# Duck Hunt

An Implementation of MIPS-based
System-on-Chip on FPGA

Andrew Drake
Lei Fan
Andrew Israel

## Table of Contents

## Introduction

The goal of this project is to implement the classic arcade game, Duck Hunt, on a complete MIPS-based computer system on a Virtex 5 LX FPGA, including the following components:

- MIPS CPU with a five-stage pipeline, with support for stalling/forwarding and exception and interrupt handling
- Memory bus with priority-based arbitration
- DVI/VGA output from Chrontel CH7301C, with video data cached in a double-framebuffer system in main memory
- AC'97 audio output with Analog AD1981B, with audio data cached in a double-audiobuffer system in main memory
- Main memory implemented with 256 MB of DDR2 SODIMM, with custom wrapper written for the auto-generated Memory Interface Generator (MIG) IP provided by Xilinx Coregen
- Custom-written Unix-like kernel, compiled for MIPS ISA

Instead of porting the original Duck Hunt game to run on the custom Unix kernel, a simplified version of the game was written in C and compiled to MIPS assembly to demonstrate the ability of the system to execute MIPS assembly in on a real-time system, including interaction with various memory storage devices and peripherals.

## Hardware Design

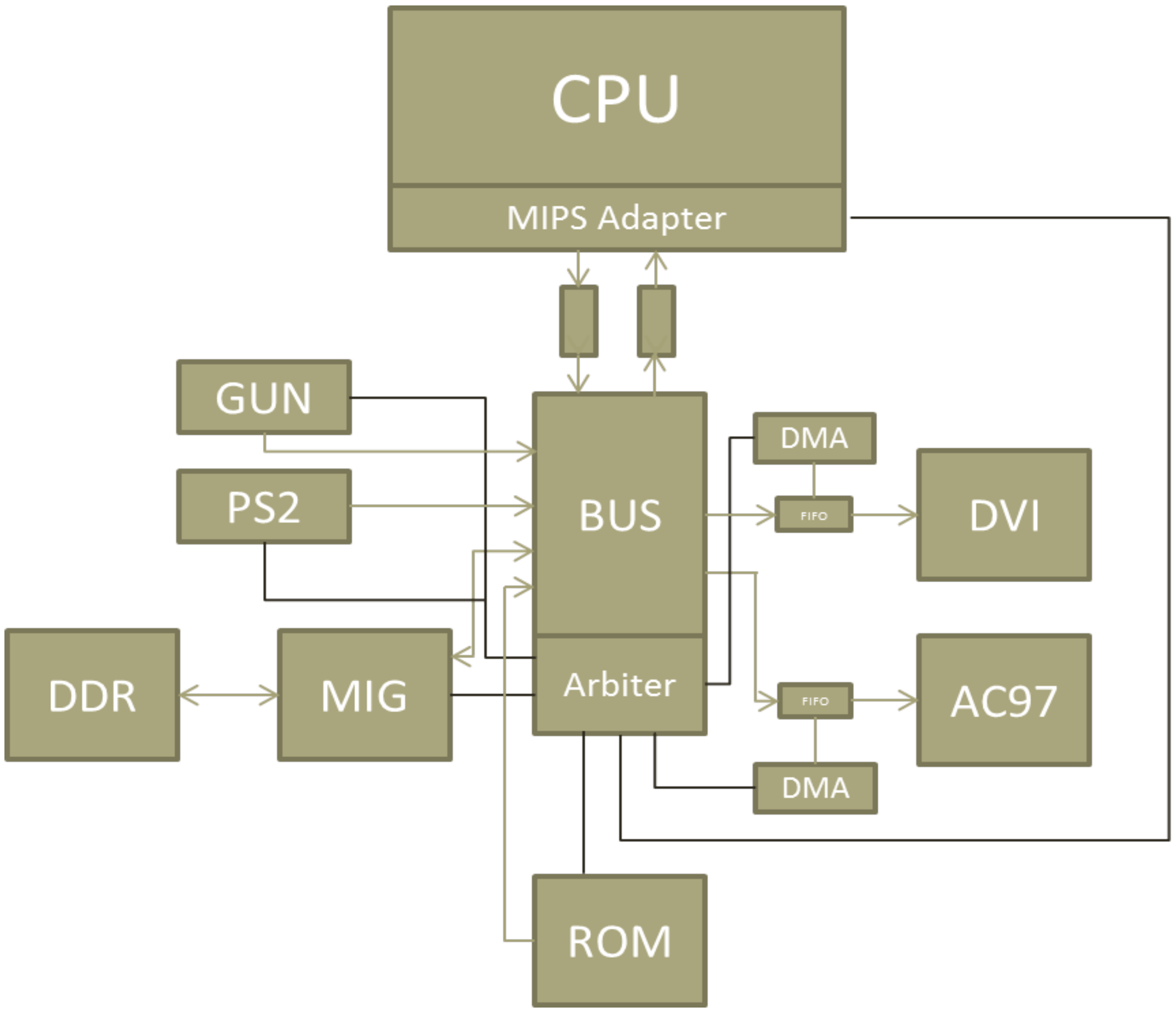


**Fig. 1**: High-level overview of the hardware implementation.

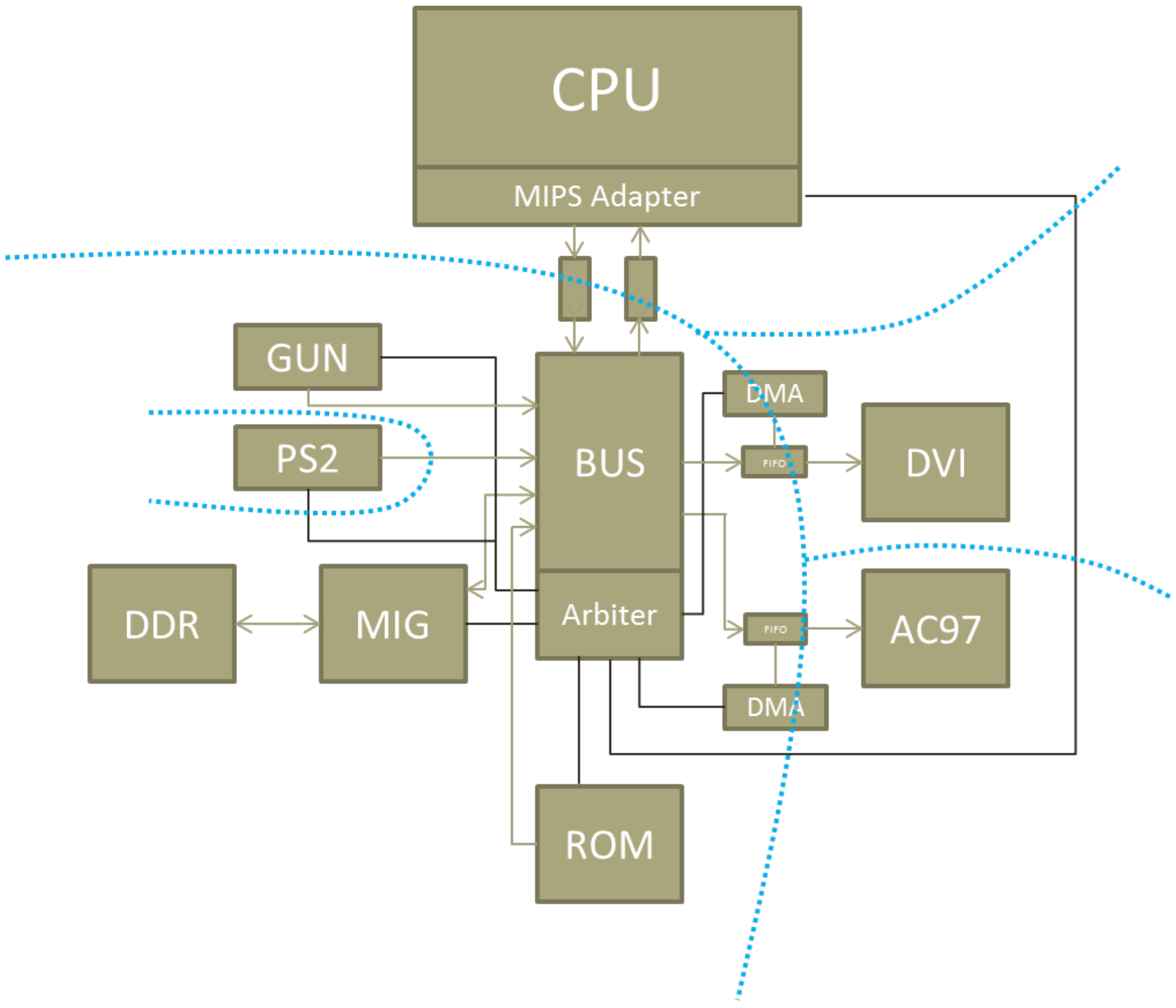


**Fig. 2:** High level overview of the five clock domains in the system.

## MIPS Core

The MIPS core is based on the MIPS CPU written in 18-447, Introduction to Computer Architecture. It has the following five pipeline stages:

- **Instruction fetch** (IF): in which the instruction at the memory location stored in the program counter (PC) is fetched and stored in a pipeline register
- **Instruction decode** (ID): in which the instruction is interpreted and the appropriate operands chosen, either as an immediate, or from the register file
- **Execute** (EX): in which the operands are passed to the arithmetic logic unit (ALU), along with the appropriate opcode, and the result of the calculation is stored in a pipeline register
- **Memory access** (MEM): in which either a value is loaded from or written to a memory address, either provided as an operand or as the output of the ALU
- **Writeback** (WB): in which either the result of the ALU calculation or fetched value from memory is stored back into the register file

Several additional sub-systems and co-processor registers are also implemented:

