1) = 100 + 40 + 3 = 143$$

The same system to calculate a number in Base 10, is used to calculate a number in Base 2, binary. One main difference between Base 10 and binary is the base that will be multiplied to each digit. Base 10 had 10 raised to the N power (10^N) where N is the digit starting at 0. Binary uses its base of 2 raised to the N power (2^N) starting at 0. This means that the bases for binary as are follows:

$$\text{Bases: } 2^0, 2^1, 2^2, 2^3, 2^4, \ldots, 2^N$$
$$\text{Values: } 1, 2, 4, 8, 16, 32, 64 \ldots$$

Knowing how the bases work and the values associated with them (by calculating the value of the base to the N power), any binary string can then be converted into a more familiar format of Base 10. Below are examples of each binary string from 0 – 9 and how they are calculated to be displayed in Base 10.

$$0 : 0000 = 0(2^3) + 0(2^2) + 0(2^1) + 0(2^0) = 0(8) + 0(4) + 0(2) + 0(1) = 0 + 0 + 0 + 0 = 0$$
$$1 : 0001 = 0(2^3) + 0(2^2) + 0(2^1) + 1(2^0) = 0(8) + 0(4) + 0(2) + 1(1) = 0 + 0 + 0 + 1 = 1$$
$$2 : 0010 = 0(2^3) + 0(2^2) + 1(2^1) + 0(2^0) = 0(8) + 0(4) + 1(2) + 0(1) = 0 + 0 + 2 + 0 = 2$$
$$3 : 0011 = 0(2^3) + 0(2^2) + 1(2^1) + 1(2^0) = 0(8) + 0(4) + 1(2) + 1(1) = 0 + 0 + 2 + 1 = 3$$
$$4 : 0100 = 0(2^3) + 1(2^2) + 0(2^1) + 0(2^0) = 0(8) + 1(4) + 0(2) + 0(1) = 0 + 4 + 0 + 0 = 4$$
$$5 : 0101 = 0(2^3) + 1(2^2) + 0(2^1) + 1(2^0) = 0(8) + 1(4) + 0(2) + 1(1) = 0 + 4 + 0 + 1 = 5$$
$$6 : 0110 = 0(2^3) + 1(2^2) + 1(2^1) + 0(2^0) = 0(8) + 1(4) + 1(2) + 0(1) = 0 + 4 + 2 + 0 = 6$$
$$7 : 0111 = 0(2^3) + 1(2^2) + 1(2^1) + 1(2^0) = 0(8) + 1(4) + 1(2) + 1(1) = 0 + 4 + 2 + 1 = 7$$
$$8 : 1000 = 1(2^3) + 0(2^2) + 0(2^1) + 0(2^0) = 1(8) + 0(4) + 0(2) + 0(1) = 8 + 0 + 0 + 0 = 8$$
$$9 : 1001 = 1(2^3) + 0(2^2) + 0(2^1) + 1(2^0) = 1(8) + 0(4) + 0(2) + 1(1) = 8 + 0 + 0 + 1 = 9$$

Binary is as simple as that! Start with a binary string at the left, apply the powers of 2 starting with 2^0 to each digit, multiply each digit by the power of 2, and then add it all together to get a string in Base 10.

CARDIAC: Basys2 Edition

Technical Manual

Eric W. Mann

5/10/2014

Understand the underlying architecture of the CARDIAC that was implemented on the Basys2 board ranging from the code used (in VHDL) to the diagrams that describe the organization of each individual components utilized. Also, see the technical decisions that went into to creating a successful implementation of the CARDIAC.

Table of Contents:

CARDIAC (Overview of Design)

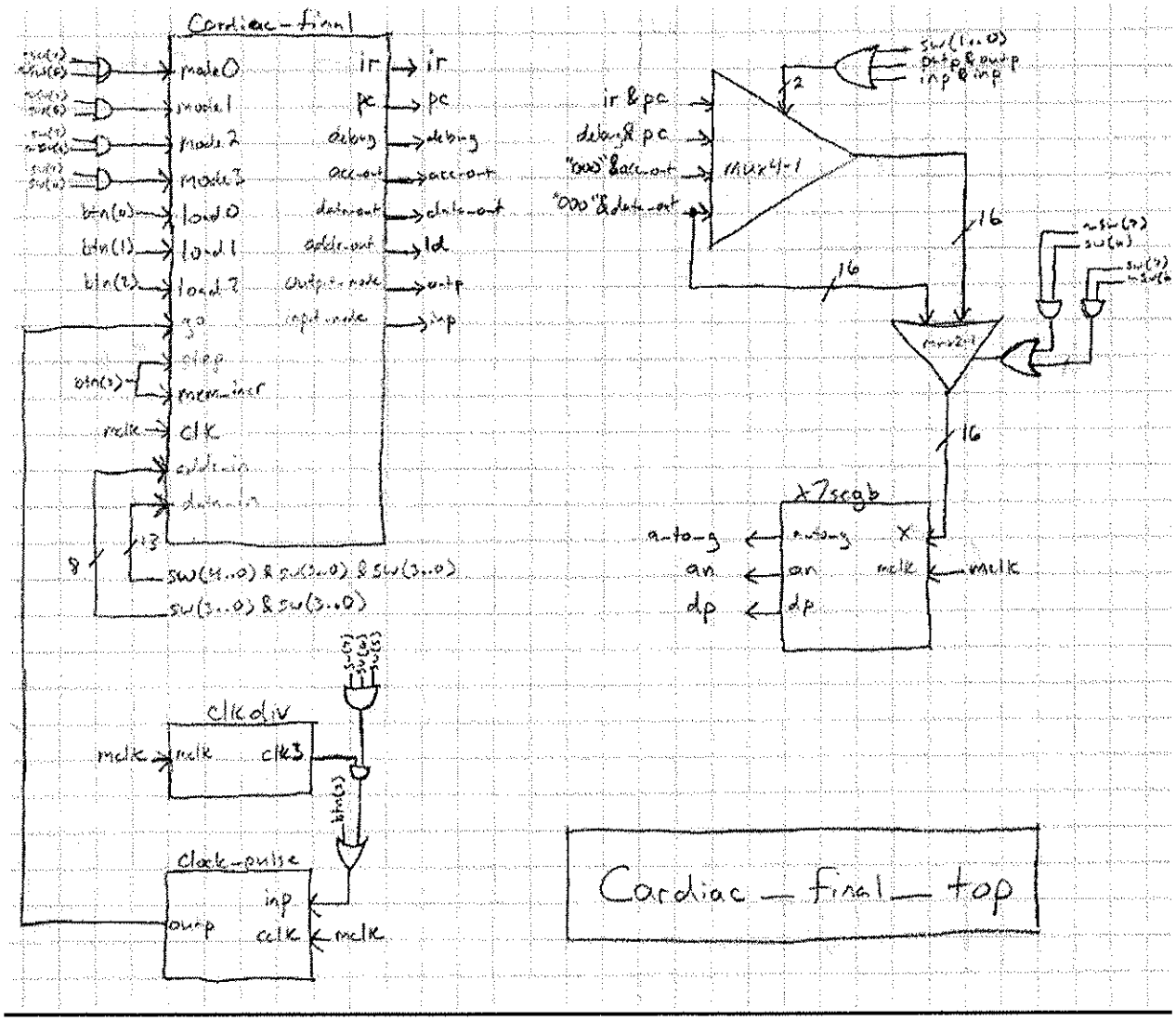
The CARDIAC was implemented on the Basys2 FPGA board. The process was not without many complications along the way including the interface, the clock speed, as well as simply connecting components together.

This manual will demonstrate each of the CARDIAC's components and the programming behind each. A diagram will be provided to show how everything interfaces. At the end of the manual will be a glossary of all the source code utilized in the project. A brief description including the challanges faced, the design desicions made, and a general overview of operations will preface each component to give a concise understanding of what each component does, and why it was done the way it was.

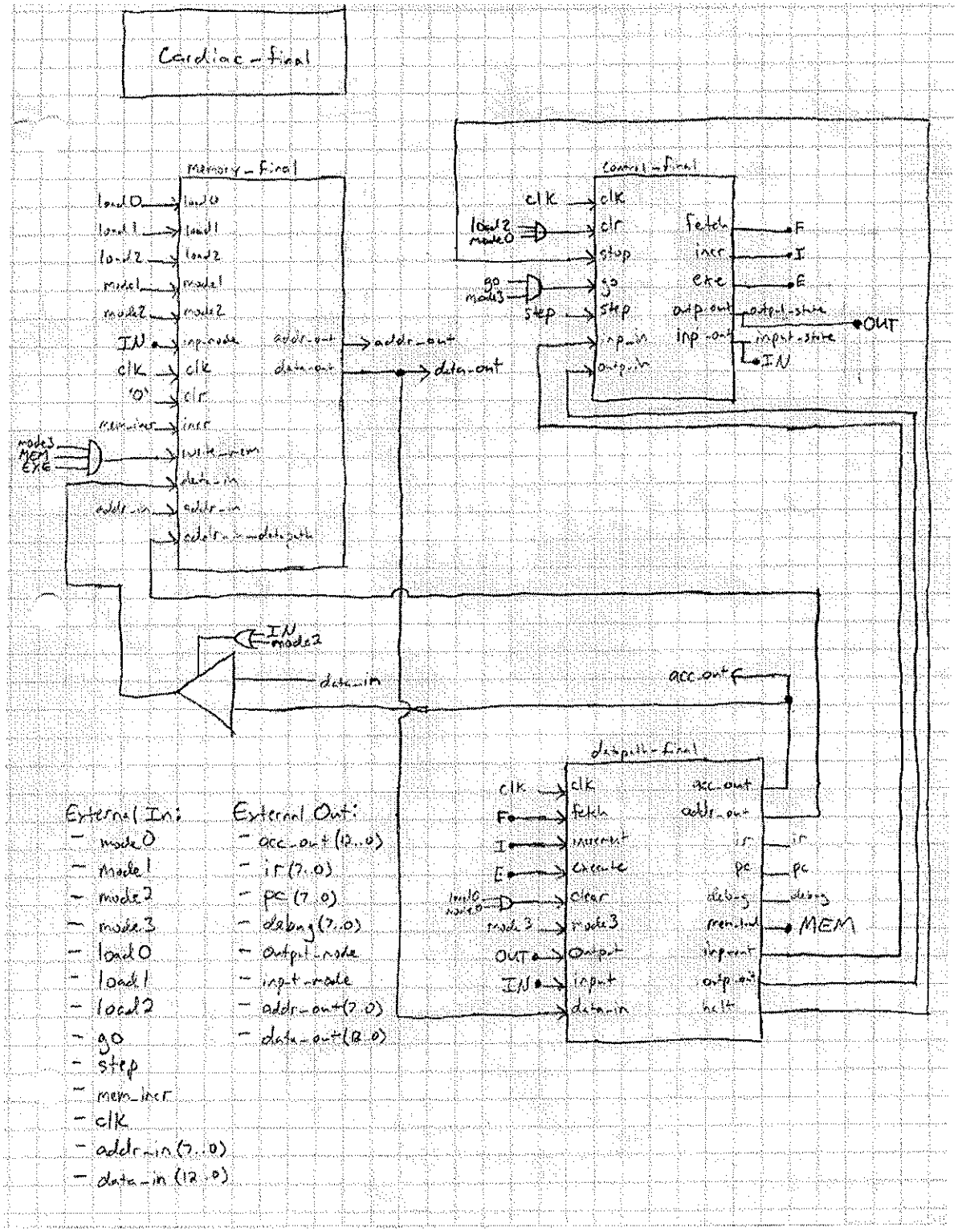
Below is the top diagram for the CARDIAC with how it interfaces with the Basys2 board as well as the CARDIAC diagram of how each component in it is connected in a general manner. Each subsequent section will dive deeper into the implementation.

