18-545 Design Document

Team: It's On

Andrew Snook

Mark Williams

Table of Contents

Background Information

Purpose

The goal of this project is to produce an emulated version of the original 1981 Donkey Kong arcade game. This is one of the games that initially launched Nintendo into a leading position in the gaming market. We intend to create the game using the MAME (Multiple Arcade Machine Emulator) as our source, and will use the game roms it provides as original game source material.

History

Donkey Kong is an arcade game released in 1981 that was a first in two ways for Nintendo: it was the first time they tried to break into the North American market, and it was the first project led by Shigeru Miyamoto, who was then a first-time game designer. This also means that Donkey Kong is a very monumental milestone for Nintendo in more ways than one. First, it was the beginning of the Mario and Donkey Kong series that still are immensely popular today. Second, it was a very successful first step for Miyamoto, who to this day remains one of the most important creative minds for Nintendo.

The game's aesthetic is somewhat influenced by Popeye, since at the time Nintendo was trying to get a license to produce a Popeye comic strip. This never happened, but the love triangle between Jumpman, Donkey Kong, and Pauline is a nod to a similar condition in the Popeye cartoons.

For its time, the game was very complex in terms of animations. The features of Jumpman were chosen very carefully to minimize the number of pixels it would take to animate him while still making it recognizable what he's doing. It was intended to be marketed in North America, so it was given a name that has meaning in English - Donkey Kong is allegedly what they came up with to convey something along the lines of "stupid ape."

Despite many people thinking Donkey Kong a strange game with a strange name, it saw great success for Nintendo, as one might expect from the fact that the franchises it spawned are still going strong today. Donkey Kong remained Nintendo's top-selling arcade game for two years straight and is estimated to have earned Nintendo around $280 million during those two years.

Motivation

We decided on this project because it fit the scope that we were aiming for (arcade game from the early 80s) and was a Nintendo product. Both of us are Nintendo fans, so Donkey Kong was naturally the first game that comes to mind when you think of arcade games. It launched the Mario and Donkey Kong franchises so we figured that it would be a great game to try and create an emulator for, so we did just that.

System Implementation

Z80 CPU

Our CPU is taken from OpenCores.com's TV80 fpga validated model. One of the reasons we thought this project was feasible was because the main cpu core implementation was already completed and validated on an FPGA. We also wrote and tested our own source in order to initially validate and learn how to use the core which consisted of simple load and store instructions that displayed to the attached LCD. At this point we are fairly confident that the core works and can interface with our predefined block rams and roms. The next steps we are taking include hooking up the donkey kong roms and rams to the CPU and validating it with our source material. These tests were run at the fpga clock speed of 100MHz, which is about 30 times faster than we need and we will be scaling down to about 3-4 MHz for the actual game.

Our current conclusion about the core, before testing our source, is that the core will not require any modifications as it can run compiled z80 source as well as works with the rams that we can generate via coregen and normal verilog memory constructs.

Memory Interface

The memory interface is what will be used to integrate all of the pieces together into one functional game. It also allows us to clock each unit (video, user interface, sound, cpu) at different speeds and not have them care about the state of each other unit, besides that it is writing to RAM. Our interface to each unit through memory is shown below. Every piece of ram memory is accessible by the CPU through the main memory bus, at the moment everything else is treated as a true dual port RAM, and each other unit will interface via memory access.

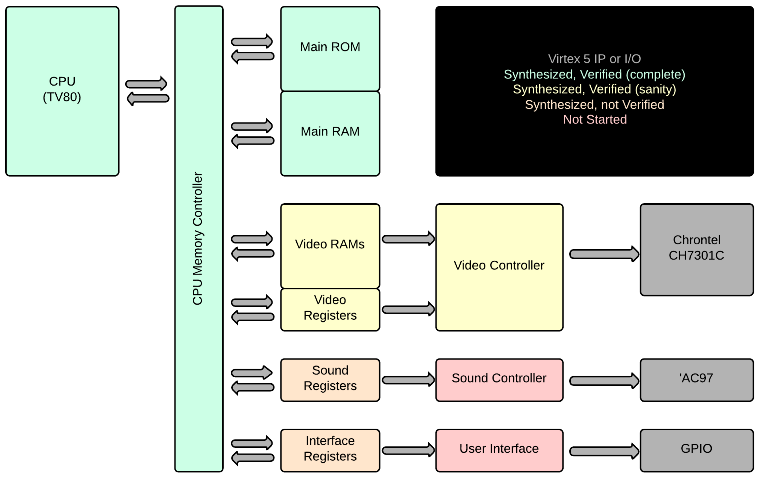


Figure: Memory Interface Diagram

The memory is mapped at the addresses listed in the tables below. These are still untested locations, however the emulator we are used listed them as follows.

Name	**Start Address**	**End Address**
Main ROM	0x0000	0x3FFF
Main RAM	0x6000	0x6BFF
Sprite Ram	0x7000	0x73FF
Tile Ram	0x7400	0x77FF
8257 DMA Registers	0x7800	0x780F
Interface 0	0x7C00	0x7C00
Interface 1	0x7C80	0x7C80
DSW 2/Audio IRQ	0x7D80	0x7D80
Flip Screen	0x7D82	0x7D82
Sprite Bank	0x7D83	0x7D83
NMI Mask	x07D84	0x7D84
8257 DRQ	0x7D85	0x7D85
Palette Bank	0x7D86	0x7D87

We see that some of the values in ram might be better suited to being registers, such as the sound select and palette bank, however to simplify our early tests of the game, we are leaving them as rams and then once we have a better idea of what needs to be changed to what we will add complexity to the memory map.

