Dance Dance Revolution

6.111 Final Project

Jacob Kitzman
Chris Wurts
Yong-yi Zhu

May 13, 2004

ABSTRACT

The Dance Dance Revolution videogame is implemented in an FPGA on the 6.111 lab kit. Game control is mediated by software running on a fully-functional implementation of the Beta, a RISC CPU similar to the one developed in MIT 6.004 Computation Structures. The system provides two forms of output: a VGA video display from a software-controlled frame buffer in memory, and audio output of prerecorded sound. A bus arbiter mediates control of a shared bus over which these components communicate. Additionally, a software tool-chain was developed allowing C code to be compiled to Beta machine code and placed in system memory.

1. Introduction

Dance Dance Revolution (DDR) is a popular video game originally released for the Sony PlayStation console game system. In DDR, the player dances on an 8-button input pad while music is played back. The objective is to dance the correct steps in sync with the on-screen cues and recorded music. Arrows are shown in the video display indicating which button on the pad the user should press at a given time. This project aims to implement the DDR video game in a Xilinx FPGA on the new 6.111 lab kit. Game logic and control of audio and video modules are mediated by software running on an implementation of the 6.004 Beta CPU. Video and audio output and game pad input are provided by separate modules in the FPGA.

2. Design Overview

2.1. Design Goals

We organized our system-level design to provide flexibility and modularity. One of the main ways we attained these goals was our implementation and use of a CPU to control game logic and display construction. Modularity is a necessity when organizing any such large system, and our design's modularity certainly paid off numerous times during its development.

2.2. System-Level Organization

We split our design was into several functional components. This system-level modularity allowed us to individually develop our respective parts in parallel and integrate them to yield the finished product. It also allowed allowed for better collaboration This would allow us to individually create each part in parallel and integrate the parts together to create a finished product. The parts used the Xilinx FPGA on the new lab kit and integrated various chips included in the kit. The system is diagrammed in Figure 1.

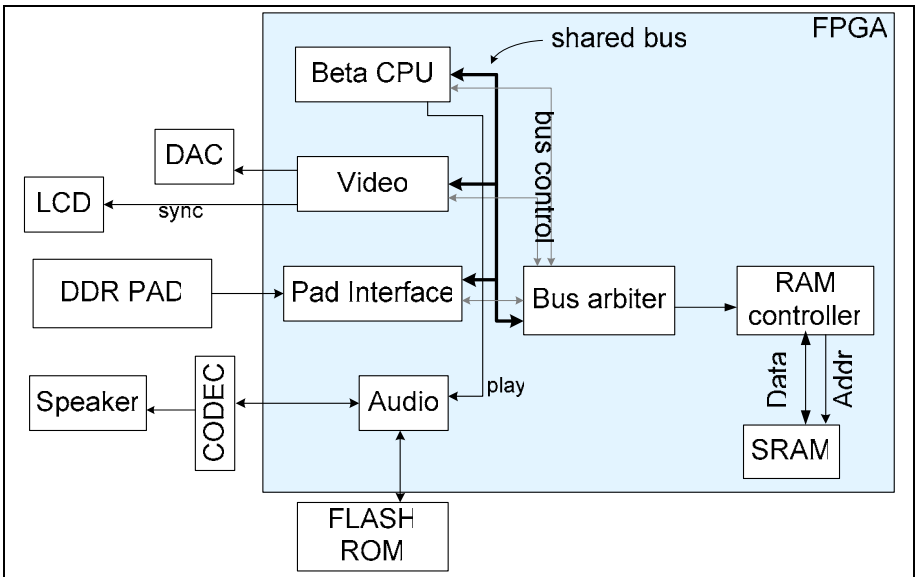

Figure 1: Overall block diagram. Shared bus is shown in bold.

DDR Pad Interface

The DDR Pad is the single form of user input to the system. The pad has various buttons that trigger input signals. The pad interface periodically samples these signals and stores them in shared memory for subsequent processing by the game control logic.

Beta Processor

The Beta Processor is the central brain of the video game. It executes machine code stored in main memory in order to perform a variety of operations including game logic, video display, interrupt handling, and bus control. The processor interfaces with other components by reading and writing values to memory over a shared bus.

Shared Bus / Bus Arbiter

The bus arbiter mediates access to the memory's shared bus interface. It is controlled by the Beta, so that bus access can be granted to specific devices by software instructions. The arbiter issues a signal to each device indicating at any given time whether that device is allowed to use the bus. The bus protocol is cooperative in the sense that once given bus access, a device has exclusive permission to the bus until it yields control. Only once the arbiter has received this signal does it revoke that device's bus access permission and give back bus access to the Beta CPU.

Video

The video in the system uses the VGA display. It reads from a frame buffer in shared memory and sends that data out to the AD7125, which then sends data out to a monitor. The video is used to display game screens.

Audio

The audio in the system is first recorded from an external source, which can later be played back. We used the National LM4550 codec to first take an analog signal from a CD player and converted that into digital data. The sound is then recorded on the Flash ROM at a rate of 4 kHz. After storage, the audio is played back during game play at the same data rate. The data is sent to the LM4550, which sends out an analog signal that the user can hear.

RAM

For the system's shared memory we used BlockRAM onboard the Xilinx FPGA. It is faster and easier to write to than the Flash ROM, but it can store much less data. Different information is stored in the RAM, including game and kernel code, the frame buffer, video sprites, and sampled DDR Pad input. We implemented a memory controller to issue control signals to the memory hardware to provide other modules with a consistent, simple interface to memory.

ROM

The Flash ROM is a large, non-volatile memory storage device. It has high capacity but is much slower for writes than the RAM. The ROM was used in this project to store audio data because even a small loop of music recorded at low sampling rate exceeded the RAM capacity.

3. Modular Detail & Implementation

3.1. Beta

3.1.1. Overview

We implemented a general-purpose RISC processor to control game logic and video display. Our processor is fully opcode-compatible with the Beta instruction set architecture (ISA)[1] and is generally an extension of the design from MIT 6.004 Computation Structures. Our overall design goal was to make the Beta as generalized as possible, so we avoided including any specialized hooks to game logic or other modules. As an example of such a mechanism, we considered a special opcode VRNDR which would issue a signal to the video system instructing it to render the frame buffer. We did not adopt such a mechanism, instead controlling video refreshes in the timer interrupt handler. In this way we kept DDR-

[1]Available at *http://6004.lcs.mit.edu/6.004/currentsemester/handouts/beta.pdf*

specific features in software which is more flexible than hardware. This would allow our Beta implementation to be used with a wide variety of peripheral hardware and used in a range of different applications.

Our Beta design is organized into several modules connected as shown in Figure 2. Note the use of registers (INSTR_REG, MEMREADREG) to latch in values read from memory. This amounts to a certain level of internal pipelining, although it should be noted that the Beta is not pipelined in the sense of being able to have multiple instructions concurrently in-flight.

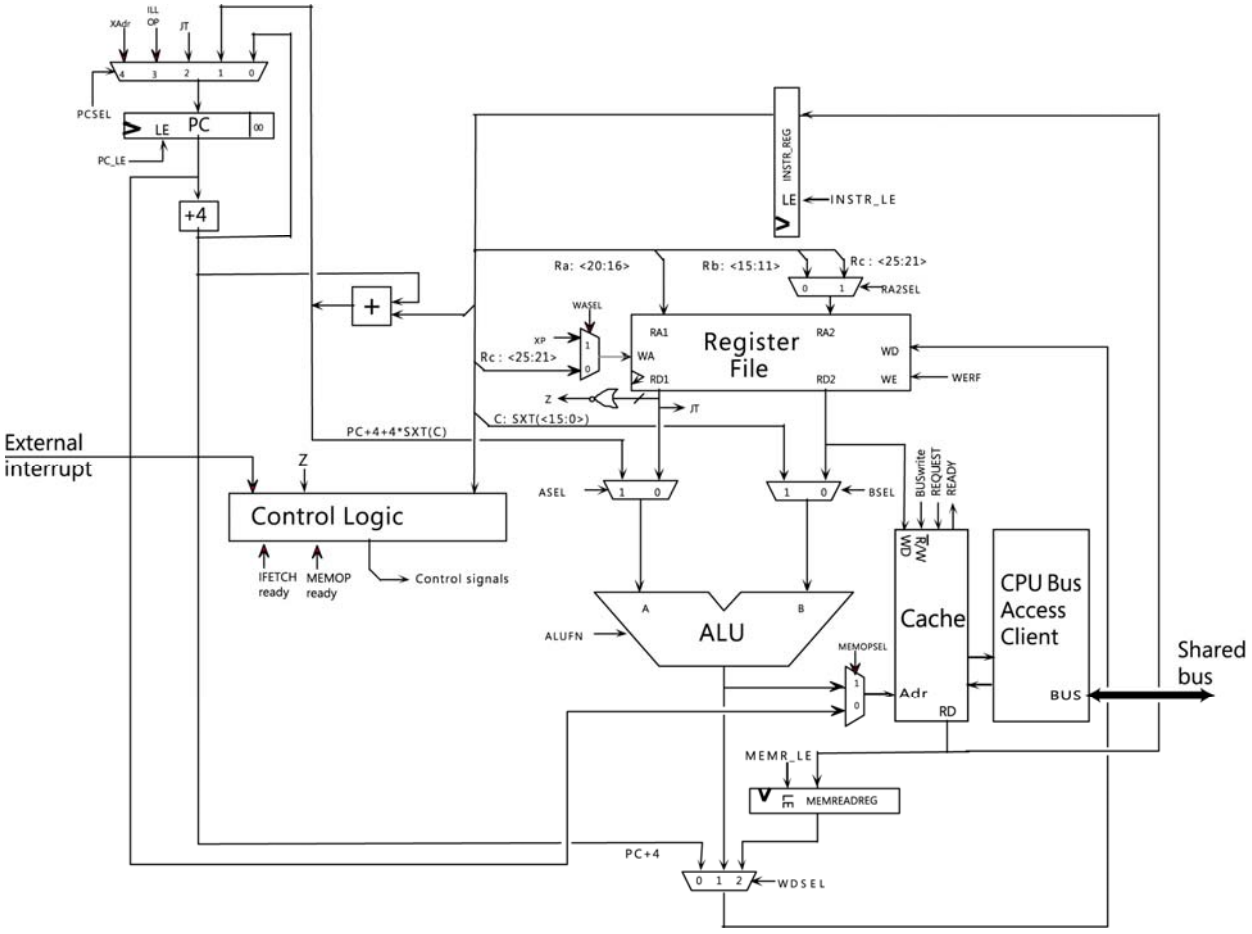

Figure 2. Modular design of the Beta.

3.1.2. Control Logic

The Control Logic module generates control signals, loading each instruction and bringing it through a series of execution stages by successively enabling the processor's various other modules (e.g., ALU, register file) at the appropriate times. The FSM implements four major stages of instruction, as shown in Figure 3. In the first of these stages, IFETCH, each instruction is fetched, or loaded from memory, by issuing an "IFETCH request" signal to the Beta's bus access client. The control logic then waits until receiving an "IFETCH ready" signal back from the bus access client along with the instruction contents. This wait can be arbitrarily long, as another device may be using the bus, causing the Beta's bus client to block until that device yields the bus. The control logic issues a signal to latch the instruction into the

instruction register, and then decodes the instruction. During this decode, the control logic issues signals based on the opcode, the 6 most significant bits of the instruction, which identify the operation to be performed. One such signal directs the ALU to perform a certain function (e.g., addition, logical AND). Also, signals are issued to numerous multiplexers setting up the datapath for this operation so that the appropriate values are routed between to the necessary submodules such as the ALU, register file, and memory subsystem.

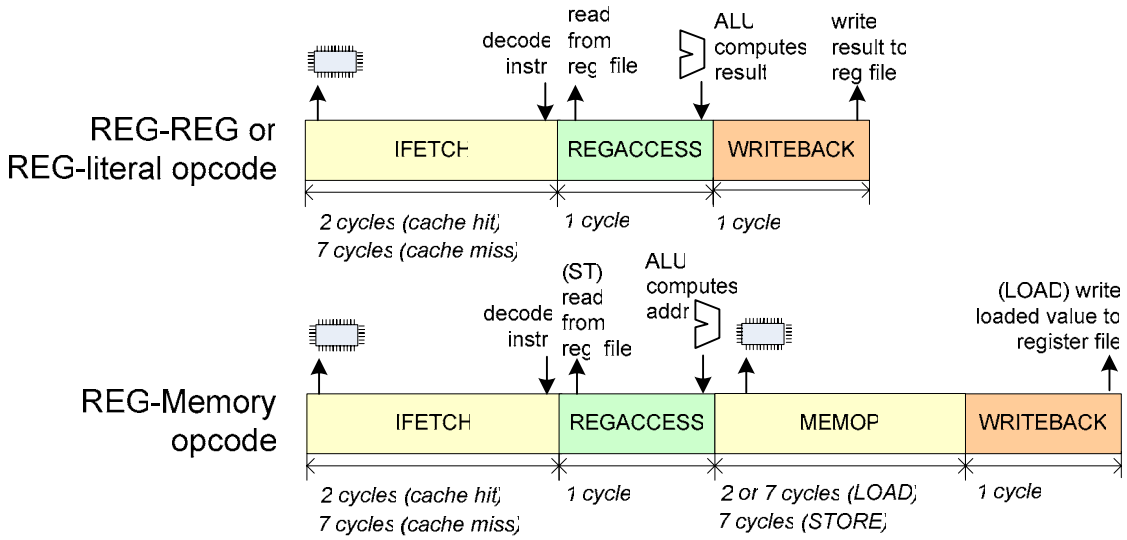

Figure 3. Beta instruction execution control stages. Execution times assume the CPU has bus access.

In the next stage, REGACCESS, values are read from the register and are used by the ALU to compute a result value (e.g., a logical AND, or a branch target address). For memory operations (LOAD and ST), a MEMOP stage follows by a mech. The final stage of execution is WRITEBACK, where values computed by ALU or loaded from memory are written into the register file. At the end of WRITEBACK, the address of the following instruction is latched into the PC (program counter) register and the cycle begins again with the IFETCH stage.

3.1.3. Memory Hierarchy

The Beta uses little-endian, byte-addressed memory model, meaning that a given 32-bit address X references the 32-bit word where X is the 0-based index of the word's least significant byte. Our Beta supports 32-bit addresses but since our implementation needs only 18 bits are needed to its 225KB of RAM, the upper 14 bits are discarded by the memory subsystem. The memory capacity is transparent to the CPU and be changed at the memory controller without modifying the CPU.

Our Beta CPU features a 512-entry, 53-bit wide, direct-mapped unified cache. All memory operation requests go from the control logic go through the cache. On reads, the cache checks the tag value (21 bits wide) to determine whether the read hits or misses. A read hit completes 2 cycles following the read request. On a read miss, the cache issues a read request to the bus access client, which then takes 4 cycles to return the data (assuming the CPU has bus access). We implemented a write-through cache, meaning that writes are blocking – on a write, the cached value is marked as invalid and a write transaction is started on the bus. The cache indicates MEMOP completion only after the value has been written over the bus to the memory controller.

We observed dramatic performance increases when operating the Beta with the cache enabled. This was particularly true because the cache is quite large relative to the ZBT SRAM size under consideration. Also, the programs we tested the Beta exhibited good spatial locality. However, when we transitioned to

using FPGA BlockRAM as our main system memory, there was no longer any speed advantage to using a cache synthesized from BlockRAM, so we disabled it.

3.1.4. CPU Bus Access Client

The CPU bus access client acts as the CPU's interface to the shared memory bus, servicing memory read and write requests from the CPU control logic by performing transactions across the shared bus to the memory controller. It is implemented by a straightforward FSM that waits until the CPU has bus access, performs the requested operation, and signals upon completion.

