Laser Pinball
Final Report

MIT 6.111

Weston Braun, Jake Isenhart, and Pauline Varley

December 10, 2014

Professor: Gim P. Hom
Term: Fall 2014

Contents

1 Introduction **3**

2 Summary **3**

3 Modules **4**
- 3.1 Vision Processing and Image Recognition . . . 4
  - 3.1.1 The OV7670 Camera . . . 4
  - 3.1.2 Image Recognition . . . 5
- 3.2 Physics and Game Engine . . . 6
  - 3.2.1 The SNES Controller . . . 8
- 3.3 Laser Controller . . . 8
  - 3.3.1 Galvanometers and DACs . . . 10
  - 3.3.2 Testing . . . 11

4 Implementation Process **11**
- 4.1 The μBeta Processor . . . 11
- 4.2 Debugging the OV7670 Camera . . . 12
- 4.3 Building a Laser Projector . . . 12
- 4.4 Xilinx ML505 FPGA Board . . . 13

5 Review and Recommendation **14**

6 Conclusion **15**

Glossary **16**

7 Code and Schematics **17**
- 7.1 Galvo DAC Schematic . . . 17
- 7.2 Full Block Diagram . . . 18
- 7.3 The Beta . . . 19
  - 7.3.1 The Two-Stage Pipelined Beta . . . 19
  - 7.3.2 Beta Address Decoding . . . 24
  - 7.3.3 Memory-Mapped I/O . . . 25
  - 7.3.4 Beta Instruction Set . . . 28
  - 7.3.5 Laser Controller Assembly . . . 34
  - 7.3.6 Tested Physics Engine for Checkoff: No implemented physics, paddle control only . . . 41
  - 7.3.7 Physics Engine Untested Working Copy . . . 47
- 7.4 Camera Module . . . 57
  - 7.4.1 Find red, green, and blue loci in a frame . . . 62
  - 7.4.2 Create signals for VGA IC . . . 65
  - 7.4.3 VGA Drive for Lower Resolution . . . 65
  - 7.4.4 I2C Setup for VGA . . . 66
  - 7.4.5 Camera Configuration . . . 68
  - 7.4.6 Capture a frame from the camera and dump into memory . . . 69
  - 7.4.7 Write object locations to the physics Beta . . . 71
  - 7.4.8 Pixel cost function . . . 72
- 7.5 SNES Controller Interface . . . 73
- 7.6 Pulling it all together . . . 78

1 Introduction

For our final project, we wanted to create a pinball-like arcade game on an FPGA that would be displayed on a wall by an RGB laser projector. The system would allow the user to reconfigure the gameboard by recognizing colored objects on a wall; each color would correspond to a different game object, and the objects would appear in-game in the location in which they were placed on the wall. This project was motivated by a desire to use technology for enjoyment rather than academics: while discussing what we might want to build for our final project we found that, over the course of our MIT education, we hadn't had much of an opportunity to just play around with tech in the way we do outside of class. We wanted a project that we could get excited about and a project that we could share with our peers without too much explanation—a laser-projected game is, undoubtedly, very cool and fits that criteria to a tee.

In our implementation of the proposed design we encountered a series of unexpected difficulties that prevented us from finishing all planned elements of the project. Interfacing with the camera module was particularly difficult (see sections 3.1.1 and 4.2), and implementing a physics engine in Verilog so complex that we decided to scrap the initial architecture entirely and build the project on Beta processors that we implemented using Verilog (section 4.1). These challenges will be discussed further in section 4.

We did, however, accomplish several exciting things: design and implementation of a "Beta" processor; functional image recognition that can detect red, green, and blue objects; physics emulation written entirely in assembly; and a laser-projected gameboard with paddles that can be controlled with an NES controller. These successes will be further discussed in sections 3 and 4.

2 Summary

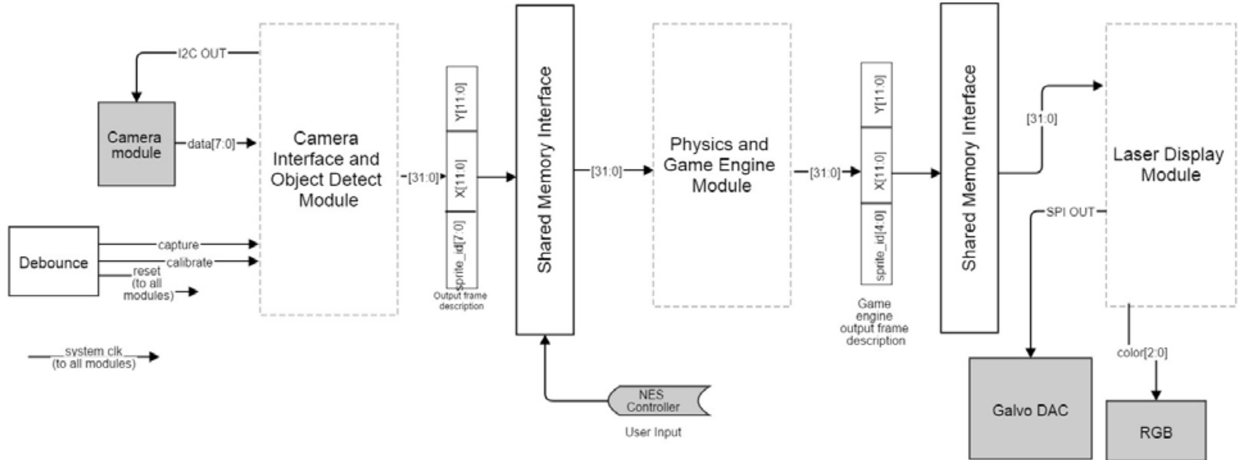


Figure 1: High-level system diagram. Each module will be explained in greater depth later in this report, and a full system diagram is included in the appendix.

Laser pinball was implemented as three discrete systems: a vision processing and object detect module, a physics and game engine, and a laser display controller.[1] Though we originally planned on implementing the system entirely in Verilog, we realized about halfway through that the complexity of both the physics engine

[1]Weston was responsible for the vision processing module; Jake and Pauline were primarily responsible for the physics engine and laser controller, respectively, but collaborated heavily on both. Weston also designed the external hardware interface and brought up the Beta processors.

and the laser controller would be better suited to implementation on a microcontroller with a higher level of abstraction from the hardware. Since a Verilog implementation of the Beta processor already existed (to some degree–the bringup of the Beta will be discussed further in section 4.1), we decided to restructure both modules as assembly language code running on two separate Beta processors communicating via a shared memory interface. Despite this, the overall architecture of the system remained much the same.

The vision processing module takes input from an OV7670 camera, detects objects of three colors (one red, one green, and one blue), and outputs the corresponding sprite IDs and locations to the physics Beta processor over a shared memory interface. The physics engine detects collisions between the pinball and game objects and adjusts its position and velocity according to the game's internal physics. Finally, the laser display module plots sprites at arbitrary locations received from the physics module using a laser projector.

3 Modules

3.1 Vision Processing and Image Recognition

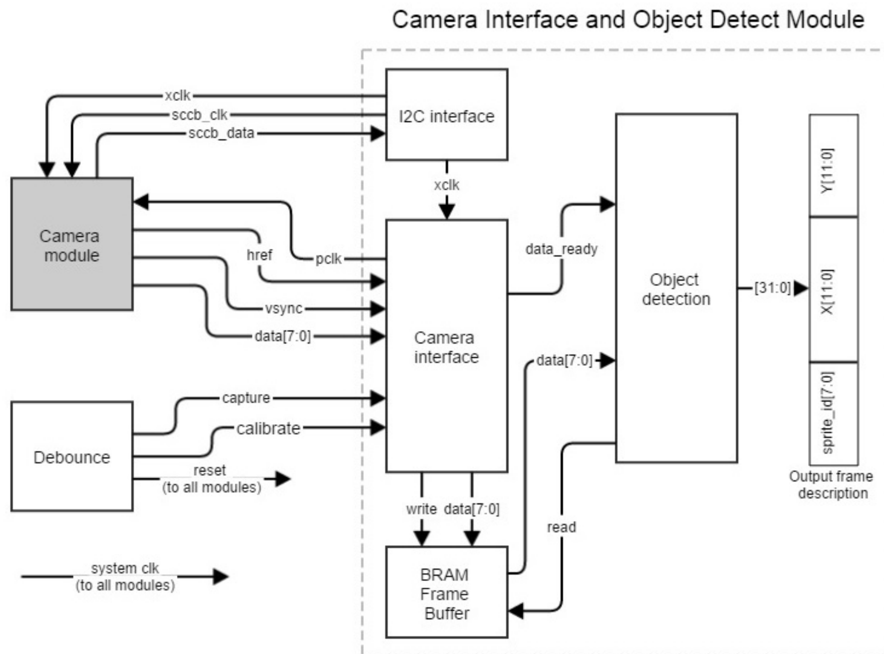


Figure 2: The camera interface module.

3.1.1 The OV7670 Camera

The OV7670 camera was chosen because of its low cost and small size, allowing it to be mounted inside the laser projector case. The camera has an 8-bit output data bus, vertical (VSYNC) and horizontal (HREF) sync pins, and an output clock, and outputs 16-bit color over a 640x480 frame. The camera's output clock is internally generated by the camera from an input clock. This output clock is the same frequency as the input clock, but with different phasing. This required the crossing of clock domains within the FPGA. The camera outputs 8-bit wide sections of a 16-bit wide pixel word at twice the pixel clock frequency. An FSM was written in Verilog to synchronize the data capture of the camera with the start of the frame in order to

capture a complete image with the correct word alignment. The frame starts on the falling edge of VSYNC and each line starts on the falling edge of HREF.

Data captured from the camera was scaled down to 9-bit color and 240x240 pixels and stored in a dual port memory, which acted as a buffer to allow the crossing of clock domains. The down-sampling of color and resolution were due to constraints on the amount of BRAM within the FPGA and 9-bit resolution was used as the BRAM blocks have an input width of 9 bits: a byte plus a 9th bit storing a parity check value. We used this 9th bit to store additional color data.

Unfortunately, the OV7670 camera has a quite complicated setup routine. The camera by default outputs YCrCb data instead of RGB. Additionally, the color balance is quite off to the point of red and green showing up reversed. The OV7670 camera contains a configuration interface called SCCB, which is essentially a royalty-free clone of I2C and is I2C compatible. An open-source I2C core from the OpenCores project and licensed under the BSD license was used for I2C configuration. The core uses a Wishbone interface, which is an open source parallel bus standard. A Verilog FSM wrapper was written for the core to handle the Wishbone interface and an additional FSM was created to send configuration commands to the OV7670 camera from a ROM.

After image capture, the stored image was output to VGA using the Chrontel CH7301C DVI IC on the FPGA development board. This IC also required configuration over an I2C interface to be put in VGA pass-through mode. A version of the FSM used for camera configuration was reused to configure the Chrontel chip. 24-bit color is sent to the CH7301C through a 12-bit interface operating at twice the pixel clock by sending 12 bits at a time on each rising clock. A modified version of the VGA drive module from 6.111 lab 3 was taken to generate hsync and vsync signals for a 640x480 image at the faster pixel clock required by the CH7301C and to generate a pixel address to recall the stored image. A Verilog module was written to take 24-bit pixels and drive the 12-bit bus of the CH7301C. Image quality was poor for reasons which will be discussed further in section 4.2.

Figure 3: Camera output when an attempt was made to put it into RGB output mode.

Figure 4: Camera output with the default settings.

3.1.2 Image Recognition

The goal of image recognition for the project was to recognize red, green, and blue game piece objects on a white background. As the game piece objects were to be small compared to the frame and were all primary colors, several simplifications could be made to the image processing process.

First, instead of calculating the center of mass of a color, the image could be segmented into small sections, and a color classifier could be applied for each color that needed to be detected. The section of the image that had the the highest value for a given classifier was then detected as the center of a color block.

The second simplification that could be made was in the classification of colors. Traditionally, the image is converted to the HSV color-space for classification. However, this process is computationally complex as it is not a linear mapping and requires the use of division, which is difficult to implement on an FPGA. Due to the low color resolution of the stored image and the need to only detect primary colors, a classifier was created that would directly translate an RGB color value into a set of color classifiers for red, green, and blue. This classifier also rejects mixtures of colors, such as white.

The classifier that was decided on for image recognition for each primary color was the value of the desired color minus the value of one of the other colors, multiplied by its value minus the other color. For red, this classifier would be $(red - green) \cdot (red - blue)$. This classifier worked well in distinguishing the colored game objects on a white wall. For processing, the image was split into 3600 squares of 16 pixels each. The 16 pixels in each block were averaged to remove noise and then run through the classifier. The three blocks that had the highest classifier values for red, green, and blue respectively were output as the coordinates of the detected objects.

Image processing was implemented using a Verilog FSM that was triggered by a button on the FPGA board. Data was acquired from the read port of the dual port ram that the camera data was stored in, allowing the user to press a button to see live video from the camera, release the button to capture the frame, and then press a second button to analyze that particular frame. Once the frame was analyzed, a sprite was overlaid onto the stored image at the location of three detected color objects so that the user could confirm that the correct objects were recognized.

Additionally, the location of each object was written into the memory space of the physics engine beta through the shared memory interface along with a done flag.

3.2 Physics and Game Engine

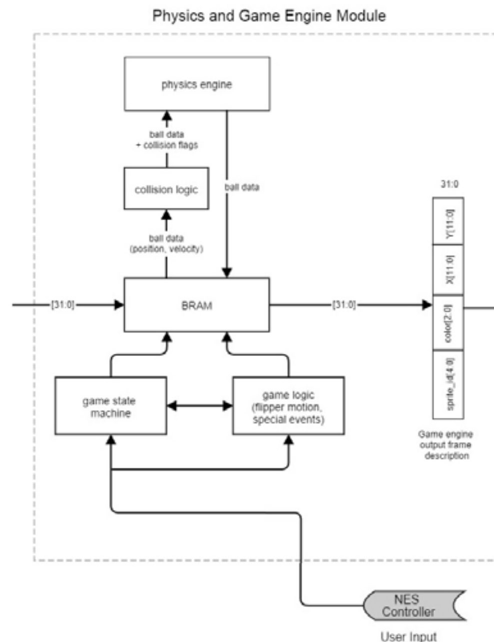


Figure 5: The physics and game engine module.

Initially, it was our intention to implement the entirety of the project in Verilog. Unfortunately, we had incorrectly assumed that the game logic would be almost an afterthought—how hard could it be to decide whether or not two objects are touching each other? Some cursory research on the subject of game engines informed us that it is, in fact, a very high- level mathematical problem.

Unlike in the game of pong we put together in lab, we didn't have the luxury of calculating our collisions using fixed logic. Our plan was to have a game board which could be reconfigured without writing and compiling a new Verilog module. It became apparent in very short order that this wasn't going to be possible with fixed logic. In addition, we hoped to make our physics simulation more complex than that of the pong game; we wanted friction, gravity, and collisions with differing elasticities. In the pong game, whenever our ball hit an object, it was simple to calculate the resulting trajectory and velocity of the ball, because all of the collisions were perfectly elastic and took place with a surface perfectly aligned on either the x or y axis. In these simple cases, negating either the x or y velocity is enough to simulate a realistic collision. Collision detection with angled surfaces (like the triangular bumpers of our gameboard) requires vector math which would be prohibitively complicated to implement as a state machine. In addition, we wanted to have a variable number of game objects—the more we discussed what we needed in order to properly build our dream pinball game, the more we realized we were essentially describing a processor.

With the realization that we wanted to be able to define parts of our project purely in software, we decided to look into the Verilog implementations of the 6.004 Beta core available on past versions of the 6.111 website. The Beta fit our specifications well. Its 32-bit architecture was adequate for doing complex math in the physics module, as well as precisely controlling the laser galvos in the projection module. In addition, it was free, and all of us had experience writing assembly code for it from 6.004. We also considered using other free cores like Xilinx PicoBlaze, but were discouraged by its 8-bit architecture, which would not afford us the precision to do accurate physics modeling, or even make full use of the 12-bit DACs we used to drive the laser galvos. The Beta architecture used for the project will be discussed in more detail in section 4.1.

While designing the physics engine, our hope was to keep the system as simple and as modular as possible. Using lookup tables, we divided the code in to loops intended to iterate over the list of existing sprite objects on the current game board and execute each object's personal "update" function. We wrote routines to take input from the camera and instantiate different sprite objects in memory, so that the list could be repopulated whenever we started a new game. While the communication between the camera and the physics module was never used for anything due to time constraints, the camera and Beta communicate through the exact same model of two-port shared memory as the Beta and the laser projector do. As such, we have every reason to believe that this functionality could be implemented with relative ease, following some further calibration of the camera module.

As the design of the lower-level functional blocks of our system were more imperative to producing a viable demo, the actual functions intended to define the physics of the game were not successfully completed. Due to the complex math involved, coding a physics engine in assembly proved to be a highly difficult task—so much so that we spent a fairly significant amount of time early in the testing stages attempting to get the old 6.004 C to Beta assembly compilers working. Unfortunately, the compilers have not been maintained in recent years, and updates to Athena have rendered them impossible to work with.

By far the most complicated part of implementing a physics engine turned out to be collision detection. Detecting a collision with an angled surface requires several numerically demanding steps of vector multiplication which we were ultimately unable to implement in time. Essentially, collision detection boils down to calculating the minimum distance between the center of the pinball the all of the other line segments and corners on the board. We successfully built functions to handle acceleration due to gravity and position updates based on velocity, but the more complex physics functions have yet to be debugged. With perhaps a few more days worth of testing, we would certainly have a playable game. Perhaps in the future we'll be able to further debug this module and run a playable pinball game on the board.

3.2.1 The SNES Controller

Though we had originally wanted to use accelerometers to control the pinball game's paddles, we instead decided to use an SNES controller because it was far simpler to read. The SNES controller has a three wire interface that allows the output state of all the pins to be shifted out. A latch signal is applied to one controller pin and shortly after a clock is applied and a 16 bit word is shifted out containing the state of each pin. An FSM was created in Verilog to continuously sample the controller and output the pin states over a parallel bus. This was connected to of the Beta's memory-mapped input ports, allowing us to read digital values from the controller.

3.3 Laser Controller

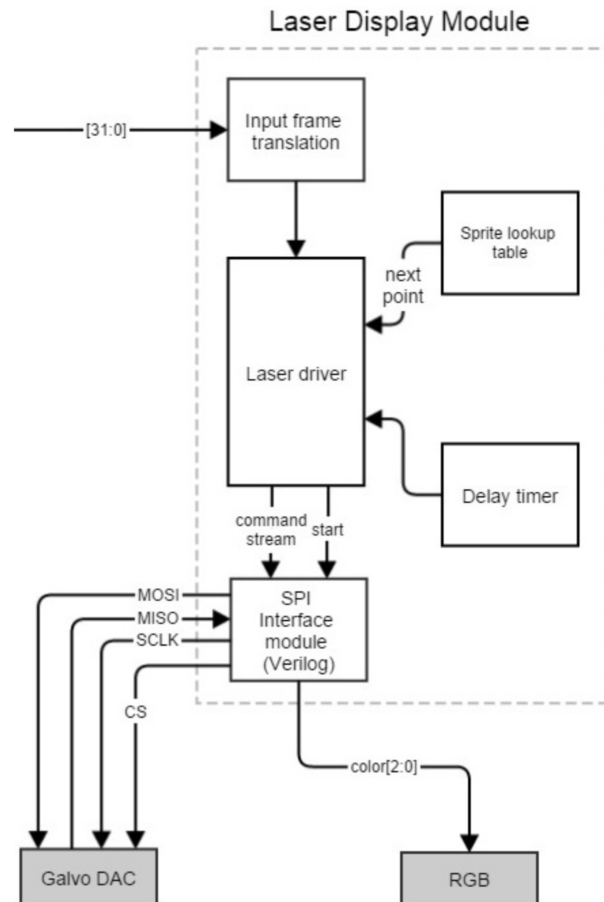


Figure 6: The block diagram for the laser display module. Unless noted, all blocks were designed in Beta assembly language.