- ALU Co-processor: in which multiply and divide operations, both signed and unsigned, are carried out
- Status/Cause registers: in which exception codes are stored for the exception handler to determine the appropriate course of action
- Count/Compare registers: in which timing information for OS-level scheduling and context-switching are stored

Data hazards such as Read-After-Write (RAW) are properly handled by either forwarding the values from the datapath as they become available, or, if appropriate, stalling the pipeline until they become available.

Software exceptions and hardware interrupts are treated equally by the MIPS core. A hardware interrupt is latched in the EX stage of the pipeline and passed down until the WB stage, where the entire pipeline is flushed and the kernel directs the core to load in and execute the appropriate exception handler in software based on the cause code.

The core runs on its own clock at 50 MHz.

## Memory Bus and Arbiter

A memory bus has been implemented as the single point of communication for all devices, aside from a few point-to-point connections for interrupt signals between various devices and the core (more details to follow). An arbiter is built to coordinate all communication on the bus by receiving requests and either accepting or rejecting them, based on a pre-established priority protocol.

The bus itself stores the following three pieces of information:

- Bus data: 64 bits
- Bus address: 29 bits
- Bus data valid: 1 bit

The bus itself has a latency of one cycle.

The arbiter is a state machine that performs the following tasks:

- Wait to receive requests from all devices
- If a request is received, the address and read/write length associated with the request is examined; attempts to read/write to non-existent addresses or of the inappropriate width will be rejected by a bus error
- If multiple requests are received in the same cycle, the following priority for grants is used, with the highest priority placed on top:
    - DVI DMA read request
    - AC'97 DMA read request
    - Core data read request
    - Core instruction read request
    - Core data write request
- The source, destination, and address of the granted request is stored, and a read or write request is sent from the arbiter to the device corresponding to the address
- Once data valid is received and the transaction is complete, the arbiter goes back to wait for new requests

Due to aggressive pipelining, the arbiter has varying latency response for different types of requests, ranging from one cycle to three cycles.