Enjoy!

Top Diagram


Cardiac_final
mode0
mode1
mode2
mode3
load0
load1
load2
go
step
mem_incr
clk
ir
pc
debug
acc_out
data_out
addr_out
output_mode
input_mode
ir
pc
debug
acc_out
data_out
ld
outp
inp
8
sw(11..0) & sw(3..0) & sw(3..0)
sw(3..0) & sw(3..0)
ir & pc
debug & pc
"000" & acc_out
"000" & data_out
mux4:1
2
16
16
16
mux2:1
x7segb
a_to_g
an
dp
mclk
clkdiv
clk3
clock_pulse
inp
oupp
clk
Cardiac_final_top

Diagram


Cardiac_final
memory_final
load0
load1
load2
mode1
mode2
IN
clk
'0'
mem_incr
mode3
MEM
EXE
inp_mode
addr_out
data_out
addr_in
data_in
control_final
clk
clr
stop
go
step
inp_in
outp_in
fetch
incr
exe
outp_out
inp_out
outpt_state
inpt_state
OUT
IN
datapath_final
F
I
E
acc_out
ir
pc
debug
halt
External In:
- mode0
- mode1
- mode2
- mode3
- load0
- load1
- load2
- go
- step
- mem_incr
- clk
- addr_in(7..0)
- data_in(12..0)
External Out:
- acc_out(12..0)
- ir(7..0)
- pc(7..0)
- debug(7..0)
- outpt_mode
- inpt_mode
- addr_out(7..0)
- data_out(12..0)

Memory

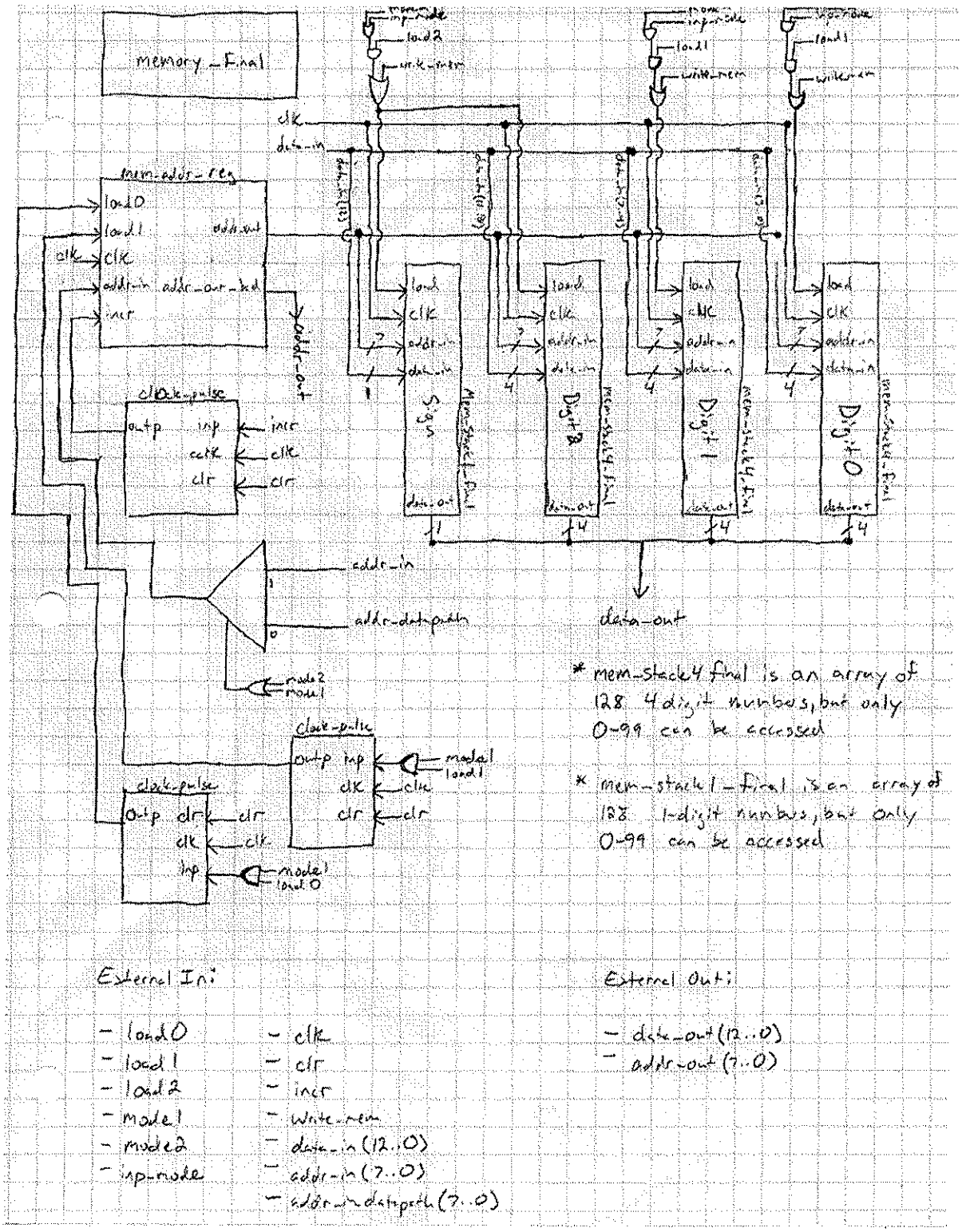
The memory is the first component of the CARDIAC implemented. It was completed first because of the simplicity to it, as well as to make certain a program could be stored and implemented later on. This allowed for data to be put into memory and tested throughout the entire process. When implementing the memory, there needed to be an interface with the board to actually input data into memory. The early stages of the interface were born and became a prototype to what would eventually become the final product.

The memory itself consists of a few internal components: four stacks for memory (one for each digit of a three digit number, and a final stack to store the sign), a memory address register, and clock pulses (which are used to synchronize button presses on the Basys2 board with the internal clock). The four memory stacks are arrays of arrays in the simplest sense. Each stack for the digit is an array of size 128, that can store 4 digit binary number, or a one digit BCD number. There are 128 slots to allow for easier compilation (using a power of 2). The stacks are limited with internal programming to only allow for slots 0 – 99 to be utilized, however, to simulate the memory of the CARDIAC. The memory address register holds the current memory address and outputs it to the memory stacks. This is used during the input state or the input modes (Modes 1 and 2) to insure the memory stays fixed to one location, advancing only when the user wants it to. The final items utilized are the clock pulses. These small structures simply insure the when a button is pressed on the Basys2 board, the pulse that results from it is clean (does not bounce at the beginning) and results in a single press. Without these clock pulses implemented, pressing the load buttons would continually load data into memory instead of loading data once. Also, the increment button for the memory address register would not be one solid clean increment to the next memory address, but rather, it would continually increment until the button was released.

These two issues were the first challenges faced when writing the memory. The second challenge came when connecting the memory circuit to the datapath and similarly the control unit. The datapath outputs an address to memory which is used during the operation of the CARDIAC. The challenge was to take another address as input, but correctly choose the address to utilize (the address from the datapath, or the address from the memory address register). This was solved by using a 2-1 mux to select the correct address depending on whether or not the computer is set to state Mode 1 or 2. Now, the only time the address from the datapath is used is during Mode 0 and Mode 3 where execution occurs. Interfacing with the control unit was not as difficult as predicted. The challenge was to allow data to be loaded into data while in Mode 3 but only if the control unit is in an input state. The solution was to or the memory load pins with a inp_mode signal that was output from the control unit. This allowed for data to still be written to memory while in Mode 3 with the correct address coming from the datapath.

At this point, the Memory circuit is complete for operation but can still use modifications. Firstly, there needs to be a "clear" operation that allows for the memory address register and the contents of memory be cleared so a new program to be input. The difficulty with this comes from the device. The many methods to implement clear that I have tried have yielded results that utilize more resources than the Basys2 board can allocate. In the end, this was cut from the design but can be implemented at a later point in time.

Diagram


memory_Final
clk
data_in
mem_addr_reg
load0
load1
clk
addr_in
addr_out_bus
incr
addr_out
clock_pulse
outp
inp
incr
clk
clr
Sign
Digit 2
Digit 1
Digit 0
mem_stack4_final
mem_stack1_final
addr_in
addr_datapath
data_out
mode2
mode1
load2
load1
load0
write_mem
inp_mode
* mem_stack4_final is an array of 128 4 digit numbers, but only 0-99 can be accessed
* mem_stack1_final is an array of 128 1-digit numbers, but only 0-99 can be accessed
External In:
- load0
- load1
- load2
- mode1
- mode2
- inp_mode
- clk
- clr
- incr
- write_mem
- data_in(12..0)
- addr_in(7..0)
- addr_in_datapath(7..0)
External Out:
- data_out(12..0)
- addr_out(7..0)

Datapath

The datapath is the component where all calculations are gated and performed. It involved the most difficult aspects of implementing the CARDIAC, but, rightfully so, it is the most important (arguably). In order to have the correct operations performed, an interface inside the datapath had to be created in order to have the correct data be gated to the correct component to perform the correct operation. It was as confusing as it sounds. The internal interface was accomplished using a decode circuit to convert an operation code and memory address in an instruction into workable signals.

Each component in the CARDIAC has specialized units to perform given operations, but there are some units (such as clock pulse) are used in some, or all of the main components of the CARDIAC. Among the components used for the datapath are the following: addition, subtraction, shifting, the Accumulator, the Instruction Register, the Program Counter, a decode circuit, and finally clock pulses. The addition circuit simply adds the content of the Accumulator to the contents of memory at the location of memory in the Instruction Register (which holds the current instruction). The current memory location will be referred to as the effective address (EA). Subtraction is much like addition, only subtracts the contents of the EA from the Accumulator. Shifting will shift the contents of the Accumulator to the right one, or to the left one depending on the EA (shifting is explained further below). The Program Counter holds the location of the next location of memory, while the Accumulator is simply storage for operations (almost as a running tally). The clock pulses used in this component are pulses that synchronize a load pin as well as the clock so the registers (instruction, program, and accumulator registers) only load once and cleanly.