The laser display module was implemented primarily on a Beta processor, with the exception of the SPI interface module, which was implemented in Verilog as a simple state machine. The SPI module was then treated as a memory-mapped I/O device accessible by the laser Beta. Each frame, the laser Beta received form the physics Beta a series of one-word (32-bit) sprite descriptions detailing the ID of the sprite to be

drawn, its color, and its x and y location within the frame.[2] A frame was known to be over if a null sprite ID was received; the physics and laser Betas could then easily coordinate frames by jumping back to the beginning of the shared memory space at the end of each frame, so first sprite in a frame was always stored at and read from shared memory location zero.

For each sprite received, the laser controller first registered each part of the input word (sprite ID, color, x location, and y location) into separate registers. Sprite ID was then translated into an offset into the laser Beta's local sprite lookup table, a portion of which is shown below:

```
. = sprite lookup+0x300
| the frame outline: 3
LONG(26)
LONG(0x00000000), LONG(TRAVEL_TIME)      | stall for travel time
LONG(0x00000000), LONG(STALL_TIME)       | stall for laser on
LONG(0x00500000), LONG(0x20)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x00000030), LONG(0x20)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFD80018), LONG(0x8)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x00280018), LONG(0x8)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x00000030), LONG(0x12)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFBA0012), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFEC0000), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFBAFFEE), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x0000FFD0), LONG(0x12)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x0028FFE8), LONG(0x8)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFD8FFE8), LONG(0x8)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x0000FFD0), LONG(0x20)
LONG(0x00000000), LONG(STALL_TIME)       | stall for laser off
```

Figure 7: The frame outline entry from the laser Beta's sprite lookup table.

Each sprite was broken up into line segments, and each line segment was broken up into sub-segments in order to not overdrive the galvos (which could only be driven a small distance at a time). The first item of each sprite (here "LONG(26)") is the number of line segments in that sprite. This was used as a countdown so the laser Beta knew when it had reached the end of the sprite. The numbers in the left column describe x

[2]Location was defined as the point in the upper left corner of the sprite, and sprites were drawn clockwise from this point.

(most significant two bytes) and y (least significant two bytes) offsets, and the numbers in the right column are the number of times the corresponding offset is repeated. These offsets describe the length and direction of each sub-segment.

To draw the sprite, each offset in the table was split into its x and y components, sign-extended, and added to the initial x and y location received from the physics Beta; the initial location, which was registered, was then overwritten to store the current location. This new x-y point was then written to the galvos via the SPI module. A timer was set at 20kHz and the program delayed sending a new value to the galvos until the timer flag went high; during this period of time, the laser Beta polled two reserved shared memory locations for flipper trigger information, overwriting its stored paddle offsets in the sprite tables with new ones that appeared in the shared memory location. This allowed for the paddles to be updated to either their "up" or "down" positions every frame. This process was repeated with the same offset the specified number of times (drawing each sub-segment one at a time) until the end of the segment was reached; the next set of offsets was then loaded and the entire process was repeated until the segment counter ran out and the end of the sprite was reached. The process then began again with the next sprite.

Null offsets were inserted into the sprite table at the beginning and end to account for laser travel and power toggle time; null offsets were also added at sharp corners to counter the effects of the galvos' inertia.

3.3.1 Galvanometers and DACs

The external hardware for the laser displaythe DAC used to control the galvanometers, the laser, and the galvanometers themselveswere treated as memory-mapped peripherals and controlled simply by writing to and reading from the corresponding location in memory. Generating a coherent display on a laser projector, however, has a fair share of problems inherent to the hardware that must be solved. The two main problems were issues of inertia: the galvanometers are extremely slow compared to the 50MHz clock of the Beta, and driving them any faster than about 20kHz could break them; similarly, the laser has a not-insignificant power on and off time. These are some of the problems with commercial laser projection systems, as well: the inertia of the galvos tends to round out sharp corners, and the designer has to be careful of laser power during travels between sprites. These issues were accounted for in the sprite drawing process described above with the addition of "null offsets": the effects of these offsets can be clearly seen in the images below.

Figure 8: The display without corner stalls or stalls for laser power.

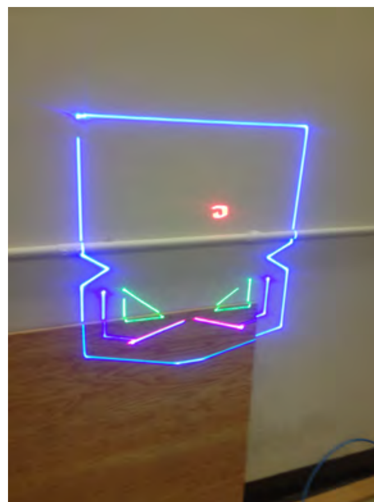


Figure 9: Adding corner stalls to sharpen corners and wait periods for laser power toggle.

We also encountered less common issues, such as one of our first figures being rotated by 90° because of a swapped pair of wires on the board. Additionally, an error in the schematic for the laser control board

resulted in a limited rotation field for the galvos–the board couldn't drive negative voltage and so was clipping the reachable field to the lower right quadrant. This resulted in our first few tests being much smaller than expected given the values we were sending to the galvos. Coding bugs (like underflow and overflow) also became painfully obvious during laser display testing.

Figure 10: One of the first laser tests, both small and rotated.

Figure 11: An assembly bug resulting in the drawing of only half of a sprite.

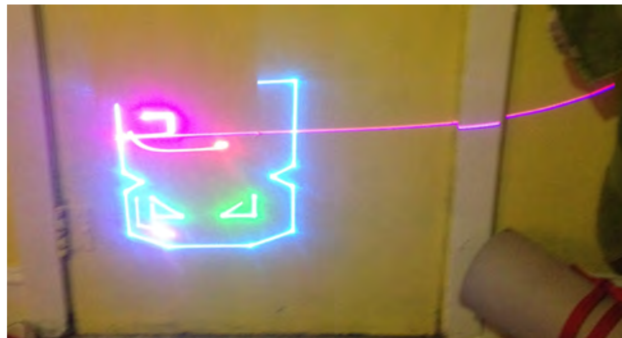


Figure 12: Problematic register underflow causing the laser beam to shoot to the maximum extent of its range.

3.3.2 Testing

Testing the laser driver would have been impossible without the 6.004 testing program BSIM. With BSIM we were able to step through the assembly program and verify that it was doing exactly what we wanted it to. This was most helpful in the drawing of the sprites; mapping each sprite to a list of signed hexadecimal offsets by hand was highly prone to error, which caused problematic overflow and underflow as in the figures above.

4 Implementation Process

4.1 The μBeta Processor

The μBeta processor was based on the two-stage Beta from the old 6.111 course website. First, the core was modified to have correct register initialization values to allow for simulation in ModelSim. We then discovered that the BEQ and BNE opcodes were incorrect, and fixed them. We added multiply to the Beta and remapped the memory space.

We also replaced the original memory mapping of the beta and added a shared memory space as well as memory-mapped I/O to turn it into a real microcontroller. Our memory-mapped I/O consisted of two 32 bit wide input ports, two 32 bit wide output ports, SPI, and a timer capable of triggering Beta interrupts. The operating frequency and number of bits to be transmitted for the SPI were both fully configurable though register settings. The timer period and interrupt enable could both be set from register settings and an overflow flag can be both read and cleared.

Timing analysis of the μBeta processor revealed that it could be operated at slightly below 100MHz on the Virtex-5 FPGA used in the project with a few extra edits, but we chose to clock it at 50MHz to avoid any undesirable effects resulting from a faster clock speed.

We found that we could build a large number of 2-stage pipeline Beta cores on our Virtex-5 board (almost 40). However, the number of processors that can feasibly be used is limited more realistically by the amount of memory each processor needs for program memory and storage. The two Beta cores that we used were given 16k words of memory each (implemented using BRAM), which proved to be sufficient with ample grow room for future modifications. The 50MHz clock speed chosen was more than adequate for our needs. Due to the physical limitations of our galvanometers, the highest estimate we could make for our frame rate was around 20Hz. In the Beta, all instructions are executed in a single cycle; running at 50MHz, we wouldn't have to worry about meeting our frame rate requirements.

To program the Betas, we needed to pre-load the program memory with .coe files detailing the assembled instructions. Conveniently, the 6.004 BSIM program, readily available on the 6.004 course website, outputs both .bin and .coe files. The latter format is a list of comma-delimited hex values which represent the compiled Beta assembly code, which can be used to compile memories for use in Verilog modules by the ISE CoreGen program. With these memory files converted into memories for the Verilog Beta to read, we were able to simulate the Beta running through arbitrary code in ModelSim. Together with the infinitely useful BSIM simulation module that comes free with your copy of the 6.004 Beta assembler, our Verilog test benches proved invaluable to our testing process.

4.2 Debugging the OV7670 Camera

The use of the OV7670 camera in this project became a major hindrance. The cameras are poorly documented, poorly constructed, and require extensive setup; the default configuration settings were unusable, as we needed the camera to output data in RGB format. This required writing configuration registers over I2C. When put into RGB mode image first displayed green as red, red as green, and blue as black. Many hours were spent debugging the camera frame capture Verilog before it was discovered (after much searching the Internet for answers) that the inverted colors were a configuration issue and not an issue with any of the project's Verilog. The C source code for a Linux driver was found for the camera and many of the configuration values were copied from this file. As a testament to the uselessness of the original camera, many of the registers written to were marked as reserved in the data sheet with no other documentation given. Even using the configuration settings found outline, the camera image was of poor quality, which made image recognition difficult.

4.3 Building a Laser Projector

The laser projector was built inside a project box scavenged from the Stata loading dock with new front, top, and bottom panels designed in SolidWorks and lasercut. We ordered the cheapest RGB laser and galvos we could off of eBay, which proved sufficient for the project. The main issue with these was the lack of mounting holes in the power supplies and galvo drivers. Luckily, liberal application of double-sided sticky tape solved the problem. The finished product was a professional-looking box which contained the entire system. A custom board was built to house the galvo-controlling DACs, a schematic of which can be found in the appendix. An image of the PCB is included below.

Figure 13: The galvo DAC PCB.

Figure 14: The housing for the laser and camera system, FPGA mounted on top.

4.4 Xilinx ML505 FPGA Board

In lieu of the 6.111 labkit, our project used a ML505 Virtex 5 FPGA board that a team member acquired from a previous internship. The Virtex 5 FPGA on this board has only a slightly smaller gate count than the the Virtex-II used on the labkit. This is in comparison to the other common FPGA boards, which typically have much smaller gate counts.

Using our own FPGA board allowed us to work on the project at any time and place of our choosing and to build the FPGA into the laser projector. However, using this board came with its own set of challenges. Unlike the student-targeted boards that Digilent and other companies make, the ML505 had no examples on how to use any of the interfaces other than pin assignments and part numbers. As a consequence, getting interfaces like VGA up and running took several days. Once over the biggest hump of the learning curve, however, we were able to use the board quite competently to implement what we needed.

5 Review and Recommendation

As is likely obvious, we deeply underestimated the complexity of this project. Things that we assumed would be quick—like getting the camera working at a basic level, for example—ended up taking us weeks of work. We realized fairly late in the game that our project really required a processor; though we did manage to get a working Beta implementation, it took up so much of our time that we had little time left to work on the relevant parts of our project (object recognition, physics, and laser control). It's no surprise, then, that we didn't finish all parts of our project. Thinking back, we should have been less ambitious and scrapped the camera section entirely. Without having to fight with the camera and deal with vision processing, we would have had enough time to generate a working physics engine and laser controller, even with the necessary Beta implementation.

On the plus side, though, implementing the Beta processor on an FPGA did teach us a lot. It was also truly amazing to be able to write code in assembly, load it into BRAM, and watch the processor behave as expected. Even things like simulating in ModelSim were exciting since we were effectively simulating a multicore processor at the gate level.

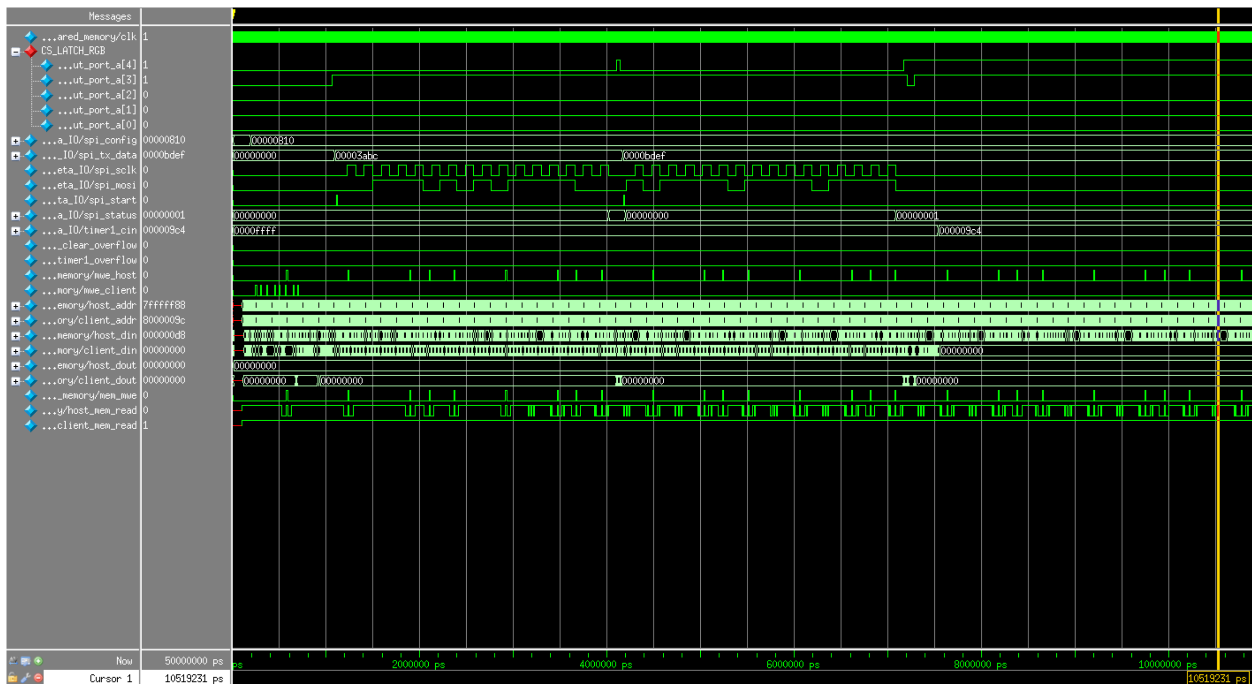


Figure 15: The first test with the two Betas interacting correctly. You can see the SPI transmission occurring and the two Betas (the physics Beta as the host and the laser Beta as the client) using the shared memory space.

And once we implemented the Beta processor we actually did manage to get fairly close to completing all parts of our project. With an extra week, we likely could have finished, even with the hangups from the camera—had we initially decided on a multicore Beta architecture rather than restarting halfway through the project we would have made much more progress towards our original goal.

One thing that would have helped tremendously (and could help in the future, if further iterations of this project are attempted) was a C-to-Beta-assembly compiler. Because the Beta instruction set is non-standard, no such industry-standard compiler exists and past students have relied on a compiler that is no longer maintained by its authors. Our attempts to revive this compiler for use on modern systems failed, but if we had had a compiler writing the physics engine would have been comparatively trivial. Fixed-point math in assembly language is not in any way intuitive and would have been much easier in a high-level language like C.

Something else that could have sped up out testing process was a custom bootloader for the Betas on the FPGA. In order to test new code, we had to rebuild the two 16k word program memories (BRAMs) in ISE, then rebuild the entire project. This process took about 15 to 20 minutes, which held up our code testing frequently. Though we thoroughly simulated in both BSIM and ModelSim, one of our biggest sources of error was the physical system itself, which could only be tested by running the assembly code. Had we been able to serially load in a memory initialization, we would have been able to test code on the system much more often without having to rebuild the entire project.

6 Conclusion

All things considered, our project came together surprisingly well. Though not all elements of the project were integrated, most of them were fully functional as of the checkoff. The only exception to this was the physics engine, which was close to functional but not yet fully debugged. Though our pinball game is unplayable, the fact that we managed to design object recognition software with a terrible camera and build effectively a commercial laser projector (to display a gameboard with controllable paddles) is fairly amazing. While we're disappointed that we weren't able to get everything finished, we are extremely proud of the work that we did pull together.

And, of course, we achieved our goal of making something really cool—a 400mW laser projecting a pinball board and an SNES controller to toggle the paddles is nothing to scoff at.

Glossary

Beta A 32-bit processor designed in the MIT course 6.004. RISC architecture with a limited instruction set.

Galvo Short for galvanometer.

Game object A sprite included as part of the game.

I2C A common communication protocol used for communicating between the FPGA and the OV7670 camera and the DVI/VGA Chrontel chip.

Laser Beta A Beta processor running code to control the laser display.

Lookup table A dedicated chunk of memory within the Beta program memory used to hold information about sprites.

Memory-mapped I/O A way of mapping a memory location in the Beta processor to an I/O port of the FPGA. Used to allow Beta code to interface with peripherals in the same way a standard microcontroller would.

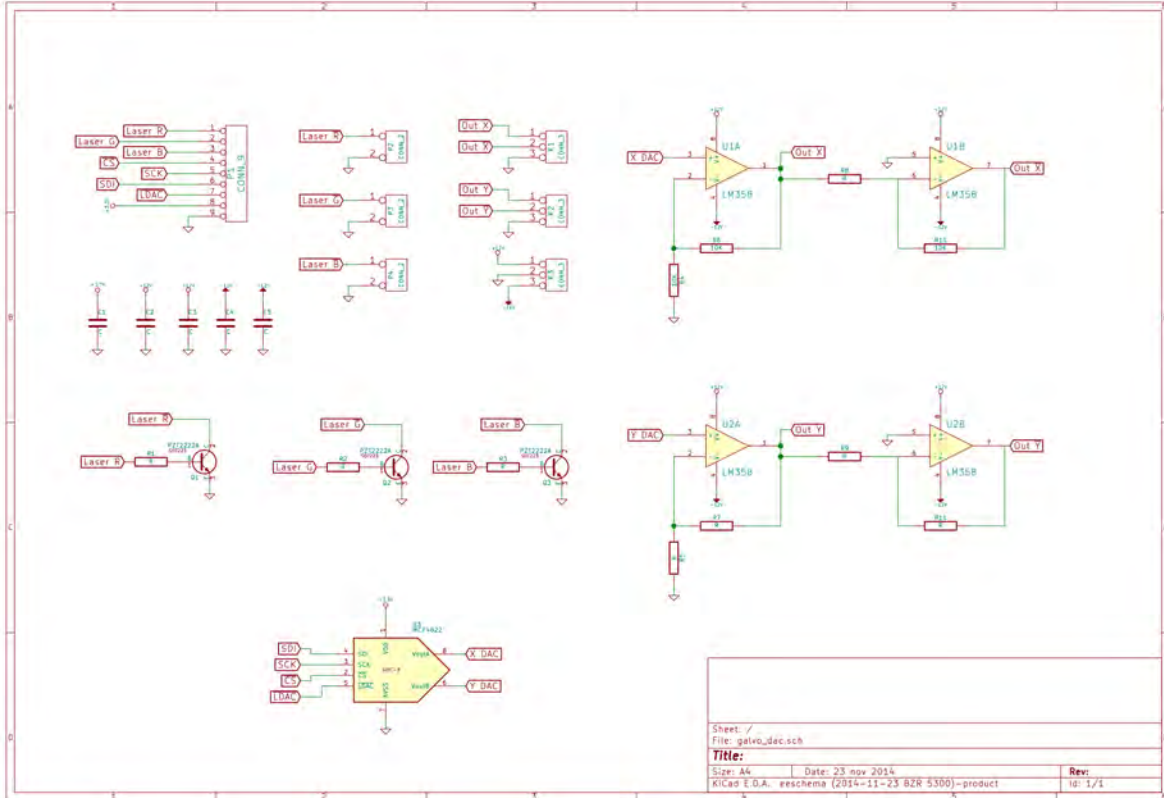
Physics Beta A Beta processor running code to control the physics and game behavior.