The bus and arbiter both run on the clock generated by the MIG at 125 MHz.

## Communication Protocol

The communication protocol is thus:

- A requestor device issues a read or write request to the arbiter and waits for the corresponding “busy” signal to go low
- Once the busy signal goes low:
    - If the device is reading, it waits for data valid on the bus
    - If the device is writing, it will write the appropriate number of cycles of data onto the bus with the address that corresponds to the first cycle; a write is complete
- The arbiter informs the device corresponding to the requested address that there is a read or write request and it should latch the address, and data for the case of a write request, on the bus
- For a read, data valid from the producer of data is latched on the bus and broadcast to all devices; only the originator of the request is expecting the data valid line and is therefore the only device that will respond

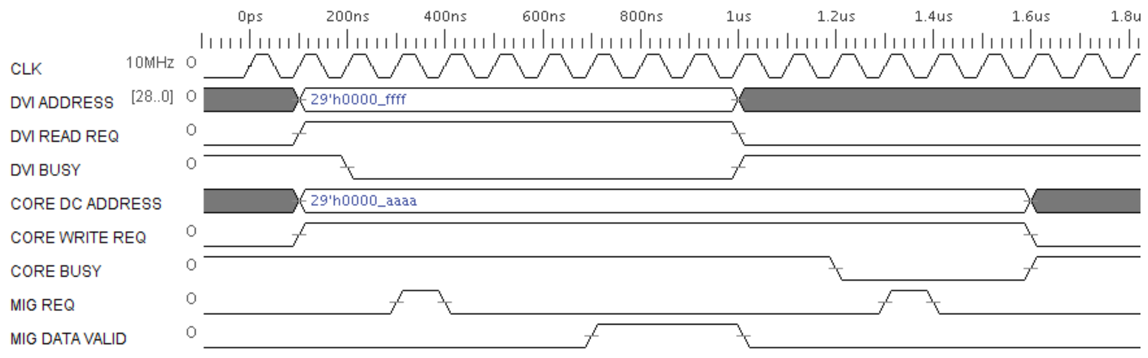


**Fig. 3:** Demonstration of DVI and MIPS Core requests to MIG.

## Cross-Clock Domain Interface Protocol

Most of the cross-clock domain interface protocol is handled by the First-In-First-Out (FIFO) queues that are automatically generated by Xilinx Coregen. These FIFO queues allow for arbitrary clock rates at the read and write ends, arbitrary width and depth for storage, and simultaneous reads and writes.

The only cross-clock domain interface that does not implement a FIFO queue is that between the PS2 keyboard and the bus. Instead, a state machine is used to keep track of whether there has been a clock edge on the keyboard clock. Given that the bus clock is around four magnitudes[1] faster, this was not considered a problem. Indeed, during actual implementation there were no problems detected reading from the keyboard.

## MIPS Adapter (Cache)

The MIPS adapter, built between the MIPS core and the rest of the system, serves as a cache to hold the data and as a converter between the core's 32-bit protocol and the bus's 256-bit-bursts protocol.

Every time the MIPS core requires access to the rest of the system (via a load/store instruction), it sends the request and the virtual address to the adapter, which converts it to a physical address and forms a request packet, which includes information about the nature of the request (read/write), destination address, write data (if needed), and so on.

The packet is then sent through a FIFO queue, crossing the clock domain between the core and the bus, and the packet is decoded on the other side of the FIFO queue and a request is sent to the arbiter for acknowledgement.

When the requested data comes back in bursts of four cycles, a shift register is used to store each cycle and the completed 256 bits are sent through a FIFO queue back to the core side of the adapter in a packet along with information about the nature of the request and the address requested. The packet is then decoded and the appropriate 32 bits chosen (using the stored virtual address) and sent back to the core.

Due to the lack of an actual cache, a write request is handled via a mechanism akin to a DRAM refresh cycle. The virtual address from the core is examined and a read request packet is constructed and sent to the destination inferred from the address. Once the 256 bits of read data come back, the write data is written into the buffer, using an appropriate mask, and the modified 256 bits of data are used to construct a write request packet and sent through the FIFO queue again.

For a non-burst read/write, appropriate masks are used to only send 64 bits of data, or only 64 bits of data will be expected to come back into the shift register.

[1] Bus clock = 200 MHz, keyboard clock ≈ 15 KHz; 200 MHz / 15 KHz ≈ 13000

## DMA Devices

Two DMA devices implemented in the system: DVI and AC'97. Each is composed of a DMA controller and the device itself. The purpose of the DMA controllers for their respective devices is two-fold:

- To cross two clock domains
- To serve as a buffer for data to be supplied to their respective devices

Each DMA controller contains a FIFO queue of width 64 and depth 1025. Data is written in from the bus and read out into the respective devices. Each FIFO queue produces a "prog_full," or "almost full" flag, that has been programmed to be asserted when the FIFO queue contains only 1021 elements; that is, when the fifo queue contains enough room for a full burst read. The prog_full flag serves as the request to the arbiter for read from DRAM.

Each DMA controller also contains six registers: four address registers of 29 bits each, two of which serve as the starting location of the two buffers in memory, the other two serve as the ending locations of the two buffers in memory; one control register of 1 bit, which is asserted when both of the buffers in memory are ready to be read from, and one status register of 1 bit, which is 0 if the first buffer is currently being read from and 1 if the second buffer is currently being read from.  This allows the CPU to write to a buffer that the DMA controller isn't reading which prevents tearing. The address and control registers are both readable and writable by the CPU; the status register is only readable by CPU and is written by the DMA controller itself.

