FPGzAm – Song Identification System

Yafim Landa, Pranav Sood

We present a song identification system, designed to listen to a clip and determine whether it's a part of one of the several songs in its database. If the system finds a match, then it displays the track number. The system was designed to identify songs in the presence of noise. FPGzAm aims to mimic the functionality of Shazam – an iPhone application that is designed to do the same thing – in hardware on the 6.111 labkit.

Table of Contents

List of Figures iii

Overview 1

Description 3

MATLAB Simulation (Landa) 3

Spectrogram using Fast Fourier Transform (Sood) 4

Audio Input (Sood) 7

Input Cascade (Landa) 7

ZAM (Landa) 11

Appendices 16

List of Figures

Figure 1: Plot of a match near the beginning (y-axis: delta, x-axis: windows in time).
Figure 2: Plot of a noisy match near the beginning (y-axis: delta, x-axis: windows in time).
Figure 3: Plot of a mismatch (y-axis: delta, x-axis: windows in time).
Figure 4: 2s spectrogram on Sting's *Desert Rose* generated using MATLAB.
Figure 5: Output obtained when the FFT is tested with a 750 Hz signal and observed on the logic analyzer.
Figure 6: Block diagram depicting the pipelining in the FFT. [1]
Figure 7: Full precision unscaled arithmetic. [1]
Figure 8: Input Cascade subsystem.
Figure 9: Simulation of the FFTOutput module in ModelSim.
Figure 10: SyncGen ModelSim test screenshot.
Figure 11: ZAM subsystem, consisting of the Searcher, the Muxes, the Memories, and the FSM.
Figures in Appendix A: ModelSim Test Diagrams

Overview

The goal of the system is to be able to identify a 2-second song given a 250ms clip. This clip can be noisy or have other signals (such as banter) overlaid on top of it, yet still be recognized ("zammed") by our system as the correct song. If a clip doesn't appear to have come from one of the songs in the database, then the system tells the user that the clip did not match any song.

This goal is accomplished through the following process. First, the audio signals are supplied to the system, sampled at 48kHz. These signals go through a spectrograph, which is generated by a Fast Fourier Transform (FFT) module every 1024 samples with a 512 sample overlay. The signals are then passed to a peak detection module that generates **slices** – 45bit values that are the concatenation of five 9-bit peak frequency bins in every Fourier transformation. The five frequency bins are chosen in the following manner. We take five bin ranges, roughly 1–100, 100–200, 200–300, 300–400, and 400–500, and take the bin in each range that has the highest amplitude generated by the Fourier transformation. A sequence of slices then forms a **fingerprint** that is used to identify that particular audio sample. The fingerprint can have two different lengths: the length of the fingerprint for the 2s song is 170, and the length of the fingerprint for the 250ms clip is 22. Note that we changed the length of the fingerprint from 180 to 170 because we were planning to use 3 memories, for which 180 would have been too long.

These fingerprints are stored into two memories, one memory bank that is for the song fingerprints and one memory that is dedicated to the clip fingerprint. The song fingerprint memory bank contains enough space to store two songs, making our system able to determine whether the given clip belongs to song 1, song 2, or neither. The clip memory is able to store the fingerprint for one clip at a time, since the clip is not information that we need to use more than once, unlike the song database.

The search module interfaces with the memories to detect whether a clip belongs to one of the two songs. It does so by scanning the clip fingerprint through the song fingerprints while calculating the fingerprint **delta**, which is a measure of the dissimilarity of the clip and the songs. If the delta crosses a certain threshold upper bound, then we say that we have found a **match**, and output the song memory bank number that yielded this match. If the delta never crosses below the threshold, then we have found no match and output the result as memory bank 0 (which means that there is no memory bank that matched the clip).

The system presents the user with output on the LED available on the labkit. LED 6 and 7 show the memory bank that stores the song that matches the clip if there is a match, or they are off if there was no match.

Testing the System

The testing methodology is as follows. First, we have constructed a MATLAB simulation of the system. The MATLAB simulation aimed to mimic what had eventually become the Verilog, FPGA project. Running the MATLAB simulation yielded the result that this project was in fact feasible, and that our error bounds were sufficient to distinguish two songs from one another and from other songs. The testing methodology differed for the two components of the system.

Pranav designed and tested the FFT, including the various modifications that reduced the error bounds (such as overlaying windows). Since the FFT is intrinsically bound to the hardware, and the Xilinx tools generate part of it, it was mostly tested on the labkit itself using the logic analyzer. The testing procedure entailed giving the FFT sample signals and checking the output values. The audio sources were also designed and tested by Pranav, and entailed generating COE files and both listening to them and passing them through the FFT to ensure the correct behavior of both. The rest of the project was designed and tested by Yafim. The FFT output module, fingerprint generator, searcher, memories, and multiplexers were tested in simulation using ModelSim. Each component was tested individually, with the FFT output module – the buffer between the FFT and the gateway to the rest of the system – being tested with fake FFT data as input. The other components were tested with data that was generated from other components in isolation. Furthermore, the components were also connected to one another and tested together, with fake FFT data being provided to this system once again. The test Verilog file and the simulation screenshots are provided in the appendix of this report. Finally, these components were brought into the lab where Pranav and Yafim tested the interconnection of the audio input and FFT with the FFT output module to complete the project.

Description

Here we describe each part of the system in more detail. Please refer to Figure XI in Appendix A for an overall ModelSim test screenshot which shows how the whole system works in simulation.

MATLAB Simulation (Landa)

We first made a MATLAB simulation of the FPGzAm system. The MATLAB simulation performed what was described in the overview in software, and was used to verify that this system was feasible. A 2s sample from a song – Sting's *Desert Rose* – was treated as the song database, and the system was asked to "zam" a 250ms clip from within that 2s sample, the same clip with noise, and a clip from a different song. The results indicated that the delta measurement was comparable (in the 0–30 range) for the two matches but much higher (in the 90 range) for the mismatch. Figure 1 shows the delta values for different offsets of a match near the beginning of the song; Figure 2 shows the delta values for the same segment but with added white noise; and Figure 3 shows the delta values for different offsets of a mismatch.

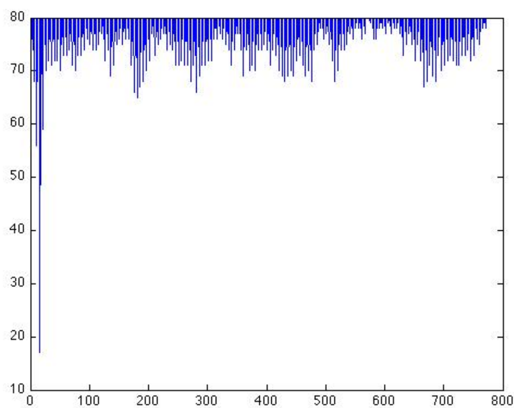


Figure 1. Plot of a match near the beginning (y-axis: delta, x-axis: windows in time).

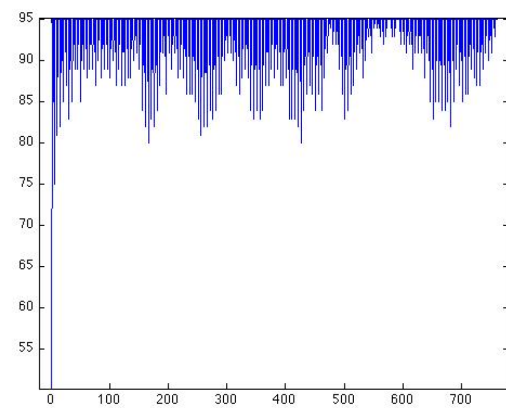


Figure 2. Plot of a noisy match near the beginning (y-axis: delta, x-axis: windows in time).

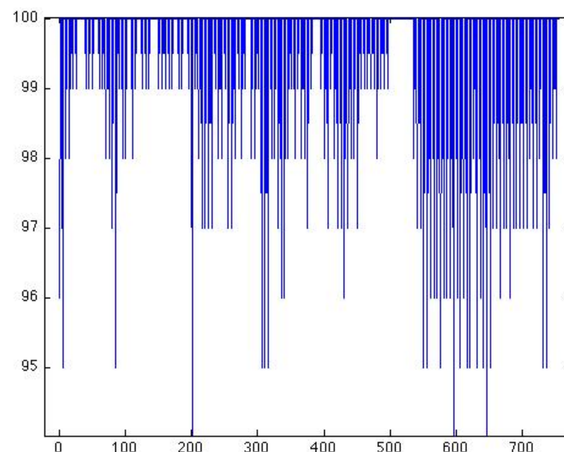


Figure 3. Plot of a mismatch (y-axis: delta, x-axis: windows in time).

Spectrogram using Fast Fourier Transform (Sood)

INPUTS : *clock enable(ce), forward inverse write enable(fwd_inv_we), fft_start(start), output ready(rfd), foward_inverse(fwd_inv), clock(clk), real input(xn_re), imaginary input(xn_im)*
OUTPUTS: *output ready(rfd), done, busy, early_done(edone), input index(xn_index), real output(xk_re), imaginary output(xk_im), output index(xk_index).*

In order to transform the input audio from the time domain into frequency domain, a fast Fourier transform is performed on the input audio samples. This produces a spectrogram that depicts the intensity variation at different frequencies. This is done so as to make sure that the process of fingerprinting is carried on the frequency domain which is a reliable method of differentiating audio samples. This is because the FFT algorithm captures the spectrum for short duration events which can be used to create fingerprints from each of the windows of the FFT which further form the signatures which are unique to every audio sample. Another advantage of using the FFT is that it is computationally faster than other transformation techniques.
The FFT algorithm is basically a computationally faster technique of calculating the discrete Fourier transform the time domain signal. We do an FFT using the Cooley-Tukey algorithm which is briefly described below.
The general formula for a DFT is:

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-jnk2\pi/N} \quad k = 0, \ldots, N-1 \tag{1.}$$

In the Cooley-Tukey algorithm, we first decompose the DFT into two equal DFTs, with the outputs added and subtracted together in pairs. We recycle values from the two DFTs into the final calculation because of the periodicity of the DFT output. Examining how pairs of outputs are collected together, we create the basic computational element known as a butterfly.
The figure below depicts the spectrograph produced by doing a FFT on a 2 second audio track (simulated in MATLAB).

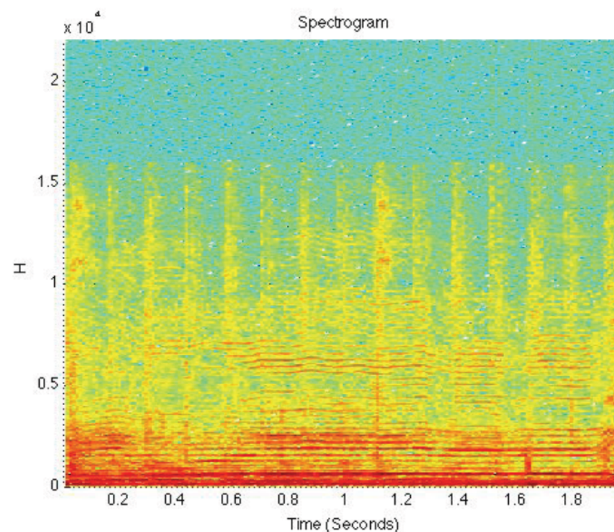


Figure 4. 2s spectrogram on Sting's *Desert Rose* generated using MATLAB.

In Verilog, an FFT module was created using the core generator in ISE 8.1. The specifications of the FFT were:

- Transform Length: - **1024**
- Implemented in **pipelined** mode.
- Input data width: - **8 bits**.
- **Unscaled** output.
- Output data width: - **19 bits**.
- The outputs of the intermediate stages were **truncated**.
- The output was produced in **natural order**.
- It consisted of a **clock enable (CE)** pin which was active high at 48khz. This is because the song input from the BRAM was sampled at 48khz.
- Also, in order to synchronize the input audio samples with the FFT a **synchronous clear (sclr)** signal was used which resets both the FFT (except its internal buffers) and the input of the audio signal when active high, thus acting as a master reset.

The FFT was initially tested with a **750hz** test signal sampled at **48khz**. Since a 1024 point FFT is being carried out, the number of frequency range of each point of the FFT is:

48000 Hz / 1024 = 46.875 Hz/point.

Now since our test signal is 750 Hz, to find the bin in which the real and imaginary outputs of this signal is stored we divide 750 by 46.875.

750/46.875 = 16th bin.

The output observed on the logic analyzer for the 750 Hz test signal is shown below, confirming the proper function of the FFT.

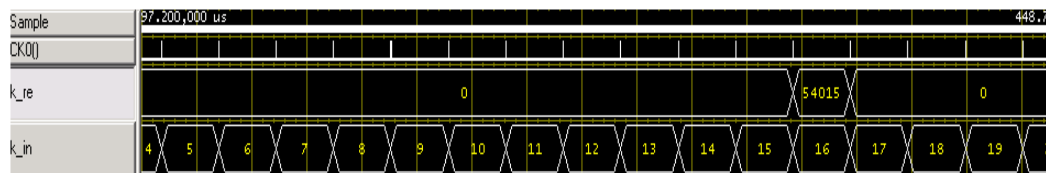


Figure 5. Output obtained when the FFT is tested with a 750 Hz signal and observed on the logic analyzer.

Pipelining

This FFT is a pipelined FFT, that is, it starts producing outputs point by point in order from the 0th point until the 1023th point as soon as the outputs are available in each of the bin. It consumes minimal space in BRAM by not storing any of the outputs in the memory, although it uses the BRAM for its intermediate calculations which covers about 5% of the BRAM.

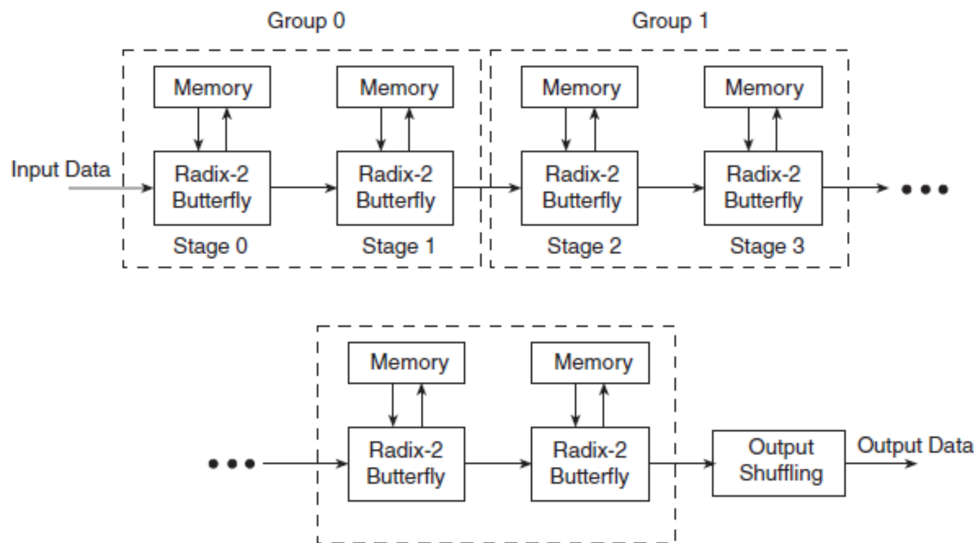


Figure 6. Block diagram depicting the pipelining in the FFT. [1]

Unscaled Transform

The width of the output produced is 19 bits. This is because the output data is larger due to the intermediate stages in the FFT. Also, since we are using an unscaled FFT, we do not check for the overflow of data anywhere in the process, thus producing a large output. The fact that the output data is large does not affect our algorithm since we just use the intensities in each of the bins to compare and find the bin with the largest value and store the bin number, and not the value itself. Also, the unscaled output leads to the generation of a more precise value of the output, thus making the comparison of the intensities more precise. The numbers of bits in the output for the unscaled transform are given by:

$$\textbf{(Input data width} + \log_2\textbf{(point size)} + 1\textbf{)}.$$

The figure below represents the result of using a full precision unscaled arithmetic.

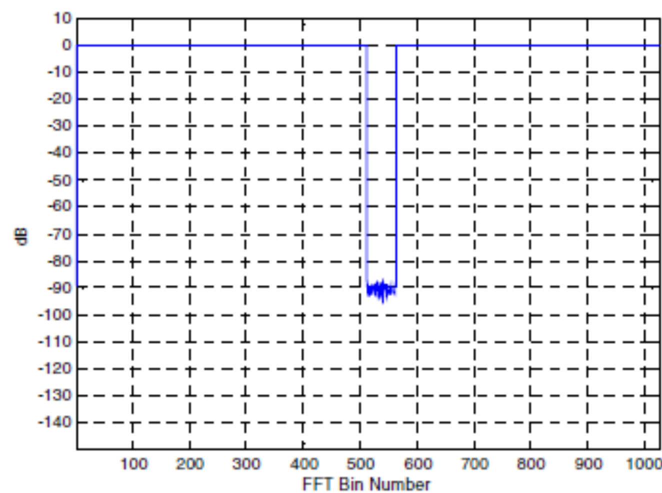


Figure 7. Full precision unscaled arithmetic. [1]

Clock Enable

By dividing the system clock of the FPGA (27 MHZ), a 48 kHz clock is created. This clock is fed into the clock enable signal of the FFT module. Its basic function is to control the rate which the FFT accepts the input from the audio source (48kHz in this case). As a consequence, the output produced is also at a rate of 48kHz.

Synchronous Clear

The synchronous clear is used as a master reset to reset the FFT explicitly at any point during the transform. The same reset is also used to reset the address of audio samples. This synchronizes the FFT and the input audio, thus avoiding any discrepancies during the search operation.

Output Intensities

For the purpose of comparison, we need to produce the intensities at different points in output of the FFT in order to determine the point that has the highest value. This is done by squaring the amplitude of the real and the imaginary parts of the output at each point and summing them to produce the intensity. Therefore intensity I is given by:

$$I = (|xk_real|)^2 + (|xk_imaginary|)^2 \quad (2.)$$

50% Overlap

In order to make the search operation more comprehensive, we do a 50% overlapping FFT so as to make sure that the output due to each input sample is taken into consideration while fingerprinting an audio sample. Most importantly the 50% overlap ensures that the intensity at point is comprehensively compared with other values. This is done by reducing the address of the input audio by 511 samples every time it reaches a value of 1023 samples.

Audio Input (Sood)

The audio input into the system during testing was created by taking two 2s audio tracks

1. Desert Rose – Sting
2. Boo Boom Pow – Black Eyed Peas.

These audio tracks were then interpolated to 48 kHz from 44.1 kHz, then using a MATLAB script the WAV sample was converted to a COE file.

These files were then written into the block RAM of the FPGA by creating a ROM using the FPGA. Thus two audio files were stored into the memory of the FPGA tat could be used to test the system while avoiding any external noise.

In addition, a 2s 750Hz signal along with a 1s 2kHz followed by 1s 7kHz signal were generated. These signals were used for the purposes of testing the system, and were not included in the final product.

Input Cascade (Landa)

The “InputCascade” module receives raw data from the FFT module and outputs fingerprint slices. It consists of the following major components:

- FFT Output;

- Frame Buffer;
- Peak Detector.

It also includes three service modules:

- BRAM;
- Address Multiplexer;
- Sync Generator.

Figure 8 shows a block diagram of this subsystem. Please refer to the diagram throughout the description that follows, as the inputs and outputs, along with the various interconnects, are specified there. Additionally, the Verilog source code of the Input Cascade is available in Appendix B.

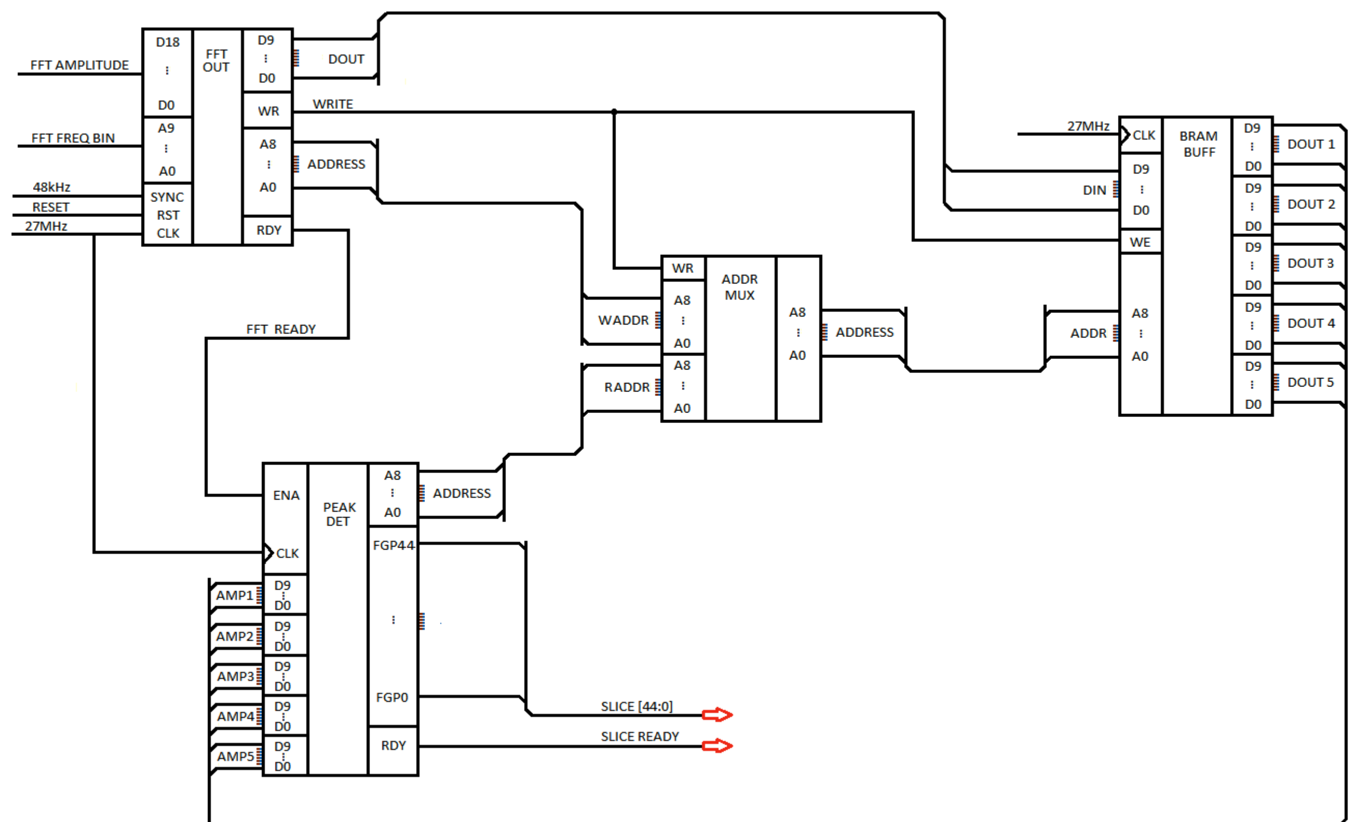


Figure 8. Input Cascade subsystem.

FFT Output

The "FFTOutput" (FFTO) is designed to interface with the FFT and produce the signal required by the other components in Figure 8. This is a critical module, so let us describe it in detail. The FFTO module's state machine has three states.

State 2'b00 – is the "idle" state. While in this state, the FFTO keeps zeroes on the control outputs "RDY" and "WE", signaling that data from FFT is not available yet. FFTO module will be forced to this state by applying "1" on its "RST" input.

It will change its state from 2’b00 to 2’b01 as soon it detects that all A9:A0 inputs are zeroes, meaning that FFT module had just finished the Fourier transformation and is about to start to output amplitude values for the first frequency bin.

State 2’b01 is the active state. While in this state, the FFT first analyzes what bin is currently supplied by the FFT. If the FFT is not ready yet (the SYNC input is low) then it stays in this state.

If this is one of the first 512 bins, meaning that the FFT is still outputting bins from the positive range, it takes the ten most significant bits (MSB) from the amplitude input [D18:D9] and stores them into the Frame Buffer in `address`, defined by the nine least significant bits (LSB) of the frequency bin input [A8:A0]. The “RDY” output is still “0”, and the “WE” output transitions to a “1” for one clock cycle to perform the write.

If the MSB of the frequency bin (input A9) becomes “1”, meaning that the FFT is outputting the negative range, then the “RDY” output becomes “1” signaling that the Frame Buffer is filled up, and “WE” becomes “0” to disable further memory writing. The FFTO remains in state 2’b01 while both the A9 and SYNC inputs are “0”. If the A9 input is “1” the FFTO switches to the idle state with a “1” on its “RDY” output.