SPI A common communication protocol used for communicating between the FPGA and the DACs to drive the galvos.

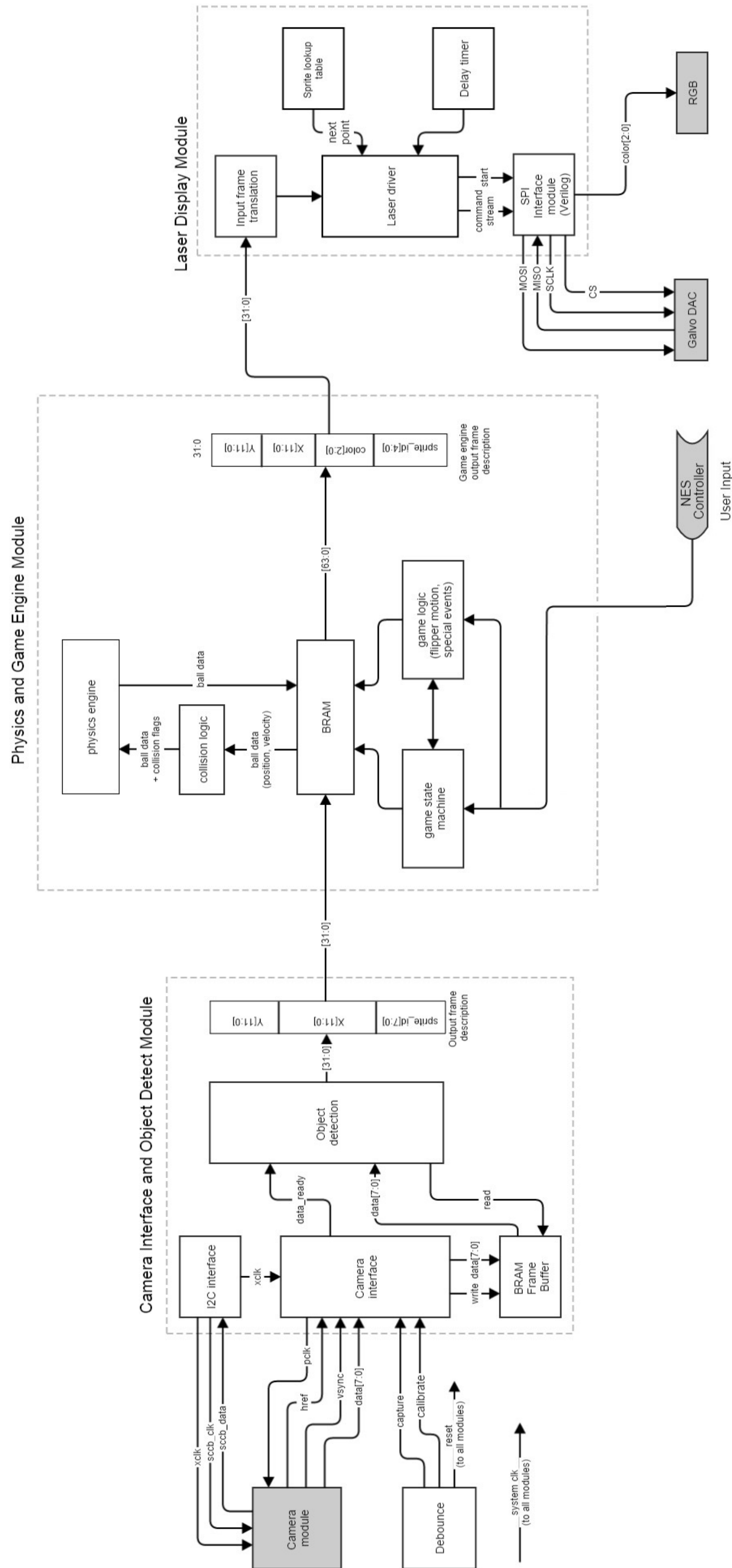
Sprite An individual object to be displayed by the laser projector (for example, the left paddle).

7 Code and Schematics

7.1 Galvo DAC Schematic



7.2 Full Block Diagram



7.3 The Beta

7.3.1 The Two-Stage Pipelined Beta

```
////////////////////////////////////////////////////////////////////////////////
//
// 2-stage pipelined Beta (one bidirectional memory port) [cjt]
//
////////////////////////////////////////////////////////////////////////////////

module beta2(clk,reset,irq,xadr,ma,mdin,mdout,mwe);
  input clk,reset,irq;
  input [30:0] xadr;
  output [31:0] ma,mdout;
  input [31:0] mdin;
  output mwe;

  // beta2 registers
  reg [31:0] npc = 0;
  reg [31:0] pc_inc = 0;
  reg [31:0] inst = 0;
  reg [4:0] rc_save = 0; // needed for second cycle on LD,LDR

  // internal buses
  wire [31:0] rd1,rd2,wd;
  wire [31:0] a,b,xb,c,addsub,cmp,shift,boole,mult;

  // control signals
  wire wasel,werf,z,asel,bsel,csel;
  wire addsub_op,cmp_lt,cmp_eq,shift_op,shift_sxt,boole_and,boole_or;
  wire wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  wire msel,msel_next,branch,trap,interrupt;

  // pc
  wire [31:0] npc_inc,npc_next;
  assign npc_inc = npc + 4;
  assign npc_next = reset ? 32'h80000000 :
                    msel ? npc :
                    branch ? {npc[31] & addsub[31],addsub[30:2],2'b00} :
                    trap ? 32'h80000004 :
                    interrupt ? {1'b1,xadr} :
                    {npc[31],npc_inc[30:0]};
  always @ (posedge clk) begin
    npc <= npc_next; // logic for msel handled above
    if (!msel) pc_inc <= {npc[31],npc_inc[30:0]};
  end

  // instruction reg
  always @ (posedge clk) if (!msel) inst <= mdin;

  // control logic
  decode ctl(.clk(clk),.reset(reset),.irq(irq & !npc[31]),.z(z),
             .opcode(inst[31:26]),
             .asel(asel),.bsel(bsel),.csel(csel),.wasel(wasel),
             .werf(werf),.msel(msel),.msel_next(msel_next),.mwe(mwe),
             .addsub_op(addsub_op),.cmp_lt(cmp_lt),.cmp_eq(cmp_eq),
             .shift_op(shift_op),.shift_sxt(shift_sxt),
             .boole_and(boole_and),.boole_or(boole_or),
```

```
            .wd_addsub(wd_addsub),.wd_cmp(wd_cmp),
            .wd_shift(wd_shift),.wd_boole(wd_boole),.wd_mult(wd_mult),
            .branch(branch),.trap(trap),.interrupt(interrupt));

  // register file
  wire [4:0] wa;
  always @ (posedge clk) if (!msel) rc_save <= inst[25:21];
  assign wa = msel ? rc_save : wasel ? 5'd30 : inst[25:21];

  regfile rf(inst[20:16],rd1,inst[15:11],rd2,inst[25:21],mdout,
                wa,wd,clk,werf);

  assign z = ~| rd1; // used in BEQ/BNE instructions

  // alu
  assign a = asel ? pc_inc : rd1;
  assign b = bsel ? c : rd2;
  assign c = csel ? {{14{inst[15]}},inst[15:0],2'b00} :
                  {{16{inst[15]}},inst[15:0]};

  wire addsub_n,addsub_v,addsub_z;
  assign xb = {32{addsub_op}} ^ b;
  assign addsub = a + xb + addsub_op;
  assign addsub_n = addsub[31];
  assign addsub_v = (addsub[31] & ~a[31] & ~xb[31]) |
                  (~addsub[31] & a[31] & xb[31]);
  assign addsub_z = ~| addsub;

  assign cmp[31:1] = 0;
  assign cmp[0] = (cmp_lt & (addsub_n ^ addsub_v)) | (cmp_eq & addsub_z);

  mul32 mpy(a,b,mult);

  wire [31:0] shift_right;
  // Verilog >>> operator not synthesized correctly, so do it by hand
  shift_right sr(shift_sxt,a,b[4:0],shift_right);
  assign shift = shift_op ? shift_right : a << b[4:0];

  assign boole = boole_and ? (a & b) : boole_or ? (a | b) : a ^ b;

  // result mux, listed in order of speed (slowest first)
  assign wd = msel ? mdin :
             wd_cmp ? cmp :
             wd_addsub ? addsub :
             wd_mult ? mult :
             wd_shift ? shift :
             wd_boole ? boole :
             pc_inc;

  // assume synchronous external memory
  assign ma = msel_next ? {npc[31],addsub[30:0]} : npc_next;
endmodule

////////////////////////////////////////////////////////////////////////////////
//
// 3-port register file
//
////////////////////////////////////////////////////////////////////////////////
```

```
// Beta register file: 32 registers of 32 bits
// R31 always reads as 0
// 3 read ports, 1 write port
module regfile(ra1,rd1,ra2,rd2,ra3,rd3,wa,wd,clk,werf);
  input [4:0] ra1,ra2,ra3,wa;
  output [31:0] rd1,rd2,rd3;
  input [31:0] wd;
  input clk,werf;

  (* ram_style = "distributed" *)
  reg [31:0] regfile[31:0];
  //this may be required for simulation?
  integer i; initial for (i = 0; i < 32; i = i + 1) regfile[i] = 32'h0;

  assign rd1 = regfile[ra1];
  assign rd2 = regfile[ra2];
  assign rd3 = regfile[ra3];

  always @ (posedge clk)
    if (werf && wa != 31) regfile[wa] <= wd;

endmodule

////////////////////////////////////////////////////////////////////////////////
//
// Instruction decode (inst => datapath control signals)
//
////////////////////////////////////////////////////////////////////////////////

module decode(clk,reset,irq,z,opcode,
              asel,bsel,csel,wasel,werf,msel,msel_next,mwe,
              addsub_op,cmp_lt,cmp_eq,
              shift_op,shift_sxt,boole_and,boole_or,
              wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult,
         branch,trap,interrupt);
  input clk,reset,irq,z;
  input [5:0] opcode;
  output asel,bsel,csel,wasel,werf,msel,msel_next,mwe;
  output addsub_op,shift_op,shift_sxt,cmp_lt,cmp_eq,boole_and,boole_or;
  output wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  output branch,trap,interrupt;

  reg asel,bsel,csel,wasel,mem_next;
  reg addsub_op,shift_op,shift_sxt,cmp_lt,cmp_eq,boole_and,boole_or;
  reg wd_addsub,wd_cmp,wd_shift,wd_boole,wd_mult;
  reg branch,trap,interrupt;

  // a little bit of state...
  reg annul,msel,mwrite;


  //always @ (opcode or z or annul or msel or irq or reset)
  always@(*)
  begin
    // initial assignments for all control signals
    asel = 1'hx;
    bsel = 1'hx;
    csel = 1'hx;
    addsub_op = 1'hx;
```

```
shift_op = 1'hx;
shift_sxt = 1'hx;
cmp_lt = 1'hx;
cmp_eq = 1'hx;
boole_and = 1'hx;
boole_or = 1'hx;

wasel = 0;
mem_next = 0;

wd_addsub = 0;
wd_cmp = 0;
wd_shift = 0;
wd_boole = 0;
wd_mult = 0;

branch = 0;
trap = 0;
interrupt = 0;

if (irq && !reset && !annul && !msel) begin
  interrupt = 1;
  wasel = 1;
end else casez (opcode)
  6'b011000: begin // LD
             asel = 0; bsel = 1; csel = 0;
             addsub_op = 0;
    mem_next = 1;
           end
  6'b011001: begin // ST
             asel = 0; bsel = 1; csel = 0;
             addsub_op = 0;
    mem_next = 1;
           end
  6'b011011: begin // JMP
             asel = 0; bsel = 1; csel = 0;
             addsub_op = 0;
             branch = !annul && !msel;
           end
  6'b011100: begin // BEQ //incorrect was 011101
             asel = 1; bsel = 1; csel = 1;
             addsub_op = 0;
             branch = !annul && !msel && z;
           end
  6'b011101: begin // BNE //incorrect was 0111100
             asel = 1; bsel = 1; csel = 1;
             addsub_op = 0;
             branch = !annul && !msel && ~z;
           end
  6'b011111: begin // LDR
             asel = 1; bsel = 1; csel = 1;
             addsub_op = 0;
    mem_next = 1;
           end
  6'b1?0000: begin // ADD, ADDC
             asel = 0; bsel = opcode[4]; csel = 0;
             addsub_op = 0;
             wd_addsub = 1;
           end
```

```
6'b1?0001: begin // SUB, SUBC
            asel = 0; bsel = opcode[4]; csel = 0;
            addsub_op = 1;
            wd_addsub = 1;
          end
6'b1?0010: begin // MUL, MULC
           asel = 0; bsel = opcode[4]; csel = 0;
           wd_mult = 1;
          end
6'b1?0100: begin // CMPEQ, CMPEQC
            asel = 0; bsel = opcode[4]; csel = 0;
            addsub_op = 1;
   cmp_eq = 1; cmp_lt = 0;
            wd_cmp = 1;
          end
6'b1?0101: begin // CMPLT, CMPLTC
            asel = 0; bsel = opcode[4]; csel = 0;
            addsub_op = 1;
   cmp_eq = 0; cmp_lt = 1;
            wd_cmp = 1;
          end
6'b1?0110: begin // CMPLE, CMPLEC
            asel = 0; bsel = opcode[4]; csel = 0;
            addsub_op = 1;
   cmp_eq = 1; cmp_lt = 1;
            wd_cmp = 1;
          end
6'b1?1000: begin // AND, ANDC
            asel = 0; bsel = opcode[4]; csel = 0;
            boole_and = 1; boole_or = 0;
            wd_boole = 1;
          end
6'b1?1001: begin // OR, ORC
            asel = 0; bsel = opcode[4]; csel = 0;
            boole_and = 0; boole_or = 1;
            wd_boole = 1;
          end
6'b1?1010: begin // XOR, XORC
            asel = 0; bsel = opcode[4]; csel = 0;
            boole_and = 0; boole_or = 0;
            wd_boole = 1;
          end
6'b1?1100: begin // SHL, SHLC
            asel = 0; bsel = opcode[4]; csel = 0;
            shift_op = 0;
            wd_shift = 1;
          end
6'b1?1101: begin // SHR, SHRC
            asel = 0; bsel = opcode[4]; csel = 0;
            shift_op = 1; shift_sxt = 0;
            wd_shift = 1;
          end
6'b1?1110: begin // SRA, SRAC
            asel = 0; bsel = opcode[4]; csel = 0;
            shift_op = 1; shift_sxt = 1;
            wd_shift = 1;
          end
default:  begin  // illegal opcode
            trap = !annul && !msel; wasel = 1;
```

```
            end
    endcase
  end

  // state
  wire msel_next = !reset && !annul && mem_next && !msel;
  wire mwrite_next = msel_next && opcode==6'b011001;

  always @ (posedge clk)
  begin
    annul <= !reset && (trap || branch || interrupt);
    msel <= msel_next;
    mwrite <= mwrite_next;
  end

  assign mwe = mwrite_next; // assume synchronous memory
  assign werf = msel ? !mwrite : (!annul & !mem_next);
endmodule

////////////////////////////////////////////////////////////////////////////////
//
// 32-bit signed/unsiged right shift
//
////////////////////////////////////////////////////////////////////////////////

module shift_right(sxt,a,b,shift_right);
  input sxt;
  input [31:0] a;
  input [4:0] b;
  output [31:0] shift_right;

  wire [31:0] w,x,y,z;
  wire sin;

  assign sin = sxt & a[31];
  assign w = b[0] ? {sin,a[31:1]} : a;
  assign x = b[1] ? {{2{sin}},w[31:2]} : w;
  assign y = b[2] ? {{4{sin}},x[31:4]} : x;
  assign z = b[3] ? {{8{sin}},y[31:8]} : y;
  assign shift_right = b[4] ? {{16{sin}},z[31:16]} : z;
endmodule
```

7.3.2 Beta Address Decoding

```
`timescale 1ns / 1ps
module beta_addr_decode(
   input wire [31:0] addr,
   output reg sel_ram = 0,
   output reg sel_IO =0,
   output reg sel_read_shared =0,
   output reg sel_write_shared =0,
   output reg [2:0] read_select =0
    );

   always@(*)
   begin
```

```
  //remove the supervisor bit dummy
  case(addr[30:16])

  0: begin //RAM select
     sel_ram <=1;
     sel_IO <= 0;
     sel_read_shared <= 0;
     sel_write_shared <= 0;
     read_select <= 0;
  end

  1: begin //Memory mapped IO select
     sel_ram <=0;
     sel_IO <= 1;
     sel_read_shared <= 0;
     sel_write_shared <= 0;
     read_select <= 1;
  end

  2: begin //read block select
     sel_ram <=0;
     sel_IO <= 0;
     sel_read_shared <= 1;
     sel_write_shared <= 0;
     read_select <= 2;
  end

  3: begin //write block select
     sel_ram <=0;
     sel_IO <= 0;
     sel_read_shared <= 0;
     sel_write_shared <= 1;
     read_select <= 3;
  end

  default : begin //this should never happen
     sel_ram <=1; //read from main mem? I could trigger error IRQ
     sel_IO <= 0;
     sel_read_shared <= 0;
     sel_write_shared <= 0;
     read_select <= 0;
  end
  endcase

  end
endmodule
```