The main difficulties encountered were with the clock, registers, and load pins. The complicated structure that makes up the load pin for the Program Counter was constructed to account for jumping, incrementing, and halting (jump on halt). Until the correct load pin was constructed, the program counter only incremented but did nothing else. Also, the specialized clock pulse 2 was used not used here because it was causing a delay in the load of the Program Counter. It would attempt to load after the load was asserted and as such, nothing would be put into it. Another challenge was to have the instruction be decoded in a manner that the data from memory would be gated to the correct location. The decode circuit (delineated below as well as each operation code in detail) kept changing until the final result was constructed. Once the challenges were rectified, the datapath functioned perfectly.

Operation Codes (op-codes)

Operation Codes (op-codes for short) are the commands in each instruction that are decoded to instruct the CARDIAC which when to load or store memory, if special states in the control unit need to be entered, what arithmetic to do, etc. The instructions are decoded by the decode (decode_final) circuit implemented in the datapath (datapath_final). Below are the detailed representations of the functions for each op-code:

Op_codes:

0 → Input
Load data from external switches and store into memory. This op-code also puts the control unit into a special state that halts execution temporarily allowing the user to enter data into memory.

1 → Clear and Add (Load): ACC ← 0, ACC ← ACC + M[IR]
Clears the Accumulator then adds the content from memory to the Accumulator. As it is implemented, the Basys2 implementation simply loads the data into memory instead of clearing and adding. This is done to allow for less calculations as well as allowing for more efficient implantation of a four to one mux utilized in the design.

2 → Add: ACC ← ACC + M[IR]
Adds the content of memory at the address in the instruction to the Accumulator

3 → Jump on Minus: If ACC > 0, PC ← IR
Jumps to the location of memory in the instruction if the Accumulator is negative

4 → Shift
Shifts the Accumulator one digit to the left or right. The CARDIAC allows for the shifting X to the left and then Y to the right. Due to difficulty in implanting multiple implementations, only a single shift is allowed. Putting an instruction other than "00", "01", or "10" will yield unpredicted results.

The following are the delineations for shifting using the rest of the instruction:
"00" → No shifting, "01" → Shift right, "10" → Shift left, "11" → No shifting

5 → Output
Displays the content of memory at the given memory address in the instruction. This op-code also puts the control unit into a special state that halts execution temporarily for the data to be displayed.

6 → Store: M[IR] ← ACC
Stores the content of the Accumulator to the specified memory address in the instruction. The Accumulator is not cleared after a store command.

7 → **Subtract:** ACC ← ACC – M[IR]
Subtracts the content of memory at the address in the instruction from the Accumulator.

8 → **Jump:** PC ← IR
Unconditionally jumps to the location of memory in the instruction.

9 → **Halt:** PC ← IR
The halt command stops execution of a program. Also, the halt command performs a jump to the location of memory in the instruction.

Decoding the Operation Codes (op-codes)

The previous section discussed the op-codes in larger detail on their functions as well as descriptions. Each op-code has its own function, but can only operate when it is decoded by the CARDIAC to perform specific operations and guide the flow of data from registers and memory to their correct destinations in order to perform the operations in the instruction. In order to do so, the decoded instruction results in a sequence of nine outputs that range from a select pin for a mux, to the load pin of the Accumulator. Below is the breakdown of how each instruction is decoded as well as the description for each output from the decode circuit does.

Note: 1s and 0s are commonly used to represent On and Off respectively.

		Outputs									
		outp	**inp**	**jump**	**halt**	**neg**	**acc_load**	**mem_load**	**m1(1)**	**m1(0)**	**m0**
Op-codes	**0**	0	1	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	1	0	1	0	1
	2	0	0	0	0	0	1	0	0	1	1
	3	0	0	0	0	1	0	0	0	0	0
	4	0	0	0	0	0	1	0	0	0	0
	5	1	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	0	1	0	0	1
	7	0	0	0	0	0	1	0	1	1	1
	8	0	0	1	0	0	0	0	0	0	0
	9	0	0	0	1	0	0	0	0	0	0

outp: sends the control unit into a special output state which halts execution to allow for the user to see the current contents of memory at a given location.

inp: sends the control unit into a special input state which halts execution to allow for the user to load data into memory.

jump: allows for the Program Counter to be loaded in the Execute state to allow for an unconditional jump command to be executed.

halt: sends the control unit into the halt state which stops all execution as well as allows the Program Counter to be loaded with the given memory address to perform a jump command.

neg: used to check whether the sign of the Accumulator is 1, negative, or 0, positive. On negative, the Program Counter will be loaded with a given memory address to jump to.

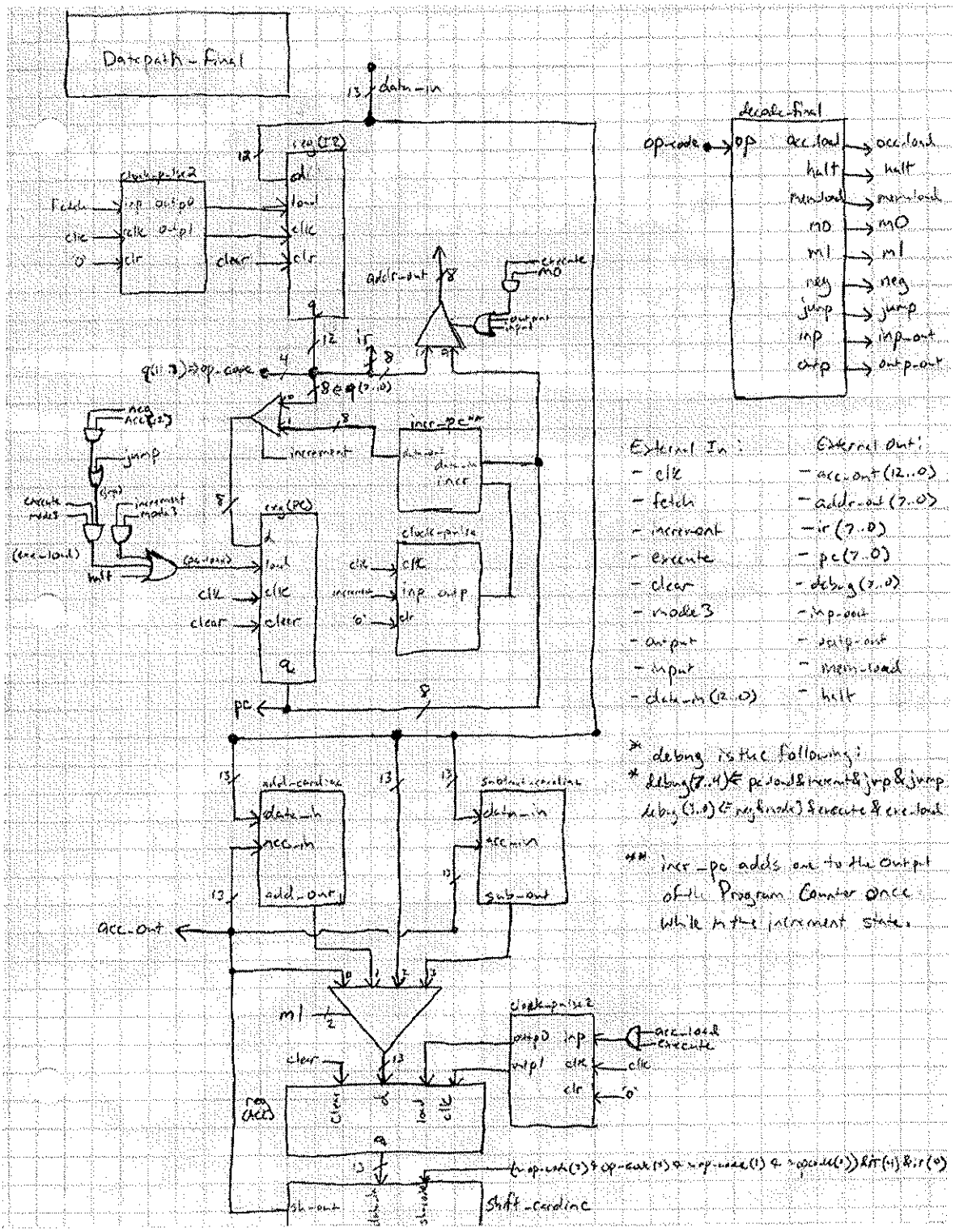
acc_load: allows for the Accumulator to be loaded.

mem_load: allows for memory to be written too during the Execution state.

m1: a two bit select for the Accumulator which chooses to either shift the Accumulator, perform an addition, perform a subtraction, or load the Accumulator. The mux select ranges from "00" to "11" in binary to perform the previous operations respectively.

m0: the single bit select to send either the contents of the Program Counter (on 0) or the Instruction Register (on 1) out to be the address in for memory, from the datapath.

Diagram:


Datapath - Final
data_in
decode_final
op_code
op
acc_load
halt
mem_load
m0
m1
neg
jump
inp
outp
acc_out
addr_out
ir
pc
debug
inp_out
outp_out
mem_load
halt
External In:
- clk
- fetch
- increment
- execute
- clear
- mode3
- output
- input
- data_in (12..0)
External out:
- acc_out (12..0)
- addr_out (7..0)
- ir (7..0)
- pc (7..0)
- debug (7..0)
- inp_out
- outp_out
- mem_load
- halt
* debug is the following:
* debug(7..4) ∈ pc_load & increment & jmp & jump
debug(3..0) ∈ neg & mode3 & execute & exe_load
** incr_pc adds one to the output of the Program Counter once while in the increment state.
acc_out
Shift_Cardinc

Control Unit

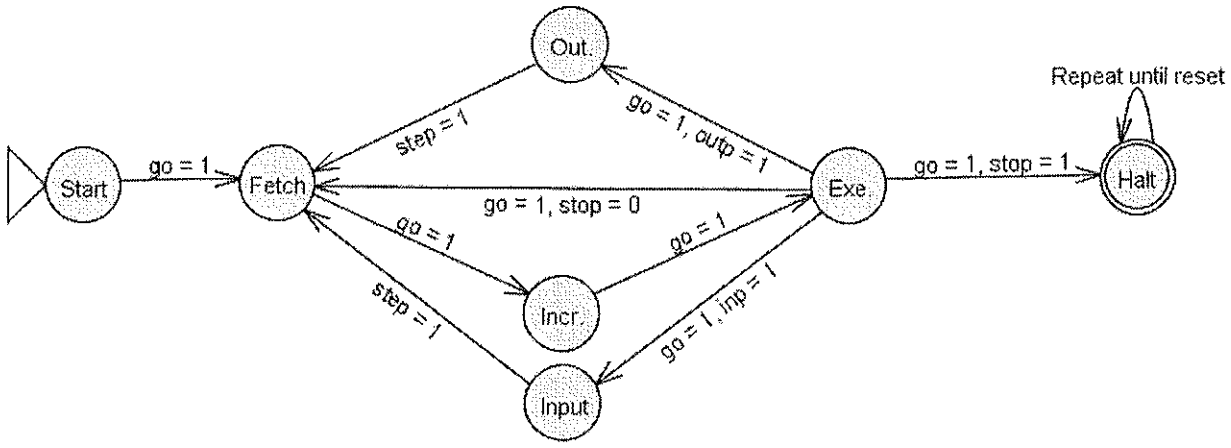
The control unit is the engine behind the machine. It moves between different cycles (Fetch, Increment, Execute) until a halt pin from the decode circuit in the datapath is asserted, where the control unit halts and does nothing until the user restarts the machine. This component is highly important. Without this component, the computer could not run.

