A User-Dependent Voice Verification System

Indy Yu
Tian He

Chris Terman
6.111
December 08, 2004

Abstract:

The Voice Verification System receives spoken keywords/passwords from a specific user through a microphone, performs data-matching analysis, and in turn outputs a verification signal indicated by an LED. The system has several modules, including a front-end speech magnitude and zero-crossing rate calculations modules, a threshold calculation module, a endpoint-detection module, a waveform warping module and a pattern-recognition module. The spectral analysis processor receives and filters the input signal. The feature extractor extracts the relevant information from the input by separating actual speech waveform from background noise. The waveform warping module then takes in the extracted data and fits them to the template waveform. The recognition module will then use a template matching method to verify the user's voice. Indy Yu is responsible for the spectral analysis processor and waveform feature extractor, while Tian He is responsible for the warping and pattern-recognition modules.

Table of Contents/List of Figures

List of Tables

List of Figures

Introduction

The Voice Verification System implemented in this project is a speaker-dependent, pattern-matching system. The unit has a store mode and an operational pattern matching mode. In store mode, a new template can replace the existing template. After storage, the system automatically goes through a template check to make sure the stored template value is the same as the input. During pattern-matching operations, the unit takes in a speakers voice, compares the waveform to that of the template and decides whether or not the input passes a threshold level.

The above system has many uses. It can be used as the base system for a simple word recognition system that can perform various tasks based on what the user tells it to do. The system can also be used as a security agent, unlocking a door or safe only when a specific user speaks a select set of passwords.

The implementation of this system can be divided into two halves: the front-end and the back-end.

The front-end consists of three main parts: The Average Magnitude calculation modules, the Zero-Crossing calculation modules and the speech location determination modules. The Average Magnitude and Zero-Crossing calculation modules store the inputs magnitude and zero-crossing rate in real-time, and calculate the mean and standard deviation values of the magnitude and zero-crossing rate of the background noise. The Location Detector then takes in these values to find threshold values used to determine the location of the speech waveform in memory.

The input speech waveform is first taken in by a microphone and converted to digital format by an Analog-to-Digital Converter. The eight-bit data is then processed to find the average magnitude and zero-crossing rate, which are stored in two separate SRAMS. There are three threshold values used: the upper and lower magnitude thresholds, ITU and ITL, respectively, and the zero-crossing rate threshold IZCT. Therefore the endpoints to the location of the speech are determined three times for high accuracy and high sensitivity.

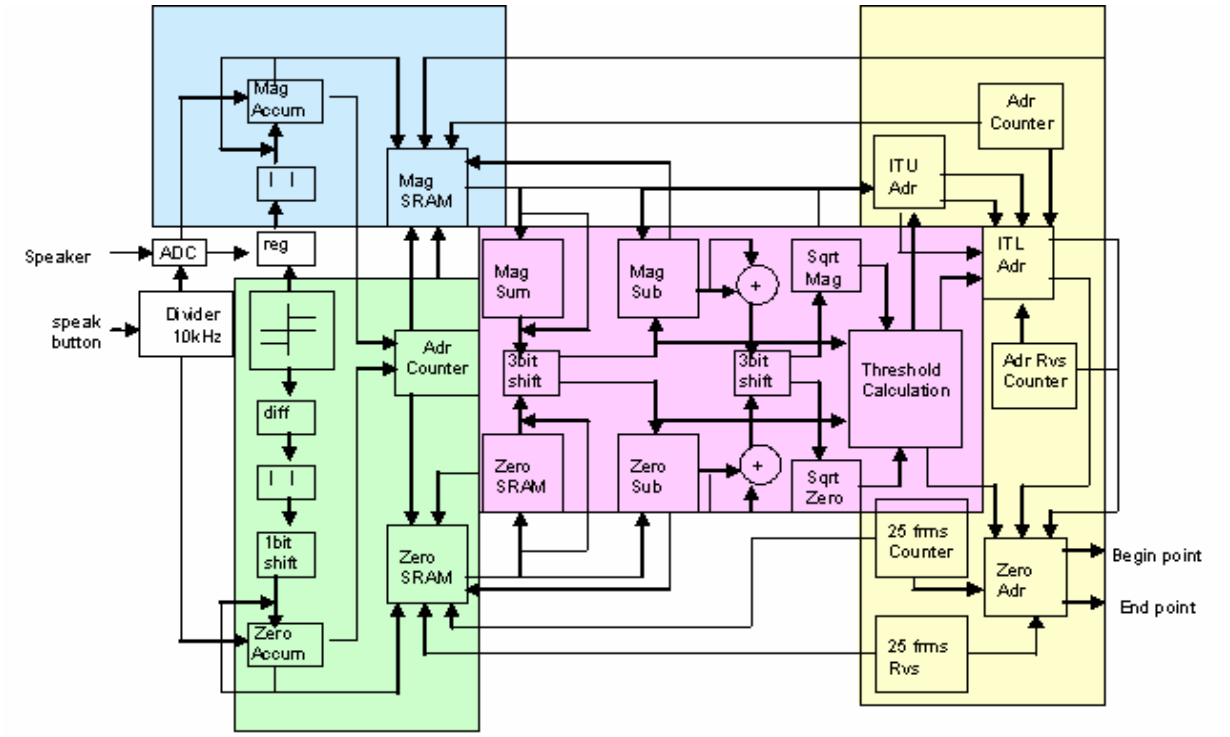
The back end consists of a waveform warping module and a pattern-recognition module. The back-end functions as a memory and storage database, as well as a decision unit that implements an algorithm based on the Itakura Time-Warping method to pattern match the template waveform (speaker's voice) to that of the input waveform. The backend tallies the distance error between the two waveforms and tests the distance sum against 1 of 4 user selected threshold levels (select by 2 switches).

As a standalone module without the front end inputs, the backend allows for 4 different input waveforms that the user can select (2 toggle switches), 2 methods of storing the template memory—Flash (1) or Internal RAM (0)—(1 toggle switch), 3 different waveforms (Cut input data, Template Memory Data, Warped Input Data) to output to the AD558 D/A DAC (2 toggle switches), and a DAC enable toggle that outputs the selected waveform. When integrate with the front-end module, the input waveform select option disappears, and the backend simply wait for a startBE signal from the front-end to start the backend major FSM. Input signal is accessed from the front end input RAM.

Module Description/Implementation

Front-End (by Indy Yu)

The analog speech waveform is accepted when a 'speak' button is on. It is sampled at 10k Hz by the ADC, and processed through mainly by the control of five Finite State Machines in the front-end system. They are the Analog-to-Digital control, the noise calculation control, and three FSMs that use the three threshold values ITU, ITL and IZCT to determine the endpoint locations to the speech waveform by searching though the Average Magnitude and Zero-Crossing Rate SRAMs. In the following paragraphs I will describe the functionality of each sub-unit of the front-end system, which is mainly controlled by the FSMs mentioned above. Figure1 is the overall block diagram of the front-end of the system.



Firgure1: Overall Front-end Block Diagram

Magnitude Calculation Unit

The Magnitude Calculation Unit stores the magnitude of the speech in a SRAM and calculates the mean and standard deviation of the magnitude of background noise. First, the magnitude of each input speech sample is taken in real-time, then the waveform passes though an impulse window of magnitude 1 and width 100 so it is smoothed out by performing the convolution M[n] = sum(|x[m]|*[h[n-m]), where x[n] is the input and h[n] is the window of 100 frames. Each of the convolution value would be the result of 100 accumulations of speech samples and is sampled at 100 Hz and stored in a memory location of the Magnitude SRAM. There are a total of 256 locations being written, and the address is incremented by a counter.

While the values are being stored, the noise accumulator takes the values of the first few locations of memory, which guarantee to not contain a valid speech waveform, and uses them to compute the average magnitude of noise. After the mean is computed and all 256 SRAM locations have being written, the Noise FSM goes back to the first location in memory and uses the consecutive values in memory to calculate the standard deviation of the noise magnitude. The mean and standard deviation values are sent to the Threshold Calculation Unit when they are available.

Zero-Crossing Calculation Unit

While the Magnitude Calculation Unit is computing, the Zero-Crossing calculations are performed in parallel, and the process is fairly similar. The module first compares two adjacent speech samples to see whether there is a sign change, then the

number of sign changes within a 100 window frame is calculated at 100Hz and stored in one location of the Zero-Crossing SRAM, such that the each convolution value of Z = sum(|sign(x[m])-sign{x[m-1]|*h[n-m], where x[n] is the speech sample and h[n] is the 100-frame window, is stored in memory in real-time. After the Zero-Crossing values fills up 256 locations of the SRAM, the Noise FSM calculates the mean and standard deviations of noise zero-crossing rate in the same way as the noise magnitude calculations are performed.

Threshold Calculation Unit

As mentioned above, the noise average is calculated in real-time. Data from location zero through seven is first accumulated and then shifted to compute the average for both magnitude and zero-crossing rate, assuming these data don't contain the real speech waveform. Then average is then subtracted from each of the values of the eight locations, and that result is squared, accumulated, shifted and taken the square-root to compute the standard deviation of the noise magnitude and zero-crossing rate. The average and deviation values are used to set the three thresholds ITU, ITL and IZCT to determine the endpoint locations.

The thresholds ITU and ITL are the upper and lower thresholds used to temporally determine the endpoint locations based on the values stored in the Magnitude SRAM, and IZCT is compared with the values in the Zero-Crossing SRAM to establish the final endpoint locations of the speech waveform. Threshold ITU = k1*(noise average magnitude), where k1 is a variable that is set manually based on the relative behavior of the waveform such that a valid speech magnitude waveform should never fall below this

value. After some experimentations are performed, k1 is set to 2. Threshold ITL is used to improve the accuracy of the endpoint detection. ITL = (average noise magnitude) + k2*(noise magnitude standard deviation), where k2 is also determined by experimentation and is set to 3. Threshold IZCT = (average zero crossing rate) + k3* (standard deviation of zero-crossing rate), where k3 is set to 3 as well.

Endpoint Detection Unit

After the threshold values are determined, the ITU FSM, ITL FSM and IZCT FSM searches through the values in the two SRAMs to establish the endpoints. The ITU FSM uses the upper threshold ITU to temporally set the endpoints of the waveform. The ITU FSM first resets the counter, which increments the Magnitude SRAM address, and then start increment the counter until the value from a particular location exceeds the upper threshold ITU, and the location is temporally set as the begin-point. Next, the address pointer is set to the last location of the SRAM and begins decrement. This is accomplished by subtracting the value of the counter by the highest possible value of the RAM address. Once a value exceeds the upper threshold, the location of the value is set as the temporary endpoint, and the ITL FSM initiates.

The ITL FSM uses the lower threshold ITL to further improve the accuracy of the endpoints determined by the ITU FSM. It first set the address pointer of the Magnitude SRAM to the memory location of the temporary begin-point, decrements the address using the same method of subtracting the counter from the highest address value. Once a value falls below the threshold, the corresponding address is set as the new begin-point. The new endpoint is found the same way by set the address counter to the temporary

endpoint and increments the counter till a value falls below the threshold. IZCT FSM initiates once the new endpoints are determined.

The IZCT FSM uses threshold IZCT to determine the final endpoints to the location of the speech waveform. If there is unvoiced speech, the zero-crossing rate should be much higher than that of the background noise. The FSM first sets the address pointer of the Zero-Crossing SRAM to the current begin-point, and traces back 25 frames by decrementing the address counter, if there are more than three values that exceeds the threshold IZCT, the begin-point is moved back to the first location where the threshold value is exceeded. Otherwise, the begin-point remains the same as previously determined. The endpoint is found the same fasion by incrementing the address counter from the current endpoint. Figure 2 demonstrates how the endpoints are found though this process of using the three thresholds ITU, ITL, and IZCT.

Once the final endpoints are determined, a ‘done’ signal is sent to initiate the Back-end of the system and the Magnitude SRAM is accessed for time warping and template matching. However, since the magnitude values are fourteen bits, only the most eight significant are selected and used for Back-end processing

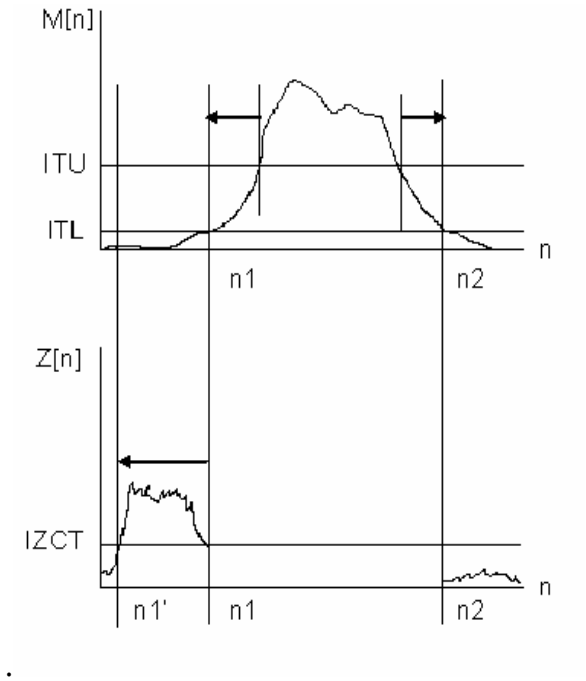


Figure 2: Endpoint Detection Process Diagram

Back-End (by Tian He)

The Back-end contains two major FSM's. When in normal operations and during store mode, the Backend Controller controls the operations of the subsystem. When the system is in DAC mode, the DAC_fsm runs to control the relay of RAM information for output to the AD558 DAC. When integrated with the front-end, the back-end receives the startBE, endpt, beginpt values from the front-end. The back-end starts operation hen the front-module turns startBE high.

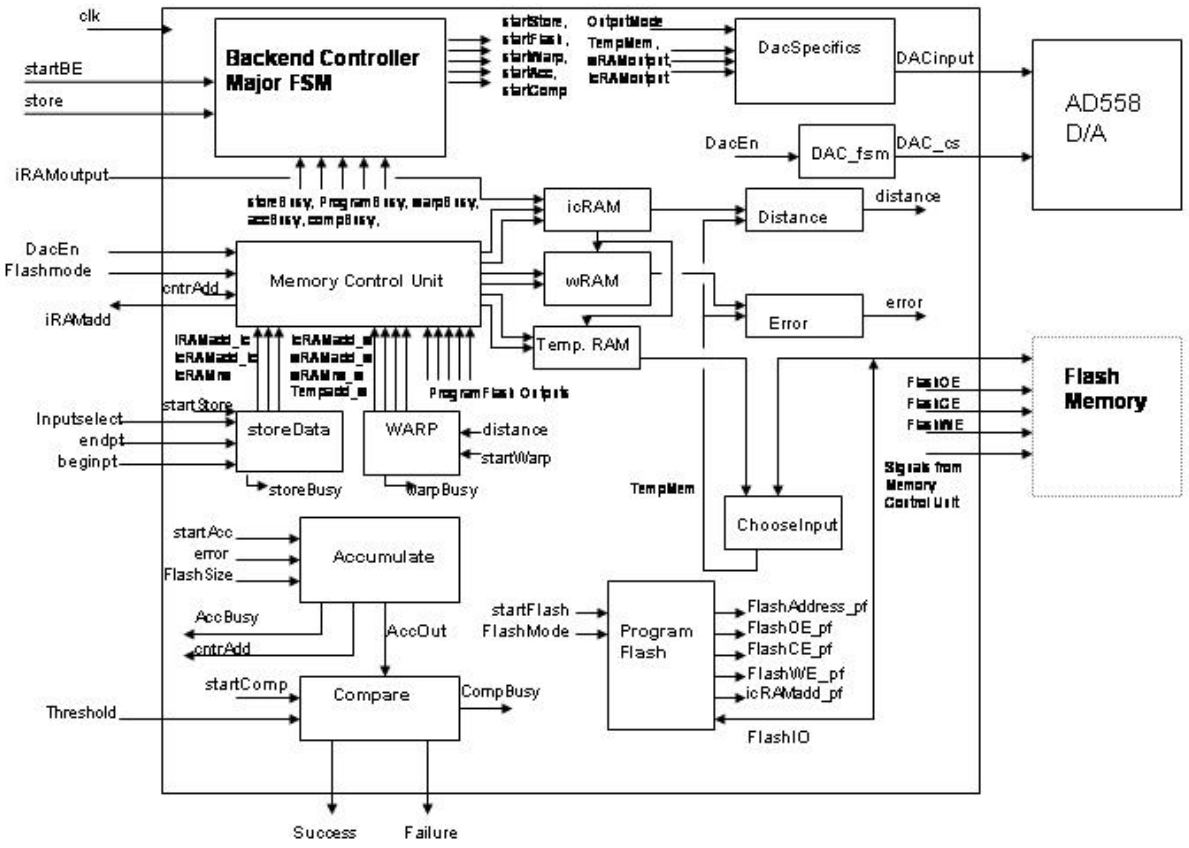


Figure 3: Overall Backend Block Diagram

Backend Controller Module (Backend Controller)

The Backend Controller Module is responsible for directing all operations of the backend subsystem. In short, it is the major FSM for the backend.

The Backend Controller takes in as inputs the signals: store, startBE(starts Backend), DACen, the various busy status from minor FSM's. The inputs "store" and "DAC_enable" are given top priorities. At any point when these signals are sensed all modules in the Backend system returns to their idle state. In particular if store is pressed,

the storeRun register is updated to a 1 so that the next cycle of backend operations will include updating the template memory with the input waveform.

The Backend Controller waits in the idle state until startBE is pressed, at which point it initiates a run of the Backend module. For each module it sends a start signal, waits for the module's busy signal to return back to zero, and then moves on to initiate the next module in the cycle. The normal comparison process involves calling storeData, Warp, Accumulate, and then the Compare module. When storeRun is high, the Backend Controller executes the ProgramFlash module between calling storeData and Warp. Thus, the isolated input data with only relevant speech data is stored inside the template memory.

Memory Control Unit (Memorycontroller)

Since there are multiple modules that drive the same RAMs at various times, the Memory Control Module is needed to mux the various signals and assign the appropriate signals to the various RAMs.

The Memory Control Module is not clocked, since all calls to RAM should immediately return values so that the various modules can obtain appropriate input. The memory controller simply takes in the busy status for the various modules. If a certain module is currently busy, then that module's RAM control signals are selected as the control signals for the appropriate RAMs. When the system is in output-to-DAC mode, all RAM are set to read mode, and the address is set to the input DataAdd (RAM access address from DAC_fsm). Thus all RAMs will output their values at DataAdd, a separate

mux (DACspecifics) is implemented to choose the appropriate RAM output depending on which Output signal the user selects.

