PixelPerfect

Kevin Avery, Scott Daw, Trevor Hill, Mac Wibbels
CS3710, Fall 2012, University of Utah

Abstract— **We discuss the design and implementation of an Instagram like application implemented on an FPGA in Verilog using a custom instruction set architecture that was designed specifically for this application, and the other custom hardware and software needed for the project. Various input output devices (I/O) were used to allow for hardware to software interaction, as well as user and hardware interactions. Limitations on design goals are presented in order to demonstrate challenges and overall scale of the project and its undertaking.**

I. Introduction

The project was designed to allow the user to capture an image with a camera, apply visual effects to the image, and then save the image back to a computer. Our implementation uses a webcam connected to a Windows PC, which is connected to a Nexys3 Spartan-6 FPGA. The FPGA requests and image from the computer, and the image is sent to the FPGA via UART, where it is stored in onboard Cellular RAM memory and drawn to a VGA monitor. The user can then cycle through a series of predefined visual effects, and then send the resulting image back to the computer, where it is converted to a JPEG image.

In this paper, we discuss the four major components of the system: the processor hardware implemented in Verilog, the assembler application implemented in C#, additional I/O drivers implemented in Verilog, and the final assembly application, written in a custom assembly language similar to CR16. We also discuss many of the engineering challenges in developing this system, and some of the limitations that had to be addressed.

II. Processor

The processor we designed implemented is based on the Baseline ISA, a simplified subset of the CompactRISC CR16A instruction set architecture that was supplied by the course instructors. Several further simplifications and modifications to the Baseline ISA were made to customize the behavior for the needs of our application.

The first version of the processor was a full implementation of the Baseline ISA. This had a 16-bit datapath and supported 33 instructions, including move, compare, store, load, branch conditionally, jump conditionally, shifting/arithmetic ops, logical ops, and more. It was implemented as a single cycle design, similar to the final design shown in Figure 1.

Some find it interesting that our processor is a single cycle design, with no state machine. The main difference between a multicycle design with a state machine and our design is the datapath controller implementation. Our controller is implemented by setting 10 datapath signals based on the instruction via combinational logic (ie. a case statement). The price of such a huge controller simplification is that the datapath must be thoughtfully designed so that any instruction can be accomplished in a single combinational path, with the result signals latched at the beginning of the next instruction.

The benefit of a single cycle design is speed. Consider a processor that uses a 100MHz clock but requires 5 cycles on average to finish an instruction. That would result in, on average, a 20MHz operating frequency. Although we ended up downclocking our processor for timing synchronization with the whole system, our datapath was rated at 78MHz by Xilinx Synthesis on the Nexys3 Spartan-6 FPGA. Since the core of our assembly application would be applying image effects, designing a processor to achieve maximum frame rates was of critical importance, and a frequency speedup of 3-4X can be the difference between choppy frame updates and incomprehensible frame updates.

One limitation of the Baseline ISA is that it uses 16-bit instructions, with 16-bit registers and mere 8-bit instruction immediates. A single image frame alone would require 18 bits to address, and because we wanted to address all 16MB of onboard Cellular RAM, our application required an address space of 23 bits. Therefore, the decision was made to upgrade to 32-bit instructions and datapath. This gives 24-bits of the instruction to the immediate, which makes writing assembly code easier since any immediate desired could be expressed in one instruction, which includes branch offsets.

Several simplifications to the Baseline ISA were also made, including removing instructions ADDC, ADDCI, SUBC, and SUBCI. The instructions that used the C were designed to incorporate the carry bit of the last instruction in the arithmetic of the current instruction, allowing for the chaining of multiple additions or subtractions to work on numbers too big to be done in one instruction. Since we had already expanded our datapath to 32-bits, and we knew we would not need to do arithmetic on numbers larger than that, these instructions were removed.