It switches to state 2’b10 – the synchronizing state – while filling up the Frame Buffer. In this state the FFTO sets “0” on its “WE” output and preserves the state of the “RDY”. It remains in this state as long as there’s a “1” on its “SYNC” input. As soon as the “SYNC” input becomes “0,” meaning the FFT will now change its outputs it switches back to the active state (2’b01).

Figure 9 depicts this process on ModelSim’s waveform window.

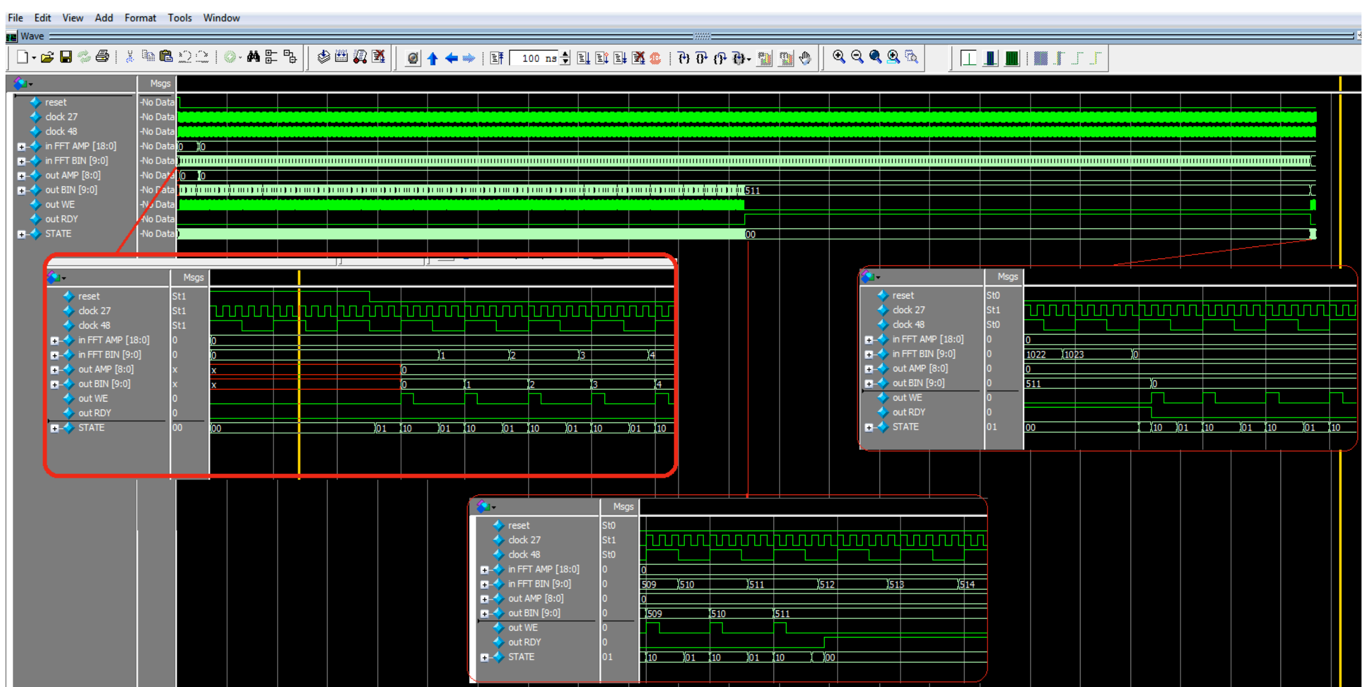


Figure 9. Simulation of the FFTOutput module in the InputCascade subsystem.

Frame Buffer

The Frame Buffer (FB) is a special kind of memory. Its input is a regular combination of the data bus (DIN[9:0]), the address bus (ADDR[8:0]), and a control signal ("WE"). Its output is a 50 bit wide data bus (DOUT[49:0]). It is synchronized by the "CLK" input and operates as follows:

- When "WE" = 1, it stores DIN to the ADDR location in memory;
- DOUT is 50'b0 if the ADDR>99, otherwise it's a concatenation of the following five memory locations: {ADDR, ADDR+100, ADDR+200,ADDR+300,ADDR+400}.

Peak Detector

The Peak Detector (PD) is synchronized to the 27 MHz clock. Its work is controlled by the "ENA" input. When this input is low, the PD module sets zeroes on all its outputs and presets all internal registers, as if there were a "1" on the "RESET" input.

When there's a "1" on the "ENA" input the PD module works as follows:

- Five bytes of data (with each byte composed of 10 bits) are received from the Frame Buffer (FB) module simultaneously; these are amplitude values from five frequency ranges: 0-99; 100-199; 200-299; 300-399 and 400-499;
- The maximum amplitude for each range is computed separately: current amplitude of a range is compared with the current MAX value for this range and if current value is greater than the MAX, it becomes a new MAX and its frequency is stored;
- The first two steps are repeated 100 times, at which point there'll be five FREQ values defined, each one denoting the frequency of maximal amplitude for its range;

PD outputs concatenation of the FREQ values, which is the "slice" (SLICE[44:0]) and indicates "slice ready" (SLICE_RDY=1)

Address Multiplexer

The purpose of Address Multiplexer (AMUX) is to dispatch access to the FB. When FFTO writes data to FB, there should be no interference on FB's address bus from PD. AMUX is driven by its "WE" input allowing its "WADDR" (write address) inputs a path through to the ADDR outputs when "WE" is high. When "WE" is low, AMUX passes its RADDR (read address) inputs through.

Note: This module is universal, and is used in other parts of the project as well.

Appendix A includes screenshots of the "testInputCascade" module simulation. There are five screenshots of testInputCascade – one for each phase (and the corresponding component in Figure 8). Please refer to the captions of Figures I through V for more details about each screen shot.

SyncGen

Generates a one-clock pulse on the positive edge of the control signal. Signals from the "learn" and "search" buttons come through this pulse generation module. Figure 10 shows a ModelSim test of this module.

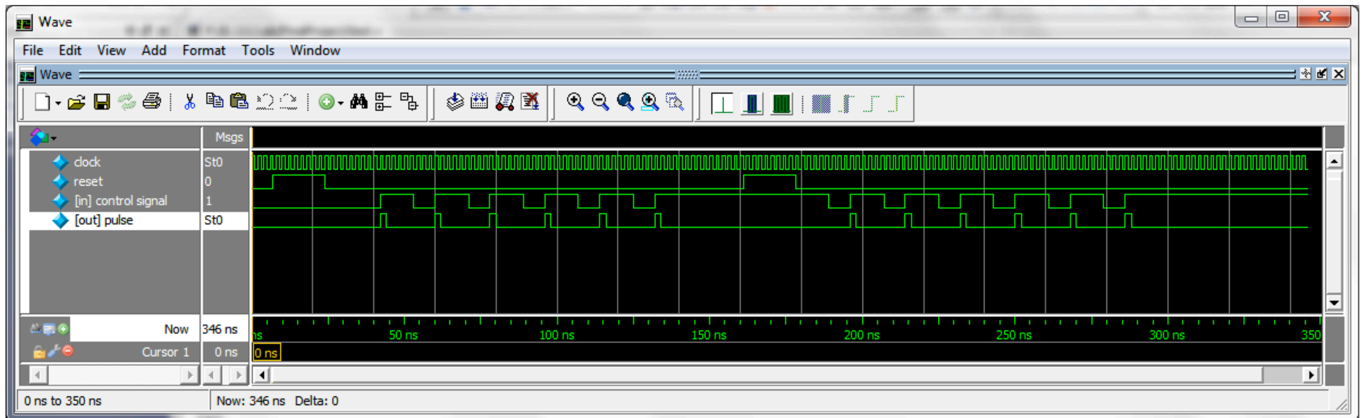


Figure 10. SyncGen ModelSim test screenshot.

ZAM (Landa)

This is the rest of the system. The ZAM subsystem diagram is depicted in Figure 11. There are three additional modules:

- FSM
- Memories (and multiplexers, described in the Input Cascade section)
- Searcher

This subsystem is connected to the InputCascade subsystem that was described in the previous section. To provide context, Figure 8 is contained in the "Input Cascade" module of Figure 11.

ZAMFSM (Finite State Machine)

The FSM has 4 main states. There are 4 additional states, which are used for interfacing with the memories, and will also be described. The register "stack" determines which state to return to after we've performed the computation in the current state. There are also hard-coded constants for where to write the song and clip, and the length of the fingerprint of each (170 and 22).

State "IDLE" is the idle state, where the FSM is not performing any actions.

State "LEARN" is the state when the user has pressed the "learn" button that was converted to a pulse and we have received this pulse. In this state, the memories get filled. IDX1 and IDX0 choose the memory bank to write into by controlling the start address of the fingerprint in the memory. The mode signal is an indicator that determines whether the system is currently learning, or if it's done learning and can accept another song.

State "ZAM" is the state when the user has pressed the "zam" button, which was handled similarly to the "learn" button. In this state, it writes a fingerprint of 22 slices into the clip memory. The "stack" register forces the FSM to transition into the "SEARCH" state after the memories are filled.

State "SEARCH" sets the enable signal for the search module. The FSM stays in this state until we receive the "stop" or "reset" signal. The "stop" signal arrives from the Searcher, and indicates that it has performed the search to completion.

The four states that interface with the memory are FRM_WAIT1, FRM_STORE, FRM_WAIT2, and FRM_NEXT.

State "FRM_WAIT1" waits for the *slice ready* signal to be asserted. When it is asserted, it transitions into the "FRM_STORE" state, where the frame is stored into the appropriate memory and the appropriate memory bank if we're storing a song fingerprint. This state essentially sets the appropriate write enable signal. We then transition into the "FRM_WAIT2" state, where we wait for the *slice ready* signal to drop again (indicating that the FFT passed into the negative frequency bins), and then moving on to the "FRM_NEXT" state where we increment the address if we're not at the last address, or the state stored in the "stack" register if we're at the last address.

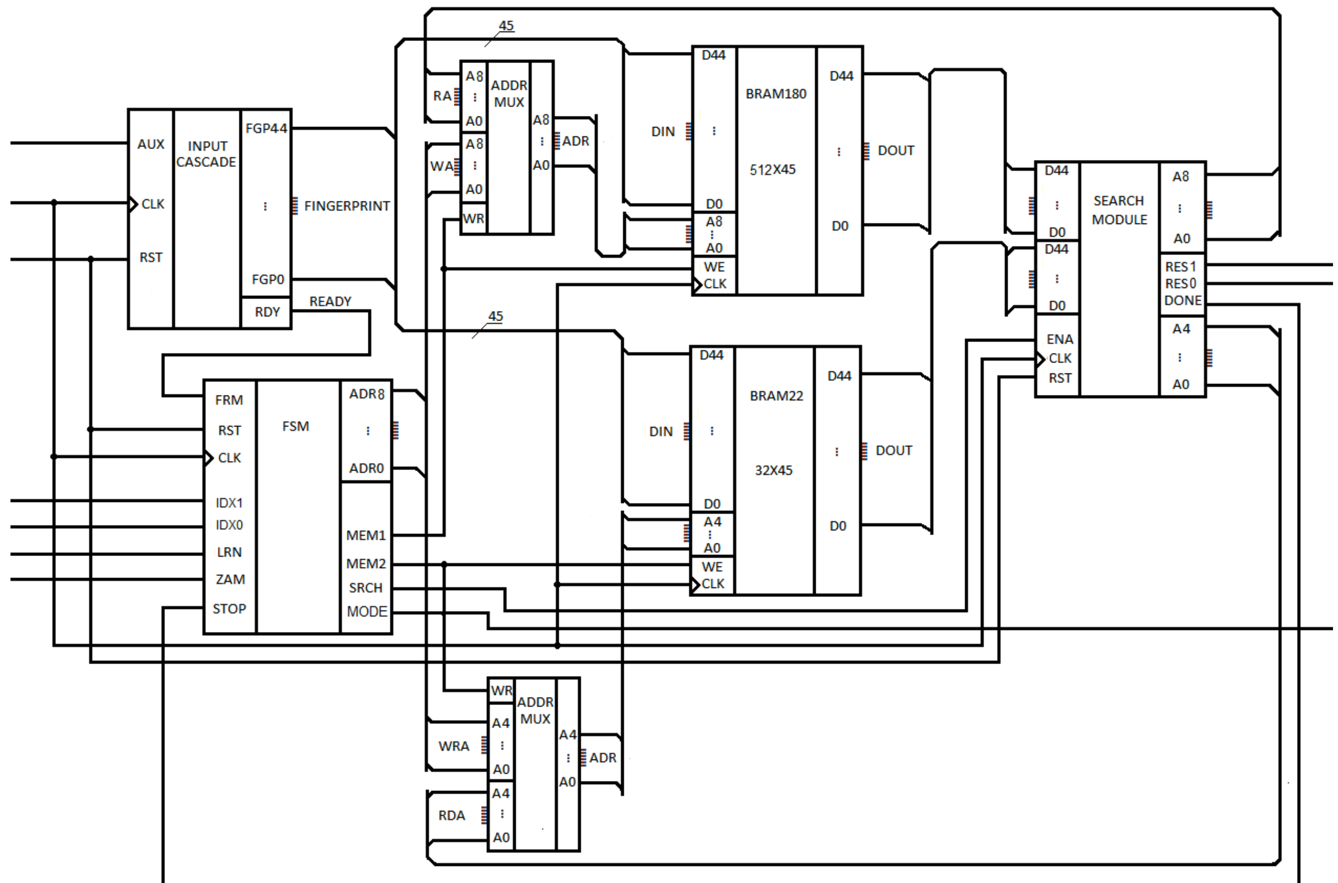


Figure 11. ZAM subsystem, consisting of the Searcher, the Muxes, the Memories, and the FSM.

I have created a ModelSim simulation to test the FSM. The simulation testZAMFSM does the following:

- After reset, the "learn" signal is applied with idx = 2; this puts the FSM in "LEARN" mode: it fills 2sec memory with 170 slices, starting from address 170 (which corresponds to the second memory bank);

- It then sets idx = 1 and issues another learn signal: FSM enters the “LEARN” mode again and fills 2sec memory with 170 slices, starting from address 0 (which corresponds to the first memory bank);
- Then, the “search” signal is applied, causing FSM to switch in “SEARCH” mode, where it first fills the 250ms memory with slices and then goes into “ZAM” mode, starting the searcher;
- Finally, the “stop” signal is asserted, emulating the “searcher done” signal, causing the FSM to return to its “IDLE” state.

The screenshot of this simulation can be found in the appendix as Figure VI. This procedure is shown in detail in Figures VII–VIII.

Memory

The memory is self-descriptive and is based on the BRAM module. The BRAM size is chosen using the parameters and is set to be 512x45 for the song memory and 22x45 for the clip memory. I’ve included a screenshot of the test in Figure

Searcher

The Searcher consists of four states. In the “IDLE” state, the Searcher is idle and is waiting for the enable signal, at which point is transitions into the “WORK” state.

In the “WORK” state, it draws on the contents of mem1 and mem2, the two memories (for the song and clip, respectively), into the five ranges. It goes along and calculates the delta values described in the overview for each range, resulting in five delta values. These deltas are calculated just by adding one if there is a difference in the current contents of the two memories, and then the five deltas are summed into the total delta. It then increments the address so that this is done for 22 calculations of delta. It then transitions into the “NEXT” state.

In the “NEXT” state, if delta is less than the threshold, then we output the value of “res” which is the result memory bank. If delta is not less than the threshold, then we shift the addresses over to compare the next offset, clear all of the deltas, and transition into the “WORK” state. If the address reaches the end of mem1, then instead of transitioning into the “WORK” state, it transitions into the “DONE” state and holds its result until the enable signal drops. If we’ve passed the point of the first memory bank, then we change “res” to point to the second memory bank.

The result is that we take the clip and compare it to every possible window of size 22 in the songs in both memory banks. If we ever cross the threshold into the match territory, then we output the memory bank in which we’ve found this match. Otherwise, we output that we didn’t find a match.

Please refer to Figure X in Appendix A for the search ModelSim test, which searches for the following: Two phases, expected result 01. First fill up both memories with test data:

Mem1: `0-------18+++39-----------------------------509`

Mem2: `0+++21`

Conclusion

We have created a Shazam clone on the FPGA. We have entered two songs into the database of songs, and also used them against one another as the clips. We only tested these against one another, but the system worked. We have included functionality that would allow for the system to handle noise and still be able to identify the tracks, but we did not get a chance to test it on hardware, since our COE files filled the memory.

For the checkoff, we demonstrated that FPGzAm distinguishes between the two tracks – Boom Boom Pow and Desert Rose – and that it shows no match when the song is not in the database.

References

[1] Xilinx, “LogiCORE IP Fast Fourier Transform v7.1,” XFFT Datasheet, April 19, 2010.

Appendices

Appendix A contains timing diagrams that contain a lot of detail. These have been inserted into the document in landscape mode, so that the detail would be more visible to the reader.

Appendix B contains Verilog code for all of the modules in the design. The only exceptions are the generated FFT code and the generated COE code.

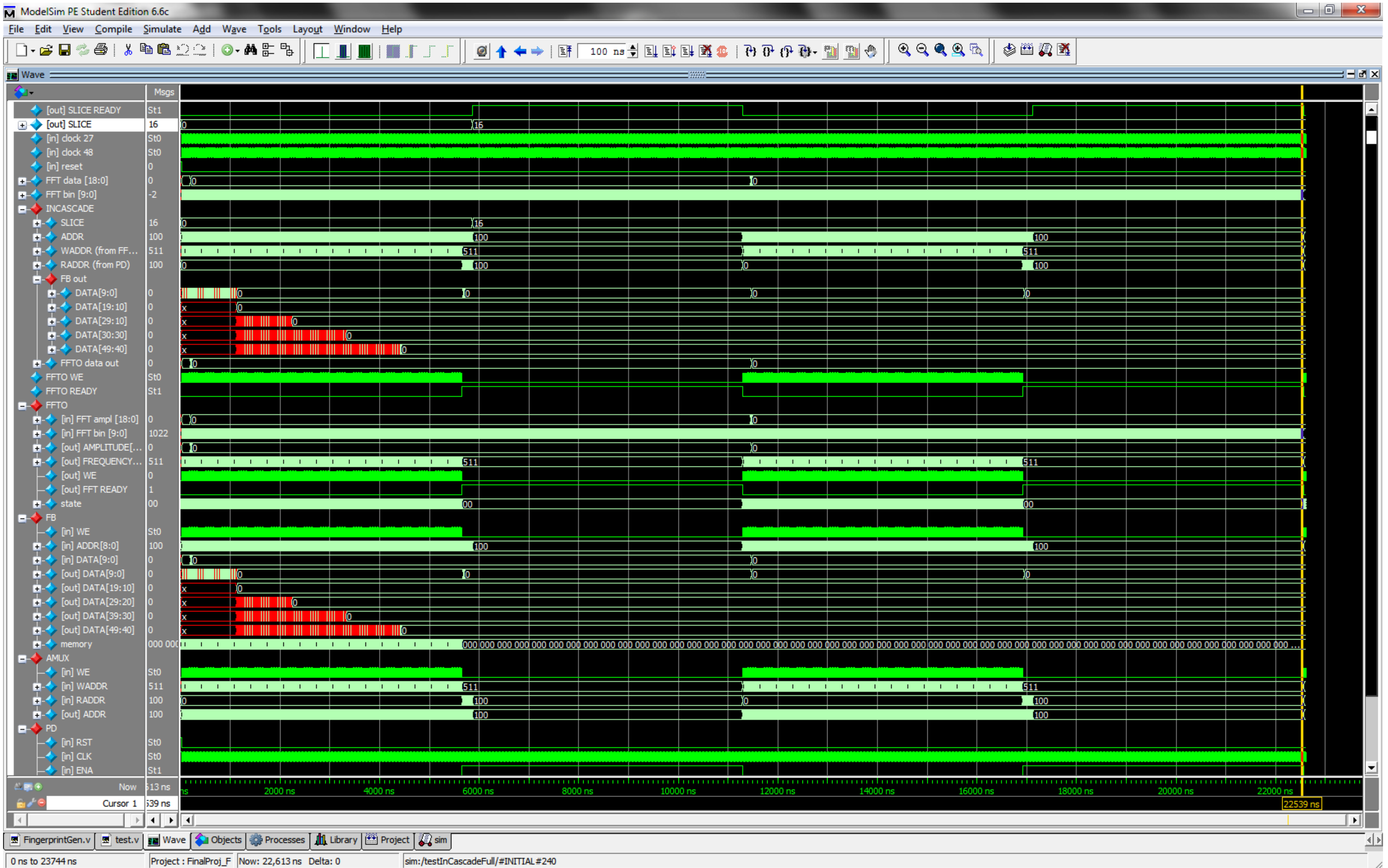


Figure I. A ModelSim simulation of the InputCascade Module – testInputCascade.

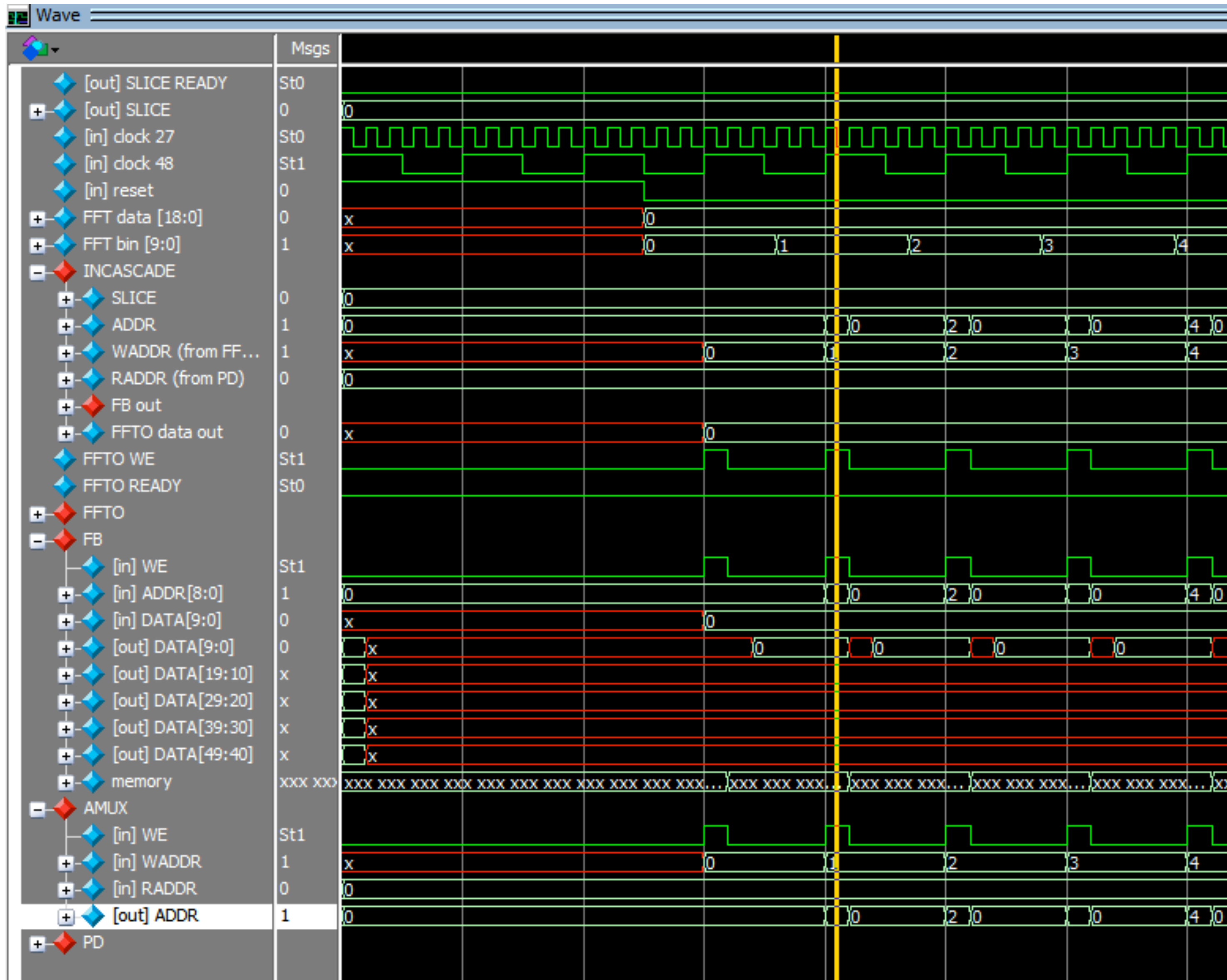


Figure II. testInputCascade – begin filling the Frame Buffer.