Isolate Speech Waveform Module (StoreData)

The storeData module accesses the front-end iRAM module which stores the entire sample wavfoerm and stores the input waveform data between beginpt and endpt into icRAM for later use. In this manner, unnecessary noise data is removed and if the system wants to store a new Template, it can just access icRAM.

The storeData module is a 5 state minor FSM. By default it is in the s_Idle state, when it receives a signal from Backendcontroller, the minor FSM transitions to state s_startStore. In startStore, the Busy signal is set high and the address of the inputRAM is set to the beginpt, icRAM is set to write, so that inputRAM[beginpt] can be transferred to icRAM[0]. In the next state, icRAM is set to read, in this manner, when there is an address change, the transition will not cause unintentional writes to icRAM. After writing the value, iRAMadd is incremented by one and the FSM loops to storeWait, where the signal is once again turned high. The system loops until the iRAMadd address is greater than endpt.

Reset Template Module (ProgramFlash)

The module resets the template memory to the isolated input speech waveform in the current run of the Backend.

The programflash module manages its memory controls based on which FlashMode the user has selected. When FlashMode is 0, the memory values are stored an

internal FPGA RAM, when FlashMode is 1, the memory values are stored in an external Flash Memory. Depending on the FlashMode, 2 different sets of states are run by the module.

If the Flash Memory is being used, to write to the Flash Memory requires a series of steps. The chip is first turned on (ce=0), set to write state (we=0), and output is disabled (oe=1). A hex value of 40 is written to the chip to signal program setup. After a wait cycle in which the chip is turned off, the actual value corresponding to the specific Flash memory address is written. The program must wait 10 us for the writing to take place. Afterwards, verification that the value was correctly written takes place. The FSM must wait an additionally 6 us for the verification to take place. Once the FlashData from the bidirectional FlashIO matches the current icRAMoutput (Value that is supposed to be stored to the Flash) the address is incremented for another cycle.

If the internal RAM is being used, the FSM state transitions follow a similar scheme as that of the storeData module.

Warp Data Module

The Warp Data Module implements a version of the Itakura Method for time warping data so that the input signal can better match the template signal. At each template signal, three consecutive addresses of input signal are looked at, and the input address with the smallest distance between input values and template value is chosen as warped value. The FSM states can be grouped into three state groups.

The first group accesses the waveform information, whereby a difference between template value and icRAM sample is stored in a register, while a counter is incremented.

The increment of the counter prepares for the storage of the next difference sample. The counter is incremented twice so that difference values t[m+1]-i[n]; t[m+1]-i[n+1]; t[m+1]-i[n+2] is evaluated. The difference values are evaluated in an external subtract module, the output of that subtract module is then fed into an absolute value module to make all errors positive. The resulting value from the absolute value module is fed back into the warpFSM for distance evaluation. None of the external operation modules are registered, so by the next clock cycle the difference value can be stored in a register in the FSM(rd0, rd1, rd2).

Once those differences have all been stored, the module goes to the path-decision states where the best path choice according to the Itakura constraints is chosen. Choosing i[n] to match t[m+1] translates to 0 time warp, choosing i[n] translates to a 1 unit time warp. The warp value needs to be stored in a register, since no consecutive 0 unit time warp or 2 unit time warp can take place between consecutive samples of the template RAM.

Then the module finishes the process by storing the appropriate icRAM value into wRAM. If the current index value of the wRAM is smaller than the template size, then the warping process continues, otherwise the FSM exits and returns to the idle state.

Choose Template Source Module

Depending on whether or not the system is in FlashMode, the source for accessing template values is different. The chooseinput module simply takes in both the outputs of FlashIO (External Flash Memory) and FlashOutput (Internal RAM), and based on the system's FlashMode status assigns TemplateMem one of the two RAM output values.

Accumulate

The Accumulate module simply adds the magnitude of the error for all template data and their corresponding warped input data point. The memories that store template data and warped data are accessed by using the address 'cntrAdd.' Each cntrAdd corresponds to a time, thus to find error over the entire window, cntrAdd is incremented from 0 to the size of the Template Data. Since the error could be a maximum of 8 bits, then the maximum number of digits for Acc_out after 256 data points is 11 bits.

Compare Module

When the Compare Module is started, it simply takes in the 11 bit output from the Accumulate function. Based on which one of the four thresholds the user chooses, the module compares the Accumulated value to the threshold value. If the value is under the threshold value, then the success bit will hold high for 1 second, otherwise the failure bit will hold high for 1 second.

Table 2: Threshold Values

00	01	10	11
00000000001	000000001111	000001111111	001111111111

DAC Unit

The DAC block is responsible for outputting RAM values (icRAM, wRAM, and tempRAM). When a DAC enable signal is turned on by the user, the DAC unit becomes active and all other processes goes to the Idle state. The unit has two main modules.

DAC_FSM:

The DAC FSM module wakes from the idle state when DACen becomes high. The module takes a size input that is either the size of the Template memory or that of the icRAM memory, depending on which Output mode is chosen. The module then simply increments address and sets dac_cs low to allow for conversion of the appropriate RAM content.

iDacSpecifics:

When the DAC FSM outputs the RAM address, DataAdd, the signal is sent to all three RAM locations by the Memory Controller Unit. To choose which of the outputs should be relayed to the DAC for conversion, the iDacSpecifics module relies on the user selected OutputMode register. The module is simply a mux that implements two operations. Depending on the OutputMode, the module selects the appropriate RAM size so that the DAC_FSM can stop increment RAM address at the appropriate time. The module also selects one of three RAM outputs (icRAM, templateRAM, warpRAM) based on OutputMode and relay the chosen signal to the DAC for D/A conversion.

Table 3: DAC Output

00	01	10	11
Input Cut Data	Warped Data	Template Data	Template Data

Input Ram Selector Module

In the stand-alone system, the Backend receives its input signals from one of four preloaded ROMs. The selector module decides, based on the user chosen "inputselect" signal, which of the four inputRAM outputs is eventually passed on to the Backend system for storage, warping, or comparisons.

Table 1: Input RAM Values

Switch	Start Index	Template Values
11	0	40, 100, 145, 165, 210, 250, 250, 235, 200, 170, 150, 80, 40 ;
10	0	40, 80 , 120, 160, 200, 240, 200, 160, 80, 40;
01	152	227, 0, 250, 250, 243, 220, 19, 255, 255, 180, 0,125, 20;
00	150	227, 250, 250, 243, 243, 19, 255, 250, 0, 125, 20;

End Points Select

The End Points Select Module is similar to the RAM Selector Module, except instead of selecting the appropriate input RAM, the module selects the appropriate begin and endpts for the selected input RAM. Begin point and end points are essential for the storeData module when it attempts to copy relevant input RAM values into icRAM.

Testing/Debugging

Testing the backend involves extensive computer simulations on MAX+plus II as that most of the memory components involved are internal RAM's within the FPGA.

In the design stage, as each module was coded, a .scf file was created to test the functionality of each module. There were relatively few problems in debugging the individual modules. It was when all the modules were connected and it came time to

debug the top-level_be that problems actually arose. During individual module testing, the case testing was often not extensive enough.

Front-End

Since the system is speaker-dependent, the input waveform should be taken in by a microphone for processing. However, due to the limitations of the FPGA memory, only the lower eight-bits of the fourteen-bits is used to calculate the noise magnitude standard deviation, therefore the system requires a signal-to-noise ratio of at least 2^6, which was difficult to achieve using the speaker and audio amplifier system available. To resolve this problem a .mif file that contains an ideal speech waveform was created and an internal ROM was used as input to the Magnitude and Zero-crossing Calculation modules.

Two tests of different waveforms were ran, and both were successful. The waveform is created by looping the noise section, the voiced speech section, and the unvoiced speech section each for a number of times. Therefore, relatively different waveforms can be created just by looping the number of cycles of each section differently and changing the order of when each section is looped. This process is performed by the two Test FSMs created.

There front-end system has two modes: regular operation mode and debugging mode. Under normal circumstances, the ‘speak’ button is use to record the input of the speech waveform, and three switches ‘mag_sw[0:2]’ are used to selected the most significant bits of the magnitude data to be sent for time warping. When in debugging mode, the switch ‘debug’ is high, and the switch ‘debug_zero’ selects which of the two SRAMs, Magnitude and Zero-Crossing, to be tested. When ‘debug_bt’ is pressed, the

output of the speech waveform is outputted by the DAC. Also two .mif files are created for debugging purposes, and they are used to initialize the Magnitude and Zero-Crossing SRAMs so that endpoint calculation process can be observed. Table 4 summerizes the overall external controls to the front-end if the system.

Name	Type	Description
Debug	Switch	0: regular operation mode 1: debugging mode
Debug_Zero	Switch	0: Magnitude SRAM 1: Zero-Crossing SRAM
Debug_bt	Button	To initiate 'DAC' output of the SRAM
Speak	Button	Speech is recorded when high
Mag_sw[0:2]	Switches	To select the most significant bits of the speech magnitude values to be processed.

Table 4: External Front-end Controls

Backend

The first problem that arose with the top level module was timing issues between various RAMs. Since the backend operates on three different RAMs and also operates on a RAM from the front-end, timing problems were frequent. Often, the write signal was not set on the correct stage of the FSM and the result was invalid stored data. As a result, many modules simply became unclocked, allowing immediate return of the signal on the next clock cycle. This design cleared up many timing issues that rose due to accumulation of clock delays from various modules. Streamlining the RAM, mathematical operations modules, and signal select muxes into asynchronous units not only solved the immediate RAM signal problems but also simplified future debugging

problems since there was less trouble in tracing from which state of a FSM a signal was derived from.

Often times, when adding new outputs to the toplevel for viewing in Simulation uncovered new errors during compilation. The compiler doesn't catch the errors until the concerned signal is outputted. In many cases, when a new signal was added, a "missing source" error would arise. Often times the cause was inadequate parameters in some function call of a recently updated function.

Once the top-level simulation compiled, each module in the backend was treated to extensive testing on the Simulator. Only one module and one function was scrutinized at a time. The going was slow but it also minimized debugging once the verilog code was loaded into the FPGA. Debugging once the code is loaded into the FPGA would take more time since it is impossible to look at all the various states on the limited number of ports available in the FPGA.

After the verilog code was loaded into the FPGA, the system was tested by loading the appropriate data, switching the system into DAC mode and then looking at the results stored in the various RAMs. At first, every time the startBE button was pressed, the system loaded a new template, so icRAM and templateRAM values were always the same. Peculiarly the success/ failure LED's simply blinked and did not light up for the full 2 seconds that they were set for. The system was examined in simulation. Inputting signals into icRAM worked fine. It was realized that since all FSMs went back to idle state when DACen or Store is set high, the conditions that the various FSMs were set to were often the same after DACen or Store is hit. However, in one case the backend controller actually should have received different treatment when detecting DACen

versus Store. The backend controller runs ProgramFlash only when it detects a Store signal. However since DACen and Store shared the same if statement, the storeRun register was set high when either DACen or Store was pressed. As a result, every time after DAC mode was turned on, the system replaced its template.

Debugging the Flash Memory was fruitless. Since the internal RAM worked and there were not enough ports to allow for integration with front-end and backend interaction with a Flash Memory module, the development of the feature was put on the back burner. Through testing, it was realized that reading from the Flash worked appropriately. However the system was unable to output a result when placed in store mode. There was a problem writing data to the Flash Memory. When time allows a careful Spectrum Analyzer analysis of the input and output values for the programFlash module will help discover the cause of the problem. Most likely the timing for writing data is not correct since on computer simulation, the module worked fine with manually set values. A closer look at the specification sheet will also be needed in order to debug the problem behind the Flash Memory.

Conclusion

This project involved extensive programming and complex testing of modules. Unlike the other labs, the project required research and an independent, unguided design for the system. The layout for the front-end was complicated by added debugging modules and numerous number of multiplexers for SRAM address and data control. It was difficult to keep track of the signals since there are relatively thirty modules used. Fortunately, the algorithm chosen for endpoint detection is relatively straightforward.

The layout for the backend module changed as the verilog coding and testing took place. The backend module eliminated timing delays by making many modules synchronous. This modification allows for faster computation, if the data is large enough then the computing time could definitely add up. Adding the Memory Control Unit, which was not in the original design, also simplified much of the RAM storage and input/output contentions. It also makes integrating any new modules that requires control of the storage memories much easier. A simple condition would need to be added to the assign statement to provide the new module access to RAM control.

There are still much that could be done to the system. In terms of front-end Functionality, a Pitch Detection Unit can be added to further increase the accuracy of the verification process. The peaks and valleys of the input speech waveform can be taken to calculate its average pitch period at specific time instances, and then the values can be compared to a pitch template using the same process as the one used for magnitude template comparison.

In terms of backend functionality, adding a Flash Memory feature would allow the system to retain the waveform for future use, even after the system has been powered down. But as is, with the internal RAM the system can reset the template at any time. To better improve the performance of the backend, two improvements can be made. The current Itakura warping variation employed by the backend, works wonders for warping input signals that vary slightly from the template, however for better results the full Itakura method can be employed. The Warp module would have to look at all data points originating from current input data, instead of simply two time units away. Finding an efficient way to implement the algorithm in hardware could be fun and challenging.

Secondly, a more mathematical comparator can be used to decide whether the signal is statistically significantly different from the template waveform. The current method simply sums the difference between the warped input and the template waveform over all time. But one can imagine, using standard deviation and square mean error as more accurate ways of evaluating the validity of the input signal.