7.3.3 Memory-Mapped I/O

```
//this is a basic memory mapped IO, SPI, inputs, and outputs
module mapped_IO(
  input wire clk,
  input wire [31:0] addr,//from beta
  input wire [31:0] din, //from beta
  input wire mwe, //memory write enable from beta
  input wire [31:0] in_port_a, in_port_b, // general use input port
```

```
output reg [31:0] dout =0, //data to beta
output reg [31:0] out_port_a = 0,
output reg [31:0] out_port_b = 0, // general output ports
input wire spi_miso, //spi stuff
output wire spi_sclk,
output wire spi_mosi,
output reg irq
 );
//this is on word offset
//registers:
// 0000 - 0004 : in_port_a - in_port b
// 0008 - 000c : out_port_a - out_port_b
// 0010 - 001c : spi_config, spi_start, spi_tx, spi_rx
// 0020 - 002c : system timer 1

wire [31:0] spi_rx_data;
reg [31:0] spi_tx_data =0;
reg [31:0] spi_config = 32'h00_00_02_0a; //default to nice value

reg spi_start;
wire [31:0]spi_status;

//timer 1 inputs
reg [31:0] timer1_cin = 32'h00_00_FF_FF;
reg timer1_irq_enable = 0;
reg timer1_clear_overflow = 0; //in always * block
reg timer1_mwe = 0; //in always * block


//timer 1 outputs
wire [31:0] timer1_cout;
wire timer1_irq;
wire timer1_overflow;



spi_io beta_spi (
 .clk(clk),
 .start(spi_start),
 .ctrl_reg(spi_config),
 .din(spi_tx_data),
 .dout(spi_rx_data),
 .status_reg(spi_status),
 .sclk(spi_sclk),
 .miso(spi_miso),
 .mosi(spi_mosi)
 );

 system_timer beta_timer1 (
 .clk(clk),
 .count_in(timer1_cin),
 .count_out(timer1_cout),
 .clear_overflow(timer1_clear_overflow),
 .irq_enable(timer1_irq_enable),
 .irq(timer1_irq),
 .mwe(timer1_mwe),
 .overflow(timer1_overflow)
 );
```

```
 //handle addresses with no readback
 always @(*)
 begin
   //spi start trigger
   spi_start <= (mwe&&(addr[15:0]==16'h0014)&&din[0]);

   //timer1 overflow clear
   timer1_clear_overflow <= (mwe&&(addr[15:0]==16'h002c)&&din[0]);
   //timer1 mwe set to load new timer value
   timer1_mwe <= (mwe&&(addr[15:0]==16'h0024));

   //deal with irq
   irq <= timer1_irq;
 end

//decode address
always @(posedge clk)
begin
   case(addr[15:0])
   16'h0000: begin //in_port_a
   //no write
   dout <= in_port_a;
   end

   16'h0004: begin //in_port_b
   //no write
   dout <= in_port_b;
   end

   16'h0008: begin //out_port_a
   dout <= out_port_a;
   out_port_a <= mwe ? din : out_port_a;
   end

   16'h000c: begin //out_port_b
   dout <= out_port_b;
   out_port_b <= mwe ? din : out_port_b;
   end

   16'h0010: begin //spi_config reg
   dout <= spi_config;
   spi_config <= mwe ? din : spi_config;
   end

   16'h0014: begin //spi_start/status
   //when wrote to, starts SPI
   //does not retain data value
   //spi start assignment occurs outside clocked logic
   //when read, gives status
   dout <= {31'b0,spi_status};
   end

   16'h0018: begin //spi_tx_data
   dout <= spi_tx_data;
   spi_tx_data <= mwe ? din : spi_tx_data;
   end

   16'h001c: begin //spi_rx_data
   //read only
```

```
      dout <= spi_rx_data;
      end

      16'h0020: begin // timer 1 cout
      //read only
      dout <= timer1_cout;
      end

      16'h0024: begin // timer 1 cin
      //write and read
      dout <= timer1_cin;
      timer1_cin <= mwe ? din : timer1_cin;
      end

      16'h0028: begin // timer 1 irq enable
      dout <= {31'b0,timer1_irq_enable};
      timer1_irq_enable <= mwe ? din[0] : timer1_irq_enable;
      end

      16'h002c: begin //timer 1 overflow flag
      //clearing is write only and occurs in unclocked block
      //read only
      dout <= {31'b0, timer1_overflow};
      end

      default: begin
      dout <= 0;
      end
      endcase
   end


endmodule
```

7.3.4 Beta Instruction Set

```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| 6.004 BETA Macro package -             revised 9/28/11 SAW |||
|||  This file defines our 32-bit Beta instruction set.       |||
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

| Global instruction definition conventions:
|  * DESTINATION arg is LAST

| Instruction set summary. Notation:
| ra, rb, rc: registers
|        CC: 16-bit signed constant
|     label: statement/location tag (becomes PC-relative offset)

| ADD(RA, RB, RC) | RC <- <RA> + <RB>
| ADDC(RA, C, RC) | RC <- <RA> + C
| AND(RA, RB, RC) | RC <- <RA> & <RB>
| ANDC(RA, C, RC) | RC <- <RA> & C
| MUL(RA, RB, RC) | RC <- <RA> * <RB>
| MULC(RA, C, RC) | RC <- <RA> * C
```

```
| DIV(RA, RB, RC) | RC <- <RA> / <RB>
| DIVC(RA, C, RC) | RC <- <RA> / C
| OR( RA, RB, RC) | RC <- <RA> | <RB>
| ORC(RA, C, RC) | RC <- <RA> | C
| SHL(RA, RB, RC) | RC <- <RA> << <RB>
| SHLC(RA, C, RC) | RC <- <RA> << C
| SHR(RA, RB, RC) | RC <- <RA> >> <RB>
| SHRC(RA, C, RC) | RC <- <RA> >> C
| SRA(RA, RB, RC) | RC <- <RA> >> <RB>
| SRAC(RA, C, RC) | RC <- <RA> >> C
| SUB(RA, RB, RC) | RC <- <RA> - <RB>
| SUBC(RA, C, RC) | RC <- <RA> - C
| XOR(RA, RB, RC) | RC <- <RA> ^ <RB>
| XORC(RA, C, RC) | RC <- <RA> ^ C
| XNOR(RA, RB, RC) | RC <- ~(<RA> ^ <RB>)
| XNORC(RA, C, RC) | RC <- ~(<RA> ^ C)

| CMPEQ(RA, RB, RC) | RC <- <RA> == <RB>
| CMPEQC(RA, C, RC) | RC <- <RA> == C
| CMPLE(RA, RB, RC) | RC <- <RA> <= <RB>
| CMPLEC(RA, C, RC) | RC <- <RA> <= C
| CMPLT(RA, RB, RC) | RC <- <RA> < <RB>
| CMPLTC(RA, C, RC) | RC <- <RA> < C


| BR(LABEL,RC)   | RC <- <PC>+4; PC <- LABEL (PC-relative addressing)
| BR(LABEL)   | PC <- LABEL (PC-relative addressing)
| BEQ(RA, LABEL, RC) | RC <- <PC>+4; IF <RA>==0 THEN PC <- LABEL
| BEQ(RA, LABEL) | IF <RA>==0 THEN PC <- LABEL
| BF(RA, LABEL, RC) | RC <- <PC>+4; IF <RA>==0 THEN PC <- LABEL
| BF(RA, LABEL)     | IF <RA>==0 THEN PC <- LABEL
| BNE(RA, LABEL, RC) | RC <- <PC>+4; IF <RA>!=0 THEN PC <- LABEL
| BNE(RA, LABEL) | IF <RA>!=0 THEN PC <- LABEL
| BT(RA, LABEL, RC) | RC <- <PC>+4; IF <RA>!=0 THEN PC <- LABEL
| BT(RA, LABEL)     | IF <RA>!=0 THEN PC <- LABEL
| JMP(RA, RC)     | RC <- <PC>+4; PC <- <RA> & 0xFFFC
| JMP(RB)     | PC <- <RB> & 0xFFFC

| LD(RA, CC, RC) | RC <- <<RA>+CC>
| LD(CC, RC)     | RC <- <CC>
| ST(RC, CC, RA) | <RA>+CC <- <RC>
| ST(RC, CC)     | CC <- <RC>
| LDR(CC, RC)    | RC <- <CC> (PC-relative addressing)

| MOVE(RA, RC)   | RC <- <RA>
| CMOVE(CC, RC)     | RC <- CC
| HALT()   | STOPS SIMULATOR.

| PUSH(RA)    | (2) <SP> <- <RA>; SP <- <SP> - 4
| POP(RA)     | (2) RA <- <<SP>+4>; SP <- <SP> + 4
| ALLOCATE(N)    | Allocate N longwords from stack
| DEALLOCATE(N)     | Release N longwords

| CALL(label)    | Call a subr; save PC in lp.
| CALL(label, n) | (2) Call subr at label with n args.
        | Saves return adr in LP.
        | Pops n longword args from stack.

| RTN()       | Returns to adr in <LP> (Subr return)
```

```
| XRTN()   | Returns to adr in <IP> (Intr return)

| WORD(val)   | Assemble val as a 16-bit datum
| LONG(val)   | Assemble val as a 32-bit datum
| STORAGE(NWORDS) | Reserve NWORDS 32-bit words of DRAM

| GETFRAME(F, RA) | RA <- <<BP>+F>
| PUTFRAME(RA, F) | <BP>+F <- <RA>

| Calling convention:
|  PUSH(argn-1)
|  ...
|  PUSH(arg0)
|  CALL(subr, nargs)
|  (return here with result in R0, args cleaned)

| Extra register conventions, for procedure linkage:
| LP = 28       | Linkage register (holds return adr)
| BP = 29       | Frame pointer (points to base of frame)

| Conventional stack frames look like:
|  arg[N-1]
|  ...
|  arg[0]
|  <saved lp>
|  <saved bp>
|  <other saved regs>
|   BP-><locals>
|       ...
|   SP->(first unused location)

| Convention: define a symbol for each arg/local giving bp-relative offset.
| Then use
|   getframe(name, r) gets value at offset into register r.
|   putframe(r, name) puts value from r into frame at offset name


||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||| End of documentation. Following are the actual definitions... |||
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

| Assemble words, little-endian:
.macro WORD(x) x%0x100 (x>>8)%0x100
.macro LONG(x) WORD(x) WORD(x >> 16) | little-endian for Maybe
.macro STORAGE(NWORDS) . = .+(4*NWORDS)| Reserve NWORDS words of RAM


| register designators
| this allows symbols like r0, etc to be used as
| operands in instructions. Note that there is no real difference
| in this assembler between register operands and small integers.

r0 = 0
r1 = 1
r2 = 2
r3 = 3
r4 = 4
r5 = 5
r6 = 6
```

```
r7 = 7
r8 = 8
r9 = 9
r10 = 10
r11 = 11
r12 = 12
r13 = 13
r14 = 14
r15 = 15
r16 = 16
r17 = 17
r18 = 18
r19 = 19
r20 = 20
r21 = 21
r22 = 22
r23 = 23
r24 = 24
r25 = 25
r26 = 26
r27 = 27
r28 = 28
r29 = 29
r30 = 30
r31 = 31

bp = 27       | frame pointer (points to base of frame)
lp = 28       | linkage register (holds return adr)
sp = 29       | stack pointer (points to 1st free locn)
xp = 30       | interrupt return pointer (lp for interrupts)


| understand upper case, too.
R0 = r0
R1 = r1
R2 = r2
R3 = r3
R4 = r4
R5 = r5
R6 = r6
R7 = r7
R8 = r8
R9 = r9
R10 = r10
R11 = r11
R12 = r12
R13 = r13
R14 = r14
R15 = r15
R16 = r16
R17 = r17
R18 = r18
R19 = r19
R20 = r20
R21 = r21
R22 = r22
R23 = r23
R24 = r24
R25 = r25
```

```
R26 = r26
R27 = r27
R28 = r28
R29 = r29
R30 = r30
R31 = r31
XP = xp
LP = lp
BP = bp
SP = sp

.macro betaop(OP,RA,RB,RC) {
      .align 4
      LONG((OP<<26)+((RC%0x20)<<21)+((RA%0x20)<<16)+((RB%0x20)<<11)) }

.macro betaopc(OP,RA,CC,RC) {
      .align 4
      LONG((OP<<26)+((RC%0x20)<<21)+((RA%0x20)<<16)+(CC%0x10000)) }


.macro ADD(RA, RB, RC)   betaop(0x20,RA,RB,RC)
.macro ADDC(RA, C, RC)   betaopc(0x30,RA,C,RC)

.macro AND(RA, RB, RC)   betaop(0x28,RA,RB,RC)
.macro ANDC(RA, C, RC)   betaopc(0x38,RA,C,RC)
.macro MUL(RA, RB, RC)   betaop(0x22,RA,RB,RC)
.macro MULC(RA, C, RC)   betaopc(0x32,RA,C,RC)
.macro DIV(RA, RB, RC)   betaop(0x23,RA,RB,RC)
.macro DIVC(RA, C, RC)   betaopc(0x33,RA,C,RC)
.macro OR( RA, RB, RC)   betaop(0x29,RA,RB,RC)
.macro ORC(RA, C, RC)    betaopc(0x39,RA,C,RC)
.macro SHL(RA, RB, RC)   betaop(0x2C,RA,RB,RC)
.macro SHLC(RA, C, RC)   betaopc(0x3C,RA,C,RC)
.macro SHR(RA, RB, RC)   betaop(0x2D,RA,RB,RC)
.macro SHRC(RA, C, RC)   betaopc(0x3D,RA,C,RC)
.macro SRA(RA, RB, RC)   betaop(0x2E,RA,RB,RC)
.macro SRAC(RA, C, RC)   betaopc(0x3E,RA,C,RC)
.macro SUB(RA, RB, RC)   betaop(0x21,RA,RB,RC)
.macro SUBC(RA, C, RC)   betaopc(0x31,RA,C,RC)
.macro XOR(RA, RB, RC)   betaop(0x2A,RA,RB,RC)
.macro XORC(RA, C, RC)   betaopc(0x3A,RA,C,RC)
.macro XNOR(RA, RB, RC)  betaop(0x2B,RA,RB,RC)
.macro XNORC(RA, C, RC)  betaopc(0x3B,RA,C,RC)

.macro CMPEQ(RA, RB, RC) betaop(0x24,RA,RB,RC)
.macro CMPEQC(RA, C, RC) betaopc(0x34,RA,C,RC)
.macro CMPLE(RA, RB, RC) betaop(0x26,RA,RB,RC)
.macro CMPLEC(RA, C, RC) betaopc(0x36,RA,C,RC)
.macro CMPLT(RA, RB, RC) betaop(0x25,RA,RB,RC)
.macro CMPLTC(RA, C, RC) betaopc(0x35,RA,C,RC)

.macro BETABR(OP,RA,RC,LABEL) betaopc(OP,RA,((LABEL-.)>>2)-1, RC)
.macro BEQ(RA, LABEL, RC) BETABR(0x1C,RA,RC,LABEL)
.macro BEQ(RA, LABEL)    BETABR(0x1C,RA,r31,LABEL)
.macro BF(RA, LABEL, RC) BEQ(RA,LABEL,RC)
.macro BF(RA,LABEL)   BEQ(RA,LABEL)
.macro BNE(RA, LABEL, RC) BETABR(0x1D,RA,RC,LABEL)
.macro BNE(RA, LABEL)    BETABR(0x1D,RA,r31,LABEL)
.macro BT(RA,LABEL,RC)   BNE(RA,LABEL,RC)
```

```
.macro BT(RA,LABEL)   BNE(RA,LABEL)
.macro BR(LABEL,RC)   BEQ(r31, LABEL, RC)
.macro BR(LABEL)   BR(LABEL, r31)
.macro JMP(RA, RC)    betaopc(0x1B,RA,0,RC)
.macro JMP(RA)     betaopc(0x1B,RA,0,r31)

.macro LD(RA, CC, RC)    betaopc(0x18,RA,CC,RC)
.macro LD(CC, RC)  betaopc(0x18,R31,CC,RC)
.macro ST(RC, CC, RA)    betaopc(0x19,RA,CC,RC)
.macro ST(RC, CC)  betaopc(0x19,R31,CC,RC)
.macro LDR(CC, RC)    BETABR(0x1F, R31, RC, CC)

.macro MOVE(RA, RC)   ADD(RA, R31, RC)
.macro CMOVE(CC, RC)  ADDC(R31, CC, RC)

.macro PUSH(RA)    ADDC(SP,4,SP) ST(RA,-4,SP)
.macro POP(RA)  LD(SP,-4,RA)  ADDC(SP,-4,SP)

.macro CALL(label) BR(label, LP)

.macro RTN()    JMP(LP)
.macro XRTN()   JMP(XP)

| Controversial Extras
| Calling convention:
|  PUSH(argn-1)
|  ...
|  PUSH(arg0)
|  CALL(subr, nargs)
|  (return here with result in R0, args cleaned)

| Extra register conventions, for procedure linkage:
| LP = 28       | Linkage register (holds return adr)
| BP = 29       | Frame pointer (points to base of frame)

| Conventional stack frames look like:
|  arg[N-1]
|  ...
|  arg[0]
|  <saved lp>
|  <saved bp>
|  <other saved regs>
|   BP-><locals>
|       ...
|   SP->(first unused location)

| Convention: define a symbol for each arg/local giving bp-relative offset.
| Then use
|   getframe(name, r) gets value at offset into register r.
|   putframe(r, name) puts value from r into frame at offset name


.macro GETFRAME(OFFSET, REG) LD(bp, OFFSET, REG)
.macro PUTFRAME(REG, OFFSET) ST(REG, OFFSET, bp)
.macro CALL(S,N) BR(S,lp) SUBC(sp, 4*N, sp)

.macro ALLOCATE(N) ADDC(sp, N*4, sp)
.macro DEALLOCATE(N) SUBC(sp, N*4, sp)
```

```
|-------------------------------------------------------
| Privileged mode instructions
|-------------------------------------------------------

.macro PRIV_OP(FNCODE)  betaopc (0x00, 0, FNCODE, 0)
.macro HALT() PRIV_OP (0)
.macro RDCHAR() PRIV_OP (1)
.macro WRCHAR() PRIV_OP (2)
.macro CYCLE() PRIV_OP (3)
.macro TIME() PRIV_OP (4)
.macro CLICK() PRIV_OP (5)
.macro RANDOM() PRIV_OP (6)
.macro SEED() PRIV_OP (7)
.macro SERVER() PRIV_OP (8)

| SVC calls; used for OS extensions

.macro SVC(code)   betaopc (0x01, 0, code, 0)

| Trap and interrupt vectors
VEC_RESET  = 0      | Reset (powerup)
VEC_II     = 4      | Illegal instruction (also SVC call)
VEC_CLK    = 8      | Clock interrupt
VEC_KBD    = 12     | Keyboard interrupt
VEC_MOUSE  = 16     | Mouse interrupt

| constant for the supervisor bit in the PC
PC_SUPERVISOR    = 0x80000000     | the bit itself
PC_MASK           = 0x7fffffff    | a mask for the rest of the PC
```