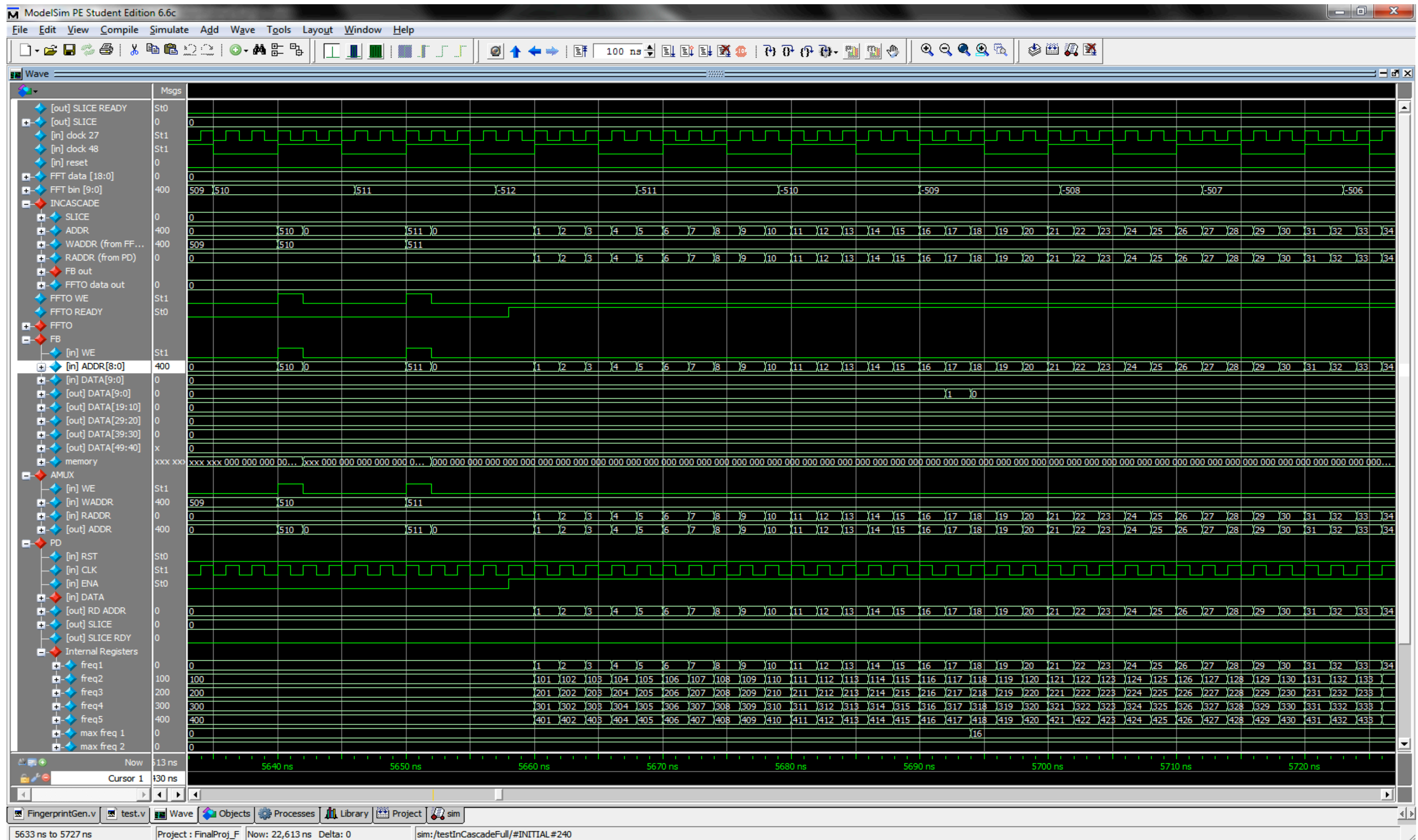


Figure III. testInputCascase – Finished filling the Frame Buffer, starting Peak Detection.

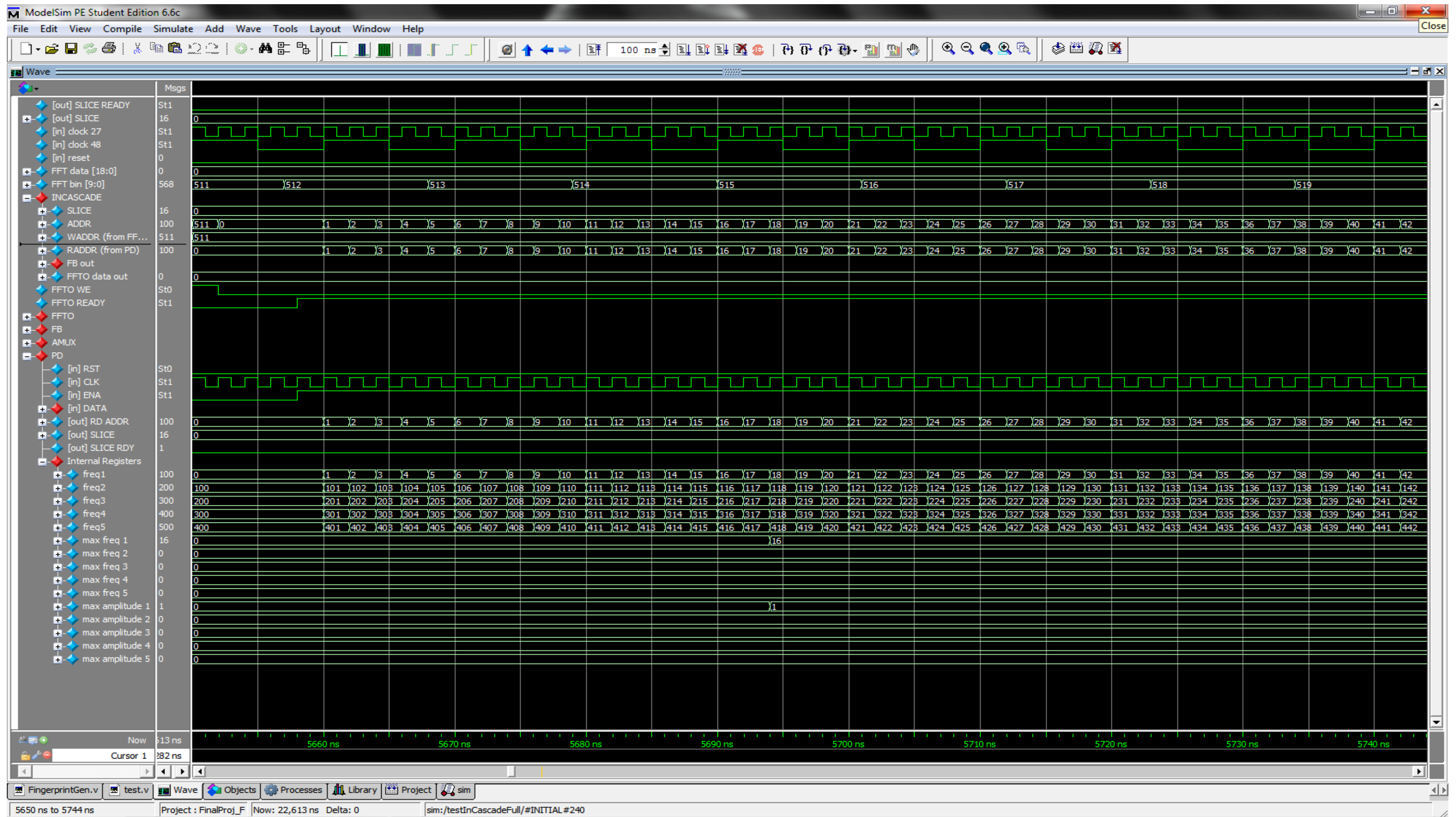


Figure IV. testInputCascade – Peak Detection in progress.

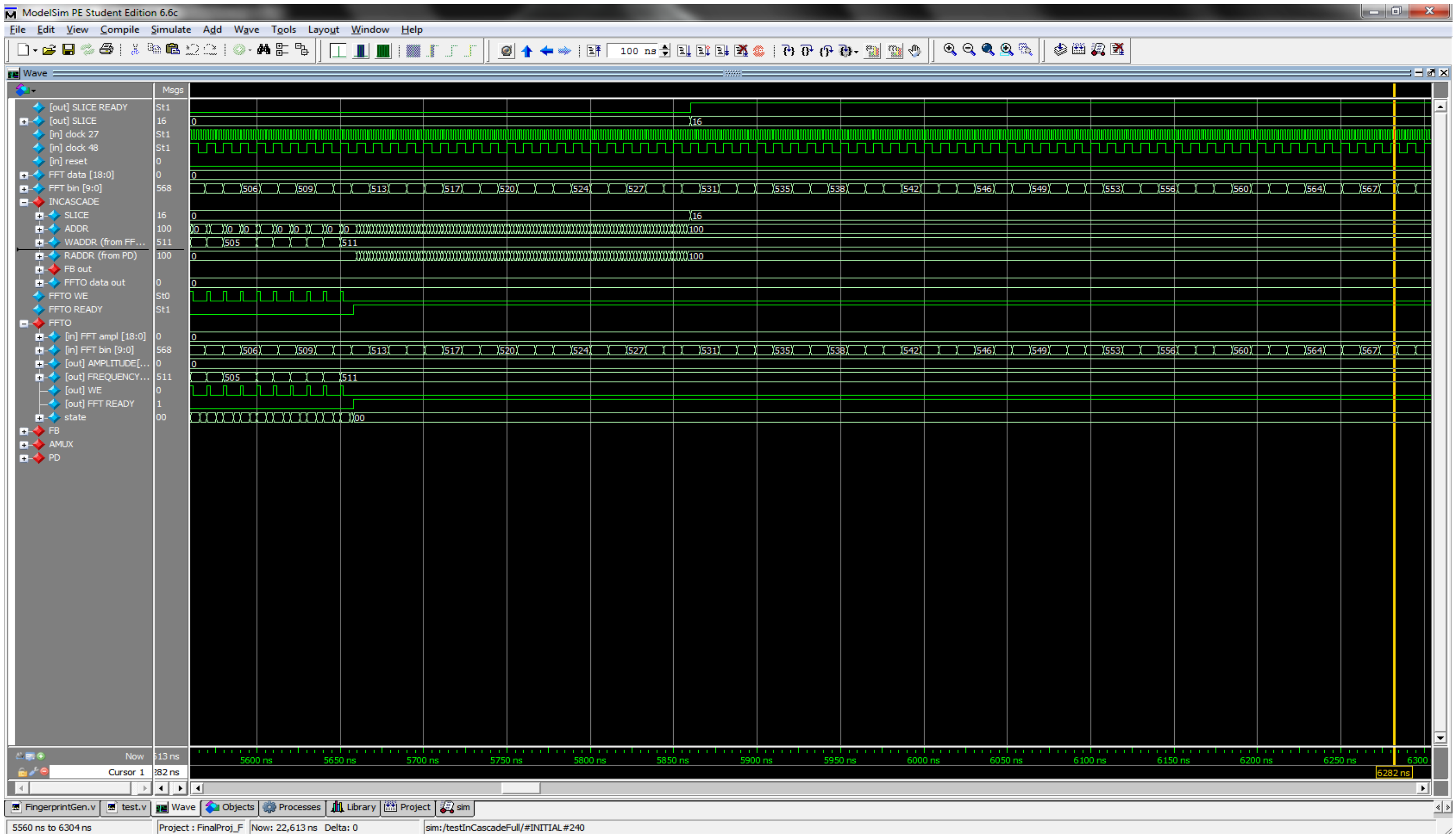


Figure V. Peak Detection done.

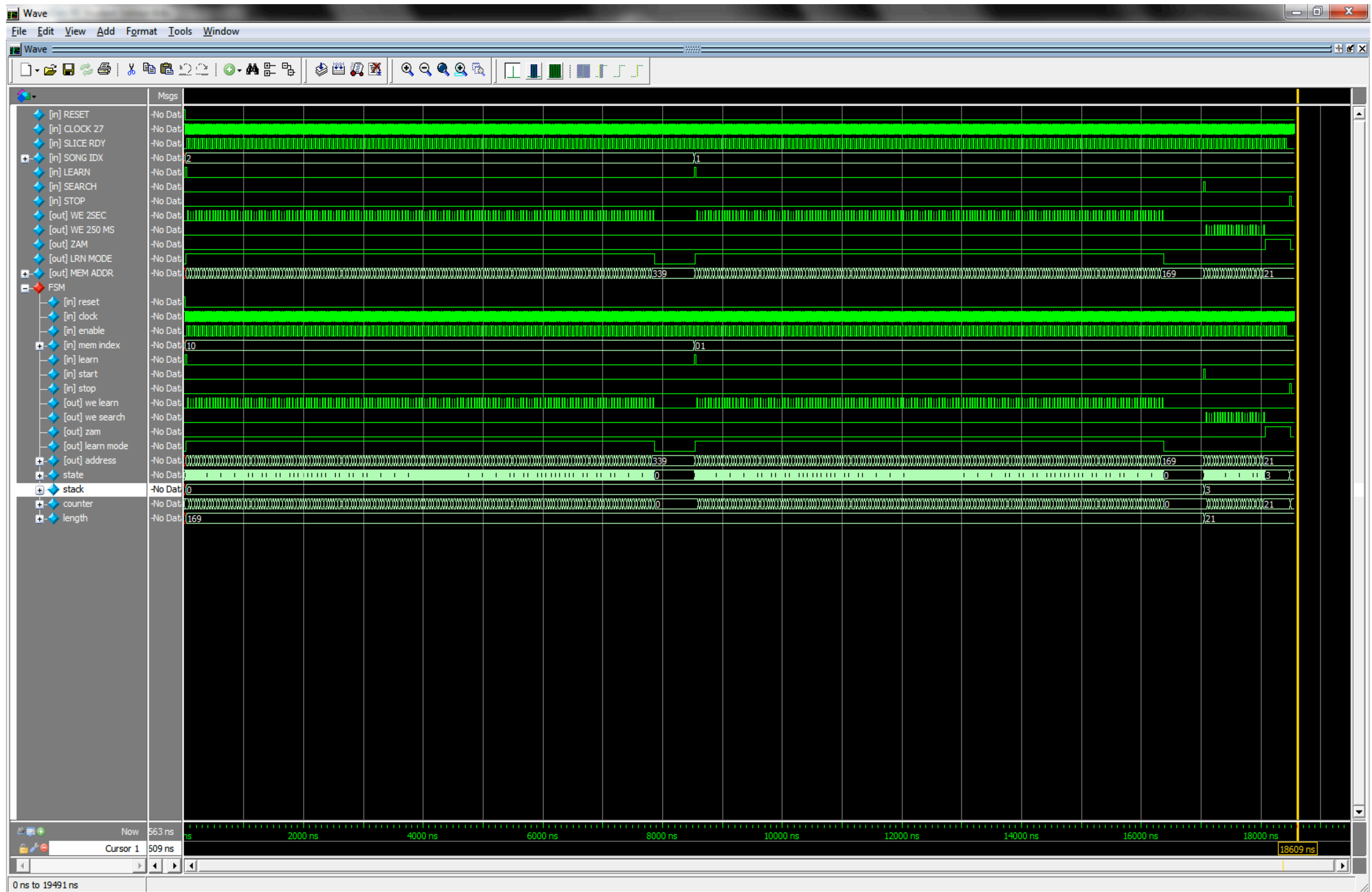


Figure VI. ZAM Finite State Machine test – testZAMFSM.

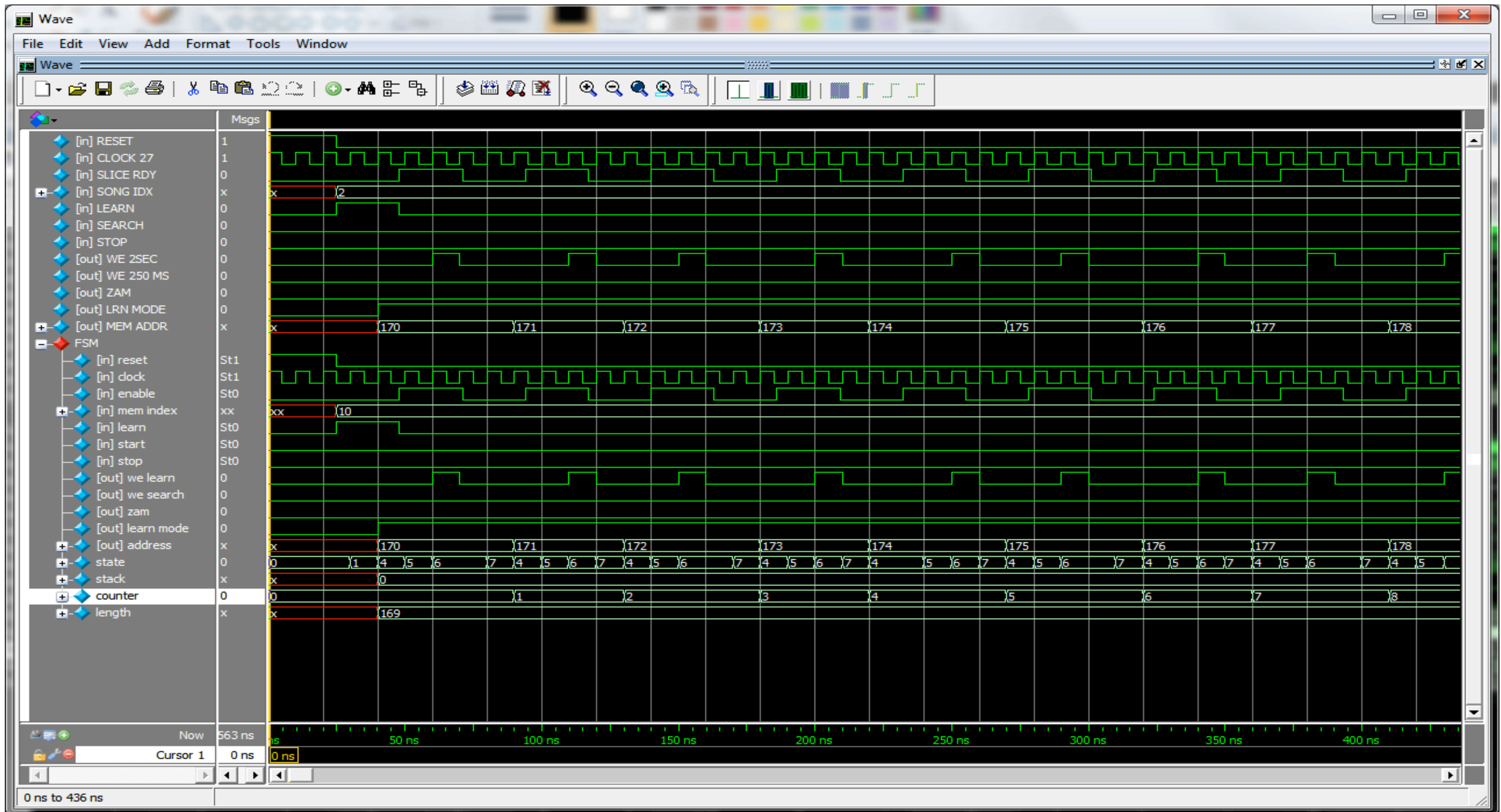


Figure VII. testZAMFSM – reset, and then enter learn mode.

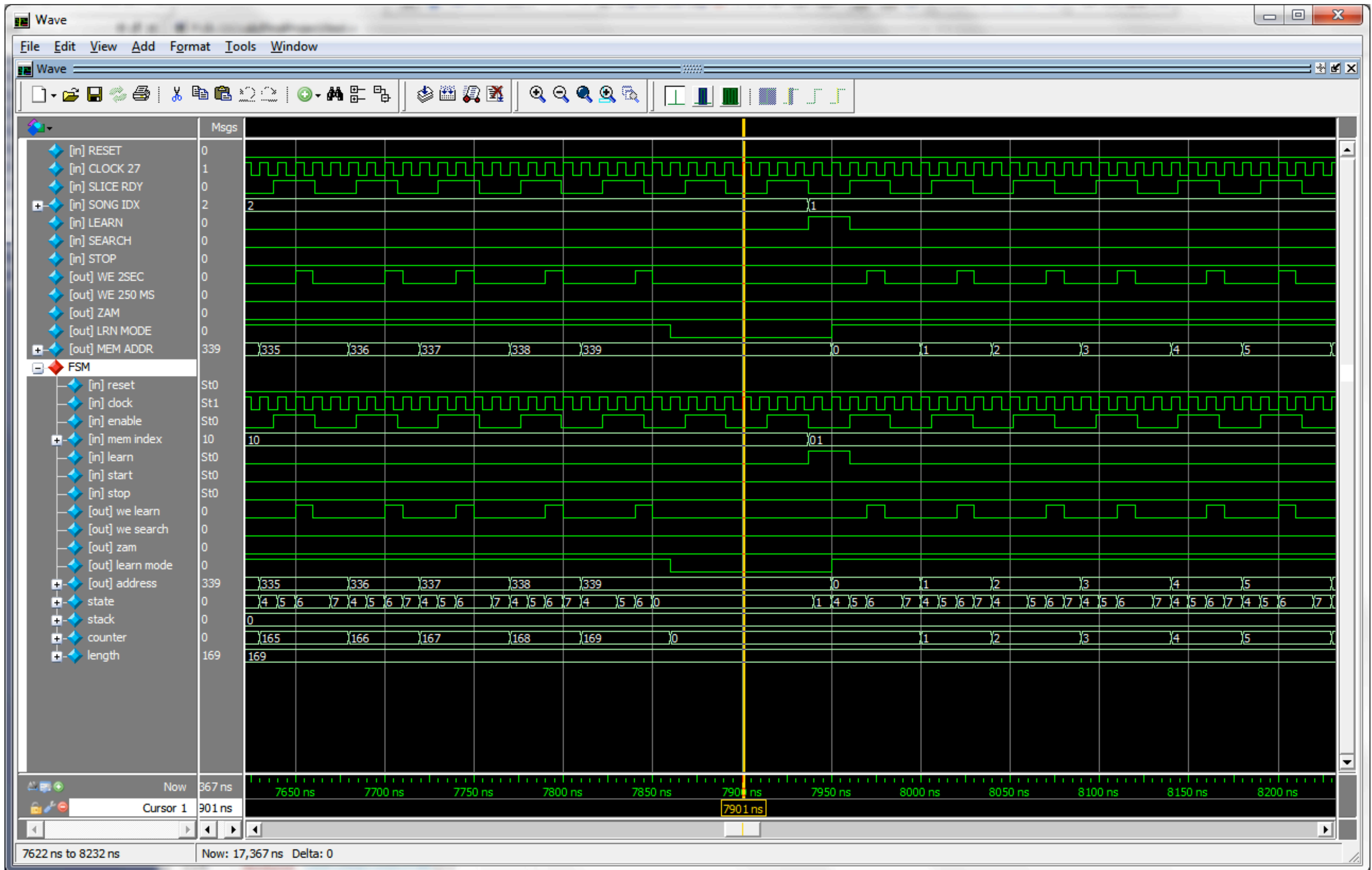


Figure VIII. Finished writing in bank 2 and starting to write into bank 1.