Video Interface

The video interface is comprised of three main parts discussed below, they are the I2C connection to the Chrontel DVI chip, the tile mapping unit, and the sprite mapping unit.

This unit runs at a 50MHz positively edged clock and a 25MHz differential clock running at both edges. The interface is shown below

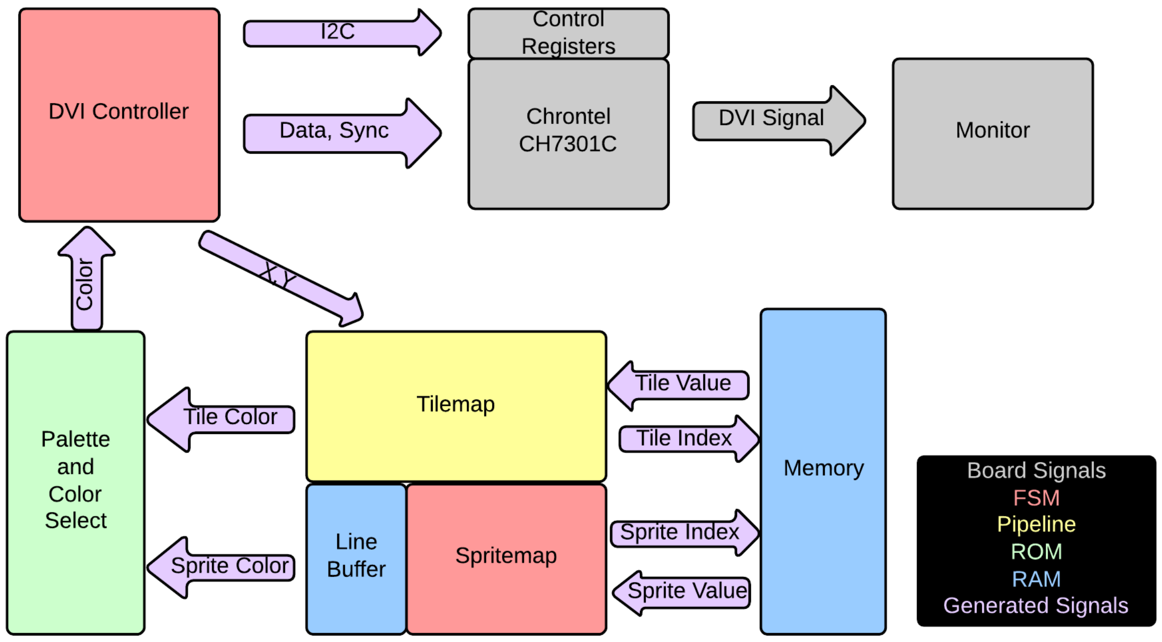


Figure: Video Interface

Chrontel DVI Chip

In order to produce any video output the Chrontel DVI output chip has to be initialized and set to display the output we give it. In order to do this we have to set its registers using I2C through the video I2C bus. We had considered using an already written I2C master controller, however much of the functionality such as reading from the bus was not necessary, so we wrote our own controller that transfers bytes when a button is pressed.

The chrontel interface also controlled the production of sync signals, and tells the video controller what X and Y coordinates are currently being transferred and displayed to the monitor. This interface is extremely simple and consists of two counters and several comparators. For the horizontal sync it counts the number of positive clock edges and then produces the hsync signal once it reaches the ~700 pulses it needs. It also observes the front porch and back porch areas, disabling output during these regions. The vsync signal is exactly the same as the hsync signal however it counts the hsync pulses and only requires around ~500 pulses before it hits vsync. This operates at 60 Hz and overall produces the smallest output resolution of 640x480.

Some issues we ran into with the Chrontel chip were mostly due to the fact that we really didn't understand it when we started working with it. The primary issue was the exact timing constraints that this chip and the DVI signal requires. If we were off by even a clock cycle, the display would not show anything, even when the chip was in test mode. Working out then synthesizing the timings first in VCS, then trying to set up the the black box Chrontel chip would save a lot of time. We tried integrating both the IIC module and the timing controller at the same time and it made debug significantly more painful than it needed to be.

Tilemap Pipeline

The tilemap contains the current background image of the frame. Each tile is 8x8 pixels made up of four colors from the currently selected 64 color palette. The tilemap is a simple pipeline that performs several convoluted operations to retrieve many pixel colors from a small amount of memory, which I have broken down below as a list of steps.