Besides removing instructions, we redesigned the shift instructions. The Baseline ISA defines 4 shift instructions (logical and arithmetic, with registers and immediates). The trouble is that the direction of the shift is determined from the sign of the operand. This caused each of the immediate based shifts to actually break down into 2 more instructions depending on the sign, for a total of 6 different shift instructions implemented in hardware, plus some ugly logic to convert the sign of the operand to the direction of the shift. By redesigning these shift instructions, we still ended

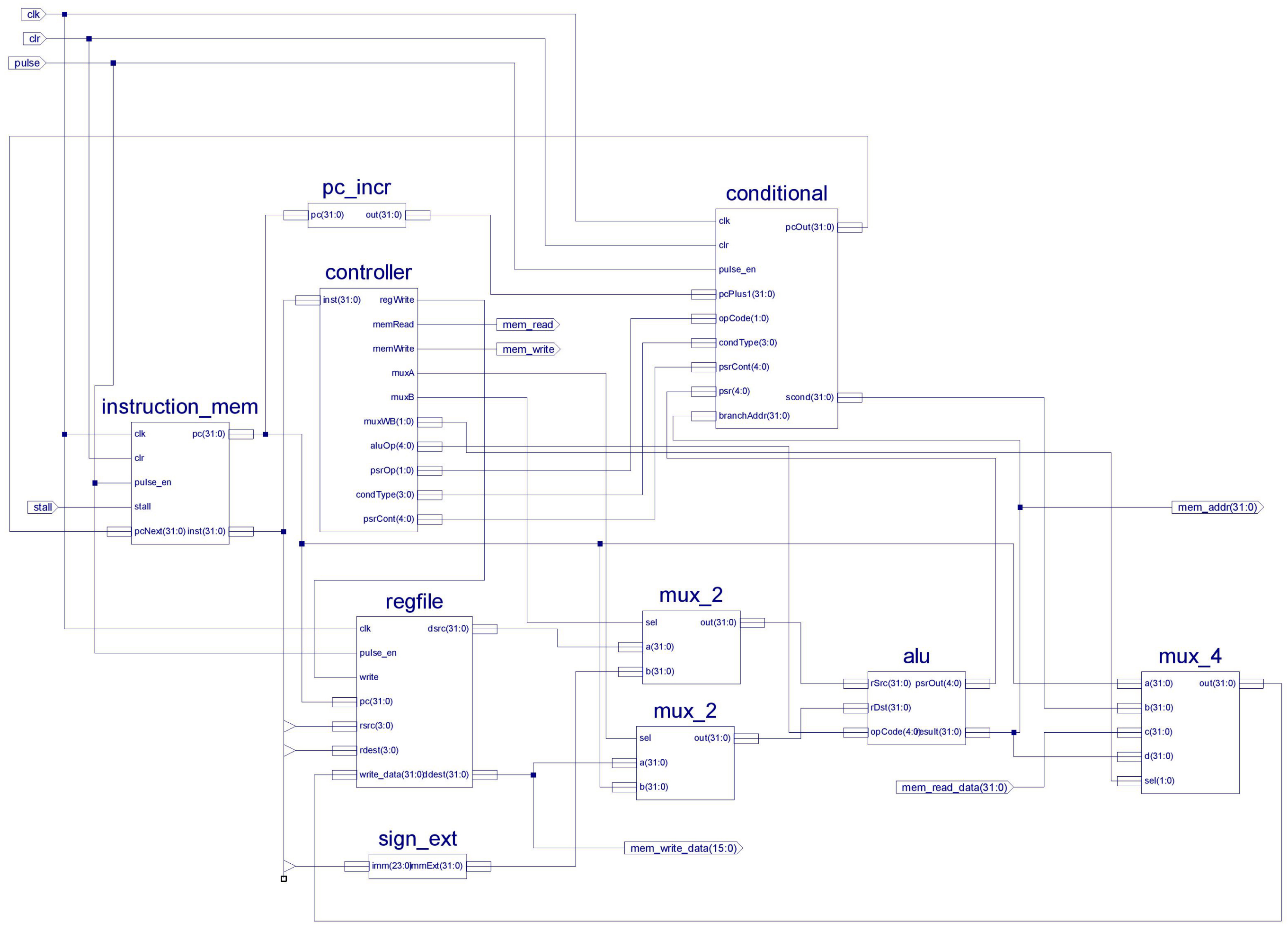

Fig. 1: Processor level schematic

with 6 instructions, but the shift amount operand is always positive, and the direction is specified by the instruction. This simplified the assembly code and the ALU logic (which also became a simple case statement).