An interrupt mechanism is implemented as a point-to-point connection between the CPU and the DMA controller. Whenever the DMA controller has read entirely from one buffer, the controller sends an interrupt to the core, and the core proceeds to read from the status register to determine which buffer has been exhausted.  This allows the core to immediately know when it is safe to write to each buffer without the need for polling.

### DVI

The board contains a Chrontel CH7301C chip that will output DVI.  The first thing we need to do is set up the device registers on the Chrontel CH7301C.  This requires that we use an IIC protocol at a clock frequency of 100 KHz.  The device was set up to supply 12 bits of a 24 bit color scheme at a double data rate (i.e. 12 bits on the positive edge of the clock, and 12 bits on the negative edge of the clock).  This means that we send 1 pixel in a 24 bit color scheme every cycle.

The core, in order to save space, wrote everything into memory as 8 bit color. This means that every bus read of 64 bits supplied the DVI controller with 8 pixels worth of data. Furthermore, every burst read supplied us with 32 pixels worth of data.

The DVI controller which would supply the data over the Chrontel device was running at 50 MHz which we considered to be slow enough in comparison to the bus such that there should never be any problems where DVI has no data to write, especially considering the fact that each bus read gave 32 pixels worth of data.

The DVI controller assumed that the data that the core was writing was in little endian format in order to reduce the amount of work that the core needed to do.

### AC97

The board contains an Analog AD1981B chip for outputting audio. It runs on a 12 MHz clock and outputs data in 16 bit samples. This means that every bus read supplies 4 samples, and every burst read supplies 16 samples.

There are 12 different output slots of 20 bits each and a corresponding valid bit. The first two slots are used to set up the device registers of the AD1981B chip. Slot 1 will specify if we are reading or writing and to which register we wish to read/write to. Slot 2 will specify the data if we are writing. We set up the registers to turn the volume up to full power as the default is muted.

Slots 3 and 4 are audio left and right. Those were supplied by the DMA controller. The audio samples that we used were regular audio files that we converted to a raw format with a sampling rate of 48000 samples per second using Audacity.

The rest of the slots were never used.

## DDR2 Memory Interface

Xilinx Coregen provides the Memory Interface Generator (MIG), which serves as the actual hardware for the SODIMM. The MIG is configured to run at 125 MHz on a single-ended clock.

A custom wrapper is provided to perform burst read/write mode and handle the transition between the bus protocol, which sends 64 bits per cycle, and the MIG protocol, which sends 64 bits per cycle. The bus provides the address and the write data, and the arbiter provides the type of request. Upon a read request, the address is read from the bus and provided to the MIG, along with the appropriate commands. The returned data is stored in two buffers of 128 bits each and passed onto the bus in four cycles of 64 bits each. Upon a write request, the address and four cycles of 64-bit data are read from the bus and stored in a buffer until all ready to be sent to the MIG. As there are no flags coming back from the MIG in the case that a write to memory fails, it is assumed that all writes to DDR2 succeeds.

The MIG controller, as well as the MIG, run at a clock rate of 125 MHz. The MIG generates the reset signal for the entire system as well.

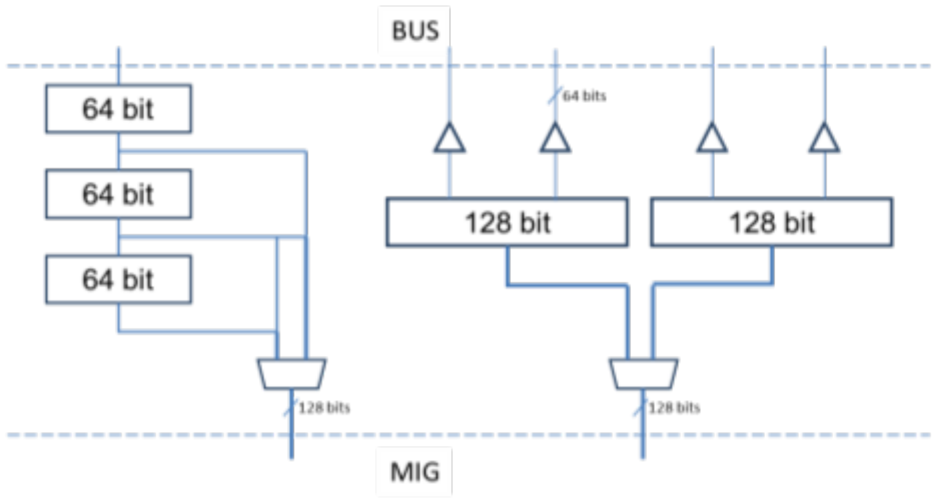


**Fig. 4**: Datapath of the MIG controller

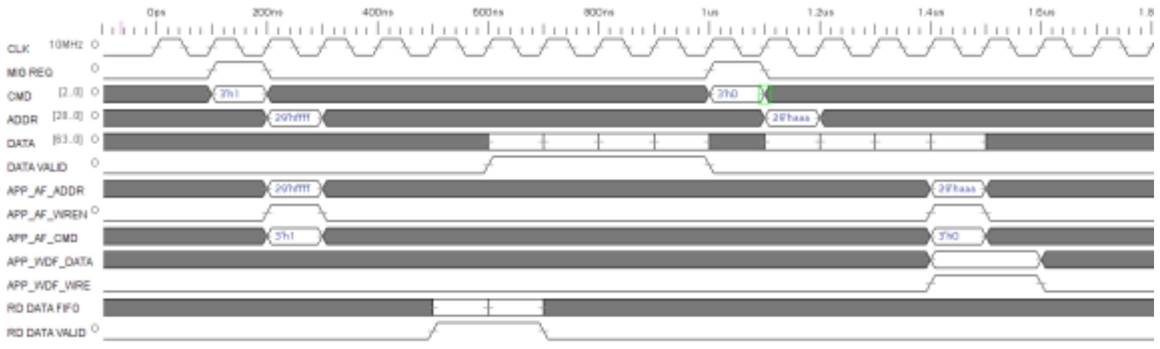


**Fig. 5**: Demonstration of a memory read followed by a memory write.

### ROM

The ROM is meant to store the kernel and all relevant files and is programmed via a hex file at synthesis time. The ROM allows for burst read of length 4.