1. Convert X and Y to a tile index by removing the lower three bits of each, inverting X, and adding a base address of 0x40 to the value. This produces a 10 bit ram address. Take this index and also produce a color index by removing bits 5 and 6 to produce an 8 bit rom address

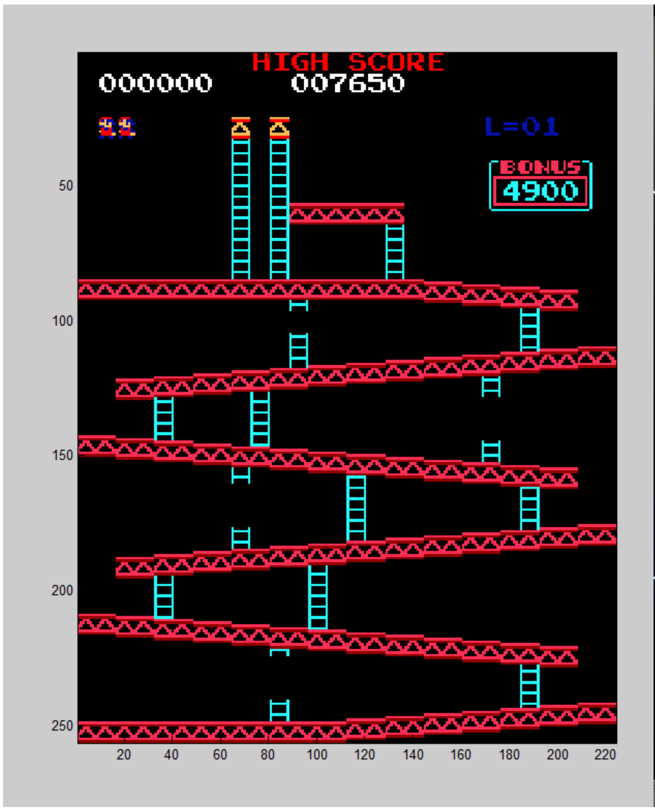
2. Using the ram and rom addresses read from video ram using the ram address and the from the color prom using the rom address. The 8 bits from vram are the "code" and the lower 4 bits from the color prom are bits [5:2] of the color.

3. Augment the 8 bit code retrieved from vram with inverted X coordinate's 3 lowest bits to produce the "tile code." Using this tile code retrieve a byte each from the two tile roms.

4. Only using the Y coordinate’s lowest three bits index into those two bytes grabbing one bit from each. These two bits form the bottom two bits of the color.

5. The final color is produced by adding the current palette, a two bit signal, to the top of the color. This is then used to index into the palette and produce the 16 bit color that is required for the chrontel chip, however it is stored as an 8 bit palette index to be compared with the sprite’s current value.

To validate this I used three stages, the emulated golden model, a matlab functional model, and finally the fpga hardware model. Images of each are shown below. You’ll notice that there are slight coloration differences due to the palette conversion from 8 to 16 bits being a bit off, however it’s a good proof that its working as intended and is minor enough that it does not affect playability or functionality. We ended up deciding not to “fix” these coloration issues through the palette since we thought the color difference made the game look better. Even it wasn’t exactly matching the colors, we felt that the deeper reds and oranges made things easier to see and overall improved the look and feel of the game. This is how we ended up with the final background tilemap module.



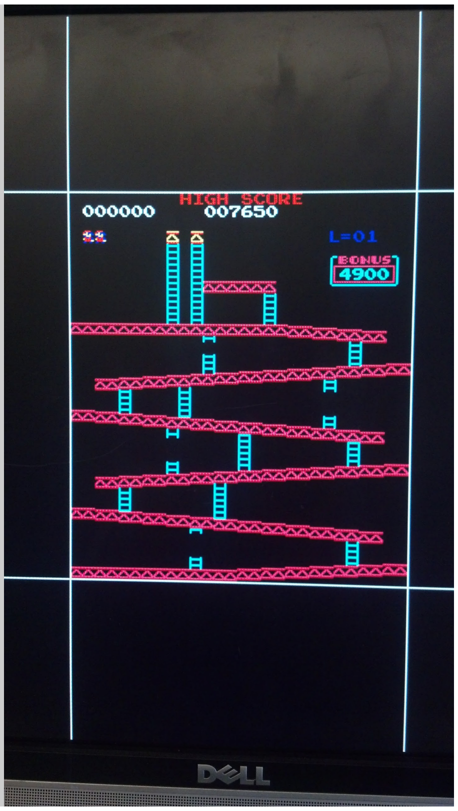


Figure: Left - Functional Model (MATLAB), Right - Hardware Model

Sprite Buffer and FSM

The sprite mapping is very similar to the tilemapping, however because each sprite takes 40 cycles to render, a line buffer and FSM was much simpler to produce than a pipeline. Each sprite is a 16x16 pixel object that can be placed anywhere on or off screen. It is made up of 3 colors and a transparent color. Since the sprite mapping mostly involves memory accesses as part of its process I'll just describe the memory makeup of each sprite and how its used to generate the final image.

Each sprite to be displayed uses 4 consecutive bytes in memory, the X-coordinate, Y-coordinate, code, color, xflip, and y-flip. From this starting point the steps taken to reach the final display are as follows.