Although the instruction memory was purely Block RAM, all other memory used in our system resides in the onboard Micron 16MB Cellular RAM chip. The RAM is configured to run at 50MHz, and a single read or write takes multiple cycles. This imposed a timing challenge for our processor, which was further complicated by the fact the VGA video pipeline needs first priority to access RAM. This was solved simply by adding a stall line in the processor. If the processor requests access to the RAM via a LOAD or STORE instruction, and RAM is busy, the stall line is set high, causing the processor to remain on the current instruction until the stall line is set low.

Lastly, in order to tune the processor frequency and synchronize with other hardware modules in the system, we used a pulse enable signal, with the pulse set high on a subset of clock cycles, where the frequency of the pulse is $\frac{100}{2^k}$MHz. This allows us to run the same 100MHz oscillator clock to every module in the system, and slow everything down by powers of 2 by adjusting the value of k. This was critical when interfacing with the UART, where the processor enqueues and dequeues data using FIFOs, and the clocks must be in sync.

With these customizations, our processor was fast, robust, and capable of everything required by the application.

III. Assembler

Writing code in hexadecimal notation is slow and error prone, so we implemented a simple C# Windows Form application that converts a custom assembly language into the hexadecimal instruction format that can be used to initialize the instruction memory on the FPGA.

Our assembler parses a .cr16 assembly file in 4 stages. In the first stage, it reads the assembly file into memory. In the second stage, it loops through the instructions over and over, decomposing pseudo instructions into actual instructions that are implemented in hardware. In the third stage, it maps all the assembly labels to addresses so that we can branch to a label offset. In the fourth and final stage, it decodes each instruction into the hexadecimal equivalent, according to our ISA definitions.

The assembler has a GUI that shows the entire assembly file on the right pane (with all the pseudo instructions decomposed) and the resulting hex instructions on the left pane. The assembly instructions and the hex instructions match line-to-line, so its easy to see what was generated. Furthermore, the application syncs the scroll position of both panes, maintaining the line-to-line match as the user scrolls through the assembly file.

The assembly process takes place on a background thread, and updates are posted to the GUI thread via Events. Only the most basic error handling is implemented: unrecognized instructions, oversized immediates, and other syntax errors that cause the parsing or decoding to fail. If an error occurs, a message box describing the error is presented to the user, and in the GUI the user can see the last line that was processed that caused the error. Its extremely lightweight, but also very effective. Once the user fixes the error in the assembly file, he or she can just click File ¿ Rebuild to have the assembler try again.

The assembler supports six pseudo instructions, instructions that are composed of other instructions. They are CALL, RET, PUSH, POP, STORD, and LOADD. These instructions are designed to allow for function calling using a program stack in memory. For example, the CALL instruction will push the current program counter and the current frame pointer onto the stack, then update the frame pointer to the next frame on the stack, and finally branch to the function label. Consider the following assembly code, which simply counts to a specified value supplied as a parameter:

```
// counts up to number supplied in $r0
f_count:
        push $r1
        movi 0, $r1
count_loop:
        addi 1, $r1
        cmp $r0, $r1
        bne count_loop
        pop $r1
        ret
```

This function could be called by the following code:

```
main:
        // count to 1000
        movi 1000, $r0
        call f_count
```

Program stacks and function calling is based on specific conventions. Our conventions were designed to be as simple as possible, while allowing us to accomplish everything we needed. For starters, parameters are passed using registers, with the first argument in $r0, the second argument in $r1, and so on. There are many reasons why a modern computer system would not use this convention, but it works great for us. Aside from the parameters, all other registers that a function uses are callee saved, ie. they should be pushed onto the stack at the start of a function, and popped back off at the end of the function. To make the assembly file more readable and help avoid label name conflicts, our naming convention is that all labels that are the start of a function have their function names prefixed with f and all labels that are internal to a function are prefixed with function name.

One of the challenges in implementing a stack on our hardware is that our word length is 32-bits, but our RAM only stores 16-bit values. The pseudo instructions STORD and LOADD solve this. They each decompose into six separate instructions, which include either two STOR or two LOAD instructions. The PUSH and POP pseudo instructions are simply wrappers that also adjust the stack pointer register up or down by two positions accordingly.

Overall, the assembler application was effective and provided the right amount of sophistication to allow for efficient assembly code development, but not requiring much time to build.