Front-end Verilog Code

```
module accum_m (clk, sample, clear_sync, speech_in, speech_out, accum_100);
	//accumulator used by the Mag and ZeroX to perform convolution
	//note, speech_in = speech_mag in magnitude module
	input clk, clear_sync, sample;
	input[6:0] speech_in;
	output[13:0] speech_out;
	output accum_100;

	reg[13:0] accum_value, speech_out;
	reg[6:0] count;
	reg accum_100;

	always @ (posedge clk)
	begin
		if (accum_100) accum_100 <= 0; //NEW
		if (clear_sync)
			begin
			speech_out <= 0;
			accum_100 <=0;
			count <= 0;
			accum_value <= 0;
			end
		else if (sample)//add the previous output to sum when there's a new
sample
			begin
			if (count == 10)//100)
				begin
				speech_out <= accum_value + speech_in;//output rewritten
every 100 samples
				accum_100 <= 1;
				count <= 1;
				accum_value <= 0;
				end
			else
				begin
				count <= count + 1;
				accum_value <= accum_value + speech_in;
				accum_100 <= 0;
				end
			end
	end
endmodule
```

```
module accum_shift (clk, clear_sync, sum_sig, nse_read_done, product, sq_value,
sqrt_begin);
	//takes in x^2 and outputs sum(x^2)/n, the term under the sq-rts
	input clk, sum_sig, clear_sync, nse_read_done;
	input[15:0] product;
	output[15:0] sq_value;
	output sqrt_begin;

	reg[15:0] sq_value;
	reg[18:0] accum;
	reg sqrt_begin;

	always @ (posedge clk)
	begin
		if (nse_read_done)
			begin
			sq_value <= accum >> 3;
			sqrt_begin <= 1;
			end
		else if (sqrt_begin) sqrt_begin <= 0;
		if (sum_sig) accum <= accum + product;
		if (clear_sync)
			begin
			accum <= 0;
			sq_value <= 0;
			sqrt_begin <= 0;
			end
	end
endmodule
```

```
module accum_z (clk, sample, clear_sync, speech_in, speech_out, accum_100);
	//used for zeroX convolution
	input clk, clear_sync, sample;
	input speech_in;
	output[6:0] speech_out;
	output accum_100;

	reg[6:0] accum_value, speech_out;
	reg[6:0] count;
	reg accum_100;

	always @ (posedge clk)
	begin
		if (accum_100) accum_100 <= 0;//NEW
		if (clear_sync)
			begin
```

```
                speech_out <= 0;
                accum_100 <=0;
                count <= 0;
                accum_value <= 0;
                end
        else if (sample)
                begin
                        if (count == 10)//100)
                                begin
                                speech_out <= accum_value + speech_in;
                                accum_100 <= 1;
                                count <= 1;
                                accum_value <= 0;
                                end
                        else
                                begin
                                count <= count + 1;
                                accum_value <= accum_value + speech_in;
                                accum_100 <= 0;
                                end
                end
        end
endmodule
```

```
module adc_fsm (clk, sample, status, status_sync, r_wbar,  cbar_a2d, reg_WE);
        //controls the I/O of ADC
        input clk, sample, status, status_sync;
        output r_wbar, cbar_a2d, reg_WE;

        parameter IDLE = 0;
        parameter HOLD = 1;
        parameter CAWLOW = 2;
        parameter CAWHIGH = 3;
        parameter HOLDW = 4;
        parameter COMP = 5;
        parameter CARLOW = 6;
        parameter WEHIGH = 7;
        parameter WELOW = 8;
        parameter CARHIGH = 9;

        reg[3:0] state;
        reg r_wbar, cbar_a2d, reg_WE;

        always @ (posedge clk)
        begin
                case (state)
```

```
IDLE:  //not reading  nor writing
                //when 'sample' is high,  starting converting analog
input
                //'sample' is high as 10khz
                begin
                r_wbar <= 1;
                reg_WE <= 0;
                cbar_a2d <= 1;
                if (sample)
                        begin
                        state <= HOLD;
                        end
                end
HOLD:
        begin
        state <= CAWLOW;
        end
CAWLOW: //write enable
                begin
                r_wbar <= 0;
                cbar_a2d <= 0;
                state <= CAWHIGH;
                end
CAWHIGH: //write disable
                begin
                r_wbar <= 1;
                cbar_a2d <= 1;
                state <= HOLDW;
                end
HOLDW:  //status is computing
                begin
                state <= COMP;
                end
COMP:   //stay in this state till end of computation
                begin
                if (status || status_sync) state <= state;
                else state <= CARLOW;
                end
CARLOW: //reading enable
                begin
                cbar_a2d <= 0;
                state <= WEHIGH;
                end
WEHIGH:  //register enable allow the read data to be stored
                begin
                reg_WE <= 1;
```

```
                        state <= WELOW;
                        end
                WELOW:  // disable reg enable, data latched
                        begin
                        reg_WE <=0;
                        state <= CARHIGH;
                        end
                CARHIGH://disable read
                        begin
                        cbar_a2d <= 1;
                        state <= IDLE;
                        end
                default:
                        begin
                        r_wbar <= 1;
                        reg_WE <= 0;
                        cbar_a2d <= 1;
                        state <= IDLE;
                        end
        endcase
    end
endmodule
```

```
module adr_counter (clk, inc_counter, clear_sync, clear_cnt, input_en, write_done,
input_adr, count_adr);
    //used to control the adr or Mag and ZeroX SRAMs
    input clk, inc_counter, clear_sync, clear_cnt, input_en, write_done;
    input[7:0] input_adr;
    output[8:0] count_adr;

    reg[8:0] count_adr;
    reg read_cycle;
    always @ (posedge clk)
    begin
        if (input_en) count_adr <= input_adr;
        if (write_done) read_cycle <= 1;
        else if (clear_sync)
            begin
            count_adr <= 0;
            read_cycle <= 0;
            end
        else if (clear_cnt || (count_adr[6] && !read_cycle)) count_adr <= 0;
        else if (inc_counter) count_adr <= count_adr + 1;
        end
endmodule
```

```
module dac_fsm (clk, debug, debug_sync, speech_adr, dac_cs);
	input clk, debug, debug_sync;
	output[7:0] speech_adr;
	output dac_cs;
	parameter IDLE = 0;
	parameter HOLD = 1;
	parameter CSLOW = 2;
	parameter CSHIGH =3;
	parameter INCADR = 4;

	reg output_done, dac_cs;
	reg[7:0] speech_adr;
	reg[2:0] state;
	always @ (posedge clk)
	begin
			case (state)
			IDLE:
				begin
				dac_cs <= 1;
				speech_adr <= 0;
				if (debug && debug_sync) state <= HOLD;
				end
			HOLD:
				if (!debug_sync) state <= CSLOW;//wait for all mux to be
in it's proper place
			CSLOW:
				begin
				dac_cs <= 0;
				state <= CSHIGH;
				end
			CSHIGH:
				begin
				dac_cs <= 1;
				state <= INCADR;
				end
			INCADR:
				begin
				if (speech_adr == 255)
					begin
					state <= IDLE;
					end
				else
					begin
					speech_adr <= speech_adr + 1;
					state <= CSLOW;
					end
```

```
                        end
                default: state <= IDLE;
            endcase
        end
endmodule
```

```
module debug_mux (clk, debug, debug_zero, speech_adr_dbg, speech_adr_rw,
mag_output, zero_data,
                        speech_adrz, data_output, mag_sram_adr, zero_sram_adr);
        input clk, debug, debug_zero;
        input[7:0] speech_adr_rw, speech_adr_dbg, mag_output, speech_adrz;
        input[6:0] zero_data;
        output[7:0] mag_sram_adr, zero_sram_adr, data_output;

        reg[7:0] mag_sram_adr;
        reg[7:0] zero_sram_adr;
        reg[7:0] data_output;

        always @ (posedge clk)
        begin
            if (debug)
                begin
                if (debug_zero)
                    begin
                    zero_sram_adr <= speech_adr_dbg;
                    data_output <= zero_data;
                    end
                else
                    begin
                    mag_sram_adr <= speech_adr_dbg;
                    data_output <= mag_output;
                    end
                end
            else
                begin
                zero_sram_adr <= speech_adrz;
                mag_sram_adr <= speech_adr_rw;
                end
        end
endmodule
```

```
module diff (clk, WE, x, sign_ch);
        // |sign(x1)-sign(x2)|
        input clk, WE;
        input[7:0] x;
        output sign_ch;
```

```
	parameter IDLE = 0;
	parameter SHIFT = 1;

	reg mem1, mem2, sign_ch, state;

	always @ (posedge clk)
	begin
		case (state)
		IDLE: //delay
			if (WE) state <= SHIFT;
		SHIFT:
			begin
			mem1 <= x[7];
			mem2 <= mem1;
			sign_ch <= mem1 - mem2;
			state <= IDLE;
			end
		endcase
	end
endmodule
```

```
module divider_10k (clk, speak_sync, sample);
	//sends samples at 10kHz to the ADC
	input clk, speak_sync;
	output sample;

	reg sample;
	reg [8:0] count;
	always @ (posedge clk)
	begin
		if (speak_sync)
		begin
			if (count == 18)//184)
			begin
			sample<= 1;
			count <= 0;
			end
			else
			begin
			sample<= 0;
			count <= count + 1;
			end
		end
	end
endmodule
```

```
module endpt_ctl (clk, thresh_done, itu_done, itl_done, itu_begin, itl_begin, izct_begin);
	//ctl signal to initiate the endpoint detection fsms
	input clk, thresh_done, itu_done, itl_done;
	output itu_begin, itl_begin, izct_begin;

	reg itu_begin, itl_begin, izct_begin;
	always @ (posedge clk)
	begin
		if (thresh_done)
			begin
			itu_begin <= 1;
			itl_begin <= 0;
			izct_begin <= 0;
			end
		else if (itu_done)
			begin
			itu_begin <= 0;
			itl_begin <= 1;
			izct_begin <= 0;
			end
		else if (itl_done)
			begin
			itu_begin <= 0;
			itl_begin <= 0;
			izct_begin <= 1;
			end
		else
			begin
			itu_begin <= 0;
			itl_begin <= 0;
			izct_begin <= 0;
			end
	end
endmodule
```

```
module frontend (clk, debug_zero, speak, status, debug, debug_bt, r_wbar, cbar_a2d,
mag_sw, speech, speech_adrin, data_output, final_enda, final_endb, izct_done, dac_cs,
clear_sync, sample, zero_data, zero_sram_adr, count_adrz, state_itu, state_itl, state_izct,
input_adrz, sub_dataaz, add_subz, end_newa, end_newb, speech_adrz, mag_sram_adr,
mag_data, end_pta, end_ptb, write_donez, write_donem, nse_read_donem, sqrt_donem,
statemm, exceed_num);

	input clk, debug_zero, speak, status, debug, debug_bt;
	input[2:0] mag_sw;
	input[7:0] speech_adrin, speech;
	output[7:0] final_enda, final_endb, data_output;
```

```
output r_wbar, cbar_a2d, izct_done, dac_cs, clear_sync, sample;
//output[13:0] mag_data;
output[3:0] statemm;// statez;
//OBSERVATION
//output[8:0] count_adrm;
output add_subz, write_donez, write_donem, nse_read_donem, sqrt_donem;
output[3:0] state_itu, state_itl, state_izct;
output[6:0] zero_data;
output[7:0] zero_sram_adr, mag_sram_adr, input_adrz, sub_dataaz, end_newa,
end_newb, speech_adrz, end_pta, end_ptb;
output[8:0] count_adrz;
output[13:0] mag_data;
output[4:0] exceed_num;

/*port functions
speak: validation button to allow the input speech to be recorded. the speech is
only sampled when this is high
status: status output from ADC
r_wbar: read_write enable for ADC
cbar_a2d: chip select/chip enable for ADC
mag_sw: external switches to determine which 8 of the 14 bits of the magnitude
data is to be selected
speech_adrin: magnitude sram address control
mag_output: corresonding magnitude SRAM data based on the given address
'speech_adrin'
speech: 8 bit output of the ADC
final_enda: address indicating the begin point of the speech
final_endb: address indicating the end point of the speech
izct_done: one-pulse signal to indicate that endpoints are available
*/

wire speak_sync, status_sync, sample, reg_WE, sign_ch, accum_100m,
accum_100z, inc_counterm, inc_counterz;
wire avg_sigz, avg_sigm, itu_begin, itl_begin, izct_begin, clear_cntm, clear_cntz,
input_enm, input_enz;
wire itu_done, itl_done, izct_done, add_subz, add_subm, sram_cntm, sram_cntz,
nse_cntm, nse_cntz, itl_cnt, itu_cnt;
wire clear_cntnse, clear_cntitu, sqrt_beginm, sqrt_beginz, sqrt_donem,
sqrt_donez, ram_wem, ram_wez;
wire write_donem, mean_sigm, sub_sigm, sum_sigm, nse_read_donem,
write_donez, mean_sigz, sub_sigz, sum_sigz, nse_read_donez;
wire thresh_done, itu_addsub, itl_addsub, izct_cnt, clear_sync, debug_sync;
wire[15:0] nse_avg_mag, sq_valuem, sq_valuez, ITL, mult_outm, mult_outz,
productm, productz;
wire[8:0] count_adrm, count_adrz;
```

```
 wire[7:0] speech_adrm, speech_adrz, input_adrm, input_adrz, end_newa,
 end_newb, end_pta, end_ptb;
 wire[7:0] mult_inm, mult_inz, differencem, differencez, mag_sram_adr,
 zero_sram_adr, speech_adr_dbg, speech_adr_dbgz, speech_adr_dbgm,
 mag_output;
wire[7:0] sp_latched, nse_avg_zero, dev_mag, dev_zero, sub_dataam, sub_dataaz,
itu_dataa, itl_dataa, speech_adr_rw;
 wire[6:0] speech_mag, zero_data, speech_outz;
 wire[17:0] ITU;
 wire[10:0] IZCT;
 wire[13:0] speech_outm, mag_data;
 //wire sram_fsm_init;

 //frontend modules
 sync synchornizer(clk, status, speak, debug_bt, status_sync, speak_sync,
 debug_sync, clear_sync);
 adc_fsm adc_ctl(clk, sample, status, status_sync, r_wbar, cbar_a2d, reg_WE);
 divider_10k divider(clk, speak_sync, sample);
 register register_input(clk, reg_WE, speech, sp_latched);

 //magnitude modules
 magnitude mag(clk, sp_latched, speech_mag);
 accum_m accum_mag(clk, sample, clear_sync, speech_mag, speech_outm,
 accum_100m);
 adr_counter mag_adr(clk, inc_counterm, clear_sync, clear_cntm, input_enm,
 write_donem, input_adrm, count_adrm);

 //assign sram_fsm_init = (count_adrm==9'h1f) & inc_counterm;
 sram_fsm mag_fsm(clk, accum_100m, clear_sync, count_adrm[6], avg_sigm,
 sram_cntm, ram_wem, write_donem, statem); //[8]
 addsub sub_mag(add_subm, sub_dataam, count_adrm[7:0], speech_adrm);
          // if add_subm high, speech_adrm = sub_addaam + countm, otherwise =
sub_addaam - countm
 nse_mag_avg mag_mean(clk, avg_sigm, clear_sync, mag_data, nse_avg_mag,
 mean_sigm);//ok
 subtract_zero dif_mag(clk, sub_sigm, mag_data[7:0], nse_avg_mag[7:0],
 differencem);
 accum_shift sq_valm(clk, clear_sync, sum_sigm, nse_read_donem, productm,
 sq_valuem, sqrt_beginm);
 noise_fsm mag_noise(clk, clear_sync, write_donem, mean_sigm, nse_cntm,
 sub_sigm, sum_sigm, nse_read_donem, clear_cntnse, statemm);
 sqrt_calc sqrtmag(clk, clear_sync, sqrt_beginm, sq_valuem, dev_mag,
 sqrt_donem);
 mult_8bit dif_sqm(differencem, differencem, productm);
 mag_sram magsram(mag_sram_adr, ram_wem, speech_outm, mag_data);
```

```
//zero-crossing modules
diff difference(clk, reg_WE, sp_latched, sign_ch);
accum_z accum_zero(clk, sample, clear_sync, sign_ch, speech_outz,
accum_100z);
adr_counter zero_adr(clk, inc_counterz, clear_sync, clear_cntz, input_enz,
write_donez, input_adrz, count_adrz);
sram_fsm zero_fsm(clk, accum_100z, clear_sync, count_adrz[6], avg_sigz,
sram_cntz, ram_wez, write_donez, statez); ///[8]
addsub sub_zero(add_subz, sub_dataaz, count_adrz[7:0], speech_adrz);
nse_zero_avg zero_mean(clk, avg_sigz, clear_sync, zero_data, nse_avg_zero,
mean_sigz);
subtract_zero dif_zero(clk, sub_sigz, zero_data, nse_avg_zero, differencez);
accum_shift sq_valz(clk, clear_sync, sum_sigz, nse_read_donez, productz,
sq_valuez, sqrt_beginz);
noise_fsm zero_noise(clk, clear_sync, write_donez, mean_sigz, nse_cntz,
sub_sigz, sum_sigz, nse_read_donez, clear_cntz, statezz);
sqrt_calc sqrtzero(clk, clear_sync, sqrt_beginz, sq_valuez, dev_zero, sqrt_donez);
mult_8bit dif_sqz(differencez, differencez, productz);
zero_sram zerosram(zero_sram_adr, ram_wez, speech_outz, zero_data);


//endpoint detection modules
threshold_calc thresh_mag(clk, clear_sync, sqrt_donem, sqrt_donez,
nse_avg_mag, nse_avg_zero, dev_mag, dev_zero, ITU, ITL, IZCT, thresh_done,
state_thr);
itu_fsm itu_ctl(clk, clear_sync, itu_begin, speech_adrm, mag_data, ITU[15:0],
itu_cnt, clear_cntitu, itu_addsub, itu_dataa, end_pta, end_ptb, itu_done, state_itu);
itl_fsm itl_ctl(clk, clear_sync, itl_begin, ITL, end_pta, end_ptb, speech_adrm,
mag_data, end_newa, end_newb, input_enm, itl_cnt, itl_done, itl_addsub,
itl_dataa, input_adrm, state_itl);
izct_fsm izct_ctl(clk, clear_sync, izct_begin, end_newa, end_newb, speech_adrz,
zero_data, IZCT[6:0], final_enda, final_endb, izct_cnt, sub_dataaz, input_enz,
input_adrz, add_subz, izct_done, state_izct, exceed_num);
endpt_ctl endpt_detctl(clk, thresh_done, itu_done, itl_done, itu_begin, itl_begin,
izct_begin);
mag_mux magmux(clk, clear_sync, izct_done, speech_adrm, speech_adrin,
itu_done, itu_addsub, itl_addsub, itu_dataa, itl_dataa, add_subm, sub_dataam,
speech_adr_rw, state_mag);
incm_mux inccntm_mux(clk, clear_sync, sram_cntm, write_donem, nse_cntm,
nse_read_donem, itu_cnt, itu_done, itl_cnt, clear_cntnse, clear_cntitu,
inc_counterm, clear_cntm, state_incm);
incz_mux inccntz_mux(clk, clear_sync, sram_cntz, write_donez, nse_cntz,
nse_read_donez, izct_cnt, inc_counterz);
mag_select magswitch(clk, mag_sw, mag_data, mag_output);
//for debug purposes
debug_mux debugmux (.clk(clk), .debug(debug), .debug_zero(debug_zero),
```

```
        .speech_adr_dbg(speech_adr_dbg), .speech_adr_rw(speech_adr_rw), .mag_outpu
        t(mag_output), .zero_data(zero_data), .speech_adrz(speech_adrz), .data_output(da
        ta_output), .mag_sram_adr(mag_sram_adr), .zero_sram_adr(zero_sram_adr));

        dac_fsm dacfsm (.clk(clk), .debug(debug), .debug_sync(debug_sync),
        .speech_adr(speech_adr_dbg), .dac_cs(dac_cs));

endmodule
-----------------------------------------------------------------------------------------------------

module incm_mux (clk, clear_sync, sram_cntm, write_donem, nse_cntm,
nse_read_donem, itu_cnt, itu_done, itl_cnt, clear_cntnse,
                                        clear_cntitu, inc_counterm, clear_cntm, state);
        input clk, clear_sync, sram_cntm, write_donem, nse_cntm, nse_read_donem,
itu_cnt, itu_done, itl_cnt, clear_cntnse, clear_cntitu;
        output inc_counterm, clear_cntm;
        output[1:0] state;
        parameter SRAMWR = 0;
        parameter NSERD = 1;
        parameter ITU = 2;
        parameter ITL = 3;

        reg inc_counterm, clear_cntm;
        reg[1:0] state;
        always @ (posedge clk)
        begin
                case (state)
                SRAMWR:
                        begin
                        clear_cntm <= clear_cntnse;//adr_counter clear controled by
noise_fsm mag
                        inc_counterm <= sram_cntm;//adr_counter incremented by
mag_fsm
                        if (write_donem)
                                begin
                                inc_counterm <= nse_cntm;//adr_counter incremented by
noise_fsm mag
                                state <= NSERD;
                                end
                        end
                NSERD:
                        begin
                        inc_counterm <= nse_cntm;
                        if (nse_read_donem)
                                begin
                                inc_counterm <= itu_cnt;
```

```
                        clear_cntm <= clear_cntitu;
                        state <= ITU;
                        end
                end
        ITU:
                begin
                inc_counterm <= itu_cnt;
                clear_cntm <= clear_cntitu;
                if (itu_done)
                        begin
                        inc_counterm <= itl_cnt;
                        state <= ITL;
                        end
                end
        ITL:
                begin
                inc_counterm <= itl_cnt;
                if (clear_sync)
                        begin
                        clear_cntm <= clear_cntnse;//adr_counter clear controled
by noise_fsm mag
                   inc_counterm <= sram_cntm;
                        state <= SRAMWR;
                        end
                end
        default: state <= SRAMWR;
        endcase
    end
endmodule
```

```
module incz_mux (clk, clear_sync, sram_cntz, write_donez, nse_cntz, nse_read_donez,
izct_cnt, inc_counterz);
    input clk, clear_sync, sram_cntz, write_donez, nse_cntz, nse_read_donez, izct_cnt;
     output inc_counterz;
     parameter SRAMWR = 0;
     parameter NSERD = 1;
     parameter IZCT = 2;

     reg inc_counterz;
     reg[1:0] state;
     always @ (posedge clk)
     begin
        case (state)
        SRAMWR:
                begin
                inc_counterz <= sram_cntz;//controlled by zero sram
```

```
                    if (write_donez)
                        begin
                        inc_counterz <= nse_cntz; //controlled by zero noise fsm
                        state <= NSERD;
                        end
                    end
                NSERD:
                    begin
                    inc_counterz <= nse_cntz;
                    if (nse_read_donez)
                        begin
                        inc_counterz <= izct_cnt;//IZCT control
                        state <= IZCT;
                        end
                    end
                IZCT:
                    begin
                    inc_counterz <= izct_cnt;
                    if (clear_sync)
                        begin
                        inc_counterz <= sram_cntz;//NEW
                        state <= SRAMWR;
                        end
                    end
                default: state <= SRAMWR;
                endcase
        end
endmodule
```

```
module itl_fsm (clk, clear_sync, itl_begin, ITL, end_pta, end_ptb, mag_adr, mag_data,
end_newa, end_newb,
                        input_en, inc_counter, itl_done, add_sub, sub_dataa,
input_adr, state);
            //find the temp endpoints based on the points given by ITU module
        input clk, itl_begin, clear_sync;
        input[7:0] end_pta, end_ptb, mag_adr;
        input[15:0] mag_data, ITL;
        output[7:0] end_newa, end_newb;
        output input_en, itl_done, add_sub, inc_counter;
        output[7:0] input_adr, sub_dataa;
        output[3:0] state;
        parameter IDLE = 0;
        parameter ENDA = 1;
        parameter HOLD = 2;
        parameter NEW = 3;
        parameter HOLD1 = 4;
```

```
	parameter HOLD2 = 5;
	parameter HOLD3 = 6;
	parameter COMP = 7;
	parameter FIND = 8;
	parameter HOLD4 = 9;
	parameter ENDB = 10;
	parameter DONE = 11;

	reg itl_done, inc_counter, add_sub, input_en;
	reg[7:0] sub_dataa, end_newa, end_newb, input_adr;
	reg[3:0] state;
	always @ (posedge clk)
	begin
		case (state)
		IDLE:
			begin
			itl_done <= 0;
			inc_counter <= 0;
			add_sub <= 0;//subtract
			sub_dataa <= 6'b111111; //8'b11111111;//11111111- adr_counter
			if (itl_begin) state <= ENDA;
			end
		ENDA:
			begin
			input_en <= 1;
			input_adr <= (/*8'b11111111*/6'b111111 - end_pta); // eq counter
value to be subtracted by
			state <= HOLD;                                                    //
11111111 to obtain end_pta, the starting pt
			end
		HOLD:
			begin
			input_en <= 0;
			state <= NEW; //count_adr valid
			end
		NEW:
			begin //speech_adrm valid
			state <= HOLD1;
			end
		HOLD1:
			begin
			state <= HOLD2;//speech_adr_rw valid
			end
		HOLD2:
			begin
			state <= HOLD3;//mag_sram_adr valid
```

```
                end
        HOLD3:
                begin
                state <= COMP;//mag_data stable
                end
        COMP:
                begin
                if (mag_data > ITL) state <= FIND; // if data > ITL, increment
address and compare again
                else
                        begin
                        if (add_sub)//end-point stage
                                begin
                                end_newb <= mag_adr;
                                itl_done <= 1;
                                state <= DONE;
                                end
                        else //begin=point stage
                                begin
                                end_newa <= mag_adr;
                                state <= ENDB;
                                end
                        end
                end
        FIND:
                begin
                inc_counter <= 1;
                state <= HOLD4;
                end
        HOLD4:
                begin
                inc_counter <= 0;//inc_counter high
                state <= HOLD;
                end
        ENDB:
                begin
                add_sub <= 1; //add
                sub_dataa <= 0; //0 + count_adr
                input_en <= 1;
                input_adr <= end_ptb; //start at address end_ptb
                state <= HOLD;
                end
        DONE:
                begin
                itl_done <= 0;
                if (clear_sync)
```

```
                        begin
                        end_newa <= 0;
                        end_newb <= 0;
                        add_sub <= 0;
                        sub_dataa <= 6'b111111;//8'b11111111;
                        state <= IDLE;
                        end
                end
        default: state <= IDLE;
        endcase
    end
endmodule
```

```
module itu_fsm(clk, clear_sync, itu_begin, count_adr, mag_data, ITU, inc_counter,
clear_cnt, add_sub,
                sub_dataa, end_pta, end_ptb, ITU_done, state);
    //ran though the Mag SRAM to find endpoints for region above ITU
    input clk, itu_begin, clear_sync;
    input[7:0] count_adr;
    input[15:0] mag_data, ITU;
    output inc_counter, add_sub, clear_cnt, ITU_done;
    output[7:0] sub_dataa, end_pta, end_ptb;
    output[3:0] state;

    parameter IDLE = 0;
    parameter DATA1 = 1;
    parameter COMPITU = 2;
    parameter INCNT = 3;
    parameter HOLD = 4;
    parameter HOLD1 = 5;
    parameter HOLD2 = 6;
    parameter HOLD3 = 7;
    parameter HOLD4 = 8;
    parameter HOLD5 = 9;
    parameter CLRCNT = 10;
    parameter HOLD6 = 11;
    parameter DONE = 12;

    reg clear_cnt, inc_counter, add_sub, ITU_done;
    reg[7:0] sub_dataa, end_pta, end_ptb;
    reg[15:0] data1, data2, data3, avg_4pts;
    reg[3:0] state;
    always @ (posedge clk)
    begin
        case (state)
        IDLE:
```

```
                begin
                data1 <= 0;
                data2 <= 0;
                data3 <= 0;
                clear_cnt <= 0;
                inc_counter <= 0;
                add_sub <= 1; //add
                sub_dataa <= 0; //0 + count_adr
                ITU_done <= 0;
                if (itu_begin) state <= DATA1;
                end
        DATA1:
                begin
                data3 <= data2;
                data2 <= data1;
                data1 <= mag_data;
                avg_4pts <= (data3 + data2 + data1 + mag_data) >> 2;//previous
data2 used
                state <= COMPITU;
                end
        COMPITU:
                begin
                        if (avg_4pts < ITU) state <= INCNT; // if avg of 4 data pts
is > than ITU, declare the latest pt the threshold
                        else
                        begin
                                if (add_sub)//begin-point finding stage
                                begin
                                end_pta <= count_adr; //declare temp address as
endpoint
                                state <= CLRCNT;
                                end
                                else//end-point finding stage
                                begin
                                end_ptb <= count_adr;
                                ITU_done <= 1;
                                state <= DONE;
                                end
                        end
                end
        INCNT:
                begin
                inc_counter <= 1;
                state <= HOLD;
                end
        HOLD:
```

```
                begin
                inc_counter <= 0;
                state <= HOLD1;//inc_counter valid
                end
        HOLD1:
                begin
                state <= HOLD2;//counter_adr valid
                end
        HOLD2:
                begin
                state <= HOLD3; //speech_adr valid
                end
        HOLD3:
                begin
                state <= HOLD4;//speech_adr_rw valid
                end
        HOLD4:
                begin
                state <= HOLD5;//mag_sram_adr valid
                end
        HOLD5:
                begin
                state <= DATA1;//mag_data valid
                end
        CLRCNT:
                begin
                clear_cnt <= 1;
                add_sub <= 0; //subtract
                sub_dataa <= 6'b111111; //8'b11111111; //11111111 - count_adr =
rev_adr
                data3 <= 0;
                data2 <= 0;
                data1 <= 0;
                state <= HOLD6;
                end
        HOLD6:
                begin
                clear_cnt <= 0;
                state <= HOLD1;//clear_cntm valid
                end
        DONE:
                begin
                ITU_done <= 0;
                if (clear_sync)
                        begin
                        data3 <= 0;
```

```
                                data2 <= 0;
                                data1 <= 0;
                                avg_4pts <= 0;
                                end_pta <= 0;
                                end_ptb <= 0;
                                sub_dataa <= 0;
                                add_sub <= 1;
                                state <= IDLE;
                                end
                        end
                default: state <= IDLE;
                endcase
        end
endmodule
```

```
module izct_fsm (clk, clear_sync, izct_begin, end_newa, end_newb, mag_adr, mag_data,
IZCT, final_enda, final_endb, inc_counter, sub_dataa,
                                input_en, input_adr, add_sub, izct_done, state,
exceed_num);
        //use temp endpoints given by ITL to find final endpoints
        input clk, clear_sync, izct_begin;
        input[7:0] end_newa, end_newb, mag_adr;
        input[6:0] mag_data, IZCT;
        output[7:0] final_enda, final_endb, input_adr, sub_dataa;
        output izct_done, input_en, add_sub, inc_counter;
        output[3:0] state;
        output[4:0] exceed_num;

        parameter IDLE = 0;
        parameter DATA = 1;
        parameter HOLD = 2;
        parameter HOLD1 = 3;
        parameter HOLD2 = 4;
        parameter HOLD3 = 5;
        parameter HOLD4 = 6;
        parameter HOLD5 = 7;
        parameter COMP = 8;
        parameter INCADR = 9;
        parameter HOLD6 = 10;
        parameter CPAR = 11;
        parameter NEWB = 12;
        parameter DONE = 13;

        reg add_sub, input_en, inc_counter, izct_done;
        reg[7:0] sub_dataa, input_adr, final_enda, final_endb;
        reg[4:0] count, exceed_num;
```

```
	reg[3:0] state;
	always @ (posedge clk)
	begin
		case (state)
		IDLE:
			begin
			add_sub <= 1;//add
			sub_dataa <= 0;
			if (izct_begin)
				begin
				add_sub <= 0;//subtract
				sub_dataa <= 6'b111111;//8'b11111111;//11111111 -
count_ adr
				count <= 4;//25;
				state <= DATA;
				end
			end
		DATA:
			begin
			input_en <= 1;
			input_adr <= (/*8'b11111111*/6'b111111 - end_newa); //
end_newa = sub_dataa - input_adr
			state <= HOLD;
			end
		HOLD:
			begin
			input_en <= 0;
			state <= HOLD1;
			end
		HOLD1:
			begin
			state <= HOLD2;//inc_counterm valid
			end
		HOLD2:
			begin
			state <= HOLD3;//counter_adr valid
			end
		HOLD3:
			begin
			state <= HOLD4;//speech_adrm valid
			end
		HOLD4:
			begin
			state <= HOLD5;//speech_adr_rw valid
			end
		HOLD5:
```

```
                begin
                state <= COMP;//mag_sram_adr valid
                end
        COMP:
                begin
                if (count == 0) state <= CPAR; //computed 25 frames
                else if (mag_data > IZCT) //keep track the number of data greater
than IZCT
                        begin
                        exceed_num <= exceed_num + 1;
                        count <= count - 1;
                        state <= INCADR;
                        end
                else
                        begin
                        count <= count - 1;
                        state <= INCADR;
                        end
                end
        INCADR://change address
                begin
                inc_counter <= 1;
                state <= HOLD6;
                end
        HOLD6:
                begin//inc_counterz high
                inc_counter <= 0;
                state <= HOLD;
                end
        CPAR:
                begin
                if (exceed_num >= 2)//3) //if there are more than 3 data values
greater than IZCT
                        begin
                        if (add_sub) //end-point stage
                                begin
                                final_endb <= mag_adr;//new adr set