1. Access the 2nd byte in memory (the Y-coordinate) and latch it.
2. Compare the Y-coordinate + 8 and Y-coordinate - 7 to the value of the next scanline being displayed.
    a. If it's not within those two values increment the memory pointer to the next sprite and go back to 1.
    b. If it's within those two values proceed to 3.
3. Over the next six clocks access the remaining 3 memory values and latch them
4. Then using the code and the current memory index, access the sprite rom and pull 8 bytes.
5. From each of these bytes, index with the scanline Y-coordinate, take two of the bits and prepend them to the sprite's "color" value.
6. Insert these created values into the scanline buffer, overwriting what is currently in the location.
7. Increment sprite memory index (unless its at the end of sprite ram, then go to 8) and repeat from 1
8. At vblank, reset sprite memory index to 0, restart from 1. Shift line buffer to the display buffer, the video controller can only access the display buffer.

Again to validate our results we used the three step process of first constructing a matlab functional model, then a verilog implementation of it. You will notice there is an error in the sprite production in that it doesn't produce the sprite that covers the blocks at the top of the screen, we are in progress of fixing this as it may mean there is an underlying problem with transparency. The blue background is to allow us to see where the screen is supposed to end, and if the pipeline inputs are incremented correctly.

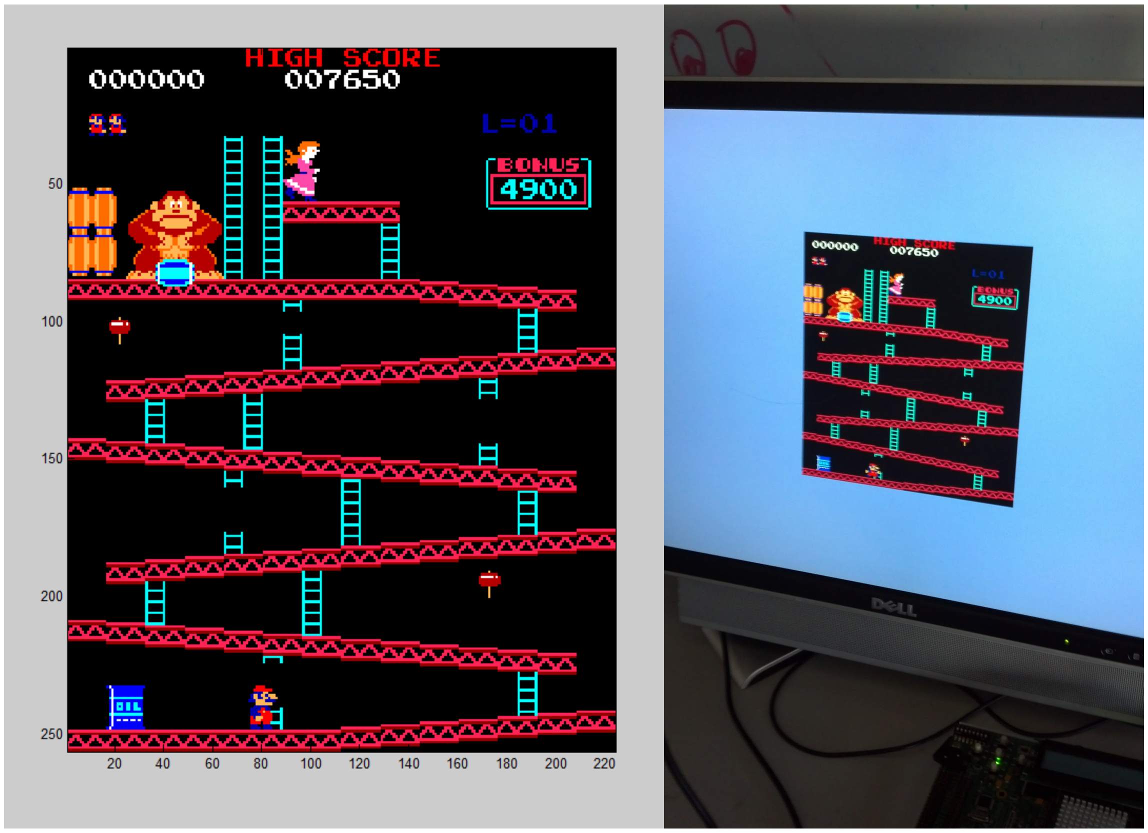


Figure: Left - Functional Model (MATLAB), Right - Hardware Model

Video Controller

The video controller was basically just a small interface between the Chrontel chip, the sprite map, the tilemap, and the palette bank. It sent the vblank signals to the sprite map as well as the X and Y coordinates of the current pixel to both (adjusting for the pipeline timings of the tile rendering pipe). It would also handle transparency using a simple mux on the tilemap value, and send out the muxed colors to the Chrontel chip. Finally it handled the differential clock and reset signals for the DVI chip and I2C controller

User Interface

Our user interface uses the buttons and joystick mounted on the MultiWilliams arcade board, as well as a few internal modules to process this input.