7.3.5 Laser Controller Assembly

```
.include beta.uasm

| REGISTER MAP
|  r0:   x location
|  r1:   y location
|  r2:   current sprite id
|  r3:   rgb value for laser
|  r4:   current offset into shared memory
|  r5:   counter
|  r6:   length of sprite
|  r7:   scratch
|  r8:   scratch
|  r9:   current location in local sprite lookup table
|  r10:  spi tx data, daca
|  r11:  spi tx data, dacb
|  r12: holds remaining number of points in current segment
|  r13:  scratch
|  r14:  scale factor - currently inactive
|  r15:  laser power flag for implementing travels
|  r16:  reserved for copy of rgb data
|  r17:  scratch
|  r18:  scratch
|  r19: holds scaling factor
|  r20:  travel time
```

```
|  r21:  stall time


| Define parameters
NEXT_SPRITE_OFFSET = 0x04
|TIMER_VALUE = 0x0D05    | 15kHz
TIMER_VALUE = 0x09C4     | 20kHz
|TIMER_VALUE = 0x01      | DEBUG

| External address offsets
SPI_CONFIG = 0x10
SPI_STATUS = 0x14
INPUT_PORT_A = 0x0
DAC_CTL_OUT = 0x08
SPI_TX = 0x18
TIMER_SET = 0x24
TIMER_OVERFLOW = 0x2C
SHARED_MEM_WRITE_STATUS = 0x100
SHARED_MEM_READ_STATUS = 0x101
SWITCHES = 0x00
STALL_TIME = 0x04          | DEBUG
TRAVEL_TIME = 0x16         | DEBUG
SCALING_FACTOR = 0x0       | full scale (DEBUG)
|SCALING_FACTOR = 0x1      | half scale (DEBUG)
LEFT_PADDLE_UPDATE = 0xF8  | location of left and right paddle offset update values
RIGHT_PADDLE_UPDATE = 0xFC


. = 0
BR(init)

. = 4
INTERRUPT:
ADD(r31, r31, r31)
XRTN()

| Output port B (0008): {ADC_CSN, ADC_latch_n, R, G, B}
| Input port A (0000): { [31:24] TRAVEL_TIME, [23:16] STALL_TIME, [15:8] UNUSED, [7:0]
    SCALING_FACTOR }

| Initialize sprite location in shared memory, initialize SPI, get scale factor
init:
   CMOVE(stack, SP)           | Initialize stack
   CMOVE(2, r4)               | initialize sprite location in shared memory
   SHLC(r4, 16, r4)

   CMOVE(1, r7)               | initialize output port location in shared memory
   SHLC(r7, 16, r7)
   CMOVE(0x0810, r8)          | configure SPI
   ST(r8, SPI_CONFIG, r7)

   LD(r7, INPUT_PORT_A, r19)  | load input port A into r19
   CMOVE(0xFF, r8)
   SHLC(r8, 24, r8)
   AND(r8, r19, r20)          | load travel time value into r20
   SHRC(r8, 8, r8)
   AND(r8, r19, r21)          | load stall time into r21
   CMOVE(0xFF, r8)
   AND(r8, r19, r19)          | load scaling factor into r19
```

```
| Draws every sprite in a frame
check_data_available:
   LD(r4, SHARED_MEM_WRITE_STATUS, r7) | check shared memory write status
   BNE(r7, check_data_available)
   CMOVE(1, r7)
   ST(r7, SHARED_MEM_READ_STATUS, r4)  | set the read flag
draw_frame:
   | Load sprite IDs until you find a null-terminated one
   LD(r4, 0, r7)              | load new sprite data into r7
   SHRC(r7, 27, r2)           | get just the sprite ID
   BEQ(r2, frame_done)        | if sprite ID is null then we're done, otherwise continue loading

   SUBC(r20, 1, r8)           | store 0xFFFFFFFF in r8
   SHRC(r8, 8, r8)            | 0x00FFFFFF in r8
   AND(r7, r8, r0)            | mask off the sprite ID and RGB data
   SHRC(r0, 12, r0)           | store only x data in r0
   SHRC(r8, 12, r8)           | 0x00000FFF in r8
   AND(r7, r8, r1)            | store only y data in r1
   SHRC(r7, 24, r16)          | mask off x and y data
   ANDC(r16, 0x07, r16)       | rgb data in r16
   CALL(draw_sprite)

   ADDC(r4, NEXT_SPRITE_OFFSET, r4) | get next sprite location in shared memory

   BR(draw_frame)

frame_done:
   CMOVE(2, r4)
   SHLC(r4, 16, r4)           | clear frame offset (shared memory)
   CMOVE(0, r7)
   ST(r7, SHARED_MEM_READ_STATUS, r4) | clear busy flag

   BR(check_data_available)


|| Start the timer, wait for it to finish, and clear the flag (have to reload counter to reset)
set_timer:

   | Initialize and reset timer
   CMOVE(1, r7)               | store timer address in r7
   SHLC(r7, 16, r7)
   CMOVE(TIMER_VALUE, r8)     | load for gavlo update frequency
   ST(r8, TIMER_SET, r7)

wait_timer:
   LD(r7, TIMER_OVERFLOW, r8)
| update paddles while we're waiting for the timer
   SHLC(r7, 1, r13)                   | address into shared memory
   LD(r13, LEFT_PADDLE_UPDATE, r17)   | update left paddle
   ST(r17, left_paddle_hops, r31)
   LD(r13, RIGHT_PADDLE_UPDATE, r17)  | update right paddle
   ST(r17, right_paddle_hops, r31)
|  BNE(r8, wait_timer)                | DEBUG
   BEQ(r8, wait_timer)                | flag should be set when timer is done
   CMOVE(0x0, r8)                     | clear flag
   ST(r8, TIMER_OVERFLOW, r7)
```

```
  RTN()
|| Draw a single sprite
draw_sprite:
  | Get length of sprite from lookup table, store in r6
  |.breakpoint
  SHLC(r2, 8, r8)               | translate sprite ID into lookup table offset
  ADDC(r8, sprite_lookup, r9)
  LD(r9, 0, r6)                 | first entry in lookup table is sprite length
  ADDC(r9, 0x4, r9)             | store location of first point for get_next_point routine
  LD(r9,0x4,r12)                | load number of points in next segment into r12
  SHLC(r12, SCALING_FACTOR, r12)  | multiply the number of points by the scaling factor
  CMOVE(0x1, r15)               | set flag signaling laser turn on after next segment

| Draw the sprite:
draw_loop:
  PUSH(LP)
  CALL(go_to_point)
  CALL(get_next_point)
  CALL(set_timer)
  POP(LP)

  BNE(r6, draw_loop)
  RTN()

|| Go to a single point by writing its location over SPI
go_to_point:

  CMOVE(0x02, r8)              | put SPI address in r8
  SHLC(r8, 16, r8)
  SHRC(r0,SCALING_FACTOR,r17)  | restore scaling
  SHRC(r1,SCALING_FACTOR,r18)

  ORC(r3, 0b01000, r7)        | Store CS & RGB data in r7
  CMOVE(0b0011, r10)          | Store config data (1) in r10
  SHLC(r10, 12, r10)          | Shift left to bit 15
  OR(r10, r17, r10)           | r10 now contains config data for write to DACA

  CMOVE(0b01011, r11)         | Store config data (2) in r11
  SHLC(r11, 12, r11)          | Shift left to bit 15
  OR(r11, r18, r11)           | r11 now contains config data for write to DACB
  |.breakpoint
  CMOVE(0x01, r8)             | put SPI address in r8
  SHLC(r8, 16, r8)
  ST(r7, DAC_CTL_OUT, r8)     | Write to output port A (memory location 8)--lower CS
  ST(r10, SPI_TX, r8)         | Write configuration and X data to SPI TX (0018)

  CMOVE(1, r13)               | Store 1 (start flag) in r13
  ST(r13, SPI_STATUS, r8)     | Start SPI
spi_wait_x:                   | Wait for SPI completion flag
  LD(r8, SPI_STATUS, r13)
  BF(r13, spi_wait_x)

  ORC(r7, 0b10000, r7)
  ST(r7, DAC_CTL_OUT, r8)     | Write to output port A (memory location 8)--raise CS
  ADD(r31, r31, r31)
  ANDC(r7, 0b01111, r7)

  ST(r7, DAC_CTL_OUT, r8)     | Write to output port A--lower CS
  ADD(r31, r31, r31)
```

```
   ST(r11, SPI_TX, r8)        | Write configuration and Y data to SPI TX (0018)
   ST(r13, SPI_STATUS, r8)    | Start SPI

spi_wait_y:                   | Wait for second SPI completion flag
   LD(r8, SPI_STATUS, r13)
   BF(r13, spi_wait_y)

   ORC(r7, 0b10000, r7)
   ST(r7, DAC_CTL_OUT, r8)    | Write to output port A (memory location 8)--raise CS
   ADD(r31, r31, r31)
   SUBC(r7, 0b01000, r7)
   ST(r7, DAC_CTL_OUT, r8)    | Lower ADC latch
   ADD(r31, r31, r31)
   ADD(r31, r31, r31)
   ADD(r31, r31, r31)
   ADD(r31, r31, r31)
   ADDC(r7, 0b01000, r7)
   ST(r7, DAC_CTL_OUT, r8)    | Raise ADC latch
   ADD(r31, r31, r31)
   ADD(r31, r31, r31)
   ADD(r31, r31, r31)
   ADD(r31, r31, r31)
   RTN()

|| Get the location of the next point within a shape and store the updated location in registers 0
    and 1 (x and y)
get_next_point:

   LD(r9, 0, r8)              | load next point into r8
   MOVE(r8, r7)               | copy point
   SUBC(r12,0x1,r12)          | decrement remaining points in the line
   SRAC(r8, 16, r8)           | Get only x data
   SHLC(r7, 16, r7)           | Get only y data, preserving sign bit
   SRAC(r7, 16, r7)
   ADD(r8, r0, r0)            | Add x offset
   ADD(r7, r1, r1)            | Add y offset
   BNE(r12, get_next_end)     | if we're done with this line segment,
      .breakpoint
         ADDC(r9, 0x8, r9)    | increment location in local table
         LD(r9,0x4,r12)       | load point count for next segment
      SUBC(r6, 0x01, r6)      | decrement points left for sprite
         CMPEQC(r6, 1, r8)    | test to see if we're about to do our last stall

      BEQ(r8, check_laser_flag)  | if we are, turn the laser off
         |.breakpoint
              CMOVE(0x0,r3)      | turn laser off
              CMOVE(0x0,r15)     | clear flag so the laser doesn't turn on during the next stall

check_laser_flag:
   BEQ(r15, get_next_end)
   |.breakpoint
      MOVE(r16,r3)               | reload RGB data to turn laser back on

get_next_end:
   RTN()


| Sprite lookup tables: one table for each sprite, memory location corresponds to sprite ID
sprite_lookup:
```

```
. = sprite_lookup+0x100
LONG(7)                                 | a four-pointed circle for the ball: 1
LONG(0x00000000), LONG(TRAVEL_TIME)     | stall for travel time
LONG(0x00000000), LONG(STALL_TIME)      | stall for laser on
LONG(0x00100000), LONG(0x08)
LONG(0x00000010), LONG(0x08)
LONG(0xFFF00000), LONG(0x08)
LONG(0x0000FFF0), LONG(0x08)
LONG(0x00000000), LONG(STALL_TIME)

. = sprite_lookup+0x200
LONG(7)                                 | arbitrary circle (three times bigger): 2
LONG(0x00000000), LONG(TRAVEL_TIME)     | stall for travel time
LONG(0x00000000), LONG(STALL_TIME)      | stall for laser on
LONG(0x00300000), LONG(0x08)
LONG(0x00000030), LONG(0x08)
LONG(0xFFD00000), LONG(0x08)
LONG(0x0000FFD0), LONG(0x08)
LONG(0x00000000), LONG(STALL_TIME)

. = sprite_lookup+0x300                 | the frame outline: 3
LONG(26)
LONG(0x00000000), LONG(TRAVEL_TIME)     | stall for travel time
LONG(0x00000000), LONG(STALL_TIME)      | stall for laser on
LONG(0x00500000), LONG(0x20)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x00000030), LONG(0x20)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFD80018), LONG(0x8)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x00280018), LONG(0x8)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x00000030), LONG(0x12)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFBA0012), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFEC0000), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFBAFFEE), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x0000FFD0), LONG(0x12)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x0028FFE8), LONG(0x8)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFD8FFE8), LONG(0x8)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x0000FFD0), LONG(0x20)
LONG(0x00000000), LONG(STALL_TIME)      | stall for laser off

. = sprite_lookup+0x400                 | left triangle bumper: 4
LONG(8)
LONG(0x00000000), LONG(TRAVEL_TIME)     | stall for travel time
LONG(0x00000000), LONG(STALL_TIME)      | stall for laser on
LONG(0x001E0018), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFE20000), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x0000FFE8), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)      | stall for laser off
```

```
. = sprite_lookup+0x500              | right triangle bumper: 5
LONG(8)
LONG(0x00000000), LONG(TRAVEL_TIME)   | stall for travel time
LONG(0x00000000), LONG(STALL_TIME)    | stall for laser on
LONG(0x00000018), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFE20000), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x001EFFE8), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)    | stall for laser off

. = sprite_lookup+0x600    | left bumpery thing: 6
LONG(6)
LONG(0x00000000), LONG(TRAVEL_TIME)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x00000022), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x00220008), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)

. = sprite_lookup+0x700
LONG(6)            | right bumpery thing: 7
LONG(0x00000000), LONG(TRAVEL_TIME)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0x00000022), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)
LONG(0xFFDE0008), LONG(0x10)
LONG(0x00000000), LONG(STALL_TIME)

. = sprite_lookup+0x800
LONG(4)            | left paddle: 8
LONG(0x00000000), LONG(TRAVEL_TIME)
LONG(0x00000000), LONG(STALL_TIME)
left_paddle_hops:
LONG(0x00180008), LONG(0x20)
LONG(0x00000000), LONG(STALL_TIME)

. = sprite_lookup+0x900
LONG(4)            | right paddle: 9
LONG(0x00000000), LONG(TRAVEL_TIME)
LONG(0x00000000), LONG(STALL_TIME)
right_paddle_hops:
LONG(0xFFE80008), LONG(0x20)
LONG(0x00000000), LONG(STALL_TIME)


stack:
STORAGE(128)

|.=0x10000         | DEBUG
|LONG(0x1)
|.=0x20000
|LONG(0x1B000000)
|LONG(0x151E01E0)
|LONG(0x0C600400)
|LONG(0x221E07E0)
|LONG(0x2A8207E0)
|.=0x40000
```

```
|LONG(0x1EEB)
```

7.3.6 Tested Physics Engine for Checkoff: No implemented physics, paddle control only

```
.include beta.uasm    | Define Beta instructions, etc.
.options clock tty

|========================================================
| REGISTER MAP
| r0: scratch
| r1: sprite ID (five bits)
| r2: position OR angle
| r3: x velocity
| r4: y velocity
| r5: color (RGB)
| r8: instance pointer
| r9: PC
| r10:
| r15: external base address
| r16: offset external
| r17: argument to write_external
| r18: byte offset sprite ID
|========================================================

POSITION_OFFSET = 0x04
COLOR_OFFSET = 0x10
NEXT_SPRITE_OFFSET = 0x14
L_PADDLE_UPDATE = 0xF8
R_PADDLE_UPDATE = 0xFC

. = 0
BR(start)

. = 0x4
INT_V:
ADDC(R31, 0, R31)
XRTN()

. = 0x10
start:
CMOVE(stack, SP)

CMOVE(0xFFFF,R10)
SHLC(R10, 16, R10)
SHRC(R10, 16, R10)

CMOVE(0x3,R1)
CMOVE(0,R2)
CMOVE(0x7,R3)
CMOVE(0x9,R4)
CMOVE(0x3,R5)
CALL(build_object) | bounding box: cyan

|CMOVE(0x2,R1)
|CMOVE(0x01E0,R2)
|SHLC(R2,12,R2)
```

```
|CMOVE(0x01E0,R0)
|AND(R0,R10,R0)
|ADD(R0,R2,R2)
|CMOVE(0x9,R3)
|CMOVE(0x1,R4)
|CMOVE(0x5,R5)
|CALL(build_object) | arbitrary circle: purple

CMOVE(0x1,R1)
CMOVE(0x0600,R2)
SHLC(R2,12,R2)
CMOVE(0x0400,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x9,R3)
CMOVE(0x1,R4)
CMOVE(0x4,R5)
CALL(build_object) | arbitrary circle (puck): red

CMOVE(0x4,R1)
CMOVE(0x01E0,R2)
SHLC(R2,12,R2)
CMOVE(0x07E0,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x9,R3)
CMOVE(0x1,R4)
CMOVE(0x2,R5)
CALL(build_object) | left bumper: green

CMOVE(0x6,R1)
CMOVE(0x00F0,R2)
SHLC(R2,12,R2)
CMOVE(0x07E0,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x9,R3)
CMOVE(0x1,R4)
CMOVE(0x1,R5)
CALL(build_object) | left slide thing: blue

CMOVE(0x8,R1)
CMOVE(0x023E,R2)
SHLC(R2,12,R2)
CMOVE(0x0A80,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x9,R3)
CMOVE(0x1,R4)
CMOVE(0x7,R5)
CALL(build_object) | left paddle: white

CMOVE(0x5,R1)
CMOVE(0x0820,R2)
SHLC(R2,12,R2)
CMOVE(0x07E0,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x9,R3)
```

```
CMOVE(0x1,R4)
CMOVE(0x2,R5)
CALL(build_object) | right bumper: green

CMOVE(0x7,R1)
CMOVE(0x0910,R2)
SHLC(R2,12,R2)
CMOVE(0x07E0,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x9,R3)
CMOVE(0x1,R4)
CMOVE(0x1,R5)
CALL(build_object) | right slide thing: blue

CMOVE(0x9,R1)
CMOVE(0x07C2,R2)
SHLC(R2,12,R2)
CMOVE(0x0A80,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x9,R3)
CMOVE(0x1,R4)
CMOVE(0x7,R5)
CALL(build_object) | right paddle: white

loop:
  CALL(update)
  BR(loop)

update:               | process next game object

update_start:
  LD(instance_list,R8)    | load the current offset into the list
  LD(R8,0,R1)              | load SPRITE_ID of next object
  LD(R8,POSITION_OFFSET,R2) | load POSITION of next object
  LD(R8,COLOR_OFFSET,R5)   | load RGB of next object

| packet build routine
  SHLC(R1,27,R17)  | shift SPRITE_ID into upper five bits of R17
  SHLC(R5,24,R0)   | shift RGB into bits [26:24] of R17
  ADD(R17,R0,R17)  | combine
  ADD(R17,R2,R17)  | include POSITION: R17 now contains {SPRITE_ID[4:0], RGB[2:0],
      POSITION_X[11:0], POSITION_Y[11:0]}
|.breakpoint

  PUSH(LP)            | gotta push LP before you call a function within a function!
  CALL(write_external)
  POP(LP)          | restore LP
  BNE(R1,cont_update,R31)    | if SPRITE_ID != NULL, continue
    CMOVE(instance_list+4,R8) | reset list pointer
    ST(R8,instance_list)
    CMOVE(0x0,R17)
    CALL(write_external)
    BR(update_start)          | start over
cont_update:
  LD(R8,4,R2)                | load coordinates of next object
  SHLC(R1,0x2,R18)          | turn SPRITE_ID into a byte offset
  LD(R18,object_routines,R18) | LD address of object-specific update routine
```

```
  .breakpoint
  JMP(R18, R9)              | JMP to execute; save PC in R9
  ADDC(R8,NEXT_SPRITE_OFFSET,R8)           | point list pointer to next next object
  ST(R8,instance_list)
  RTN()

|+========
pinball:
JMP(R9)                     | return to update function

|+========
r_flipper:
.breakpoint
PUSH(LP)
CALL(paddle_update)
POP(LP)
JMP(R9)                   | return to update function

|+========
l_flipper:
.breakpoint
PUSH(LP)
CALL(paddle_update)
POP(LP)
JMP(R9)                   | return to update function

|+========
tri_bump:
JMP(R9)                   | return to update function

|+========
circ_bump:
JMP(R9)                   | return to update function

|+========
board:
JMP(R9)                   | return to update function

|+========
fixme_1:
JMP(R9)                   | return to update function

|+========
fixme_2:
JMP(R9)                   | return to update function

|+========
fixme_3:
JMP(R9)                   | return to update function

|=======================================================
| OBJECT ROUTINE LOOKUP
|   index into list using SPRITE_ID*4
|   load address, then JMP to execute update routine
|======================================================

object_routines:
  LONG(0xABCD)
  LONG(pinball)
```

```
  LONG(r_flipper)
  LONG(l_flipper)
  LONG(tri_bump)
  LONG(circ_bump)
  LONG(board)
  LONG(fixme_1)
  LONG(fixme_2)
  LONG(fixme_3)

|=========================================================
| OBJECT INSTANCE LIST STRUCTURE:
|   0x00 pointer to object list: increment by 8 after
|        processing an object
|   0x04 SPRITE_ID - indicates what class of object we are
|        handling
|   0x08 POSITION - x and y data
|   0x0C VEL_X - x velocity
|   0x10 VEL_Y - y velcoity
|   0x14 RGB - sprite color
|   ...
|   ... list of objects to be processed
|   ...
|   0x4n - NULL
|=========================================================

instance_list: LONG(.+4) | pointer to the object list
  STORAGE(256)           | allocated object memory

|+==========+
| for writing to the memory block shared with the laser projector
|===========
write_external:
  CMOVE(0x3000,R15)
  SHLC(R15,4,R15)    | load R15 with the bottom address of the external mem
  ADD(R15,R16,R15)   | add offset into memory from R16
  ST(R17,0,R15)      | write the contents of R17 into memory
|.breakpoint
  ADDC(R16,4,R16)    | increment offset for next time
  BNE(R17,write_end)
    CMOVE(0x0,R16)   | if we're done with the frame, clear offset
write_end:
  RTN()
|+==========+
| builds an object in the instance_list
| before calling, put the appropriate values in R1-5
|***********
build_object:
  LD(count, R0)
  ADDC(R0,1,R0)               | increment the count for next time
  ST(R0, count)               | store the new count back in memory
  SUBC(R0,1,R0)
  MULC(R0,20,R0)              | turn count in R0 into the offset into the object list
  ST(R1,instance_list+4,R0)   | store object attributes
  ST(R2,instance_list+8,R0)
  ST(R3,instance_list+12,R0)
  ST(R4,instance_list+16,R0)
  ST(R5,instance_list+20,R0)
  ST(R31,instance_list+24,R0) | end list with NULL; gets overwritten by next object
.breakpoint
```

```
  RTN()
count: LONG(0x0)    | for storing the object instance count
paddle_update:
   PUSH(r0)
   PUSH(r1)
   PUSH(r2)
   PUSH(r7)
   PUSH(r8)
   CMOVE(0x1, r7) | right paddle mask
   CMOVE(0x2, r8) | left paddle mask

   CMOVE(0x1, r0)
   SHLC(r0, 16, r0)
   LD(r0, 0, r0)    | get value from input port a. lsb is right paddle, second bit is left paddle.

   CMOVE(0x3, r1)
   SHLC(r1, 16, r1)
   ORC(r1, 0x00F8, r3) | left paddle update location
   ORC(r1, 0x00FC, r2) | right paddle update location

   AND(r7, r0, r7)
   BEQ(r7, right_paddle_down)
right_paddle_up:
   LD(right_paddle_up_val, r13)
   ST(r13, 0, r2)
   AND(r8, r0, r8)
   BEQ(r8, left_paddle_down)
   BR(left_paddle_up)
right_paddle_down:
   LD(right_paddle_down_val, r13)
   ST(r13, 0, r2)
   AND(r8, r0, r8)
   BEQ(r8, left_paddle_down)
   BR(left_paddle_up)
left_paddle_up:
   LD(left_paddle_up_val, r13)
   ST(r13, 0, r3)
   POP(r8)
   POP(r7)
   POP(r2)
   POP(r1)
   POP(r0)
   RTN()

left_paddle_down:
   LD(left_paddle_down_val, r13)
   ST(r13, 0, r3)
   POP(r8)
   POP(r7)
   POP(r2)
   POP(r1)
   POP(r0)
   RTN()
left_paddle_down_val:
LONG(0x00180008)
left_paddle_up_val:
LONG(0x0018FFF8)
right_paddle_down_val:
LONG(0xFFE80008)
```

```
right_paddle_up_val:
LONG(0xFFE8FFF8)
stack:
STORAGE(256)
|.=0x10000
|LONG(0xFE000001)
|. = 0x00400000
|LONG(0x1EEB)
```