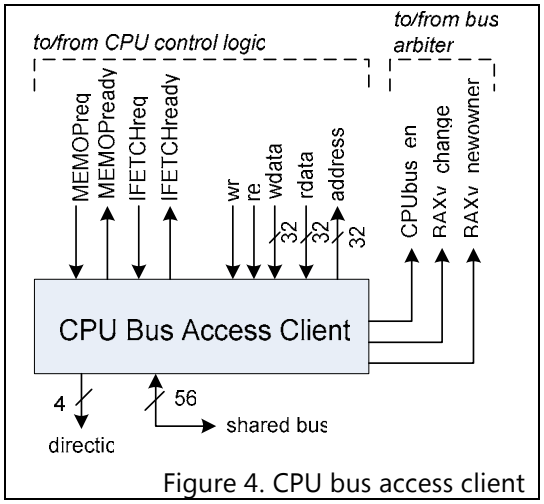

Figure 4. CPU bus access client

3.1.5. Bus Control Mechanism

The bus access client interfaces with the bus arbiter to provide software control of bus access. Bus access is controlled in software not by a dedicated opcode but rather by issuing a store (ST) instruction to a special address, called the bus access vector (abbreviated BAXv). The data to be written indicates the identifier of the device to be granted bus access (thus storing 3 to BAXv would grant bus access to device #3). Write requests to BAXv are intercepted by the bus access client and the number of the access recipient is forwarded to the bus arbiter along with a pulse of the BAXv_newowner signal, indicating that bus ownership should be changed.

3.1.6. Interrupt Functionality

The Beta control logic additionally has an 8-bit interrupt request (IRQ) vector input, each bit of which may be pulsed by a different device to indicate that an interrupt is being requested by that specific device. Interrupt requests are a mechanism widely used by modern computers to notify the CPU of asynchronous input (such as a keystroke or even just a clock tick). We used a clock divider to issue an interrupt request every 1/100 sec.

Upon receiving an interrupt request, the control logic latches in the identity of the requestor and notes that an interrupt request is pending. No other requests may be handled until the pending one is serviced. The control logic does so by completing the instruction currently inflight[2] and then writing PC+4 (the address of the instruction following the one just executed) to the exception pointer (XP) register. The CPU is then placed in supervisor mode and execution is redirected an interrupt handler specific to the device requesting an interrupt (at address $0x80000000+4*IRQ_{device\ ID}$). In supervisor mode, the PC MSB is 1, and the CPU executes kernel code and does not service interrupt requests (the kernel is not reentrant). When the interrupt has been serviced by its handler, supervisor mode is exited JMP instruction to XP, allowing execution to continue where it was interrupted. The CPU masks out the MSB of jump and branch targets, making it impossible for user code to invoke kernel execution.

[2]Our interrupt semantics differ from those of the 6.004 Beta in that in their design, the instruction is halted for interrupt processing. We chose for the sake of simplicity to handle interrupt requests only when memory and register writes had completed. Additionally, interrupt requests are noted during but not serviced following branch and jump instructions (i.e., those that change the program counter to an arbitrary target) to simplify testing.

The extent[3] of our Beta CPU's adherence to specification is exhibited by the fact that our Beta ran the assembled kernel code used by 6.004. We created a timer interrupt handler to execute code every time the clock divider issued an interrupt request pulse (every 1/100 sec). This interrupt handler increments a counter in memory. When the counter reaches a certain value, the handler issues a store instruction to BAXv in order to yield the bus to the pad control board to allow it to store sampled user input in memory. The same mechanism would have been used to instruct the video module to periodically (~40Hz) read the frame buffer across the shared bus. This counter is also used to indicate the current time to user software, for example, so the game code "knows" to display a visual cue for a dance move at a certain time.

3.1.7. Differences from 6.004 Beta

The 6.004 description of the Beta contains a number of major abstractions which required us to extend its design. First, the 6.004 design assumes separate instruction/data memories. We used a unified main memory in order to allow for flexibility in the size and arrangement of code and data. The 6.004 design also assumes instantaneous access to internal memory (the register file) as well as to external main memory. This is not a reasonable assumption, particularly for main memory which can have latencies of tens or hundreds of cycles in modern systems. The 6.004 design also assumes exclusive CPU access to system memory, which our shared bus design does not provide. We handed both these problems by designing for variable, nonzero memory access latency – seven cycles when the CPU has bus access, and indeterminately longer when the CPU is waiting for the bus.

3.2. Software

3.2.1. Development Toolchain

We used existing tools and created a number of new ones to develop software for the Beta. The final output of our toolchain is the full-length (225 KB) initial contents of system memory. Into this assembly we need to combine the kernel and program code as well as initial data structures used by the program (e.g., display sprites, dance moves with time coefficients).

We used the Beta ISA port of gcc created by the 6.004 staff to compile C code to Beta assembly instructions. We used regular expressions of our own derivation to perform post-processing steps on gcc's assembled output in order to make it compatible with uasm. We then used uasm to assemble this along with the kernel assembly code to a binary stream of Beta machine code. We created a small program (bmp2hex) that converts display sprites (actually in PNG format) to raw binary stream in our display format (2 bytes per pixel). We created a program called makecoe that takes these binary streams (machine code and graphics), merges them at user-specified offsets, and outputs the merged binary stream in the COE format used to initialize the Xilinx BlockRAM. We used a make file to automate this process.

3.2.2. Video Game Software

Our video game software design was separated into two levels – game control and display construction. The game control routines were to read the periodically-sampled step data and the current time (as counted by the timer interrupt handler). A data structure was devised to store the times of each "correct" step in the dance routine. A user's step at a given time was identified by comparing the current pad sample to the previous sample; if the previous sample indicated that no buttons were stepped on, or if the two samples differed, the program interpreted that the user had stepped on a button in the last 1/100 sec.

[3] The lone exception being that we did not support software traps, which would be necessary were we to develop an operating system for our platform. We disabled the parts of the kernel that serviced these traps as well as those with bsim-specific hooks.

In this case, the algorithm compared the step read in against the "correct" step closest (either after or before) to and within ½ sec of the current time. The step found would be marked as "scored", and a score proportional to the difference between the step's "correct" time and the actual time would be awarded if the step was correct (e.g., the user hit the correct button on the pad).

The other major responsibility of the software was to construct the visuals to be displayed by the video module. To do so, the software constructs a frame buffer which stores the 16-bit RGB representation of each of the 76,800 pixels (320x240) to be displayed on-screen. Subroutines were developed to fill in portions of the frame buffer with a desired color, copy (or "bitblt") pre-stored pictures (or "sprites") into the framebuffer, and draw lines. We demonstrated these routines superimposing text and animating the color of an onscreen region.

3.2.3. Debugging practices

During the development process, we used complementary tools to debug our design. One major asset in this process was bsim, a Beta simulator created by Prof. Chris Terman and the 6.004 staff. We used bsim to simulate execution of our code assemblies in order to work out software bugs. This sped the design process considerably because one run of CoreGen to integrate software changes into the FPGA memory often took up to half and hour. We used a number of techniques to enhance bsim's functionality. For example, bsim provides no way to run until a certain breakpoint in the code; we simulated this ability by adding an inline assembly statement in our code that would store to an out-of-range address, causing bsim to pause. To simulate inter-module integration, we used ModelSim in behavioral and post-place-and-route modes. We used this method to confirm that after handling a certain number of timer interrupts, the Beta correctly granted the bus to the pad interface, which wrote the sampled input to memory and yielded the bus back to the Beta.

3.2. RAM

Although we were unable to create an interface to the ZBT SRAM, we created a RAM controller that conformed to the same shared bus interface and controlled a 56320x32 BlockRAM internal to the FPGA. That we were able to switch RAM technologies with relative ease illustrates the power of our design's modularity – because the numerous devices using the bus conformed to an agreed protocol, the only device requiring modification was the RAM's dedicated controller.

The RAM controller interfaces with a bidirectional, shared 56-bit bus, of which 2 bits are read/write control, 22 are address bits[4], and 32 are data bits. The controller implements a simple protocol: when the read control line is pulsed, it samples the address lines and three cycles later, supplies the word read from that address. Writes are performed similarly.

A major source of complexity in the RAM controller was that it was required to support non-word-aligned memory operations (i.e., those that begin at bytes that are not multiples of four). Because the underlying memory components are 32 bytes long, this required two read operations along with result masking and shifting to be performed. Because the Xilinx memory did not have byte write select control, performing unaligned writes would have been very difficult – the two addresses to be written to would first have to be read and the write contents masked into them before they were both written back. This was not implemented; the software was instead extensively analyzed and optimized to prevent unaligned writes.

3.3. Bus Arbiter

[4] There are 22 address bits rather than 18 because we originally planned to address 4MB of memory.

The bus arbiter module mediates access to the shared bus. It implements a simple cooperative algorithm – a device is granted access and may use the bus until it yields access. In order to prevent bus contention, access is by design mutually exclusive – while one device is using the bus, no other device may. The arbiter takes a signal (BAXv) from the CPU indicating when it should give another device bus access, and for which device it should do so. The CPU is otherwise a generic device on the bus – while another device (e.g., the pad controller) uses the bus, the CPU's incoming bus access signal is low and its execution stalls upon the next instruction requiring memory access (i.e, the next write or the next read causing a cache miss).

3.4. Audio

3.4.1 Overview

The audio module manages all audio played and generated by the DDR system. The original design called for the game music to be stored digitally in the system and played back by the audio subsystem. The design evolved to one where the game music is digitally looped through the FPGA, and a short sound effect is stored digitally and played on demand. The audio system is built around the National LM4550 AC97 audio codec on the lab kit. A control unit manages two minor FSMs that send and receive AC97 frames to and from the codec. A block diagram of the audio module is included in Figure 5.

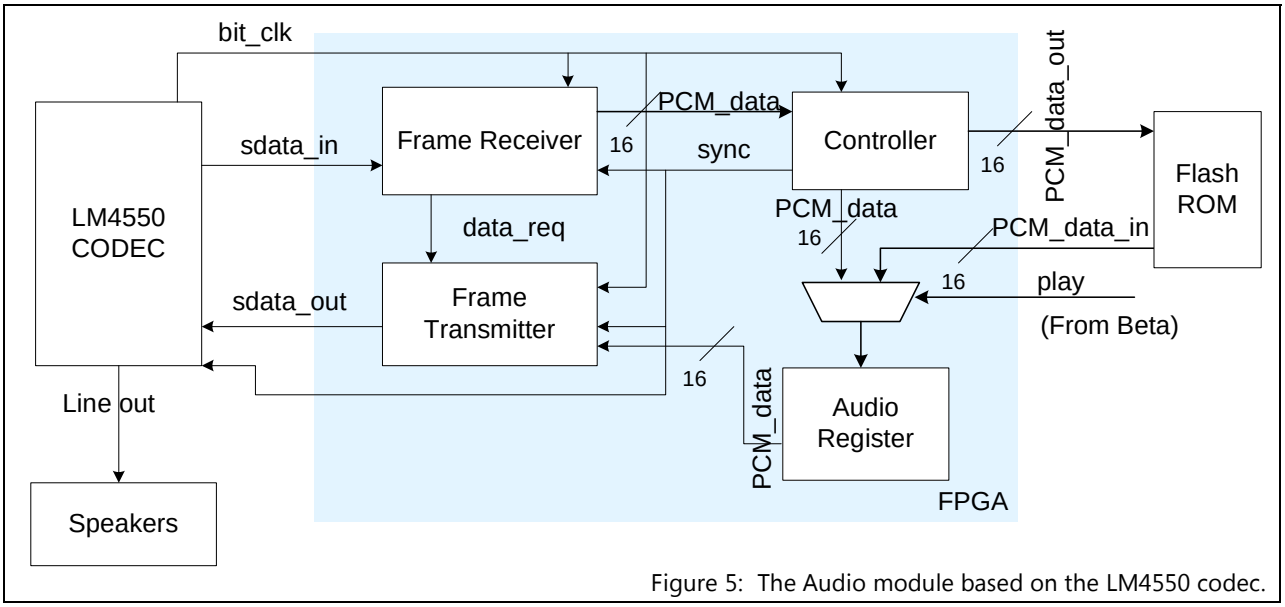

Figure 5: The Audio module based on the LM4550 codec.

3.4.2. Controller

The audio controller operates in three modes: pass-through mode, play mode, and record mode. The controller module controls which mode the audio system is in based on single-bit-inputs *play* and *record*. Additionally, the controller receives the AC97 bit clock from the codec, and generates a sync signal for the codec. An 8-bit bit_count vector is generated and supplied to the frame receiver and transmitter FSMs. The AC97 data is frame based, and data is sent serially along the single-bit sdata line. The sync signal indicates the beginning of a new frame, and the bit_count represents the bit-offset in the frame.

In all modes, frames are sent and received from the CODEC. Incoming analog audio is digitized by the ADC in the codec and stored in the audio receive register. Outgoing PCM digital audio stored in the audio transmit register is interpolated by the DAC in the codec and outputted through the line out port as analog audio.

In pass-through mode, the audio receive and audio transmit registers are connected together, so the incoming audio stream is passed through to the outgoing audio stream, going through the ADC and the DAC in the process. No modifications are made to the digital audio data.

In record mode, digitized incoming data from the audio receive register is sent out of the audio module to the flash ROM module and stored in the ROM for later playback. In addition, control signals are sent to the ROM controller indicating that it should store each incoming audio sample as it is received. Nothing is presented to the line output when in record mode. Due to timing limitations of the flash ROM controller, only two or three samples of audio could be recorded per second. Playing back these stored samples at full 48kHz sample rate results in a scratchy sound effect that works well for the game. Record mode is designed to be used only once when the system is initially setup, and the unit must be manually configured to record mode by using a switch on the kit.

In play mode, digital audio is read from the flash ROM and stuffed into the audio transmit register for conversion by the DAC. The audio controller sends control signals to the flash ROM when it would like to start playing and whenever it needs an audio sample read from the ROM. Playback is started over each time the sound effect is played. Play mode is triggered by a dedicated output signal from the Beta when it wants the sound effect to be played. Originally the audio controller was to read from a dedicated location in memory to receive play signals from the Beta. The audio-memory interface was scrapped, however in favor of the dedicated play signal to reduce complexity and development time.

3.4.3 Frame Receiver

The frame receiver receives AC97 frames from the codec and extracts data and control signals.
Control signals indicating whether the frame contains valid PCM audio data (data_valid), and whether the controller and transmitter FSM should send outgoing PCM data to the codec in the next frame (data_request). The receiver is implemented as an FSM based on the bit_count input from the controller. The bit_count value resets at each rising edge of the sync signal. Based on the bit_count value, the receiver routes the bits incoming from the codec on sdata_in to the appropriate place. The bit_count values and corresponding slot descriptions are included in Table 1. If the data_valid bit it set, PCM data from the frame is sent to a temporary register. On the rising edge of the sync signal, the temporary register is copied to the output register. If the data_request bit is set, a signal is sent to the transmitter indicating that it should send data in the next frame. The receiver is negative-edge-triggered to sample data that changes at the positive clock edge by the codec.

Table 1: Incoming AC97 Frame.

Bit_count / Bit Range	Description
4-5	Data valid left/right
25-26	Data_request left/right
57-76	Slot 3: PCM left
77-96	Slot 4: PCM right

3.4.4 Frame Transmitter

The frame transmitter constructs AC97 frames and sends them to the codec. Two types of data are included in the outgoing frames: control register data and audio data. The transmitter generates frames to send one bit at a time. The control data is used to configure registers in the codec for the desired configuration. The control commands sent to the codec are summarized in Table 2. A frame count is maintained (starting with zero on reset), and one command is sent in each frame until all commands are issued. The frame count is allowed to wrap around, so the command sequence is sent repeatedly. Command values are hard-coded into the Verilog module. Commands are setup in a command register, and the command register data is sent serially during the appropriate part of the frame based on bit_count. A possible extension would have been to include a digital volume control (up/down buttons) that sends commands to the codec.