The ROM runs on the MIG clock of 125 MHz.

## Peripheral Devices

Three peripheral devices (two implemented) are built as part of this project. All three peripheral devices cross clock domains.

### PS2 Keyboard

The keyboard follows the simple PS2 protocol: 8 bits are sent in serially on the negative edge of the keyboard clock, stored and parity-checked, and an interrupt signal is sent to the CPU for the key value to be read. The clock domains are crossed by simply checking the keyboard clock at the bus clock speed.

### NES Zapper

The original NES Zapper has a simple implementation: a trigger signal that is low-assert, and a detected signal that is high-assert. A pull-up circuit is built on a breadboard to drive the circuit in the NES Zapper with a 5V input source from the FPGA and stepped down to a 3.3V output to the FPGA (as the FPGA will only take inputs as high as 3.3V).

When the trigger is pulled, an interrupt signal is sent to the core and the trigger signal is latched. If light is detected anytime between the trigger and the core reading from the NES Zapper interface, the "detected" flag is also latched and signals to the core.

### SystemACE Compact Flash

The original plan is to use the Compact Flash to store the kernel and the entire system; however, this plan was later simplified to use the ROM instead. The SystemACE interface uses a FIFO queue to cross the clock domain – SystemACE runs on a 33 MHz clock. Device registers within SystemACE are set and data is read from the card sector by sector.

## Software

### Kernel

The implementation-level details of the hardware are abstracted away from the application sofware by a custom Linux-like operating system kernel. We elected to use a custom solution rather than to port Linux or one of the BSDs as a time-saving measure: none of us knew any common operating systems well enough to be able to make the changes required for a port, let alone how to write drivers for all of the subsystems we needed. The custom kernel, CircOS, was pulled in from a 15-412 (Operating System Practicum) project by Andrew Drake and Eric Faust, and was a fairly small codebase which could be readily understood and extended.

CircOS is a fairly standard-looking Linux-like kernel. It supports preemptive multitasking and multithreading, kernel-assisted userspace synchronization primitives, and a notion of a filesystem, as well as miscellaneous system calls for memory management and debug I/O. The hardware facilities are abstracted behind a common interface to allow for easy ports to new hardware.

Preemption is accomplished via a time infrastructure that allows additional time sources to be registered by the platform. By default the only time source is the MIPS architecture's Count and Compare registers, which is sufficient for receiving a periodic interrupt for context switching. The core was extended to support this feature in a few minutes.

Most of the kernel's functionality is exposed through the filesystem abstraction, which supports both a physical on-disk FAT16 image as well as a virtual "device filesystem". This virtual filesystem supports mapping devices into memory (such as the framebuffer), as well as read/write access for AC97 streaming support or light gun I/O.

Conveniently, CircOS was already targeted at the little-endian MIPS architecture from 15-412, and since the MIPS core stays very close to the MIPS specification, we were able to make it boot on the hardware without too much difficulty. All that we needed to do was add the simple drivers for the memory-mapped I/O devices to the existing kernel infrastructure.

Unfortunately, at the last minute, we encountered some interaction between the kernel and hardware resulting in a kernel panic, likely as a result of the rushed porting work. This issue proved to not be debuggable in time for the demo day for a variety of reasons. The biggest reason was that the simulator was so slow: we had to wait a solid ten minutes of simulation before the bug triggered, and if we realized we needed a probe on another signal we had to resimulate. Secondarily, debugging a large piece software by watching waveforms of the CPU and bus on a laggy Cadence UI is more difficult than it sounds. Together, these meant that we weren't able to build duck hunt on top of the kernel by the demo.

## Booting Process

Booting is accomplished from a block ram-based boot ROM. The programmable-size boot ROM is connected to the bus and supports burst accesses, so the MIPS core begins execution at the architecturally-defined reset vector of 0xBFC00000, where the first byte of ROM is mapped. Located at that address in ROM is the small bootloader we wrote in C, ABoot, which copies the kernel image from ROM into RAM and branches to it. The ROM is filled by the Xilinx Data2Mem tool, so it is possible to switch out the ROM in the final bitfile without having to re-run the long place and route process.

## Game

We started to write duck hunt before we had the kernel done which meant that we had to take special precaution when testing it.  We assumed that there were four buffers in memory (which would be provided by the kernel) for video and audio.  We wrote a simple function for dumping the contents of the video and audio buffers into a file so that we can verify that the file contents were correct.

To test the dump of the video frame buffer, we wrote a simple script that parsed the buffer and displayed it exactly as the video frame buffer does.  We had tested the video frame buffer many times in both synthesis and simulation, so we were confident that our python script parsed it exactly the same way as the DVI controller did.  Later on, we had the core load in test programs that wrote the same patterns to memory and had the DVI controller display them on screen to further verify that our test framework was correct.

To test the dump of the audio frame buffer, we took the raw file and used our lab 2 code to test how it sounded on the board.  Since our current version of audio is the same as our lab 2 code except with a DMA controller around it, this was sufficient for us to test correctness.

Once we had our testing framework set up, we were able to write and test duck hunt without the need for the kernel.