7.3.7 Physics Engine Untested Working Copy

```
.include beta.uasm    | Define Beta instructions, etc.
.options clock tty

|========================================================
| REGISTER MAP
| r0: scratch
| r1: sprite ID (five bits)
| r2: position OR angle
| r3: x velocity --- both xvel and yvel are left shifted
| r4: y velocity --- by a byte to give fractional components
| r5: color (RGB)
| r6: accel due to gravity based on timestep
| r8: instance pointer
| r9: PC
| r10: 0x0000FFFF
| r11-14 : scratch
| r15: external base address
| r16: offset external
| r17: argument to write_external
| r18: byte offset sprite ID
|========================================================

t0 = R19
t1 = R20
t2 = R21
t3 = R22
t4 = R23
t5 = R24
t6 = R25
t7 = r26

| Define parameters

TIMER_VALUE = 0x8000 | 1.526 kHz
|TIMER_VALUE = 0x1     | DEBUG

GRAV_FRACT = 0xC      | .75 pixels/second^2 left shifted by a byte
FRACT_SHIFT = 0x8      | otherwise known as 3/4

POSITION_OFFSET = 0x04
COLOR_OFFSET = 0x10
NEXT_SPRITE_OFFSET = 0x14
TIMER_SET = 0x24
TIMER_OVERFLOW = 0x2C
INPUT_PORT_A = 0x0
```

```
. = 0
BR(start)

. = 0x4
INT_V:
ADDC(R31, 0, R31)
XRTN()

. = 0x10
start:
CMOVE(stack, SP)

CMOVE(0xFFFF,R10)
SHLC(R10, 16, R10)
SHRC(R10, 16, R10)

CMOVE(0x3,R1)
CMOVE(0,R2)
CMOVE(0x0,R3)
CMOVE(0x0,R4)
CMOVE(0x3,R5)
CALL(build_object) | bounding box: cyan

CMOVE(0x2,R1)
CMOVE(0x02E0,R2)
SHLC(R2,12,R2)
CMOVE(0x01E0,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x0,R3)
CMOVE(0x0,R4)
CMOVE(0x5,R5)
CALL(build_object) | arbitrary circle: purple

CMOVE(0x1,R1)
CMOVE(0x0600,R2)
SHLC(R2,12,R2)
CMOVE(0x0400,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x0,R3)
CMOVE(0x0,R4)
CMOVE(0x4,R5)
CALL(build_object) | arbitrary circle (puck): red

CMOVE(0x4,R1)
CMOVE(0x01E0,R2)
SHLC(R2,12,R2)
CMOVE(0x07E0,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x0,R3)
CMOVE(0x0,R4)
CMOVE(0x2,R5)
CALL(build_object) | left bumper: green

CMOVE(0x6,R1)
CMOVE(0x00F0,R2)
```

```
SHLC(R2,12,R2)
CMOVE(0x07E0,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x0,R3)
CMOVE(0x0,R4)
CMOVE(0x1,R5)
CALL(build_object) | left slide thing: blue

CMOVE(0x8,R1)
CMOVE(0x023E,R2)
SHLC(R2,12,R2)
CMOVE(0x0A80,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x0,R3)
CMOVE(0x0,R4)
CMOVE(0x7,R5)
CALL(build_object) | left paddle: white

CMOVE(0x5,R1)
CMOVE(0x0820,R2)
SHLC(R2,12,R2)
CMOVE(0x07E0,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x0,R3)
CMOVE(0x0,R4)
CMOVE(0x2,R5)
CALL(build_object) | right bumper: green

CMOVE(0x7,R1)
CMOVE(0x0910,R2)
SHLC(R2,12,R2)
CMOVE(0x07E0,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x0,R3)
CMOVE(0x0,R4)
CMOVE(0x1,R5)
CALL(build_object) | right slide thing: blue

CMOVE(0x9,R1)
CMOVE(0x06F0,R2)
SHLC(R2,12,R2)
CMOVE(0x0A80,R0)
AND(R0,R10,R0)
ADD(R0,R2,R2)
CMOVE(0x0,R3)
CMOVE(0x0,R4)
CMOVE(0x7,R5)
CALL(build_object) | right paddle: white

loop:
  CALL(set_timer)
  CALL(scan)
  CALL(update)
  BR(loop)
```

```
r_bump = R22
l_bump = R24
y_vel = R21
x_vel = R23

scan:
  .breakpoint
  CMOVE(0x1000,R15)
  SHLC(R15,4,R15)   | load R15 with the bottom address of the external mem
  LD(R15,INPUT_PORT_A,R0)

scan_rbump:
  CMOVE(0x0,r_bump)
  ANDC(R0,0x1,R20)
  BEQ(R20, scan_lbump)
    CMOVE(0x1,r_bump)

scan_lbump:
  CMOVE(0x0,l_bump)
  ANDC(R0,0x2,R20)
  BEQ(R20, scan_up)
    CMOVE(0x1,r_bump)

scan_up:
  ANDC(R0,0x4,R20)
  BEQ(R20, scan_rarrow)
    ADDC(y_vel,2,y_vel)

scan_rarrow:
  ANDC(R0,0x8,R20)
  BEQ(R20, scan_down)
    ADDC(x_vel,2,x_vel)

scan_down:
  ANDC(R0,0x10,R20)
  BEQ(R20, scan_larrow)
    SUBC(y_vel,2,y_vel)

scan_larrow:
  ANDC(R0,0x20,R20)
  BEQ(R20, end_scan)
    SUBC(x_vel,1,x_vel)
end_scan:
  RTN()

update:               | process next game object

update_start:
  LD(instance_list,R8)    | load the current offset into the list
  LD(R8,0,R1)              | load SPRITE_ID of next object
  LD(R8,POSITION_OFFSET,R2) | load POSITION of next object
  LD(R8,COLOR_OFFSET,R5)   | load RGB of next object

| packet build routine
  SHLC(R1,27,R17)   | shift SPRITE_ID into upper five bits of R17
  SHLC(R5,24,R0)    | shift RGB into bits [26:24] of R17
  ADD(R17,R0,R17)   | combine
  ADD(R17,R2,R17)   | include POSITION: R17 now contains {SPRITE_ID[4:0], RGB[2:0],
      POSITION_X[11:0], POSITION_Y[11:0]}
```

```
|.breakpoint

  PUSH(LP)            | gotta push LP before you call a function within a function!
  CALL(write_external)
  POP(LP)          | restore LP
  BNE(R1,cont_update,R31)    | if SPRITE_ID != NULL, continue
    CMOVE(instance_list+4,R8) | reset list pointer
    ST(R8,instance_list)
    CMOVE(0x0,R17)
    CALL(write_external)
    BR(update_start)         | start over
cont_update:
  LD(R8,4,R2)               | load coordinates of next object
  SHLC(R1,0x2,R18)         | turn SPRITE_ID into a byte offset
  LD(R18,object_routines,R18) | LD address of object-specific update routine
  JMP(R18, R9)             | JMP to execute; save PC in R9
  ADDC(R8,NEXT_SPRITE_OFFSET,R8)           | point list pointer to next next object
  ST(R8,instance_list)
  RTN()

|+========
pinball:
.breakpoint
LD(R8,8,R3)               | load x and y velocities
LD(R8,12,R4)
PUSH(LP)
CALL(update_pos)
|CALL(detect_collision)
CALL(update_velocity)
POP(LP)
JMP(R9)                     | return to update function

pinball_x:
  LONG(0x0)           | memory for storing pinball x and y
pinball_y:
  LONG(0x0)           | coordinates 32 bit form, w/ last byte as fractional data

|+========
circ_bump:
JMP(R9)                   | return to update function

|+========
board_outline:
JMP(R9)                   | return to update function

|+========
l_tri_bump:
JMP(R9)                   | return to update function

|+========
r_tri_bump:
JMP(R9)                   | return to update function

|+========
l_slide:
JMP(R9)                   | return to update function

|+========
r_slide:
```

```
JMP(R9)                  | return to update function

|+========
l_paddle:
.breakpoint
PUSH(LP)
CALL(paddle_update)
POP(LP)
JMP(R9)                  | return to update function

|+========
r_paddle:
.breakpoint
PUSH(LP)
CALL(paddle_update)
POP(LP)
JMP(R9)                  | return to update function

|=======================================================
| OBJECT ROUTINE LOOKUP
|   index into list using SPRITE_ID*4
|   load address, then JMP to execute update routine
|======================================================

object_routines:
  LONG(0xABCD)
  LONG(pinball)
  LONG(circ_bump)
  LONG(board_outline)
  LONG(l_tri_bump)
  LONG(r_tri_bump)
  LONG(l_slide)
  LONG(r_slide)
  LONG(l_paddle)
  LONG(r_paddle)

|=======================================================
| OBJECT INSTANCE LIST STRUCTURE:
|   0x00 pointer to object list: increment by 8 after
|        processing an object
|   0x04 SPRITE_ID - indicates what class of object we are
|        handling
|   0x08 POSITION - x and y data
|   0x0C VEL_X - x velocity
|   0x10 VEL_Y - y velcoity
|   0x14 RGB - sprite color
|   ...
|   ... list of objects to be processed
|   ...
|   0x4n - NULL
|=======================================================

instance_list: LONG(.+4) | pointer to the object list
  STORAGE(256)           | allocated object memory


|+===========+
| for writing to the memory block shared with the laser projector
|===========
```

```
write_external:
  CMOVE(0x3000,R15)
  SHLC(R15,4,R15)   | load R15 with the bottom address of the external mem
  ADD(R15,R16,R15)  | add offset into memory from R16
  ST(R17,0,R15)     | write the contents of R17 into memory
  ADDC(R16,4,R16)   | increment offset for next time
  BNE(R17,write_end)
    CMOVE(0x0,R16)  | if we're done with the frame, clear offset
write_end:
  RTN()
|+==========+
| builds an object in the instance_list
| before calling, put the appropriate values in R1-5
|***********
build_object:
.breakpoint
  LD(count, R0)
  ADDC(R0,1,R0)              | increment the count for next time
  ST(R0, count)              | store the new count back in memory
  SUBC(R0,1,R0)
  MULC(R0,20,R0)             | turn count in R0 into the offset into the object list
  ST(R1,instance_list+4,R0)  | store object attributes
  ST(R2,instance_list+8,R0)
  ST(R3,instance_list+12,R0)
  ST(R4,instance_list+16,R0)
  ST(R5,instance_list+20,R0)
  ST(R31,instance_list+24,R0) | end list with NULL; gets overwritten by next object
  SHRC(R1,1,R11)    | check to see if this object is the pinball
  BNE(R11,build_done) | if not, we're done
    | if it's the pinball, store its x and y in pinball_xy
    SHRC(R2,0x10,R11)   | x pos
    SHLC(R11,0x8,R11)   | shift left to represent fractional portion
    AND(R2,R10,R12)     | y pos
    SHLC(R12,0x8,R12)
    ST(R11,pinball_x)
    ST(R12,pinball_y)
build_done:
  RTN()
count: LONG(0x0)    | for storing the object instance count
|| Start the timer, wait for it to finish, and clear the flag (have to reload counter to reset)
set_timer:

   | Initialize and reset timer
   CMOVE(1, r11)         | store timer address in r7
   SHLC(r11, 16, r11)
   CMOVE(TIMER_VALUE, r12)       | load for gavlo update frequency
   ST(r12, TIMER_SET, r11)

wait_timer:
   LD(r11, TIMER_OVERFLOW, r12)
|  BNE(r12, wait_timer)     | DEBUG
   BEQ(r12, wait_timer)       | flag should be set when timer is done
   CMOVE(0x0, r12)          | clear flag
   ST(r12, TIMER_OVERFLOW, r11)
   RTN()
update_pos:
  LD(pinball_x,R11)
  LD(pinball_y,R12)
  |ADDC(R4, GRAV_FRACT, R4) | add acceleration due to gravity
```

```
  ADD(R11,R3,R11)     |
  ADD(R12,R4,R12)     | increase position by v*t
  ST(R11,pinball_x)   | store them back in pinball_x and _y
  ST(R12,pinball_y)
  SHRC(R11,0x8,R11)   | shift out fractional byte
  SHLC(R11,0x10,R2)
  SHRC(R12,0x8,R12)
  AND(R12,R10,R12)
  ADD(R12,R2,R2)
  ST(R2,4,R8)         | update R2 with 16 bit approximations
  RTN()
detect_collision:
  PUSH(R8)            | save offset into instance list
  LD(pinball_x,R13)
  LD(pinball_y,R14)
d_outer_loop:
  ADDC(R8,NEXT_SPRITE_OFFSET,R8) | point R8 at the next sprite in the list
  LD(R8,0,t7)        | load next sprite id
  BNE(t7, detect_continue)
    CMOVE(0x0,R0)
    BR(end_detect)
detect_continue:
  SHLC(R20,0x10,t0)  | R22 = O_y
  SRAC(t0,0x10,t0)
  SUB(R14,t0,t0)
  LD(R8,POSITION_OFFSET,t1)
  SRAC(t1,0x10,t1) | R21 = O_x
  SUB(R13,t1,t1)
  SHLC(R19,0x5,R19)  | save R19 as an offset into the table
  LD(R19,sprite_lookup,R11)  | get face offset coordinates
  BEQ(R11, d_outer_loop)    | if NULL, look at the next sprite
end_detect:
  POP(R8)
  RTN()
detect_one:
  |  argument order: Xx, Xy, Radius_squared, Px, Py, Dx, Dy, L
  |  BP - 12 = L
  |  BP - 16 = Dy
  |  BP - 20 = Dx
  |  BP - 24 = Py
  |  BP - 28 = Px
  |  BP - 32 = Radius_squared
  |  BP - 36 = Xy
  |  BP - 40 = Xx
  PUSH(LP)
  PUSH(BP)
  MOVE(SP,BP)
  PUSH(R1)
  PUSH(R2)
  PUSH(R3)
  LD(BP,-40,R0)
  LD(BP,-28,R1)
  SUB(R0,R1,R0)
  LD(BP,-36,R1)
  LD(BP,-24,R2)
  SUB(R1,R2,R1)
  LD(BP,-20,R2)
  MUL(R2,R0,R2)
  LD(BP,-16,R3)
```

```
  MUL(R3,R1,R3)
  ADD(R2,R3,R2)
  MUL(R0,R0,R0)
  MUL(R1,R1,R1)
  ADD(R0,R1,R0)
  SUB(R0,R2,R0) | R0 has distance from line
  SHRC(R2,31,R1)
  BT(R1,detect_one_false)
  LD(BP,-12,R1)
  CMPLT(R1,R2,R1)
  BT(R1,detect_one_false)
  LD(BP,-32,R1)
  CMPLT(R1, R0,R1)
  BT(R1,detect_one_false)
|  call(update_velocity)
  CMOVE(1,R0)
  BR(detect_one_exit)
detect_one_false:
  MOVE(R31,R0)
detect_one_exit:
  POP(R3)
  POP(R2)
  POP(R1)
  MOVE(BP,SP)
  POP(BP)
  POP(LP)
  RTN()
update_velocity:
  ADDC(R3,x_vel,R3)
  ADDC(R4,x_vel,R4)
  ST(R3,8,R8)
  ST(R4,12,R8)
  RTN()
| Sprite lookup tables: one table for each sprite, memory location corresponds to sprite ID
sprite_lookup:
. = sprite_lookup+0x20
LONG(0x00004000)  | RADIUS^2
LONG(0x00000000)  | null termination of segments
. = sprite_lookup+0x40      | arbitrary circle (three times bigger): 2
LONG(0x00009000)           | RADIUS^2
LONG(0x00000000)
. = sprite_lookup+0x60     | the frame outline: 3
LONG(0x0A000000)
LONG(0x00000600)
LONG(0xFEC000C0)   | right side
LONG(0x014000C0)   | pointy bit
LONG(0x00000360)
LONG(0xFBA00120)    |*
LONG(0xFEC00000)    | bottom bit
LONG(0xFBA0FFE0)    |*
LONG(0x0000FCA0)
LONG(0x0140FF40)    | left side
LONG(0xFEC0FF40)    | left side
LONG(0x0000FA00)
LONG(0x00000000)    | NULL means end of sprite!
. = sprite_lookup+0x80
|. = 0x500         | left triangle bumper: 4
LONG(0x01E00180)
LONG(0xFE200000)
```

```
LONG(0x0000FE80)
LONG(0x00000000)
. = sprite_lookup+0xA0
|.= 0x600          | right triangle bumper: 5
LONG(0x00000180)
LONG(0xFE200000)
LONG(0x01E0FE80)
LONG(0x00000000)
. = sprite_lookup+0xC0     | left slidey thing: 6
LONG(0x00000220)
LONG(0x02200080)
LONG(0x00000000)
. = sprite_lookup+0xE0   | right slidey thing: 7
LONG(0x00000220)
LONG(0xFDE00080)
LONG(0x00000000)
|| TODO: figure out paddle representation
. = sprite_lookup+0x100  | left paddle: 8
LONG(0x00180008)
LONG(0x00000000)
. = sprite_lookup+0x120  | right paddle: 9
LONG(0xFFE80008)
LONG(0x00000000)
paddle_update:
   PUSH(r0)
   PUSH(r1)
   PUSH(r2)
   PUSH(r7)
   PUSH(r8)
   CMOVE(0x1, r7) | right paddle mask
   CMOVE(0x2, r8) | left paddle mask

   CMOVE(0x1, r0)
   SHLC(r0, 16, r0)
   LD(r0, 0, r0)   | get value from input port a. lsb is right paddle, second bit is left paddle.

   CMOVE(0x3, r1)
   SHLC(r1, 16, r1)
   ORC(r1, 0x00F8, r3) | left paddle update location
   ORC(r1, 0x00FC, r2) | right paddle update location

   AND(r7, r0, r7)
   BEQ(r7, right_paddle_down)
right_paddle_up:
   LD(right_paddle_up_val, r13)
   ST(r13, 0, r2)
   AND(r8, r0, r8)
   BEQ(left_paddle_down,r8)
   BR(left_paddle_up)
right_paddle_down:
   LD(right_paddle_down_val, r13)
   ST(r13, 0, r2)
   AND(r8, r0, r8)
   BEQ(left_paddle_down,r8)
   BR(left_paddle_up)
left_paddle_up:
   LD(left_paddle_up_val, r13)
   ST(r13, 0, r3)
   POP(r8)
```

```
   POP(r7)
   POP(r2)
   POP(r1)
   POP(r0)
   RTN()

left_paddle_down:
   LD(left_paddle_down_val, r13)
   ST(r13, 0, r3)
   POP(r8)
   POP(r7)
   POP(r2)
   POP(r1)
   POP(r0)
   RTN()
left_paddle_down_val:
LONG(0x00180008)
left_paddle_up_val:
LONG(0x0018FFF8)
right_paddle_down_val:
LONG(0xFFE80008)
right_paddle_up_val:
LONG(0xFFE8FFF8)
stack:
STORAGE(128)
|. = 0x10000
|LONG(0x4)
|. = 0x400000
|LONG(0x1EEB)
```