                                izct_done <= 1;
                                state <= DONE;
                                end
                        else //begin-point stage
                                begin
                                final_enda <= mag_adr;
                                state <= NEWB;
                                end
                        end
```

```
                else
                        begin
                        if (add_sub)//end-point stage
                                begin
                                final_endb <= end_newb;//keep old adr
                                izct_done <= 1;
                                state <= DONE;
                                end
                        else  //begin-point stage
                                begin
                                final_enda <= end_newa;
                                state <= NEWB;
                                end
                        end
                end
        NEWB:
                begin
                add_sub <= 1;
                sub_dataa <= 0;//0 + count_adr
                count <= 4;//25;
                exceed_num <= 0;
                input_en <= 1;
                input_adr <= end_newb; //set end_newb as the new address
                state <= HOLD;
                end
        DONE:
                begin
                izct_done <= 0;
                if (clear_sync)
                        begin
                        add_sub <= 0;
                        sub_dataa <= 6'b111111;//8'b11111111;
                        count <= 4;//25;
                        exceed_num <= 0;
                        final_enda <= 0;
                        final_endb <= 0;
                        input_adr <= 0;
                        state <= IDLE;
                        end
                end
        default: state <= IDLE;
        endcase
    end
endmodule
```

```
module mag_mux (clk, clear_sync, izct_done, speech_adrm, speech_adrin, itu_done,
itu_addsub, itl_addsub, itu_dataa, itl_dataa, add_subm, sub_dataa, speech_adr_rw, state);
	//mux ctl for related magnitude calc
	input clk, clear_sync, itu_done, itl_addsub, itu_addsub, izct_done;
	input[7:0] itu_dataa, itl_dataa, speech_adrm, speech_adrin;
	output[7:0] sub_dataa, speech_adr_rw;
	output add_subm;
	output state;
	parameter IDLE = 0;
	parameter ITU = 1;


	reg muxen, add_subm, state;
	reg[7:0] sub_dataa, speech_adr_rw;
	always @ (posedge clk)
	begin
		case (state)
			IDLE:
				begin
				speech_adr_rw <= speech_adrm;
				sub_dataa <= itu_dataa;
				add_subm <= itu_addsub;
				muxen <= 0;
				if (itu_done)
					begin
					sub_dataa <= itl_dataa;
					add_subm <= itl_addsub;
					state <= ITU;
					end
				end
			ITU:
				begin
				speech_adr_rw <= speech_adrm;
				sub_dataa <= itl_dataa;
				add_subm <= itl_addsub;
				if (clear_sync)
					begin
					speech_adr_rw <= speech_adrm;
					sub_dataa <= itu_dataa;
					add_subm <= itu_addsub;
					muxen <= 0;
					state <= IDLE;
					end
				else if (izct_done) muxen <= 1;
				else if (muxen) speech_adr_rw <= speech_adrin;
				end
```

```
                default: state <= IDLE;
            endcase
        end
endmodule
```

```
module mag_select (clk, mag_sw, mag_data, mag_output);
        //externally selects 8 bits of the mag data
        input clk;
        input[13:0] mag_data;
        input[2:0] mag_sw;
        output[7:0] mag_output;

        reg[7:0] mag_output;
        always @ (posedge clk)
        begin
            case (mag_sw)
            3'b000: mag_output <= mag_data[7:0];
            3'b001: mag_output <= mag_data[8:1];
            3'b010: mag_output <= mag_data[9:2];
            3'b011: mag_output <= mag_data[10:3];
            3'b100: mag_output <= mag_data[11:4];
            3'b101: mag_output <= mag_data[12:5];
            3'b110: mag_output <= mag_data[13:6];
            endcase
        end
endmodule
```

```
module magnitude (clk, speech_in, speech_mag);
        // takes x and finds |x|
        input clk;
        input[7:0] speech_in;
        output[6:0] speech_mag;

        reg[6:0] speech_mag;
        always @ (posedge clk)
        begin
            if (speech_in[7])
                begin
                speech_mag <= (speech_in[6:0] ^ 7'b1111111) +1;

                end
            else
                begin
                speech_mag <= speech_in[6:0];
                end
        end
endmodule
```

```
module noise_fsm (clk, clear_sync, read_begin, mean_sig, inc_counter, sub_sig, sum_sig,
nse_read_done, clear_cnt, state);
	//controls the threshold calc operations
	input clk, clear_sync, read_begin, mean_sig;
	output inc_counter, sub_sig, sum_sig, nse_read_done, clear_cnt;
	output[3:0] state;

	parameter IDLE = 0;
	parameter HOLDCL = 1;
	parameter HOLD = 2;
	parameter HOLD1 = 3;
	parameter HOLD2 = 4;
	parameter HOLD3 = 5;
	parameter SUB = 6;
	parameter MULT = 7;
	parameter SUM = 8;
	parameter HOLD4 = 9;
	parameter DONE = 10;

	reg inc_counter, sub_sig, sum_sig, nse_read_done, mean, read, clear_cnt;
	reg[3:0] count;
	reg[3:0] state;
	always @ (posedge clk)
	begin
		if (read_begin) read <= 1; //latch write-to-ram done
		else if (mean_sig) mean <= 1; //latch mean-is-available
		else case (state)
		IDLE:
			if (read && mean)
				begin
				clear_cnt <= 1; //sram adrress will be 0
				state <= HOLDCL;
				end
		HOLDCL:
			begin
			clear_cnt <= 0;
			state <= HOLD;//clear_cnt valid
			end
		HOLD:
			begin
			//counter_adr valid
			state <= HOLD1;
			end
		HOLD1: //speech_adrm valid
			begin
			state <= HOLD2;
```

```
        end
HOLD2://speech_adr_rw valid
        begin
        state <= HOLD3;
        end
HOLD3://mag_sram_adr valid
        begin
        state <= SUB;
        end
SUB:
        begin
        sub_sig <= 1;
        state <= MULT;
        end
MULT:
        begin
        sub_sig <= 0;
        state <= SUM;
        end
SUM:
        begin
        sum_sig <= 1;
        inc_counter <= 1;
        count <= count + 1;
        state <= HOLD4;
        end
HOLD4:
        begin
        inc_counter <= 0;
        sum_sig <= 0;
        if (count == 4)//8)
                begin
                nse_read_done <= 1;
                state <= DONE;
                end
        else state <= HOLDCL;
        end
DONE:
        begin
        nse_read_done <= 0;
        if (clear_sync)
        begin
        count <= 0;
        read <= 0;
        mean <= 0;
        state <= IDLE;
```

```
                end
                end
        default:
                state <= IDLE;
        endcase
    end
endmodule
```

```
module nse_mag_avg(clk, avg_sig, clear_sync, ram_in, nse_avg_mag, mean_sig);
    //finds the mean of noise mag
    input clk, avg_sig, clear_sync;
    input[15:0] ram_in;
    output[15:0] nse_avg_mag;
    output mean_sig;
    parameter ACCUM = 0;
    parameter DONE = 1;

    reg[18:0] avg_accum;
    reg[15:0] nse_avg_mag;
    reg[3:0] count;
    reg mean_sig, state;

    always @ (posedge clk)
    begin
        case (state)
        ACCUM:
        begin
        if (count == 4)//8)// 8 sums available
            begin
            nse_avg_mag <= avg_accum >> 2;//3; //sum/8
            mean_sig <= 1;
            state <= DONE;
            end
        else if (avg_sig)
            begin
            avg_accum <= avg_accum + ram_in;
            count <= count + 1;
            end
        end
        DONE:
            begin
            mean_sig <= 0;
            if (clear_sync)
                begin
                nse_avg_mag <= 0;
                avg_accum <= 0;
```

```
                        count <= 0;
                        state <= ACCUM;
                        end
                end

        default: state <= ACCUM;
        endcase
    end
endmodule
```

```
module nse_zero_avg(clk, avg_sig, clear_sync, ram_in, nse_avg_zero, mean_sig);
    //finds the mean of noise zeroX rate
    input clk, avg_sig, clear_sync;
    input[7:0] ram_in;
    output[7:0] nse_avg_zero;
    output mean_sig;
    parameter ACCUM = 0;
    parameter DONE = 1;

    reg[10:0] avg_accum;
    reg[7:0] nse_avg_zero;
    reg[3:0] count;
    reg mean_sig, state;

    always @ (posedge clk)
    begin
        case (state)
        ACCUM:
        begin
        if (count == 4)//8)
            begin
            nse_avg_zero <= avg_accum >> 2;//3;
            mean_sig <= 1;
            state <= DONE;
            end
        else if (avg_sig)
            begin
            avg_accum <= avg_accum + ram_in;
            count <= count + 1;
            end
        end
        DONE:
            begin
            mean_sig <= 0;
            if (clear_sync)
                begin
```

```
                                nse_avg_zero <= 0;
                                avg_accum <= 0;
                                count <= 0;
                                state <= ACCUM;
                                end
                        end
                default: state <= ACCUM;
                endcase
        end
endmodule
```

```
module register (clk, WE, D, Q);
        //regular register usd to store output of ADC
        input clk, WE;
        input[7:0] D;
        output[7:0] Q;

        reg[7:0] Q;
        always @ (posedge clk)
        begin
                if (WE) Q <= D;
        end
endmodule
```

```
module speech_adr_mux (clk, debug, izct_done, speech_adrm, speech_adrin,
speech_adr_dbgm, mag_sram_adr);
        //when endpoints are found, input adr speech_adrin controls the mag SRAM adr
        input clk, debug, izct_done;
        input[7:0] speech_adrm, speech_adrin, speech_adr_dbgm;
        output[7:0] mag_sram_adr;

        reg adr_en;
        reg[7:0] mag_sram_adr;
        always @ (posedge clk)
        begin
                if (izct_done)
                begin
                adr_en <= 1;
                mag_sram_adr <= speech_adrin;
                end
                if (debug) mag_sram_adr <= speech_adr_dbgm;
                else if (adr_en) mag_sram_adr <= speech_adrin;
                else mag_sram_adr <= speech_adrm;
        end
endmodule
```

```
module sqrt (clk, clear_sync, sqrt_begin, input_sq, outputsq, mult_out, mult_in,
sqrt_done);
	input clk, clear_sync, sqrt_begin;
	input[15:0] input_sq, mult_out;
	output[7:0] outputsq, mult_in;
	output sqrt_done;
	parameter IDLE = 0;
	parameter CPT7 = 1;
	parameter HOLD7 = 2;
	parameter CMP7 = 3;
	parameter CPT6 = 4;
	parameter HOLD6 = 5;
	parameter CMP6 = 6;
	parameter CPT5 = 7;
	parameter HOLD5 = 8;
	parameter CMP5 = 9;
	parameter CPT4 = 10;
	parameter HOLD4 = 11;
	parameter CMP4 = 12;
	parameter CPT3 = 13;
	parameter HOLD3 = 14;
	parameter CMP3 = 15;
	parameter CPT2 = 16;
	parameter HOLD2 = 17;
	parameter CMP2 = 18;
	parameter CPT1 = 19;
	parameter HOLD1 = 20;
	parameter CMP1 = 21;
	parameter CPT0 = 22;
	parameter HOLD0 = 23;
	parameter CMP0 = 24;
	parameter DONE = 25;

	reg[4:0] state;
	reg[7:0] mult_in, outputsq;
	reg sqrt_done;
	always @ (posedge clk)
	begin
		case (state)
		IDLE:
			begin
			mult_in <= 0;
			outputsq <= 0;
			if (sqrt_begin) state <= CPT7;
			end
		CPT7:
```

```
                begin
                mult_in <= 8'b10000000; //input to the mult
                state <= HOLD7;
                end
        HOLD7:  state <= CMP7;
        CMP7:
                begin
                if (mult_out <= input_sq) outputsq[7] <= 1; //compare mult output
with actual input
                else outputsq[7] <= 0;
                state <= CPT6;
                end
        CPT6:
                begin
                mult_in <= {outputsq[7], 7'b1000000};
                state <= HOLD6;
                end
        HOLD6:  state <= CMP6;
        CMP6:
                begin
                if (mult_out <= input_sq) outputsq[6] <= 1;
                else outputsq[6] <= 0;
                state <= CPT5;
                end
        CPT5:
                begin
                mult_in <={outputsq[7:6], 6'b100000};
                state <= HOLD5;
                end
        HOLD5:  state <= CMP5;
        CMP5:
                begin
                if (mult_out <= input_sq) outputsq[5] <= 1;
                else outputsq[5] <= 0;
                state <= CPT4;
                end
        CPT4:
                begin
                mult_in <={outputsq[7:5], 5'b10000};
                state <= HOLD4;
                end
        HOLD4:  state <= CMP4;
        CMP4:
                begin
                if (mult_out <= input_sq) outputsq[4] <= 1;
                else outputsq[4] <= 0;
```

```
        state <= CPT3;
        end
CPT3:
        begin
        mult_in <={outputsq[7:4], 4'b1000};
        state <= HOLD3;
        end
HOLD3:      state <= CMP3;
CMP3:
        begin
        if (mult_out <= input_sq) outputsq[3] <= 1;
        else outputsq[3] <= 0;
        state <= CPT2;
        end
CPT2:
        begin
        mult_in <={outputsq[7:3], 3'b100};
        state <= HOLD2;
        end
HOLD2:      state <= CMP2;
CMP2:
        begin
        if (mult_out <= input_sq) outputsq[2] <= 1;
        else outputsq[2] <= 0;
        state <= CPT1;
        end
CPT1:
        begin
        mult_in <={outputsq[7:2], 2'b10};
        state <= HOLD1;
        end
HOLD1:      state <= CMP1;
CMP1:
        begin
        if (mult_out <= input_sq) outputsq[1] <= 1;
        else outputsq[1] <= 0;
        state <= CPT0;
        end
CPT0:
        begin
        mult_in <={outputsq[7:1], 1'b1};
        state <= HOLD0;
        end
HOLD0:      state <= CMP0;
CMP0:
        begin
```

```
                    if (mult_out <= input_sq) outputsq[0] <= 1;
                    else
                            begin
                            outputsq[0] <= 0;
                            sqrt_done <= 1;
                            state <= DONE;
                            end
                    end
            DONE:
                    begin
                    sqrt_done <= 0;
                    if (clear_sync)
                            begin
                            mult_in <= 0;
                            outputsq <= 0;
                            state <= IDLE;
                            end
                    end
            default: state <= IDLE;
            endcase
    end
endmodule
```

```
module sqrt_calc (clk, clear_sync, sqrt_begin, sq_value, dev, sqrt_done);
    input clk, sqrt_begin, clear_sync;
    input[15:0] sq_value;
    output[7:0] dev;
    output sqrt_done;
    wire[15:0] mult_out;
    wire[7:0] mult_in;

    sqrt sqrt_m(clk, clear_sync, sqrt_begin, sq_value, dev, mult_out, mult_in,
sqrt_done);
    mult_8bit mult8bitm(mult_in, mult_in, mult_out);

    endmodule
```

```
module sram_fsm(clk, accum, clear_sync, adr_full, avg_sig, inc_counter, ram_we,
write_done, state);
    //writes data to Mag and ZeroX SRAMS
    input clk, accum, clear_sync, adr_full;
    output inc_counter, ram_we, avg_sig, write_done;
    output[2:0] state;

    parameter IDLE = 0;
    parameter RAMWE = 1;
```

```
	parameter RAMWD = 2;
	parameter INCCNT = 3;
	parameter DONE = 4;

	reg inc_counter, ram_we, avg_sig, write_done;
	reg[2:0] state;

	always @ (posedge clk)
	begin
		case (state)
		IDLE:
			begin
			inc_counter <= 0;
			if (!adr_full) // if it's not the last adr
				begin
				if (accum) state <= RAMWE; //new sum available
				end
			else
				begin //adr pointer points to the last ram adr. send a pulse
for indication
				write_done <= 1;
				state <= DONE;
				end
			end
		RAMWE: //ram write enable
			begin
			ram_we <= 1;
			state <= RAMWD;
			end
		RAMWD://ran write disable, ram data availabe, enable store for avg calc
			begin
			ram_we <= 0;
			avg_sig <= 1;
			state <= INCCNT;
			end
		INCCNT: //disanable store for avg calc, inc_counter
			begin
			avg_sig <= 0;
			inc_counter <= 1;
			state <= IDLE;
			end
		DONE:
			begin
			write_done <= 0;
			if (clear_sync) state <= IDLE;
			end
```

```
                default: state <= IDLE;
                endcase
        end
endmodule
```

```
module subtract_mag(clk, sub_sig, ram_data, mean, difference);
        //mag - mean of mag
        input clk, sub_sig;
        input[15:0] ram_data, mean;
        output[15:0] difference;

        reg[15:0] difference;
        always @ (posedge clk)
        begin
                if (sub_sig)
                        begin
                        difference <= ram_data - mean;
                        end
        end
endmodule
```

```
module subtract_zero(clk, sub_sig, ram_data, mean, difference);
        //zerox - mean of zerox,
        //to be sent into the multiplier by MagaWizard
        input clk, sub_sig;
        input[7:0] ram_data, mean;
        output[7:0] difference;

        reg[7:0] difference;
        always @ (posedge clk)
        begin
                if (sub_sig)
                        begin
                        difference <= ram_data - mean;
                        end
        end
endmodule
```

```
module sync (clk, status, speak, debug_bt, status_sync, speak_sync, debug_sync,
clear_sync);
        //sync speaker and stutus signals
        input clk, status, speak, debug_bt;
        output status_sync, speak_sync, debug_sync, clear_sync;

        reg done, status1, status_sync, speak1, speak_sync, debug1, debug_sync,
clear_sync;
```

```
	always @ (posedge clk)
	begin
		status1 <= status;
		status_sync <= status1;
		speak1 <= speak;
		speak_sync <= speak1;
		debug1 <= debug_bt;
		debug_sync <= debug1;

		if (speak && !done)
			begin
			clear_sync <= 1;
			done <= 1;
			end
		else if (done) clear_sync <= 0;
		end
endmodule
```

```
module sync (clk, status, speak, debug_bt, status_sync, speak_sync, debug_sync,
clear_sync);
	//sync speaker and stutus signals
	input clk, status, speak, debug_bt;
	output status_sync, speak_sync, debug_sync, clear_sync;

	reg done, status1, status_sync, speak1, speak_sync, debug1, debug_sync,
clear_sync;
	always @ (posedge clk)
	begin
		status1 <= status;
		status_sync <= status1;
		speak1 <= speak;
		speak_sync <= speak1;
		debug1 <= debug_bt;
		debug_sync <= debug1;

		if (speak && !done)
			begin
			clear_sync <= 1;
			done <= 1;
			end
		else if (!speak) done <= 0;
		else if (done) clear_sync <= 0;
				end
endmodule
```

```
module test (clk, debug_zero, speak, debug, debug_bt, mag_sw, speech_adrin,
```

```
                                final_enda, final_endb, izct_done, data_output, dac_cs,
zero_data,
                                zero_sram_adr, count_adrz, state_itu, state_itl, state_izct,
input_adrz,
                                sub_dataaz, add_subz, end_newa, end_newb, speech_adrz,
mag_sram_adr, mag_data, end_pta, end_ptb,
                                write_donez, write_donem, rom_adr, rom_data,
nse_read_donem, sqrt_donem, statemm, exceed_num);
        input clk, speak, debug_zero, debug, debug_bt;
        input[7:0] speech_adrin;
        input[2:0] mag_sw;
        output[7:0] final_enda, final_endb, data_output;
        output izct_done, dac_cs;
        output[7:0] rom_data, rom_adr;
        output[7:0] mag_sram_adr;
        //output sample;
        output[13:0] mag_data;

        //for observation
        output add_subz, write_donez, write_donem, nse_read_donem, sqrt_donem;
        output[3:0] state_itu, state_itl, state_izct, statemm;
        output[6:0] zero_data;
        output[7:0] zero_sram_adr, input_adrz, sub_dataaz, end_newa, end_newb,
speech_adrz, end_pta, end_ptb;
        output[8:0] count_adrz;
        output[4:0] exceed_num;

        wire[7:0] rom_adr, rom_data;
        wire sample;

        test_fsm testfsm(clk, sample, clear_sync, rom_adr); );//result: (00, 33)

       frontend frontendmodule(clk, debug_zero, speak, status, debug, debug_bt, r_wbar,
cbar_a2d, mag_sw, rom_data,
                                speech_adrin, data_output, final_enda, final_endb,
izct_done, dac_cs, clear_sync,
                                sample, zero_data, zero_sram_adr, count_adrz, state_itu,
state_itl, state_izct, input_adrz,
                                sub_dataaz, add_subz, end_newa, end_newb, speech_adrz,
mag_sram_adr, mag_data, end_pta, end_ptb,
                                write_donez, write_donem, nse_read_donem, sqrt_donem,
statemm, exceed_num);

        testmag testmagrom (rom_adr, rom_data);

        assign status = 0;
```

```
endmodule
```

```
module test_fsm (clk, sample, clear_sync, rom_adr);
	input clk, sample, clear_sync;
	output[7:0] rom_adr;

	parameter NOISE = 0;
	parameter MAGDATA= 1;
	parameter ZERO = 2;
	parameter NSEAG = 3;

	reg[7:0] rom_adr;
	reg[8:0] count;
	reg[1:0] state;
	always @ (posedge clk)
	begin
		if (clear_sync)
			begin
			count <= 0;
			rom_adr <= 0;
			end
		else if (sample)
		begin
			case (state)
			NOISE:
				begin
				if ((rom_adr == 7) && (count == 6))//500)) //loop address
0-7 500 times
					begin
					count <= 0;
					rom_adr <= rom_adr + 1;
					state <= MAGDATA;
					end
				else if (rom_adr == 7)
					begin
					rom_adr <= 0;
					count  <= count + 1;
					end
				else rom_adr <= rom_adr + 1;
				end
			MAGDATA:
				begin
				if ((rom_adr == 209) && (count == 1))//80))//loop address
8-209 80 times
					begin
					count <= 0;
```

```
                                        rom_adr <= rom_adr + 1;
                                        //rom_adr <= 210;
                                        state <= ZERO;
                                        end
                                else if (rom_adr == 209)
                                        begin
                                        rom_adr <= 8;
                                        count <= count + 1;
                                        end
                                else rom_adr <= rom_adr + 1;
                                end
                        ZERO:
                                begin
                                        if ((rom_adr == 211) && (count ==
20))//1000))//loop 210-211 1000 times
                                        begin
                                        count <= 0;
                                        rom_adr <= rom_adr + 1;
                                        rom_adr <= 0;
                                        state <= NSEAG;
                                        end
                                else if (rom_adr == 211)
                                        begin
                                        rom_adr <= 210;
                                        count <= count + 1;
                                        end
                                else rom_adr <= rom_adr + 1;
                                end
                        NSEAG:
                                begin
                                if ((rom_adr == 7) && (count == 4))//380)) //loop address
0-7 380 times to fill
                                        begin
        //up the rest of the inputs needed for the srams
                                        end
                                else if (rom_adr == 7)
                                        begin
                                        rom_adr <= 0;
                                        count  <= count + 1;
                                        end
                                else rom_adr <= rom_adr + 1;
                                end
                        endcase
                end
        end
```

```
endmodule
```

```
module test1 (clk, debug_zero, speak, debug, debug_bt, mag_sw, speech_adrin,
                                final_enda, final_endb, izct_done, data_output, dac_cs,
zero_data,
                                zero_sram_adr, count_adrz, state_itu, state_itl, state_izct,
input_adrz,
                                sub_dataaz, add_subz, end_newa, end_newb, speech_adrz,
mag_sram_adr, mag_data, end_pta, end_ptb,
                                write_donez, write_donem, rom_adr, rom_data,
nse_read_donem, sqrt_donem, statemm, exceed_num);
        input clk, speak, debug_zero, debug, debug_bt;
        input[7:0] speech_adrin;
        input[2:0] mag_sw;
        output[7:0] final_enda, final_endb, data_output;
        output izct_done, dac_cs;
        output[7:0] rom_data, rom_adr;
        output[7:0] mag_sram_adr;
        //output sample;
        output[13:0] mag_data;

        //for observation
        output add_subz, write_donez, write_donem, nse_read_donem, sqrt_donem;
        output[3:0] state_itu, state_itl, state_izct, statemm;
        output[6:0] zero_data;
        output[7:0] zero_sram_adr, input_adrz, sub_dataaz, end_newa, end_newb,
speech_adrz, end_pta, end_ptb;
        output[8:0] count_adrz;
        output[4:0] exceed_num;

        wire[7:0] rom_adr, rom_data;
        wire sample;

        test1_fsm test1fsm(clk, sample, clear_sync, rom_adr);//result: (0B, 2C)
       frontend frontendmodule(clk, debug_zero, speak, status, debug, debug_bt, r_wbar,
cbar_a2d, mag_sw, rom_data,
                                speech_adrin, data_output, final_enda, final_endb,
izct_done, dac_cs, clear_sync,
                                sample, zero_data, zero_sram_adr, count_adrz, state_itu,
state_itl, state_izct, input_adrz,
                                sub_dataaz, add_subz, end_newa, end_newb, speech_adrz,
mag_sram_adr, mag_data, end_pta, end_ptb,
                                write_donez, write_donem, nse_read_donem, sqrt_donem,
statemm, exceed_num);

        testmag testmagrom (rom_adr, rom_data);
```

```
        assign status = 0;
endmodule
```

```
module test1_fsm (clk, sample, clear_sync, rom_adr);
        input clk, sample, clear_sync;
        output[7:0] rom_adr;

        parameter NOISE = 0;
        parameter MAGDATA= 1;
        parameter ZERO = 2;
        parameter NSEAG = 3;

        reg[7:0] rom_adr;
        reg[8:0] count;
        reg[1:0] state;
        always @ (posedge clk)
        begin
                if (clear_sync)
                        begin
                        count <= 0;
                        rom_adr <= 0;
                        end
                else if (sample)
                begin
                        case (state)
                        NOISE:
                                begin
                                if ((rom_adr == 7) && (count == 20))//500)) //loop address
0-7 500 times
                                        begin
                                        count <= 0;
                                        rom_adr <= rom_adr + 1;
                                        state <= MAGDATA;
                                        end
                                else if (rom_adr == 7)
                                        begin
                                        rom_adr <= 0;
                                        count  <= count + 1;
                                        end
                                else rom_adr <= rom_adr + 1;
                                end
                        MAGDATA:
                                begin
```

```
                        if ((rom_adr == 100) && (count == 1))//80))//loop address
8-209 80 times
                                begin
                                count <= 0;
                                rom_adr <= rom_adr + 1;
                                rom_adr <= 210;
                                state <= ZERO;
                                end
                        else if (rom_adr == 100)
                                begin
                                rom_adr <= 8;
                                count <= count + 1;
                                end
                        else rom_adr <= rom_adr + 1;
                        end
                ZERO:
                        begin
                                if ((rom_adr == 211) && (count ==
20))//1000))//loop 210-211 1000 times
                                begin
                                count <= 0;
                                rom_adr <= rom_adr + 1;
                                rom_adr <= 0;
                                state <= NSEAG;
                                end
                        else if (rom_adr == 211)
                                begin
                                rom_adr <= 210;
                                count <= count + 1;
                                end
                        else rom_adr <= rom_adr + 1;
                        end
                NSEAG:
                        begin
                        if ((rom_adr == 7) && (count == 10))//380)) //loop address
0-7 380 times to fill
                                begin
        //up the rest of the inputs needed for the srams
                                end
                        else if (rom_adr == 7)
                                begin
                                rom_adr <= 0;
                                count  <= count + 1;
                                end
                        else rom_adr <= rom_adr + 1;
                        end
```

```
                endcase
            end
        end

endmodule
```

```
module threshold_calc (clk, clear_sync, sqrt_donem, sqrt_donez, mag_avg, zero_avg,
mag_dev, zero_dev,
                        ITU, ITL, IZCT, thresh_done, state);
    //use mean and dev values to find the thresholds
    input clk, clear_sync, sqrt_donem, sqrt_donez;
    input[15:0] mag_avg;
    input[7:0] zero_avg, mag_dev, zero_dev;
    output[17:0] ITU;
    output[15:0] ITL;
    output[10:0] IZCT;
    output thresh_done;
    output[1:0] state;
    parameter IDLE = 0;
    parameter FIRST = 1;
    parameter SEC = 2;
    parameter THRD = 3;

    reg[15:0] ITL;
    reg[10:0] IZCT;
    wire[10:0] mag_prod, zero_prod;
    reg[1:0] state;
    reg thresh_done, sqrtm, sqrtz;

    mag_mult mult_mag(mag_avg, 2, ITU); //17-bit output
    dev_mult dev_mag(mag_dev, 3, mag_prod); //11-bit output
    dev_mult dev_zero(zero_dev, 3, zero_prod); //11-bit output

    always @ (posedge clk)
    begin
        if (sqrt_donem) sqrtm <= 1;
        if (sqrt_donez) sqrtz <= 1;
        else case (state)
        IDLE:
            begin
            if (sqrtm && sqrtz) // if both dev available, then compute ITL and
IZCT
                begin
                sqrtm <= 0;
                sqrtz <= 0;
                state <= FIRST;
```

```
                        end
                end
        FIRST:
                begin
                ITL <= mag_prod + mag_avg;
                IZCT <= zero_prod + zero_avg;
                state <= SEC;
                end
        SEC:
                begin
                thresh_done <= 1;
                state <= THRD;
                end
        THRD:
                begin
                thresh_done <= 0;
                if (clear_sync)
                        begin
                        sqrtm <= 0;
                        sqrtz <= 0;
                        ITL <= 0;
                        IZCT <= 0;
                        state <= IDLE;
                        end
                end
        default: state <= IDLE;
        endcase
    end
endmodule
```

```
module zero_adr_mux (clk, debug, zero_wr_adr, speech_adrinz, speech_adrz);
    input clk, debug;
    input[7:0] zero_wr_adr, speech_adrinz;
    output[7:0] speech_adrz;

    reg switch;
    reg[7:0] speech_adrz;
    always @ (posedge clk)
    begin
        if (debug)
        begin
            speech_adrz <= speech_adrinz;
        end
        else speech_adrz <= zero_wr_adr;
    end
endmodule
```

```
% ROM inputs%

WIDTH = 8;  % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE
%
DEPTH = 256;  % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL
VALUE %


ADDRESS_RADIX = DEC;  % Address and data radixes are optional, default is hex %

DATA_RADIX = DEC;     % Valid radixes = BIN,DEC,HEX or OCT  %


CONTENT BEGIN
        0       :       1       ;
        1       :       1       ;
        2       :       0       ;
        3       :       0       ;
        4       :       255     ;
        5       :       255     ;
        6       :       0       ;
        7       :       0       ;
        8       :       30      ;
        9       :       33      ;
        10      :       37      ;
        11      :       40      ;
        12      :       44      ;
        13      :       47      ;
        14      :       51      ;
        15      :       54      ;
        16      :       58      ;
        17      :       61      ;
        18      :       64      ;
        19      :       67      ;
        20      :       70      ;
        21      :       73      ;
        22      :       76      ;
        23      :       79      ;
        24      :       82      ;
        25      :       85      ;
        26      :       87      ;
        27      :       90      ;
        28      :       92      ;
        29      :       94      ;
        30      :       97      ;
        31      :       99      ;
```