Each cycle in the control unit has specific functions. The Fetch cycle grabs data from memory and stores it in the Instruction Register. The Increment cycle moves the Program Counter up by one so the next instruction could be fetched. Finally, the Execute cycle is where all the calculations are performed. The control unit then cycles back to Fetch after Execute and repeats until the end of the program is reached. Special states are included for Input and Output to allow execution to halt, and allow for the user to either input data into memory or view a given location in memory.

There were very view challenges faced with the control unit. It was simple enough to transition between states. The control unit outputs pins for each state which are then utilized by the other main components to tell what functions need to be carried out. This is not the job of the control unit. So, simply enough, the control unit gives out the information needed, and it is up to the other components to utilize it correctly.

Below is the diagram for the control unit:

Diagram:



Input / Output

Input and Output were the last feature to be implemented for the CARDIAC. It was thought to be the most difficult (which is why it was put off until the end) but turned out be one of the simplest. In a general sense, the purpose of Input and Output is to allow for a user to input data into memory while a program is executing or to read a memory location while a program is executing. These are used to allow for greater usability within programs as well as to debug program by seeing the contents of memory locations to see if the correct result is being saved at any specific moment in a program. They are also used to alleviate the need to pause execution to change memory or view memory.

There is no overall diagram or specific code for Input and Output. These functions were implemented by making small changes to each of the main components of the CARDIAC. In the Memory, pins were added to allow for data to be input into memory outside of Mode 1 or 2 (specifically Mode 3: execution). The datapath was where most of the modifications came. The decode circuit had to be altered to allow for the Input and Output operations to be decoded. Also, the datapath had to output of some of the decoded results to the control unit to allow for specific states to be entered. The control unit then outputs different pins when the Input or Output cycle is reached. The new pins tell the datapath if a temporary jump to view memory is needed, as well as to tell the Memory circuit to load data into that given location in the instruction. Once these fixes were implemented, the Input and Output was finished.

Source Code

add_cardiac:

```
-- Adds the Accumulator and datapath --
-- add_cardiac.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity add_cardiac is
     port (
     acc_in : in std_logic_vector(12 downto 0);
     data_in : in std_logic_vector(12 downto 0);
     add_out : out std_logic_vector(12 downto 0)
     );
end add_cardiac;

architecture add_cardiac of add_cardiac is
component tens_compliment13 is
     port (
     a : in std_logic_vector(12 downto 0);
     s : out std_logic_vector(12 downto 0)
     );
end component;

component bcd_adder34 is
     port (
     a : in std_logic_vector(12 downto 0);
     b : in std_logic_vector(12 downto 0);
     s : out std_logic_vector(13 downto 0)
     );
end component;

signal a, b, acc_comp, data_comp, result_comp:
std_logic_vector(12 downto 0);
signal result: std_logic_vector(13 downto 0);
begin

     process (a, b, acc_in, data_in, acc_comp, data_comp)
     begin
          if (acc_in(12) = '1') then
               a <= acc_comp;
          else
               a <= acc_in;
          end if;
          if (data_in(12) = '1') then
```

```
            b <= data_comp;
        else
            b <= data_in;
        end if;
    end process;

    process (result, result_comp)
    begin
        if (result(13 downto 12) = "00") then
            add_out <= result(12 downto 0);
        elsif (result(13 downto 12) = "01") then
            add_out <= result_comp;
        elsif (result(13 downto 12) = "10") then
            add_out <= result(12 downto 0);
        else
            add_out <= result_comp;
        end if;
    end process;

    T1: tens_compliment13
    port map (
    a => acc_in,
    s => acc_comp
    );

    T2: tens_compliment13
    port map (
    a => data_in,
    s => data_comp
    );

    T3: tens_compliment13
    port map (
    a => result(12 downto 0),
    s => result_comp
    );

    B1: bcd_adder34
    port map(
    a => a,
    b => b,
    s => result
    );

end add_cardiac;
```

bcd_adder4:

```
-- 4 digit bcd addition --
-- bcd_adder4.vhd --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity bcd_adder4 is
     port (
     a : in std_logic_vector(3 downto 0);
     b : in std_logic_vector(3 downto 0);
     c_in : in std_logic;
     c_out : out std_logic;
     s : out std_logic_vector(3 downto 0)
     );
end bcd_adder4;

architecture bcd_adder4 of bcd_adder4 is
begin
     process(a, b, c_in)
     variable temp: std_logic_vector(4 downto 0);
     begin
          temp := ('0' & a) + ('0' & b) + ('0' & c_in);
          if (temp > 9) then
               temp := temp + 6;
          end if;
          s <= temp(3 downto 0);
          c_out <= temp(4);
     end process;
end bcd_adder4;
```

bcd_adder34:

```
-- Adds two signed 3 digit BCD numbers --
-- bcd_adder34.vhd --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity bcd_adder34 is
     port (
     a : in std_logic_vector(12 downto 0);
     b : in std_logic_vector(12 downto 0);
     s : out std_logic_vector(13 downto 0)
     );
end bcd_adder34;

architecture bcd_adder34 of bcd_adder34 is
component bcd_adder4 is
     port (
     a : in std_logic_vector(3 downto 0);
     b : in std_logic_vector(3 downto 0);
     c_in : in std_logic;
     c_out : out std_logic;
     s : out std_logic_vector(3 downto 0)
     );
end component;

signal c0_out, c1_out, c2_out: std_logic;
signal s0, s1, s2: std_logic_vector(3 downto 0);
signal carry, a_sign, b_sign, cf: std_logic_vector(1 downto 0);
begin
     a_sign <= "0" & a(12);
     b_sign <= "0" & b(12);
     cf <= "0" & c2_out;
     carry <= a_sign + b_sign + cf;
     s <= carry & s2 & s1 & s0;


     B0: bcd_adder4
     port map (
     a => a(3 downto 0),
     b => b(3 downto 0),
     c_in => '0',
     c_out => c0_out,
     s => s0
     );
```

```
    B1: bcd_adder4
    port map (
    a => a(7 downto 4),
    b => b(7 downto 4),
    c_in => c0_out,
    c_out => c1_out,
    s => s1
    );

    B2: bcd_adder4
    port map (
    a => a(11 downto 8),
    b => b(11 downto 8),
    c_in => c1_out,
    c_out => c2_out,
    s => s2
    );
end bcd_adder34;
```

cardiac_final:

```
-- VHDL implementation of the CARDIAC Computer for Basys2 FPGA
Board -
-- cardiac_final.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity cardiac_final is
     port (
     mode0 : in std_logic;
     mode1 : in std_logic;
     mode2 : in std_logic;
     mode3 : in std_logic;
     load0 : in std_logic;
     load1 : in std_logic;
     load2 : in std_logic;
     go : in std_logic;
     step : in std_logic;
     mem_incr : in std_logic;
     clk : in std_logic;
     addr_in : in std_logic_vector(7 downto 0);
     data_in : in std_logic_vector(12 downto 0);
     acc_out : out std_logic_vector(12 downto 0);
     ir : out std_logic_vector(7 downto 0);
     pc : out std_logic_vector(7 downto 0);
     debug : out std_logic_vector (7 downto 0);
     output_mode : out std_logic;
     input_mode : out std_logic;
     addr_out : out std_logic_vector(7 downto 0);
     data_out : out std_logic_vector(12 downto 0)
     );
end cardiac_final;

architecture cardiac_final of cardiac_final is
component memory_final is
     port (
     load0 : in std_logic;
     load1 : in std_logic;
     load2 : in std_logic;
     mode1 : in std_logic;
     mode2 : in std_logic;
     inp_mode : in std_logic;
     clk : in std_logic;
     clr : in std_logic;
     incr : in std_logic;
```

```
      write_mem : in std_logic;

      data_in : in std_logic_vector(12 downto 0);
      addr_in : in std_logic_vector(7 downto 0);
      addr_in_datapath : in std_logic_vector(7 downto 0);
      addr_out : out std_logic_vector(7 downto 0);
      data_out : out std_logic_vector(12 downto 0)
      );
end component;
component datapath_final is
      port (
      clk : in std_logic;
      fetch : in std_logic;
      increment : in std_logic;
      execute : in std_logic;
      clear : in std_logic;
      mode3 : in std_logic;
      output : in std_logic;
      input : in std_logic;
      data_in : in std_logic_vector(12 downto 0);
      acc_out : out std_logic_vector(12 downto 0);
      addr_out : out std_logic_vector(7 downto 0);
      ir : out std_logic_vector(7 downto 0);
      pc : out std_logic_vector(7 downto 0);
      debug : out std_logic_vector(7 downto 0);
      inp_out : out std_logic;
      outp_out : out std_logic;
      mem_load : out std_logic;
      halt : out std_logic
      );
end component;
component control_final is
      port (
      clk : in std_logic;
      clr : in std_logic;
      stop : in std_logic;
      go : in std_logic;
      step : in std_logic;
      inp_in : in std_logic;
      outp_in : in std_logic;
      fetch : out std_logic;
      incr : out std_logic;
      exe : out std_logic;
      inp_out : out std_logic;
      outp_out : out std_logic
      );
end component;
```

```
component mux2g is
     generic(N:integer :=4);
     port(
    a : in std_logic_vector(N-1 downto 0);
     b : in std_logic_vector(N-1 downto 0);
     s : in std_logic;
     y : out std_logic_vector(N-1 downto 0)
     );
end component;
signal run, stop, fetch, increment, execute, write_mem,
mem_load: std_logic;
     -- Signal pins
signal clear_datapath, clear_memory, clear_control: std_logic;
     -- Clear CARDIAC (Memory Clear does not work)
signal addr_out_datapath: std_logic_vector(7 downto 0);
          -- Address out form Datapath
signal data_out_mem, accum_out, data_mux_out:
std_logic_vector(12 downto 0);
     -- Data input mux
signal inp, outp: std_logic;
     -- IO state controls
signal data_sel, load_mem: std_logic;
          -- data input pins
signal output_state, input_state: std_logic;
     -- IO States from control unit

begin
     write_mem <= execute and mem_load and mode3;
     acc_out <= accum_out;
     data_out <= data_out_mem;
     clear_datapath <= load0 and mode0;
     clear_memory <= load1 and mode0;
     clear_control <= load2 and mode0;
     run <= go and mode3;
     data_sel <= mode2 or input_state;
     load_mem <= write_mem;
     output_mode <= output_state;
     input_mode <= input_state;
     --state_out <= fetch & increment & execute;

     MemoryCardiac: memory_final
     port map (
     load0 => load0,
     load1 => load1,
     load2 => load2,
     mode1 => mode1,
     mode2 => mode2,
```

```
inp_mode => input_state,
clk => clk,
clr => clear_memory,
incr => mem_incr,
write_mem => load_mem,
data_in => data_mux_out,
addr_in => addr_in,
addr_in_datapath => addr_out_datapath,
addr_out => addr_out,
data_out => data_out_mem
);

Datapath: datapath_final
port map (
clk => clk,
fetch => fetch,
increment => increment,
execute => execute,
mode3 => mode3,
output => output_state,
input => input_state,
clear => clear_datapath,
data_in => data_out_mem,
acc_out => accum_out,
addr_out => addr_out_datapath,
ir => ir,
pc => pc,
debug => debug,
inp_out => inp,
outp_out => outp,
mem_load => mem_load,
halt => stop
);

Data_Mux: mux2g
generic map (N=>13)
port map (
a => accum_out,
b => data_in,
s => data_sel,
y => data_mux_out
);

CARDIAC_Comp: control_final
port map (
clk => clk,
clr => clear_control,
```

```
        stop => stop,
        go => run,
        step => step,
        inp_in => inp,
        outp_in => outp,
        fetch => fetch,
        incr => increment,
        exe => execute,
        inp_out => input_state,
        outp_out => output_state
        );

end cardiac_final;
```

