**The Epoch: An FPGA Smartphone**

**David McGinnis**

**Software Engineer**

**mcginnda@rose-hulman.edu**

**Kyle Green**

**Computer Engineer**

**greenkt@rose-hulman.edu**

**Submitted for the 2011 Digilent Design Contest**

**Terre Haute, Indiana**

**5/6/2011**

**Advisor: Dr. Mihaela Radu**

**Rose-Hulman Institute of Technology**

**Terre Haute, Indiana**

# Introduction

**Abstract**

This project incorporates many different disciplines, including Computer Architecture, Embedded System Design, Compiler Construction and Operating System (OS) Design to make a fully-functional smartphone which is able to load an OS and apps from an SD card, and use a touchscreen interface to run apps and make calls.

**Objectives**

This design’s main goal was to make a computing platform which was able to be developed for by any competent developer, and could be easily used by anyone who is reasonably comfortable with a console-based environment. The design should be able to make loading the OS, and running programs as easy as possible, covering up the complexities of the various interfaces to which the system is connected to (i.e. PS/2 keyboard, microphones, speaker outputs, SD Card, etc.).

**Features-in-Brief**

Below are all of the features of the system:

* Programs can be loaded from an SD card formatted in the proper architecture.
* The OS is automatically loaded on system boot.
* The system allows software interrupts, allowing user-space programs to communicate with the OS.
* Memory is segmented in order to allow multiple programs to be in memory at once.
* The SD card is formatted to allow up to 16 programs to be saved on the card.
* Each program has space for metadata about the program for the OS or bootloader to use.
* Touchscreen and physical keyboard interface, allowing easy, customizable input.
* Phone interface, allowing you to make and receive calls using a microphone and headphones or speakers.
* VGA output for easy viewing of screen’s text and graphics.
* Minimal GPU for easy displaying of colored boxes and text.

**Project Summary**

The design itself is split into two major regions, each of which are split into more regions. Below is a view of the hierarchy of these regions:

* Hardware Solution
  + Processor
  + GPU
  + Phone Interfacing Hardware
  + Touchscreen Interfacing Hardware
  + VGA Interfacing Hardware
  + SD Card Reader
* Software Solution
  + Assembler
  + C Compiler
  + Operating System
  + Apps

The compiler and assembler were written in C++ using Visual Studio 2010, while the operating system and apps are written in C and compiled using the custom compiler. All hardware solutions were written in Verilog, with physical discrete hardware components as needed.

This design separation was very effective given the team. David was able to work on the software side of the solution independently from Kyle, who implemented most of the hardware solution. Daily meetings facilitated the exchange in ideas and design changes for the project, and helped keep the design cohesive. Things like multipliers, which would be better implemented in hardware, were instead implemented in software, due to space constraints in the processor. Despite this, a multiplier was still able to be added, and works just fine, just a little less efficiently than a hardware solution would have. There are a few examples like this, where the separation was not the best, but overall, the design is effective and efficient. All objectives were easily met, with an easy interface in the assembly to retrieve data from and send data to peripherals. Each peripheral driver could be easily taken and used in other parts, as could the compiler and OS with some modifications to retarget the compiler.

While the design was developed for the Nexys-2 with certain peripherals, it could be used on any FPGA board which could store the appropriate amount of circuitry required by the project. Only the mappings would need to be updated, and peripherals could be left off if not available. This makes the system very modular and portable, increasing the value of the system as a whole.

**Digilent Products Required**

The following Digilent products were required to implement this system:

* Nexys-2 Board
* PModSD
* PModAMP1
* PModMIC
* PMod-DA2
* PMod-AD1 (2)

**Tools Required**

The following tools were required in the implementation of the system:

* 2 GB SD Card
* VGA Cable
* VGA-Enabled Monitor
* PS/2 Keyboard
* Xilinx ISE
* Visual Studio 2010
* Team Foundation Server (TFS)
* Windows 7
* Windows Powershell
* Ubuntu 10.04 (although any BASH-enabled OS will work)
* dd utility in BASH
* Coco Parser-Generator
* Notepad++ for Windows Hex Editing
* Emacs for Linux Hex Editing

**Design Status**

At this point, it is projected the project is 80% completed. The main CPU portion is complete, and the assembler and compiler are mostly complete as well (barring a few bugs here and there that are still being found in all three).