32 : 101 ;
33 : 103 ;
34 : 105 ;
35 : 107 ;
36 : 108 ;
37 : 110 ;
38 : 111 ;
39 : 113 ;
40 : 114 ;
41 : 115 ;
42 : 116 ;
43 : 117 ;
44 : 118 ;
45 : 118 ;
46 : 119 ;
47 : 119 ;
48 : 120 ;
49 : 120 ;
50 : 120 ;
51 : 120 ;
52 : 120 ;
53 : 120 ;
54 : 119 ;
55 : 119 ;
56 : 118 ;
57 : 117 ;
58 : 117 ;
59 : 116 ;
60 : 114 ;
61 : 113 ;
62 : 112 ;
63 : 111 ;
64 : 109 ;
65 : 108 ;
66 : 106 ;
67 : 104 ;
68 : 102 ;
69 : 100 ;
70 : 98 ;
71 : 96 ;
72 : 93 ;
73 : 91 ;
74 : 88 ;
75 : 86 ;
76 : 83 ;
77 : 81 ;

78 : 78 ;
79 : 75 ;
80 : 72 ;
81 : 69 ;
82 : 66 ;
83 : 63 ;
84 : 59 ;
85 : 56 ;
86 : 53 ;
87 : 49 ;
88 : 46 ;
89 : 42 ;
90 : 39 ;
91 : 35 ;
92 : 32 ;
93 : 28 ;
94 : 24 ;
95 : 21 ;
96 : 17 ;
97 : 13 ;
98 : 9 ;
99 : 6 ;
100 : 2 ;
101 : 254 ;
102 : 250 ;
103 : 247 ;
104 : 243 ;
105 : 239 ;
106 : 236 ;
107 : 232 ;
108 : 228 ;
109 : 225 ;
110 : 221 ;
111 : 217 ;
112 : 214 ;
113 : 210 ;
114 : 207 ;
115 : 204 ;
116 : 200 ;
117 : 197 ;
118 : 194 ;
119 : 191 ;
120 : 187 ;
121 : 184 ;
122 : 181 ;
123 : 178 ;