cardiac_final_top:

```
-- Top file for the CARDIAC Computer for the Basys2 FPGA Board -
-
-- cardiac_final_top.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity cardiac_final_top is
     port (
     sw : in std_logic_vector(7 downto 0);
     btn : in std_logic_vector(3 downto 0);
     mclk : in std_logic;
     ld : out std_logic_vector(7 downto 0);
     a_to_g : out std_logic_vector(6 downto 0);
     an : out std_logic_vector(3 downto 0);
     dp : out std_logic
     );
end cardiac_final_top;

architecture cardiac_final_top of cardiac_final_top is
component cardiac_final is
     port (
     mode0 : in std_logic;
     mode1 : in std_logic;
     mode2 : in std_logic;
     mode3 : in std_logic;
     load0 : in std_logic;
     load1 : in std_logic;
     load2 : in std_logic;
     go : in std_logic;
     step : in std_logic;
     mem_incr : in std_logic;
     clk : in std_logic;
     addr_in : in std_logic_vector(7 downto 0);
     data_in : in std_logic_vector(12 downto 0);
     acc_out : out std_logic_vector(12 downto 0);
     ir : out std_logic_vector(7 downto 0);
     pc : out std_logic_vector(7 downto 0);
     debug : out std_logic_vector (7 downto 0);
     output_mode : out std_logic;
     input_mode : out std_logic;
     addr_out : out std_logic_vector(7 downto 0);
     data_out : out std_logic_vector(12 downto 0)
     );
end component;
```

```
component x7segb is
      port (
      x : in std_logic_vector(15 downto 0);
      clk : in std_logic;
      clr : in std_logic;
      a_to_g : out std_logic_vector(6 downto 0);
      an : out std_logic_vector(3 downto 0);
      dp : out std_logic
      );
end component;
component clock_pulse is
      port (
      inp : in std_logic;
      cclk : in std_logic;
      clr : in std_logic;
      outp : out std_logic
      );
end component;
component mux2g is
      generic(N:integer :=4);
      port(
     a : in std_logic_vector(N-1 downto 0);
      b : in std_logic_vector(N-1 downto 0);
      s : in std_logic;
      y : out std_logic_vector(N-1 downto 0)
      );
end component;
component clkdiv is
      port (
      mclk: in std_logic;
      clr: in std_logic;
      clk25: out std_logic;
      clk190: out std_logic;
      clk3: out std_logic
      );
end component;
component mux4g is
      generic(N:integer :=4);
      port(
       a : in std_logic_vector(N-1 downto 0);
      b : in std_logic_vector(N-1 downto 0);
       c : in std_logic_vector(N-1 downto 0);
       d : in std_logic_vector(N-1 downto 0);
       s : in std_logic_vector(1 downto 0);
       z : out std_logic_vector(N-1 downto 0)
      );
end component;
```

```
signal go_pulse, mode0, mode1, mode2, mode3, mode1_2: std_logic;
     -- modes
signal addr_in: std_logic_vector(7 downto 0);
          -- address input from IO
signal data_in, data_out, acc_out: std_logic_vector(12 downto
0);
     -- Transitional pins from components
signal ir, pc, debug: std_logic_vector(7 downto 0);
     -- cardiac outputs for display
signal dsp_sel: std_logic_vector(1 downto 0);
          -- Display Select
signal dsp0, dsp1, dsp2, dsp3: std_logic_vector(15 downto 0);
     -- Display options
signal dsp, x: std_logic_vector(15 downto 0);
          -- Display output for 7 segment display
signal mode3a, run, clk3: std_logic;
          -- run pin for fast execution
signal step: std_logic;
     -- single step for execution
signal outp, inp: std_logic;
     -- output/input pin to display data contents.

begin
     data_in <= sw(4 downto 0) & sw(3 downto 0) & sw(3 downto
0);
     addr_in <= sw(3 downto 0) & sw(3 downto 0);
     mode0 <= not sw(7) and not sw(6);
     mode1 <= not sw(7) and sw(6);
     mode2 <= sw(7) and not sw(6);
     mode3 <= sw(7) and sw(6);
     mode1_2 <= mode1 or mode2;

     dsp_sel <= sw(1 downto 0) or (outp & outp) or (inp & inp);
     dsp0 <= ir & pc;
     dsp1 <= debug & pc;
     dsp2 <= "000" & acc_out;
     dsp3 <= "000" & data_out;

     mode3a <= sw(7) and sw(6) and sw(5);
     run <= clk3 and mode3a;
     step <= run or btn(3);

     RunPulse: clkdiv
     port map (
     mclk => mclk,
     clr => '0',
```

```
clk3 => clk3
);

DisplayMux: mux4g
generic map (N => 16)
port map (
a => dsp0,
b => dsp1,
c => dsp2,
d => dsp3,
s => dsp_sel,
z => dsp
);

DisplayMux2: mux2g
generic map (N => 16)
port map (
a => dsp,
b => dsp3,
s => mode1_2,
y => x
);

ClockPulseGo: clock_pulse
port map (
inp => step,
cclk => mclk,
clr => '0',
outp => go_pulse
);

Cardiac_LowLevel: cardiac_final
port map (
mode0 => mode0,
mode1 => mode1,
mode2 => mode2,
mode3 => mode3,
load0 => btn(0),
load1 => btn(1),
load2 => btn(2),
go => go_pulse,
step => btn(3),
mem_incr => btn(3),
clk => mclk,
addr_in => addr_in,
data_in => data_in,
acc_out => acc_out,
```

```
        ir => ir,
        pc => pc,
        debug => debug,
        output_mode => outp,
        input_mode => inp,
        addr_out => ld,
        data_out => data_out
        );

        Display: x7segb
        port map(
        x => x,
        clk => mclk,
        clr => '0',
        a_to_g => a_to_g,
        an => an,
        dp => dp
        );
end cardiac_final_top;
```

clkdiv:

```
-- Divides clock into three different speeds for expanded
execution -
-- clkdiv.vhd --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity clkdiv is
     port (
     mclk: in std_logic;
     clr: in std_logic;
     clk25: out std_logic;
     clk190: out std_logic;
     clk3: out std_logic
     );
end clkdiv;

architecture clkdiv of clkdiv is
signal q: std_logic_vector(23 downto 0);
begin
     process(mclk, clr)
     begin
          if clr = '1' then
               q <= X"000000";
          elsif mclk'event and mclk = '1' then
               q <= q + 1;
          end if;
     end process;
     clk25 <= q(0);
     clk190 <= q(17);
     clk3 <= q(23);
end clkdiv;
```

clock_pulse:

```
-- Example 48: Clock Pulse --
-- clock_pulse.vhd --
library IEEE;
use IEEE.std_logic_1164.all;

entity clock_pulse is
     port (
     inp : in std_logic;
     cclk : in std_logic;
     clr : in std_logic;
     outp : out std_logic
     );
end clock_pulse;

architecture clock_pulse of clock_pulse is
signal delay1, delay2, delay3: std_logic;
begin
     process (cclk, clr)
     begin
          if clr = '1' then
               delay1 <= '0';
               delay2 <= '0';
               delay3 <= '0';
          elsif cclk'event and cclk = '1' then
               delay1 <= inp;
               delay2 <= delay1;
               delay3 <= delay2;
          end if;
     end process;
     outp <= delay1 and delay2 and not delay3;
end clock_pulse;
```

clock_pulse2:

```
-- Synchronizes clock and button --
-- clock_pulse2.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity clock_pulse2 is
     port (
     inp : in std_logic;
     cclk : in std_logic;
     clr : in std_logic;
     outp0 : out std_logic;
     outp1 : out std_logic
     );
end clock_pulse2;

architecture clock_pulse2 of clock_pulse2 is

signal delay1, delay2, delay3, delay4, delay5: std_logic;
begin
     process (cclk, clr)
     begin
          if clr = '1' then
               delay1 <= '0';
               delay2 <= '0';
               delay3 <= '0';
               delay4 <= '0';
               delay5 <= '0';
          elsif (cclk'event and cclk = '1') then
               delay1 <= inp;
               delay2 <= delay1;
               delay3 <= delay2;
               delay4 <= delay3;
               delay5 <= delay4;
          end if;
     end process;

     outp0 <= delay1 and delay2 and not delay5;
     outp1 <= delay2 and delay3 and not delay4;

end clock_pulse2;
```