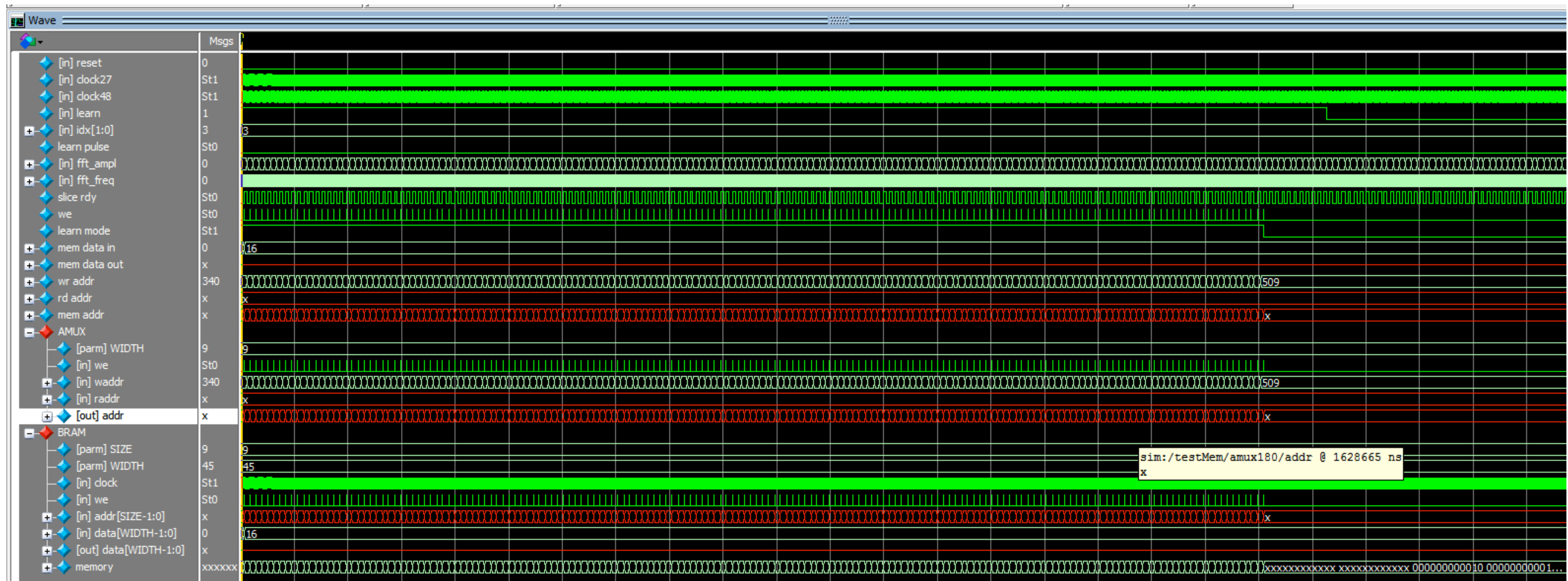


Figure IX. Memory test, writing into memory bank 3.

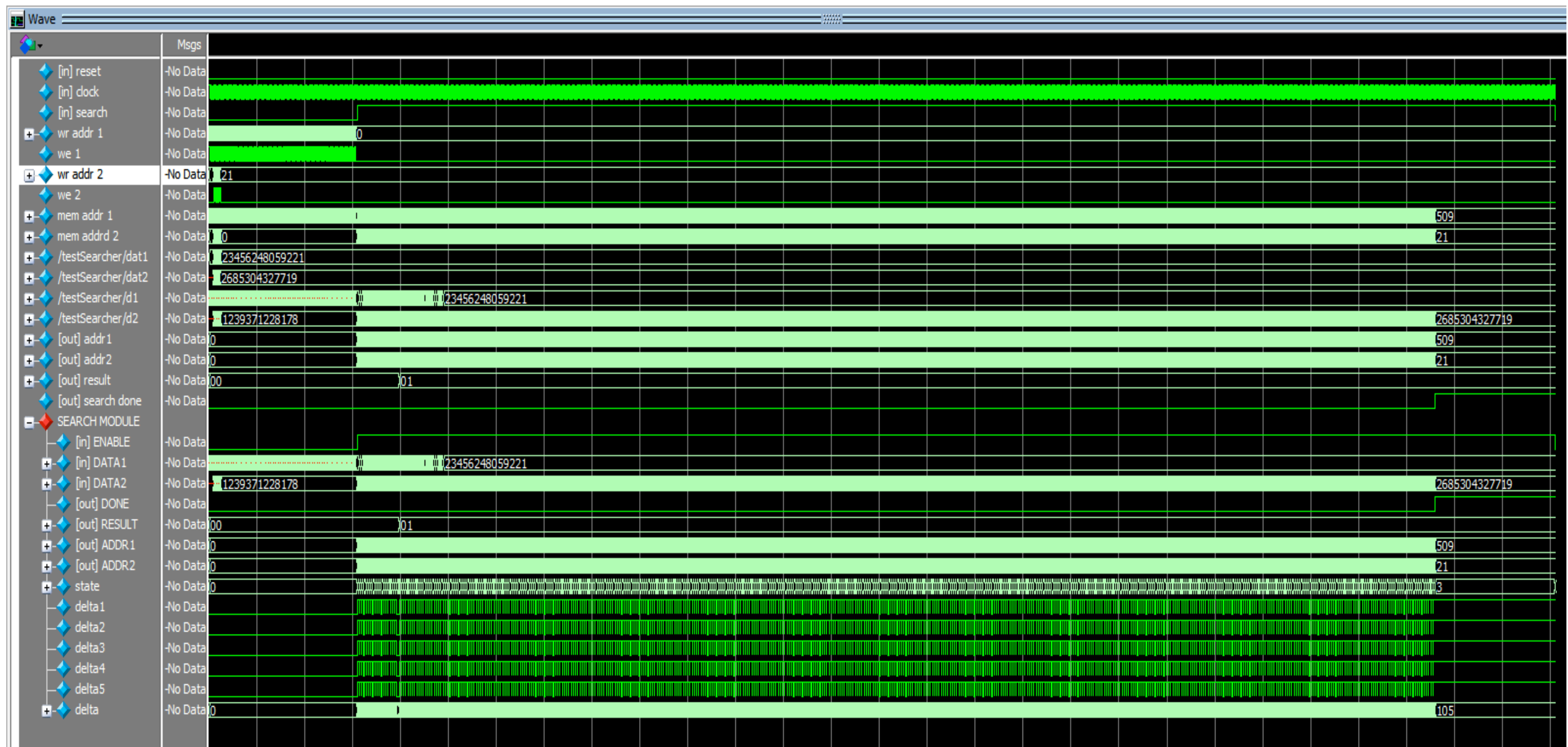


Figure X. testSearch – which tests the search module as described in the document body.

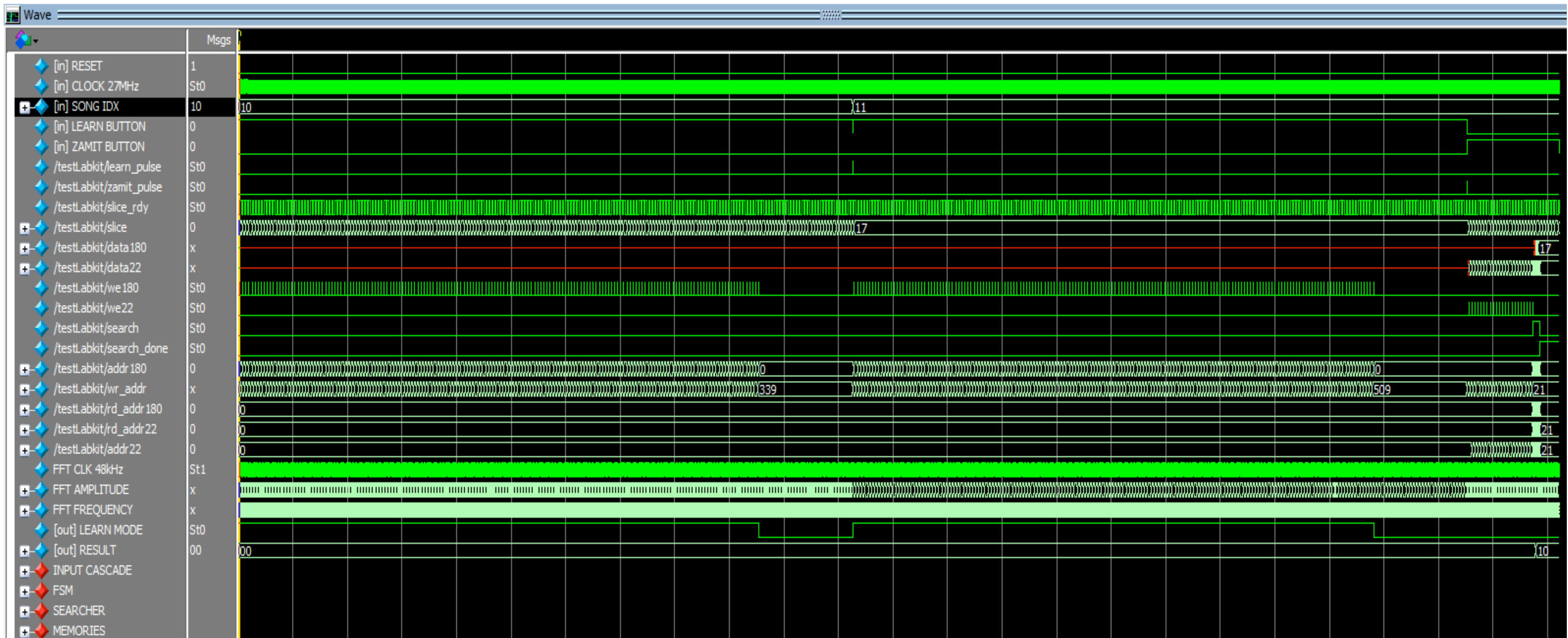


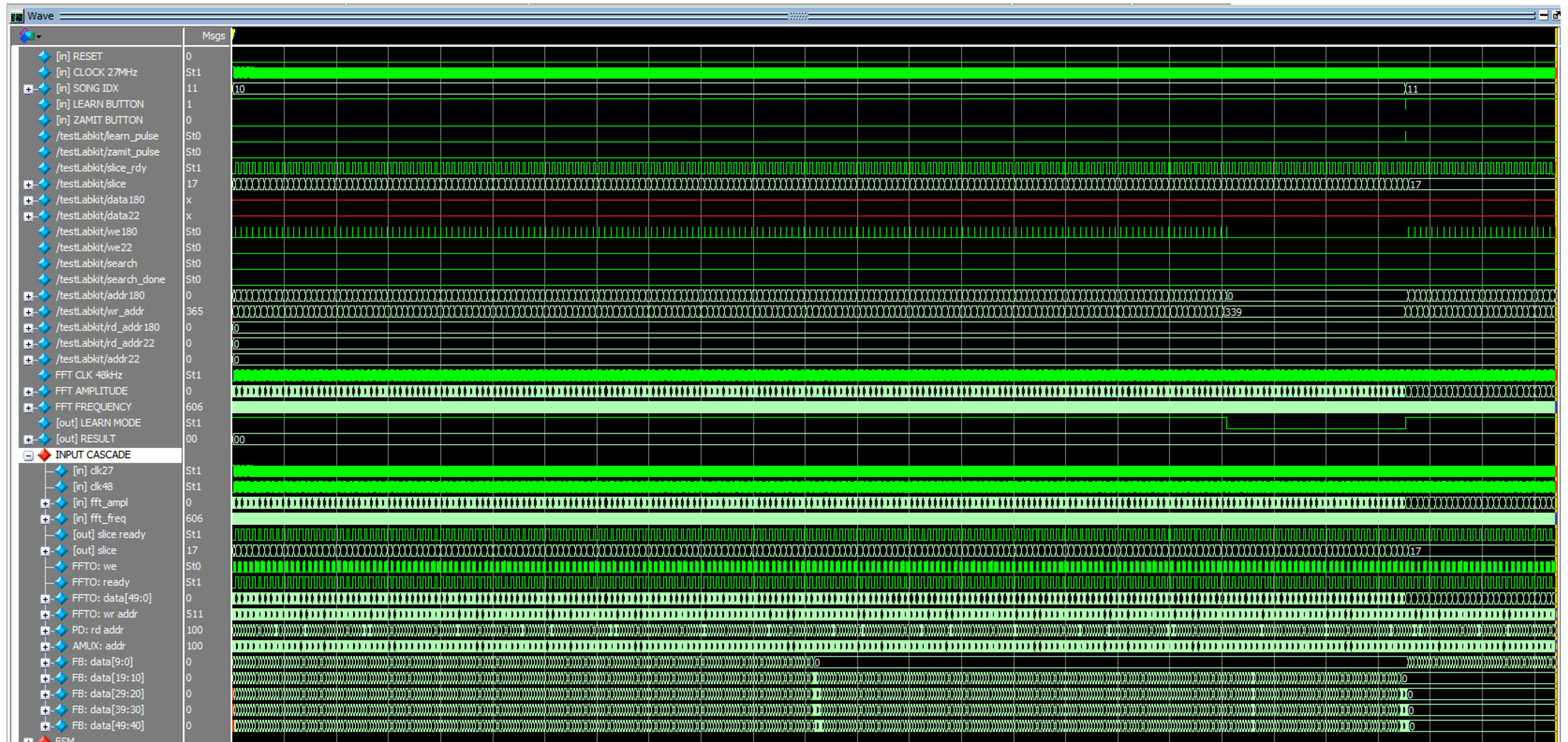
Figure XI. The complete test.

The complete test. What follows is a series of screenshots of the final tests.

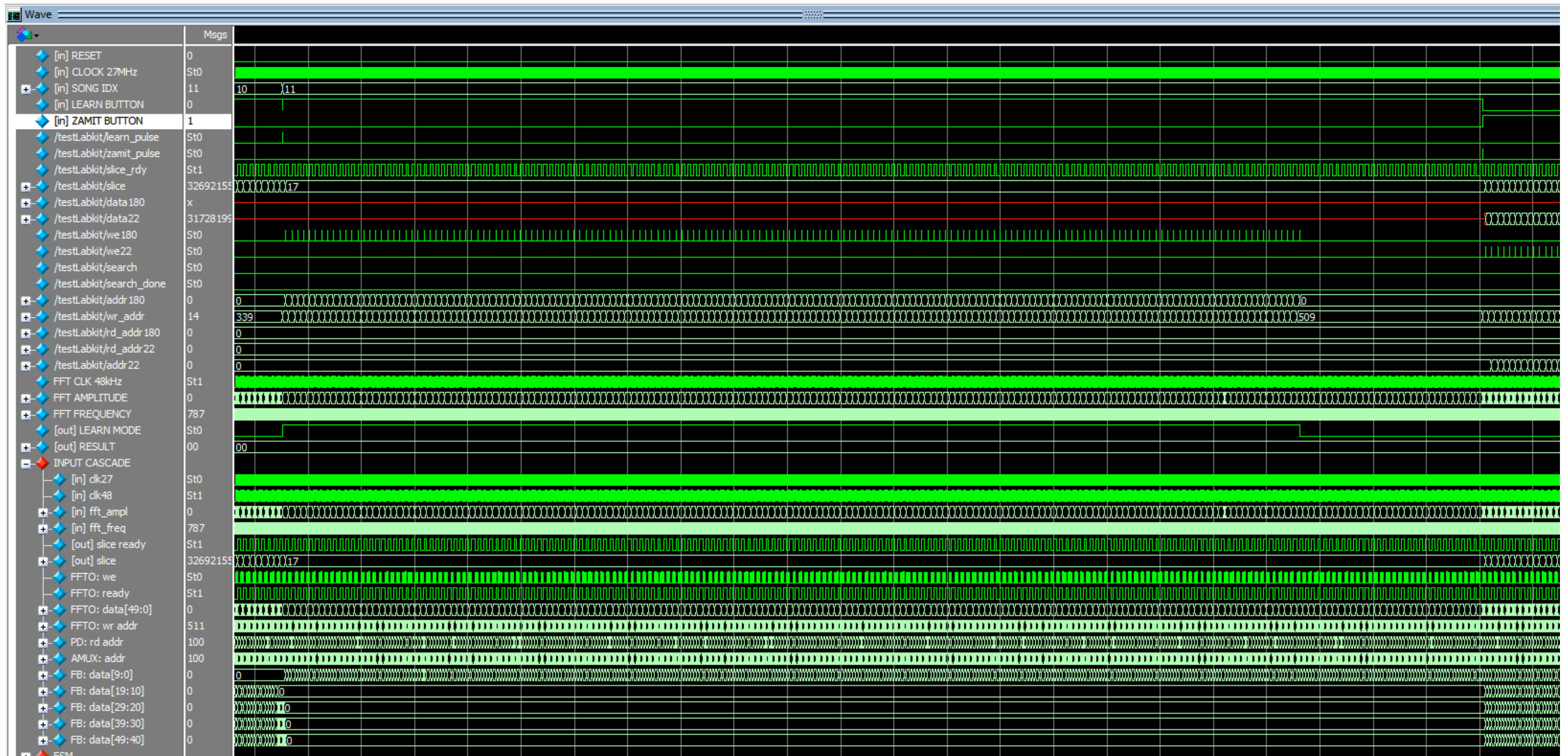
Phase one: learn song A, put it in bank 2

Phase two: learn song B, put it in bank 3

Phase three: search a fragment from song A



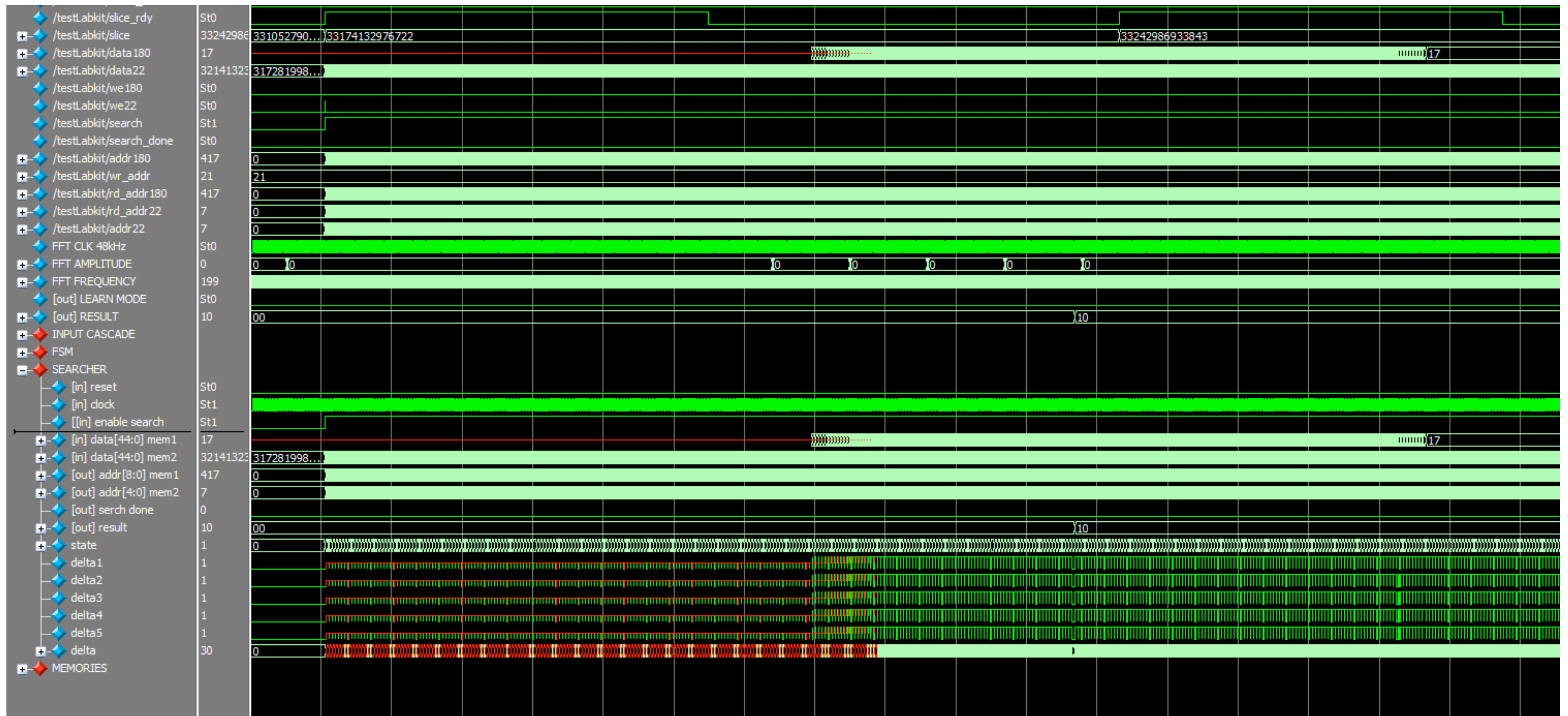
Phase 1



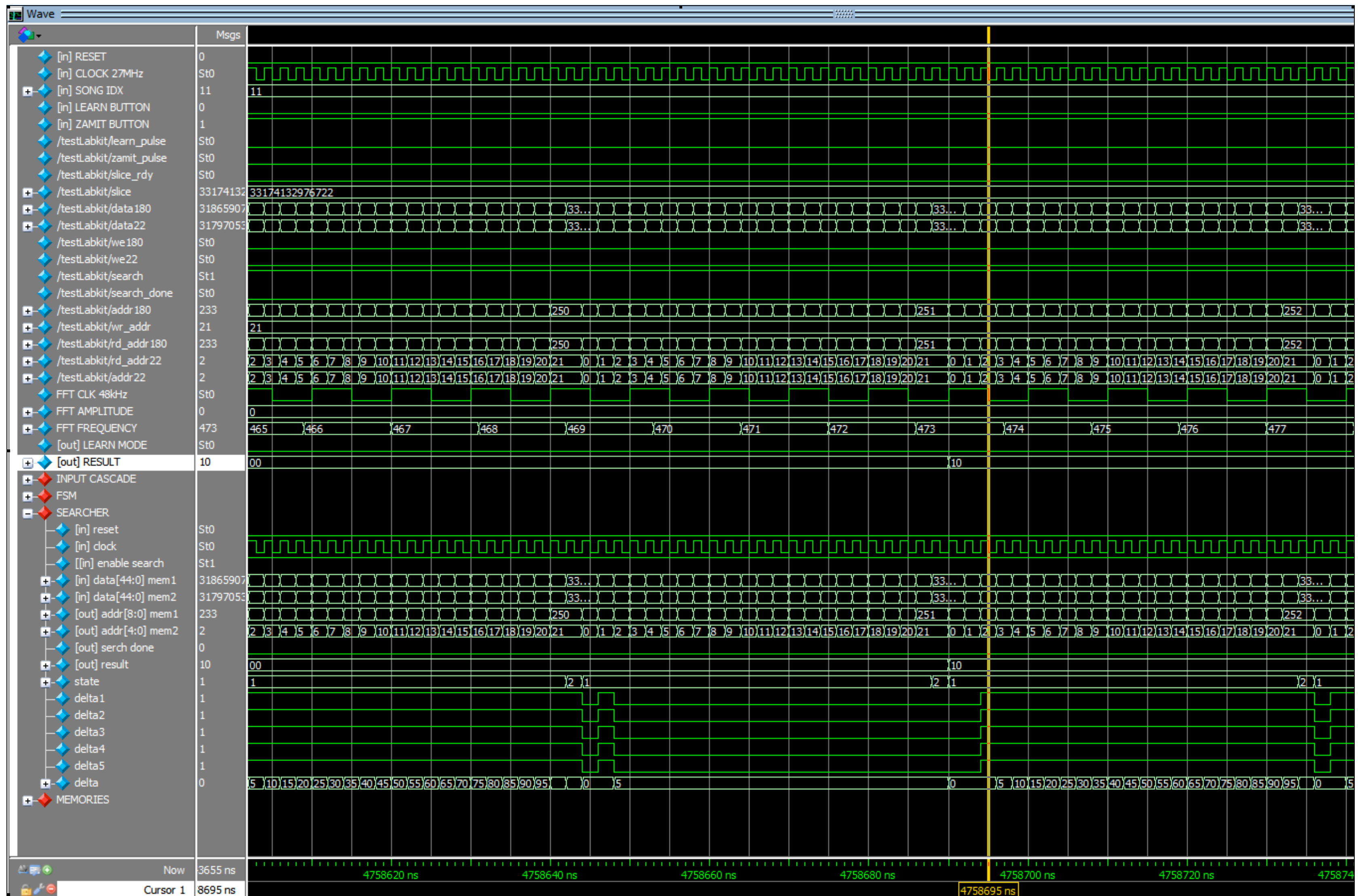
Phase 2

Phase 3

Fill memories



Search



Finally, the result!