IV. Input/Output

A. Memory

This application relies heavily on the Micron 16MB Cellular RAM chip on the Nexys3. Because every individual pixel is saved, the amount of data that needs to be stored far exceeds the capacity of the available FPGA BRAM. The two modules that require access to memory are the processor and the video pipeline. Since both cannot access memory simultaneously, the need for a module to arbitrate the accesses arises.

The dual port frontend module, also referred to as the arbiter, is responsible for interleaving requests from the video pipeline and the processor, and stalling the processor if necessary. In general, issuing a single read or write to RAM would take ten 100MHz clock cycles to complete, with the data becoming available on the 9th clock cycle. Therefore, the arbiter must control the stall line to the processor, and it must latch the read data from memory, so that the data is available when the processor begins the next instruction.

To make a request to memory, the module making the request can issue the command as if it were talking directly to memory. The arbiter gets these signals first manages requests to the actual RAM controller module.

To initiate the request, the arbiter sends a read or write signal to the RAM controller. This indicates to the controller to visit the address and perform the requested read or write. If it was a read, the controller outputs a data_ok signal to indicate to the that the data is ready. No matter what the operation was, it also sends a signal to indicate the operation has completed successfully and is ready for the next request. The data must be registered on the next clock cycle, as it is not guaranteed to be preserved afterwards. The memory also supports a burst read and burst write feature. When a burst is requested, the memory will visit each linear increment of the initial address, up to the end of the line, or 128 addresses.

B. Video Pipeline

The video pipeline module is responsible for two primary tasks. First, it must generate the necessary horizontal sync (h-sync) and vertical sync (v-sync) pulses. These pulses are what drive the VGA and determine resolution and refresh rate. To do this, a VGA controller module is used. The VGA controller has two different internal counters, one for the h-sync and one for the v-sync.

The second task is to produce the correct RGB outputs. This is accomplished by keeping a small internal dualported BRAM buffer which gets periodically filled by burst reading lines from the frame buffer in RAM, where each line is 128 addresses. In order to keep things running smoothly, the size of the buffer is four lines, or 512 addresses, or 1024 pixel bytes, and two lines are bursted out of RAM at a time.

This allows half of the buffer to be read at 25MHz to the VGA, while the half is being filled up with the next two lines from RAM at 50MHz. To keep the buffer data current, as soon as the VGA finishes with one half of the buffer, it is immediately updated while the VGA continues into the next half. This process repeats indefinitely, reading through the entire frame buffer over and over, in order to keep the screen refreshed.

C. Buttons

For the user to interact with the application, the hardware was designed to accept button clicks to determine what the user wanted to do. The control buttons are three push buttons mounted on the Nexys3. There is no special hardware for latching or debouncing, but those issues are accounted for in software. For example, if the application is waiting for user input, it will loop over some instructions that repeatedly read the values on the switches, and breaks out of the loop when a switch is high. This is effective because the processor can sample the buttons much faster than a user can press and release a physical button.

D. UART

A 115200 baud UART was integrated into our design in order to transfer information between the computer and FPGA. Due to the asynchronous nature of the UART, there is a FIFO for both the send and receive lines of the UART. Since we send entire images back and forth over UART, the size of these FIFOs would ideally be the size of an image. Of course, there is not enough BRAM to allow that, so instead we adopt the policy of only sending one horizontal image line at a time, and then waiting for an acknowledge before sending the next line. Since the images are 640 pixels wide, the FIFOs only need to be at least 640 bytes, so we used 1KB for each FIFO. Then the we needed to expose the FIFO counts to the rest of the system, so that the processor can issue a UART read and dequeue data when it is available.

E. Memory Mapped I/O

A memory mapped I/O module allows the processor to interact with the various I/O using standard LOAD and STOR instructions by setting particular addresses. Any address with the 24th bit low is assumed to be using first 23 bits to address RAM. However, addresses with the 24th bit high may read or write to the UART, read the byte count in the UARTs send or receive FIFO, read from the buttons, or write to the LEDs (for debugging), depending on the remaining bits of the address.