control_final:

```
-- Control unit for the CARDIAC --
-- control_final.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity control_final is
     port (
     clk : in std_logic;
     clr : in std_logic;
     stop : in std_logic;
     go : in std_logic;
     step : in std_logic;
     inp_in : in std_logic;
     outp_in : in std_logic;
     fetch : out std_logic;
     incr : out std_logic;
     exe : out std_logic;
     inp_out : out std_logic;
     outp_out : out std_logic
     );
end control_final;

architecture control_final of control_final is

type state_type is (start, fetch_state, incr_state, exe_state,
input_state, output_state, halt); --need input_state, and
output_state which will go to fetch on step = '1'
signal present_state, next_state: state_type;
begin
     sreg: process (clk, clr)
     begin
          if clr = '1' then
               present_state <= start;
          elsif clk'event and clk = '1' then
               present_state <= next_state;
          end if;
     end process;

     C1: process (present_state, go, stop, inp_in, outp_in,
step)
     begin
          case present_state is
               when start =>
               if go = '1' then
```

```
     next_state <= fetch_state;
else
     next_state <= start;
end if;

when fetch_state =>
if go = '1' then
     next_state <= incr_state;
else
     next_state <= fetch_state;
end if;

when incr_state =>
if go = '1' then
     next_state <= exe_state;
else
     next_state <= incr_state;
end if;

when exe_state =>
if go = '1' and inp_in = '1' then
     next_state <= input_state;
elsif go = '1' and outp_in = '1' then
     next_state <= output_state;
elsif go = '1' and stop = '0' then
     next_state <= fetch_state;
elsif go = '1' and stop = '1' then
     next_state <= halt;
else
     next_state <= exe_state;
end if;

when input_state =>
if step = '1' then
     next_state <= fetch_state;
else
     next_state <= input_state;
end if;

when output_state =>
if step = '1' then
     next_state <= fetch_state;
else
     next_state <= output_state;
end if;

when halt =>
```

```
                next_state <= halt;

                when others => null;
            end case;
        end process;

        C2: process (present_state)
        begin
            fetch <= '0';
            incr <= '0';
            exe <= '0';
            inp_out <= '0';
            outp_out <= '0';

            case present_state is
                when fetch_state =>
                fetch <= '1';

                when incr_state =>
                incr <= '1';

                when exe_state =>
                exe <= '1';

                when input_state =>
                inp_out <= '1';

                when output_state =>
                outp_out <= '1';

                when others => null;
            end case;
        end process;
end control_final;
```

datapath_final:

```
-- Datapath for the Cardiac Computer --
-- datapath_final.vhd --
library IEEE;
use IEEE.std_logic_1164.all;

entity datapath_final is
     port (
     clk : in std_logic;
     fetch : in std_logic;
     increment : in std_logic;
     execute : in std_logic;
     clear : in std_logic;
     mode3 : in std_logic;
     output : in std_logic;
     input : in std_logic;
     data_in : in std_logic_vector(12 downto 0);
     acc_out : out std_logic_vector(12 downto 0);
     addr_out : out std_logic_vector(7 downto 0);
     ir : out std_logic_vector(7 downto 0);
     pc : out std_logic_vector(7 downto 0);
     debug : out std_logic_vector(7 downto 0);
     inp_out : out std_logic;
     outp_out : out std_logic;
     mem_load : out std_logic;
     halt : out std_logic
     );
end datapath_final;

architecture datapath_final of datapath_final is
component add_cardiac is
     port (
     acc_in : in std_logic_vector(12 downto 0);
     data_in : in std_logic_vector(12 downto 0);
     add_out : out std_logic_vector(12 downto 0)
     );
end component;
component subtract_cardiac is
     port (
     acc_in : in std_logic_vector(12 downto 0);
     data_in : in std_logic_vector(12 downto 0);
     sub_out : out std_logic_vector(12 downto 0)
     );
end component;
component shift_cardiac is
     port (
```

```
      data_in : in std_logic_vector(12 downto 0);
      sh_code : in std_logic_vector(2 downto 0);
      sh_out : out std_logic_vector(12 downto 0)
      );
end component;
component decode_final is
      port (
      op : in std_logic_vector(3 downto 0);
      acc_load : out std_logic;
      halt : out std_logic;
      mem_load : out std_logic;
      m0 : out std_logic;
      m1 : out std_logic_vector(1 downto 0);
      neg : out std_logic;
      jump : out std_logic;
      inp : out std_logic;
      outp : out std_logic
      );
end component;
component mux4g is
      generic(N:integer :=4);
      port(
       a : in std_logic_vector(N-1 downto 0);
      b : in std_logic_vector(N-1 downto 0);
       c : in std_logic_vector(N-1 downto 0);
       d : in std_logic_vector(N-1 downto 0);
       s : in std_logic_vector(1 downto 0);
       z : out std_logic_vector(N-1 downto 0)
      );
end component;
component clock_pulse2 is
      port (
      inp : in std_logic;
      cclk : in std_logic;
      clr : in std_logic;
      outp0 : out std_logic;
      outp1 : out std_logic
      );
end component;
component clock_pulse is
      port (
      inp : in std_logic;
      cclk : in std_logic;
      clr : in std_logic;
      outp : out std_logic
      );
end component;
```

```
component mux2g is
     generic(N:integer :=4);
      port(
     a : in std_logic_vector(N-1 downto 0);
      b : in std_logic_vector(N-1 downto 0);
      s : in std_logic;
      y : out std_logic_vector(N-1 downto 0)
     );
end component;
component reg is
     generic (N:integer := 8);
     port (
     load : in std_logic;
     clk : in std_logic;
     clr : in std_logic;
     d : in std_logic_vector(N-1 downto 0);
     q: out std_logic_vector(N-1 downto 0)
     );
end component;
component incr_pc is
     port (
     incr : in std_logic;
     data_in : in std_logic_vector(7 downto 0);
     data_out : out std_logic_vector(7 downto 0)
     );
end component;
signal addr_mux_sel: std_logic;
          -- selects addr_out
signal acc_load_exe: std_logic;
          -- ACC load pin
signal acc_pulse_load, acc_pulse_clk: std_logic;
          -- for ACC clock_pulse2
signal ir_pulse_load, ir_pulse_clk: std_logic;
          -- for IR clock_pulse2
signal incr_pc_load, exe_pc_load, pc_load: std_logic;
          -- PC load pins
signal incr_pulse_load: std_logic;
     -- for Incr circuit
signal acc_load, m0, neg, halt_out: std_logic;
          -- decode outputs
signal m1: std_logic_vector(1 downto 0);
          -- decode output for ALU mux select
signal op_code : std_logic_vector(3 downto 0);
          -- op_code to decode
signal sh_code : std_logic_vector(2 downto 0);
          -- sh_code for shifting
```

```
signal ir_out, pc_data_in, incr_out: std_logic_vector(7 downto
0);
signal sh_out, add_out, sub_out, alu_mux_out:
std_logic_vector(12 downto 0);
     -- ALU outpins into mux for ACC
signal ir_reg_out: std_logic_vector(11 downto 0);
          -- IR output
signal pc_reg_out: std_logic_vector(7 downto 0);
          -- PC output
signal acc_reg_out: std_logic_vector(12 downto 0);
          -- ACC output
signal jump, jmp: std_logic;
     -- jump selection for PC

begin
     halt <= halt_out;

     addr_mux_sel <= (m0 and execute) or output or input;

     jmp <= jump or (acc_reg_out(12) and neg);
     incr_pc_load <= increment and mode3;
     exe_pc_load <= execute and mode3 and jmp;
     pc_load <= halt_out or incr_pc_load or exe_pc_load;
     --pc_load <= increment or (execute and jump);

     acc_out <= sh_out;
     acc_load_exe <= execute and acc_load;
     sh_code <= (not op_code(3) and op_code(2) and not
op_code(1) and not op_code(0)) & ir_out(4) & ir_out(0);

     ir_out <= ir_reg_out(7 downto 0);
     op_code <= ir_reg_out(11 downto 8);

     debug <= pc_load & increment & jmp & jump & neg & mode3 &
execute & exe_pc_load;

     ir <= ir_out;
     pc <= pc_reg_out;

     Dec: decode_final
     port map (
     op => op_code,
     acc_load => acc_load,
     halt => halt_out,
     mem_load => mem_load,
     m0 => m0,
     m1 => m1,
```

```
neg => neg,
jump => jump,
inp => inp_out,
outp => outp_out
);

AddrMux: mux2g
generic map (N => 8)
port map (
a => pc_reg_out,
b => ir_out,
s => addr_mux_sel,
y => addr_out
);

IRClockPulse2: clock_pulse2
port map (
inp => fetch,
cclk => clk,
clr => '0',
outp0 => ir_pulse_load,
outp1 => ir_pulse_clk
);

IRRegister: reg
generic map (N => 12)
port map (
load => ir_pulse_load,
clk => ir_pulse_clk,
clr => clear,
d => data_in(11 downto 0),
q => ir_reg_out
);

PCDataMux: mux2g
generic map (N => 8)
port map (
a => ir_out,
b => incr_out,
s => increment,
y => pc_data_in
);

PCRegister: reg
generic map (N => 8)
port map (
load => pc_load, --pc_pulse_load,
```

```
clk => clk, --pc_pulse_clk,
clr => clear,
d => pc_data_in,
q => pc_reg_out
);

IncrClockPulse2: clock_pulse
port map (
inp => increment,
cclk => clk,
clr => '0',
outp => incr_pulse_load
);

Incr: incr_pc
port map (
incr => incr_pulse_load,
data_in => pc_reg_out,
data_out => incr_out
);

Add: add_cardiac
port map (
acc_in => sh_out,
data_in => data_in,
add_out => add_out
);

Sub: subtract_cardiac
port map (
acc_in => sh_out,
data_in => data_in,
sub_out => sub_out
);

ALUMux: mux4g
generic map (N => 13)
port map (
a => sh_out,
b => add_out,
c => data_in,
d => sub_out,
s => m1,
z => alu_mux_out
);

ACCClockPulse2: clock_pulse2
```

```
        port map (
        inp => acc_load_exe,
        cclk => clk,
        clr => '0',
        outp0 => acc_pulse_load,
        outp1 => acc_pulse_clk
        );

        Acc: reg
        generic map (N => 13)
        port map (
        load => acc_pulse_load,
        clk => acc_pulse_clk,
        clr => clear,
        d => alu_mux_out,
        q => acc_reg_out
        );

        Shift: shift_cardiac
        port map (
        data_in => acc_reg_out,
        sh_code => sh_code,
        sh_out => sh_out
        );

end datapath_final;
```

decode_final:

```
-- decode the op code for the cardiac computer --
-- decode_final.vhd --
library IEEE;
use IEEE.std_logic_1164.all;

entity decode_final is
     port (
     op : in std_logic_vector(3 downto 0);
     acc_load : out std_logic;
     halt : out std_logic;
     mem_load : out std_logic;
     m0 : out std_logic;
     m1 : out std_logic_vector(1 downto 0);
     neg : out std_logic;
     jump : out std_logic;
     inp : out std_logic;
     outp : out std_logic
     );
end decode_final;
architecture decode_final of decode_final is
signal s: std_logic_vector(9 downto 0);
begin
     process (s, op)
     begin
          case op is
               when "0000" => s <= "0100000000";
               when "0001" => s <= "0000010101";
               when "0010" => s <= "0000010011";
               when "0011" => s <= "0000100000";
               when "0100" => s <= "0000010000";
               when "0101" => s <= "1000000000";
               when "0110" => s <= "0000001001";
               when "0111" => s <= "0000010111";
               when "1000" => s <= "0010000000";
               when "1001" => s <= "0001000000";
               when others => s <= "0000000000";
          end case;
     end process;

     outp <= s(9);
     inp <= s(8);
     jump <= s(7);
     halt <= s(6);
     neg <= s(5);
     acc_load <= s(4);
```

```
        mem_load <= s(3);
        m1 <= s(2 downto 1);
        m0 <= s(0);
    end decode_final;
```