Appendix B: Code

```
////////////////////////////////////////////////////////////////////////////
//
//   Yafim Landa    landa@mit.edu
//   Written for 6.111, Fall 2010
//
//   Description: This is the Input Cascade, which corresponds to Figure
//   8 in the accompanying report. It contains the SyncGen, and all of
//   the other modules are located in FingerprintGen.v.
//
////////////////////////////////////////////////////////////////////////////

//Generates a pulse that is one clock wide on posedge of input signal
module SyncGen(input clock, signal, output reg pulse);
  reg state;
  always @ (posedge clock)
  if (state) begin
    pulse <= 0;
    if (signal) state <= 1; //remain in state=1 while signal lasts
    else state <= 0;
  end
  else //wait for signal posedge
    if (signal) begin
      pulse <= 1;
      state <= 1;
    end
    else begin
      pulse <= 0;
      state <= 0;
    end
endmodule

//Generates fingerprints with control signal
//Consists of the following modules:
//FFT, FrameBuffer, AddressMultiplexer and FingerprintGen
module InputCascade(
  input reset, clock, fft_clk,
  input [18:0] fft_amp,
  input [9:0]  fft_idx, //sign in bit9
  output ready,
  output [44:0] slice);

  wire fft_we,fft_rdy;
```

```
  wire[8:0] abus,fft_adr,fpg_adr;
  wire[9:0] dbus1,dbus2,dbus3,dbus4,dbus5,fft_dat;

  FFTOutput   trns(.reset(reset), .clk27M(clock), .clk48K(fft_clk),
.fft_amp(fft_amp), .fft_idx(fft_idx),
                   .amp(fft_dat), .bin(fft_adr), .we(fft_we), .rdy(fft_rdy));
  FrameBuffer frbf(.clock(clock), .we(fft_we), .addr(abus), .din(fft_dat),
                   .dout1(dbus1), .dout2(dbus2), .dout3(dbus3), .dout4(dbus4),
.dout5(dbus5));
  AddressMux  amux(.write(fft_we), .waddr(fft_adr), .raddr(fpg_adr),
.addr(abus));
  FngprintGen fgen(.reset(reset), .clock(clock), .enable(fft_rdy),
                   .din1(dbus1), .din2(dbus2), .din3(dbus3), .din4(dbus4),
.din5(dbus5),
                   .adr(fpg_adr), .fingerprint(slice), .ready(ready));
endmodule

module FFTOutput (
  input reset, clk27M, clk48K,
  input [18:0] fft_amp,
  input [9:0] fft_idx,
  output reg [9:0] amp,
  output reg [8:0] bin,
  output reg we,
  output reg rdy);

  //state "00" wait fot fft_idx become 0
  //state "01" store one FFT value
  //state "10" wait for next byte
  reg [1:0] state;

  always @ (posedge clk27M) begin
    if (reset) begin
      state <= 2'b00;
      rdy <= 0;
      we <= 0;
    end
    else case (state)
      2'b00: state <= (fft_idx == 0) ? 2'b01 : 2'b00;
      2'b01: begin
        if (fft_idx >= 0 && fft_idx < 512) begin
          //process FFT results for positive frequencies
```

```
          //make one "we" pulse on each fft conversion
          if (clk48K == 1) begin
            amp <= fft_amp[18:9];
            bin <= fft_idx[8:0];
            rdy <= 0;
            we <= 1;
            state <= 2'b10;
          end
        end
        else begin
          rdy <= 1;
          we <= 0;
          state <= 2'b00;
        end
      end
      2'b10: begin
        if (fft_idx >= 0 && fft_idx < 512) begin
          we <= 0;
          if (clk48K == 0) state <= 2'b01;
        end
        else begin
          rdy <= 1;
          we <= 0;
          state <= 2'b00;
        end
      end
    endcase
  end
endmodule
```

```
////////////////////////////////////////////////////////////////////////////
//
//   Yafim Landa    landa@mit.edu
//   Written for 6.111, Fall 2010
//
//   Description: Contains the components needed for InputCascade to work.
//
////////////////////////////////////////////////////////////////////////////

//SIZE=9 addresses 9^b111111111 = 1FFh = 512 bytes
//WIDTH=10 denotes 10 bit in byte
module BRAM
 #(parameter SIZE=9, WIDTH=10)
  (input wire clock, we,
   input wire [SIZE-1:0] addr,
   input wire [WIDTH-1:0] din,
   output reg [WIDTH-1:0] dout);

  reg [WIDTH-1:0] mem[(1<<SIZE)-1:0];
  always @(posedge clock) begin
    if (we) mem[addr] <= din;
    dout <= mem[addr];
  end
endmodule

module FrameBuffer //buffer 512xWIDTH: 512 bytes, WIDTH bit in byte
 #(parameter WIDTH=10)
  (input wire clock, we,
   input wire [8:0] addr,
   input wire [WIDTH-1:0] din,
   output reg [WIDTH-1:0] dout1,dout2,dout3,dout4,dout5);

  reg [WIDTH-1:0] mem[511:0];
  always @(posedge clock) begin
    if (we) mem[addr] <= din;
    if (addr<100) begin
      dout1 <= mem[addr];
      dout2 <= mem[addr+100];
      dout3 <= mem[addr+200];
      dout4 <= mem[addr+300];
      dout5 <= mem[addr+400];
    end
```

```
    else begin
      dout1 <= 0; dout2 <= 0; dout3 <= 0; dout4 <= 0; dout5 <= 0;
    end
  end
endmodule

module AddressMux
 #(parameter WIDTH=9)
  (input wire write,
   input wire [WIDTH-1:0] waddr,raddr,
   output wire [WIDTH-1:0] addr);

  assign addr = write ? waddr : raddr;
endmodule

module FngprintGen
  (input reset,clock,enable,
   input wire [9:0] din1,din2,din3,din4,din5, //data
   output wire [8:0] adr,   //addr
   output reg [44:0] fingerprint,
   output reg ready);

  reg [8:0] f1c,f2c,f3c,f4c,f5c; //current frequency
  reg [8:0] f1,f2,f3,f4,f5; //max frequency
  reg [9:0] a1,a2,a3,a4,a5; //max amplitude

  assign adr = f1c;
  always @ (posedge clock) begin
    //adr <= f1c;

    if (reset) begin
      fingerprint <= 0;
      ready <= 0;
      //set cur freq to the bottom of bands
      //skip DC component (freq=0)
      f1c <= 0; f2c <= 100; f3c <= 200; f4c <= 300; f5c <= 400;
      f1 <= 0; f2 <= 0; f3 <= 0; f4 <= 0; f5 <= 0; //reset max freq
      a1 <= 0; a2 <= 0; a3 <= 0; a4 <= 0; a5 <= 0; //reset max amp
    end

    else begin
      if (enable) begin //FFT done, frame spectrogram in buffer
```

```
        if (f1c < 100) begin
          ready <= 0;

          if (f1c == 0) begin //reset result
            f1 <= 0; f2 <= 0; f3 <= 0; f4 <= 0; f5 <= 0;
          end

          if (din1>a1) begin f1 <= f1c-1; a1 <= din1; end
          if (din2>a2) begin f2 <= f2c-1; a2 <= din2; end
          if (din3>a3) begin f3 <= f3c-1; a3 <= din3; end
          if (din4>a4) begin f4 <= f4c-1; a4 <= din4; end
          if (din5>a5) begin f5 <= f5c-1; a5 <= din5; end

          f1c <= f1c + 1; f2c <= f2c + 1;
          f3c <= f3c + 1; f4c <= f4c + 1;
          f5c <= f5c + 1;
        end

        else begin //adr > 99
          ready <= 1;
          fingerprint <= {f5,f4,f3,f2,f1};
        end
      end

      else begin  //FFT working
        f1c <= 0; f2c <= 100; f3c <= 200; f4c <= 300; f5c <= 400;
        a1 <= 0; a2 <= 0; a3 <= 0; a4 <= 0; a5 <= 0; //reset max amp
        ready <= 0; //??????????????????????
      end

    end //not reset ends

  end  //on clock ends
endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
//   Yafim Landa    landa@mit.edu
//   Written for 6.111, Fall 2010
//
//   Description: This file contains the Searcher and FSM modules as shown
//   in Figure 11 of the accompanying report.
//
///////////////////////////////////////////////////////////////////////////////

 module Searcher(reset,clock,enable,din1,din2, adr1,adr2,done,result);
  input wire reset, clock, enable;
  input wire [44:0] din1, din2;
  output reg done;
  output reg [1:0] result;
  output reg [8:0] adr1;
  output reg [4:0] adr2;

  //define states
  reg [3:0] state;
  parameter IDLE = 0; //wait for "enable" go high
  parameter WORK = 1; //compares mem2 to mem1+offset
  parameter NEXT = 2; //increments offset
  parameter DONE = 3; //wait for "enable" go low

  //defien constants
  parameter MAXMEM1 = 509;  //179; //number of the last frame in 2sec
  parameter MAXMEM2 = 21;  //number of the last frame in 250ms
  parameter THRSHLD = 50;  //match threshold: res <= 1 if delta < THRES

  //define variables
  reg dlt1,dlt2,dlt3,dlt4,dlt5;
  reg [6:0] delta;
  reg [1:0] res;

  always @ (posedge clock) begin
    if (reset) begin
      adr1 <= 0;
      adr2 <= 0;
      done <= 0;
      dlt1 <= 0; dlt2 <= 0; dlt3 <= 0; dlt4 <= 0; dlt5 <= 0;
      delta <= 0;
```

```
      res <= 1;
      result <= 0;
      state <= IDLE;
    end

    else begin
      case (state)
        IDLE: begin
          if (enable) begin
            adr1 <= 0;
            adr2 <= 0;
            done <= 0;
            dlt1 <= 0; dlt2 <= 0; dlt3 <= 0; dlt4 <= 0; dlt5 <= 0;
            delta <= 0;
            res <= 1;
            result <= 0;
            state <= WORK;
          end
          else state <= IDLE;
        end

        WORK: begin
            delta <= delta + dlt1 + dlt2 + dlt3 + dlt4 + dlt5;
            dlt1 <= (din1[8:0]   != din2[8:0])   ? 1 : 0;
            dlt2 <= (din1[17:9]  != din2[17:9])  ? 1 : 0;
            dlt3 <= (din1[26:18] != din2[26:18]) ? 1 : 0;
            dlt4 <= (din1[35:27] != din2[35:27]) ? 1 : 0;
            dlt5 <= (din1[44:36] != din2[44:36]) ? 1 : 0;

            if (adr2 >= MAXMEM2) state <= NEXT;
            else begin
              if (adr1 == 170) res <= 2;
              if (adr1 == 340) res <= 3;
              adr1 <= adr1 + 1;
              adr2 <= adr2 + 1;
              state <= WORK;
            end
          end

        NEXT: begin
          if (delta < THRSHLD) result <= res;
          if (adr1 >= MAXMEM1) begin
```

```
            done <= 1;
            state <= DONE;
          end
          else begin
            adr1 <= adr1 - MAXMEM2 + 1;
            adr2 <= 0;
            dlt1 <= 0; dlt2 <= 0; dlt3 <= 0; dlt4 <= 0; dlt5 <= 0;
            delta <= 0;
            state <= WORK;
          end
        end

        default: state <= enable ? DONE : IDLE;
      endcase
    end
  end
endmodule

module ZAMFSM (
  input reset, clock, frame, learn, start, stop,
  input wire [1:0] idx,
  output reg mem1, mem2, srch, mode,
  output reg [8:0] addr );

  //define FSM states
  reg [3:0] state;
  parameter IDLE = 0;
  parameter LEARN = 1;
  parameter ZAM = 2;
  parameter SEARCH = 3;
  parameter FRM_WAIT1 = 4; //wait for "frame"=1
  parameter FRM_STORE = 5; //store the frame
  parameter FRM_WAIT2 = 6; //wait for "frame"=0
  parameter FRM_NEXT  = 7; //increment address

  //define FSM constants
  parameter CNT2SEC = 169; //number of frames in 2sec
  parameter CNT250MS = 21; //number of frames in 250ms

  //define FSM private variables
  reg [3:0] stack;         //state to return when count expires
  reg [8:0] cntr, length;  //current address count and max address to count
```

```
  always @(posedge clock) begin

    if (reset) begin
      state <= IDLE;
      mem1 <= 0;
      mem2 <= 0;
      srch <= 0;
      mode <= 0;
      cntr <= 0;
    end

    else case (state)
      IDLE: begin
        mem1 <= 0;
        mem2 <= 0;
        srch <= 0;
        mode <= 0;
        cntr <= 0;
        if (learn) state <= LEARN;
        else if (start) state <= ZAM;
        else state <= IDLE;
      end

      LEARN: begin
        if (idx == 0) state <= IDLE;
        else begin
          case (idx)
            1: addr <= 0;
            2: addr <= 170;
            default: addr <= 340;
          endcase
          mode <= 1;
          stack <= IDLE;
          state <= FRM_WAIT1;
          length <= CNT2SEC;
          state <= FRM_WAIT1;
        end
      end

      ZAM: begin
        addr <= 0;
```

```
        mode <= 0;
        stack <= SEARCH;
        length <= CNT250MS;
        state <= FRM_WAIT1;
      end

      //write cycle
      FRM_WAIT1: begin //wait for "frame"=1
        state <= frame ? FRM_STORE : FRM_WAIT1;
      end
      FRM_STORE: begin //store the frame
        mem1 <= mode ? 1 : 0;
        mem2 <= mode ? 0 : 1;
        state <= FRM_WAIT2;
      end
      FRM_WAIT2: begin//wait for "frame"=0
        mem1 <= 0;
        mem2 <= 0;
        if (cntr < length) state <= frame ? FRM_WAIT2 : FRM_NEXT;
        else state <= stack;
      end
      FRM_NEXT: begin
        addr <= addr + 1;
        cntr <= cntr + 1;
        state <= FRM_WAIT1;
      end

      default: begin  //SEARCH
        mem1 <= 0;
        mem2 <= 0;
        srch <= 1;
        if (stop) state <= IDLE;
      end
    endcase
  end
endmodule
```

```
/*
_////////_///////     _////                  _/
_//      _//    _// _/    _//               _/ //
_//      _//    _//_//        _//// _//    _/  _//    _/// _// _//
_//////  _///////  _//             _//    _//   _//    _//  _/  _//
_//      _//       _//   _////   _//     _////// _//   _//  _/  _//
_//      _//        _//    _/   _//     _//       _//  _//  _/  _//
_//      _//         _/////   _////////_//         _//_///  _/  _//

                    Yafim Landa   and    Pranav Sood
                   landa@mit.edu        soodp@mit.edu
*/

// Please note: this project should compile and run, however, it is missing
// the generated FFT module and COE modules. To compile and run this project,
// please generate these files as described in the accompanying report.

///////////////////////////////////////////////////////////////////////////////
//
// Switch Debounce Module
//
///////////////////////////////////////////////////////////////////////////////
module debounce (input reset, clock, noisy, output reg clean);
   reg [18:0] count;
    reg new;

   always @(posedge clock)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 270000) clean <= new;
     else count <= count+1;
endmodule
//////////////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////

module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
              ac97_bit_clock,

              vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
              vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
              vga_out_vsync,
```

```
            tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
            tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
            tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

            tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
            tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
            tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
            tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

            ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
            ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

            ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
            ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

            clock_feedback_out, clock_feedback_in,

            flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
            flash_reset_b, flash_sts, flash_byte_b,

            rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

            mouse_clock, mouse_data, keyboard_clock, keyboard_data,

            clock_27mhz, clock1, clock2,

            disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
            disp_reset_b, disp_data_in,

            button0, button1, button2, button3, button_enter, button_right,
            button_left, button_down, button_up,

            switch,

            led,

            user1, user2, user3, user4,

            daughtercard,

            systemace_data, systemace_address, systemace_ce_b,
            systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,
```

```
              analyzer1_data, analyzer1_clock,
              analyzer2_data, analyzer2_clock,
              analyzer3_data, analyzer3_clock,
              analyzer4_data, analyzer4_clock);

output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
input  ac97_bit_clock, ac97_sdata_in;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
        vga_out_hsync, vga_out_vsync;

output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
        tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
        tv_out_subcar_reset;

input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
        tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
        tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

inout  [35:0] ram1_data;
output [18:0] ram1_address;
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output [3:0] ram1_bwe_b;

input  clock_feedback_in;
output clock_feedback_out;

inout  [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;
```

```
 output rs232_txd, rs232_rts;
 input  rs232_rxd, rs232_cts;

 input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

 input  clock_27mhz, clock1, clock2;

 output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
 input  disp_data_in;
 output  disp_data_out;

 input  button0, button1, button2, button3, button_enter, button_right,
         button_left, button_down, button_up;
 input  [7:0] switch;
 output [7:0] led;

 inout [31:0] user1, user2, user3, user4;

 inout [43:0] daughtercard;

 inout  [15:0] systemace_data;
 output [6:0]  systemace_address;
 output systemace_ce_b, systemace_we_b, systemace_oe_b;
 input  systemace_irq, systemace_mpbrdy;

 output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
                analyzer4_data;
 output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;
 ////////////////////////////////////////////////////////////////////////////
////////////////////////////////

 ///////////////////////////////////////////////////////////////////////////
 //
 // RESET Generation
 //
 ///////////////////////////////////////////////////////////////////////////
 wire pwr_reset, btn_reset, reset;
  SRL16 reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(pwr_reset), .A0(1'b1),
.A1(1'b1), .A2(1'b1), .A3(1'b1));
  defparam reset_sr.INIT = 16'hFFFF;
  debounce db1(.reset(pwr_reset), .clock(clock_27mhz), .noisy(~button_enter),
```

```
.clean(btn_reset));
  assign reset = pwr_reset || btn_reset:

 ////////////////////////////////////////////////////////////////////////////
 //
 // I/O Assignments
 //
 ////////////////////////////////////////////////////////////////////////////

 // Audio Input and Output
 assign beep= 1'b0;
// assign audio_reset_b = 1'b0;
 //assign ac97_synch = 1'b0;
 //assign ac97_sdata_out = 1'b0;
 // ac97_sdata_in is an input

 // VGA Output
// assign vga_out_red = 8'h0;
// assign vga_out_green = 8'h0;
// assign vga_out_blue = 8'h0;
// assign vga_out_sync_b = 1'b1;
// assign vga_out_blank_b = 1'b1;
// assign vga_out_pixel_clock = 1'b0;
// assign vga_out_hsync = 1'b0;
// assign vga_out_vsync = 1'b0;

 // Video Output
 assign tv_out_ycrcb = 10'h0;
 assign tv_out_reset_b = 1'b0;
 assign tv_out_clock = 1'b0;
 assign tv_out_i2c_clock = 1'b0;
 assign tv_out_i2c_data = 1'b0;
 assign tv_out_pal_ntsc = 1'b0;
 assign tv_out_hsync_b = 1'b1;
 assign tv_out_vsync_b = 1'b1;
 assign tv_out_blank_b = 1'b1;
 assign tv_out_subcar_reset = 1'b0;

 // Video Input
 assign tv_in_i2c_clock = 1'b0;
 assign tv_in_fifo_read = 1'b0;
 assign tv_in_fifo_clock = 1'b0;
```

```
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
assign ram0_address = 19'h0;
assign ram0_adv_ld = 1'b0;
assign ram0_clk = 1'b0;
assign ram0_cen_b = 1'b1;
assign ram0_ce_b = 1'b1;
assign ram0_oe_b = 1'b1;
assign ram0_we_b = 1'b1;
assign ram0_bwe_b = 4'hF;
assign ram1_data = 36'hZ;
assign ram1_address = 19'h0;
assign ram1_adv_ld = 1'b0;
assign ram1_clk = 1'b0;
assign ram1_cen_b = 1'b1;
assign ram1_ce_b = 1'b1;
assign ram1_oe_b = 1'b1;
assign ram1_we_b = 1'b1;
assign ram1_bwe_b = 4'hF;
assign clock_feedback_out = 1'b0;
// clock_feedback_in is an input

// Flash ROM
assign flash_data = 16'hZ;
assign flash_address = 24'h0;
assign flash_ce_b = 1'b1;
assign flash_oe_b = 1'b1;
assign flash_we_b = 1'b1;
assign flash_reset_b = 1'b0;
assign flash_byte_b = 1'b1;
// flash_sts is an input

// RS-232 Interface
assign rs232_txd = 1'b1;
assign rs232_rts = 1'b1;
```

```
 // rs232_rxd and rs232_cts are inputs

 // PS/2 Ports
 // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

 // LED Displays
   assign disp_blank = 1'b1;
   assign disp_clock = 1'b0;
   assign disp_rs = 1'b0;
   assign disp_ce_b = 1'b1;
   assign disp_reset_b = 1'b0;
   assign disp_data_out = 1'b0;
 // disp_data_in is an input

  // Buttons, Switches, and Individual LEDs
  // assign led = 8'hFF;
  // button0, button1, button2, button3, button_enter, button_right,
  // button_left, button_down, button_up, and switches are inputs
  wire learn, zamit, lrn_mode;
  wire [1:0] result;
  debounce db2(.reset(pwr_reset), .clock(clock_27mhz), .noisy(~button1),
.clean(learn));
  debounce db3(.reset(pwr_reset), .clock(clock_27mhz), .noisy(~button2),
.clean(zamit));
  assign led[5] = lrn_mode;
  assign led[6] = ~result[0];
  assign led[7] = ~result[1];



 // User I/Os
 assign user1 = 32'hZ;
 assign user2 = 32'hZ;
 assign user3 = 32'hZ;
 assign user4 = 32'hZ;

 // Daughtercard Connectors
 assign daughtercard = 44'hZ;

 // SystemACE Microprocessor Port
 assign systemace_data = 16'hZ;
 assign systemace_address = 7'h0;
 assign systemace_ce_b = 1'b1;
```

```
 assign systemace_we_b = 1'b1;
 assign systemace_oe_b = 1'b1;
 // systemace_irq and systemace_mpbrdy are inputs

// Logic Analyzer
//   assign analyzer1_data = 16'h0;
//   assign analyzer1_clock = 1'b1;
//   assign analyzer2_data = 16'h0;
//   assign analyzer2_clock = 1'b1;
//   assign analyzer3_data = 16'h0;
//   assign analyzer3_clock = 1'b1;
//   assign analyzer4_data = 16'h0;
//   assign analyzer4_clock = 1'b1;

////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////
//Assign wires registers in the top module
////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////

wire kclk;// wire for the 48khz clock

wire clock ;
assign clock = clock_27mhz;

wire r_f;// wire fo the the output ready of FFT
assign led[0] = ~(r_f);

wire d;// wire for the data valid output of FFT
assign led[1] = ~d;

wire done ;// wire for the done signal.
assign led[2] = ~done;

wire busy;//wire for the busy signal of the FFT.
assign led[3] = ~busy;

wire e; // wire for the early done signal
assign led[4] = ~e;

reg rstart;//reg that is used for controling the start of the FFT
```

```
wire ready; // wire for the ready signal used in the ac97 codec.

wire  [9:0] n_index,k_in;// n_index is the index of the input and k_in is the
output data index.
wire signed [18:0] k_re, k_im;// k_re is the real part of the output and k_im
is the imaginary part of the output.

wire signed [7:0] tone_input,stingtone,btone;// tone_input is the input to the
FFT. stingtone is the output from the coe file of the sting song