On the hardware side, the GPU needs to be enhanced to allow colored blocks and clearing of the screen, which should only take an hour or two at the most. Interfaces for the keyboard, touchscreen and phone also need to be added. The keyboard will only take an hour or two as well, while the touchscreen could take up to 5 engineering hours. The phone is unknown at this time, due to issues with making it work alone. This is estimated at the moment, however, as 50 engineering hours.

On the software side, the OS needs to be implemented completely, pending the keyboard and touchscreen’s completion. This should take between 10 and 30 engineering hours. Thus, overall, the estimated time needed is around 100 engineering hours left to finish this project completely. We hope to have the keyboard and a basic OS done by Friday for the presentation, with a stretch goal of making the touchscreen work.

# Background

**Why This Project?**

This project was of particular interest to both of the designers, since we both enjoy working on the line between hardware and software, leaning towards different sides of the line. This project also provides a chance to explore new design models for phone processors, as well as real life testing of operating systems concepts.

This system also functions as a prototype of an idea of low-cost, but effective computing systems, which, with a good manufacturing process, could be brought to poverty-stricken areas of the world, in order to give them the technology they need to communicate with people all over the world, and help better bring their problems to light to the public of first-world countries such as the U.S.

A final way in which this system could be used is for education. It is reasonable to imagine a school system where a system such as this would be given to students in elementary school, and as they go through the school system, they would dig deeper into the system until high school, where they might have classes which would modify and add-on to the existing architecture. This would provide a great introduction to the world of Computer Science and Computer Engineering, and would help get more young students excited in this field.

The approach with which we developed this system was based on both of our strengths. Kyle specializes in Verilog and hardware development, and is skilled at interfacing with various protocols to peripherals. Meanwhile, David is skilled at higher-level software systems, especially compiler construction. Operating Systems are also something he was interested in, so developing one using a compiler he has written worked well.

**Reference Material**

This system must interface with many different peripherals in order to work. Below is a list of all of the protocols which it must interface with, along with links describing each of them.

* Nexys-2 Manual (Useful for PS/2 module and Seven-Segment Display)
  + <http://www.digilentinc.com/Data/Products/NEXYS2/Nexys2_rm.pdf>
* SD Card
  + <http://alumni.cs.ucr.edu/~amitra/sdcard/Additional/sdcard_appnote_foust.pdf>
* SD Card Model (from SD Card Interface project for testing our SD card reader)
  + <http://www.digilentinc.com/showcase/contests/designcontest.cfm?ContestID=4>
* Resistive Touchscreen
  + <http://www.sparkfun.com/tutorials/139>
* Cell Memory
  + <http://www.micron.com/products/dram/PSRAM-CellularRAM.html>
* Phone
  + <http://www.electronickits.com/kit/complete/tele/ak-700.pdf>

## Design

**Features and Specifications**

The following features are supported by the Epoch:

* 32-bit processing power
* Easy-to-use MIPS-like assembly code
* Hardware and software interrupts
* Ability to load data or instructions from an SD card
* Support for segmentation of programs
* Easy seven-segment display for displaying debug information
* VGA adapter for easy-to-read standard output
* GPU in order to allow basic shapes and text to be drawn to VGA display
* Home and reset buttons, to allow navigation through phone OS.
* Connection to hardware phone, to allow calling to other phones, using a hardware cable.
* Assembler and tools for assembling and loading programs onto phone.
* ANSI C compiler for higher-level development of code.
* Simple OS which can load programs and controls I/O for system.

**Design Overview**

Below is the overall architecture of the system, as seen from the side. Each piece was built separately, but simultaneously, ensuring close connections between all of the parts. David designed and developed most of the software system, while Kyle designed and developed most of the hardware system, with the original processor design coming from David. Constant communication ensured that the two parts mesh very well together.