incr_bcd2:

```
-- Increments a 2 digit BCD number --
-- incr_bcd2.vhd --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity incr_bcd2 is
     port (
     data_in : in std_logic_vector(7 downto 0);
     data_out : out std_logic_vector(7 downto 0)
     );
end incr_bcd2;

architecture incr_bcd2 of incr_bcd2 is
component bcd_adder4 is
     port (
     a : in std_logic_vector(3 downto 0);
     b : in std_logic_vector(3 downto 0);
     c_in : in std_logic;
     c_out : out std_logic;
     s : out std_logic_vector(3 downto 0)
     );
end component;

signal c0_out: std_logic;
signal s0, s1: std_logic_vector(3 downto 0);
signal data: std_logic_vector(7 downto 0);
begin
     data <= s1 & s0;


     B4: bcd_adder4
     port map (
     a => data_in(3 downto 0),
     b => "0001",
     c_in => '0',
     c_out => c0_out,
     s => s0
     );

     B5: bcd_adder4
     port map (
     a => data_in(7 downto 4),
     b => "0000",
```

```
        c_in => c0_out,
        s => s1
        );

        process (data)
        begin
             if data = "00000000" then
                  data_out <= "00000001";
             else
                  data_out <= data;
             end if;
        end process;

end incr_bcd2;
```

incr_pc:

```
-- Increment Circuit for the PC --
-- incr_pc.vhd --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity incr_pc is
     port (
     incr : in std_logic;
     data_in : in std_logic_vector(7 downto 0);
     data_out : out std_logic_vector(7 downto 0)
     );
end incr_pc;

architecture incr_pc of incr_pc is
component incr_bcd2 is
     port (
     data_in : in std_logic_vector(7 downto 0);
     data_out : out std_logic_vector(7 downto 0)
     );
end component;
signal data: std_logic_vector(7 downto 0);
begin

     IncrementCircuit: incr_bcd2
     port map (
     data_in => data_in,
     data_out => data
     );

     process (incr, data, data_in)
     begin
          if incr = '1' then
               data_out <= data;
          else
               data_out <= data_in;
          end if;
     end process;
end incr_pc;
```

mem_addr_reg:

```
-- Memory Address Register that contains an address from 0 - 99
--
-- mem_addr_reg.vhd --
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mem_addr_reg is
     port (
     load0 : in std_logic;
     load1 : in std_logic;
     clk : in std_logic;
     incr : in std_logic;
     addr_in : in std_logic_vector(7 downto 0);
     addr_out_bcd : out std_logic_vector(7 downto 0);
     addr_out : out std_logic_vector(6 downto 0)
     );
end mem_addr_reg;
architecture mem_addr_reg of mem_addr_reg is
type mem_type is array(0 to 1) of std_logic_vector(3 downto 0);
signal mem_array: mem_type;
signal addr_out_trimmed: std_logic_vector(7 downto 0);
begin
     process (clk)
     begin
          if (clk'event and clk = '1') then
               if load0 = '1' then
                    mem_array(0) <= addr_in(3 downto 0);
               end if;
               if load1 = '1' then
                    mem_array(1) <= addr_in(7 downto 4);
               end if;
               if incr = '1' then
                    if conv_integer(mem_array(0)) >= 9 then
                         if conv_integer(mem_array(1)) >= 9 then
                              mem_array(1) <= "0000";
                         else
                              mem_array(1) <= mem_array(1) +
                              "0001";
                         end if;
                         mem_array(0) <= "0000";
                    else
                         mem_array(0) <= mem_array(0) + "0001";
                    end if;
               end if;
```

```
            end if;
        end process;
        addr_out_trimmed <= (mem_array(1) * "1010") + mem_array(0);
        addr_out <= addr_out_trimmed(6 downto 0);
        addr_out_bcd <= mem_array(1) & mem_array(0);
end mem_addr_reg;
```

mem_stack1_final:

```
-- Creates a 100 x 1 bit memory stack --
-- mem_stack4.vhd --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mem_stack1_final is
     port (
     load: in std_logic;
     clk : in std_logic;
     --clr : in std_logic;
     addr : in std_logic_vector(6 downto 0);
     data_in : in std_logic;
     data_out : out std_logic
     );
end mem_stack1_final;

architecture mem_stack1_final of mem_stack1_final is
type ram_type is array(0 to 127) of std_logic;
signal ram_array: ram_type;
begin
     process(clk) --, clr)
     begin
          if (clk'event and clk = '1') then
               if load = '1' then
                    ram_array(conv_integer(addr)) <= data_in;
               end if;
          end if;
     end process;
     data_out <= ram_array(conv_integer(addr));
end mem_stack1_final;
```

mem_stack4_final:

```
-- Creates a 100 x 4 bits memory stack --
-- mem_stack4_final.vhd --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity mem_stack4_final is
     port (
     load: in std_logic;
     clk : in std_logic;
     --clr : in std_logic;
     addr : in std_logic_vector(6 downto 0);
     data_in : in std_logic_vector(3 downto 0);
     data_out : out std_logic_vector(3 downto 0)
     );
end mem_stack4_final;

architecture mem_stack4_final of mem_stack4_final is
type ram_type is array(0 to 127) of std_logic_vector(3 downto
0);
signal ram_array: ram_type;
begin
     process(clk) --, clr)
     begin
--        if clr = '1' then
--             ram_array <= (others=> (others=>'0'));
--        els
          if (clk'event and clk = '1') then
               if load = '1' then
                    ram_array(conv_integer(addr)) <= data_in;
               end if;
          end if;
     end process;
     data_out <= ram_array(conv_integer(addr));
end mem_stack4_final;
```

memory_final:

```
-- 100 by 1000 memory circuit where it takes an address input
from datapath as well and is muxed depending on the mode--
-- memory_final.vhd --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity memory_final is
     port (
     load0 : in std_logic;
     load1 : in std_logic;
     load2 : in std_logic;
     mode1 : in std_logic;
     mode2 : in std_logic;
     inp_mode : in std_logic;
     clk : in std_logic;
     clr : in std_logic;
     incr : in std_logic;
     write_mem : in std_logic;
     data_in : in std_logic_vector(12 downto 0);
     addr_in : in std_logic_vector(7 downto 0);
     addr_in_datapath : in std_logic_vector(7 downto 0);
     addr_out : out std_logic_vector(7 downto 0);
     data_out : out std_logic_vector(12 downto 0)
     );
end memory_final;

architecture memory_final of memory_final is
component clock_pulse is
     port (
     inp : in std_logic;
     cclk : in std_logic;
     clr : in std_logic;
     outp : out std_logic
     );
end component;

component mem_addr_reg is
     port (
     load0 : in std_logic;
     load1 : in std_logic;
     clk : in std_logic;
     incr : in std_logic;
     addr_in : in std_logic_vector(7 downto 0);
```

```
      addr_out_bcd : out std_logic_vector(7 downto 0);
      addr_out : out std_logic_vector(6 downto 0)
      );
end component;

component mem_stack4_final is
      port (
      load: in std_logic;
      clk : in std_logic;
      addr : in std_logic_vector(6 downto 0);
      data_in : in std_logic_vector(3 downto 0);
      data_out : out std_logic_vector(3 downto 0)
      );
end component;

component mem_stack1_final is
      port (
      load: in std_logic;
      clk : in std_logic;
      addr : in std_logic_vector(6 downto 0);
      data_in : in std_logic;
      data_out : out std_logic
      );
end component;

component mux2g is
      generic(N:integer :=4);
      port(
      a : in std_logic_vector(N-1 downto 0);
      b : in std_logic_vector(N-1 downto 0);
      s : in std_logic;
      y : out std_logic_vector(N-1 downto 0)
      );
end component;

signal internal_addr_sel, mem_load0_pulse, mem_load1_pulse,
mem_load0,
      mem_load1, mem_incr, sign_load, s2_load, s1_load, s0_load,
      sign_out, incr_pulse: std_logic;
signal addr, addr_datapath, internal_addr: std_logic_vector
      (6 downto 0);
signal digit2_out, digit1_out, digit0_out: std_logic_vector
      (3 downto 0);

begin
      internal_addr_sel <= mode1 or mode2;
      mem_load0 <= (mode1 and load0);
```

```
      mem_load1 <= (mode1 and load1);
      mem_incr <= mode2 and incr_pulse;
      sign_load <= ((mode2 or inp_mode) and load2) or write_mem;
      s2_load <= sign_load;
      s1_load <= ((mode2 or inp_mode) and load1) or write_mem;
      s0_load <= ((mode2 or inp_mode) and load0) or write_mem;
      data_out <= sign_out & digit2_out & digit1_out &
digit0_out;
      addr_datapath <= (addr_in_datapath(7 downto 4) * "1010") +
           addr_in_datapath(3 downto 0);

      InternalAddressMux: mux2g
      generic map (N=>7)
      port map (
      a => addr_datapath,
      b => addr,
      s => internal_addr_sel,
      y => internal_addr
      );

      Clock: clock_pulse
      port map(
      inp => incr,
      cclk => clk,
      clr => clr,
      outp => incr_pulse
      );

      MemLoad0: clock_pulse
      port map(
      inp => mem_load0,
      cclk => clk,
      clr => clr,
      outp => mem_load0_pulse
      );

      MemLoad1: clock_pulse
      port map(
      inp => mem_load1,
      cclk => clk,
      clr => clr,
      outp => mem_load1_pulse
      );

      MemAddrReg: mem_addr_reg
      port map (
      load0 => mem_load0_pulse,
```

```
      load1 => mem_load1_pulse,
      clk => clk,
      incr => mem_incr,
      addr_in => addr_in,
      addr_out_bcd => addr_out,
      addr_out => addr
      );

      RamStackSign: mem_stack1_final
      port map (
      load => sign_load,
      clk => clk,
      addr => internal_addr,
      data_in => data_in(12),
      data_out => sign_out
      );

      RamStackDigit2: mem_stack4_final
      port map(
      load => s2_load,
      clk => clk,
      --clr => clr,
      addr => internal_addr,
      data_in => data_in(11 downto 8),
      data_out => digit2_out
      );

      RamStackDigit1: mem_stack4_final
      port map(
      load => s1_load,
      clk => clk,
      addr => internal_addr,
      data_in => data_in(7 downto 4),
      data_out => digit1_out
      );

      RamSTackDigit0: mem_stack4_final
      port map(
      load => s0_load,
      clk => clk,
      addr => internal_addr,
      data_in => data_in(3 downto 0),
      data_out => digit0_out
      );
end memory_final;
```