Table 2: AC97 Configuration Commands.

Address	Data	Description
0x02	0x0000	Un-mute line out
0x04	0x0000	Un-mute headphones
0x0C	0xFFFF	Mute line in to Mix 2
0x0E	0xFFFF	Mute mic in
0x10	0xFFFF	Mute line-in to mix 1 (no analog loopback)
0x1A	0x0404	Record select = line in
0x1C	0x0000	Record gain = 0dB
0x18	0x0808	Unmute DAC output – slightly attenuate
0x20	0x8000	Bypass 3D Surround

Audio data is stuffed into the frame based on the data_request input to the transmitter. If the data_request signal is received from the receiver, PCM data is loaded from the audio transmit register at the start of the frame, and the data_valid register is set to 1. The bit_count values and associated slots and data are summarized in Table 3. The transmitter is positive-edge-triggered to set up data for codec to sample on falling clock edge. The logic in the receiver and transmitter that sends and receives the frames is clocked based on the 12.5MHz bit clock from the audio codec. The logic in the transmitter that loads the PCM data from the audio transmit register is clocked on the 27Mhz system clock.

Table 3: Outgoing AC97 frame summary.

Bit_count / Bit Range	Description
0	Frame valid
1	Command address valid
2	Command data valid
3-4	PCM data valid left/right
16-35	Command address
36-55	Command data
56-75	PCM left
76-95	PCM right

3.4.5 Testing

The audio units were tested both in simulation and in hardware. The receiver and transmitter units were tested separately and in concert. The transmitter was tested by setting its control and data inputs, and viewing the simulation of sdata_out and verifying that the appropriate slots in the frame were filled appropriately. To test the receiver, a test bench was written that generated an AC97 frame and sent it serially to the receiver. Bit_count values shown in the table above were derived from the LM4550 data sheet so frames could be verified. The entire audio module containing both the sender and the receiver was tested using a test bench that generated AC97 frames as well. All simulations were behavioral.

Testing in hardware was more difficult. Due to limits on logic analyzer pins, the 8-bit bit_count could not be observed to verify that frame slots were being sent at the correct time. A spreadsheet was created listing the offset in nanoseconds of each relevant bit of the frame, and the offsets were used to locate the corresponding parts of the frame on the logic analyzer.

Once the synthesized audio unit passed-through audio, a test was necessary to verify that the ADC and DAC were actually being used (that the loopback was indeed digital). To accomplish this, the sample rate was lowered to 4KHz. The frequency response content of the output audio was sufficiently reduced, indicating digital loopback. Additionally, the audio transmit register was disconnected, and it was observed that the audio stopped playing.

3.5. User Input – Pad

3.5.1. Pad Controller

The user interface for the game consists of the dance pad or mat and the pad controller module (a synchronizer and an FSM). A block diagram is included in Figure 6. The pad used is a standard “DDR Pad” for the Sony PlayStation. The pad has a clocked serial interface to be used with the PlayStation. The original design called for use of the serial interface to the pad. Lack of a trustworthy schematic of the serial protocol, however, made it impractical to reverse-engineer the protocol. Instead, the pad was opened up to access a parallel bus with one bit for each of the ten buttons on the pad. The parallel bus was fed directly into the FPGA.

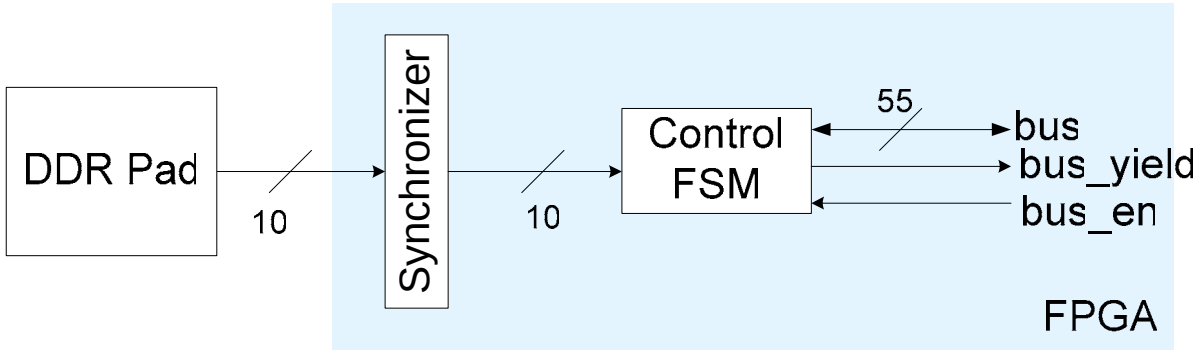

Figure 6: The pad controller module.

The control unit contains a synchronizer and a control FSM. The synchronizer is a three-register-chain unit that ensures that asynchronous inputs from the pad do not put any FPGA logic in a metastable state. The FSM samples the input from the pad and stores input in a dedicated RAM location. The FSM sits in the WAIT_SAMPLE state until it receives a bus enable signal from the bus controller, indicating that the FSM has permission to access the RAM. The FSM then stores data from the output of the synchronizer to the RAM by pulsing the read line on the bus and providing address and data. Data is presented to the bus for a conservative two clock-cycles (only one clock cycle is required) just to make sure all setup and hold times are met. The FSM then cycles through seven sates to wait for the latency in the RAM controller. This delay is present to ensure that the RAM controller is ready to accept input as soon as the pad controller yields the bus. After the delay, the FSM asserts the bus_yield signal for two cycles, stalls for two cycles, and returns to the WAIT_SAMPLE state. The two cycle stall is present to give the bus_en signal a chance to go low before returning the WAIT_SAMPLE state. If bus_en were still high upon return to the WAIT_SAMPLE state, the FSM would launch into another RAM write despite the fact that the bus has already been yielded.

3.5.2 Testing

The bus controller was tested in simulation to verify that the input data gets presented on the data bus line of the RAM. Most importantly, simulation verified that the pad controller properly yields the bus and does not write to the bus after the bus is yielded.

3.6. Video

3.6.1 Purpose

The general purpose of the video is to be able to display each screen of the video game or any function run by the Beta Processor. Because of the focus on generality in this project, the video component was designed to read from a frame buffer location in RAM and convert the data in memory to pixel values, assuming that the processor has previously written relevant values to those memory locations. In addition to being an important part of the output feedback to the user, the video can also be a critical debugging tool to visualize other components of the digital design. For example, not only can the Beta processor output sprites onto the screen, but it can also write its own internal signals to the frame buffer.

3.6.2 Implementation

VGA video was used in this design instead of composite video because of the simpler interface to the VGA video DAC. The VGA monitors receive their data from the ADV7125 VGA DAC chip. Several important signals had to be generated by the video control unit. The horizontal and vertical sync signals guide the video as to where to output signals on the screen. The horizontal and vertical blank signals blank out the data while the sync signals are being sent to the data chip. The blanks must envelop their respective sync signals.

The pixels in the AD7125 chip receive three inputs: red, blue and green. Each color in a pixel is assigned an 8-bit value, allowing the display of 256 shades per primary color. In this project, because of memory constraints, each pixel received 15 bits of data; five each for red, green and blue, which allowed us to have 2^{15} colors, or around 32,000 colors. The lower order three bits for each of the pixel colors were tied to ground so that black can be achieved if all the data was set to low.

Each piece of data read from memory would produce two pixel values, from bits 31 to 17 and from bits 15 to 1. Bits 16 and 0 were not used. The screen size we displayed on the monitor was 640 by 480. However, since there is only room for a 320 by 240 image to be stored on the FPGA, data had to be extrapolated. Each pixel is written twice horizontally and is also repeated on the following line.

The video control was implemented in a major-minor FSM scheme. A block diagram of this overall scheme is further outlined in Figure 7. The major FSM controls the vertical sync pulse and keeps track of which vertical line it is currently on. The horizontal sync FSM is the minor FSM, which is controlled by the vertical sync component. The vertical sync module sends a signal to the horizontal sync module to begin writing a line. It will also indicate which line the current screen drawing is on, allowing the horizontal sync component to keep track of where it is on the screen and what address to access in order to retrieve the correct piece of data.

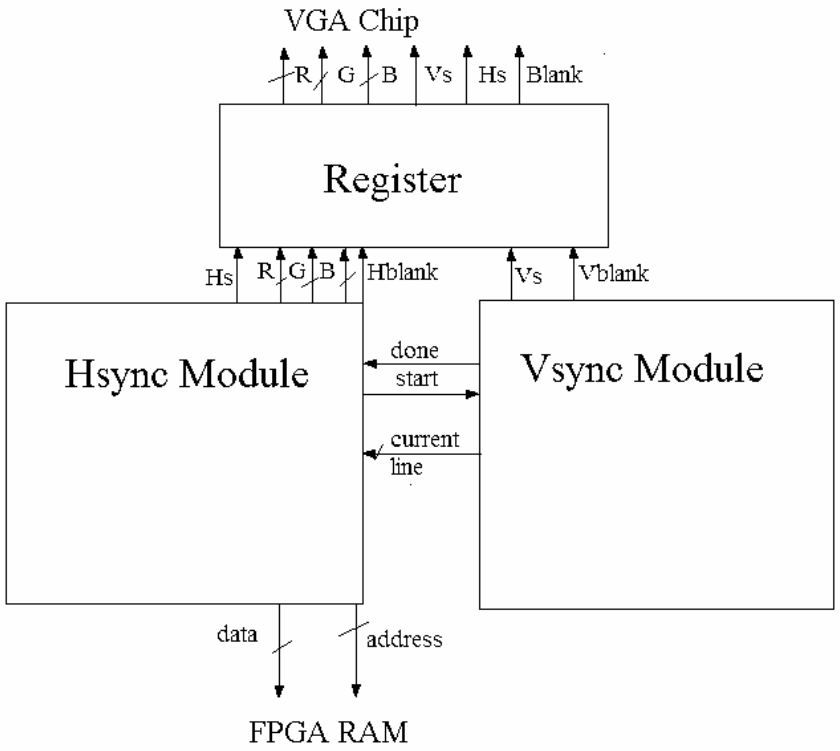

Figure 7: Block Diagram of Video Module

On reset, both FSM's go to the initial states. The vertical sync FSM then goes first to the front porch, where blanking is enabled while syncing is not. It then enables the syncing signal along with the blanking signal. From there, the FSM will transition to the back porch, where once again the sync is high and the blank is low. After that, the data is retrieved and displayed for 480 lines. When the data finishes, the state machine will transition back to the front porch portion and the process repeats.

The horizontal sync is very similar. It begins by outputting each individual piece of data twice. It cycles through, displaying the first 15 bits of the data twice on the screen, and then the second 15 bits twice. After outputting 640 pieces of data to the screen, the FSM will go to the front porch mode, then to outputting the sync pulse and finally to the back porch.

3.6.3 Timing Issues

The major timing issue involved with the functionality of the video display is the blanking and the syncing of the video as well as being able to send out a new piece of data every cycle to the video screen. Figure 8 shows the needed blanking and syncing timings for a 640 by 480 pixel display using a 27 MHz clock. The front porch, sync pulse and back porch period constraints must be met. If the constraints are not met, then the data can either be misaligned on the screen, or simply not be displayed altogether. Also, if a piece of data is unable to be retrieved every clock period, pipelining must be done in order to increase throughput of the output pixels.
In this project, a worst-case 2 cycle delay between the pulse of read to the output of valid data can be handled. The FSM uses one piece of data every 4 clock cycles because it can get 2 pixels per 32-bits and two clock cycles per each pixel information.

Also, in addition to the data to the VGA chip, the horizontal and vertical syncs are fed in directly into the VGA connector. Therefore, not only must they be registered, but they must have a two-clock-cycle delay that the rest of the signals will suffer when going through the VGA chip.

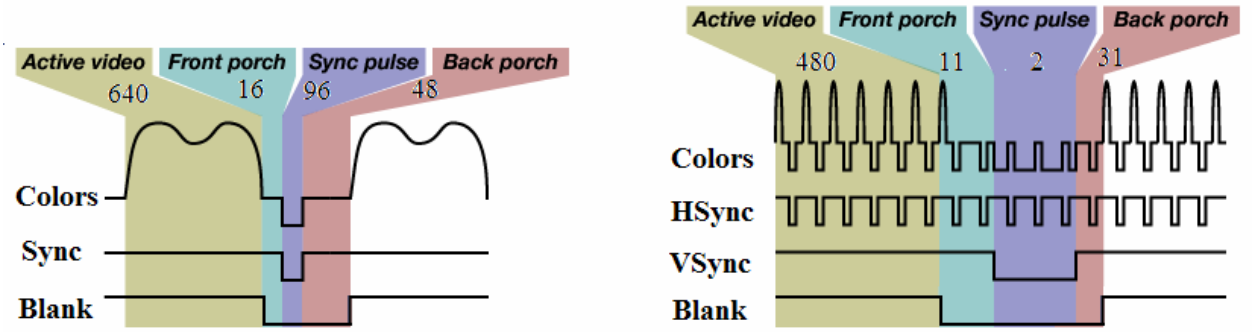

Figure 8: Timing of Data, Sync, and Blank Signals For 640 by 480 VGA Video
Courtesy: Nathan Ickes.

3.6.4 Debugging

There were several debugging steps employed in trying to perfect the signals coming out of the video unit. First, there was an attempt to simply draw any image on the screen. This tests the correctness of the blanking and syncing signals in order to output any data at all. The image drawn out on the screen was a solid color.
Next, one piece of data was tied as the input and contains pixel values for two different colors. The video unit will then draw two alternating colors on the screen with each color filling up vertical lines. This showed whether the blanking and syncing signals lined up from one line to the next. If there were discrepancies, we saw a series of jagged lines rather than a series of smooth vertical lines cutting down the screen.

Finally, the testing phase included reading from a set of preset RAM locations which contained an image in order to see whether the addressing was correct and whether it was displaying the correct data in the correct locations.

If the functionality of the module could not be guaranteed, ModelSim and Max+II were used to simulate the module. It was easy to see whether syncs were being enveloped by blanking signals and whether there were 640 pieces of data being displayed per line.

3.7. Flash ROM

3.7.1. Purpose

The Flash ROM is non-volatile memory that can store data, like a RAM, but does not lose that data if the power to the device is taken away. Regular SRAM would lose all of its data on power-off; therefore, it is difficult to store data on SRAM over long periods of time because of the potential loss of power. The Flash ROM on the kits provided a method to safely store data that could be used over and over at a later date. The 28F128J3A chip produced by Intel was the chip used in this project. Not only does it provide the non-volatile memory, it also has the capability to store

a lot of data, because it has 16 Megabytes of storage space. The music therefore can be stored on the Flash ROM once, and played back whenever music is required.

3.7.2. Implementation

In order to store data onto the ROM, one had to first figure out the timing constraints involved with the chip as well as all the signals that the chip required. One also had to learn about the different modes that the flash ROM can be in. One of the most important modes is the erase mode, where the chip erases an entire block of data. It takes the top 7 most significant bits of the address and indicates that section is ready for erase. The user must send in the data 0x20, followed by the data 0xD0, while the address is any address within that section which the user wants to erase. Then, the status signal will indicate when the chip has started and completed its erase operation. The status signal will often indicate what mode the flash ROM is in. If the status is low, the chip is in programming mode and when the signal is high, the chip is available for other operations.

Another mode that the chip allows is the mode to perform a write. After erasing the data from a certain block, one can write to that block exactly once. When erased, the address will have its data set to FFFF, and writing to that address will change the necessary ones to zeroes. In performing such a write, one must first issue the command to write to the chip. To do that, the write enable is set low while the chip is on. Then, the command 0x40 is issued through the address bus. Then, the data is sent. The status of the chip will go low to indicate that it is making a write. Then, the status will go back high to represent that the write is done.