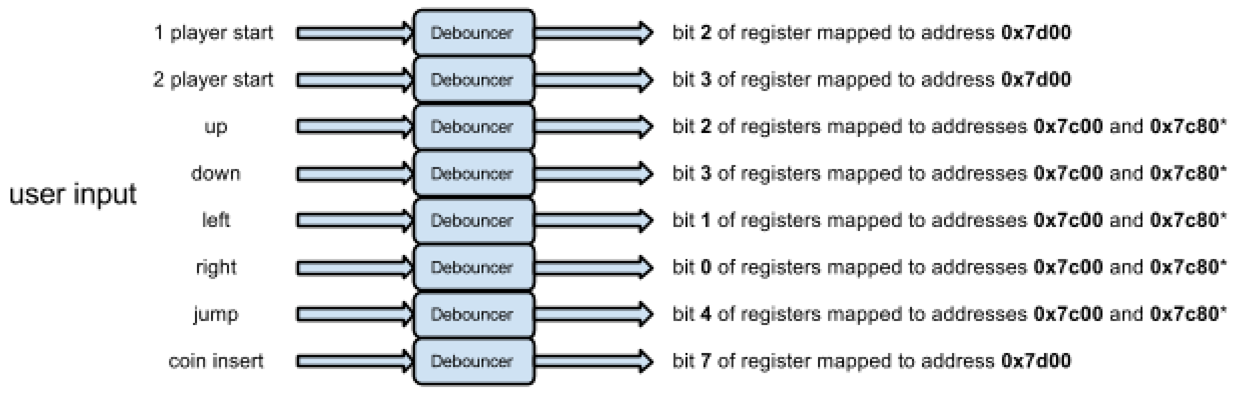

user input
1 player start
2 player start
up
down
left
right
jump
coin insert
Debouncer
bit 2 of register mapped to address 0x7d00
bit 3 of register mapped to address 0x7d00
bit 2 of registers mapped to addresses 0x7c00 and 0x7c80*
bit 3 of registers mapped to addresses 0x7c00 and 0x7c80*
bit 1 of registers mapped to addresses 0x7c00 and 0x7c80*
bit 0 of registers mapped to addresses 0x7c00 and 0x7c80*
bit 4 of registers mapped to addresses 0x7c00 and 0x7c80*
bit 7 of register mapped to address 0x7d00

Figure: Input System and Memory-Mapping

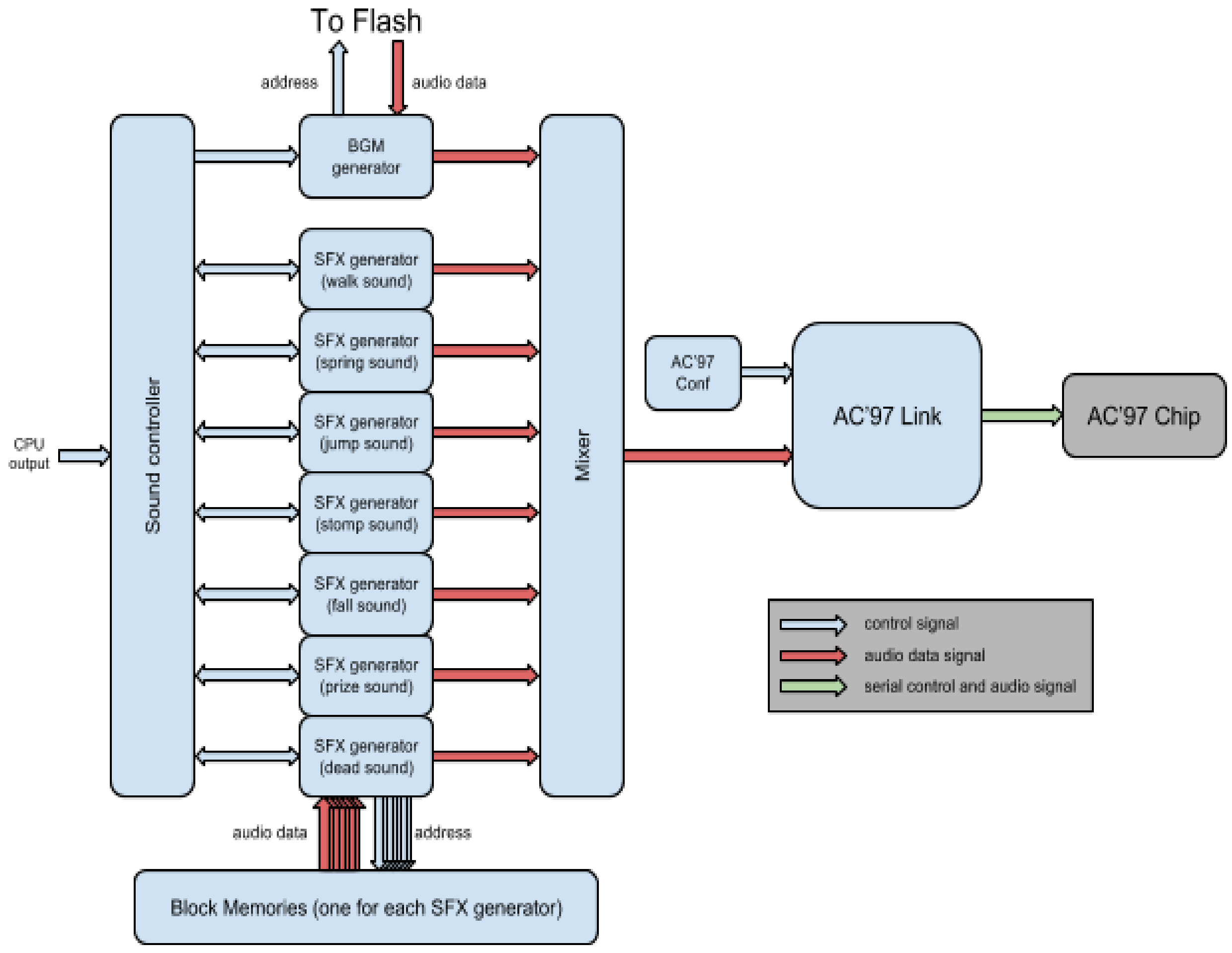
The system consists of eight user inputs: 1- and 2-player starts, four directional inputs, jump, and a coin insert input. Each of these inputs is run through a separate debouncing module, then fed into a specific bit in a memory-mapped register. It implements the following memory mapping:

IN0 (0x7c00)		**IN1 (0x7c80)**		**IN2 (0x7d00)**	
Bit	**Contents**	**Bit**	**Contents**	**Bit**	**Contents**
7	*unused*	**7**	*unused*	**7**	Coin
6	*unused*	**6**	*unused*	**6**	*unused*
5	*unused*	**5**	*unused*	**5**	*unused*
4	P1 Jump	**4**	P2 Jump	**4**	*unused*
3	P1 Down	**3**	P2 Down	**3**	2-player start
2	P1 Up	**2**	P2 Up	**2**	1-player start
1	P1 Left	**1**	P2 Left	**1**	*unused*
0	P1 Right	**0**	P2 Right	**0**	*unused*

It is worth noting that, since we had only one joystick and jump button, we opted to hook those up to both the player one and player two inputs. Since two-player mode involves players alternating playing one life at a time, with no simultaneous play, this does not prevent two-player mode on our system.

Sound Controller

The original Donkey Kong arcade game used a processor to handle sound. Rather than attempt to get this to work, we opted to design our own sound controller starting from Team Dragonforce's Audio Codec '97 interface code. The system is structured so as to have a sound controller snoop in on CPU outputs, and from that control a number of sound generators, each with their own purpose. The MAME source helped a great deal in this, in that it specified where the CPU writes to when it wants to do each sound.



To Flash
address
audio data
BGM generator
SFX generator (walk sound)
SFX generator (spring sound)
SFX generator (jump sound)
SFX generator (stomp sound)
SFX generator (fall sound)
SFX generator (prize sound)
SFX generator (dead sound)
Sound controller
CPU output
Mixer
AC'97 Conf
AC'97 Link
AC'97 Chip
control signal
audio data signal
serial control and audio signal
audio data
address
Block Memories (one for each SFX generator)

Figure: Sound system overview

Background Music Generator

The MAME source specified two types of sound: foreground and background. Background sounds are triggered by writes to address **0x7c00**. The low 8 bits of the write contents determine which background sound to start playing. Background sounds have the distinction that only one of them is ever playing at a time. Not all of them are, in fact, music, but we still use the term BGM (background music) as an abbreviation for them.

The BGM generator is essentially a glorified counter whose output is used as an index into flash memory. When it receives a signal to start a sound from the sound controller, it changes its output to a particular address which is the start of the appropriate background sound chip in flash memory. It then continues to increment that address each strobe, until it reaches an address designated as the end of a clip. Then, based on how the generator is configured (this is done through Verilog macros), it will either loop the clip or stop playing anything until it gets a new trigger. This functionality is implemented because the CPU and ROM expect some clips to loop and not others.

Notably, the MAME source seemed to have a couple of errors when it comes to specifying background sounds. Strangely, it seemed to designate the “hammer hit” sound as a background sound, when it behaves like a foreground sound in many ways. To remedy this, we moved the hammer hit sound to be a foreground sound.

SFX generators

Foreground sounds are like “sound effects,” in that they play on top of background sounds and multiple of them can be playing at a time. The sound generators for foreground sounds are much simpler than the BGM generator. They each provide addresses into block RAMs that contain each individual sound effect. As the MAME source specifies, the sound controller triggers each one based on writes to the addresses **0x7d00-0x7d05** and **0x7d80**. These SFX generators, as we call them, are essentially counters with defined endpoints that start counting up whenever the sound controller tells them to. Each generator’s output is used as an index into a different block RAM which contains a different sound.

Notably, the walk sound effect has a slightly different generator because the walking sound effect follows different rules. Due to the possibility that the player starts and stops walking multiple times very quickly, the walk sound effect has to be ready to start and stop whenever the CPU writes to it. In addition, the original game doesn’t always start at the beginning of the sample when a player starts walking - there are multiple different step sounds, and the game seems to rotate through them. We implement logic in the walk SFX generator that we believe implements this functionality fairly accurately.

Sound Controller

The sound controller is the module that listens to CPU output and controls the aforementioned modules based on what it sees. In the case of the BGM generator, it only essentially passes triggers forward - the BGM generator then does its own thing, and doesn't communicate anything back to the controller. The SFX generators are different, however: we designed the sound controller so that (a) it can stop sounds whenever it wants to and (b) it will ensure that no more than 3 SFX are playing at once. The reason for this limit is so that mixing the outputs of the modules while guaranteeing no clipping is an easy task. To do this, we implemented the SFX generators with a done signal, which the sound controller uses to determine when a sound as finished playing and thus maintain an accurate idea of how many sound effects are actually playing at any given time.