7.4 Camera Module

```
module camera_full(
   input wire clk_50,
   input wire camera_pclk,
   input wire reset,
   input wire system_start,
   input wire capture_frame,
   input wire process_frame,
   input wire camera_href,
   input wire camera_vsync,
   input wire [7:0] camera_data,
   inout wire camera_scl,
   inout wire camera_sda,
   inout wire vga_scl,
   inout wire vga_sda,
   output wire camera_xclk,
   output wire [11:0] vga_drive,
   output reg vga_hsync,
   output reg vga_vsync,
   output reg vga_blank,
   output wire beta_mwe,
   output wire [31:0] beta_addr,
   output wire [31:0] beta_din
    );
```

```
wire [9:0] vga_hcount; //pixels counted from top left
wire [9:0] vga_vcount;
wire [15:0] vga_addr;

wire [9:0] camera_hcount;
wire [9:0] camera_vcount;
wire [15:0] camera_addr;

wire [9:0] proc_hcount;
wire [9:0] proc_vcount;
wire [15:0] proc_addr;

wire [15:0] buffer_read_addr;

wire local_vsync; //local signals for VGA, need to be delayed
wire local_hsync;
wire local_blank;

wire image_process; //controlls mux for frame buffer


wire [23:0] vga_data;
wire [8:0] vga_pixel;
wire [15:0] camera_pixel; //processed camera pixel
wire mwe_camera;

wire [8:0] frame_data;
wire image_hunt_mem_request;

//locations of detected objects
wire [9:0] red_x, red_y;
wire [9:0] green_x, green_y;
wire [9:0] blue_x, blue_y;
//detected object pixels
wire [8:0] red_blob_pixel;
wire [8:0] green_blob_pixel;
wire [8:0] blue_blob_pixel;
wire [8:0] output_pixel;

wire vga_pixel_invalid;
wire frame_proc_done;

assign output_pixel = vga_pixel_invalid ? 0 : vga_pixel | red_blob_pixel | green_blob_pixel |
    blue_blob_pixel;

assign vga_data = {output_pixel[8:6], {5{output_pixel[6]}},
                  output_pixel[5:3], {5{output_pixel[3]}},
                  output_pixel[2:0], {5{output_pixel[0]}}};

//mux for frame buffer
assign vga_pixel = image_hunt_mem_request ? 9'b000_000_000: frame_data;
assign buffer_read_addr = image_hunt_mem_request ? proc_addr : vga_addr ;

reg blank_temp;
reg hsync_temp;
reg vsync_temp;

//assign vga_vsync = local_vsync;
//assign vga_hsync = local_hsync;
```

```
//assign vga_blank = local_blank;

//solve output delay, delay by two clock cycles
always@(posedge clk_50)
begin
blank_temp <= local_blank;
hsync_temp <= local_hsync;
vsync_temp <= local_vsync;

vga_blank <= blank_temp;
vga_hsync <= hsync_temp;
vga_vsync <= vsync_temp;
end

image_hunt process_a_frame (
.clk(clk_50),
.mem_pixel_data(frame_data),
.start(process_frame),
.mem_request(image_hunt_mem_request),
.done(frame_proc_done),
.mem_hcount(proc_hcount),
.mem_vcount(proc_vcount),
.red_x(red_x),
.red_y(red_y),
.green_x(green_x),
.green_y(green_y),
.blue_x(blue_x),
.blue_y(blue_y)
);



//generate clock for camera
camera_clk clk12_5_gen (
.CLKIN_IN(clk_50),
.RST_IN(0),
.CLKDV_OUT(camera_xclk),
.CLK0_OUT(),
.LOCKED_OUT()
);

//create signals for VGA IC
vga_pixel_drive drive_vga (
.hsync(local_hsync),
.pixel_clk_2x(clk_50),
.pixel_data(vga_data),
.vga_out(vga_drive)
);

//generate pixel count for VGA
vga_drive vga_signal_gen (
.vclock(clk_50),
.hcount(vga_hcount),
.vcount(vga_vcount),
.vsync(local_vsync),
.hsync(local_hsync),
.blank(local_blank)
);
```

```
 //i2c setup for vga
 vga_setup vga_monitor (
 .clk(clk_50),
 .reset(reset),
 .start(system_start),
 .scl(vga_scl),
 .sda(vga_sda)
 );

 //i2c setup for camera
 camera_configure camera_setup(
 .clk(clk_50),
 .reset(reset),
 .start(system_start),
 .scl(camera_scl),
 .sda(camera_sda)
 );

//capture a frame from camera and dump to memory
camera_capture get_a_frame (
 .camera_clk(camera_pclk),
 .start(capture_frame),
 .camera_vsync(camera_vsync), //input from camera
 .camera_href(camera_href), //input from camera
 .camera_dout(camera_data), //input data from camera
 .mem_we(mwe_camera), //mwe for frame buffer
 .mem_din(camera_pixel), //data to memory for frame buffer
 .vcount(camera_vcount), // used to generate memory address
 .hcount(camera_hcount), //used to generate camera memory address
 .done() // done signal
 );

 image_addr_gen camera_addr_gen (
 .hcount(camera_hcount),
 .vcount(camera_vcount),
 .addr(camera_addr),
 .blank()
 );

 image_addr_gen buffer_read_addr_gen (
 .hcount(vga_hcount),
 .vcount(vga_vcount),
 .addr(vga_addr),
 .blank(vga_pixel_invalid)
 );

image_addr_gen image_proc_gen(
 .hcount(proc_hcount),
 .vcount(proc_vcount),
 .addr(proc_addr),
 .blank()
 );


//240x240 9 bit color
//image is half resolution
shared_video_mem frame_buff (
```

```
    .clk_read(clk_50),
    .clk_write(camera_pclk),
    .write_addr(camera_addr),
    .read_addr(buffer_read_addr),
    .dout_camera(camera_pixel),
    .mwe_camera(mwe_camera),
    .dout_pixel(frame_data)
    );


   camera_write_to_beta beta_dump (
    .clk(clk_50),
    .start(frame_proc_done),
    .red_x(red_x),
    .red_y(red_y),
    .green_x(green_x),
    .green_y(green_y),
    .blue_x(blue_x),
    .blue_y(blue_y),
    .beta_addr(beta_addr),
    .beta_data(beta_din),
    .beta_mwe(beta_mwe)
    );

   color_blob red_blob (
    .clk(clk_50),
    .hcount(vga_hcount),
    .vcount(vga_vcount),
    .x_loc(red_x),
    .y_loc(red_y),
    .enable(1'b1),
    .color(9'b111_000_000),
    .pixel(red_blob_pixel)
    );


color_blob green_blob (
    .clk(clk_50),
    .hcount(vga_hcount),
    .vcount(vga_vcount),
    .x_loc(green_x),
    .y_loc(green_y),
    .enable(1'b1),
    .color(9'b000_111_000),
    .pixel(green_blob_pixel)
    );

color_blob blue_blob (
    .clk(clk_50),
    .hcount(vga_hcount),
    .vcount(vga_vcount),
    .x_loc(blue_x),
    .y_loc(blue_y),
    .enable(1'b1),
    .color(9'b000_000_111),
    .pixel(blue_blob_pixel)
    );
```

```
endmodule
```

7.4.1 Find red, green, and blue loci in a frame

```
`timescale 1ns / 1ps
//processes a 240x240 pixel frame
//outputs locus of red, green, and blue
module image_hunt(
   input wire clk,
   input wire [8:0] mem_pixel_data,
   input wire start,
   output reg mem_request = 0,
   output reg done = 0,
   output wire [9:0] mem_hcount,
   output wire [9:0] mem_vcount,
   output reg [9:0] red_x = 0,
   output reg [9:0] red_y = 0,
   output reg [9:0] green_x = 0,
   output reg [9:0] green_y = 0,
   output reg [9:0] blue_x = 0,
   output reg [9:0] blue_y = 0
    );

    parameter block_max = 58; //number of blocks minus 1

    reg [3:0] FSM_state = 0;

    //storage for cost function values
    reg [31:0] red_cost;
    reg [31:0] green_cost;
    reg [31:0] blue_cost;

    reg [31:0] red_cost_max;
    reg [31:0] green_cost_max;
    reg [31:0] blue_cost_max;

    reg [7:0] block_x = 0;
    reg [7:0] block_y = 0; //current image segment

    wire [8:0] pixel_0, pixel_1, pixel_2, pixel_3;
    wire [8:0] pixel_4, pixel_5, pixel_6, pixel_7;
    wire [8:0] pixel_8, pixel_9, pixel_a, pixel_b;
    wire [8:0] pixel_c, pixel_d, pixel_e, pixel_f;

    reg start_fetch = 0;

    //pipeline values

    reg[7:0] super_pixel_red;
    reg[7:0] super_pixel_green;
    reg[7:0] super_pixel_blue;

    wire load_done;

    always@(posedge clk)
```

```
begin

   case(FSM_state)

   0: begin //reset and wait for start
   FSM_state <= start ? 1 : 0;
   mem_request <= start ? 1 : 0;
   done <= 0;

   block_x <= 2;
   block_y <= 2;

   red_cost <= 0;
   green_cost <= 0;
   blue_cost <= 0;

   red_cost_max <= 0;
   green_cost_max <= 0;
   blue_cost_max <= 0;
   end

   1: begin //fetch_data;
   FSM_state <= 2;
   start_fetch <= 1;
   end

   2: begin //wait for memory read
   FSM_state <= load_done ? 3 : 2;
   start_fetch <= 0;
   end

   3: begin //process data //I think I should limit the number of cascaded adds?
   FSM_state <= 4;
   super_pixel_red <= pixel_0[8:6]+pixel_1[8:6]+pixel_2[8:6]+pixel_3[8:6]
   +pixel_4[8:6]+pixel_5[8:6]+pixel_6[8:6]+pixel_7[8:6]
   +pixel_8[8:6]+pixel_9[8:6]+pixel_a[8:6]+pixel_b[8:6]
   +pixel_c[8:6]+pixel_d[8:6]+pixel_e[8:6]+pixel_f[8:6];

   super_pixel_green <= pixel_0[5:3]+pixel_1[5:3]+pixel_2[5:3]+pixel_3[5:3]
   +pixel_4[5:3]+pixel_5[5:3]+pixel_6[5:3]+pixel_7[5:3]
   +pixel_8[5:3]+pixel_9[5:3]+pixel_a[5:3]+pixel_b[5:3]
   +pixel_c[5:3]+pixel_d[5:3]+pixel_e[5:3]+pixel_f[5:3];

   super_pixel_blue <= pixel_0[2:0]+pixel_1[2:0]+pixel_2[2:0]+pixel_3[2:0]
   +pixel_4[2:0]+pixel_5[2:0]+pixel_6[2:0]+pixel_7[2:0]
   +pixel_8[2:0]+pixel_9[2:0]+pixel_a[2:0]+pixel_b[2:0]
   +pixel_c[2:0]+pixel_d[2:0]+pixel_e[2:0]+pixel_f[2:0];
   end

   4: begin //process data
   FSM_state <= 5;
   red_cost <= ((super_pixel_red>super_pixel_green) && (super_pixel_red>super_pixel_blue)) ?
   (super_pixel_red-super_pixel_green)*(super_pixel_red-super_pixel_blue) : 0;

   green_cost <= ((super_pixel_green>super_pixel_red) && (super_pixel_green>super_pixel_blue)) ?
   (super_pixel_green-super_pixel_red)*(super_pixel_green-super_pixel_blue) : 0;

   blue_cost <= ((super_pixel_blue>super_pixel_green) && (super_pixel_blue>super_pixel_red)) ?
   (super_pixel_blue-super_pixel_red)*(super_pixel_blue-super_pixel_green) : 0;
```

```
    end

    5: begin //process results of block
    FSM_state <= (block_y >= block_max) && (block_x >= block_max) ? 6 : 1;

    red_cost_max <= (red_cost > red_cost_max) ? red_cost : red_cost_max;
    green_cost_max <= (green_cost > green_cost_max) ? green_cost : green_cost_max;
    blue_cost_max <= (blue_cost > blue_cost_max) ? blue_cost : blue_cost_max;

    red_x <= (red_cost >= red_cost_max) ? (block_x<<3) : red_x;
    red_y <= (red_cost >= red_cost_max) ? (block_y<<3) : red_y;

    green_x <= (green_cost >= green_cost_max) ? (block_x<<3) : green_x;
    green_y <= (green_cost >= green_cost_max) ? (block_y<<3) : green_y;

    blue_x <= (blue_cost >= blue_cost_max) ? (block_x<<3) : blue_x;
    blue_y <= (blue_cost >= blue_cost_max) ? (block_y<<3) : blue_y;

    //go x, then y,
    block_x <= (block_x >= block_max) ? 0 : block_x+1;
    block_y <= (block_x >= block_max) ? block_y+1 : block_y;

    //reset
    red_cost <= 0;
    green_cost <= 0;
    blue_cost <= 0;
    end

    6:begin //end state
    FSM_state <= 0;
    done <= 1;
    end

    endcase
    end


load_pixel_block data_fetch (
 .clk(clk),
 .start(start_fetch),
 .block_x(block_x),
 .block_y(block_y),
 .memory_in(mem_pixel_data),
 .mem_hcount(mem_hcount),
 .mem_vcount(mem_vcount),
 .done(load_done),
 .pixel_0(pixel_0),
 .pixel_1(pixel_1),
 .pixel_2(pixel_2),
 .pixel_3(pixel_3),
 .pixel_4(pixel_4),
 .pixel_5(pixel_5),
 .pixel_6(pixel_6),
 .pixel_7(pixel_7),
 .pixel_8(pixel_8),
 .pixel_9(pixel_9),
 .pixel_a(pixel_a),
 .pixel_b(pixel_b),
 .pixel_c(pixel_c),
```

```
    .pixel_d(pixel_d),
    .pixel_e(pixel_e),
    .pixel_f(pixel_f)
    );


      endmodule
```

7.4.2 Create signals for VGA IC

```
//this contains some trick logic delays
module vga_pixel_drive(
   input wire hsync,
   input wire pixel_clk_2x, //50 MHz
   input wire [23:0] pixel_data,
   output reg [11:0] vga_out
    );

    reg pixel_ab; //0 for a, 1 for b
    reg hsync_last = 0;
    reg [23:0] stored_pixel;
    reg [11:0] pixel_b;


    always@(posedge pixel_clk_2x)
    begin
       hsync_last <= hsync;

      pixel_ab <= ((hsync_last==0)&&(hsync==1)) ? 1 : ~pixel_ab;

      //latch input data on pixel_ab = a
      pixel_b <= pixel_data[11:0];
      vga_out <= pixel_ab ? pixel_b : pixel_data[23:12];
    end

endmodule
```

7.4.3 VGA Drive for Lower Resolution

```
//taken from 6.111 lab and altered for lower resolution
//need to make this double the vclock
module vga_drive(
            input vclock, //twice the pixel clock
            output wire [9:0] hcount, // pixel number on current line
            output wire [9:0] vcount, // line number
            output reg vsync = 0,
            output reg hsync = 0,
            output reg blank = 0
            );

   // horizontal: 800 pixels total
   // display 640 pixels per line
```

```
  reg hblank = 0;
  reg vblank = 0;
  reg [10:0] vcount_d = 0;
  reg [10:0] hcount_d = 0;
  assign vcount = vcount_d[10:1];
  assign hcount = hcount_d[10:1];


  wire hsyncon,hsyncoff,hreset,hblankon;

  assign hblankon = (hcount_d == (639*2)); //blank after display width
  assign hsyncon = (hcount_d == (655*2)); // active video + front porch
  assign hsyncoff = (hcount_d == (751*2)); //active video + front portch + sync
  assign hreset = (hcount_d == (799*2)); //plus back porch

  // vertical: 806 lines total
  // display 480 lines
  wire vsyncon,vsyncoff,vreset,vblankon;
  assign vblankon = hreset & (vcount_d == (479*2));
  assign vsyncon = hreset & (vcount_d == (489*2));
  assign vsyncoff = hreset & (vcount_d == (491*2));
  assign vreset = hreset & (vcount_d == (524*2));

  // sync and blanking
  wire next_hblank,next_vblank;
  assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
  assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
  always @(posedge vclock) begin
    hcount_d <= hreset ? 0 : hcount_d + 1;
    hblank <= next_hblank;
    hsync <= hsyncon ? 0 : hsyncoff ? 1 : hsync; // active low

    vcount_d <= hreset ? (vreset ? 0 : vcount_d + 2) : vcount_d;
    vblank <= next_vblank;
    vsync <= vsyncon ? 0 : vsyncoff ? 1 : vsync; // active low

    blank <= next_vblank | (next_hblank & ~hreset);
  end
endmodule
```

7.4.4 I2C Setup for VGA

```
module vga_setup(
  input wire clk,
  input wire reset, //reset on signal going high
  input wire start,
  inout scl,
  inout sda
   );

   parameter device_id = 8'hEC;

  // reg [7:0] write_id;
   reg [7:0] write_reg;
   reg [7:0] write_data;
   reg [7:0] write_id;
```

```
reg [4:0] FSM_state = 0;

wire done;

reg start_write = 0;



i2c_configure_reg instance_name (
.clk(clk),
.reset(reset),
.start(start_write),
.write_reg(write_reg),
.write_id(write_id),
.write_data(write_data),
.done(done),
.scl(scl),
.sda(sda)
);

always@(posedge clk)
begin
case(FSM_state)

  0: begin //start
  FSM_state <= start ? 1 : 0;
  start_write <= 0;
  end

  1: begin //configure clk mode reg
  FSM_state <= 2;
  write_id <= device_id;
  write_reg <= 8'h9C; //clk setup
  //write_data <= 8'h00; //clk is at pixel frequency
  write_data <= 8'h01; //clk is at double pixel frequency
  start_write <= 1;
  end

  2: begin //wait for finish
  FSM_state <= done ? 3 : 2;
  start_write <= 0;
  end

  3: begin //input data format
  FSM_state <= 4;
  write_id <= device_id;
  write_reg <= 8'h9F; //input data format
  write_data <= 8'b1000_0000; //Vsync, Hsync are active low
  start_write <= 1;
  end

  4: begin //wait for finish
  FSM_state <= done ? 5 : 4;
  start_write <= 0;
  end

  5: begin //VGA bypass mode
  FSM_state <= 6;
  write_id <= device_id;
```

```
    write_reg <= 8'hA1; //input data format
    write_data <= 8'b0000_1001; //enable sync outputs, VGA bypass enabled
    start_write <= 1;
    end

    6: begin //wait for finish
    FSM_state <= done ? 7 : 6;
    start_write <= 0;
    end

    7: begin //power enable
    FSM_state <= 8;
    write_id <= device_id;
    write_reg <= 8'hc9; //power register
    write_data <= 8'b0000_0000; //negate FPD
    start_write <= 1;
    end


    8: begin //wait for finish
    FSM_state <= done ? 9 : 8;
    start_write <= 0;
    end

    9: begin //return to start
    FSM_state <= 0;
    end
    endcase
    end

  endmodule
```