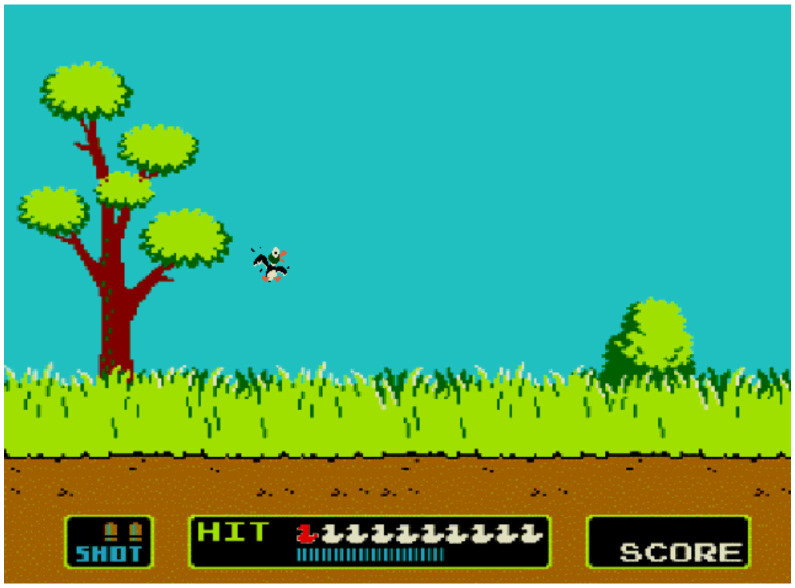
Our version of Duck Hunt was very similar to the original version with a few differences:

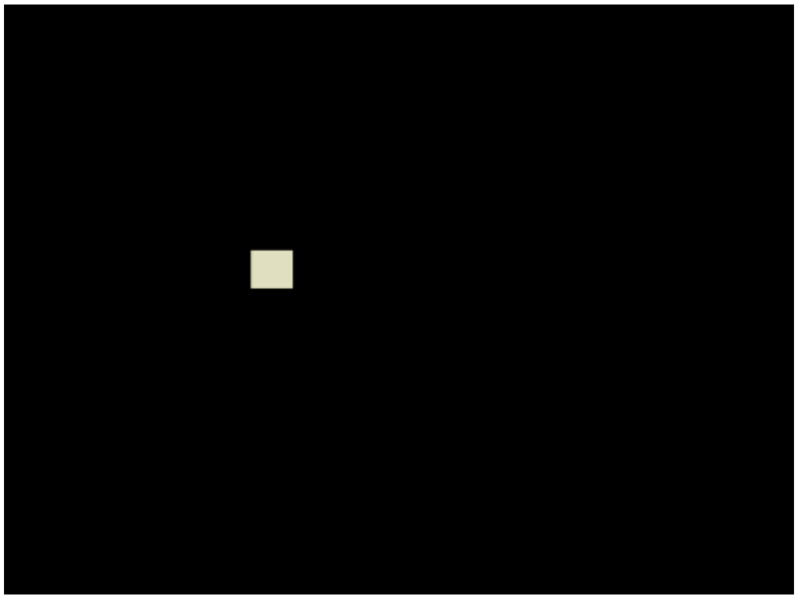
- There was no dog (sorry, we're sad about that too)
- The birds didn't flap their wings as they flew

- The birds didn't get faster as the rounds increased
- The birds did display the "I have been shot" image, but didn't fall to the ground

There were a lot of notable similarities:

- The ducks at the bottom of the screen turned red when you killed them
- The bullets in the bottom left disappeared as fired
- The ducks were animated when they were shot
- The ducks "flew" around the screen
- Your score was displayed in the bottom left





The kernel that we were using did not have a file system; therefore, we had to hardcode the images into the source code of Duck Hunt. These images were very large and we therefore needed to encode them in some way to make them smaller. Since a large portion of the Duck Hunt background image is the same pixel repeated for a large period of time (see the background in the screenshot above), we decided to use a run length encoding (RLE). This is a simple encoding in which instead of writing down each pixel at each location, you instead write down each pixel and the number of times it is repeated.

By using run length encoding on the background image, we were able to reduce its size by roughly one fifth and it would then fit in ROM.

We also tried to encode the whole program and then write a decoder that would decompress the program and then run it, but we found that the images were really the only part of the program benefiting from RLE, so encoding the whole program was not worth it.

## Testing and Verification Methodology

### Simulation

Major component of the hardware were written and tested in simulation before testing on the FPGA. Simulation testbenches, mainly in the form of implicit FSMs, were written for these components and the waveforms are examined in either VCS or NCSim to check for compliance with pre-established protocols.

A simulation script was developed by Andrew Drake that was able to simulate the entire hardware system and generate waveforms viewable in NCSim's Simvision software. This script was heavily used near the end of the project, in conjunction with the pre-loaded ROM to examine the behavior of the kernel in the core and its interaction with the rest of the system.

One problem that we ran into was that simulating the Memory Interface Generator (MIG) was very difficult. Not only did it take a long time to initialize, it also didn't always work. We instead wrote our own simulation only MIG once we knew that the MIG was working in synthesis and had heavily tested it (see the section below on synthesis testing).

## Synthesis

Once the components are known to work in simulation, they were synthesized using Xilinx's xflow tool chain. All warnings and errors generated by xflow were examined in detail and either fixed or deemed safe to ignore (which was often the case, especially dealing with Xilinx auto-generated code).

At first, the major components, such as DVI, AC97, and the MIG, are tested independently on the FPGA. For example, both DVI and AC97 were manually fed data (the inputs were tied to constant values), and the output was examined for correctness; for the MIG, a synthesizeable testbench was written to mimic the MIPS core and issued requests with data and addresses to the MIG. We performed many stress tests of the MIG, such as writing a different value to every single address in memory and reading them all back to be checked against what was supposed to have been written into them.

Once the main hardware components are known to work on the FPGA, the synthesizeable testbench used for the MIG is expanded greatly to take into account all devices on the FPGA. The largest stress test we did was we had both DMA controllers (DVI and AC97) set up to read from large areas of memory constantly. We also had the synthesizeable testbench constantly writing/reading to other areas of memory, simultaneously. Finally, we used the keyboard and gun to control the flow of our synthesizeable testbench. All of these read/writes were verified to be correct and all of them went over the bus. This helped prove to us that our hardware was correct.