Finally, one must also be able to read from the Flash ROM. In order to read from the ROM, the user must be in read array mode. In order to get into read array mode, the signal 0xFF is sent to the data bus of the chip while the write is enabled. Then, as long as there are only read operations being made, the chip remains in the read array mode. However, as soon as one changes the operation, it would have to be set to read array mode before continuing.

In order to control this ROM, a major-minor FSM structure was used. The block diagram can be seen in Figure 9. The minor state machine controls the chip directly; setting the output enable and write enable high or low. If given the signal to erase, the two-step procedure above is used. The same two-step procedure is used for any read or write operation as well. In addition, the minor FSM can be told to prepare itself for a read, where it will set itself into the read array state. While nothing is going on, the FSM stays in an idle state where it will wait for any one of the signals to be sent to it. If it completes that operation, except for the preparation to read, a done signal will be sent back to the major FSM controlling the minor state machine.

The inputs from the users go into the major FSM, which then go down to the minor FSM. For example, the user tells the system when to erase. After given a signal to erase, the FSM will loop through all the blocks that it can erase and erase them. If given the command to write, the ROM controller will go either to the first address, or to another address to write to that location, and same with read. There is a signal that must be given to the system before it can make any reads, that is to set the system into read array mode. On either an erase or write operation, the FSM will wait for the done signal to be raised in the minor FSM before going on to the next erase or write.

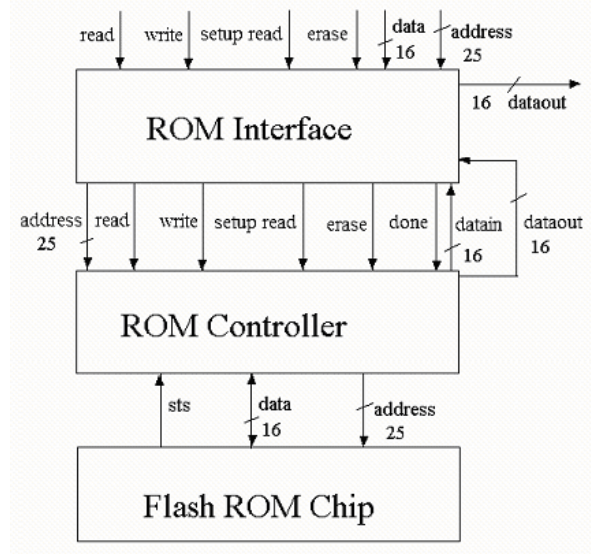

Figure 9: Block Diagram of ROM Control Module

Problems and Debugging

There was one major timing issue with the ROM. It was how slow it took the ROM to perform writes and erases. The write appears to take on the order of hundreds of milliseconds despite the datasheet presenting information otherwise. The block erase can take up to 5 seconds, although it typically took around a second to perform. A timing diagram involving generic writes is outlined in Figure 10.

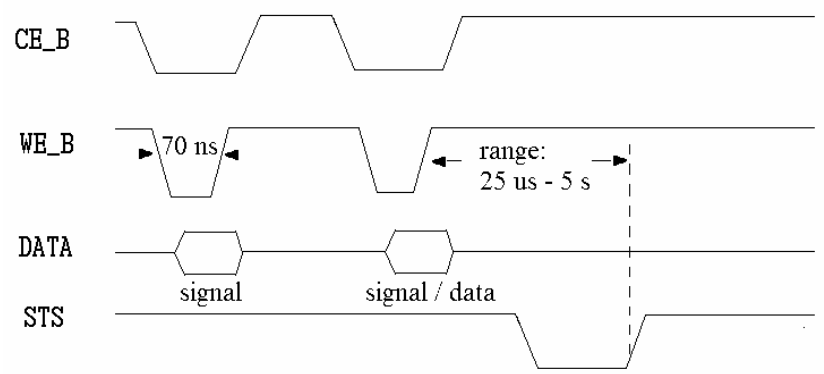

Figure 10: Timing Diagram of Writing to Flash ROM

4. Conclusions

4.1 System Integration

4.1.1 FlashROM

The main problem was in integrating this Flash ROM with collecting audio samples. The audio samples happened at the slowest possible sample rate supported by the audio hardware—4 KHz. However, the ROM would sample only twice per second. Thus, there were not enough data points to create an audible piece of music. However, it was not always clear that this was the problems, because we were not sure whether we were writing correctly or reading correctly from the ROM. The method we employed to attempt to debug the ROM was simply by sending every signal out onto the logic analyzer. It was clear that the ROM was getting dataat first, but quickly stopped getting data as we sent more and more requests for reads. In addition, we tested moving addresses manually, but assigning addresses and data inputs to switches and seeing if those values that were input are actually the values obtained in the output.

There was no way of getting around the problem of sampling the audio a lot faster than it is being recorded. The only salvageable solution was to turn the audio, which we had stored on the ROM and incorporate that somehow in the rest of the digital system design. We chose to include the audio on top of the previous audio because it shows the benefit at which the digital signals store on the Flash ROM can be integrated into the external, analog data. Not only are they easily mixable through the audio codec once they are digitized, but the codec can also take sounds from various sources. Thus, that was the main functionality of the ROM and the rational behind its functionality.

4.1.2. The Whole Thing

In the demonstrated version of the project, the system was not fully integrated. All modules functioned, but not all modules were connected together. The Beta and the video worked to display images, sprites, and an animated gradient, illustrating program execution and significant proof-of-concept. The pad was connected to the FPGA and as mentioned, simulation illustrated proper triggering by timer interrupt-invoked kernel code, and proper use of the bus by the pad controller. The audio played and passed-through, but the dedicated “play” output signal from the beta did not work properly.

It is easy to see how with a bit more work the system could be integrated. Once the pad can write to RAM, the beta can display arrows on the screen when a button is pressed. The Beta can output a “start” signal to the audio and is able to keep its own time, meaning that we have the capability to run synchronized music. Now a sound effect could be played and an arrow could be displayed each time a button on the pad is pressed. This integration was attempted at the last minute, but a strange error in the synthesis stage prevented confirmation of the promising results we observed in simulation.

4.2 Reflection

The implementation of the audio and pad controllers went fairly smoothly. As usual, neither of them worked the first time they were synthesized. However, with careful debugging on the logic analyzer, the two modules worked with a reasonable amount of effort.

The road was not so well-paved for the ZBT SRAM controller, which was not finally included in out project. The ZBT controller never worked, despite consuming an entire week of work for one member of the group. When the ZBT controller still did not function and the project was due in four days, the ZBT on-kit RAM was scrapped in favor of a CoreGen RAM in the FPGA. Several factors caused the ZBT controller to be a compete failure: First, the high-performance nature of the ZBT chip made it difficult to use conservative timing. Additionally, timing complications with the ZBT mechanisms were difficult. Finally, the need for byte-addressed memory for the Beta processor added complexity to the RAM controller. Once the ZBT interface was figured out and the controller appeared to read and write from memory, the values read and written appeared to be inconsistent. However, by this point the rest of the project was blocking on the RAM controller, so Jacob created a controller for the FPGA RAM to allow software development and testing to continue.

While the generality of our approach may have earned us flexibility at the expense of complexity, we feel that it was nevertheless well-justified. We ended up with a fully-functional CPU and a quickly developing software set. Every half-hour, we compiled new software and had a neat new video demo to look at, underlining the flexibility granted by encoding display and game logic in software.