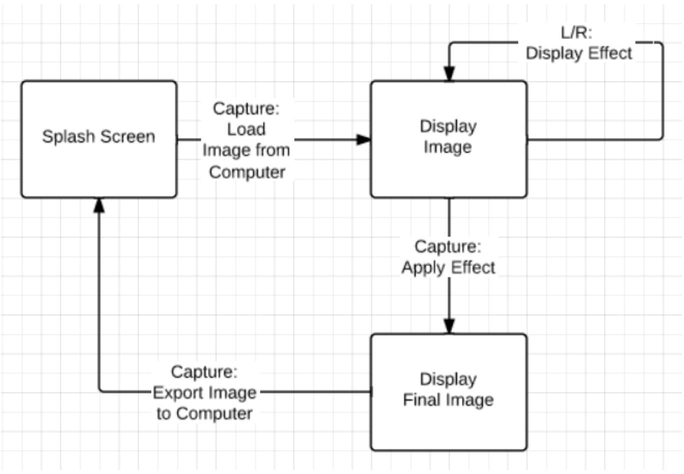

Fig. 2: State diagram of the application

V. Application

The goal of our application was to provide an Instagram like experience, in which the user could capture an image, choose an effect to apply to the image, and save the image with the desired effect. With this goal in mind, we were able to break the main application up into three main states. Figure 2 below shows the state diagram for our application.

The application begins by displaying a splash screen with the team logo and graphic. When the User hits the Capture button, the board requests an image from the computer over UART, and loads the image into memory. After loading the image, the user can preview effects using the Left and Right buttons. Applying the effect is achieved by hitting the Capture button again, upon which, the board loads a new screen displaying the final image, and prompts the user to either export the image to the computer or return to the splash screen. However, due to time constraints, the user must export the image.

Each application state is implemented using a function call. The main application consists of four function calls: f_do_splash_screen, f_get_image, f_do_choose_effect, and f_do_show_last_screen.

f_do_splash_screen: The f_do_splash_screen method calls f_draw_spash_screen, which calls f_move_image with parameters 0x4B000 and 0, which copies an image from a offset 0x4B000 to offset 0, which is the offset that the VGA controller reads from. After moving the image, the function calls f_get_button, which waits for a button to be pressed and returns the code of the button that was pressed. The function f_draw_spash_screen repeatedly calls f_get_button until capture button is pressed, upon which the function returns.

f_get_image: The f_get_image function is responsible for requesting an image from the computer and copying sent data from the UART FIFO to two separate frame buffers at RAM starting at addresses 0 and 153600, as seen in Figure 3. The frame buffer at starting at address 0 is where the VGA video pipeline reads from. The second frame buffer at address 153600 is hold a copy of the original image, so that

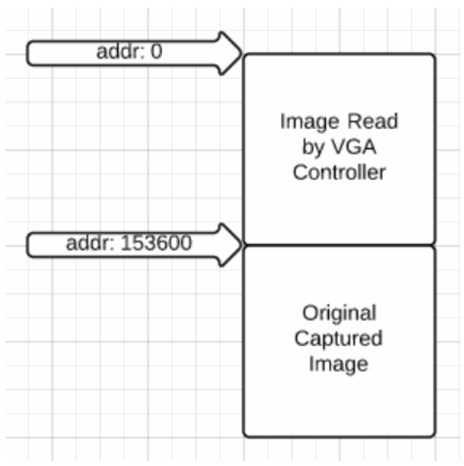

Fig. 3: Diagram of image offsets in memory

each image effect is apply to the original.

The f_get_image function requests and image by sending 10 bytes of 0x0A. After requesting an image, the method repeatedly polls the UART FIFO to determine if any data is available, if at least two pixels have been received, the method saves the data and resume the polling loop until all of the pixels one row of the image have been received. Then it requests the next line by sending another 10 bytes of 0x0A, until the entire image has been transferred and copied into RAM.

f_do_choose_effect: The f_do_choose_effect function calls f_apply_filter, which uses left and right button presses to increment and decrement a counter. The value of the counter is used to choose the effect. The f_apply_filter function calls the filter function specified by the index, passing a source parameter of 153600, and a destination parameter of 0, these parameter cause the filter function to read from offset 153600 save the filtered image at offset 0. This methodology allows the application to repeatedly apply filters as well as automatically update the image that is read by the VGA controller. Upon receiving the capture button code, the f_do_choose_effect function overwrites the original image with the specified filter by calling the filter function with a source parameter of 153600, and a destination parameter of 153600. The f_do_choose_effect function then returns.