124 : 176 ;
125 : 173 ;
126 : 170 ;
127 : 168 ;
128 : 165 ;
129 : 163 ;
130 : 160 ;
131 : 158 ;
132 : 156 ;
133 : 154 ;
134 : 152 ;
135 : 150 ;
136 : 149 ;
137 : 147 ;
138 : 145 ;
139 : 144 ;
140 : 143 ;
141 : 142 ;
142 : 141 ;
143 : 140 ;
144 : 139 ;
145 : 138 ;
146 : 137 ;
147 : 137 ;
148 : 136 ;
149 : 136 ;
150 : 136 ;
151 : 136 ;
152 : 136 ;
153 : 136 ;
154 : 137 ;
155 : 137 ;
156 : 138 ;
157 : 138 ;
158 : 139 ;
159 : 140 ;
160 : 141 ;
161 : 142 ;
162 : 143 ;
163 : 145 ;
164 : 146 ;
165 : 148 ;
166 : 149 ;
167 : 151 ;
168 : 153 ;
169 : 155 ;

170 : 157 ;
171 : 159 ;
172 : 161 ;
173 : 164 ;
174 : 166 ;
175 : 169 ;
176 : 171 ;
177 : 174 ;
178 : 177 ;
179 : 180 ;
180 : 183 ;
181 : 186 ;
182 : 189 ;
183 : 192 ;
184 : 195 ;
185 : 198 ;
186 : 202 ;
187 : 205 ;
188 : 208 ;
189 : 212 ;
190 : 215 ;
191 : 219 ;
192 : 222 ;
193 : 226 ;
194 : 230 ;
195 : 233 ;
196 : 237 ;
197 : 241 ;
198 : 245 ;
199 : 248 ;
200 : 252 ;
201 : 256 ;
202 : 4 ;
203 : 7 ;
204 : 11 ;
205 : 15 ;
206 : 18 ;
207 : 22 ;
208 : 26 ;
209 : 29 ;
210 : 2 ;
211 : 254 ;
212 : 0 ;
213 : 0 ;
214 : 0 ;
215 : 0 ;

```
        216     :       0       ;
        217     :       0       ;
        218     :       0       ;
        219     :       0       ;
        220     :       0       ;
        221     :       0       ;
        222     :       0       ;
        223     :       0       ;
        224     :       0       ;
        225     :       0       ;
        226     :       0       ;
        227     :       0       ;
        228     :       0       ;
        229     :       0       ;
        230     :       0       ;
        231     :       0       ;
        232     :       0       ;
        233     :       0       ;
        234     :       0       ;
        235     :       0       ;
        236     :       0       ;
        237     :       0       ;
        238     :       0       ;
        239     :       0       ;
        254     :       0       ;
        255     :       0       ;

END;
```

```
% Magnitude SRAM inputs %
WIDTH = 14;  % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE
%
DEPTH = 256;  % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL
VALUE %


ADDRESS_RADIX = DEC;  % Address and data radixes are optional, default is hex %

DATA_RADIX = DEC;     % Valid radixes = BIN,DEC,HEX or OCT  %


CONTENT BEGIN
        0       :       100     ;
        1       :       105     ;
        2       :       110     ;
        3       :       114     ;
```

4 : 118 ;
5 : 121 ;
6 : 123 ;
7 : 125 ;
8 : 125 ;
9 : 124 ;
10 : 123 ;
11 : 120 ;
12 : 117 ;
13 : 113 ;
14 : 108 ;
15 : 104 ;
16 : 99 ;
17 : 94 ;
18 : 89 ;
19 : 85 ;
20 : 81 ;
21 : 78 ;
22 : 76 ;
23 : 75 ;
24 : 75 ;
25 : 76 ;
26 : 78 ;
27 : 81 ;
28 : 84 ;
29 : 88 ;
30 : 93 ;
31 : 98 ;
32 : 103 ;
33 : 108 ;
34 : 112 ;
35 : 116 ;
36 : 120 ;
37 : 122 ;
38 : 124 ;
39 : 125 ;
40 : 125 ;
41 : 124 ;
42 : 121 ;
43 : 118 ;
44 : 115 ;
45 : 110 ;
46 : 106 ;
47 : 101 ;
48 : 96 ;
49 : 91 ;

50 : 116 ;
51 : 348 ;
52 : 580 ;
53 : 810 ;
54 : 1039 ;
55 : 1267 ;
56 : 1493 ;
57 : 1716 ;
58 : 1937 ;
59 : 2156 ;
60 : 2372 ;
61 : 2584 ;
62 : 2794 ;
63 : 3000 ;
64 : 3202 ;
65 : 3400 ;
66 : 3593 ;
67 : 3783 ;
68 : 3968 ;
69 : 4147 ;
70 : 4322 ;
71 : 4492 ;
72 : 4656 ;
73 : 4815 ;
74 : 4967 ;
75 : 5114 ;
76 : 5255 ;
77 : 5389 ;
78 : 5517 ;
79 : 5638 ;
80 : 5753 ;
81 : 5861 ;
82 : 5962 ;
83 : 6056 ;
84 : 6142 ;
85 : 6221 ;
86 : 6293 ;
87 : 6358 ;
88 : 6415 ;
89 : 6465 ;
90 : 6506 ;
91 : 6541 ;
92 : 6567 ;
93 : 6586 ;
94 : 6597 ;
95 : 6600 ;

96 : 6595 ;
97 : 6583 ;
98 : 6563 ;
99 : 6535 ;
100 : 6500 ;
101 : 6456 ;
102 : 6405 ;
103 : 6347 ;
104 : 6281 ;
105 : 6208 ;
106 : 6127 ;
107 : 6039 ;
108 : 5944 ;
109 : 5842 ;
110 : 5733 ;
111 : 5617 ;
112 : 5495 ;
113 : 5366 ;
114 : 5230 ;
115 : 5088 ;
116 : 4940 ;
117 : 4787 ;
118 : 4627 ;
119 : 4462 ;
120 : 4291 ;
121 : 4116 ;
122 : 3935 ;
123 : 3749 ;
124 : 3559 ;
125 : 3365 ;
126 : 3166 ;
127 : 2963 ;
128 : 2757 ;
129 : 2547 ;
130 : 2333 ;
131 : 2117 ;
132 : 1898 ;
133 : 1676 ;
134 : 1452 ;
135 : 1226 ;
136 : 999 ;
137 : 769 ;
138 : 539 ;
139 : 307 ;
140 : 270 ;
141 : 213 ;

142 : 97 ;
143 : 92 ;
144 : 87 ;
145 : 83 ;
146 : 80 ;
147 : 77 ;
148 : 76 ;
149 : 75 ;
150 : 75 ;
151 : 77 ;
152 : 79 ;
153 : 82 ;
154 : 86 ;
155 : 90 ;
156 : 95 ;
157 : 100 ;
158 : 105 ;
159 : 109 ;
160 : 114 ;
161 : 118 ;
162 : 121 ;
163 : 123 ;
164 : 125 ;
165 : 125 ;
166 : 124 ;
167 : 123 ;
168 : 120 ;
169 : 117 ;
170 : 113 ;
171 : 109 ;
172 : 104 ;
173 : 99 ;
174 : 94 ;
175 : 89 ;
176 : 85 ;
177 : 81 ;
178 : 78 ;
179 : 76 ;
180 : 75 ;
181 : 75 ;
182 : 76 ;
183 : 78 ;
184 : 80 ;
185 : 84 ;
186 : 88 ;
187 : 93 ;

188 : 98 ;
189 : 103 ;
190 : 107 ;
191 : 112 ;
192 : 116 ;
193 : 120 ;
194 : 122 ;
195 : 124 ;
196 : 125 ;
197 : 125 ;
198 : 124 ;
199 : 122 ;
200 : 119 ;
201 : 115 ;
202 : 111 ;
203 : 106 ;
204 : 101 ;
205 : 96 ;
206 : 91 ;
207 : 87 ;
208 : 83 ;
209 : 80 ;
210 : 77 ;
211 : 76 ;
212 : 75 ;
213 : 75 ;
214 : 77 ;
215 : 79 ;
216 : 82 ;
217 : 86 ;
218 : 91 ;
219 : 95 ;
220 : 100 ;
221 : 105 ;
222 : 110 ;
223 : 114 ;
224 : 118 ;
225 : 121 ;
226 : 123 ;
227 : 125 ;
228 : 125 ;
229 : 124 ;
230 : 123 ;
231 : 120 ;
232 : 117 ;
233 : 113 ;

```
        234     :       108     ;
        235     :       103     ;
        236     :       98      ;
        237     :       93      ;
        238     :       89      ;
        239     :       84      ;
        240     :       81      ;
        241     :       78      ;
        242     :       76      ;
        243     :       75      ;
        244     :       75      ;
        245     :       76      ;
        246     :       78      ;
        247     :       81      ;
        248     :       85      ;
        249     :       89      ;
        250     :       93      ;
        251     :       98      ;
        252     :       103     ;
        253     :       108     ;
        254     :       113     ;
        255     :       117     ;

END;
```

```
% Zero-Crossing inputs%

WIDTH = 7;  % WIDTH OF OUTPUT IS REQUIRED, ENTER A DECIMAL VALUE
%
DEPTH = 256;  % DEPTH OF MEMORY IS REQUIRED, ENTER A DECIMAL
VALUE %


ADDRESS_RADIX = DEC;  % Address and data radixes are optional, default is hex %

DATA_RADIX = DEC;     % Valid radixes = BIN,DEC,HEX or OCT  %


CONTENT BEGIN
        0       :       10      ;
        1       :       12      ;
        2       :       14      ;
        3       :       15      ;
        4       :       15      ;
        5       :       13      ;
```

6 : 11 ;
7 : 8 ;
8 : 6 ;
9 : 5 ;
10 : 5 ;
11 : 6 ;
12 : 9 ;
13 : 11 ;
14 : 13 ;
15 : 15 ;
16 : 15 ;
17 : 14 ;
18 : 12 ;
19 : 10 ;
20 : 7 ;
21 : 6 ;
22 : 5 ;
23 : 6 ;
24 : 7 ;
25 : 10 ;
26 : 12 ;
27 : 14 ;
28 : 15 ;
29 : 15 ;
30 : 13 ;
31 : 11 ;
32 : 9 ;
33 : 6 ;
34 : 5 ;
35 : 5 ;
36 : 6 ;
37 : 8 ;
38 : 11 ;
39 : 13 ;
40 : 15 ;
41 : 15 ;
42 : 14 ;
43 : 12 ;
44 : 10 ;
45 : 8 ;
46 : 6 ;
47 : 5 ;
48 : 5 ;
49 : 7 ;
50 : 9 ;
51 : 12 ;

52 : 14 ;
53 : 15 ;
54 : 15 ;
55 : 13 ;
56 : 11 ;
57 : 9 ;
58 : 7 ;
59 : 5 ;
60 : 5 ;
61 : 6 ;
62 : 8 ;
63 : 10 ;
64 : 13 ;
65 : 14 ;
66 : 15 ;
67 : 14 ;
68 : 13 ;
69 : 10 ;
70 : 8 ;
71 : 6 ;
72 : 5 ;
73 : 5 ;
74 : 7 ;
75 : 9 ;
76 : 11 ;
77 : 14 ;
78 : 15 ;
79 : 15 ;
80 : 14 ;
81 : 12 ;
82 : 9 ;
83 : 7 ;
84 : 5 ;
85 : 5 ;
86 : 6 ;
87 : 8 ;
88 : 10 ;
89 : 12 ;
90 : 14 ;
91 : 15 ;
92 : 15 ;
93 : 13 ;
94 : 11 ;
95 : 8 ;
96 : 6 ;
97 : 5 ;

98 : 5 ;
99 : 7 ;
100 : 9 ;
101 : 11 ;
102 : 13 ;
103 : 15 ;
104 : 15 ;
105 : 14 ;
106 : 12 ;
107 : 10 ;
108 : 7 ;
109 : 6 ;
110 : 5 ;
111 : 6 ;
112 : 7 ;
113 : 10 ;
114 : 12 ;
115 : 14 ;
116 : 15 ;
117 : 15 ;
118 : 13 ;
119 : 11 ;
120 : 8 ;
121 : 6 ;
122 : 5 ;
123 : 5 ;
124 : 6 ;
125 : 8 ;
126 : 11 ;
127 : 13 ;
128 : 15 ;
129 : 15 ;
130 : 14 ;
131 : 12 ;
132 : 10 ;
133 : 7 ;
134 : 6 ;
135 : 5 ;
136 : 6 ;
137 : 7 ;
138 : 79 ;
139 : 83 ;
140 : 85 ;
141 : 82 ;
142 : 77 ;
143 : 75 ;