Appendix: Verilog Code

```
/*** betaputer.v ******************************
  Jacob Kitzman | 6.111 Final Project S04
  beta computer top-level file
     instantiates: the beta cpu, shared bus arbiter,
system ram (no longer using zbt sram) and shared bus
controller interface, clock interrupt generator
*********************************************/

module betaputer(clk, reset, bus, IRQvec, devYield,
devEnable, _dbg_instr, _dbg_pc, _dbg_other,
_dbg_mem_rdata, vid_addr, vid_data, direct_io);
        input clk;
        input reset;
        input[7:0] IRQvec;

        input[15:0] devYield;
        output[15:0] devEnable;

        inout[55:0] bus;

        output[31:0] _dbg_instr;
        output[31:0] _dbg_pc;
        output[31:0] _dbg_other;
        output[31:0] _dbg_mem_rdata;

        input[15:0] vid_addr;
        output[31:0] vid_data;

        output[3:0] direct_io;

        wire[3:0] BAXv;
        wire BAXvchg;

        wire clock_irq;

        beta cpu
            (.clk(clk),
                .reset(reset),
                .IRQvec({clock_irq, IRQvec[6:1],
                    clock_irq}),
                .CPUBus_en(devEnable[0]),
                .bus(bus),
                .BAXv(BAXv),
                .BAXvchg(BAXvchg),
                .direct_io(direct_io),
                ._dbg_instr(_dbg_instr),
                 ._dbg_pc(_dbg_pc),
                 ._dbg_other(_dbg_other),
                 ._dbg_mem_rdata(_dbg_mem_rdata));

        bus_arbiter arb(
                .clk(clk),
                .reset(reset),
                .BAXv(BAXv),
                .BAXvchg(BAXvchg),
                .devEnable(devEnable),
                .devYield(devYield)
        );

        sysram_busintf fakeram(
                .clk(clk),
                .reset(reset),
                .bus(bus),
                .vid_addr(vid_addr),
                .vid_data(vid_data));

        b_clock_interrupt clkirq(
                .clk(clk),
                .reset(reset),
                .clock_irq(clock_irq)
        );
endmodule
```

```
/*** beta.v ***********************************
        Jacob Kitzman | 6.111 Final Project S04
        beta risc cpu
*********************************************/
module beta(clk, reset, IRQvec, BAXvchg,  BAXv,
CPUBus_en,  bus,  direct_io, _dbg_instr, _dbg_pc,
_dbg_other, _dbg_mem_rdata);

        input clk, reset;
        input[7:0] IRQvec;
        output BAXvchg;
        output[3:0] BAXv;
        output[3:0] direct_io;
        reg[3:0] direct_io;
        input CPUBus_en;

        inout[55:0] bus;

        wire[2:0] PCsel;
        wire PCle;
        wire MEMRle;
        wire IFETCHready, IFETCHreq;
        wire MEMOPready, MEMOPreq;
```

```
        wire INSTRle;
        wire[1:0] WDsel;
        wire WAsel;
        wire RA2sel;
        wire Asel;
        wire Bsel;
        wire[3:0] ALUFN;
        wire WERF;
        wire BUSwrite;
        wire BUSread;
        wire MEMOPsel;
        wire[31:0] instr;
        wire[6:0] IRQid_handling;
        wire[4:0] _dbg_ctrlsig_state;

        wire[31:0] PCmuxed;
        wire[31:0] alu_out;
        wire[31:0] mem_rdata;
        wire[31:0] wd_muxed;
        wire[31:0] asel_muxed;
        wire[31:0] bsel_muxed;
        wire[4:0] Rb_muxed;
        wire[31:0] mem_addr_muxed;
        wire[4:0] wasel_muxed;
        wire[31:0] Rda, Rdb;
        wire[31:0] mem_rdata_buffered;
        reg Z;

        wire[31:0] PC;
        reg[31:0] PCplus4;
        reg[31:0] PCp4pLiteral;
        always @(PC) PCplus4 = PC + 32'd4;

        reg[31:0] sxt_lit;
        always @(PC or instr) begin
                sxt_lit = { {16{instr[15]}}, instr[15:0]
};
                PCp4pLiteral = PC + 32'd4 + {
{14{instr[15]}}, instr[15:0] , 2'd0 };
        end

        busmux8 pcmux(
                .sel(PCsel),
                .out(PCmuxed),
                .bus0(PCplus4),                 //
pc+4, next instruction.
                .bus1(PCp4pLiteral),            //
branch target
                .bus2(Rda & 32'h7FFFFFFC),      //
jump target - may not set the supe bit.
                .bus3(32'hDEADBEEF),
                .bus4({20'h80000, 3'b000,
IRQid_handling[6:0], 2'b00}),     // interrupt vector.
                .bus5(32'h80000000),            //
startup address.
                .bus6(32'hDEADBEEF),
                .bus7(32'hDEADBEEF));

        reg32 pcreg(
                .clk(clk),
                .reset(reset),
                .d(PCmuxed),
                .q(PC),
                .le(PCle)
        );

        reg32 memreadreg(
                .clk(clk),
                .reset(reset),
                .d(mem_rdata),
                .q(mem_rdata_buffered),
                .le(MEMRle)
        );

        busmux2 mem_addr_mux(
                .sel(MEMOPsel),
                .out(mem_addr_muxed),
                .bus0(PC),
                .bus1(alu_out)
        );

        busmux4 wdsel_mux(
                .sel(WDsel),
                .out(wd_muxed),
                .bus0(PCplus4),
                .bus1(alu_out),
                .bus2(mem_rdata_buffered),
                .bus3(32'd0)
        );

        busmux2 aselmux(
                .sel(Asel),
                .out(asel_muxed),
                .bus0(Rda),
                .bus1(PCp4pLiteral)
        );

        busmux2 bselmux(
                .sel(Bsel),
                .out(bsel_muxed),
                .bus0(Rdb),
                .bus1(sxt_lit)
        );

        b_alu alu(
                .alufn(ALUFN),
                .a(asel_muxed),
                .b(bsel_muxed),
                .out(alu_out)
        );

    reg32 instr_reg(
                .clk(clk),
                .reset(reset),
                .d(mem_rdata),
                .q(instr),
                .le(INSTRle)
        );

        wire[3:0] direct_io_del0;
        always @(posedge clk) begin
                direct_io <= direct_io_del0;
        end

        b_cpuTestBusClient bus_client(
                .clk(clk),
                .reset(reset),
                .IFETCHready(IFETCHready),
                .MEMOPready(MEMOPready),
                .IFETCHreq(IFETCHreq),
                .MEMOPreq(MEMOPreq),
                .CPUbus_en(CPUBus_en),
                .wr(BUSwrite),
                .re(BUSread),
                .wdata(Rdb),
                .address(mem_addr_muxed),
                .rdata(mem_rdata),
                .bus(bus),
                .direct_io(direct_io_del0),
                .BAXVchange(BAXvchg),
                .BAXVchgowner(BAXv)
        );

        b_ctrlsig control_sigs(
                .clk(clk),
                .reset(reset),
                .IRQvec(IRQvec),
                .IFETCHready(IFETCHready),
                .MEMOPready(MEMOPready),
                .IFETCHreq(IFETCHreq),
                .MEMOPreq(MEMOPreq),
                .INSTRle(INSTRle),
                .Z(Z),
                .PCle(PCle),
                .PCsel(PCsel),
                .WDsel(WDsel),
                .WAsel(WAsel),
                .RA2sel(RA2sel),
                .Asel(Asel),
                .Bsel(Bsel),
                .ALUFN(ALUFN),
                .WERF(WERF),
                .BUSwrite(BUSwrite),
                .BUSread(BUSread),
                .MEMOPsel(MEMOPsel),
                .instr(instr),
                .PCsupv(PC[31]),
                .IRQid(IRQid_handling),
                .MEMRle(MEMRle),
                ._dbg_state(_dbg_ctrlsig_state)
        );
```

```
        lilbusmux2 ra2mux(
                .sel(RA2sel),
                .out(Rb_muxed),
                .bus0(instr[15:11]),
                .bus1(instr[25:21])
        );

        lilbusmux2 waselmux(
                .sel(WAsel),
                .out(wasel_muxed),
                .bus0(instr[25:21]),
                .bus1(5'd30) // XP
        );


        always @(Rda)
                Z = (Rda==32'h0);

        b_regfile register_file(
                .clk(clk),
                .reset(reset),
                .Ra(instr[20:16]),
                .Rb(Rb_muxed),
                .Rda(Rda),
                .Rdb(Rdb),
                .Wa(wasel_muxed),
                .Wdata(wd_muxed),
                .Wen(WERF)
        );      // if desired can wire up
_dbg_WenToChip, _dbg_AddrBToChip.

        output[31:0] _dbg_instr;
        output[31:0] _dbg_pc;
        assign _dbg_instr = instr;
        assign _dbg_pc = PC;

        output[31:0] _dbg_other;

        assign _dbg_other[31] = IFETCHready;
        assign _dbg_other[30] = IFETCHreq;
        assign _dbg_other[29] = MEMOPready;
        assign _dbg_other[28] = MEMOPreq;
        assign _dbg_other[27] = INSTRle;
        assign _dbg_other[26] = Z;
        assign _dbg_other[25] = PCle;
        assign _dbg_other[24:23] = WDsel[1:0];
        assign _dbg_other[22] = WAsel;
        assign _dbg_other[21] = RA2sel;
        assign _dbg_other[20] = Asel;
        assign _dbg_other[19] = Bsel;
        assign _dbg_other[18] = WERF;
        assign _dbg_other[17:14] =
_dbg_ctrlsig_state[3:0];
        assign _dbg_other[13:11] = PCsel;
        assign _dbg_other[10:8] = 2'd0;
//      assign _dbg_other[3:0] = _dbg_dmcache_state;

        output[31:0] _dbg_mem_rdata;
        assign _dbg_mem_rdata = mem_rdata;

endmodule
```

```
/*** b_alu.v ***************************************
        Jacob Kitzman | 6.111 Final Project S04
        beta cpu arithmatic logic unit (ALU)
*************************************************/

module b_alu(alufn, a, b, out);
        input[3:0] alufn;
        input[31:0] a, b;
        output[31:0] out;
        reg[31:0] out;

        // ALU function codes.
        parameter ALUFN_ADD   = 4'b0000;
        parameter ALUFN_SUB   = 4'b0001;
        parameter ALUFN_MUL   = 4'b0010;
        parameter ALUFN_DIV   = 4'b0011;
        parameter ALUFN_AND   = 4'b1000;
        parameter ALUFN_OR    = 4'b1001;
        parameter ALUFN_XOR   = 4'b1010;
        parameter ALUFN_AEQB  = 4'b0100;
        parameter ALUFN_ALTB  = 4'b0101;
        parameter ALUFN_ALEB  = 4'b0110;
        parameter ALUFN_SHL   = 4'b1100;
        parameter ALUFN_SHR   = 4'b1101;
        parameter ALUFN_SRA   = 4'b1110;
        parameter ALUFN_A     = 4'b1111;

        always @(alufn or a or b) begin
                case (alufn)
                        ALUFN_ADD:
                                out = a + b;
                        ALUFN_SUB:
                                out = a - b;
                        ALUFN_MUL:
                                out = a * b;

                        ALUFN_AND:
                                out = a & b;
                        ALUFN_OR:
                                out = a | b;
                        ALUFN_XOR:
                                out = a ~^ b;
                        ALUFN_AEQB:
                                out = (a==b);
                        ALUFN_ALTB: begin
                                if (a[31]&&(!b[31]))
        // a NEG and b POS ==> a LT b.
                                        out = 32'h1;

                                else if
((!a[31])&&b[31])// a POS and b NEG => a NOT LT b.
                                        out = 32'h0;
                                else if
((!a[31])&&(!b[31]))      // a POS and b POS => compare.
                                        out = (a<b);
                                else // a NEG and b NEG
=> compare in 2'sC
                                        out = (b>a);
                        end
                        ALUFN_ALEB: begin
                                if (a==b)
                                        out = 32'h1;
                                else begin
                                        if
(a[31]&&(!b[31]))// a NEG and b POS ==> a LT b.
                                                out =
32'h1;
                                        else if
((!a[31])&&b[31])// a POS and b NEG => a NOT LT b.
                                                out =
32'h0;
                                        else if
((!a[31])&&(!b[31]))      // a POS and b POS => compare.
                                                out =
(a<b);
                                        else // a NEG
and b NEG => compare in 2'sC
                                                out =
(b>a);
                                end
                        end
                        ALUFN_SHL:
                                out = a << b[4:0];
                        ALUFN_SHR:
                                out = a >> b[4:0];
                        ALUFN_SRA:
                                out = (a >> b[4:0]) |
({32{a[31]}} & ~(32'hffffffff>>(b[4:0])));
                        ALUFN_A:
                                out = a;
                        default:
                                out = 0;
                endcase
        end
endmodule
```

```
/*** b_ctrlsig.v ***********************************
        Jacob Kitzman | 6.111 Final Project S04
        beta cpu control signal module
*************************************************/
module b_ctrlsig(clk, reset, IRQvec, IFETCHready,
MEMOPready, IFETCHreq, MEMOPreq, INSTRle, Z, PCle, PCsel,
WDsel, WAsel, RA2sel, Asel, Bsel, ALUFN, WERF, BUSwrite,
BUSread, MEMOPsel, instr, MEMRle, PCsupv, IRQid,
_dbg_state);
        input           clk;
        input           reset;
```

```
        input[7:0]      IRQvec;
        input           IFETCHready;
        input           MEMOPready;
        output  IFETCHreq;
        reg             IFETCHreq, IFETCHreq_int;
        output          MEMOPreq;
        reg                     MEMOPreq, MEMOPreq_int;
        output          INSTRle;
        reg                     INSTRle;
        input           Z;
        output          PCle;
        reg                     PCle;
        output[2:0]     PCsel;
        reg[2:0]        PCsel;
        output[1:0]     WDsel;
        reg[1:0] WDsel;
        output          WAsel;
        reg                     WAsel;
        output          RA2sel;
        reg                     RA2sel;
        output          Asel;
        reg                     Asel;
        output          Bsel;
        reg                     Bsel;
        output          BUSwrite;
        reg                     BUSwrite;
        output          BUSread;
        reg                     BUSread;
        output          MEMOPsel;
        reg                     MEMOPsel;
        input[31:0]     instr;
        output          WERF;
        reg                     WERF, WERF_int;
        output[3:0]     ALUFN;
        reg[3:0]        ALUFN, ALUFN_last;
        input           PCsupv;
        output[6:0]     IRQid;
        reg[6:0]        IRQid;
        output          MEMRle;
        reg                     MEMRle;

        // internals
        reg[4:0]                state, next;

        // states
        parameter STARTUP0              = 4'h0;
        parameter STARTUP1      = 4'h1;
        parameter IFETCH        = 4'h2;
        parameter IFETCH_WAIT   = 4'h3;
        parameter REGACCESS     = 4'h4;
        parameter REGACCESS_BR  = 4'hA;
        parameter MEMOP         = 4'h5;
        parameter MEMOP_WAIT    = 4'h6;
        parameter WRITEBACK0    = 4'h8;
        parameter WRITEBACK1    = 4'h8;
        parameter IRQ_SENSED    = 4'h9;

        // debug
        output[4:0]     _dbg_state;
        assign _dbg_state = state;

        reg setPCsel, setPCsel_int;  // should writeback
set pcsel?

        reg[2:0] PCsel_last;
        reg Asel_last, Bsel_last;
        reg[1:0] WDsel_last;
        reg WAsel_last;
        reg RA2sel_last;
        reg MEMOPsel_last;
        reg WERF_int_last;
        reg BUSread_last, BUSwrite_last;

        reg irq_pending_last, irq_pending;
        reg[6:0] IRQid_last;

        always @(posedge clk or negedge reset) begin
                if (!reset) begin
                        PCsel_last <= 3'd5;
                        Asel_last <= 0;
                        Bsel_last <= 0;
                        WDsel_last <= 0;
                        WAsel_last <= 0;
                        RA2sel_last <= 0;
                        MEMOPsel_last <= 0;
                        WERF_int_last <= 0;
                        BUSread_last <= 0;
                        BUSwrite_last <= 0;

                        setPCsel <= 0;

                        ALUFN_last <= 0;

                        irq_pending_last <= 0;
                        IRQid_last <= 7'd0;

                        state <= STARTUP0;
                        IFETCHreq <= 0;
                        MEMOPreq <= 0;
                end
                else begin
                        PCsel_last <= PCsel;
                        Asel_last <= Asel;
                        Bsel_last <= Bsel;
                        WDsel_last <= WDsel;
                        WAsel_last <= WAsel;
                        RA2sel_last <= RA2sel;
                        MEMOPsel_last <= MEMOPsel;
                        WERF_int_last <= WERF_int;
                        BUSread_last <= BUSread;
                        BUSwrite_last <= BUSwrite;
                        IRQid_last <= IRQid;

                        setPCsel <= setPCsel_int;

                        ALUFN_last <= ALUFN;

                        irq_pending_last <=
irq_pending;

                        IFETCHreq <= IFETCHreq_int;
                        MEMOPreq <= MEMOPreq_int;

                        if (next==WRITEBACK0 ||
next==WRITEBACK1 || next==IRQ_SENSED)
                                WERF <= WERF_int;
                        else
                                WERF <= 0;

                        state <= next;
                end
        end

        always @(state or IRQvec or IFETCHready or
MEMOPready or Z or instr or IRQvec or PCsupv or
PCsel_last or WDsel_last or WAsel_last or RA2sel_last or
Asel_last or Bsel_last or     BUSwrite_last or
BUSread_last or MEMOPsel_last or WERF_int_last or
irq_pending_last or                   IRQid_last or
setPCsel or ALUFN_last) begin
                // defaults:
                PCle = 0;
                IFETCHreq_int = 0;
                MEMOPreq_int = 0;
                INSTRle = 0;
                MEMRle = 0;
                //PCsel = PCsel;
                PCsel = PCsel_last;
                WDsel = WDsel_last;
                WAsel = WAsel_last;
                RA2sel = RA2sel_last;
                Asel = Asel_last;
                Bsel = Bsel_last;
                BUSwrite = BUSwrite_last;
                BUSread = BUSread_last;
                MEMOPsel = MEMOPsel_last;
                WERF_int = WERF_int_last;
                ALUFN = ALUFN_last;

                setPCsel_int = 0;

                irq_pending = irq_pending_last |
(IRQvec[7] & ~PCsupv);

                // which interrupt id is requesting our
attention?
                // don't take it if we're already
handling one.
                if (!irq_pending_last && IRQvec[7])
                        IRQid = IRQvec[6:0];
                else
                        IRQid = IRQid_last;
```

```
case (state)
        STARTUP0: begin
                PCsel = 3'd5;
                WDsel = 2'd0;
                WAsel = 0;
                RA2sel = 0;
                Asel = 0;
                Bsel = 0;
                MEMOPsel = 0;
                WERF_int = 0;
                BUSread = 0;
                BUSwrite = 0;

                next = STARTUP1;
        end

        STARTUP1: begin
                PCle = 1;
                IFETCHreq_int = 1;
                next = IFETCH;
        end

        IFETCH: begin
                MEMOPsel = 0;
                PCsel = 3'd0;
                next = IFETCH_WAIT;
        end

        IFETCH_WAIT: begin
                if (IFETCHready) begin
                  INSTRle = 1;
                  next = REGACCESS;
                end

                else
                next = IFETCH_WAIT;
        end

        REGACCESS: begin
        if (instr[31:30] == 2'b10)
        begin
                // OP(Ra, Rb, Rc)
                WERF_int = 1;

                Bsel = 0;
                WDsel = 2'd1;
                RA2sel = 0;
                Asel = 0;
                WAsel = 0;
                PCsel = 3'd0;
                ALUFN = instr[29:26];

                next = WRITEBACK0;
        end
        else if (instr[31:30] == 2'b11)
        begin
                 // OPC(Ra, Rb, Rc)
                 WERF_int = 1;
                 Bsel = 1;
                 WDsel = 2'd1;
                 RA2sel = 0;
                 Asel = 0;
                 WAsel = 0;
                 PCsel = 3'd0;
                 ALUFN = instr[29:26];
                 next = WRITEBACK0;
        end
        else if (instr[31:30] == 6'b01)
        begin
                if (instr[29:26] ==
        4'b1000) begin
                   // LD(Ra,literal,Rc)
                   ALUFN = 4'b0000;
                   WERF_int = 1;
                   Bsel = 1;
                   WDsel = 2;
                   PCsel = 3'd0;
                   Asel = 0;
                   WAsel = 0;
                   BUSread = 1;
                   MEMOPsel = 1;
                   MEMOPreq_int = 1;
                   next = MEMOP;
                end
                        else if (instr[29:26]
                        == 4'b1111) begin
                           // LDR(label, Rc)
                           ALUFN = 4'b1111;
                           WERF_int = 1;
                           WDsel = 2;
                           PCsel = 3'd0;
                           Asel = 1;
                           WAsel = 0;
                      BUSread = 1;
                      MEMOPsel = 1;
                      MEMOPreq_int = 1;
                      next = MEMOP;
                    end
                    else if (instr[29:26] ==
                    4'b1001) begin
                        // ST(Rc,literal,Ra)
                        ALUFN = 4'b0000;
                        WERF_int = 0;
                        Bsel = 1;
                        RA2sel = 1;
                        PCsel = 0;
                        Asel = 0;
                        BUSwrite = 1;
                        MEMOPsel = 1;
                        MEMOPreq_int = 1;
                        next = MEMOP;
                    end
                     else if (instr[29:26] ==
                    4'b1011) begin
                       // JMP(Ra, Rc)
                       WERF_int = 1;
                       WDsel = 0;
                       PCsel = 2;
                       WAsel = 0;
                       next = WRITEBACK0;
                    end
                    else if (instr[29:26] ==
                    4'b1101) begin
                       // BEQ/BF(Ra, label, Rc)
                       WERF_int = 1;
                       WDsel = 0;
                       WAsel = 0;
                       // Z not ready until next
cycle => cannot set PCsel now.
                       setPCsel_int = 1;
                       next = WRITEBACK0;
                    end
                    else if (instr[29:26] ==
                    4'b1110) begin
                       // BNE/BT(Ra, label, Rc)
                       WERF_int = 1;
                       WDsel = 0;
                       WAsel = 0;
                       setPCsel_int = 1;
                       next = WRITEBACK0;
                    end
                end
           end

            WRITEBACK1: begin
               if (setPCsel) begin
                 if (instr[27:26]==2'b01)
                     PCsel = Z ? 1 : 0;
                 else
                     PCsel = Z ? 0 : 1;
               end

               WERF_int = 0;
               BUSread = 0;
               BUSwrite = 0;

               // ignore irq until we're not
               // going to a non-pc+4 target.
               if (irq_pending && !setPCsel
               && (PCsel_last==0) ) begin
                   IFETCHreq_int = 0;
                   PCle = 0;
                   irq_pending = 0;
                   PCsel = 4;
                   WAsel = 1;
                   WDsel = 0;
                   WERF_int = 1;
                   next = IRQ_SENSED;
               end
               else begin
```

```
                        PCsel = 1;
                        IFETCHreq_int = 1;
                        PCle = 1;
                        next = IFETCH;
                    end
                end

                MEMOP: begin
                    MEMOPsel = 1;
                    if (MEMOPready) begin
                            MEMRle = 1;
                            next = WRITEBACK0;
                        end
                        else
                            next = MEMOP;
                     end

                     IRQ_SENSED: begin
                             irq_pending = 0;
                             WERF_int = 0;

                             IFETCHreq_int = 1;
                             PCle = 1;

                             next = IFETCH;
                     end

            endcase
    end
endmodule
```