Figure 1: Physical Architecture of System

The lowest level of circuitry was the various peripherals: the resistive touchscreen, the phone, VGA, SD card interpretation, and the seven-segment display. These were accessed utilizing a mixture of physical hardware and Verilog circuit definitions, which deciphered the input signals to be used by the processor in an easier fashion. Each individual peripheral was different, and will be explained by them in the Detailed Design Description section later.

On top of this was the main portion of the system: The BitEpicness Processor. This processor is a 32-bit processor, but processes 16-bit instructions. This allows us the maximum addressing space for instructions, since we can address up to 232 instructions. The processor supports a MIPS-like architecture, although there are unique design elements in the architecture, which will also be discussed later. Read port and write port instructions interface with the peripherals below it, using a system of port numbers and register values to interact with them.

Finishing the hardware portion of the project, above this we have the BitEpicness assembler, which assembles various assembly instructions into either coe or bin files which can be loaded onto the phone itself. Coe files are used mainly for loading onto the phone directly, for debugging purposes. This method bypasses the OS completely, but takes out many factors in development. Meanwhile, the bin file can be combined using other tools in order to make an SD card image, which can be loaded onto the phone.

An ANSI C compiler then compiles standard C code into assembly code for the BitEpicness assembler to assemble for the phone. The C compiler has been checked and tested to follow all ANSI guidelines, and allows inline assembly code, for use in the standard libraries, which are also implemented. Libraries also exist to interact with the touch screen and draw graphics on the VGA output.

This compiler can compile user programs, as well as the McOS operating system. McOS uses a simple segmentation scheme in order to allow one program, plus itself, to run at a given time on the phone. McOS controls all I/O routines using software interrupts, and controls the lifecycle of programs.

**Detailed Design Description**

#### The Epoch Processor

The Epoch processor is a 32 bit processor with a MIPS-like instruction set. Its architecture has been modified so that it supports 16 bit length instructions. As with the basic MIPS processor, the Epoch processor uses a five stage pipeline.

The processor automatically forwards data to the proper stages when necessary and possible. When data is accessed from the Memory Fetch stage it is not possible to forward the data, so the processor automatically stalls for one cycle. This allows users to write assembly programs without having to know the architecture of the datapath, as well as saving memory by not requiring nop instructions

The BitEpicness processor contains 33 native instructions, all of which are listed below, along with their opcodes (bits at the beginning of the instruction), func codes (when applicable), which are appended at the end of the instruction, and their function. Instructions have opcodes of either four or five bits, and some instructions include three bit long function codes. Any addresses used in the instructions are interpreted as absolute, extending zeros for any extra bits.

One interesting set of instructions is the ldl and ldi instructions. These instructions actually take up 32 bits, the first half of which contains the op code and register which the data should be loaded into. The second half allows a full 16-bit number to be loaded into the register. If ldi is used, this number will also be sign-extended before being put into the register.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction Name | Instruction Mnemonic | Op Code | Function Code | Comment |
| Jump | J | 00000 | --- | Jumps to immediate value |
| Jump and link | Jal | 00001 | --- | Saves return address to $ra |
| Branch if equal | Beq | 1010 | --- | Branches to register value if other two registers are equal |
| Branch if greater than | Bgt | 1000 | --- |  |
| Branch if less than | Blt | 1001 | --- |  |
| Branch if not equal | Bne | 1011 | --- |  |
| Load word | lwn | 1110 | --- | Loads single word (16 bits) |
| Store word | Swn | 1111 | --- | Stores single word (16 bits) |
| Load double word | Ldw | 1100 | --- | Loads two words (address and address + 1) |
| Store double word | Sdw | 1101 | --- | Stores two words (address and address + 1) |
| Addition Immediate | Addi | 01100 | --- | Adds register value and immediate |
| Logical Or Immediate | Ori | 01101 | --- | Ors register value and immediate |
| Shift Left Logical Immediate | Slli | 01111 | --- | If immediate is negative, value is shifted right |
| Addition | Add | 01110 | 000 |  |
| Logical And | And | 01110 | 001 |  |
| Jump Register | Jr | 00010 | --- | Jumps to register value |
| Transfer Registers | Move | 00100 | --- |  |
| Negation | Neg | 00110 | --- |  |
| Nop | Nop | 00100 | --- | Implemented as move from $0 to $0 |
| Logical Not | Not | 00111 | --- |  |
| Logical Or | Or | 01110 | 010 |  |
| Shift Left Logical | Sll | 01110 | 011 | If second value is negative, value is shifted right |
| Subtract | Sub | 01110 | 110 |  |
| Load Immediate | Ldi | 00101 | --- | Loads 16-bit value, see above. |
| Load Lower | Ldl | 01001 | --- | Loads 16-bit value, see above. |
| Write Port | Wp | 01000 | 010 | Writes to a peripheral |
| Read Port | Rp | 01000 | 011 | Reads from a peripheral |
| Disable Interrupts | Disi | 01000 | 100 |  |
| Enable Interrupts | Eni | 01000 | 101 |  |
| Software Interrupt | Swi | 01000 | 110 | Causes software interrupt |
| Jump and link register | Jalr | 00011 | --- | Same as JAL, but uses register value |
| Move to Coprocessor 0 | Mtc0 | 01000 | 000 |  |
| Move from Coprocessor 0 | Mfc0 | 01000 | 001 |  |