// btone is the wire for the output for the boom boom pow song.

 reg  [16:0] ap,an,counter; // counter is a reg that incremets with address of
the coe file and is used for 50% overlap.
 // reg an is used to increment the address of the coe file during the search.
 //reg ap is used to increment the address of the coe file while playing the
audio through the ac97.

 wire [16:0] a;// wire a is either an or ap.

 wire  signed [38:0] sq_summ;// takes the value of the sum of squares of the
real and imaginary part in order to compare the intensities during search.

 // instantiate the sting coe file.
  stingcoe stcoe(.clk(clock_27mhz),.addr(a),.dout(stingtone));
  // instantiate the boom boom pow coe file.
  coeboomboom  cbb(.clk(clock_27mhz),.addr(a),.dout(btone));

 wire reset1=1'b0// reset1 is used the module where the 48khz is generated.

 //instantiate the module for the 48khz clock.
 k_clk  cllk(.clk(clock),.reset(reset1),.kclk(kclk));

 wire [7:0]  to_ac97_data;
 wire [7:0] from_ac97_data;

 //instantiate the sum of squares module.
 wire signed [38:0] k_re_sqr, k_im_sqr;
 sqrnsum
sns(.k_re(k_re),.k_im(k_im),.sq_sum(sq_summ),.k_re_sqr(k_re_sqr),.k_im_sqr(k_i
_sqr));

// mux the input into the the FFT module.
```

```
assign tone_input = switch[0] ? btone : stingtone;

//instantiate the FFT module.
zamfft my_fft(
               .ce(kclk),           // clock enable FFT at 48khz
               .fwd_inv_we (1'b0),  // forward transform
               .rfd(r_f),           // ready for data output, not  used
               .start(rstart),
               .fwd_inv(1'b1),      // forward transform
               .dv(d),              // data valid output not used
               .done(done),         // done output
               .clk(clock_27mhz),   // fft clock
               .busy(busy),         // busy output
               .edone(e),           // early done
               .xn_re(tone_input),        // input tone
               .xn_im(8'b0),
               .xn_index(n_index),
               .xk_re(k_re),
               .xk_im(k_im),
               .xk_index(k_in),
                         .sclr(reset)          // master reset of the FFT
               );

// landa@mit.edu --
  wire learn_pulse, zamit_pulse;
  wire[44:0] slice, data180, data22;
  wire slice_rdy, we180, we22, search, search_done;
  wire[8:0] wr_addr, rd_addr180, addr180;
  wire[4:0] rd_addr22, addr22;
  InputCascade inp1 (
    .reset(reset), .clock(clock_27mhz), .fft_clk(kclk),
.fft_amp(sq_summ[24:6]), .fft_idx(k_in),
    .ready(slice_rdy), .slice(slice));

  SyncGen sync1 (.clock(clock_27mhz), .signal(learn), .pulse(learn_pulse));
  SyncGen sync2 (.clock(clock_27mhz), .signal(zamit), .pulse(zamit_pulse));
  ZAMFSM fsm1 (
    .reset(reset), .clock(clock_27mhz), .frame(slice_rdy),
    .idx(switch[7:6]), .learn(learn_pulse), .start(zamit_pulse),
.stop(search_done),
    .mem1(we180), .mem2(we22), .srch(search), .mode(lrn_mode), .addr(wr_addr));
```

```
  AddressMux #(.WIDTH(9))
  amux180(.write(we180), .waddr(wr_addr), .raddr(rd_addr180), .addr(addr180));
  BRAM #(.SIZE(9), .WIDTH(45))
  mem180 (.clock(clock_27mhz), .we(we180), .addr(addr180), .din(slice),
.dout(data180));

  AddressMux #(.WIDTH(5))
  amux22(.write(we22), .waddr(wr_addr[4:0]), .raddr(rd_addr22), .addr(addr22));
  BRAM #(.SIZE(5), .WIDTH(45))
  mem22 (.clock(clock_27mhz), .we(we22), .addr(addr22), .din(slice),
.dout(data22));

  Searcher src1(.reset(reset), .clock(clock_27mhz), .enable(search),
.din1(data180), .din2(data22),
    .adr1(rd_addr180), .adr2(rd_addr22), .done(search_done), .result(result));
// --




    // allow user to adjust volume
   wire vup,vdown;
   reg old_vup,old_vdown;
   debounce bup(reset, clock_27mhz, ~button_up, vup);
   debounce bdown(reset, clock_27mhz, ~button_down, vdown);
   reg [4:0] volume;
   always @ (posedge clock_27mhz) begin
     if (reset) volume <= 5'd8;
     else begin
    if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
    if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
     end
     old_vup <= vup;
     old_vdown <= vdown;
   end


   // AC97 driver
   fft_audio acc(clock_27mhz, reset, volume, from_ac97_data, to_ac97_data,
ready,
           audio_reset_b, ac97_sdata_out, ac97_sdata_in,
           ac97_synch, ac97_bit_clock);
```

```
// mux the address into the coe file.
assign a = switch[1] ? an : ap ;

  always @(posedge kclk) begin
    if (reset || learn_pulse || zamit_pulse) begin// if the reset ,learn or
zammit buttons are hit the address resets
    ap <= 0;                                     //  and the also counter
resets
    counter<=0;
    rstart <=0;
    end
    else if  (counter == 16'd1023)begin// if the counter reaches a value of
1023 , the address recues by 511 in the next clock cycle.
   ap<= ap-15'd511;                  // and the counter restes to 0.
    counter<= 0;
    rstart<=1'b1;
    end
    else begin // if none of the above happens the adrress and the counter
increments
    counter <= counter+1; // and the FFt is started.
    ap<=ap+1;
   rstart<= 1'b1;
   end
end
//reg for playing the coe files.
reg [7:0] op;
reg [7:0] next_to_ac97_data;

// logic for playing the matched song.
always @ * begin
if (~led[6] && ~switch[0])
next_to_ac97_data = btone;
else if (~led[6] && switch[0])
next_to_ac97_data = stingtone;
else if (~led[7] && ~switch[0])
    next_to_ac97_data = btone;
else if (~led[7] && switch[0])
    next_to_ac97_data = stingtone;
else
    next_to_ac97_data = from_ac97_data;
```

```
    end

    always@ (posedge ready) begin
    op<= next_to_ac97_data;
    if (~switch[1])
        an<=0;
        else
         an<=an+1;
  end

assign to_ac97_data = op;
///////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////
////////////////////////////////////////////////////////
// Logic Analyzer Settings
// CK0            CK0()
// learn_pulse    A3(6)
// zamit_pulse      A3(5)
// dbg_state        A3(4-1)                             unused
// search_done      A3(0)
// result           A2(7)
// we180          A2(1)
// we22           A2(0)
// data180          D1(7-0)
// data22           D0(7-0)
// addr180        C3(7-0)
// addr22         C2(7-0)
////////////////////////////////////////////////////////
  //logic analyzer assignments.

  // FFT amplitudes (top 10 bits)
  assign analyzer1_data = {k_re[18:3]};
  assign analyzer1_clock = clock_27mhz;

  // 0
  // Learn pulse (1 bit)
  // Zamit pulse (1 bit)
  // FSM state (4 bits)
  // Search done (1 bit)
  // Result (1 bit)
  // 00000
  // Write enable signal we180 (1 bit)
```

```
  // Write enable signal we22 (1 bit)
  assign analyzer2_data = {14'b0,~led[6],~led[7]};
  assign analyzer2_clock = clock_27mhz;

  // First range
  assign analyzer3_data = sq_summ[24:9];//{data180[7:0],data22[7:0]};
  assign analyzer3_clock = clock_27mhz;

  // Addresses
  assign analyzer4_data = {6'b0,k_in};//{rd_addr180, rd_addr22};
  assign analyzer4_clock = clock_27mhz;


endmodule

//module for the 750 hz signal.
module sevenhz(
 input wire clk,
 input wire ready,
 output reg [7:0] pcm_data
);
 reg [8:0] index;

 initial begin
    index <= 8'h00;
    // synthesis attribute init of index is "00";
    pcm_data <= 20'h00000;
    // synthesis attribute init of pcm_data is "00000";
 end



 always @(posedge clk) begin
    if (ready) index <= index+1;
 end

 // one cycle of a sinewave in 64 20-bit samples
 always @(index) begin
    case (index[5:0])
      6'h00: pcm_data <= 20'h00000;
      6'h01: pcm_data <= 20'h0C8BD;
      6'h02: pcm_data <= 20'h18F8B;
      6'h03: pcm_data <= 20'h25280;
```

```
      6'h04: pcm_data <= 20'h30FBC;
      6'h05: pcm_data <= 20'h3C56B;
      6'h06: pcm_data <= 20'h471CE;
      6'h07: pcm_data <= 20'h5133C;
      6'h08: pcm_data <= 20'h5A827;
      6'h09: pcm_data <= 20'h62F20;
      6'h0A: pcm_data <= 20'h6A6D9;
      6'h0B: pcm_data <= 20'h70E2C;
      6'h0C: pcm_data <= 20'h7641A;
      6'h0D: pcm_data <= 20'h7A7D0;
      6'h0E: pcm_data <= 20'h7D8A5;
      6'h0F: pcm_data <= 20'h7F623;
      6'h10: pcm_data <= 20'h7FFFF;
      6'h11: pcm_data <= 20'h7F623;
      6'h12: pcm_data <= 20'h7D8A5;
      6'h13: pcm_data <= 20'h7A7D0;
      6'h14: pcm_data <= 20'h7641A;
      6'h15: pcm_data <= 20'h70E2C;
      6'h16: pcm_data <= 20'h6A6D9;
      6'h17: pcm_data <= 20'h62F20;
      6'h18: pcm_data <= 20'h5A827;
      6'h19: pcm_data <= 20'h5133C;
      6'h1A: pcm_data <= 20'h471CE;
      6'h1B: pcm_data <= 20'h3C56B;
      6'h1C: pcm_data <= 20'h30FBC;
      6'h1D: pcm_data <= 20'h25280;
      6'h1E: pcm_data <= 20'h18F8B;
      6'h1F: pcm_data <= 20'h0C8BD;
      6'h20: pcm_data <= 20'h00000;
      6'h21: pcm_data <= 20'hF3743;
      6'h22: pcm_data <= 20'hE7075;
      6'h23: pcm_data <= 20'hDAD80;
      6'h24: pcm_data <= 20'hCF044;
      6'h25: pcm_data <= 20'hC3A95;
      6'h26: pcm_data <= 20'hB8E32;
      6'h27: pcm_data <= 20'hAECC4;
      6'h28: pcm_data <= 20'hA57D9;
      6'h29: pcm_data <= 20'h9D0E0;
      6'h2A: pcm_data <= 20'h95927;
      6'h2B: pcm_data <= 20'h8F1D4;
      6'h2C: pcm_data <= 20'h89BE6;
      6'h2D: pcm_data <= 20'h85830;
```

```
      6'h2E: pcm_data <= 20'h8275B;
      6'h2F: pcm_data <= 20'h809DD;
      6'h30: pcm_data <= 20'h80000;
      6'h31: pcm_data <= 20'h809DD;
      6'h32: pcm_data <= 20'h8275B;
      6'h33: pcm_data <= 20'h85830;
      6'h34: pcm_data <= 20'h89BE6;
      6'h35: pcm_data <= 20'h8F1D4;
      6'h36: pcm_data <= 20'h95927;
      6'h37: pcm_data <= 20'h9D0E0;
      6'h38: pcm_data <= 20'hA57D9;
      6'h39: pcm_data <= 20'hAECC4;
      6'h3A: pcm_data <= 20'hB8E32;
      6'h3B: pcm_data <= 20'hC3A95;
      6'h3C: pcm_data <= 20'hCF044;
      6'h3D: pcm_data <= 20'hDAD80;
      6'h3E: pcm_data <= 20'hE7075;
      6'h3F: pcm_data <= 20'hF3743;
    endcase // case(index[5:0])
 end // always @ (index)
endmodule

// module for producing a 48khz clock.
module k_clk(input clk ,input reset, output kclk);

 reg [25:0] count1, next_count1;

 always @(posedge clk)
   count1 <= next_count1;

   always @*  begin
   if (reset) next_count1 = 0;
   else next_count1 = (count1 == 563) ? 0 : count1 + 1; // next_count will
start incrementing if only if count1 is not equal to 523.
   end
         assign kclk = (count1 == 563); // output of the  clock divider is the
1 when count is 523.
         endmodule

// module for playing the audio file .
module fft_audio (clock_27mhz, reset, volume,
                audio_in_data, audio_out_data, ready,
```

```
                audio_reset_b, ac97_sdata_out, ac97_sdata_in,
               ac97_synch, ac97_bit_clock);

input clock_27mhz;
input reset;
input [4:0] volume;
output [15:0] audio_in_data;
input [15:0] audio_out_data;
output ready;

//ac97 interface signals
output audio_reset_b;
output ac97_sdata_out;
input ac97_sdata_in;
output ac97_synch;
input ac97_bit_clock;

wire [2:0] source;
assign source = 0;       //mic

wire [7:0] command_address;
wire [15:0] command_data;
wire command_valid;
wire [19:0] left_in_data, right_in_data;
wire [19:0] left_out_data, right_out_data;

reg audio_reset_b;
reg [9:0] reset_count;

//wait a little before enabling the AC97 codec
always @(posedge clock_27mhz) begin

   if (reset) begin
      audio_reset_b = 1'b0;
      reset_count = 0;
   end else if (reset_count == 1023)
     audio_reset_b = 1'b1;
   else
     reset_count = reset_count+1;
end

wire ac97_ready;
```

```
 ac97 ac97(ac97_ready, command_address, command_data, command_valid,
           left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
           right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
           ac97_bit_clock);

 // generate two pulses synchronous with the clock: first capture, then ready
 reg [2:0] ready_sync;
 always @ (posedge clock_27mhz) begin
   ready_sync <= {ready_sync[1:0], ac97_ready};
 end
 assign ready = ready_sync[1] & ~ready_sync[2];

 reg [15:0] out_data;
 always @ (posedge clock_27mhz)
   if (ready) out_data <= audio_out_data;
 assign audio_in_data = left_in_data[19:4];
 assign left_out_data = {out_data, 4'b0000};
 assign right_out_data = left_out_data;

 // generate repeating sequence of read/writes to AC97 registers
 ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                   command_valid, volume, source);
endmodule

// assemble/disassemble AC97 serial frames
module ac97 (ready,
           command_address, command_data, command_valid,
           left_data, left_valid,
           right_data, right_valid,
           left_in_data, right_in_data,
           ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

 output ready;
 input [7:0] command_address;
 input [15:0] command_data;
 input command_valid;
 input [19:0] left_data, right_data;
 input left_valid, right_valid;
 output [19:0] left_in_data, right_in_data;

 input ac97_sdata_in;
 input ac97_bit_clock;
```

```
output ac97_sdata_out;
output ac97_synch;

reg ready;

reg ac97_sdata_out;
reg ac97_synch;

reg [7:0] bit_count;

reg [19:0] l_cmd_addr;
reg [19:0] l_cmd_data;
reg [19:0] l_left_data, l_right_data;
reg l_cmd_v, l_left_v, l_right_v;
reg [19:0] left_in_data, right_in_data;

initial begin
   ready <= 1'b0;
   // synthesis attribute init of ready is "0";
   ac97_sdata_out <= 1'b0;
   // synthesis attribute init of ac97_sdata_out is "0";
   ac97_synch <= 1'b0;
   // synthesis attribute init of ac97_synch is "0";

   bit_count <= 8'h00;
   // synthesis attribute init of bit_count is "0000";
   l_cmd_v <= 1'b0;
   // synthesis attribute init of l_cmd_v is "0";
   l_left_v <= 1'b0;
   // synthesis attribute init of l_left_v is "0";
   l_right_v <= 1'b0;
   // synthesis attribute init of l_right_v is "0";

   left_in_data <= 20'h00000;
   // synthesis attribute init of left_in_data is "00000";
   right_in_data <= 20'h00000;
   // synthesis attribute init of right_in_data is "00000";
end

always @(posedge ac97_bit_clock) begin
   // Generate the sync signal
   if (bit_count == 255)
```

```
      ac97_synch <= 1'b1;
    if (bit_count == 15)
      ac97_synch <= 1'b0;

    // Generate the ready signal
    if (bit_count == 128)
      ready <= 1'b1;
    if (bit_count == 2)
      ready <= 1'b0;

    // Latch user data at the end of each frame. This ensures that the
    // first frame after reset will be empty.
    if (bit_count == 255)
      begin
         l_cmd_addr <= {command_address, 12'h000};
         l_cmd_data <= {command_data, 4'h0};
         l_cmd_v <= command_valid;
         l_left_data <= left_data;
         l_left_v <= left_valid;
         l_right_data <= right_data;
         l_right_v <= right_valid;
      end

    if ((bit_count >= 0) && (bit_count <= 15))
      // Slot 0: Tags
      case (bit_count[3:0])
        4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
        4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
        4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
        4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
         4'h4: ac97_sdata_out <= l_right_v; // Right data valid
        default: ac97_sdata_out <= 1'b0;
      endcase

    else if ((bit_count >= 16) && (bit_count <= 35))
      // Slot 1: Command address (8-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

    else if ((bit_count >= 36) && (bit_count <= 55))
      // Slot 2: Command data (16-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;
```

```
    else if ((bit_count >= 56) && (bit_count <= 75))
      begin
         // Slot 3: Left channel
         ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
         l_left_data <= { l_left_data[18:0], l_left_data[19] };
      end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
         ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

    bit_count <= bit_count+1;

 end // always @ (posedge ac97_bit_clock)

 always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
      right_in_data <= { right_in_data[18:0], ac97_sdata_in };
 end

endmodule

// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                   command_valid, volume, source);

 input clock;
 input ready;
 output [7:0] command_address;
 output [15:0] command_data;
 output command_valid;
 input [4:0] volume;
 input [2:0] source;

 reg [23:0] command;
 reg command_valid;
```

```
reg [3:0] state;

initial begin
   command <= 4'h0;
   // synthesis attribute init of command is "0";
   command_valid <= 1'b0;
   // synthesis attribute init of command_valid is "0";
   state <= 16'h0000;
   // synthesis attribute init of state is "0000";
end

assign command_address = command[23:16];
assign command_data = command[15:0];

wire [4:0] vol;
assign vol = 31-volume;  // convert to attenuation

always @(posedge clock) begin
   if (ready) state <= state+1;

   case (state)
     4'h0: // Read ID
       begin
          command <= 24'h80_0000;
          command_valid <= 1'b1;
       end
     4'h1: // Read ID
       command <= 24'h80_0000;
     4'h3: // headphone volume
       command <= { 8'h04, 3'b000, vol, 3'b000, vol };
     4'h5: // PCM volume
       command <= 24'h18_0808;
     4'h6: // Record source select
       command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
     4'h7: // Record gain = max
        command <= 24'h1C_0F0F;
     4'h9: // set +20db mic gain
       command <= 24'h0E_8048;
     4'hA: // Set beep volume
       command <= 24'h0A_0000;
     4'hB: // PCM out bypass mix1
       command <= 24'h20_8000;
```

```
      default:
        command <= 24'h80_0000;
    endcase // case(state)
 end // always @ (posedge clock)
endmodule // ac97commands

/////////////////////////////






///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
///////////////////////////////////////////////////////////////////////////////
// moudle for hex display.
module display_16hex (reset, clock_27mhz, data,
               disp_blank, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_out);

 input reset, clock_27mhz;    // clock and reset (active high reset)
 input [63:0] data;            // 16 hex nibbles to display

 output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b,
        disp_reset_b;

 reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;
```

```
////////////////////////////////////////////////////////////////////////////
//
// Display Clock
//
// Generate a 500kHz clock for driving the displays.
//
////////////////////////////////////////////////////////////////////////////
reg [4:0] count;
reg [7:0] reset_count;
reg clock;
wire dreset;

always @(posedge clock_27mhz)
  begin
      if (reset)
        begin
           count = 0;
           clock = 0;
        end
      else if (count == 26)
        begin
           clock = ~clock;
           count = 5'h00;
        end
      else
        count = count+1;
  end

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign dreset = (reset_count != 0);

assign disp_clock = ~clock;

////////////////////////////////////////////////////////////////////////////
//
// Display State Machine
```

```
//
////////////////////////////////////////////////////////////////////////////
reg [7:0] state;         // FSM state
reg [9:0] dot_index;     // index to current dot being clocked out
reg [31:0] control;      // control register
reg [3:0] char_index;    // index of current character
reg [39:0] dots;         // dots for a single digit
reg [3:0] nibble;        // hex nibble of current character

assign disp_blank = 1'b0; // low <= not blanked

always @(posedge clock)
  if (dreset)
    begin
        state <= 0;
        dot_index <= 0;
        control <= 32'h7F7F7F7F;
    end
  else
    casex (state)
       8'h00:
         begin
            // Reset displays
            disp_data_out <= 1'b0;
            disp_rs <= 1'b0; // dot register
            disp_ce_b <= 1'b1;
            disp_reset_b <= 1'b0;
            dot_index <= 0;
            state <= state+1;
         end

       8'h01:
         begin
            // End reset
            disp_reset_b <= 1'b1;
            state <= state+1;
         end

       8'h02:
         begin
            // Initialize dot register (set all dots to zero)
            disp_ce_b <= 1'b0;
```

```
          disp_data_out <= 1'b0; // dot_index[0];
          if (dot_index == 639)
            state <= state+1;
          else
            dot_index <= dot_index+1;
       end

     8'h03:
       begin
          // Latch dot data
          disp_ce_b <= 1'b1;
          dot_index <= 31;          // re-purpose to init ctrl reg
          disp_rs <= 1'b1; // Select the control register
          state <= state+1;
       end

     8'h04:
       begin
          // Setup the control register
          disp_ce_b <= 1'b0;
          disp_data_out <= control[31];
          control <= {control[30:0], 1'b0}; // shift left
          if (dot_index == 0)
            state <= state+1;
          else
            dot_index <= dot_index-1;
       end

     8'h05:
       begin
          // Latch the control register data / dot data
          disp_ce_b <= 1'b1;
          dot_index <= 39;          // init for single char
          char_index <= 15;         // start with MS char
          state <= state+1;
          disp_rs <= 1'b0;          // Select the dot register
       end

     8'h06:
       begin
          // Load the user's dot data into the dot reg, char by char
          disp_ce_b <= 1'b0;
```

```
            disp_data_out <= dots[dot_index]; // dot data from msb
            if (dot_index == 0)
              if (char_index == 0)
                state <= 5;                   // all done, latch data
              else
              begin
                char_index <= char_index - 1; // goto next char
                dot_index <= 39;
              end
            else
              dot_index <= dot_index-1;       // else loop thru all dots
          end

    endcase

 always @ (data or char_index)
   case (char_index)
     4'h0:             nibble <= data[3:0];
     4'h1:             nibble <= data[7:4];
     4'h2:             nibble <= data[11:8];
     4'h3:             nibble <= data[15:12];
     4'h4:             nibble <= data[19:16];
     4'h5:             nibble <= data[23:20];
     4'h6:             nibble <= data[27:24];
     4'h7:             nibble <= data[31:28];
     4'h8:             nibble <= data[35:32];
     4'h9:             nibble <= data[39:36];