Omitted: Beta register file (b_regfile.v)

```
/*** b_cpuTestBusClient.v ****************************
        Jacob Kitzman | 6.111 Final Project S04
        beta CPU shared bus client
        handles memory/ifetch requests from the cpu upon
cache misses, issuing memory operations to shared bus
when cpu has bus access.
*****************************************************/

module b_cpuTestBusClient(clk,reset,IFETCHready,
MEMOPready, IFETCHreq, MEMOPreq, CPUbus_en, wr, re,
wdata, address, rdata, bus, BAXVchange, BAXVchgowner,
direct_io);
        input clk, reset;
        input wr;
        input re;
        input[31:0] wdata;
        input[31:0] address;
        output[31:0] rdata;
        reg[31:0] rdata;

        output[3:0] direct_io;
        reg[3:0] direct_io, direct_io_int;

        inout[55:0] bus; // 55: read 54: wr, 53-32:
address, 31-0: data
        reg[31:0] bus_address, bus_address_int,
bus_address_int_last;
        reg[31:0] bus_data, bus_data_int,
bus_data_int_last;
        reg BUSwrite, BUSwrite_int;
        reg BUSread, BUSread_int;

        input CPUbus_en;

        assign bus[55] = CPUbus_en?BUSread:1'bz;
        assign bus[54] = CPUbus_en?BUSwrite:1'bz;
        assign bus[53:32] = CPUbus_en?bus_address:32'bz;
        assign bus[31:0] =
(CPUbus_en&BUSwrite)?bus_data:32'bz;

        input IFETCHreq, MEMOPreq;
        output IFETCHready, MEMOPready;
        reg IFETCHready, MEMOPready;
        reg IFETCHready_int, MEMOPready_int;

        output BAXVchange;
        output[3:0] BAXVchgowner;

        reg[3:0] state, next;
        reg data_back_int, data_back;

        parameter IDLE = 4'h0;
        parameter BUSREADY = 4'h2;
        parameter BUSWAIT_IFT = 4'h3;
        parameter BUSWAIT_MOP = 4'h4;
        parameter BEGINIFETCH = 4'h5;
        parameter BEGINMEMOP = 4'h6;
        parameter IFETCHPENDING = 4'h7;
        parameter MEMOPPENDING = 4'h8;
        parameter MEMOPPENDING1 = 4'h9;
        parameter BUSCHG_STALL1 = 4'hA;
        parameter BUSCHG_STALL0 = 4'hB;

        parameter BAXV_ADDR = 32'h7fffffff;
        parameter DIRECTIO_ADDR = 32'h7ffffff0;

        reg[3:0] mem_stall_cnt_int;
        reg[3:0] mem_stall_cnt;

        reg BAXVchange_int, BAXVchange;
        reg[3:0] BAXVchgowner_int, BAXVchgowner;

        always @(posedge clk or negedge reset) begin
                if (!reset) begin
                        state <= IDLE;

                        IFETCHready <= 0;
                        MEMOPready <= 0;

                        data_back <= 0;

                        BAXVchange <= 0;
                        BAXVchgowner <= 0;

                        bus_address <= 32'd0;
                        bus_data <= 32'b0;
                        BUSwrite <= 0;
                        BUSread <= 0;
                        bus_address_int_last <= 32'd0;
                        bus_data_int_last <= 32'd0;

                        mem_stall_cnt <= 3'd0;

                        direct_io <= 0;
                end
                else begin
                        state <= next;

                        data_back <= data_back_int;

                        BUSread <= BUSread_int;
                        BUSwrite <= BUSwrite_int;

                        bus_address <= bus_address_int;

                        if (BUSwrite_int)
                          bus_data <= bus_data_int;

                        bus_address_int_last <=
                                bus_address_int;
                        bus_data_int_last <=
                                bus_data_int;
                        direct_io <= direct_io_int;
                        BAXVchange <= BAXVchange_int;
                        BAXVchgowner <=
                                BAXVchgowner_int;
                        IFETCHready <= IFETCHready_int;
                        MEMOPready <= MEMOPready_int;
                        mem_stall_cnt <=
                                mem_stall_cnt_int;
                        if (state == IDLE)
                                i <= i_int;
                end
        end

        always @(state or CPUbus_en or IFETCHreq or
MEMOPreq or address or mem_stall_cnt or data_back or bus
or wr or wdata or re or bus_address_int_last or
bus_data_int_last) begin
                next = state;
                IFETCHready_int = 0;
                MEMOPready_int = 0;
                rdata = 32'bX;
                rdata = bus[31:0];
                BAXVchange_int = 0;
```

```
BAXVchgowner_int = 4'd0;

mem_stall_cnt_int = 0;
bus_data_int = bus_data_int_last;
bus_address_int = bus_address_int_last;
data_back_int = 0;

BUSread_int = 0;
BUSwrite_int = 0;
i_int = i;

direct_io_int = 0;

case (state)
 (IDLE): begin
    if (!data_back)
       rdata = 32'dX;
    else  begin
       rdata = bus[31:0];
       data_back_int = 0;
    end

    if (CPUbus_en && IFETCHreq) begin
       BUSread_int = 1;
       bus_address_int = address;
       mem_stall_cnt_int = 3'd0;
       next = BEGINIFETCH;
    end
    else if (CPUbus_en && MEMOPreq)
       if (wr) begin
          if (address==BAXV_ADDR) begin
            BAXVchange_int = 1;
            BAXVchgowner_int =
                  wdata[3:0];
            next = BUSCHG_STALL1;
          end
          else if
          (address==DIRECTIO_ADDR) begin
             direct_io_int = wdata[3:0];
             MEMOPready_int = 1;
             next = IDLE;
          end
       else begin
           mem_stall_cnt_int = 3'd0;
           BUSwrite_int = 1;
           bus_address_int = address;
           bus_data_int = wdata;
           next = BEGINMEMOP;
       end
     end
     else if (re) begin
        mem_stall_cnt_int = 3'd0;
        BUSread_int = 1;
        bus_address_int = address;
        next = BEGINMEMOP;
     end
   else if (CPUbus_en)
      next = BUSREADY;
  else if (!CPUbus_en && IFETCHreq)
      next = BUSWAIT_IFT;
  else if (!CPUbus_en && MEMOPreq)
      next = BUSWAIT_MOP;
end

(BUSREADY): begin
   if (~CPUbus_en)
     next = IDLE;
   else if (IFETCHreq) begin
     BUSread_int = 1;
     bus_address_int = address;
     mem_stall_cnt_int = 3'd0;
     next = BEGINIFETCH;
  end
  else if (MEMOPreq) begin
     if (wr) begin
         if (address==BAXV_ADDR) begin
             BAXVchange_int = 1;
             BAXVchgowner_int =
                   wdata[3:0];
             next = BUSCHG_STALL1;
         end
         else if
         (address==DIRECTIO_ADDR) begin
             direct_io_int = wdata[3:0];
             MEMOPready_int = 1;
             next = IDLE;
         end
         else begin
             mem_stall_cnt_int = 3'd0;
             BUSwrite_int = 1;
             bus_address_int = address;
             bus_data_int = wdata;
             next = BEGINMEMOP;
         end
       end
       else if (re) begin
          mem_stall_cnt_int = 3'd0;
          BUSread_int = 1;
          bus_address_int = address;
          next = BEGINMEMOP;
       end
     end
 end

 (BUSWAIT_IFT): begin
     if (CPUbus_en) begin
        BUSread_int = 1;
        bus_address_int = address;
        mem_stall_cnt_int = 3'd0;
        next = BEGINIFETCH;
    end
 end

 (BUSWAIT_MOP): begin
    if (CPUbus_en) begin
       if (wr && address==BAXV_ADDR) begin
          BAXVchange_int = 1;
          BAXVchgowner_int = wdata[3:0];
          next = BUSCHG_STALL1;
       end
       else if (wr &&
       address==DIRECTIO_ADDR) begin
          direct_io_int = wdata[3:0];
          MEMOPready_int = 1;
          next = IDLE;
      end
      else begin
         if (wr) begin
            mem_stall_cnt_int = 3'd0;
            BUSwrite_int = 1;
            bus_address_int = address;
            bus_data_int = wdata;
            next = BEGINMEMOP;
         end
         else if (re) begin
              mem_stall_cnt_int = 3'd0;
              BUSread_int = 1;
              bus_address_int = address;
              next = BEGINMEMOP;
         end
       end
    end
 end

(BEGINMEMOP): begin
    bus_address_int = bus_address_int_last;
    if (bus[54])
       bus_data_int = bus_data_int_last;

    if (mem_stall_cnt == 3'd1)
       next = MEMOPPENDING;
    else begin
        mem_stall_cnt_int =
             mem_stall_cnt + 1;
        next = BEGINMEMOP;
    end
 end

 (MEMOPPENDING): begin
     MEMOPready_int = 1;
    data_back_int = 1;
    next = IDLE;
 end

 (BEGINIFETCH): begin
    bus_address_int = bus_address_int_last;
    if (mem_stall_cnt == 3'd1)
       next = IFETCHPENDING;
    else begin
       mem_stall_cnt_int =
            mem_stall_cnt + 1;
       next = BEGINIFETCH;
```

```
                end
             end

            (IFETCHPENDING): begin
               IFETCHready_int = 1;
                data_back_int = 1;
                next = IDLE;
            end

            (BUSCHG_STALL1): begin
                if (!CPUbus_en) begin
                  MEMOPready_int = 0;
                  next = BUSCHG_STALL1;
                             end
                             else begin
                                 MEMOPready_int = 0;
                                 next = BUSCHG_STALL0;
                             end
                     end
                    (BUSCHG_STALL0): begin
                       if (!CPUbus_en) begin
                          MEMOPready_int = 0;
                          next = BUSCHG_STALL0;
                       end
                       else begin
                            MEMOPready_int = 1;
                            next = IDLE;
                        end
                    end
                endcase
        end
endmodule
```

Omitted: b_dmcache.v (Beta 512x32 Direct-mapped write-through cache and control logic)

```
/*** sysram_busintf.v ***********************************
    Jacob Kitzman | 6.111 Final Project S04
    Main Memory and Controller using FPGA BlockRAM
    Uses shared bus interface to allow easy subsititution
    For external memory.  Instantiates 56384x32 memory
*****************************************************/

// reads can be unaligned to word boundaries
// writes must be aligned.

module sysram_busintf(clk, reset, bus, vid_addr,
vid_data);
        input clk;
        input reset;
        inout[55:0] bus;
        input[15:0] vid_addr;
        output[31:0] vid_data;

        wire bus_resig = bus[55];
        wire bus_wrsig = bus[54];
        wire[21:0] bus_addr  = bus[53:32];
        wire[31:0] bus_data  = bus[31:0];

        reg mem_wr, mem_wr_int;
        reg mem_rdHi, mem_rdHi_int;

        reg[15:0] mem_addr_rdhigh;
        reg[15:0] mem_addr_wr;

        wire[15:0] ram_addr;
        assign ram_addr =
mem_wr?mem_addr_wr:(mem_rdHi?mem_addr_rdhigh:bus_addr[17:
2]);

        wire[31:0] ram_dout;
       reg[31:0] ram_data;

        sysmemdp main_memory(
                .addra(ram_addr),
                .addrb(vid_addr),
                .clka(clk),
                .clkb(clk),
                .dina(ram_data),
                .douta(ram_dout),
                .doutb(vid_data),
                .wea(mem_wr)
        );

        reg[31:0] read_masked, read_masked_int;
        reg[31:0] read_low;


        reg bus_oen, bus_oen_int;
        assign bus[31:0] = bus_oen ?
          read_masked : 32'bz;

        reg[63:0] read_mask, read_mask_last;
        reg[5:0] read_shift, read_shift_last;

        reg[2:0] state, next;
        parameter IDLE = 3'd0;
        parameter READ_LO_WORD = 3'd1;
        parameter READ_HI_WORD = 3'd2;
        parameter READ_COMPLETE = 3'd3;
        parameter WRITE = 3'd4;
        parameter WRDELAY0 = 3'd5;
        parameter WRDELAY1 = 3'd6;

        always @(posedge clk or negedge reset) begin
          if (!reset) begin
            read_mask_last <= 0;
            read_shift_last <= 0;
            mem_wr <= 0;
            mem_rdHi <= 0;
            read_masked <= 0;
            read_low <= 0;
            state <= IDLE;
          end
          else begin
            if (state==IDLE) begin
              mem_addr_rdhigh <= (bus_addr[17:2])+1;
              mem_addr_wr <= bus_addr[17:2];
              ram_data <= bus_data;
            end

            read_mask_last <= read_mask;
            read_shift_last <= read_shift;

            mem_wr <= mem_wr_int;
            mem_rdHi <= mem_rdHi_int;
            read_masked <= read_masked_int;
            bus_oen <= bus_oen_int;
            if (next == READ_HI_WORD) begin
              read_low <= ram_dout;
            end

            state <= next;
          end
        end

        always @(state or bus_resig or bus_wrsig or
bus_addr or bus_data or read_mask_last,
read_shift_last, read_low, ram_dout) begin

             read_mask = read_mask_last;
            read_shift = read_shift_last;
            mem_wr_int = 0;
            mem_rdHi_int = 0;
            read_masked_int = 0;
            bus_oen_int = 0;

           case (state)
               IDLE: begin
                   if (bus_resig) begin
                      mem_rdHi_int = 1;

                      if (bus_addr[1:0] == 2'd0) begin
                           read_mask =
                             64'hffffffff00000000;
                           read_shift = 32;
                      end
                      else if (bus_addr[1:0] == 2'd1)
                      begin
                           read_mask =
                             64'h00ffffffff000000;
                           read_shift = 24;
                      end
                      else if (bus_addr[1:0] == 2'd2)
                      begin
                        read_mask =
                            64'h0000ffffffff0000;
                        read_shift = 16;
                      end
```

```
                    else if (bus_addr[1:0] == 2'd3)
                    begin
                        read_mask =
                          64'h000000ffffffff00;
                        read_shift = 8;
                    end

                      next = READ_LO_WORD;
                    end
                    else if (bus_wrsig) begin
                        next =  WRITE;
                    end
                 else
                    next = IDLE;
               end

               READ_LO_WORD: begin
                  next = READ_HI_WORD;
               end

               READ_HI_WORD: begin
                 read_masked_int=(({read_low[31:0],
ram_dout[31:0]} & read_mask_last) >> read_shift_last);

                 bus_oen_int = 1;
                 next = IDLE;
               end

               WRITE: begin
                  next = WRDELAY0;
                  mem_wr_int = 1;
               end

               WRDELAY0: begin
                  next = WRDELAY1;
               end

               WRDELAY1: begin
                 next = IDLE;
               end
            endcase
        end
endmodule
```

Omitted: various bus multiplexors, 32-bit register with load enable, and clock interrupt generator