Figure 2: Supported Instructions

The Epoch processor contains sixteen 32-bit registers. Thirteen of these are general purpose registers. The other three are modified by other instructions or not assignable. As with MIPS processors, register 0 ($0) always contains the value 0. If any value is written to it the result is discarded. Certain instructions (such as all ALU instructions) do not contain a save register. In these cases, the result is stored in the result register, register 14 ($rr). When a function is called using JAL or JALR, the return address is stored in the return address register, register 15 ($ra).

The Epoch processor contains a simple 32-bit Algebra and Logic Unit, or ALU.

The processor uses the built in Micron CellularRAM to allow for 16 megabytes of instructions and data. The RAM is used in asynchronous mode for easy access. To allow for instruction and data loading, one clock cycle of the processor is partitioned into four parts. The memory module first deasserts the memory’s chip enable as required by the RAM, and then reasserts the signal. Next, the module reads the memory for the instruction and stores it in a register to output to the processor. The next cycle reads or writes data at the data’s address. If it is a 32 bit store, it writes the upper 16 bits, otherwise it writes the lower 16 bits. Finally, the module reads or writes the data at the address one above the data address if it is a 32 bit read or write.

The Epoch processor also has a coprocessor module. This handles operating system type functions, such as reading from and writing to external modules, causing and handling software interrupts, etc. This module is located in the memory fetch stage, so it will stall the processor when necessary. For example:

mfc0 $s0, $epc

jr $s0

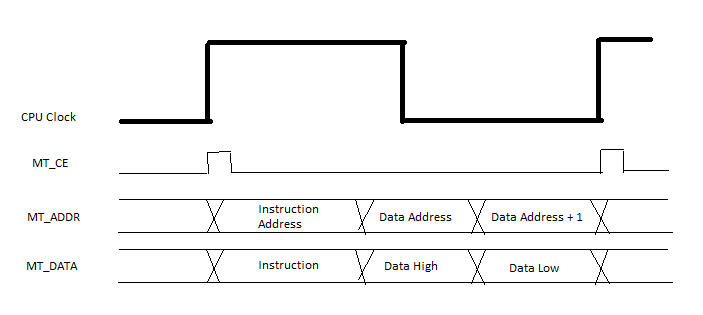


Figure 3: Timing Diagram for Memory Operations

The JR instruction relies on $s0, which is not available until one cycle after it normally executes. The StallDetector module will detect this in the Instruction Fetch stage and stall the JR instruction for one cycle.

The Epoch processor supports an easy to use and adaptable interface to communicate with external modules. Each device is connected to a 32-bit bi-directional bus, as well as a write enable and read enable flag. The modules also can send an interrupt signal back to the processor. The processor (or more specifically, the coprocessor module inside the processor) controls all communications on the bus by sending the write and read signals to the correct external modules. The processor can support up to sixteen external modules.