Chipscope modules were installed in every major hardware component and enabled, one at a time, to examine the waveforms in the case of unexpected behavior. We have found Chipscope to be immensely helpful in finding bugs, some of which were not obvious in simulation testing, or simply did not show up when simulated (auto-generated hardware components, such as the FIFO queues, are known to behave differently in simulation than in synthesis). However, given the fact that synthesizing only part of our design takes up to 20 minutes, our reliance on Chipscope for synthesis testing proved to be overly time-consuming.

## Tools

We used the Xilinx command line driver tool Xflow for synthesis, place and route, and configuration file generation. Previous group reports as well as group member personal experience indicated that we should steer clear of ISE, which we did. We used the Xilinx graphical application PlanAhead to visualize long routes on the FPGA a couple of times, and iMPACT to load the configuration file onto the board over JTAG whenever we needed to test. We used ChipScope heavily to debug, which was immensely helpful. All of the Xilinx tools had a nasty habit of crashing, internal error-ing, or simply refusing to work at the worst possible times, although we never actually encountered any synthesis bugs in the latest version of the toolchain.

For simulation, we made use of Cadence's NCVerilog with Xilinx's functional simulation libraries for the FPGA primitives we had to instantiate. We had some difficulty debugging the models when they didn't do what we expected, but having the virtually unlimited full-wave visualization of the entire system made up for the couple hours of debugging.

For software, we used a GCC configured to cross compile to MIPS, and encountered no real obstacles.

## Final Thoughts

### Status and Future Work

We were unable to meet our ultimate goal of having Duck Hunt running on our custom hardware and kernel at the end of the semester. However, all the hardware described in this reported have been completed and tested to work on the FPGA. A boot loader was written and works correctly in loading the kernel into the core. However, the kernel was unable to execute correctly due to kernel panics that we did not have adequate time to debug fully. We were able to load the kernel into ROM and have the core begin executing instructions from it correctly. Furthermore, when the kernel panicked, it did actually jump to the right location.

The actual game, Duck Hunt, was written, despite having been stripped down from the original version. Unfortunately we were not able to put Duck Hunt onto the board because the kernel was not fully working. In the end, we had all of the hardware working (except for a few bugs in the core), and Duck Hunt working, but we couldn't connect the two as the full kernel was not yet running.

In the very near future, Andrew Drake is planning on fixing the last couple of bugs in the core and kernel so that the kernel will run and that Duck Hunt will run on top of it.

### Successes

We are proud of the fact that we have successfully built a complete computer system in Verilog that is capable of executing MIPS assembly and interacting with a variety of devices available to us on the FPGA. We are proud of the fact that we have managed to achieve this with almost entirely custom code.

We have learned a lot more about digital logic design during this project:

- Dealing with multiple clock domains and crossing them effectively; the fact that we had five clock domains in our final design (six in the original) was a major complication in the project and many design decisions were made to reduce the complexity of these interfaces.
- Pipelining; we had established very stringent timing requirements on our design that we were able to successfully meet for our final hardware implementation. The static timing analysis reports generated by Xilinx were immensely helpful in helping us determine the critical path and pipelining it to meet our timing constraints.
- Thinking more like the synthesizer: this project really drove home the point that every line of code was translated into actual hardware and therefore had actual fan-outs, parasitic capacitances, drive efforts, and so on. A multiplexor on a 64-bit bus was so wide that it would have to be isolated in one clock cycle – something that we successfully dealt with in the project, but would have no idea about before actually implementing it.

Chipscope has proved to be immensely helpful during the course of this project. For the purposes of debugging the MIG, for example, simulation proved much too cumbersome and unreliable; Chipscope was installed to examine the waveform of the communication between the MIG controller and the autogenerated MIG to see if the protocol is followed correctly.

## Difficulties

Many difficulties were encountered during the course of the project, the technical aspects of which will be highlighted in this section.

- Xilinx auto-generated code: we have had lots of trouble dealing with the code that Coregen produced, most notably the FIFO queues, which we used extensively in our design, and the MIG. FIFO timing, in simulation at least, is arbitrary and does not follow the timing diagram provided by Xilinx (for example, after writing to an empty FIFO queue, the data will not be available for an arbitrary number of cycles, though the timing diagram shows it to be available in the next cycle). Our FSMs had to be redesigned to handle this arbitrary waiting period. The sample MIG code show different timing protocol in both simulation and synthesis than found in the documentation, which caused lots of confusion.

- Timing constraints: our success in meeting timing constraints was built on weeks of examining critical paths, even using the FPGA Editor to look at the place-and-route data in the case between the core and the bus, and figuring out the most optimal way to pipeline signals with minimal disruption to the existing protocols around the changes.

## Words of Wisdom

Despite failing to meet our ultimate goal, we have learned many valuable lessons, some of which we will list below for future reference.

### Communication

Given such a large-scale project with a large team (originally four members, later reduced to three), we were not able to communicate as effectively as we had originally planned. This had implications for the following:

- Scheduling: despite having a schedule since the second week, we were not able to enforce it as strictly as we would have liked. Emails of inquiry went un-answered, group meetings and classes were un-attended, but there was a lack of a sense of urgency at the beginning of the semester to deal with this problem.
- Communication protocols: changes in the communication protocol on the memory bus were made several times during the project; however, not all members were made sufficiently aware of such changes, some of which have cost us days of work that need to be rolled back.
- Individual trust: despite lack of communication, we still placed trust in each other to complete the individually assigned portions without supervising each other. This has led to problems down the road when work that was expected to have been completed was found to be wrong, incomplete, or simply not started.

### Project Scope

We did not react in time to deal with the loss of a team member. The original project was designed to be completed with four members and required all four members to be able to put in the effort to complete his individual assignments per week. We did not take steps early enough to deal with the fact that one member has not been meeting the goals and there was no way to reach out to him despite our attempts.