If a sound effect starts to play when three sound effects are already playing, the oldest sound effect will be stopped. The controller does this by de-asserting a 'go' output to the appropriate SFX generator. Doing this silences the generator's output and resets its address to zero (except for walk; see above).

Mixer

As mentioned above, we only allow 3 sound effects to play at once on top of one background. This makes our mixer extremely simple: take all the outputs of the generators, arithmetic shift the outputs of them by 2 bits, then add them together. Since no more than four generators produce sound at a time (one BGM, three SFX), this ensures that clipping will not occur. The output of this mixer is passed on to the Team Dragonforce AC'97Link module.

System Reset

One small thing that was fairly integral to the system working was the automated reset we included in the system. On startup it would wait one second and then start a full system reset. It would reset the CPU and video interfaces. This was extremely important because if the IIC controller started before the Chrontel chip was ready, no video would appear on screen. Similarly if the cpu happened to start in a strange state or junk was in its registers, the game would not start correctly. So we needed to send a global reset slightly after the FPGA started. We initially had this tied to a button on the FPGA, but it was much simpler to just have it reset automatically after start.

System Planning and Tools

Plan of Attack / Project Goals

As mentioned earlier we wanted to produce a functional emulator of the original Donkey Kong system using the MultiWilliams gamepad. We would achieve this goal though a fairly front loaded schedule and research from the 1942 team. The schedule below is what we ended with, and it was almost exactly what we had planned with the exception that the sound was started a week earlier and ended a week later than planned.

Schedule

The schedule for our project using a Gantt chart because we agreed that there was no good way to add a percentage finished to the unit. It was one of four things, not started, not working, passing sanity tests, and competed, so we opted for a task list. The schedule for our project in weeks is listed below.

- (weeks 1-2) acquaint ourselves with the CPU
- (week 1) Implement DVI controller
- (week 2) Tile Map Functional Model (MATLAB)
- (week 3) Tile Map Verilog implementation, sprite map funct. model (MATLAB)
- (week 3) Memory Mapping
- (week 4) Sprite Map Implementation
- (week 4) Memory Implementation
- (week 5) Game ROMs and RAMs
- (week 5) Video + CPU + Memory Integration
- (week 6) Verilog for Controls
- (week 7) Plan out hardware for Multi-Williams board
- (week 8) Interface Integration
- (week 9) Integration Full Test
- (week 9) Start Implementing Sound
- (weeks 10-11) Background Sound (Enable Flash)
- (weeks 11-12) Mixer/Sound Effects Sound
- (week 12) Testing and Debugging

Figure: Schedule

We followed the schedule above to complete our goal. Week 12 was the week of the demo, so we had finished all the units just in time for the demo. The workload was distributed such that in the first 6-7 weeks we put in about 20 hours/week/person on average. Then once video and CPU were complete and tested, the workload shifted down to around 12 hours/week/person finally down to 8 hours/week/person. Had we had the full three people in our team we would have included another set of milestones for the AY-8035 sound CPU to produce our sounds.

We chose to schedule in this manner for two reasons. Primarily we chose the initial schedule because we wanted a viewable product early on in the process. Even if we couldn’t finish the other pieces of the project, we would have a demo to show and also have a way to blackbox test the other game play pieces. This may not have been the best way since we it’s very hard to test a CPU system without the surrounding video or audio systems, however getting this milestone completed really raised our moral and kept project momentum going. It also produced a tangible goal to be able to see our results and to get the game up and running (in any form) as soon as possible.

System Tools

We used several tools to accomplish our goals. The first and primary tools we used were XST, ISE, CoreGen, and iMPACT. These were the primary tools used to generate verilog blocks, compile, and synthesize our implementations. We also used VCS to run simulations and test most of our implementation. Finally we had a set of Makefiles that were used to automate the compilation, simulation, and synthesis of our code. The Makefiles were fairly integral to our productivity because it allowed us to compile all our code over AFS and let us do work on our project while we were not in lab. We also did not require that ISE be open, because that really was not ideal as we did not need to have a bunch of other project files included in our repository. It also helped us produce the required

Something of note is that a lot of time was spent trying to avoid using the Xilinx tools because they take a very long time to use. If you want something resynthesized it could take up to 15 minutes for the cycle to process to complete. This would severely limit productivity as it generally became a block until the process was complete. Adding intermediate testing points through VCS simulation or MATLAB functional models allowed us to get much of the verification done before having to subject ourselves to ISE or XST.

Challenges Faced

This section represents a conglomeration of the challenges we faced either with tools or project complexity. We decided to include this because future teams that happen to have the same problems as we did might find some solutions from our many struggles.

iMPACT USB Drivers

These things were not reliable. You could have been using iMPACT one minute fine, then after loading or during the loading of drivers, they would crash and never recover. We tried to solve this problem by reinstalling drivers, but they would just die again in the worst times. The one solution we found, that worked well enough, was to just download a local copy of iMPACT on to our laptops. It worked much better on Windows than RedHat and the drivers were very stable. It added a step of sending bit and MCS files to dropbox, but it saved a lot of frustration and hassle.