7.4.5 Camera Configuration

```
module camera_configure(
input wire clk,
  input wire reset, //reset on signal going high
  input wire start,
  inout scl,
  inout sda
   );

   parameter device_id = 8'h42; //camera address for writes
   parameter rom_locations = 30;

   wire [7:0] write_reg;
   wire [7:0] write_data;
   reg [4:0] FSM_state = 0;

   wire done;

   reg start_write = 0;
   reg [5:0] rom_addr = 0;


   i2c_configure_reg camera_i2c (
```

```
.clk(clk),
.reset(reset),
.start(start_write),
.write_reg(write_reg),
.write_id(device_id),
.write_data(write_data),
.done(done),
.scl(scl),
.sda(sda)
);

camera_setup_rom config_rom (
.clk(clk),
.addr(rom_addr),
.register(write_reg),
.value(write_data)
);


always@(posedge clk)
begin
case(FSM_state)

0: begin //start
FSM_state <= start ? 1 : 0;
start_write <= 0;
rom_addr <= 0;
end

1: begin //check rom value, give time for rom load
FSM_state <= (rom_addr>=rom_locations) ? 5 : 2;
end

2: begin //start I2C
FSM_state <= 3;
start_write <= 1;
end

3: begin //wait for end
FSM_state <= done ? 4 : 3;
end

4: begin
FSM_state <= 1;
rom_addr <= rom_addr +1;
end

5: begin //end state
FSM_state <= 0;
end
endcase
end

endmodule
```

7.4.6 Capture a frame from the camera and dump into memory

```
//takes 12.5Mhz camera clk, writes to memory, then frees up memory
//wrapper for camera
//stored as r[2:0]g[2:0]b[1:0]
module camera_capture(
   input wire camera_clk,
   input wire start,
   input wire camera_vsync,
   input wire camera_href,
   input wire [7:0] camera_dout,
   output reg mem_request = 0,
   output reg mem_we =0,
   output reg [15:0] mem_din =0,
   output reg [9:0] vcount,
   output reg [9:0] hcount,
   output reg done = 0
   );


   reg [3:0] FSM_state = 0;
   reg camera_start = 0;


   wire [15:0] pixel_data;
   wire pixel_done;
   wire camera_done;
   wire [9:0] camera_vcount;
   wire [9:0] camera_hcount;

   camera_read camera_reader (
    .p_clock(camera_clk),
    .vsync(camera_vsync),
    .href(camera_href),
    .p_data(camera_dout),
    .start(camera_start),
    .pixel_data(pixel_data),
    .pixel_done(pixel_done),
    .done(camera_done),
    .hcount(camera_hcount),
    .vcount(camera_vcount)
    );


    always@(posedge camera_clk)
    begin

    case(FSM_state)

    0: begin //start state
    FSM_state <= start ? 1 : 0;
    done <= 0;
    end

    1: begin //start camera read
    FSM_state <= 2;
    camera_start <= 1;
    mem_request <= 1;
    hcount <=0;
    vcount <=0;
```

```
 mem_we <= 0;
 end

 2: begin //wait
 FSM_state <= camera_done ? 3:2;
 camera_start <= 0;
 mem_we <= pixel_done ? 1 :0;
 //mem_din <= {pixel_data[14:12],pixel_data[9:7], pixel_data[4:3]};
 mem_din <= pixel_data;
 hcount <= camera_hcount;
 vcount <= camera_vcount;
 end

 3: begin //free up memory
 FSM_state <= 0;
 mem_request <= 0;
 mem_we <= 0;
 done <= 1;
 end
 endcase
 end

endmodule
```

7.4.7 Write object locations to the physics Beta

```
module camera_write_to_beta(
   input wire clk,
   input wire start,
   input wire [9:0] red_x,
   input wire [9:0] red_y,
   input wire [9:0] green_x,
   input wire [9:0] green_y,
   input wire [9:0] blue_x,
   input wire [9:0] blue_y,
   output reg [31:0] beta_addr =0,
   output reg [31:0] beta_data =0,
   output reg beta_mwe = 0
    );


   reg [5:0] FSM_state = 0;


   always@(posedge clk)
   begin
   case(FSM_state)

   0: begin //wait for start signal

   FSM_state <= start ? 1 : 0;
   end

   1: begin //write red
   FSM_state <= 2;
   beta_addr <= 0;
```

```
  beta_data <= {6'b0, red_x, 6'b0, red_y};
  beta_mwe <= 1;
  end

  2: begin //write green
  FSM_state <= 3;
  beta_addr <= 1;
  beta_data <= {6'b0, green_x, 6'b0, red_y};
  beta_mwe <= 1;
  end

  3: begin //write blue
  FSM_state <= 4;
  beta_addr <= 2;
  beta_data <= {6'b0, blue_x, 6'b0, blue_y};
  beta_mwe <= 1;
  end

  4: begin //write status register
  FSM_state <= 5;
  beta_addr <= 32'b100;
  beta_data <= 1; //done!
  beta_mwe <= 1;
  end

  5: begin //end state
  FSM_state <= 0;
  beta_mwe <= 0;
  end
  endcase
  end

endmodule
```

7.4.8 Pixel cost function

```
module pixel_cost_function(
  input [8:0] pixel_data,
  output [8:0] pixel_red_cost,
  output [8:0] pixel_green_cost,
  output [8:0] pixel_blue_cost
   );

  wire [2:0] red = pixel_data[8:6];
  wire [2:0] green = pixel_data[5:3];
  wire [2:0] blue = pixel_data[2:0];


  assign pixel_red_cost = ((red>(green<<1)) && (red>(blue<<1)) && (red>(blue+red)) && (red>2)) ?
                        (red<<2)-(green<<1)-(blue<<1) : 0;

  assign pixel_blue_cost = ((blue>(green<<1)) && (blue>(red<<1)) && (blue>(red+green)) &&
      (blue>2)) ?
                        (blue<<2) - (green<<1) - (red<<1) : 0;
```

```
  assign pixel_green_cost = ((green>(blue<<1)) && (green>(red<<1)) && (green>(red+blue))
      &&(green>2)) ?
                        (green<<2) - (red<<1) - (blue<<1) : 0;


endmodule
```

7.5 SNES Controller Interface

```
module nes_interface(
   input wire clk_50,
   input wire controller_dout,
   output reg controller_clk,
   output reg controller_latch,
   output reg updated,
   output reg button_A,
   output reg button_B,
   output reg button_X,
   output reg button_Y,
   output reg button_L,
   output reg button_R,
   output reg button_N,
   output reg button_E,
   output reg button_S,
   output reg button_W,
   output reg button_START,
   output reg button_SELECT
    );


    parameter timer_state = 36;
    parameter clk_period = 300;
    parameter update_delay = 30000;
    reg [31:0] counter = 0;
    reg [5:0] FSM_state_return = 0;
    reg [5:0] FSM_state = 0;


    //sample data on falling clock edge
    always@(posedge clk_50)
    begin
    case(FSM_state)

    0: begin //start state
    FSM_state <= 1;
    updated <= 0;
    end

    1: begin //set latch and set timer
    FSM_state <= timer_state;
    FSM_state_return <= 2;
    controller_latch <= 1;
    counter <= 2*clk_period; //12us
    end
```

```
2: begin //latch goes low
FSM_state <= timer_state;
FSM_state_return <= 3;
controller_latch <= 0;
counter <= clk_period;
end

3: begin //latch B and clk go low
FSM_state <= timer_state;
FSM_state_return <= 4;
controller_clk <= 0;
button_B <= ~controller_dout;
counter <= clk_period;
end

4: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 5;
controller_clk <= 1;
counter <= clk_period;
end

5: begin //latch Y, clk go low
FSM_state <= timer_state;
FSM_state_return <= 6;
controller_clk <= 0;
button_Y <= ~controller_dout;
counter <= clk_period;
end

6: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 7;
controller_clk <= 1;
counter <= clk_period;
end

7: begin //latch SELECT, clk go low
FSM_state <= timer_state;
FSM_state_return <= 8;
controller_clk <= 0;
button_SELECT <= ~controller_dout;
counter <= clk_period;
end

8: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 9;
controller_clk <= 1;
counter <= clk_period;
end

9: begin //latch START, clk go low
FSM_state <= timer_state;
FSM_state_return <= 10;
controller_clk <= 0;
button_START <= ~controller_dout;
counter <= clk_period;
end
```

```
10: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 11;
controller_clk <= 1;
counter <= clk_period;
end

11: begin //latch N, clk go low
FSM_state <= timer_state;
FSM_state_return <= 12;
controller_clk <= 0;
button_N <= ~controller_dout;
counter <= clk_period;
end

12: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 13;
controller_clk <= 1;
counter <= clk_period;
end

13: begin //latch S, clk go low
FSM_state <= timer_state;
FSM_state_return <= 14;
controller_clk <= 0;
button_S <= ~controller_dout;
counter <= clk_period;
end

14: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 15;
controller_clk <= 1;
counter <= clk_period;
end

15: begin //latch W, clk go low
FSM_state <= timer_state;
FSM_state_return <= 16;
controller_clk <= 0;
button_W <= ~controller_dout;
counter <= clk_period;
end

16: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 17;
controller_clk <= 1;
counter <= clk_period;
end

17: begin //latch E, clk go low
FSM_state <= timer_state;
FSM_state_return <= 18;
controller_clk <= 0;
button_E <= ~controller_dout;
counter <= clk_period;
```

```
end

18: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 19;
controller_clk <= 1;
counter <= clk_period;
end

19: begin //latch A, clk go low
FSM_state <= timer_state;
FSM_state_return <= 20;
controller_clk <= 0;
button_A <= ~controller_dout;
counter <= clk_period;
end

20: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 21;
controller_clk <= 1;
counter <= clk_period;
end

21: begin //latch X, clk go low
FSM_state <= timer_state;
FSM_state_return <= 22;
controller_clk <= 0;
button_X <= ~controller_dout;
counter <= clk_period;
end

22: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 23;
controller_clk <= 1;
counter <= clk_period;
end

23: begin //latch L, clk go low
FSM_state <= timer_state;
FSM_state_return <= 24;
controller_clk <= 0;
button_L <= ~controller_dout;
counter <= clk_period;
end

24: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 25;
controller_clk <= 1;
counter <= clk_period;
end

25: begin //latch R, clk go low
FSM_state <= timer_state;
FSM_state_return <= 26;
controller_clk <= 0;
button_R <= ~controller_dout;
```

```
counter <= clk_period;
end

26: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 27;
controller_clk <= 1;
counter <= clk_period;
end

27: begin //clock go low //empty 1
FSM_state <= timer_state;
FSM_state_return <= 28;
controller_clk <= 0;
counter <= clk_period;
end

28: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 29;
controller_clk <= 1;
counter <= clk_period;
end

29: begin //clock go low //empty 2
FSM_state <= timer_state;
FSM_state_return <= 30;
controller_clk <= 0;
counter <= clk_period;
end

30: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 31;
controller_clk <= 1;
counter <= clk_period;
end

31: begin //clock go low //empty 3
FSM_state <= timer_state;
FSM_state_return <= 32;
controller_clk <= 0;
counter <= clk_period;
end

32: begin //clock go high
FSM_state <= timer_state;
FSM_state_return <= 33;
controller_clk <= 1;
counter <= clk_period;
end

33: begin //clock go low //empty 4
FSM_state <= timer_state;
FSM_state_return <= 34;
controller_clk <= 0;
counter <= clk_period;
end
```

```
    34: begin //clock go high
    FSM_state <= timer_state;
    FSM_state_return <= 35;
    controller_clk <= 1;
    counter <= clk_period;
    end

    35: begin //wait for next update interval
    FSM_state <= timer_state;
    FSM_state_return <= 0;
    counter <= update_delay;
    end

    timer_state: begin //count down and return to stored state
    FSM_state <= (counter == 0) ? FSM_state_return : timer_state;
    counter <= (counter == 0) ? 0 : counter - 1;
    end
    endcase
    end



endmodule
```

7.6 Pulling it all together

```
module laser_projector_full(
   input wire clk,
   input wire reset,
   input wire [7:0] dip_sw,
   input wire paddle_l,
   input wire paddle_r,
   input wire up_button,
   input wire down_button,
   input wire left_button,
   input wire right_button,
   input wire [31:0] camera_addr,
   input wire [31:0] camera_dout,
   input wire camera_mwe,
   output wire [2:0] laser_rgb,
   output wire dac_mosi,
   output wire dac_csn,
   output wire dac_latchn,
   output wire dac_sclk,
   output wire [7:0] debug_led
    );


    parameter xadr_physics = 31'h04;
    parameter xadr_laser = 31'h04;

   //wires for physics beta
   wire mwe_physics;
   wire irq_physics;
   wire [31:0] ma_physics;
   wire [31:0] mdin_physics;
```

```
wire [31:0] mdout_physics;
//memory mapped devices for physics beta
wire [31:0] ram_dout_physics;
wire [31:0] IO_dout_physics;
wire [31:0] shared_read_dout_physics;
wire [31:0] shared_write_dout_physics;
//selects for multiplexing the buss
wire [2:0] read_select_physics;
wire sel_ram_physics;
wire sel_IO_physics;
wire sel_read_shared_physics;
wire sel_write_shared_physics;


//wires for laser beta
wire mwe_laser;
wire irq_laser;
wire [31:0] ma_laser;
wire [31:0] mdin_laser;
wire [31:0] mdout_laser;
//memory mapped devices for laser beta
wire [31:0] ram_dout_laser;
wire [31:0] IO_dout_laser;
wire [31:0] shared_read_dout_laser;
wire [31:0] shared_write_dout_laser;
//selects for multiplexing the buss for laser beta
wire [2:0] read_select_laser;
wire sel_ram_laser;
wire sel_IO_laser;
wire sel_read_shared_laser;
wire sel_write_shared_laser;

wire [31:0] IO_port;
assign debug_led = IO_port[7:0];

//memory mdin is data into beta
//memory mdout is data from beta to be written

beta2 cpu_physics(
.clk(clk),
.reset(reset),
.irq(irq_physics),
.xadr(xadr_physics),
.ma(ma_physics),
.mdin(mdin_physics),
.mdout(mdout_physics),
.mwe(mwe_physics)
);

//16k mem
//14 bits wide, takes beta [15:2] because memory is byte aligned
physics_beta_mem physics_beta_ram (
.clka(clk),
.dina(mdout_physics),
.addra(ma_physics[15:2]),
.wea(mwe_physics && sel_ram_physics && (~reset)),
.douta(ram_dout_physics)
);
```

```
//decode data adress for reads
beta_addr_decode physics_decode (
.addr(ma_physics),
.sel_ram(sel_ram_physics),
.sel_IO(sel_IO_physics),
.sel_read_shared(sel_read_shared_physics),
.sel_write_shared(sel_write_shared_physics),
.read_select(read_select_physics)
);

//select mdin for beta
beta_read_decode physics_read_decode (
.clk(clk),
.reset(reset),
.read_select(read_select_physics),
.ram_dout(ram_dout_physics),
.IO_dout(IO_dout_physics),
.shared_read_dout(shared_read_dout_physics),
.shared_write_dout(shared_write_dout_physics),
.beta_mdin(mdin_physics)
);

//memory mapped IO for physics beta
mapped_IO physics_beta_IO (
 .clk(clk),
 .addr(ma_physics),
 .din(mdout_physics),
 .mwe(mwe_physics & sel_IO_physics),
 .in_port_a({26'b0,left_button,down_button,right_button,up_button,paddle_l,paddle_r}),
 .in_port_b({24'b0,dip_sw}),
 .dout(IO_dout_physics),
 .out_port_a(),
 .out_port_b(),
 .spi_miso(1'b0),
 .spi_sclk(),
 .spi_mosi(),
 .irq(irq_physics)
 );

//interface between camera fsm(host) and physics_beta (client)
memory_share physics_shared_memory (
 .clk(clk),
 .mwe_host(camera_mwe), //camera we
 .mwe_client(mwe_physics&&sel_read_shared_physics),
 .host_addr(camera_addr), //connection to camera FSM
 .client_addr(ma_physics),
 .host_din(camera_dout), //dout of camera
 .client_din(mdout_physics), //input to memory, output of beta
 .host_dout(), //no camera yet
 .client_dout(shared_read_dout_physics) //output from memory
 );


//laser beta
beta2 cpu_laser(
.clk(clk),
.reset(reset),
.irq(irq_laser),
.xadr(xadr_laser),
```

```
.ma(ma_laser),
.mdin(mdin_laser),
.mdout(mdout_laser),
.mwe(mwe_laser)
);

//16k mem
//14 bits wide, takes beta [15:2] because memory is byte aligned
hardware_beta_mem laser_beta_ram (
.clka(clk),
.dina(mdout_laser),
.addra(ma_laser[15:2]),
.wea(mwe_laser&&sel_ram_laser&&(~reset)),
.douta(ram_dout_laser)
);

//decode memory addresses
beta_addr_decode laser_decode (
.addr(ma_laser),
.sel_ram(sel_ram_laser),
.sel_IO(sel_IO_laser),
.sel_read_shared(sel_read_shared_laser),
.sel_write_shared(sel_write_shared_laser),
.read_select(read_select_laser)
);


//write data back
beta_read_decode laser_read_decode (
.clk(clk),
.reset(reset),
.read_select(read_select_laser),
.ram_dout(ram_dout_laser),
.IO_dout(IO_dout_laser),
.shared_read_dout(shared_read_dout_laser),
.shared_write_dout(shared_write_dout_laser),
.beta_mdin(mdin_laser)
);

//memory mapped IO for laser beta
mapped_IO laser_beta_IO (
 .clk(clk),
 .addr(ma_laser),
 .din(mdout_laser),
 .mwe(mwe_laser & sel_IO_laser),
 .in_port_a({24'b0,dip_sw}),
 .in_port_b(32'b0),
 .dout(IO_dout_laser),
 .out_port_a({dac_csn,dac_latchn, laser_rgb}),
 .out_port_b(IO_port),
 .spi_miso(1'b0),
 .spi_sclk(dac_sclk),
 .spi_mosi(dac_mosi),
 .irq(irq_laser)
 );

//laser_beta is client, physics_beta is host
memory_share laser_shared_memory (
 .clk(clk),
```

```
    .mwe_host(mwe_physics&&sel_write_shared_physics),
    .mwe_client(mwe_laser&&sel_read_shared_laser),
    .host_addr(ma_physics),
    .client_addr(ma_laser),
    .host_din(mdout_physics), //input to memory out of beta
    .client_din(mdout_laser),
    .host_dout(shared_write_dout_physics),
    .client_dout(shared_read_dout_laser)
    );

endmodule
```