     4'hA:             nibble <= data[43:40];
     4'hB:             nibble <= data[47:44];
     4'hC:             nibble <= data[51:48];
     4'hD:             nibble <= data[55:52];
     4'hE:             nibble <= data[59:56];
     4'hF:             nibble <= data[63:60];
   endcase

 always @(nibble)
   case (nibble)
     4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
     4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
     4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
     4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
     4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
```

```
     4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
     4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
     4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
     4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
     4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
     4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
     4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
     4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
     4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
     4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
     4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
   endcase

endmodule

// modilue for producing the sum of squares.
module sqrnsum( input signed [18:0]k_re, input signed [18:0]k_im,output signed
[38:0] sq_sum);

reg  signed [38:0] sum;   // red for the sum of squares of the real and the
imaginary part

always @* begin
  sum = ((k_im)*(k_im))+((k_re)*(k_re)); // sum is produced squaring the
iamaginary and the real parts and adding them up.
  end

assign sq_sum = sum;


endmodule
```

```
////////////////////////////////////////////////////////////////////////////
//
//   Yafim Landa    landa@mit.edu
//   Written for 6.111, Fall 2010
//
//   Description: This is the test jig for all of the components, with the
//   exception of the FFT, for which sample values have been generated.
//
////////////////////////////////////////////////////////////////////////////

//500MHz generator
module gen2ns(output reg pulse);
  initial begin
    pulse = 1;
    forever #1 pulse = ~pulse;
  end
endmodule
//100MHz generator
module gen10ns(output reg pulse);
  initial begin
    pulse = 1;
    forever #5 pulse = ~pulse;
  end
endmodule

module gen60 (input wire clk,
                            input wire reset,
                            output wire enable);
  reg [1:0] count;
  always@(posedge clk) begin
     if (reset) count <= 0;
     else if (count==3) count <= 0;
     else   count <= count + 1;
  end
  assign enable = (count==3);
endmodule

module testInCascade(output ready, [44:0] result);
  reg reset;
  wire clock;
  wire[8:0] abus;
  wire[9:0] dbus1,dbus2,dbus3,dbus4,dbus5;
```

```
  //FFT fft1 (.rst(reset), .clk(clock), .we(wr_ena), .adr(fft_adr),
.dout(fft_dat), .rdy(fft_rdy));
  reg fft_we,fft_rdy;
  reg [8:0] fft_adr;
  wire [8:0] fpg_adr;
  reg [9:0] fft_dat;

  gen10ns clk1(.pulse(clock));
  FrameBuffer frbf(.clock(clock), .we(fft_we), .addr(abus), .din(fft_dat),
                   .dout1(dbus1), .dout2(dbus2), .dout3(dbus3), .dout4(dbus4),
.dout5(dbus5));
  AddressMux  amux(.write(fft_we), .waddr(fft_adr), .raddr(fpg_adr),
.addr(abus));
  FngprintGen fgen(.reset(reset), .clock(clock), .enable(fft_rdy),
                   .din1(dbus1), .din2(dbus2), .din3(dbus3), .din4(dbus4),
.din5(dbus5),
                   .adr(fpg_adr), .fingerprint(result), .ready(ready));

  integer d,up;
  integer cnt;
  initial begin
    fft_rdy = 0; fft_we = 0; fft_dat = 0; fft_adr = 0;
    reset = 1; #25reset = 0;
    //simulate FFT
    d=0; up=1;
    for (cnt=0; cnt<512; cnt=cnt+1) begin
      d = up ? (d+1) : (d-1);
      if (d==1) up = 1;
      if (d==50) up = 0;

      #5
      fft_adr = cnt[8:0];
      fft_dat = d[9:0]; //cnt[9:0]; //
      fft_we = 1; #5fft_we=0;
    end

    fft_rdy = 1;
    #1000 //wait for 100 clocks to generate a fingerprint slice
    #20 $display("...done");
    $stop();
  end
endmodule
```

```
module testZAMFSM();
  reg rst, frm, lrn, srch, done;
  wire clk, we2sec, we250ms, zam, lrn_mode;
  wire [7:0] mem_adr;

  gen10ns clk1(.pulse(clk));

  //module ZAMFSM(input reset,clock,frame,learn,start,stop,
  //output reg mem1, mem2, srch, mode, output reg[7:0] addr);

  ZAMFSM  fsm1(.reset(rst), .clock(clk), .frame(frm), .learn(lrn),
.start(srch), .stop(done),
    .mem1(we2sec), .mem2(we250ms), .srch(zam), .mode(lrn_mode),
.addr(mem_adr));

  integer cnt;
  initial begin
    frm = 0; lrn = 0; srch = 0; done = 0;
    rst = 1; #25rst = 0;

    //learn
    lrn = 1;
    for (cnt=0; cnt<185; cnt=cnt+1) begin
      #23frm = 1; lrn = 0; #23frm=0;
    end

    //zam
    srch = 1;
    for (cnt=0; cnt<30; cnt=cnt+1) begin
      #23frm = 1; srch = 0;  #23frm=0;
    end
    #53 //searching...
    //signal "done" from SearchModule
    done = 1; #25done = 0;

    #60 $display("...done");
    $stop();
  end
endmodule

module testSearcherFSM();
```

```
  reg rst, srch;
  wire clk;
  reg[44:0] d1,d2;
  wire[7:0] a1;
  wire[4:0] a2;
  wire result,done;

  gen2ns  clk1(.pulse(clk));
  //module Searcher(reset,clock,enable,din1,din2, adr1,adr2,done,result);
  Searcher src1(.reset(rst), .clock(clk), .enable(srch), .din1(d1), .din2(d2),
    .adr1(a1), .adr2(a2), .done(done), .result(result));

  integer cnt;
  initial begin
    d1 = 100; d2 = 100;
    srch = 0;
    rst = 1; #25rst = 0;
    srch = 1; #7400srch=0;
    #60 $display("...done");
    $stop();
  end
endmodule

module testSearcher();
  reg rst, srch;
  reg [7:0] adr1;
  reg [4:0] adr2;
  wire clk;

  //bram inputs
  reg we1,we2;
  wire[7:0] abus1;
  wire[4:0] abus2;
  reg [44:0] dat1,dat2;

  //bram outputs
  wire[44:0] d1,d2;

  //searcher outputs
  wire[7:0] a1;
  wire[4:0] a2;
  wire result,done;
```

```
gen2ns  clk1(.pulse(clk));

AddressMux #(.WIDTH(8))
amux1(.write(we1), .waddr(adr1), .raddr(a1), .addr(abus1));
BRAM #(.SIZE(8), .WIDTH(45))
mem1 (.clock(clk), .we(we1), .addr(abus1), .din(dat1), .dout(d1));

AddressMux #(.WIDTH(5))
amux2(.write(we2), .waddr(adr2), .raddr(a2), .addr(abus2));
BRAM #(.SIZE(5), .WIDTH(45))
mem2 (.clock(clk), .we(we2), .addr(abus2), .din(dat2), .dout(d2));

Searcher src1(.reset(rst), .clock(clk), .enable(srch), .din1(d1), .din2(d2),
  .adr1(a1), .adr2(a2), .done(done), .result(result));

integer cnt;
initial begin
  adr1 = 0; adr2 = 0;
  we1 = 0; we2 = 0; srch = 0;
  rst = 1; #5 rst = 0;

  //1. prepare memories
  // 0-------18+++39--------------179
  //          0+++21
  for (cnt=0; cnt<180; cnt=cnt+1) begin
    if (cnt<18 || cnt>39) begin
      dat1 = 45'b101010101010101010101010101010101010101010101;
      #3we1 = 1; #3we1=0;
    end
    else begin
      dat1 = {cnt[8:0],cnt[8:0],cnt[8:0],cnt[8:0],cnt[8:0]};
      dat2 = {cnt[8:0],cnt[8:0],cnt[8:0],cnt[8:0],cnt[8:0]};
      adr2 = adr1 - 18;
      #3we1 = 1; we2=1;
      #3we1 = 0; we2=0;
    end
    adr1 = adr1 + 1;
  end
  #25
  //2. search
  srch = 1; #7400 srch=0;
```

```
    #12 $display("...done");
    $stop();
  end
endmodule

module testFFTOutput();
  reg rst, srch;
  wire clk,syn;
  reg[18:0] d1;
  reg[9:0] a1;
  wire[9:0] ampl;
  wire[8:0] freq;
  wire we,ready;

  gen2ns  clk1(.pulse(clk));
  gen10ns clk2(.pulse(syn));
  FFTOutput ffto(.reset(rst), .clk27M(clk), .clk48K(syn),
    .fft_amp(d1), .fft_idx(a1),
    .amp(ampl), .bin(freq), .we(we), .rdy(ready));

  integer cnt;
  initial begin
    d1 = 0; a1 = 0;
    rst = 1; #25rst = 0;
    for (cnt=0; cnt<1024; cnt=cnt+1) begin
      #11
      if (cnt==16) d1 = 512;
      else d1 = 0;
      a1 = a1+1;
    end

    #60 $display("...done");
    $stop();
  end
endmodule

module testZam();
  wire clock1,clock2,learn_pulse,zamit_pulse;
  reg reset,learn,zamit;
  reg[18:0] k_re;
  reg[9:0] k_in;
  wire slice_rdy,we,we22,search,lrn_mode;
```

```
  wire [44:0] data, data_out,data_out22;
  wire [7:0] wr_addr,rd_addr, mem_addr;
  wire [4:0] rd_addr22, mem_addr22;
  wire zero,one;

  assign zero = 0;
  assign one = 1;

  gen2ns gen1(.pulse(clock1));
  gen10ns gen2(.pulse(clock2));

  InputCascade inp1 (
    .reset(reset), .clock(clock1), .fft_clk(clock2), .fft_amp(k_re),
.fft_idx(k_in),
    .ready(slice_rdy), .slice(data)
  );

  SyncGen snc1 (.clock(clock1), .signal(learn), .pulse(learn_pulse));
  SyncGen snc2 (.clock(clock1), .signal(zamit), .pulse(zamit_pulse));

  ZAMFSM fsm1 (
    .reset(reset), .clock(clock1), .frame(slice_rdy),
    .learn(learn_pulse), .start(zamit_pulse), .stop(search_done),
    .mem1(we), .mem2(we22), .srch(search), .mode(lrn_mode), .addr(wr_addr));

  AddressMux #(.WIDTH(8))
  amux180(.write(we), .waddr(wr_addr), .raddr(rd_addr), .addr(mem_addr));
  BRAM #(.SIZE(8), .WIDTH(45))
  mem180 (.clock(clock1), .we(we), .addr(mem_addr), .din(data),
.dout(data_out));

  AddressMux #(.WIDTH(5))
  amux22(.write(we22), .waddr(wr_addr[4:0]), .raddr(rd_addr22),
.addr(mem_addr22));
  BRAM #(.SIZE(5), .WIDTH(45))
  mem22 (.clock(clock1), .we(we22), .addr(mem_addr22), .din(data),
.dout(data_out22));

  Searcher src1(.reset(reset), .clock(clock1), .enable(search),
.din1(data_out), .din2(data_out22),
    .adr1(rd_addr), .adr2(rd_addr22), .done(search_done), .result(result));
```

```
  integer cnt,cnt1;
  initial begin
    learn = 0; zamit = 0;
    reset = 1; #25reset = 0;
    learn = 1;
    for (cnt1=0; cnt1<200; cnt1=cnt1+1) begin
      k_re = 0; k_in = 0;
      for (cnt=0; cnt<1024; cnt=cnt+1) begin
        #11
        if
((cnt==cnt1)||(cnt==cnt1+100)||(cnt==cnt1+200)||(cnt==cnt1+300)||(cnt==cnt1+40
)) k_re = 512;
        else k_re = 0;
        k_in = k_in+1;
      end
    end
    learn = 0; #25zamit = 1;
    for (cnt1=60; cnt1<90; cnt1=cnt1+1) begin
      k_re = 0; k_in = 0;
      for (cnt=0; cnt<1024; cnt=cnt+1) begin
        #11
        if
((cnt==cnt1)||(cnt==cnt1+100)||(cnt==cnt1+200)||(cnt==cnt1+300)||(cnt==cnt1+40
)) k_re = 512;
        else k_re = 0;
        k_in = k_in+1;
      end
    end
    zamit = 0;

    #60 $display("...done");
    $stop();
  end

endmodule



module testSyncGen();
  wire clk, pulse;
  reg rst, signal;

  gen2ns gen1(.pulse(clk));
```

```
  SyncGen snc1 (.clock(clk), .signal(signal), .pulse(pulse));

  integer cnt;
  initial begin
    rst = 0;
    signal = 0;
    #7rst= 1; #17rst = 0;
    for (cnt=0; cnt<10; cnt=cnt+1) begin
      #11signal = 0; #7signal = 1;
    end

    #22

    signal = 1;
    #7rst= 1; #17rst = 0;
    for (cnt=0; cnt<10; cnt=cnt+1) begin
      #11signal = 0; #7signal = 1;
    end

        #60 $display("...done");
    $stop();
  end
endmodule

module testMem();
  wire clock1,clock2,learn_pulse;
  reg reset,learn;
  reg[18:0] k_re;
  reg[9:0] k_in;
  wire slice_rdy,we,lrn_mode;
  wire [44:0] data, data_out;
  wire [7:0] wr_addr, mem_addr;
  wire zero,one;
  wire [1:0] dummy;
  wire [7:0] switch;

  assign zero = 0;
  assign one = 1;

  gen2ns gen1(.pulse(clock1));
  gen10ns gen2(.pulse(clock2));
```

```
  InputCascade inp1 (
    .reset(reset), .clock(clock1), .fft_clk(clock2), .fft_amp(k_re),
.fft_idx(k_in),
    .ready(slice_rdy), .slice(data)
  );

  SyncGen snc1 (.clock(clock1), .signal(learn), .pulse(learn_pulse));

  ZAMFSM fsm1 (
    .reset(reset), .clock(clock1), .frame(slice_rdy),
    .learn(learn_pulse), .start(zero), .stop(zero),
    .mem1(we), .mem2(dummy[0]), .srch(dummy[1]), .mode(lrn_mode),
.addr(wr_addr));

  AddressMux #(.WIDTH(8))
  amux180(.write(we), .waddr(wr_addr), .raddr(switch[7:0]), .addr(mem_addr));
  BRAM #(.SIZE(8), .WIDTH(45))
  mem180 (.clock(clock1), .we(we), .addr(mem_addr), .din(data),
.dout(data_out));

  integer cnt,cnt1,sw;
  assign switch[7:0] = sw[7:0];
  initial begin
    learn = 0; sw = 0;
    reset = 1; #25reset = 0;
    learn = 1;
    //write
    for (cnt1=0; cnt1<220; cnt1=cnt1+1) begin
      k_re = 0; k_in = 0;
      if (cnt1 == 180) learn = 0;

      for (cnt=0; cnt<1024; cnt=cnt+1) begin
        #11
        if (cnt==15) k_re = 512;
        else k_re = 0;
        k_in = k_in+1;
      end
    end
    //read
    for (cnt=0; cnt<180; cnt=cnt+1) begin
        #5sw = sw + 1;
    end
```

```
    #60 $display("...done");
    $stop();
  end

endmodule
```

```
///////////////////////////////////////////////////////////////////////////////
//
//   Yafim Landa    landa@mit.edu
//   Written for 6.111, Fall 2010
//
//   Description: This is the labkit used to test that the memories are
//   being filled with the correct slices from the Input Cascade, and that
//   exactly 180 memory addresses are filled. Notice that this is an
//   alternate labkit file.
//
///////////////////////////////////////////////////////////////////////////////

///////////////////////////////////////////////////////////////////////////////
//
// Switch Debounce Module
//
///////////////////////////////////////////////////////////////////////////////
module debounce (input reset, clock, noisy, output reg clean);
   reg [18:0] count;
   reg new;

   always @(posedge clock)
     if (reset) begin new <= noisy; clean <= noisy; count <= 0; end
     else if (noisy != new) begin new <= noisy; count <= 0; end
     else if (count == 270000) clean <= new;
     else count <= count+1;
endmodule

module labkit (beep, audio_reset_b, ac97_sdata_out, ac97_sdata_in, ac97_synch,
               ac97_bit_clock,

               vga_out_red, vga_out_green, vga_out_blue, vga_out_sync_b,
               vga_out_blank_b, vga_out_pixel_clock, vga_out_hsync,
               vga_out_vsync,

               tv_out_ycrcb, tv_out_reset_b, tv_out_clock, tv_out_i2c_clock,
               tv_out_i2c_data, tv_out_pal_ntsc, tv_out_hsync_b,
               tv_out_vsync_b, tv_out_blank_b, tv_out_subcar_reset,

               tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1,
               tv_in_line_clock2, tv_in_aef, tv_in_hff, tv_in_aff,
               tv_in_i2c_clock, tv_in_i2c_data, tv_in_fifo_read,
```

```
            tv_in_fifo_clock, tv_in_iso, tv_in_reset_b, tv_in_clock,

            ram0_data, ram0_address, ram0_adv_ld, ram0_clk, ram0_cen_b,
            ram0_ce_b, ram0_oe_b, ram0_we_b, ram0_bwe_b,

            ram1_data, ram1_address, ram1_adv_ld, ram1_clk, ram1_cen_b,
            ram1_ce_b, ram1_oe_b, ram1_we_b, ram1_bwe_b,

            clock_feedback_out, clock_feedback_in,

            flash_data, flash_address, flash_ce_b, flash_oe_b, flash_we_b,
            flash_reset_b, flash_sts, flash_byte_b,

            rs232_txd, rs232_rxd, rs232_rts, rs232_cts,

            mouse_clock, mouse_data, keyboard_clock, keyboard_data,

            clock_27mhz, clock1, clock2,

            disp_blank, disp_data_out, disp_clock, disp_rs, disp_ce_b,
            disp_reset_b, disp_data_in,

            button0, button1, button2, button3, button_enter, button_right,
            button_left, button_down, button_up,

            switch,

            led,

            user1, user2, user3, user4,

            daughtercard,

            systemace_data, systemace_address, systemace_ce_b,
            systemace_we_b, systemace_oe_b, systemace_irq, systemace_mpbrdy,

            analyzer1_data, analyzer1_clock,
            analyzer2_data, analyzer2_clock,
            analyzer3_data, analyzer3_clock,
            analyzer4_data, analyzer4_clock);

 output beep, audio_reset_b, ac97_synch, ac97_sdata_out;
```

```
input  ac97_bit_clock, ac97_sdata_in;

output [7:0] vga_out_red, vga_out_green, vga_out_blue;
output vga_out_sync_b, vga_out_blank_b, vga_out_pixel_clock,
        vga_out_hsync, vga_out_vsync;

output [9:0] tv_out_ycrcb;
output tv_out_reset_b, tv_out_clock, tv_out_i2c_clock, tv_out_i2c_data,
        tv_out_pal_ntsc, tv_out_hsync_b, tv_out_vsync_b, tv_out_blank_b,
        tv_out_subcar_reset;

input  [19:0] tv_in_ycrcb;
input  tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2, tv_in_aef,
        tv_in_hff, tv_in_aff;
output tv_in_i2c_clock, tv_in_fifo_read, tv_in_fifo_clock, tv_in_iso,
        tv_in_reset_b, tv_in_clock;
inout  tv_in_i2c_data;

inout  [35:0] ram0_data;
output [18:0] ram0_address;
output ram0_adv_ld, ram0_clk, ram0_cen_b, ram0_ce_b, ram0_oe_b, ram0_we_b;
output [3:0] ram0_bwe_b;

inout  [35:0] ram1_data;
output [18:0] ram1_address;
output ram1_adv_ld, ram1_clk, ram1_cen_b, ram1_ce_b, ram1_oe_b, ram1_we_b;
output [3:0] ram1_bwe_b;

input  clock_feedback_in;
output clock_feedback_out;

inout  [15:0] flash_data;
output [23:0] flash_address;
output flash_ce_b, flash_oe_b, flash_we_b, flash_reset_b, flash_byte_b;
input  flash_sts;

output rs232_txd, rs232_rts;
input  rs232_rxd, rs232_cts;

input  mouse_clock, mouse_data, keyboard_clock, keyboard_data;

input  clock_27mhz, clock1, clock2;
```

```
 output disp_blank, disp_clock, disp_rs, disp_ce_b, disp_reset_b;
 input  disp_data_in;
 output  disp_data_out;

 input  button0, button1, button2, button3, button_enter, button_right,
         button_left, button_down, button_up;
 input  [7:0] switch;
 output [7:0] led;

 inout [31:0] user1, user2, user3, user4;

 inout [43:0] daughtercard;

 inout  [15:0] systemace_data;
 output [6:0]  systemace_address;
 output systemace_ce_b, systemace_we_b, systemace_oe_b;
 input  systemace_irq, systemace_mpbrdy;

 output [15:0] analyzer1_data, analyzer2_data, analyzer3_data,
                analyzer4_data;
 output analyzer1_clock, analyzer2_clock, analyzer3_clock, analyzer4_clock;

 ////////////////////////////////////////////////////////////////////////////
 //
 // RESET Generation
 //
 ////////////////////////////////////////////////////////////////////////////
  wire pwr_reset, btn_reset, reset;
  SRL16 reset_sr(.D(1'b0), .CLK(clock_27mhz), .Q(pwr_reset), .A0(1'b1),
.A1(1'b1), .A2(1'b1), .A3(1'b1));
  defparam reset_sr.INIT = 16'hFFFF;
  debounce db1(.reset(pwr_reset), .clock(clock_27mhz), .noisy(~button_enter),
.clean(btn_reset));
  assign reset = pwr_reset || btn_reset;

 ////////////////////////////////////////////////////////////////////////////
 //
 // I/O Assignments
 //
 ////////////////////////////////////////////////////////////////////////////
```

```
// Audio Input and Output
assign beep= 1'b0;
assign audio_reset_b = 1'b0;
assign ac97_synch = 1'b0;
assign ac97_sdata_out = 1'b0;
// ac97_sdata_in is an input

// VGA Output
assign vga_out_red = 8'h0;
assign vga_out_green = 8'h0;
assign vga_out_blue = 8'h0;
assign vga_out_sync_b = 1'b1;
assign vga_out_blank_b = 1'b1;
assign vga_out_pixel_clock = 1'b0;
assign vga_out_hsync = 1'b0;
assign vga_out_vsync = 1'b0;

// Video Output
assign tv_out_ycrcb = 10'h0;
assign tv_out_reset_b = 1'b0;
assign tv_out_clock = 1'b0;
assign tv_out_i2c_clock = 1'b0;
assign tv_out_i2c_data = 1'b0;
assign tv_out_pal_ntsc = 1'b0;
assign tv_out_hsync_b = 1'b1;
assign tv_out_vsync_b = 1'b1;
assign tv_out_blank_b = 1'b1;
assign tv_out_subcar_reset = 1'b0;

// Video Input
assign tv_in_i2c_clock = 1'b0;
assign tv_in_fifo_read = 1'b0;
assign tv_in_fifo_clock = 1'b0;
assign tv_in_iso = 1'b0;
assign tv_in_reset_b = 1'b0;
assign tv_in_clock = 1'b0;
assign tv_in_i2c_data = 1'bZ;
// tv_in_ycrcb, tv_in_data_valid, tv_in_line_clock1, tv_in_line_clock2,
// tv_in_aef, tv_in_hff, and tv_in_aff are inputs

// SRAMs
assign ram0_data = 36'hZ;
```

```
 assign ram0_address = 19'h0;
 assign ram0_adv_ld = 1'b0;
 assign ram0_clk = 1'b0;
 assign ram0_cen_b = 1'b1;
 assign ram0_ce_b = 1'b1;
 assign ram0_oe_b = 1'b1;
 assign ram0_we_b = 1'b1;
 assign ram0_bwe_b = 4'hF;
 assign ram1_data = 36'hZ;
 assign ram1_address = 19'h0;
 assign ram1_adv_ld = 1'b0;
 assign ram1_clk = 1'b0;
 assign ram1_cen_b = 1'b1;
 assign ram1_ce_b = 1'b1;
 assign ram1_oe_b = 1'b1;
 assign ram1_we_b = 1'b1;
 assign ram1_bwe_b = 4'hF;
 assign clock_feedback_out = 1'b0;
 // clock_feedback_in is an input

 // Flash ROM
 assign flash_data = 16'hZ;
 assign flash_address = 24'h0;
 assign flash_ce_b = 1'b1;
 assign flash_oe_b = 1'b1;
 assign flash_we_b = 1'b1;
 assign flash_reset_b = 1'b0;
 assign flash_byte_b = 1'b1;
 // flash_sts is an input

 // RS-232 Interface
 assign rs232_txd = 1'b1;
 assign rs232_rts = 1'b1;
 // rs232_rxd and rs232_cts are inputs

 // PS/2 Ports
 // mouse_clock, mouse_data, keyboard_clock, and keyboard_data are inputs

 // LED Displays
//   assign disp_blank = 1'b1;
//   assign disp_clock = 1'b0;
//   assign disp_rs = 1'b0;
```

```
//   assign disp_ce_b = 1'b1;
//   assign disp_reset_b = 1'b0;
//   assign disp_data_out = 1'b0;
 // disp_data_in is an input

  // Buttons, Switches, and Individual LEDs
  // assign led = 8'hFF;
  // button0, button1, button2, button3, button_enter, button_right,
  // button_left, button_down, button_up, and switches are inputs
  assign led[5] = 1'b1;
  assign led[6] = 1'b1;

 // User I/Os
 assign user1 = 32'hZ;
 assign user2 = 32'hZ;
 assign user3 = 32'hZ;
 assign user4 = 32'hZ;

 // Daughtercard Connectors
 assign daughtercard = 44'hZ;

 // SystemACE Microprocessor Port
 assign systemace_data = 16'hZ;
 assign systemace_address = 7'h0;
 assign systemace_ce_b = 1'b1;
 assign systemace_we_b = 1'b1;
 assign systemace_oe_b = 1'b1;
 // systemace_irq and systemace_mpbrdy are inputs

// Logic Analyzer
//   assign analyzer1_data = 16'h0;
//   assign analyzer1_clock = 1'b1;
//   assign analyzer2_data = 16'h0;
//   assign analyzer2_clock = 1'b1;
//   assign analyzer3_data = 16'h0;
//   assign analyzer3_clock = 1'b1;
//   assign analyzer4_data = 16'h0;
//   assign analyzer4_clock = 1'b1;

//wire clock_48khz;
//divider out_48khz(clock_27mhz, reset, clock_48khz);
//wire [19:0] pcm_data;
```

```
//tone750hz tone_750(clock_27mhz, clock_48khz, pcm_data);
wire kclk;
wire clock ;
assign clock = clock_27mhz;

wire r_f;
assign led[0] = ~(r_f);

wire d;
assign led[1] = ~d;

wire done ;
assign led[2] = ~done;

wire busy;
assign led[3] = ~busy;

wire e;
assign led[4] = ~e;

wire ready;
assign ready = 1'b1;

wire signed [9:0] n_in,k_in;
wire signed [18:0] k_re, k_im;

wire signed [7:0] tone;

 sevenhz hz(.clk(kclk),.ready(ready),.pcm_data(tone));

 k_clk  cllk(.clk(clock),.reset(reset),.kclk(kclk));
 wire [7:0] from_ac97_data, to_ac97_data;
 myfft my_fft(
              .ce(kclk),           // clock enable FFT at 48khz
              .fwd_inv_we (1'b0),  // forward transform
              .rfd(r_f),           // ready for data output, not  used