Color Shift (Video Output)

One of the banes of the video system we had was that sometimes when we made small changes to unrelated systems (like the sound system) a small change in timing would occur and the video would then flip the red and blue with the really strange effect that the image would be blue instead of red. We attributed this to a fault in XST optimization where it would change the placement of some of the video systems to lower area or timing overall, while slightly disturbing the output timing to the Chrontel DVI chip. Two things helped us, adding another pipeline stage to the muxed color output (since the colors need to be sent in halves) solved the problems initially and it was sort of a stopgap to our problems. The second part, which probably would have solved the problem from the beginning, is that the Chrontel chip has a clock skew register. You tell the Chrontel chip that it has to wait X amount of delay after the clock edge before it should latch the data. We didn’t measure this timing X, but when we added this our problems were solved and never returned.

BRAM and BROM Space

When we were first implementing sound we ran into a problem where we ran out of block RAM and ROM space on our FPGA. The LX110T has ~140 block ram blocks that can be used, however they run out very quickly when you put 48kHz sounds on them. Unfortunately there really isn't a good solution to this, but we tried several work around for this. The big thing that we did to get more space was to move half the sounds to flash (the ones that don't have to play at the same time) in order to alleviate utilization issues. Then we lowered the sample rate of the sounds and that helped a enough to solve the problem. One solution that really did not work well was to produce distributed ROMs. We tried that and it ultimately failed to produce any results, so much so that it actually affected circuit timings so much that it broke the already working systems. It's not a really good solution to generate distributed ROMs. We also did change compilation flags to use distributed rams instead of block rams for verilog constructs, but it didn't do anything but increase the compilation time.

Acknowledgements

MAME

This was the emulator we based our system off of, it really was the basis for our project.

Team Dragonforce

We used their code as a base to our audio systems and Makeflow. We did not use their video system, and everything that we attached was heavily modified, but it was very helpful in our understanding of AC97 and the XST Makeflow.

Team MultiWilliams

Your board was awesome. We put a cool poster on it.

TV80 Core

Rather than building a whole processor on our own, we used this one from OpenCores.org. Donkey Kong probably doesn’t use all its functionality, but it worked well for our needs.

Personal Statements

Mark

It was unfortunate to lose a team member early in the year, but it did sort of motivate me to complete a project, comparable to a previous year's project, but with one fewer people. This was mainly the reason I did not want to give up this project, and it made solving the challenges of this project all the more fun to work on. It was a great experience to work and try and manage this project, and I was extremely happy to even have created a complete product at the end of the semester.

I got to completely build and debug the video system while Andy did the CPU and memory, which I had asked for mostly to learn more about how graphical systems translate to hardware. While doing this I also got the repository and makeflow up and running so that we didn't have to deal with the ISE systems. After this I created the physical interface for the FPGA board to the MultiWilliams board and wrote some of the debouncing logic. Finally I generated the sound assets for the game and bashed my head against the flash interface that we eventually got working. I also did a lot of verification and debug for the system integration, which wasn't really too much since we had a lot of unit verification that made sure that the major pieces worked at least well enough. We had split the work up fairly evenly in terms of hours spent, and I didn't get the feeling that either of us were working too hard or slacking off at all.

On average I worked from 15-20 hours per week in the first 6 weeks, then it lowered to around 10-15 over the next few, and settle down to around 5-10 in the final weeks. This felt really good in terms of a timeline because we got the project rolling early and had time for other courses (and sleep) when other classes ramped up.

My advice to other teams would be to try to frontload the work. It's hard enough to do one project, let alone many projects at the same time, so plan accordingly. Also take the time to write a Makeflow, ISE and XST suck -- avoid using them in their raw forms. Finally, when picking a project spend a lot more time researching and make sure there are tons of resources out there for you to use, make your life not suck.

Andrew

Honestly, I never thought we'd get this far or do this well. Having never done Build18 or anything similar before, I had never actually been tasked with building a system this big, and I was really nervous about this. After we lost our third member, I was especially worried, though I tried not to show it. Mark and I stuck with it, though, and imagine my surprise when we were done with everything but sound before Thanksgiving! Frontloading work is really a fantastic strategy for this course, and I highly recommend it - especially if you're having morale problems early on, or you know your life is going to be really hard during the last few weeks of classes.

While Mark was given basically complete responsibility over video, I designed the first versions of most of the other systems in the design, though Mark had a hand in debugging and tweaking them. Notably, I designed the system that mapped memory according to the memory layout expected by the ROM. I also designed the system that properly routes user input to the correct bits in memory-mapped registers. Finally, I did most of the sound system and interfaced it with Team Dragonforce's AC'97 controller. I also did most of the early black-box testing of the TV80 Z80 core from OpenCores.org.

Overall, I worked about 10-12 hours a week on this project on average. I didn't end up frontloading my work as much as Mark did, since his task was much more integral to testing and I was still having a few morale problems. It worked out in the end, though, and we drafted a rough schedule that we followed after he got video working, leaving a lot of grace time at the end.

As far as advice, I agree with everything Mark mentioned on the last page. I just wanted to add one thing: be aware of how complex your project is when you're deciding on it. Many of our classmates tried making game consoles this year. If you choose something very complex like a game console, things probably aren't going to go as smoothly as you like, crunch-time is practically inevitable towards the end, and your end product probably isn't going to be the full console unless you do *really* well. If you have any doubts as to what your group can put out, consider making something simpler and/or better documented.