mux_2g:

```
-- A two to one mux for any size inputs --
-- mux2g.vhd--

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux2g is
     generic(N:integer :=4);
     port(
     a : in std_logic_vector(N-1 downto 0);
     b : in std_logic_vector(N-1 downto 0);
     s : in std_logic;
     y : out std_logic_vector(N-1 downto 0)
     );
end mux2g;

architecture mux2g of mux2g is
begin
     p1: process (a, b, s)
     begin
          if s = '0' then
               y <= a;
          else
               y <= b;
          end if;
     end process;

end mux2g;
```

mux_4g:

```
-- A generic four to one mux for any sized input --
-- mux4g.vhd--

library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity mux4g is
     generic(N:integer :=4);
     port(
      a : in std_logic_vector(N-1 downto 0);
     b : in std_logic_vector(N-1 downto 0);
      c : in std_logic_vector(N-1 downto 0);
      d : in std_logic_vector(N-1 downto 0);
      s : in std_logic_vector(1 downto 0);
      z : out std_logic_vector(N-1 downto 0)
     );
end mux4g;

architecture mux4g of mux4g is

component mux2g
     generic(N:integer :=4);
      port(
      a : in std_logic_vector(N-1 downto 0);
      b : in std_logic_vector(N-1 downto 0);
      s : in std_logic;
      y : out std_logic_vector(N-1 downto 0)
     );
end component;

signal v, w: std_logic_vector(N-1 downto 0);

begin

     M1: mux2g generic map(N => N) port map
     (a => a, b => b, s => s(0), y => v);
     M2: mux2g generic map(N => N) port map
     (a => c, b => d, s => s(0), y => w);
     M3: mux2g generic map(N => N) port map
     (a => v, b => w, s => s(1), y => z);

end mux4g;
```

reg:

```
-- A register to hold any sized input -
-- reg.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity reg is
     generic (N:integer := 8);
     port (
     load : in std_logic;
     clk : in std_logic;
     clr : in std_logic;
     d : in std_logic_vector(N-1 downto 0);
     q: out std_logic_vector(N-1 downto 0)
     );
end reg;

architecture  reg of reg is
begin
     process (clk, clr)
     begin
          if clr = '1' then
               q <= (others => '0');
          elsif clk'event and clk = '1' then
               if load = '1' then
                    q <= d;
               end if;
          end if;
     end process;
end reg;
```

shift_cardiac:

```
-- Does a single right shift, a single left shift, or no shift -
-
-- shift_cardiac.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity shift_cardiac is
     port (
     data_in : in std_logic_vector(12 downto 0);
     sh_code : in std_logic_vector(2 downto 0);
     sh_out : out std_logic_vector(12 downto 0)
     );
end shift_cardiac;

architecture shift_cardiac of shift_cardiac is

component mux4g is
     generic(N:integer :=4);
     port(
      a : in std_logic_vector(N-1 downto 0);
      b : in std_logic_vector(N-1 downto 0);
      c : in std_logic_vector(N-1 downto 0);
      d : in std_logic_vector(N-1 downto 0);
      s : in std_logic_vector(1 downto 0);
      z : out std_logic_vector(N-1 downto 0)
     );
end component;

component mux2g is
     generic(N:integer := 2);
     port (
     a : in std_logic_vector(N-1 downto 0);
     b : in std_logic_vector(N-1 downto 0);
     s : in std_logic;
     y : out std_logic_vector(N-1 downto 0)
     );
end component;

signal shr, shl: std_logic_vector(12 downto 0);
signal sh_sel: std_logic_vector(1 downto 0);

begin
     shr <= data_in(12) & "0000" & data_in(11 downto 8) &
          data_in(7 downto 4);
```

```
    shl <= data_in(12) & data_in(7 downto 4) & data_in(3 downto
0) &
          "0000";

    M1: mux2g
    generic map (N => 2)
    port map (
    a => "00",
    b => sh_code(1 downto 0),
    s => sh_code(2),
    y => sh_sel
    );

    M2: mux4g
    generic map (N => 13)
    port map (
    a => data_in,
    b => shr,
    c => shl,
    d => data_in,
    s => sh_sel,
    z => sh_out
    );

end shift_cardiac;
```

subtract_cardiac:

```
-- Subtracts the datapath from the Accumulator --
-- subtract_cardiac.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity subtract_cardiac is
     port (
     acc_in : in std_logic_vector(12 downto 0);
     data_in : in std_logic_vector(12 downto 0);
     sub_out : out std_logic_vector(12 downto 0)
     );
end subtract_cardiac;

architecture subtract_cardiac of subtract_cardiac is

component add_cardiac is
     port (
     acc_in : in std_logic_vector(12 downto 0);
     data_in : in std_logic_vector(12 downto 0);
     add_out : out std_logic_vector(12 downto 0)
     );
end component;
signal data: std_logic_vector(12 downto 0);
begin
     data <= (not data_in(12)) & data_in(11 downto 0);

     A3: add_cardiac
     port map (
     acc_in => acc_in,
     data_in => data,
     add_out => sub_out
     );
end subtract_cardiac;
```

tens_compliment13:

```
-- creates a tens compliment of a signed 3 digit BCD number (13
bits)-
-- tens_compliment13.vhd --

library IEEE;
use IEEE.std_logic_1164.all;

entity tens_compliment13 is
     port (
     a : in std_logic_vector(12 downto 0);
     s : out std_logic_vector(12 downto 0)
     );
end tens_compliment13;

architecture tens_compliment13 of tens_compliment13 is
component bcd_adder34 is
     port (
     a : in std_logic_vector(12 downto 0);
     b : in std_logic_vector(12 downto 0);
     s : out std_logic_vector(13 downto 0)
     );
end component;

signal digit2, s2, digit1, s1, digit0, s0: std_logic_vector
     (3 downto 0);
signal sf: std_logic_vector(13 downto 0);
signal s_out: std_logic_vector(12 downto 0);
begin
     digit2 <= a(11 downto 8);
     digit1 <= a(7 downto 4);
     digit0 <= a(3 downto 0);

     process (digit2, digit1, digit0, s0, s1, s2)
     begin
          case digit2 is
               when "0000" => s2 <= "1001";
               when "0001" => s2 <= "1000";
               when "0010" => s2 <= "0111";
               when "0011" => s2 <= "0110";
               when "0100" => s2 <= "0101";
               when "0101" => s2 <= "0100";
               when "0110" => s2 <= "0011";
               when "0111" => s2 <= "0010";
               when "1000" => s2 <= "0001";
               when "1001" => s2 <= "0000";
```

```
            when others => s2 <= "0000";
        end case;
        case digit1 is
            when "0000" => s1 <= "1001";
            when "0001" => s1 <= "1000";
            when "0010" => s1 <= "0111";
            when "0011" => s1 <= "0110";
            when "0100" => s1 <= "0101";
            when "0101" => s1 <= "0100";
            when "0110" => s1 <= "0011";
            when "0111" => s1 <= "0010";
            when "1000" => s1 <= "0001";
            when "1001" => s1 <= "0000";
            when others => s1 <= "0000";
        end case;
        case digit0    is
            when "0000" => s0 <= "1001";
            when "0001" => s0 <= "1000";
            when "0010" => s0 <= "0111";
            when "0011" => s0 <= "0110";
            when "0100" => s0 <= "0101";
            when "0101" => s0 <= "0100";
            when "0110" => s0 <= "0011";
            when "0111" => s0 <= "0010";
            when "1000" => s0 <= "0001";
            when "1001" => s0 <= "0000";
            when others => s0 <= "0000";
        end case;
    end process;

    s_out <= a(12) & s2 & s1 & s0;

    B1: bcd_adder34
    port map (
    a => s_out,
    b => "0000000000001",
    s => sf
    );

    s <= sf(12 downto 0);
end tens_compliment13;
```

x7segbc:

```
-- Example 52: x7segbc - input cclk should be 190 Hz --
-- x7segbc.vhd --
-- x7segb.vhd --

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity x7segbc is
     port (
     x : in std_logic_vector(15 downto 0);
     cclk : in std_logic;
     clr : in std_logic;
     a_to_g : out std_logic_vector(6 downto 0);
     an : out std_logic_vector(3 downto 0);
     dp : out std_logic
     );
end x7segbc;

architecture x7segbc of x7segbc is

signal s : std_logic_vector(1 downto 0);
signal digit : std_logic_vector(3 downto 0);
signal aen : std_logic_vector(3 downto 0);

begin
     dp <= '1';
     aen(3) <= x(15) or x(14) or x(13) or x(12);
     aen(2) <= x(15) or x(14) or x(13) or x(12) or x(11) or
x(10) or x(9) or x(8);
     aen(1) <= x(15) or x(14) or x(13) or x(12) or x(11) or
x(10) or x(9) or x(8) or x(7) or x(6) or x(5) or x(4);
     aen(0) <= '1'; -- digit 0 always on

-- Quad 4-to-1 MUX: mux44
     process (s, x)
     begin
          case s is
               when "00" => digit <= x(3 downto 0);
               when "01" => digit <= x(7 downto 4);
               when "10" => digit <= x(11 downto 8);
               when others => digit <= x(15 downto 12);
          end case;
     end process;
```

```
--// 7-segment decoder: hex7seg
     process (digit)
     begin
          case digit is
               when X"0" => a_to_g <= "0000001";
               when X"1" => a_to_g <= "1001111";
               when X"2" => a_to_g <= "0010010";
               when X"3" => a_to_g <= "0000110";
               when X"4" => a_to_g <= "1001100";
               when X"5" => a_to_g <= "0100100";
               when X"6" => a_to_g <= "0100000";
               when X"7" => a_to_g <= "0001101";
               when X"8" => a_to_g <= "0000000";
               when X"9" => a_to_g <= "0000100";
               when X"A" => a_to_g <= "0001000";
               when X"B" => a_to_g <= "1100000";
               when X"C" => a_to_g <= "0110001";
               when X"D" => a_to_g <= "1000010";
               when X"E" => a_to_g <= "0110000";
               when others => a_to_g <= "0111000";
          end case;
     end process;

-- Digit select: ancode
     process (s, aen)
     begin
          an <= "1111";
          if aen(conv_integer(s)) = '1' then
               an(conv_integer(s)) <= '0';
          end if;
     end process;

-- 2-bit counter
     process (cclk, clr)
     begin
          if clr = '1' then
               s <= "00";
          elsif cclk'event and cclk = '1' then
               s <= s + 1;
          end if;
     end process;
end x7segbc;
```