---

Audio AC97 Frame Transmitter:

```
module frameTransmitFSM(clk, reset, syncPulse, bit_clk,
sdata_out, pcm_left, pcm_right, data_request_left,
data_request_right,bit_count, frame_count);

input syncPulse, bit_clk, clk, reset, data_request_left,
data_request_right;
input [7:0] bit_count;
output sdata_out;
reg sdata_out;

reg [23:0] command;
wire [19:0] command_data;
wire [19:0] command_address;

output [14:0] frame_count;
reg [14:0] frame_count;


input [19:0] pcm_left, pcm_right;
reg [19:0] pcm_left_temp, pcm_right_temp;

reg [1:0] state, next;

reg left_valid, right_valid, frame_valid, command_valid;

//reg [7:0] bit_count;
reg syncDone;

parameter IDLE = 0;
parameter ACTION = 1;

/*
what do we care about in the outgoing frame?
PCM data left
PCM data right
left_data_valid
right_data_valid
frame_valid
command
command_address
command_data
*/


always @ (posedge clk) begin
        if (reset) begin
                state <= IDLE;
                //frame_count <= 0; // this is assigned
in 2 always blocks.  Is this OK? NO
        end
        else state <= next;

if (state == ACTION) begin
        //bit_count <= 0;  // reset bit count
        if (bit_count == 0) begin
                //load pcm data from inputs (moved from
@posedge sync)
                if (data_request_left) begin
                        pcm_left_temp <= pcm_left;
                        left_valid <= 1;
                end
                else begin
                        pcm_left_temp <=
20'b0;//shouldn't we leave what was there?
                        left_valid <= 0;
                end

                if (data_request_right) begin
                        pcm_right_temp <= pcm_right;
                        right_valid <= 1;
                end
                else begin
                        pcm_right_temp <= 20'b0;
//shouldn't we leave what was there?
                        right_valid <= 0;
                end

        end


        //bit_count <= bit_count + 1;
end //state == ACTION
else begin //state == IDLE
        //bit_count <= 0;
end //else


end

always @ (state or reset or syncPulse) begin
        case (state)
                IDLE: if (syncPulse) begin
                        next = ACTION;
                        //bit_count = 0; //maybe not a
good idea
                        end
                        else next = IDLE;

                ACTION: if (syncPulse) next = ACTION;
        endcase
end

always @ (posedge bit_clk) begin

        if (reset) begin
                //syncDone <= 0;
                //syncPulse <= 0;
                frame_count <= 0;
        end
        /*
        else if (sync) begin
                if (!syncDone) begin
                        syncPulse <= 1;
                        syncDone  <= 1;
                end
                else syncPulse <= 0;
        end
        else if (syncPulse) begin
                //bit_count <= 0;
```

```
                syncPulse <= 0;
        end
        else if (!sync && syncDone) syncDone <= 0;
        */
        else if (bit_count == 255) frame_count <=
frame_count + 1;

        if ((bit_count >= 0) && (bit_count <= 15))
        // Slot 0: Tags
        case (bit_count[3:0])
          5'h0: sdata_out <= 1; // Frame valid
          5'h1: sdata_out <= 1; // Command address valid
          5'h2: sdata_out <= 1; // Command data valid
          5'h3: if (left_valid) sdata_out <= 1; //left
audio valid
          5'h4: if (right_valid) sdata_out <= 1; //right
audio valid
           default: sdata_out <= 1'b0;
        endcase

       else if ((bit_count >= 16) && (bit_count <= 35))
        // Slot 1: Command address
        sdata_out <= command_address[35-bit_count];

       else if ((bit_count >= 36) && (bit_count <= 55))
        // Slot 2: Command data
        sdata_out <= command_data[55-bit_count]; //maybe
should be 54

        else if (left_valid && (bit_count >= 56 &&
bit_count <= 75)) begin //was 55,74
                //Slot 3: PCM left
                sdata_out <= pcm_left_temp[75-
bit_count]; //this should maybe be 75 //was 74
                end
        else if (right_valid && (bit_count >= 76 &&
bit_count <= 95)) begin
                //Slot 4: get PCM right data
                case(frame_count)
                        20000: sdata_out <= 1;
                        20001: sdata_out <= 0;
                default: sdata_out <= pcm_right_temp[95-
bit_count];
                endcase
        end
        else sdata_out <= 0;


end

always @(frame_count)
     case (frame_count)
     /* commands from rev1
       4'h0: command = 24'h02_0000; // Unmute line
outputs (master volume)
       4'h1: command = 24'h04_0000; // Unmute headphones
       4'h2: command = 24'h10_0808; // Unmute line inputs
       4'h3: command = 24'h2C_AC44; //set sample rate
left = 44100 kHz
       4'h4: command = 24'h32_AC44; //set sample rate
right = 44100 kHz
       4'h5: command = 24'h0A_8000; //mute PC_beep
       4'h6: command = 24'h18_0808; //unmute DAC output,
slightly attenuate.
       4'h7: command = 24'h1C_0000; //unmute record gain
       4'h8: command = 24'h1A_0404; // record select =
line in
       */
       4'h0: command = 24'h02_0000; // Unmute line
outputs (master volume)
       4'h1: command = 24'h04_0000; // Unmute headphones
       4'h2: command = 24'h10_FFFF; // MUTE line inputs
(no analog loopback)
       4'h3: command = 24'h1A_0404;// record select =
line in
       4'h4: command = 24'h1C_0000; //unmute record gain
0dB
       4'h5: command = 24'h18_0808; //unmute DAC output,
slightly attenuate
       4'h6: command = 24'h20_8000; //set bypass 3D
       4'h7: command = 24'h0C_FFFF; // mute line in to
mix 2
       4'h8: command = 24'h0E_FFFF; //MUTE mic  (no
analog)
         //4'h9: command = 24'h2A_0001; //enable
variable sample rate
          4'hA: command = 24'h2C_0FA0; //set sample rate
left = 8 kHz
       4'hB: command = 24'h32_0FA0; //set sample rate
right = 8 kHz
       default: command = 24'hFC_0000; // Read vendor ID
     endcase

   // Separate the address and data portions of the
command
   // and pad them to 20 bits
   assign command_address = {command[23:16], 12'h000};
   assign command_data = {command[15:0], 4'h0};

endmodule
```

Hsync module (from video)

```
module hsync(clk, reset, LineStart, done, R, G, B,
h_sync, h_blank,
                currentLine, address, data, offset,
stop, state, h_sync2, h_blank2,
                currentHLine);
input stop;
input [9:0] currentLine;
input clk, reset, LineStart;
input [15:0] offset;
output [7:0] R, G, B;
output done, h_sync, h_blank, h_sync2, h_blank2;
reg h_sync2, h_blank2;
output[10:0] currentHLine;
reg [10:0] currentHLine;
reg [7:0] R, G, B;
reg done, h_sync, h_blank;
input [31:0] data;
reg [31:0] data_int;
output [15:0] address;
reg [15:0] address, address_int;
reg countv, countven, resetcountv;
// states
parameter INIT   = 0;
parameter idle   = 1;
parameter out11  = 2;
parameter out12  = 3;
parameter out21  = 4;
parameter out22  = 5;
parameter FPHs   = 8;
parameter FPHe   = 9;
parameter HSPs   = 10;
parameter HSPe   = 11;
parameter BPHs   = 12;
parameter BPHe   = 13;
parameter waitstate      = 14;
output [3:0] state;
reg [3:0] state, next;

//internal variables
reg [10:0] count;
reg h_sync_int, h_blank_int, done_int, resetcount,
counten, currentHLineCount;
reg [7:0] R_int, G_int, B_int;

always @ (posedge clk)
begin
        if(!reset)
        begin
                state <= INIT;
        end
        else
        begin
                state <= next;
        end


        if(stop)
                state <= idle;

        if(resetcount)
        begin
                count <= 0;
                currentHLine <=0;
        end
        else if(counten)
                count <= count + 1;

        done = done_int;
```

```
        h_sync <= h_sync_int;
        h_sync2 <= h_sync_int;
        h_blank <= h_blank_int;
        h_blank2 <= h_blank_int;

        R <= R_int;
        G <= G_int;
        B <= B_int;
        //R <= 8'b11111000;
        //G <= 8'b00000000;
        //B <= 8'b00000000;
        address <= address_int;
        if(state == out22)
                data_int <= data;
        if(currentHLineCount)
                currentHLine <= currentHLine + 1;
        else    if(resetcount)
                currentHLine <= 0;
        if(resetcountv)
                countv <= 0;
        else if(countven)
                countv <= countv + 1;
end


always @ (state or LineStart or count)
begin
        done_int = 0;
        resetcount = 0;
        resetcountv = 0;
        h_sync_int = 1;
        h_blank_int = 1;
        counten = 0;
        currentHLineCount = 0;
        countven = 0;

        // jk: avoid inferring latches --->
        R_int = R;
        G_int = G;
        B_int = B;
        address_int = address;
        next = state;
        /// <----

        case(state)
                INIT:
                begin
                        resetcountv = 1;
                        R_int = 8'b0;
                        G_int = 8'b0;
                        B_int = 8'b0;
                        address_int = 0;
                        next = idle;
                end
                idle:
                begin
                        if(LineStart)
                        begin
                                next = FPHs;
                                resetcount = 1;
                                address_int =
currentLine * 160 + offset;
                        end
                        else
                                next = idle;
                end
                out11:
                begin
                        counten = 1;
                        R_int[7:3] = data_int[31:27];
                        G_int[7:3] = data_int[26:22];
                        B_int[7:3] = data_int[21:17];
                        next = out12;
                end
                out12:
                begin
                        counten = 1;
                        R_int[7:3] = data_int[31:27];
                        G_int[7:3] = data_int[26:22];
                        B_int[7:3] = data_int[21:17];
                        next = out21;
                end
                out21:
                begin
                        counten = 1;
                        R_int[7:3] = data_int[15:11];
                        G_int[7:3] = data_int[10:6];
                        B_int[7:3] = data_int[5:1];
                        next = out22;
                        currentHLineCount = 1;
                end
                out22:
                begin
                        R_int[7:3] = data_int[15:11];
                        G_int[7:3] = data_int[10:6];
                        B_int[7:3] = data_int[5:1];
                        if((count >= 639) && (countv ==
1))
                        begin
                                resetcountv = 1;
                                next = idle;
                                done_int = 1;
                                resetcount = 1;
                        end
                        else if(count >= 639)
                        begin
                                countven = 1;
                                done_int = 1;
                                next = FPHs;
                                resetcount = 1;
                        end
                        else
                        begin
                                next = out11;
                                counten = 1;
                                //address_int =
currentLine * 160 + (currentHLine + 1) / 4 + offset;
                                address_int =
(currentLine>>2) * 160 + (currentHLine + 1) + offset;
                        end

                end
                waitstate:
                begin
                        next = out11;
                end
                FPHs:
                begin
                        next = FPHe;
                        h_blank_int = 0;
                        R_int = 0;
                        G_int = 0;
                        B_int = 0;
                        counten = 1;
                end
                FPHe:
                begin
                        h_blank_int = 0;
                        counten = 1;
                        if(count == 16)
                        begin
                                next = HSPs;
                                resetcount = 1;
                        end
                        else
                                next = FPHe;
                end
                HSPs:
                begin
                        h_sync_int = 0;
                        h_blank_int = 0;

                        next = HSPe;
                        counten = 1;
                end
                HSPe:
                begin
                        h_sync_int = 0;
                        h_blank_int = 0;
                        counten = 1;
                        if(count == 96)
                        begin
                                next = BPHs;
                                resetcount = 1;
                        end
                        else
                                next = HSPe;

                end
                BPHs:
                begin
                        next = BPHe;
```

```
                        h_blank_int = 0;
                        counten = 1;
                end
                BPHe:
                begin
                        h_blank_int = 0;
                        counten = 1;
                        if((count == 48) && (countv ==
1))
                        begin
                                next = waitstate;
                                resetcount = 1;
                        end
                        else if(count == 48)
                        begin
```

```
module flashromcontroller (clk, reset, write, readin,
datain, addr, readnow, erase,

        address, outdata, ce_b, oe_b, we_b,
resetflash, sts, dataout,

        addressb, ce_b2, oe_b2, we_b2, resetflash2,
state, testdata, donesignal, next, done2);
input clk, reset, write, readin, readnow, erase;
input [24:0] addr;
input [15:0] datain;
output [15:0] dataout;
output donesignal;
reg [15:0] dataout;
input sts;
inout [15:0] outdata;
output [24:0] address;
output ce_b, oe_b, we_b, resetflash;
output [24:0] addressb;
output ce_b2, oe_b2, we_b2, resetflash2;
wire [15:0] outdata;
output [15:0] testdata;
reg [15:0] testdata;
reg [15:0] storedata;
reg [24:0] address;
reg ce_b, oe_b, we_b, resetflash;
reg [24:0] addressb;
reg ce_b2, oe_b2, we_b2, resetflash2;
reg ce_b_int, oe_b_int, we_b_int;
output done2;
reg done2;
//internal stuff
reg w, w_int, r_int, r, donesignal, donesignal_int,
countreset, counten;
reg [5:0] count;

//States
parameter INIT   = 0;
parameter idle   = 1;
parameter write1 = 2;
parameter write2 = 3;
parameter write3 = 4;
parameter write4 = 5;
parameter write5 = 6;
parameter write6 = 7;
parameter write7 = 8;
parameter write11= 9;
parameter write12= 10;
parameter write13= 11;
parameter write14= 12;
parameter write15= 13;
parameter write16= 14;
parameter write17= 15;
parameter waitlow= 16;
parameter waithigh        = 17;
parameter writedone       = 18;
parameter read            = 19;
parameter readdone        = 20;
parameter blockerase= 21;
parameter erase1          = 22;
parameter        erase2   = 23;
parameter        erase3   = 24;
parameter        erase4   = 25;
parameter        erase5   = 26;
parameter        erase6   = 27;
parameter erase11   = 28;
parameter        erase12  = 29;
parameter        erase13  = 30;
parameter        erase14  = 31;
parameter        erase15  = 32;
```

```
                        next = out11;
                        resetcount = 1;
                end
                else
                        next = BPHe;

        end
endcase


end
endmodule
```