f_do_show_last_screen: The f_do_show_last_screen function is responsible for drawing the last screen GUI, which allows the user to select between exporting the image to the computer or returning to the splash screen state. Although the team didnt have sufficient time to fix a bug preventing the application from returning without exporting, fortunately, the GUI functionality is still implemented. The f_do_show_last_screen function draws a GUI with the proper selected item by calling f_draw_last_screen_gui with either a 0 or 1 as a parameter. The f_draw_last_screen_gui function loads the GUI image by selecting either offset 0xA5000 or offset 0xA1400, and stores the image into offset 138240, which creates a GUI at the bottom of the image being displayed by the VGA controller. After returning from

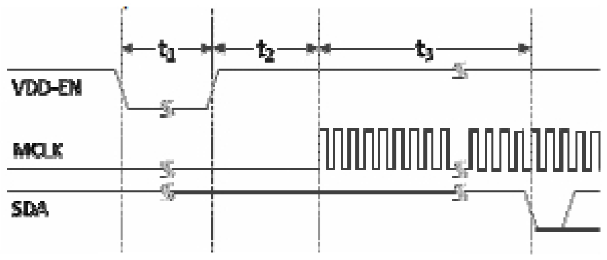

Fig. 4: Timing diagram for VmodCam power up sequence

f_draw_last_screen_gui, the function enters a loop which calls f_get_button, if the select button is pressed, the function either exports the image to the computer by calling f_send_image and then returning or simply just returns.

This sequence of states is repeated in a loop so that images can be taken repeatedly from the computer.

VI. Limitations

Originally, we wanted to use a the VmodCam stereo camera module that attaches directly to the Nexys3 via VHDCI port. However, the initialization and communication protocol proved to be challenging. Communication with the module is performed through a two wire interface (I2C), in which a master node uses two lines, the Serial Data Line (SDA) and the Serial Clock Line (SCL), to serially transfer instructions with respect to the a clock passed through the SCL line. The interface employs two bidirectional open drain lines to allow for alternating master and slave nodes. In the board to camera communication, the camera module always acts as the slave device. Camera initialization requires the board to enable the camera modules voltage supply rails with strict timing constraints in relation to the clock, as seen in Figure 4. Additionally, image configuration requires a number of instruction to be issued in order to write image parameters to registers in the on the camera chip. The only example project that we could find for using the VmodCam was written for the ATLYS board and was implemented in VHDL.

Although the example VHDL project provided a module for the I2C camera initialization and communication, the project used DDR2 RAM on the ATLYS to create two image buffers. In order to integrate the camera controller module with our project, we built a new module with a buffer to allow the camera to write into the buffer and then burst write data into ram. Xilinx was unable to synthesize the design for our board. Given the limited amount of time that was left for the project and the amount of work that had to be done on the application as well as integrating the UART with the system, our team removed the VmodCam from the system project and decided that it would be better to pursue data capture from a webcam connected to the computer.

We also built a custom peripheral: a plexiglass dashboard with 3 light up arcade buttons, and a circuit that connects to the PMOD ports on the Nexys3. Unfortunately, using it seemed to cause pin conflicts with the rest of our hardware. We tried several different sets of ports, but we could not get it to work consistently for the demo, so we had to settle for the onboard push buttons.

Another problem we experienced seems to come from the Digilent Adept tool used to program data into Cellular RAM on the Nexys3. This tool was used to load our GUI images into offsets in RAM, but the images contained artifacts when they were drawn back to the screen. The images captured by the application did not suffer from the same artifacts. One workaround would be load the GUI images into RAM as part of the application startup sequence, rather than relying on the Digilent tool.

VII. Conclusion

We successfully developed the system that allows users to capture an image, apply visual effects, and send the image back to a computer. There were a great number of difficult technical challenges along the way, but by focusing on implementing the most important features of our design we were able to deliver a working demonstration at the end of the semester.

In hindsight, one of our biggest challenges was interfacing with a separate RAM chip. But as a testament to the robustness of our memory system, consider that our final assembly application had over 40 function calls and over 60 push/pop pairs, resulting in over 2300 instructions. Since several functions get called repeatedly, especially as different effects are applied, in the course of a single picture being taken and sent back to the computer, over 1000 stack manipulations take place before the application is ready to take the next picture. Now consider that the processor is also manipulating the frame buffers, bringing the total number of individual memory operations into the millions, and all of this interleaved with the video pipeline burst reading. It is quite an impressive feat for all the hardware and software working together to make the application a success.