144 : 78 ;
145 : 82 ;
146 : 85 ;
147 : 83 ;
148 : 78 ;
149 : 75 ;
150 : 76 ;
151 : 81 ;
152 : 85 ;
153 : 84 ;
154 : 80 ;
155 : 76 ;
156 : 76 ;
157 : 80 ;
158 : 84 ;
159 : 85 ;
160 : 81 ;
161 : 76 ;
162 : 75 ;
163 : 78 ;
164 : 83 ;
165 : 85 ;
166 : 82 ;
167 : 78 ;
168 : 75 ;
169 : 77 ;
170 : 82 ;
171 : 85 ;
172 : 84 ;
173 : 79 ;
174 : 75 ;
175 : 76 ;
176 : 80 ;
177 : 84 ;
178 : 84 ;
179 : 80 ;
180 : 76 ;
181 : 13 ;
182 : 11 ;
183 : 8 ;
184 : 6 ;
185 : 5 ;
186 : 5 ;
187 : 7 ;
188 : 9 ;
189 : 11 ;

190 : 13 ;
191 : 15 ;
192 : 15 ;
193 : 14 ;
194 : 12 ;
195 : 9 ;
196 : 7 ;
197 : 6 ;
198 : 5 ;
199 : 6 ;
200 : 7 ;
201 : 10 ;
202 : 12 ;
203 : 14 ;
204 : 15 ;
205 : 15 ;
206 : 13 ;
207 : 11 ;
208 : 8 ;
209 : 6 ;
210 : 5 ;
211 : 5 ;
212 : 6 ;
213 : 8 ;
214 : 11 ;
215 : 13 ;
216 : 15 ;
217 : 15 ;
218 : 14 ;
219 : 12 ;
220 : 10 ;
221 : 7 ;
222 : 6 ;
223 : 5 ;
224 : 6 ;
225 : 7 ;
226 : 10 ;
227 : 12 ;
228 : 14 ;
229 : 15 ;
230 : 15 ;
231 : 13 ;
232 : 11 ;
233 : 9 ;
234 : 7 ;
235 : 5 ;

```
236   :   5   ;
237   :   6   ;
238   :   8   ;
239   :   11  ;
240   :   13  ;
241   :   14  ;
242   :   15  ;
243   :   14  ;
244   :   12  ;
245   :   10  ;
246   :   8   ;
247   :   6   ;
248   :   5   ;
249   :   5   ;
250   :   7   ;
251   :   9   ;
252   :   12  ;
253   :   14  ;
254   :   15  ;
255   :   15  ;

END;
```