              .start(1'b1),        // always starting
              .fwd_inv(1'b1),      // forward transform
              .dv(d),              // data valid output not used
              .done(done),         // done output
              .clk(clock_27mhz),   // fft clock
              .busy(busy),         // busy output
```

```
               .edone(e),           // early done
               .xn_re(tone),        // 750 tone
               .xn_im(8'b0),
               .xn_index(n_in),
               .xk_re(k_re),
               .xk_im(k_im),
               .xk_index(k_in)
               );

  //wiring
  wire btn_lrn, lrn_pulse;
  debounce db0(.reset(pwr_reset), .clock(clock_27mhz), .noisy(~button0),
.clean(btn_lrn));
  SyncGen sync1 (.clock(clock_27mhz), .signal(btn_lrn), .pulse(lrn_pulse));

  wire dummy[1:0];
  wire zero,one;
  assign zero = 0;
  assign one = 1;

  wire slice_rdy, we;
  wire [7:0] wr_addr;
  wire [44:0] slice, data;

  InputCascade inp1 (
    .reset(reset), .clock(clock_27mhz), .fft_clk(kclk), .fft_amp(k_re),
.fft_idx(k_in),
    .ready(slice_rdy), .slice(slice));

  ZAMFSM fsm1 (
    .reset(reset), .clock(clock_27mhz), .frame(slice_rdy),
    .learn(lrn_pulse), .start(zero), .stop(zero),
    .mem1(we), .mem2(dummy[0]), .srch(dummy[1]), .addr(wr_addr));

  AddressMux #(.WIDTH(8))
  amux180(.write(we), .waddr(wr_addr), .raddr(switch[7:0]), .addr(addr));
  BRAM #(.SIZE(8), .WIDTH(45))
  mem1 (.clock(clock_27mhz), .we(we), .addr(addr), .din(slice), .dout(data));

 assign led[7] = slice_rdy;

 assign display_data = {19'b0,data};
```

```
 display_16hex hexdisplay (.reset(reset), .clock_27mhz(clock_27mhz),
.data(display_data),
               .disp_blank(disp_blank), .disp_clock(disp_clock),
.disp_rs(disp_rs), .disp_ce_b(disp_ce_b),
               .disp_reset_b(disp_reset_b), .disp_data_out(disp_data_out));


////////////////////////////////////////////////////////
// Logic Analyzer Settings
// CK0              CK0()
// amps(k_re)       A1(1-0),A0(7-0)
// bins(k_in)       C3(1-0),C2(7-0)
// we               A3(7)
// lrn_pulse        A3(6)
// slice            A3(0),A2(7-0)
// addr             D0(7-0)
////////////////////////////////////////////////////////

  // FFT amplitudes (top 10 bits)
  assign analyzer1_data = {6'b0,k_re[18:9]};
  assign analyzer1_clock = clock_27mhz;

  // Write enable signal we (1 bit)
  // Learn pulse (1 bit)
  // 00000
  // Slice from InputCascade (bottom 9 bits)
  assign analyzer2_data = {we,lrn_pulse,5'b0,slice[8:0]};
  assign analyzer2_clock = clock_27mhz;

  // Address (8 bits)
  assign analyzer3_data = {8'b0,addr};
  assign analyzer3_clock = clock_27mhz;

/*
  // Read address (8 bits)
  // Write address (8 bits)
  assign analyzer3_data = {rd_addr,wr_addr};
  assign analyzer3_clock = clock_27mhz;
*/

  // FFT bins (10 bits)
  assign analyzer4_data = {6'b0, k_in};
```

```
  assign analyzer4_clock = clock_27mhz;


//module fft1024gh (ce, fwd_inv_we, rfd, start, fwd_inv, dv, done,
//clk, busy, edone, xn_re, xk_im, xn_index, xk_re, xn_im, xk_index);
//  input ce;
//  input fwd_inv_we;
//  output rfd;
//  input start;
//  input fwd_inv;
//  output dv;
//  output done;
//  input clk;
//  output busy;
//  output edone;
//  input [7 : 0] xn_re;
//  output [18 : 0] xk_im;
//  output [9 : 0] xn_index;
//  output [18 : 0] xk_re;
//  input [7 : 0] xn_im;
//  output [9 : 0] xk_index;
//
////////////////////////////////////////////////////////////////////////////////
///////////////////
//wire [7:0] from_ac97_data, to_ac97_data;
 //wire ready;
//
//   // allow user to adjust volume
//   wire vup,vdown;
//   reg old_vup,old_vdown;
//   debounce bup(reset, clock_27mhz, ~button_up, vup);
//   debounce bdown(reset, clock_27mhz, ~button_down, vdown);
//   reg [4:0] volume;
//   always @ (posedge clock_27mhz) begin
//     if (reset) volume <= 5'd8;
//     else begin
//      if (vup & ~old_vup & volume != 5'd31) volume <= volume+1;
//      if (vdown & ~old_vdown & volume != 5'd0) volume <= volume-1;
//     end
//     old_vup <= vup;
//     old_vdown <= vdown;
//   end
```

```
//
//   // AC97 driver
//   fft_audio a(clock_27mhz, reset, volume, from_ac97_data,
to_ac97_data,ready,
//               audio_reset_b, ac97_sdata_out, ac97_sdata_in,
//               ac97_synch, ac97_bit_clock);
//
//   // loopback incoming audio to headphones
//   assign to_ac97_data = from_ac97_data;
///////////////////////////////////////////////////////////////////////////////
/////////////////////
endmodule

module sevenhz(
 input wire clk,
 input wire ready,
 output reg [7:0] pcm_data
);
 reg [8:0] index;

 initial begin
    index <= 8'h00;
    // synthesis attribute init of index is "00";
    pcm_data <= 20'h00000;
    // synthesis attribute init of pcm_data is "00000";
 end


 always @(posedge clk) begin
    if (ready) index <= index+1;
 end

 // one cycle of a sinewave in 64 20-bit samples
 always @(index) begin
    case (index[5:0])
      6'h00: pcm_data <= 20'h00000;
      6'h01: pcm_data <= 20'h0C8BD;
      6'h02: pcm_data <= 20'h18F8B;
      6'h03: pcm_data <= 20'h25280;
      6'h04: pcm_data <= 20'h30FBC;
      6'h05: pcm_data <= 20'h3C56B;
      6'h06: pcm_data <= 20'h471CE;
```

```
      6'h07: pcm_data <= 20'h5133C;
      6'h08: pcm_data <= 20'h5A827;
      6'h09: pcm_data <= 20'h62F20;
      6'h0A: pcm_data <= 20'h6A6D9;
      6'h0B: pcm_data <= 20'h70E2C;
      6'h0C: pcm_data <= 20'h7641A;
      6'h0D: pcm_data <= 20'h7A7D0;
      6'h0E: pcm_data <= 20'h7D8A5;
      6'h0F: pcm_data <= 20'h7F623;
      6'h10: pcm_data <= 20'h7FFFF;
      6'h11: pcm_data <= 20'h7F623;
      6'h12: pcm_data <= 20'h7D8A5;
      6'h13: pcm_data <= 20'h7A7D0;
      6'h14: pcm_data <= 20'h7641A;
      6'h15: pcm_data <= 20'h70E2C;
      6'h16: pcm_data <= 20'h6A6D9;
      6'h17: pcm_data <= 20'h62F20;
      6'h18: pcm_data <= 20'h5A827;
      6'h19: pcm_data <= 20'h5133C;
      6'h1A: pcm_data <= 20'h471CE;
      6'h1B: pcm_data <= 20'h3C56B;
      6'h1C: pcm_data <= 20'h30FBC;
      6'h1D: pcm_data <= 20'h25280;
      6'h1E: pcm_data <= 20'h18F8B;
      6'h1F: pcm_data <= 20'h0C8BD;
      6'h20: pcm_data <= 20'h00000;
      6'h21: pcm_data <= 20'hF3743;
      6'h22: pcm_data <= 20'hE7075;
      6'h23: pcm_data <= 20'hDAD80;
      6'h24: pcm_data <= 20'hCF044;
      6'h25: pcm_data <= 20'hC3A95;
      6'h26: pcm_data <= 20'hB8E32;
      6'h27: pcm_data <= 20'hAECC4;
      6'h28: pcm_data <= 20'hA57D9;
      6'h29: pcm_data <= 20'h9D0E0;
      6'h2A: pcm_data <= 20'h95927;
      6'h2B: pcm_data <= 20'h8F1D4;
      6'h2C: pcm_data <= 20'h89BE6;
      6'h2D: pcm_data <= 20'h85830;
      6'h2E: pcm_data <= 20'h8275B;
      6'h2F: pcm_data <= 20'h809DD;
      6'h30: pcm_data <= 20'h80000;
```

```
      6'h31: pcm_data <= 20'h809DD;
      6'h32: pcm_data <= 20'h8275B;
      6'h33: pcm_data <= 20'h85830;
      6'h34: pcm_data <= 20'h89BE6;
      6'h35: pcm_data <= 20'h8F1D4;
      6'h36: pcm_data <= 20'h95927;
      6'h37: pcm_data <= 20'h9D0E0;
      6'h38: pcm_data <= 20'hA57D9;
      6'h39: pcm_data <= 20'hAECC4;
      6'h3A: pcm_data <= 20'hB8E32;
      6'h3B: pcm_data <= 20'hC3A95;
      6'h3C: pcm_data <= 20'hCF044;
      6'h3D: pcm_data <= 20'hDAD80;
      6'h3E: pcm_data <= 20'hE7075;
      6'h3F: pcm_data <= 20'hF3743;
    endcase // case(index[5:0])
 end // always @ (index)
endmodule



module k_clk(input clk ,input reset, output kclk);

 reg [25:0] count1, next_count1;

 always @(posedge clk)
   count1 <= next_count1;

   always @*  begin
   if (reset) next_count1 = 0;
   else next_count1 = (count1 == 563) ? 0 : count1 + 1; // next_count will
start incrementing if onlt if count1 is not equal to 27 mill.
   end
         assign kclk = (count1 == 563); // output of the divider is the 1 when
count is 27 mill.
         endmodule

         module fft_audio (clock_27mhz, reset, volume,
                audio_in_data, audio_out_data, ready,
                 audio_reset_b, ac97_sdata_out, ac97_sdata_in,
                ac97_synch, ac97_bit_clock);

 input clock_27mhz;
```

```
input reset;
input [4:0] volume;
output [15:0] audio_in_data;
input [15:0] audio_out_data;
output ready;

//ac97 interface signals
output audio_reset_b;
output ac97_sdata_out;
input ac97_sdata_in;
output ac97_synch;
input ac97_bit_clock;

wire [2:0] source;
assign source = 0;       //mic

wire [7:0] command_address;
wire [15:0] command_data;
wire command_valid;
wire [19:0] left_in_data, right_in_data;
wire [19:0] left_out_data, right_out_data;

reg audio_reset_b;
reg [9:0] reset_count;

//wait a little before enabling the AC97 codec
always @(posedge clock_27mhz) begin

   if (reset) begin
      audio_reset_b = 1'b0;
      reset_count = 0;
   end else if (reset_count == 1023)
     audio_reset_b = 1'b1;
   else
     reset_count = reset_count+1;
end

wire ac97_ready;
ac97 ac97(ac97_ready, command_address, command_data, command_valid,
          left_out_data, 1'b1, right_out_data, 1'b1, left_in_data,
          right_in_data, ac97_sdata_out, ac97_sdata_in, ac97_synch,
          ac97_bit_clock);
```

```
 // generate two pulses synchronous with the clock: first capture, then ready
 reg [2:0] ready_sync;
 always @ (posedge clock_27mhz) begin
   ready_sync <= {ready_sync[1:0], ac97_ready};
 end
 assign ready = ready_sync[1] & ~ready_sync[2];

 reg [15:0] out_data;
 always @ (posedge clock_27mhz)
   if (ready) out_data <= audio_out_data;
 assign audio_in_data = left_in_data[19:4];
 assign left_out_data = {out_data, 4'b0000};
 assign right_out_data = left_out_data;

 // generate repeating sequence of read/writes to AC97 registers
 ac97commands cmds(clock_27mhz, ready, command_address, command_data,
                  command_valid, volume, source);
endmodule

// assemble/disassemble AC97 serial frames
module ac97 (ready,
           command_address, command_data, command_valid,
           left_data, left_valid,
           right_data, right_valid,
           left_in_data, right_in_data,
           ac97_sdata_out, ac97_sdata_in, ac97_synch, ac97_bit_clock);

 output ready;
 input [7:0] command_address;
 input [15:0] command_data;
 input command_valid;
 input [19:0] left_data, right_data;
 input left_valid, right_valid;
 output [19:0] left_in_data, right_in_data;

 input ac97_sdata_in;
 input ac97_bit_clock;
 output ac97_sdata_out;
 output ac97_synch;

 reg ready;
```

```
reg ac97_sdata_out;
reg ac97_synch;

reg [7:0] bit_count;

reg [19:0] l_cmd_addr;
reg [19:0] l_cmd_data;
reg [19:0] l_left_data, l_right_data;
reg l_cmd_v, l_left_v, l_right_v;
reg [19:0] left_in_data, right_in_data;

initial begin
   ready <= 1'b0;
   // synthesis attribute init of ready is "0";
   ac97_sdata_out <= 1'b0;
   // synthesis attribute init of ac97_sdata_out is "0";
   ac97_synch <= 1'b0;
   // synthesis attribute init of ac97_synch is "0";

   bit_count <= 8'h00;
   // synthesis attribute init of bit_count is "0000";
   l_cmd_v <= 1'b0;
   // synthesis attribute init of l_cmd_v is "0";
   l_left_v <= 1'b0;
   // synthesis attribute init of l_left_v is "0";
   l_right_v <= 1'b0;
   // synthesis attribute init of l_right_v is "0";

   left_in_data <= 20'h00000;
   // synthesis attribute init of left_in_data is "00000";
   right_in_data <= 20'h00000;
   // synthesis attribute init of right_in_data is "00000";
end

always @(posedge ac97_bit_clock) begin
   // Generate the sync signal
   if (bit_count == 255)
     ac97_synch <= 1'b1;
   if (bit_count == 15)
     ac97_synch <= 1'b0;
```

```
    // Generate the ready signal
    if (bit_count == 128)
      ready <= 1'b1;
    if (bit_count == 2)
      ready <= 1'b0;

    // Latch user data at the end of each frame. This ensures that the
    // first frame after reset will be empty.
    if (bit_count == 255)
      begin
         l_cmd_addr <= {command_address, 12'h000};
         l_cmd_data <= {command_data, 4'h0};
         l_cmd_v <= command_valid;
         l_left_data <= left_data;
         l_left_v <= left_valid;
         l_right_data <= right_data;
         l_right_v <= right_valid;
      end

    if ((bit_count >= 0) && (bit_count <= 15))
      // Slot 0: Tags
      case (bit_count[3:0])
        4'h0: ac97_sdata_out <= 1'b1;      // Frame valid
        4'h1: ac97_sdata_out <= l_cmd_v;   // Command address valid
        4'h2: ac97_sdata_out <= l_cmd_v;   // Command data valid
        4'h3: ac97_sdata_out <= l_left_v;  // Left data valid
         4'h4: ac97_sdata_out <= l_right_v; // Right data valid
        default: ac97_sdata_out <= 1'b0;
      endcase

    else if ((bit_count >= 16) && (bit_count <= 35))
      // Slot 1: Command address (8-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_addr[35-bit_count] : 1'b0;

    else if ((bit_count >= 36) && (bit_count <= 55))
      // Slot 2: Command data (16-bits, left justified)
      ac97_sdata_out <= l_cmd_v ? l_cmd_data[55-bit_count] : 1'b0;

    else if ((bit_count >= 56) && (bit_count <= 75))
      begin
         // Slot 3: Left channel
         ac97_sdata_out <= l_left_v ? l_left_data[19] : 1'b0;
```

```
         l_left_data <= { l_left_data[18:0], l_left_data[19] };
      end
    else if ((bit_count >= 76) && (bit_count <= 95))
      // Slot 4: Right channel
         ac97_sdata_out <= l_right_v ? l_right_data[95-bit_count] : 1'b0;
    else
      ac97_sdata_out <= 1'b0;

    bit_count <= bit_count+1;

 end // always @ (posedge ac97_bit_clock)

 always @(negedge ac97_bit_clock) begin
    if ((bit_count >= 57) && (bit_count <= 76))
      // Slot 3: Left channel
      left_in_data <= { left_in_data[18:0], ac97_sdata_in };
    else if ((bit_count >= 77) && (bit_count <= 96))
      // Slot 4: Right channel
      right_in_data <= { right_in_data[18:0], ac97_sdata_in };
 end

endmodule

// issue initialization commands to AC97
module ac97commands (clock, ready, command_address, command_data,
                    command_valid, volume, source);

 input clock;
 input ready;
 output [7:0] command_address;
 output [15:0] command_data;
 output command_valid;
 input [4:0] volume;
 input [2:0] source;

 reg [23:0] command;
 reg command_valid;

 reg [3:0] state;

 initial begin
    command <= 4'h0;
```

```
   // synthesis attribute init of command is "0";
   command_valid <= 1'b0;
   // synthesis attribute init of command_valid is "0";
   state <= 16'h0000;
   // synthesis attribute init of state is "0000";
end

assign command_address = command[23:16];
assign command_data = command[15:0];

wire [4:0] vol;
assign vol = 31-volume;  // convert to attenuation

always @(posedge clock) begin
   if (ready) state <= state+1;

   case (state)
     4'h0: // Read ID
       begin
          command <= 24'h80_0000;
          command_valid <= 1'b1;
       end
     4'h1: // Read ID
       command <= 24'h80_0000;
     4'h3: // headphone volume
       command <= { 8'h04, 3'b000, vol, 3'b000, vol };
     4'h5: // PCM volume
       command <= 24'h18_0808;
     4'h6: // Record source select
       command <= { 8'h1A, 5'b00000, source, 5'b00000, source};
     4'h7: // Record gain = max
        command <= 24'h1C_0F0F;
     4'h9: // set +20db mic gain
       command <= 24'h0E_8048;
     4'hA: // Set beep volume
       command <= 24'h0A_0000;
     4'hB: // PCM out bypass mix1
       command <= 24'h20_8000;
     default:
       command <= 24'h80_0000;
   endcase // case(state)
end // always @ (posedge clock)
```

```
endmodule // ac97commands

///////////////////////////////////////////////////////////////////////////////
//
// 6.111 FPGA Labkit -- Hex display driver
//
// File:   display_16hex.v
// Date:   24-Sep-05
//
// Created: April 27, 2004
// Author: Nathan Ickes
//
// 24-Sep-05 Ike: updated to use new reset-once state machine, remove clear
// 28-Nov-06 CJT: fixed race condition between CE and RS (thanks Javier!)
//
// This verilog module drives the labkit hex dot matrix displays, and puts
// up 16 hexadecimal digits (8 bytes).  These are passed to the module
// through a 64 bit wire ("data"), asynchronously.
//
///////////////////////////////////////////////////////////////////////////////

module display_16hex (reset, clock_27mhz, data,
               disp_blank, disp_clock, disp_rs, disp_ce_b,
               disp_reset_b, disp_data_out);

 input reset, clock_27mhz;    // clock and reset (active high reset)
 input [63:0] data;            // 16 hex nibbles to display

 output disp_blank, disp_clock, disp_data_out, disp_rs, disp_ce_b,
        disp_reset_b;

 reg disp_data_out, disp_rs, disp_ce_b, disp_reset_b;

 ////////////////////////////////////////////////////////////////////////////
 //
 // Display Clock
```

```
//
// Generate a 500kHz clock for driving the displays.
//
////////////////////////////////////////////////////////////////////////////
reg [4:0] count;
reg [7:0] reset_count;
reg clock;
wire dreset;

always @(posedge clock_27mhz)
  begin
      if (reset)
        begin
           count = 0;
           clock = 0;
        end
      else if (count == 26)
        begin
           clock = ~clock;
           count = 5'h00;
        end
      else
        count = count+1;
  end

always @(posedge clock_27mhz)
  if (reset)
    reset_count <= 100;
  else
    reset_count <= (reset_count==0) ? 0 : reset_count-1;

assign dreset = (reset_count != 0);

assign disp_clock = ~clock;

////////////////////////////////////////////////////////////////////////////
//
// Display State Machine
//
////////////////////////////////////////////////////////////////////////////
reg [7:0] state;         // FSM state
reg [9:0] dot_index;     // index to current dot being clocked out
```

```
reg [31:0] control;      // control register
reg [3:0] char_index;    // index of current character
reg [39:0] dots;         // dots for a single digit
reg [3:0] nibble;        // hex nibble of current character

assign disp_blank = 1'b0; // low <= not blanked

always @(posedge clock)
  if (dreset)
    begin
        state <= 0;
        dot_index <= 0;
        control <= 32'h7F7F7F7F;
    end
  else
    casex (state)
       8'h00:
         begin
            // Reset displays
            disp_data_out <= 1'b0;
            disp_rs <= 1'b0; // dot register
            disp_ce_b <= 1'b1;
            disp_reset_b <= 1'b0;
            dot_index <= 0;
            state <= state+1;
         end

       8'h01:
         begin
            // End reset
            disp_reset_b <= 1'b1;
            state <= state+1;
         end

       8'h02:
         begin
            // Initialize dot register (set all dots to zero)
            disp_ce_b <= 1'b0;
            disp_data_out <= 1'b0; // dot_index[0];
            if (dot_index == 639)
              state <= state+1;
            else
```

```
          dot_index <= dot_index+1;
        end

      8'h03:
        begin
           // Latch dot data
           disp_ce_b <= 1'b1;
           dot_index <= 31;          // re-purpose to init ctrl reg
           disp_rs <= 1'b1; // Select the control register
           state <= state+1;
        end

      8'h04:
        begin
           // Setup the control register
           disp_ce_b <= 1'b0;
           disp_data_out <= control[31];
           control <= {control[30:0], 1'b0}; // shift left
           if (dot_index == 0)
             state <= state+1;
           else
             dot_index <= dot_index-1;
        end

      8'h05:
        begin
           // Latch the control register data / dot data
           disp_ce_b <= 1'b1;
           dot_index <= 39;          // init for single char
           char_index <= 15;         // start with MS char
           state <= state+1;
           disp_rs <= 1'b0;          // Select the dot register
        end

      8'h06:
        begin
           // Load the user's dot data into the dot reg, char by char
           disp_ce_b <= 1'b0;
           disp_data_out <= dots[dot_index]; // dot data from msb
           if (dot_index == 0)
             if (char_index == 0)
               state <= 5;                   // all done, latch data
```

```
              else
              begin
                char_index <= char_index - 1; // goto next char
                dot_index <= 39;
              end
            else
              dot_index <= dot_index-1;       // else loop thru all dots
          end

    endcase

 always @ (data or char_index)
   case (char_index)
     4'h0:             nibble <= data[3:0];
     4'h1:             nibble <= data[7:4];
     4'h2:             nibble <= data[11:8];
     4'h3:             nibble <= data[15:12];
     4'h4:             nibble <= data[19:16];
     4'h5:             nibble <= data[23:20];
     4'h6:             nibble <= data[27:24];
     4'h7:             nibble <= data[31:28];
     4'h8:             nibble <= data[35:32];
     4'h9:             nibble <= data[39:36];
     4'hA:             nibble <= data[43:40];
     4'hB:             nibble <= data[47:44];
     4'hC:             nibble <= data[51:48];
     4'hD:             nibble <= data[55:52];
     4'hE:             nibble <= data[59:56];
     4'hF:             nibble <= data[63:60];
   endcase

 always @(nibble)
   case (nibble)
     4'h0: dots <= 40'b00111110_01010001_01001001_01000101_00111110;
     4'h1: dots <= 40'b00000000_01000010_01111111_01000000_00000000;
     4'h2: dots <= 40'b01100010_01010001_01001001_01001001_01000110;
     4'h3: dots <= 40'b00100010_01000001_01001001_01001001_00110110;
     4'h4: dots <= 40'b00011000_00010100_00010010_01111111_00010000;
     4'h5: dots <= 40'b00100111_01000101_01000101_01000101_00111001;
     4'h6: dots <= 40'b00111100_01001010_01001001_01001001_00110000;
     4'h7: dots <= 40'b00000001_01110001_00001001_00000101_00000011;
     4'h8: dots <= 40'b00110110_01001001_01001001_01001001_00110110;
```

```
    4'h9: dots <= 40'b00000110_01001001_01001001_00101001_00011110;
    4'hA: dots <= 40'b01111110_00001001_00001001_00001001_01111110;
    4'hB: dots <= 40'b01111111_01001001_01001001_01001001_00110110;
    4'hC: dots <= 40'b00111110_01000001_01000001_01000001_00100010;
    4'hD: dots <= 40'b01111111_01000001_01000001_01000001_00111110;
    4'hE: dots <= 40'b01111111_01001001_01001001_01001001_01000001;
    4'hF: dots <= 40'b01111111_00001001_00001001_00001001_00000001;
  endcase

endmodule
```