FlashRomController

```
parameter        erase16  = 33;
parameter erase17         = 34;
parameter setupread = 35;
parameter        reada1   = 36;
parameter        reada2   = 37;
parameter        reada3   = 38;
parameter        reada4   = 39;
parameter        reada5   = 40;
parameter        reada6   = 41;
parameter waitlow1        = 42;
parameter waithigh1       = 43;
parameter eraseE = 44;
parameter erase1E= 45;
parameter writeE = 46;
parameter write1E= 47;
parameter readaE = 48;
parameter wordwriteXXX = 49;
parameter XXX1   = 50;
parameter XXX2   = 51;
parameter XXX3 = 52;
parameter writ1 = 53;
parameter writ2 = 54;
parameter writ3 = 55;
parameter writ4 = 56;
parameter writ5 = 57;
parameter writ6 = 58;
parameter writ7 = 59;
parameter writE = 62;
parameter waitlow2 = 61;
parameter waithigh2 = 63;

output [5:0] state, next;
reg [5:0] state, next;

assign outdata = w ? storedata : 16'hz;

always @ (posedge clk)
begin
        if(!reset)
        begin
                state <= INIT;
                resetflash <= 0;
        end
        else
        begin
                state <= next;
                resetflash <= 1;
        end
        done2 <= donesignal;
        donesignal <= donesignal_int;
        testdata <= outdata;
        resetflash <= 1;
        address <= addr;
        ce_b <= ce_b_int;
        oe_b <= oe_b_int;
        we_b <= we_b_int;
        resetflash2 <= 1;
        addressb <= addr;
        ce_b2 <= ce_b_int;
        oe_b2 <= oe_b_int;
        we_b2 <= we_b_int;
        w <= w_int;
        r <= r_int;
        if(state == read)
                dataout <= outdata;
        else dataout <= dataout;
        if((state == write2) || (state == write3) ||
(state == write4) || (state == writeE))
                storedata <= 64;
        else if((state == erase1) || (state ==
erase2) || (state == erase3) || (state == eraseE))
```

```
                storedata <= 32;
        else if((state == erase12) || (state ==
erase13) || (state == erase14) || (state == erase1E))
                storedata <= 208;
        else if((state == reada1) || (state ==
reada2) || (state == reada3) || (state == readaE))
                storedata <= 255;
        else if(r_int)
        begin
                storedata <= outdata;
        end
        else if((state == write12) || (state ==
write13) || (state == write14) || (state == write1E))
        begin
                storedata <= datain;//232;
        end
        else if((state == writ2) || (state == writ3)
|| (state == writ4) || (state == writE))
        begin
                storedata <= 208;
        end
        else if((state == XXX1) || (state == XXX2)
|| (state == XXX3) || (state == wordwriteXXX))
                storedata <= datain;
        else
                storedata <= storedata;

        if(countreset)
                count <= 0;
        else if(counten)
                count <= count + 1;
        else
                count <= count;
end

always @ (state or write or readin or sts or count or
readnow or erase)
begin
        ce_b_int = 1;
        oe_b_int = 1;
        we_b_int = 1;
        counten = 0;
        w_int = 0;
        r_int = 0;
        countreset = 0;
        donesignal_int = 0;
        case(state)
                INIT:
                begin
                        countreset = 0;
                        next = idle;
                end
                idle:
                begin
                        if(~write)
                                next = write1;
                                //next = write11;
                        else if(~readin)
                                next = setupread;
                        else if(~readnow)
                        begin
                                next = read;
                                countreset = 1;
                        end
                        else if(~erase) begin
                           next = blockerase;
                        end
                        else next = idle;
                end
                blockerase:
                begin
                        ce_b_int = 0;
                        next = erase1;
                end
                erase1:
                begin
                        ce_b_int = 0;
                        we_b_int = 0;
                        next = erase2;
                end
                erase2:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = erase3;
end
erase3:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = eraseE;
end
eraseE:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = erase4;
end
erase4:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = erase5;
end
erase5:
begin
        next = erase6;
end
erase6:
begin
        next = erase11;
end
erase11:
begin
        ce_b_int = 0;
        next = erase12;
end
erase12:
begin
        ce_b_int = 0;
        we_b_int = 0;
        next = erase13;
end
erase13:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = erase1E;
end
erase1E:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = erase14;
end
erase14:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = erase15;
end
erase15:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = erase16;
end
erase16:
begin
        next = erase17;
end
erase17:
begin
        next = waitlow1;
```

```
end
waitlow1:
begin
        if(sts) next = waitlow1;
        else next = waithigh1;
end
waithigh1:
begin
        if(sts) next = writedone;
        else next = waithigh1;
end
write1:
begin
        ce_b_int = 0;
        next = write2;
end
write2:
begin
        ce_b_int = 0;
        we_b_int = 0;
        next = write3;
end
write3:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = writeE;
end
writeE:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = write4;
end
write4:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = write5;
end
write5:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = write6;
end
write6:
begin
        next = write7;
end
write7:
begin
        next = write11;
end
write11:
begin
        ce_b_int = 0;
        next = write12;
end
write12:
begin
        ce_b_int = 0;
        we_b_int = 0;
        next = write13;
end
write13:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = write1E;
end
write1E:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = write14;
end
write14:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = write15;
end
write15:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = write16;
end
write16:
begin
        next = write17;
end
write17:
begin
        next = waitlow;
end
waitlow:
begin
     if(sts)
        next = waitlow;//write11;
     else
        next = waithigh;
end
waithigh:
begin
        if(sts)
                next =
writedone;//wordwriteXXX;
        else
                next = waithigh;
end
writedone:
begin
        next = idle;
        donesignal_int = 1;
end


wordwriteXXX:
begin
        countreset = 1;
        next = XXX1;
        w_int = 1;
end
XXX1:
begin
        counten = 1;
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = XXX2;
end
XXX2:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = XXX3;
end
XXX3:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        if(count == 32)
                next = writ1;
        else    if(~write)
                next = XXX1;
        else
                next = XXX3;
end
writ1:
begin
```

```
        w_int = 1;
        ce_b_int = 0;
        next = writ2;
end
writ2:
begin
        w_int = 1;
        ce_b_int = 0;
        we_b_int = 0;
        next = writ3;
end
writ3:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = writE;
end
writE:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = writ4;
end
writ4:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = writ5;
end
writ5:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = writ6;
end
writ6:
begin
        next = writ7;
end
writ7:
begin
        next = waitlow2;
end
waitlow2:
begin
        if(sts)
                next = waitlow2;
        else
                next = waithigh2;
end
waithigh2:
begin
        if(sts)
                next = writedone;
        else
                next = waithigh2;
end

setupread: begin
        ce_b_int = 0;
        next = reada1;
end
reada1:
begin
        ce_b_int = 0;
        we_b_int = 0;
        next = reada2;
end
reada2:
begin
        ce_b_int = 0;
        we_b_int = 0;
        w_int = 1;
        next = readaE;
end
readaE:
begin
                ce_b_int = 0;
                we_b_int = 0;
                w_int = 1;
                next = reada3;
        end
        reada3:
        begin
                ce_b_int = 0;
                we_b_int = 0;
                w_int = 1;
                next = reada4;
        end
        reada4:
        begin
                ce_b_int = 0;
                we_b_int = 0;
                w_int = 1;
                next = reada5;
        end
        reada5:
        begin
                next = reada6;
        end
        reada6:
        begin
                next = idle;

        end

        read:
        begin
                countreset = 0;
                ce_b_int = 0;
                oe_b_int = 0;
                counten = 1;
                if(count == 15)
                        next = readdone;
                else
                        next = read;
        end
        readdone:
        begin
                next = idle;
                r_int = 1;
                donesignal_int = 1;
        end
        endcase
end

endmodule
```

Game Software Code:

```
kernel.uasm (adapted from kernel.uasm ©1994 Steve
Ward)
. = VEC_RESET
    BR(I_Reset) | on Reset (start-up)
. = VEC_CLK
    BR(I_Clk)   | On clock interrupt
. = KERN_START

UserMState:
STORAGE(32) | R0-R31... (PC is in XP!)
.macro SS(R) ST(R, UserMState+(4*R))    | (Auxiliary
macro)
.macro SAVESTATE_8() {SS(0)  SS(1)  SS(2)  SS(3)
SS(4)  SS(5)  SS(6)  SS(7) SS(30) }
.macro RESTORESTATE_8() {RS(0)  RS(1)  RS(2)  RS(3)
RS(4)  RS(5)  RS(6)  RS(7) RS(30) }

KStack: LONG(.+4)       | Pointer to ...
    STORAGE(256)        |  ... the kernel stack.

I_Clk:
    SAVESTATE_8()
    CMOVE(CLK_TICKS_CUR_l, r0)
    CMOVE(0x3, r2)
    LD(r0, 0, r1)
    ADDC(r1, 1, r1)
    ST(r1, 0, r0)
    LD(r0, 4, r1)
    ADDC(r1, 1, r1)
```

```
    CMPEQ(r1, r2, r3)| have we counted up to 2.7e+6
cycles?
    ST(r1, 4, r0)                   | save count
    BF(r3, I_Clk_ret)               | no, return...
    ADD(r31, r31, r1)               | clear io count
    ST(r1, 4, r0)                   | save count
         XOR(r31, r31, r4)
         SHRC(r4, 1, r4)            | construct 0x7fff
ffff in R4

         ADDC(r31, 3, r1)

         ST(r1, 0, r4)                      | write
baxv - yield to IO
I_Clk_ret:
    RESTORESTATE_8()
    JMP(XP)

I_Reset:
    CMOVE(0x0800, XP)    | program start place.
    JMP(XP)
```

```
/*** ddr.c ******************************************
   Jacob Kitzman | 6.111 Final Project S04
    DDR video game and support software, for the Beta
    Platform
************************************************/
 #define FRAMEBUFFER 0x000117f0L
#define SCREEN_WIDTH 320
#define SCREEN_HEIGHT 240
#define TICK_COUNT 0x00000700
#define CORRECT_STEPS 0x00006000
#define N_CORR_STEPS 10
#define DIRECT_IO 0x7ffffff0
#define STEPBUF 0x00006000

// takes in chars, left-aligned 8-bit color values
#define MK2PIXEL(R1, G1, B1, R2, G2, B2)
((((int)R1<<24)&0xf8000000)|(((int)G1<<19)&0x07C00000
)|(((int)B1<<14)&0x003f0000)) |
((((int)R2<<8)&0x0000f800)|(((int)G1<<3)&0x000007C0)|
(((int)B1>>2)&0x0000003f))
// (omitted SPRITE start and end location #define's

// bitblt a sprite into the framebuffer
void copyrgn_entirew(
        int *src,        // base of source sprite
        int x,
        int y,
        int src_w,       // width of source glyph
        int src_h)  // height of source glyph
{
        int iSrc=0;
        int i_x, i_y;

        for (i_y=0;i_y<src_h>>1;i_y++)
        {
                int *FB_BASE =
(int*)(FRAMEBUFFER+((x+((y+i_y)*320))<<1));

                iSrc = i_y*src_w;

                for (i_x=0;i_x<(src_w>>1);i_x++)
                {
                        int *pCurDest =
(int*)(FB_BASE+i_x);
                        *pCurDest =
*((int*)src+iSrc);
//                      *pCurDest =
*((int*)src+iSrc)|x|(y<<16); //DEBUG ONLY, see what
position were copying…
                        iSrc++;
                }
        }
}
/* bitblt a sprite into the framebuffer, using the
color of the sprite's upper left-hand pixel as the
transparency value */
void copyrgn_entirew_transp(
        int *src,        // base of source sprite
        int x,
        int y,
        int src_w,       // width of source glyph
        int src_h,  // height of source glyph
        int maskColor) // color to mask through to
black
{
        int iSrc=0;
        int i_x, i_y;

        int transp_col = (*src);
        transp_col=(transp_col>>16)&0x0000ffff;

        for (i_y=0;i_y<src_h>>1;i_y++)
        {
                int *FB_BASE =
(int*)(FRAMEBUFFER+((x+((y+i_y)*320))<<1));

                iSrc = i_y*src_w;

                for (i_x=0;i_x<(src_w>>1);i_x++)
                {
                        int *pCurDest =
(int*)(FB_BASE+i_x);
                        int destCur = *pCurDest;
                        int src =
*((int*)src+iSrc);

        // high order pixel, mask dest through
        if (((src>>16)&0x0000ffff) == transp_col)
          src = destCur&0xffff0000 | src&0x0000ffff;
        else if (((src>>16)&0x0000ffff) ==
0x00000000)
           src = maskColor&0xffff0000 |
src&0x0000ffff;
           // same for low-order pixel
           if (((src)&0x0000ffff) == transp_col)
             src = destCur&0x0000ffff |
src&0xffff0000;
          else if (((src)&0x0000ffff) == 0x00000000)
        src = maskColor&0x0000ffff | src&0xffff0000;
        *pCurDest = src;
                        iSrc++;
                }
        }
}

// omitted const int ltr_offets[26] and
// num_offets[10] (simply start addrs)
// omitted const int ltr_widths[26] and
// num_widths[10] (simply sprite widths in pixels)

void itoh(int num, char* pstr)
{
        int i;
        int j=0;
        for (i=28;i>=0;i-=4)
        {
                int val = (num >> i)&0x000000ff;
                if (val < 10)
                        pstr[j++]=((char)'0'+val);
                else
                        pstr[j++]=((char)'A'+val);
        }

        pstr[j] = (char)0;
}

// Display a character string on the screen by
copying text sprites into the frame buffer
//#define _DBG_DISPLAY_STR 1
void displayStr(int x,
                int y,
                char *pzStrDisp)
{
#ifdef _DBG_DISPLAY_STR
        register int disp_char asm("%R21");
        #endif

        int si=0;
        #ifdef _DBG_DISPLAY_STR
        register int x_acc asm("%R22");
        x_acc = x;
        #else
        int x_acc = x;
        #endif

        char c = pzStrDisp[si];
        while (c!=(char)0)
        {
          if (c>='a'&&c<='z')     // convert to
uppercase.
           c=(char)(c-32);
           if (c>='A' && c<='Z') {
               #ifdef _DBG_DISPLAY_STR
               // debugging approach - marker and
copy to a register we can follow.
                /*
                __asm__ __volatile__ (
                  "ADD(%1,%%R31,%%R4) | copy
        character over
                     ADD(%%R4,%%R31,%0)"
                   :"=r"(disp_char)   //output
```

```
                 :"r"(x_acc)    //input
                 :"%R4"  //clobber list
                );

                // this one additionally causes bsim
to pause.
                 __asm__ __volatile__ (
                  "ADD(%1,%%R31,%%R4) | copy
character over
                    SHLC(%%R4, 10, %%R5)
                    ST(%%R31, 0, %%R5) | generate
bsim breakpoint
                    ADD(%%R4,%%R31,%0)"
                 :"=r"(disp_char)   //output
                 :"r"(c)    //input
                 :"%R4", "%R5"    //clobber list
                ); */
               #endif

            copyrgn_entirew((int*)ltr_offsets[c-'A'],
x_acc, y, ltr_widths[c-'A'], LTR_SPRITE_HEIGHT);

               x_acc += (ltr_widths[c-'A'] + 2);
            }
            else if (c>='0'&&c<='9') {
               copyrgn_entirew((int*)num_offsets[c-
'0'], x_acc, y, num_widths[c-'0'],LTR_SPRITE_HEIGHT);
               x_acc += (num_widths[c-'0'] + 2);
            }

            c = pzStrDisp[++si];
        }
}

//#define _DBG_HLINE 1

__inline__ void hline(int x_start,int x_end, int y,
char r, char g, char b)
{
        int twopix = MK2PIXEL(r,g,b,r,g,b);

        #ifdef _DBG_HLINE
        register int _dbg_ptr asm("%R20");
        #endif

        int y_ofs = y*320;
        int i_x;
        for (i_x=x_start;i_x<x_end;i_x+=2)
        {
                int* dispPtr =
(int*)(FRAMEBUFFER+((i_x+y_ofs)<<1));

                #ifdef _DBG_HLINE
                _dbg_ptr = (int)dispPtr;
                __asm__ __volatile__ (
                        "ADD(%1,%%R31,%%R4)
                         ADD(%%R4,%%R31,%0) | copy
over disp ptr"
                        :"=r"(_dbg_ptr)   //output
                        :"r"((int)dispPtr)
//input
                        :"%R4"    //clobber list
                );
                #endif

                *dispPtr = twopix;
        }
}

void fillrgn(int x,
             int y,
             int fill_w,
             int fill_h,
             int col2pix)
{
        int i_x, i_y;

        for (i_y=0;i_y<fill_h>>1;i_y++)
        {
                int *FB_BASE =
(int*)(FRAMEBUFFER+((x+((y+i_y)*320))<<1));

                for (i_x=0;i_x<(fill_w>>1);i_x++)
                {
                        int *pCurDest =
(int*)(FB_BASE+i_x);
                        *pCurDest = col2pix;
                }

        }
}

void fillscrn(int *dest,
                              int twopix)
{
        int ofs = 0;
        while (ofs < 0x9600)
                *(dest+(ofs++)) = twopix;
}

void fillrgn_g(int x,   int y,
int fill_w,   int fill_h, int r,
int g,      int b) {
        int i_x, i_y;

        for (i_y=0;i_y<fill_h>>1;i_y++)
        {
                int *FB_BASE =
(int*)(FRAMEBUFFER+((x+((y+i_y)*320))<<1));

                int color;
                color = MK2PIXEL(r,g,b,r,g,b);

                r++;
                b++;
                g--;

                for (i_x=0;i_x<(fill_w>>1);i_x++)
                {
                        int *pCurDest =
(int*)(FB_BASE+i_x);
                        *pCurDest = color;
                }

        }
}

// return (roughly) a/b
int div(int a, int b)
{
    if (b>a) return 0;
    else {
        int acc=0;
        int res=0;
        while (acc<a) {
            acc+=b;
            res++;
        }
        return res;
    }
}

// TEST4 - Fill the screen, the display a string.
THIS IS WHAT WE DEMOED
void system_test_4()
{
        char r, g, b;
        r=g=0;
        b=0x40;
        int i=0;

        char *msg = "pain";
        char *msg2 = "6111";


        displayStr(10, 50, msg);
        displayStr(50,200, msg2);

        while (1)
        {
                r++; g--;
                fillrgn_g(250, 150, 60, 70, r, g,
b);
        }
}

int main() {
        *((int*)DIRECT_IO) = 0x0000000c;
        // signal for audio to start.
        system_test_4();

        while (1) {
          __asm__ __volatile__ (
           "ADD(%%R4,%%R31,%%R4) | do nothing -
program done - prevent jump back to krnl."
            :   // output
            :    // input
            :"%R4"     // clobber
            );
        }
}
```