Backend Verilog Code

```
module itoplevel_be1(clk, store, startBE, FlashMode, success, failure, OutputMode,
DACen, FlashCE, FlashOE, FlashWE, FlashAddress, FlashIO, DACinput, dac_cs,
threshold, state, Warp, wRAMoutput, signeddistance, wRAMrw, TemplateMem,
wRAMadd, distance, icRAMrw, icRAMadd, icRAMoutput, FlashOutput, inputselect);
//During Simulation Testing (beginpt, endpt, iRAMoutput, iRAMadd) are simply set

//parameter beginpt = 150;
//parameter endpt   = 160;
input [1:0] inputselect;
wire [7:0] beginpt;
wire [7:0] endpt;
wire [7:0] iRAMoutputZero, iRAMoutputOne, iRAMoutputTwo, iRAMoutputThree;

//-------------------------------------------------
input clk, store, startBE;
//input [7:0] endpt;
//input [7:0] beginpt;


//------------------------DAC Operations-------------------------
output dac_cs;
input DACen;
output [7:0] DACinput;
input [1:0] OutputMode;
output [3:0] state;

//-------------------------Store Data-----------------------------
//output [7:0] iRAMoutput;
//output [7:0] iRAMadd;
wire [7:0] iRAMadd;
wire [7:0] iRAMoutput;

//Cut Data
output icRAMrw;
output [7:0] icRAMadd;
output [7:0] icRAMoutput;
wire icRAMrw;
wire [7:0] icRAMadd;
wire [7:0] icRAMoutput;
//input RAM
wire startCut;
wire StoreBusy;
wire Cut;
//output startCut;
```

```
//output StoreBusy;
//output Cut;

//-----------------------Flash Memory----------------------------
input FlashMode;
output FlashCE, FlashOE, FlashWE;
output [7:0] FlashAddress;
inout [7:0] FlashIO;

//output ProgramBusy;
wire ProgramBusy;
output [7:0] FlashOutput;
//wire [7:0] FlashOutput;


//---------------------------Warp-------------------------------
output [7:0] wRAMoutput;
//wire [7:0] wRAMoutput;
output wRAMrw;
//wire wRAMrw;
output [7:0] wRAMadd;
//wire [7:0] wRAMadd;
output [7:0] distance;
//wire [7:0] distance;
output Warp;
//wire Warp;
//output startWarp;
wire startWarp;
//output WarpBusy;
wire WarpBusy;

//---------------------------Acc--------------------------
wire AccBusy;
wire StartAcc;

wire Acc;

wire [7:0] FlashSize;
wire [7:0] TemplateSize;

//---------------------------Comp-------------------------
input [1:0] threshold;
output success, failure;
//output [7:0] iRAMadd;
wire CompBusy;
wire startComp;
```

```
subtract getTempSize(endpt, beginpt, TemplateSize);
//Store Template Size

//----------------------DAC Operations---------------------
wire [7:0] size;
wire [7:0] DataAdd;

iDacSpecifics DS(OutputMode, FlashSize, TemplateSize, size,
        icRAMoutput, wRAMoutput, TemplateMem, DACinput);
idac_fsm DACFSM(clk, DACen, DataAdd, dac_cs, size);
//-------------------------------------------------------------------------

//-------------SIMULATE FLASH MEMORY with INTERNAL RAM-------------------------
--
//input icRAMoutput, FlashAddress from imemory_control, FlashWE from
imemory_control,
ram fakeFlash(FlashAddress, FlashWE, icRAMoutput, FlashOutput);
//-------------------------------------------------------------------------

//----BE FSM-----------------------------------------------
ibackendcontroller bec(DACen, clk, store, startBE, StoreBusy, ProgramBusy,
WarpBusy, AccBusy, CompBusy, startCut, startFlash, startWarp, startAcc,
startComp, TemplateSize, FlashSize, state);
//-------------------------------------------------------

//----Cut, Flash, Warp, Acc, Comp is turned on when those minor FSM's are called
or2 or_Cut(startCut, StoreBusy, Cut);
or2 or_Warp(startWarp, WarpBusy, Warp);
or2 or_Acc(startAcc, AccBusy, Acc);

wire [7:0] iRAMadd_ic, icRAMadd_ic, icRAMrw_ic;
wire [7:0] icRAMadd_pf, FlashAddress_pf;
wire [7:0] FlashAddress_w, icRAMadd_w, wRAMadd_w;
wire wRAMrw_w;
wire [7:0] cntrAdd;


//----As Cut, Flash, etc. is turned on the Memory Control is
//    switched to outputs from the respective FSM modules
imemory_control mc(FlashMode, DACen, DataAdd, Cut, ProgramBusy, Warp, Acc,
        iRAMadd_ic, icRAMrw_ic, icRAMadd_ic,                    //cutdata
        icRAMadd_pf, FlashAddress_pf, FlashOE_pf, FlashCE_pf, FlashWE_pf,
        //program flash
        FlashAddress_w, icRAMadd_w, wRAMrw_w, wRAMadd_w,        //warp
        cntrAdd,                                                //accum
```

```
        iRAMadd, icRAMrw, icRAMadd, wRAMrw, wRAMadd, //outputs
        FlashAddress, FlashWE, FlashOE, FlashCE); //outputs
//-------------------------------------------------------------------------
//wire [7:0] icRAMoutput;

//-----------------SIMULATE INPUT RAM----------// Can Select 1 of 4 Wave forms
inramsel sel(inputselect,                       //Input4, Input5, Input2, Input3 .mif
 iRAMoutputZero, iRAMoutputOne, iRAMoutputTwo, iRAMoutputThree, iRAMoutput);

endptssel ptssel(inputselect, beginpt, endpt);
rammod0 iRAM0(iRAMadd, iRAMoutputZero);
rammod1 iRAM1(iRAMadd, iRAMoutputOne);
rammod2 iRAM2(iRAMadd, iRAMoutputTwo);
rammod3 iRAM3(iRAMadd, iRAMoutputThree);


//-----------------------------------------//

//-----------------CUT the iRAM data, store in icRAM-------------------------
istoredata cutdata(DACen, clk, store, startCut, beginpt, endpt,
        StoreBusy, iRAMadd_ic, icRAMrw_ic, icRAMadd_ic);
//-------------------------------------------------------------------------

//RAM stores Cut Value
ram icRAM(icRAMadd, icRAMrw, iRAMoutput, icRAMoutput);
//-------------------------------------------------------------------------


//----------------GET the data from icRAM and store in FLASH or FakeFlash-------
iprogramflash PF(FlashMode, DACen, clk, startFlash, FlashSize, ProgramBusy,
icRAMoutput, icRAMadd_pf, FlashAddress_pf, FlashIO, FlashWE_pf, FlashOE_pf,
FlashCE_pf);
//-------------------------------------------------------------------------


//----------------------WARP data from ic-----------------------------------
output [7:0] signeddistance;
iwarp warpdata(DACen, clk, store, startWarp, FlashSize, distance, WarpBusy,
                        FlashAddress_w, icRAMadd_w, wRAMrw_w, wRAMadd_w,
wState);

//RAM stores Warped Values
ram wRAM(wRAMadd, wRAMrw, icRAMoutput, wRAMoutput);
output [7:0] TemplateMem;

//Subtract Flash and icRAM
```

```
chooseinput cin1(FlashIO, FlashOutput, FlashMode, TemplateMem);
sub subw(TemplateMem, icRAMoutput, signeddistance, cout1);
absolute absw(signeddistance, distance, cout1);

//-------------------------------------------------------------------------

//-------------------------Accumulate-----------------------------------------wire [7:0] error;
wire [7:0] signederror;
wire [11:0] out;

iaccumulate acc(DACen, clk, store, AccBusy, startAcc, FlashSize, error, cntrAdd, out);

sub suba(TemplateMem, wRAMoutput, signederror, cout2);
absolute absa(signederror, error, cout2);
//-------------------------------------------------------------------------

//--------Compare----------------------------------------------------------
icomparator cmp(DACen, clk, store, CompBusy, startComp,
        threshold, out, success, failure, rstate, rcntr);
//-------------------------------------------------------------------------
endmodule
```

```
module imemory_control(FlashMode, DACen, DataAdd, Cut, Flash, Warp, Acc,
iRAMadd_ic, icRAMrw_ic, icRAMadd_ic,                    //cutdata
icRAMadd_pf, FlashAddress_pf, FlashOE_pf, FlashCE_pf, FlashWE_pf,  //program
flash
FlashAddress_w, icRAMadd_w, wRAMrw_w, wRAMadd_w,        //warp
cntrAdd,
                        //accum
iRAMadd, icRAMrw, icRAMadd, wRAMrw, wRAMadd, FlashAddress, FlashWE,
FlashOE, FlashCE);

//DAC
input DACen;
input [7:0] DataAdd;
input Cut, Flash, Warp, Acc;

//cutdata
input [7:0] iRAMadd_ic;
input icRAMrw_ic;
input [7:0] icRAMadd_ic;

//program flash
input FlashMode;
input [7:0] icRAMadd_pf;
input [7:0] FlashAddress_pf;
input FlashOE_pf;
input FlashCE_pf;
input FlashWE_pf;

//warp
input [7:0] FlashAddress_w;

input [7:0] icRAMadd_w;
input wRAMrw_w;
input [7:0] wRAMadd_w;

//accum
input [7:0] cntrAdd;
output [7:0] iRAMadd;

output icRAMrw;
output [7:0] icRAMadd;

output wRAMrw;
output [7:0] wRAMadd;
```

```
output [7:0] FlashAddress;
output FlashWE, FlashOE, FlashCE;



assign iRAMadd              = DACen ? DataAdd : Cut ? iRAMadd_ic : 0;

assign icRAMrw              = DACen ? 0 : Cut ? icRAMrw_ic : 0;
assign icRAMadd             = DACen ? DataAdd :  Cut ? icRAMadd_ic :
                                    Flash ? icRAMadd_pf :  Warp ? icRAMadd_w : 0;

assign wRAMrw               = DACen ? 0 : Warp ? wRAMrw_w : 0 ;
assign wRAMadd              = DACen ? DataAdd :  Warp ?
                                        wRAMadd_w : Acc ? cntrAdd : 0;

assign FlashAddress   = DACen ? DataAdd : Flash ? FlashAddress_pf :
                                        Warp ? FlashAddress_w : Acc ? cntrAdd : 0;

//FlashMode 0- RAMstore     1-Flashstore
assign FlashWE              = DACen ? FlashMode : Flash ? FlashWE_pf : FlashMode;
                                         //Automatically in store mode
assign FlashOE              = DACen ? 0 : ~FlashMode ? 1 :
                                    Flash ? FlashOE_pf : Warp ? 0 : Acc ? 0 : 1 ;
                                        //Output should be high
assign FlashCE              = DACen ? 0 : ~FlashMode ? 1 :
                                    Flash ? FlashCE_pf : Warp ? 0 : Acc ? 0 : 1 ;
//if in Warp or Acc AND FlashMode is 1 then
//state CE should be on other times off

endmodule
```

```
module ibackendcontroller(DACen, clk, store, startBE, StoreBusy, ProgramBusy,
WarpBusy, AccBusy, CompBusy, startCut, startFlash, startWarp,
startAcc, startComp, TemplateSize, FlashSize, state);

input store, clk, startBE, DACen;
input StoreBusy, ProgramBusy, WarpBusy, AccBusy, CompBusy;
output startCut, startFlash, startWarp, startAcc, startComp;
input [7:0] TemplateSize;
output [7:0] FlashSize;
output [3:0] state;

parameter       s_Idle          = 0;
parameter       s_startCut      = 1;
parameter   s_Wait4Busy1= 2;
parameter       s_CutCont       = 3;
parameter       s_startFlash= 4;
parameter   s_Wait4Busy2= 5;
parameter       s_FlashCont     = 6;
parameter       s_startWarp     = 7;
parameter   s_Wait4Busy3= 8;
parameter       s_WarpCont      = 9;
parameter       s_startAcc      = 10;
```

```
parameter   s_Wait4Busy4= 11;
parameter       s_AccCont      = 12;
parameter       s_startComp    = 13;
parameter   s_Wait4Busy5= 14;
parameter       s_CompCont   = 15;

reg rstartCut, rstartFlash, rstartWarp, rstartAcc, rstartComp, storeRun;
reg [3:0] state;
reg [7:0] rFlashSize;

always @ (posedge clk)
begin
        if(store||DACen)
        begin
                if(store)
                begin   storeRun<=1; end

                rstartCut<=0; rstartFlash<=0;
                rstartWarp<=0; rstartAcc<=0; rstartComp<=0;
                state<=s_Idle;
        end

        else
        begin
                case(state)
                s_Idle: begin
                                        state<=startBE; rstartCut<=0;rstartFlash<=0;
                                        rstartWarp<=0; rstartAcc<=0; rstartComp<=0;
                                end

                s_startCut:     begin rFlashSize<=storeRun ? TemplateSize :
                                                        rFlashSize; rstartCut<=1;
                                        state<=s_Wait4Busy1;  end
                //set the size of the Flash,
                //if its a store cycle then flash size is the same as Template size,
                //otherwise keep the same
                s_Wait4Busy1:           begin
                                                rstartCut<=0; state<= StoreBusy ?
                                                s_CutCont : state; end
                s_CutCont:  begin rstartCut<=0; state<= StoreBusy ? state :
                                                storeRun ? s_startFlash : s_startWarp;
                                end
        //After the BE first filter the input, check if in store mode,
        //if so store info in FlashMem
```

```
                s_startFlash:    begin rstartFlash<=1; storeRun<=0;
                                        state<=s_Wait4Busy2; end
                s_Wait4Busy2:           begin rstartFlash<=0; state<= ProgramBusy ?
                                                s_FlashCont : state; end
                s_FlashCont:   begin rstartFlash<=0; state<=ProgramBusy ? state :
                                        s_startWarp; end

                s_startWarp: begin rstartWarp<=1; state<=s_Wait4Busy3; end //
                s_Wait4Busy3: begin rstartWarp<=0; state<= WarpBusy ? s_WarpCont :
                                        state; end
                s_WarpCont:  begin rstartWarp<=0; state<=WarpBusy ? state :
                                        s_startAcc; end

                s_startAcc:  begin rstartAcc<=1;  state<=s_Wait4Busy4; end
                s_Wait4Busy4: begin rstartAcc<=0; state<= AccBusy ? s_AccCont :
                                        state; end
                s_AccCont:   begin rstartAcc<=0;  state<= AccBusy ? state :
                                        s_startComp; end

                s_startComp: begin rstartComp<=1; state<=s_Wait4Busy5; end
               s_Wait4Busy5: begin rstartComp<=0; state<= CompBusy ? s_CompCont :
                                        state; end
                s_CompCont:  begin rstartComp<=0; state <= CompBusy ? state :
                                        s_Idle; end
                default :    state<=s_Idle;
                endcase
        end
end

assign startCut = rstartCut;
assign startFlash = rstartFlash;
assign startWarp = rstartWarp;
assign startAcc  = rstartAcc;
assign startComp = rstartComp;
assign FlashSize = rFlashSize;

endmodule

module istoredata(DACen, clk, store, startCut, beginpt, endpt, storeBusy, iRAMadd,
icRAMrw, icRAMadd);
input DACen;
input [7:0] beginpt, endpt;
input clk, startCut, store;


output [7:0] icRAMadd, iRAMadd;
```

```
output icRAMrw, storeBusy;

parameter s_Idle = 0;
parameter s_startStore = 1;
parameter s_storeWait = 2;
parameter s_ChangeAdd = 3;
parameter s_write = 4;
parameter s_done = 5;

reg [2:0] state;

reg [7:0] ricRAMadd, ricRAMinput, riRAMadd;
reg ricRAMrw, Busy;

always @ (posedge clk)
begin
if(store||DACen)
begin
        ricRAMadd<=0; state<=s_Idle;
        ricRAMrw<=1; riRAMadd<=beginpt;
end

else
begin
        case(state)
                s_Idle:
                begin
                        ricRAMadd<=0; state<=startCut;
                        ricRAMrw<=0;        //ricRAM in read state;
                        Busy<=0;
                end
                s_startStore:                   //sets up values for checking
                begin
                        riRAMadd<=beginpt;
    //send address to input ram, start at beginpt

                        ricRAMrw<=1;
    //output of iRAM directly connected to input of icRAM

    ricRAMadd<=0;
//so values of output immediately stored in icRAM

                        state<=s_storeWait;             //iRAM[beginpt]=icRAM[0]
                        Busy<=1;                                //turn on Busy
                end
                s_storeWait:
```

```
                begin
                        ricRAMrw<=0;
                        state<=s_ChangeAdd;//write to ICRAM the next value
                end
                s_ChangeAdd:
                begin
                        if(riRAMadd+1>endpt)
                                state<=s_done;
                        else
                        begin
                                state<=s_write;
                                riRAMadd<=riRAMadd+1;    //Increment Input Ram
Address
                                ricRAMadd<=ricRAMadd+1;
                //Output the Input Ram Data to Cutoff Input Ram, on the same cycle;
                        end
                end
                s_write:
                begin
                        ricRAMrw<=1;
                        state<=s_storeWait;
                end

                s_done:
                begin
                        ricRAMrw<=0; state<=s_Idle; Busy<=0;
                end
        endcase
end
end

assign icRAMadd=ricRAMadd;
assign iRAMadd=riRAMadd;
assign icRAMrw=ricRAMrw;
assign storeBusy=Busy;

endmodule
```

```
module iprogramflash(FlashMode, DACen, clk, startFlash, TemplateSize, ProgramBusy,
icRAMoutput,
icRAMadd_pf, FlashAddress_pf, FlashData_pf, FlashWE, FlashOE, FlashCE);
input DACen;
input startFlash, clk;
input [7:0] TemplateSize;
input FlashMode;

output [7:0] icRAMadd_pf;
input [7:0] icRAMoutput;

output FlashOE, FlashWE, FlashCE;
output [7:0] FlashAddress_pf;
inout  [7:0] FlashData_pf;

output ProgramBusy;

parameter s_Idle=0;
parameter s_ProgramSetup=1;
parameter s_DataIn40h=2;
parameter s_ProgramSWait=3;
parameter s_ProgramData=4;
parameter s_Wait=5;
parameter s_VerifyCommand=6;
parameter s_Wait2=7;
parameter s_ProgramVerify=8;
parameter s_FlashReset=9;
parameter s_FlashReset2=10;
parameter s_StartRAM=12;
parameter s_RAMwriting=13;
parameter s_Repeat=14;
```

```
parameter s_Done=11;

reg [7:0] rFlashData_pf, rFlashAddress_pf;
integer cntr;
reg ce, we, oe, Busy;
reg [7:0] ricRAMadd;
reg [3:0] state;
reg wedelay;

always @ (posedge clk)
begin
if(DACen)
begin
        state<=s_Idle;
        Busy<=0;
        rFlashAddress_pf<=0; wedelay<=0; ricRAMadd<=0;        ce<=1; we<=1; oe<=0;
end
else
begin
        case(state)
                s_Idle: begin   Busy<=0; rFlashAddress_pf<=0; wedelay<=0;
                                ricRAMadd<=0;        ce<=1; we<=0; oe<=1;
                                if(startFlash)
                                begin
     state<=FlashMode ? s_ProgramSetup : s_StartRAM;
                                end //wait for the systemmode to switch to Store Mode;
                                else
                                        state<=state;
                                end
//------------------------------FLASH STORE PROCEDURE-----------------------//
//Follow diagram instruction from AM28F020 Spec Sheet
        s_ProgramSetup:       begin Busy<=1; ce<=0; oe<=1; we<=0;
                                ricRAMadd<=ricRAMadd; state<=s_DataIn40h;
                                end
        s_DataIn40h:  begin ce<=1; oe<=1; we<=1; wedelay<=0;
                                rFlashData_pf<=8'b01000000;
state<=s_ProgramSWait;                                  end
        s_ProgramSWait:       begin ce<=0; oe<=1; we<=0; wedelay<=0;
                                state<=s_ProgramData;
                                rFlashData_pf<=icRAMoutput;
                                        rFlashAddress_pf<=rFlashAddress_pf; end
        s_ProgramData:        begin ce<=1; oe<=1; we<=1;  state<=s_Wait; end
        s_Wait:                 begin
                                if(cntr<19)  //10 us wait at least 19 clock cycles
                                begin
```

```
                                        cntr<=cntr+1; state<=state;
                                    end

                                    else
                                    begin
                                        ce<=0; we<=0; oe<=1;
                                        cntr<=0; state<=s_VerifyCommand;
                                        rFlashData_pf<=8'b11000000;
                                    end
                                    end
        s_VerifyCommand:   begin
                                    ce<=1; we<=1; oe<=1;
rFlashData_pf<=8'b11000000;                                         state<=s_Wait2;
                                end
        s_Wait2:        begin
                                    if(cntr<12)  //6 us wait at least 12 clock cycles
                                    begin   cntr<=cntr+1; state<=state;    end
                                    else
                                    begin   ce<=0; we<=1; oe<=0; cntr<=0;
                                        state<=s_ProgramVerify; end
                                    end

        s_ProgramVerify:       begin
                                    ce<=1; we<=1; oe<=1;


                                    if(FlashData_pf!=icRAMoutput)
                                    begin
                                        state<=s_ProgramSetup;
                                    end
                                    else if (ricRAMadd<TemplateSize)
        //if current address less than endpt, increment everything and program
                                    begin
                                    state<=s_ProgramSetup;
ricRAMadd<=ricRAMadd+1;
                                         rFlashAddress_pf<=rFlashAddress_pf+1;
                                    end
                                    else
                                        state<=s_FlashReset;
                                    end

        s_FlashReset:   begin
                                    rFlashData_pf<=8'b11111111; ce<=0; we<=0;
oe<=1;                                  state<=s_FlashReset2;
                                end
        s_FlashReset2:          begin
```

```
                                        rFlashData_pf<=8'b11111111; ce<=0; we<=0;
oe<=1;                                          state<=s_Done;
                                end

///-------------------RAM STORE PROCEDURE-----------------------------------//
        s_StartRAM:             begin
                                        ricRAMadd<=ricRAMadd;
                                        we<=1; //turn on RAM write
                                        ce<=1;
                                //Make Sure Flash is in Standby Mode since not using it
                                        rFlashAddress_pf<=rFlashAddress_pf;
                                        state<=s_RAMwriting;
                                        Busy<=1;
                                end
        s_RAMwriting:           begin
                                        state<=s_Repeat;
                                        we<=0;
                                //turn off WE, before advancing to next address
                                end
        s_Repeat:               begin
                                        if(ricRAMadd<TemplateSize)
                                        //haven't gotten all the Data yet
                                        begin
                                                state<=s_StartRAM;
                                                rFlashAddress_pf<=rFlashAddress_pf+1;
                                                ricRAMadd<=ricRAMadd+1;
                                        end
                                        else
                                                state<=s_Done;
                                end
        s_Done:                 begin
                                        if(FlashMode)
                                        begin   we<=1; ce<=1; oe<=1; end
                                        else
                                                we<=0;
                                                Busy<=0; state<=s_Idle;
                                        end
        default:                state<=s_Idle;
        endcase
end
//-----------------------------------------------------------------------------
end

assign icRAMadd_pf = ricRAMadd;
assign FlashAddress_pf = rFlashAddress_pf;
assign FlashData_pf =  we ? 8'bzzzzzzzz : rFlashData_pf;
```

```
assign ProgramBusy = Busy;
assign FlashOE = oe;
assign FlashWE = we;
assign FlashCE = ce;
endmodule


module iwarp(DACen, clk, store, startWarp, TempSize, distanceout, warpbusy,
tRAMadd, icRAMadd, wRAMrw, wRAMadd);
input DACen;
output warpbusy;

output [7:0] icRAMadd;

output [7:0] tRAMadd;

output [7:0] wRAMadd;
output wRAMrw;

input clk, store, startWarp;
input [7:0] TempSize;
input [7:0] distanceout;

parameter s_Idle=0;
parameter s_Store0=1;
parameter s_Store0w=9;
parameter s_StartGather=2;
parameter s_Gather1=3;
parameter s_Gather2=4;
parameter s_WarpChoose=5;
parameter s_StoreD=6;
parameter s_WaitACyc=7;
parameter s_WarpDone=8;

reg [7:0] iCounter;

reg [7:0] rd0, rd1, rd2;
reg [1:0] rwarp;
reg [3:0] state;
reg rwarpbusy, rwRAMrw;
reg [7:0] rtRAMadd, ricRAMadd, rwRAMadd;


always @ (posedge clk)
begin
if(store||DACen)
```

```
begin
        state<=s_Idle;
        ricRAMadd<=0; rwRAMrw<=0;
   rtRAMadd<=0; rwRAMadd<=0;     iCounter<=0; rwarpbusy<=0;
end
else
begin
        case(state)
        s_Idle: begin rwarp<=1; state<=startWarp; rwRAMadd<=0; rwRAMrw<=0;
                                iCounter<=0; rtRAMadd<=1; rwarpbusy<=0; end
        //rwarp = 1 because that way first warp can be either 0, 1, or 2,
        //if rwarp = 0, then next step can only be 1 or 2
        //start on rtRAMadd=1, since tRAM[0] is automatically paired with icRAM[0]

s_Store0:   begin rwRAMrw<=1; ricRAMadd<=0; rwRAMadd<=0; rwarpbusy<=1;
state<=s_Store0w;
                                  rtRAMadd<=1;
                        end //store ricRAM[0]->wRAM[0]
                                                //prepare ricRAM[1]
        s_Store0w:              begin rwRAMrw<=0; state<=s_StartGather; end

     s_StartGather:     begin rd0<=distanceout; state<=s_Gather1;
//can get distanceout {ricRAM[0]-tRAM[0]} also prepare to get //{ricRAM[1]-tRAM[1]}
by sending instruction for ricRAM[1]
rwRAMrw<=0; ricRAMadd<=iCounter+1; rtRAMadd<=rtRAMadd;

                                end

        s_Gather1:              begin rd1<=distanceout; state<=s_Gather2;
                                        ricRAMadd<=iCounter+2;
rtRAMadd<=rtRAMadd;
                                        //Prepare to get {ricRAM[2]-tRAM[0]}
                                        end
        s_Gather2:              begin rd2<=distanceout; state<=s_WarpChoose; end
     //get {ricRAM[2]-tRAM[0]}
        s_WarpChoose:           begin //Choose how much to Warp
                                        if((rd0<rd1)&&(rd0<=rd2))
                                        //if distance is smallest for n and m
                                        begin
                                                if(rwarp!=0)
        //if last time didn't stay as same spot, can stay at that spot once.
                                                begin
                                                        rwarp<=0; ricRAMadd<=iCounter;
        //assign correct ricRAMadd for storing in warp
                                                end
                                                else if(rd1<=rd2)
```

```
                                        begin
                                                rwarp<=1; ricRAMadd<=iCounter+1;

                                        end
                                        else
                                        begin
                                                rwarp<=2; ricRAMadd<=iCounter+2;
                                        end
                                end

                        else if ((rd1<=rd0)&&(rd1<=rd2))
        //if n-1 and m smallest that will work
                        begin
                                rwarp<=1; ricRAMadd<=iCounter+1;
                        end

                        else
                                begin
                                if (rwarp!=2)
        //else distance is smallest for n-2 and m
                                begin
                                        rwarp<=2; ricRAMadd<=iCounter+2;
                                end
                                else if(rd1<rd0)
                                begin
                                        rwarp<=1; ricRAMadd<=iCounter+1;
                                end
                                else
                                begin
                                        rwarp<=0; ricRAMadd<=iCounter;
                                        end
                                end
                                rwRAMadd<=rtRAMadd;
                //store ricRAM[n+warp]->rwRAM[n] the correct warp for next time
                                state<=s_StoreD;
                        end
           s_StoreD:    begin
                                rwRAMrw<=1;
                        //Call to respective address of rwRAM and ricRAM

                                rtRAMadd<=rtRAMadd+1;
        //increment rtRAM prepare for next d0
           ricRAMadd<=ricRAMadd;
//store value for warp[n+1]

                                iCounter<=iCounter+rwarp;
```

```
                                        if(rtRAMadd==TempSize)
                                                state<=s_WaitACyc;
                                        else
                                                state<=s_StartGather;
                                end
        s_WaitACyc:             begin
                                        rwRAMrw<=0;
                                        state<=s_WarpDone;
                                end
        s_WarpDone:             begin
                                        rwRAMrw<=0;
                                        state<=s_Idle; rwarpbusy<=0;
    //wait a cycle so that data can be stored
                                end
        default:                begin
                                        state<=s_Idle;
                                end
        endcase
end end

assign warpbusy=rwarpbusy;

assign tRAMadd=rtRAMadd;
assign icRAMadd=ricRAMadd;
assign wRAMadd=rwRAMadd;
assign wRAMrw=rwRAMrw;

endmodule




module iaccumulate(DACen, clk, store, Accbusy, startAcc, TemplateSize, error, cntrAdd,
out);
input [7:0] error;
input [7:0] TemplateSize;
input clk, store, startAcc;
```

```
output [11:0] out;
output Accbusy;
output [7:0] cntrAdd;
input DACen;

parameter s_Idle=2'b00;

parameter s_Acc=2'b01;
parameter s_done=2'b10;

reg [1:0] state;

reg [11:0] Acc;
reg [11:0] tempOut;
reg rAccbusy;
reg rRAMadd;
reg [7:0] rcntrAdd;

always @ (posedge clk)
begin
if(store||DACen)
begin
        state<=s_Idle;
        Acc<=0;
        rAccbusy<=0;
        rcntrAdd<=0;
end
else
begin
        case(state)
        s_Idle: begin
                        Acc<=0;
                        state<=startAcc;
                        rAccbusy<= startAcc ? 1 : 0;
                        rcntrAdd<=0;
                        end
//Assume FLASH Ram and Rom is accurately controlled by the RAM/ROM controller.

        s_Acc:  begin
                        rAccbusy<=1;
                        Acc<=Acc+error;       //Sum up inputs;
                        rcntrAdd<=rcntrAdd+1;
                        if(rcntrAdd+1<TemplateSize)
                                state<= s_Acc;
//when sense the done signal we know CNTR has reached TemplateSize
                        else
```

```
                        state<= s_done;
                    end
        s_done: begin
                    tempOut<=Acc;
                    state<=s_Idle;
                    rAccbusy<=0;
                    rcntrAdd<=0;
                    end
        endcase
end
end

assign Accbusy = rAccbusy;
assign out = tempOut;
assign cntrAdd = rcntrAdd;
endmodule



module icomparator (DACen, clk, store, CompBusy, startComparator, threshold,
accumulate, success, failure);
input DACen;
input clk, store, startComparator;
input [11:0] accumulate;
input [1:0] threshold;
output success, failure, CompBusy;


reg rsuccess, rfailure, rbusy;
integer cntr;
reg [1:0] state;
reg [11:0] rthres;

parameter s_Idle=2'b00;
parameter s_startCompare=2'b01;
parameter s_Wait2Sec=2'b10;

always @ (posedge clk)
begin
   if(store||DACen)
        begin
            rsuccess<=0;
            rfailure<=0;
            state<=s_Idle;
            rbusy<=0;
            cntr<=0;
```

```
        end
        else
        begin
                case(state)
                s_Idle:
                begin
                        rbusy<=0;
                        rsuccess<=0;
                        rfailure<=0;
                        cntr<=0;
                        state<=startComparator;
                        case(threshold)
                                2'b11: rthres<=12'b001111111111;
                                2'b10: rthres<=12'b000001111111;
                                2'b01: rthres<=12'b000000001111;
                                2'b00: rthres<=12'b000000000001;
                        endcase
                end
                s_startCompare:
                begin
                        rbusy<=1;
                        cntr<=0;

                        if(accumulate<rthres)
                        begin   rsuccess<=1; end

                        else
                        begin   rfailure<=1; end
                        state<=s_Wait2Sec;

                end
                s_Wait2Sec:
                begin
                        if(cntr==3000000)
                          state<=s_Idle;
                        else
                        begin
                          cntr<=cntr+1;
                          state<=s_Wait2Sec;
                        end
                end
        endcase
end
end
```

```
//Output results, output busy status
assign success=rsuccess;
assign failure=rfailure;
assign CompBusy=rbusy;

endmodule

module idac_fsm(clk, dac_en, speech_adr, dac_cs, size);
	input clk, dac_en;
	input [7:0] size;
	output[7:0] speech_adr;
	output dac_cs; //when high latched, when low inputting new data
	parameter IDLE = 0;
	parameter CSLOW = 1;
	parameter CSHIGH = 2;
	parameter INCADR = 3;

	reg output_done, dac_cs;
	reg[7:0] speech_adr;
	reg[1:0] state;
	always @ (posedge clk)
	begin
		case (state)
			IDLE:
				begin
				dac_cs <= 1;
				speech_adr <= 0;
				if (dac_en) state <= CSLOW;
				end
			CSLOW:
				begin
				dac_cs <= 0;
				state <= CSHIGH;
				end
			CSHIGH:
				begin
				dac_cs <= 1;
				state <= INCADR;
				end
			INCADR:
				begin
				if (speech_adr == size)//255)
				begin
				state <= IDLE;
				end
				else
```

```
                                begin
                                speech_adr <= speech_adr + 1;
                                state <= CSLOW;
                                end
                                end
                default: state <= IDLE;
                endcase
        end
endmodule


module iDacSpecifics(DacMode, FlashSize, DataSize, size, icRAMoutput, wRAMoutput,
TemplateMem, DACinput);
input [7:0] FlashSize;
input [7:0] DataSize;
output [7:0] size;
input [1:0] DacMode;

input [7:0] icRAMoutput, wRAMoutput, TemplateMem;
output [7:0] DACinput;

//output select mode
//--0     Cut Data
//--1     Warped Memory
//--2     Flash Memory

assign size = DacMode[1] ? FlashSize : DacMode[0] ?  FlashSize : DataSize;
assign DACinput = DacMode[1] ? TemplateMem : DacMode[0] ?  wRAMoutput :
icRAMoutput;

endmodule
```