We should have immediately scaled the project down to be manageable with three people instead of four. Having a difficult project for four people was already hard enough for four people; having the same project for three people was nearly impossible.

## Acknowledgements and Final Thoughts

### Andrew Drake

I learned a lot from this project. While I'm disappointed we didn't manage to get everything together in time to put together an awesome demo in software, numerous tests both in software and in software were pretty convincing that the hardware system as a whole is solid. Whether or not we had a demo to show off by Friday, we built a MIPS PC! I'm pretty happy we got that far, at least, and it was really cool to see projects from other class I'd taken coming together into something bigger.

We had a couple of problems contributing to the lack of a demo. We had a four-person project, and effectively two and a half group members. One disappeared off the face of the earth a few weeks in, and finally dropped on the last week. I took three project courses this semester, and so couldn't put in anywhere near as much time as Lei and Andrew. Not to mention that I got sick about two thirds of the way through the semester, eating even more of my time. Overall, we had less labor resources than we thought, so we collectively bit off more than we could chew.

If I could do it all again, I would have done a couple of things differently. We saw the schedule slipping early on, and rather than cut features to compensate, we vowed to work faster. By the end of the semester, we had completely burned our (voluminous) slack and debug times and, as close as we were, we didn't yet have a demo. We started cutting features and redefining the project a few weeks prior, but at that point it wasn't enough to save us. We probably would have been fine had we realized that we were understaffed earlier on. Personally, I would have taken at least one fewer project course.

My biggest piece of advice to future groups would be not to ignore problems that show up early on: it might be awesome to have all the features you initially thought of in the final project, but if you run out of time and don't have a demo, it's significantly less so.

## Lei Fan

I am disappointed with the fact that we did not fully finish the project. Looking back, we really have no one to blame but ourselves. We didn't follow the schedule as strictly as we should have and we did not hold anyone accountable. We never communicated effectively through Emails, and our group meetings were never attended by the entire group. We failed to scale down the project once we lost a group member – it should have been more obvious to us that we lost him long ago in the first place – and trying to tackle a project designed for four people with three people half-way through the semester was perhaps not a wise idea.

There were many things that I wish we could have done differently… Though I hate to be remembered through the history of 18-545 as one of those groups who would say "if only we had one more week…", the fact remains that we are one of those groups, going down posterity serving as a warning for the future generations.

However, I am proud of the fact that we were able to, in essence, build an entire computer system from scratch in Verilog. I can hold my head high and say to those future 18-545 students when I'm TA-ing in the future, "yes, we built an entire computer system on an FPGA in Verilog." It is something that I was told that I'd get a chance to do when I was a freshman and served to inspire me throughout my years in ECE; now that I have managed to do something similar to it, I hope that it would serve as an inspiration for the future generations.

There are many things that we could have taken away from this project. All the technical skills that I've learned in the courses prior to this were thoroughly tested during this class and were found to be inadequate. The largest scale project before this class that I took was 18-447, in which the datapath for the MIPS processor was provided without that many design decisions left up to you. This project, however, was entirely custom-designed from the beginning. The bus communication protocol, the DMA device registers, the way the core talks to the bus, so on and so forth, were of our own creation.

I wanted to be an architect – I got a tiny taste of it in this project, and I wish that I had more. I come out of this class knowing much more than what I did before – but, more importantly, I realize there was still so much more to be learned. This is only the beginning of a life-long journey – and I'm proud of the way it started. It was not the best, but it was something I am proud of.

## Andrew Israel

I am a little upset that we didn't fully finish our project; however, I still think the parts that we have finished are very impressive. We were able to essentially build all the hardware for a computer, but didn't have enough time to debug bug fixes in the kernel. I think we are yet another group to pull the "If we had another day, we could have finished it" card.

Designing a full system was much more complicated than I had originally anticipated. Whenever there was a problem we would initially have to figure out at what level the problem was, and then why it was occurring. For example, when we were debugging why DVI was hanging forever on a bus read, we needed to check the read request over the bus from DVI, the DVI controllers' registers, the write request over the bus from the core to the DVI registers, the address that crossed the clock domain from the core to the bus, the address translation in the core, the program that was running in ROM, and finally, whether or not ROM was copied over correctly. As it turned out the address translation was incorrect, so the wrong address was written to the DVI controller's register and DVI's read request would cause a bus error. Small bugs like this cost us a lot of valuable time and were also inevitable. The really dangerous thing about bugs like those are that they are so easy to underestimate and assume that they will take no time and then end up consuming a few hours.

In the end, I think our biggest problem was that we took on a difficult project for 4 people, and attempted to tackle it with 3 very busy people. Looking back, once we lost one of our group members, we should have changed the scale of the project immediately. On this note, whenever a team member begins a "disappearing act," we should have probably hounded him more than we actually did.

Another problem I think we ran into was we never really made anyone accountable for missing deadlines. Our original schedule was very ambitious so we all, at one point or another, didn't get something done on time, but after a certain point that became the norm and not the exception. We definitely could have done that better.

Finally, we also fell victim somewhat to the curse of premature optimization. We spent some amount of time attempting to get all the hardware (except the core) to meet timing, only to find out that once we added the core it would introduce a critical path so big that we would need to slow everything down. Those weeks we spent optimizing were essentially wasted.

Overall, this class has taught me a lot about both FPGA programming and the difficulties of working in groups with people who you do not know.

## Final Thoughts

In the end, we believe that we can be proud of what the three of us have achieved, despite not being able to meet our ultimate goal. It has been quite a learning experience for all of us, not just on a technical level, acquiring new skills and experiences with new tools, but also on a personal level, learning to work in groups with others who work differently and think differently. The estimated 20 hours per week per person that we have spent on this project were certainly well-spent.

We only have one last thought:

Be excellent to each other.