Additionally, any port larger than 32 can be used during software interrupts, in order to denote different types of interrupts. Below is a listing of all of the available ports, and their uses in the processor. Note that these numbers are adjustable, as new peripherals are added to the phone.

|  |  |
| --- | --- |
| Port Number | Use |
| 0 | Seven Segment Display |
| 1 | Millisecond Timer |
| 2 | Switches |
| 3 | VGA |
| 4 | SD Card |
| 5 | Keyboard |
| 6 | Touchscreen |
| 7 | Phone |
| 8 | Audio I/O |
| 9 | Not Assigned |
| 10 | Not Assigned |
| 11 | Not Assigned |
| 12 | Not Assigned |
| 13 | Not Assigned |
| 14 | Not Assigned |
| 15 | Not Assigned |
| 16 – 31 | Not Available |
| > 32 | Software Interrupt Use |

Figure 4: Port Assignments

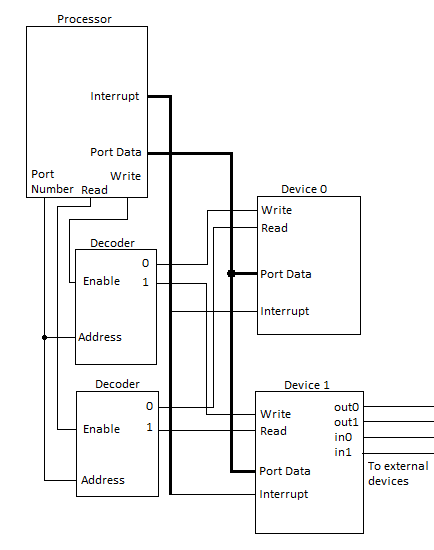


Figure 5: External modules connected to processor

#### Resistive Touchscreen Peripheral

The touchscreen module is used to drive a touchscreen. The touchscreen works by applying a voltage across one layer of the screen. When the screen is touched, the voltage at that point is transferred to the other layer and can be read by an analog to digital converter. The touchscreen driver does this repeatedly, alternating between driving the voltage vertically and horizontally. With this information an x and y coordinate can be determined. The touchscreen is a replacement Nintendo DS Lite screen.

#### Phone Peripheral

As neither of the designers are experts in analog electronics, a landline telephone kit has been bought to use as the connection between the Nexys2 board and the telephone line. The FPGA board is connected to the microphone and speaker using digital to analog and analog to digital converters, respectively.

#### VGA Peripheral

The VGA supports a screen resolution of 256 pixels wide by 320 pixels tall, 3 bit color. It is controlled by sending commands to a simple GPU. This GPU can draw pixels, ASCII characters, and filled boxes. The processor can communicate with the VGA by writing commands and values to the port which the VGA module is connected. The format of these commands is given below.

|  |  |  |
| --- | --- | --- |
| Bits 31-16: Ignored | Bits 16-9: Command number | Bits 8-0: Value |

Command:

1: Sets the X location to the value in bits 8-0

2: Sets the Y location to the value in bits 8-0

3: Sets the current color to the value in bits 2-0

4: Draws the ASCII character given in bits 7-0 at the X and Y location

5: Sets the lower right X location to the value in bits 8-0

6: Sets the lower right Y location to the value in bits 8-0

7: Draws a pixel

8: Draws a filled box

While the VGA module is busy drawing, it outputs the value 1 when read. The program must wait for this value to change to 0 before it can send new commands. This prevents glitches in drawing.

#### SD Card Peripheral

The SD card module communicates to the SD card using the SPI protocol. In this protocol, the SD module sends a command serially over the MOSI line, and the card responds by sending an eight bit response. In the case of a block read command, the SD card then sends a 512 byte response.

The SD card module starts up by initializing the SD card. It initializes by sending command 0, and then command 41 and 55 repeatedly until the idle bit in the response (bit 0) is 0. When a block is to be read from the card, command 17 is sent with the address of the block to be read.

The block of data is read into a temporary block of block memory. After the block is read, the SD module stalls the processor and takes control of memory. It then transfers the data into the main memory and releases the stall on the processor. Finally, the SD module sends the processor an interrupt to signal the finished loading.

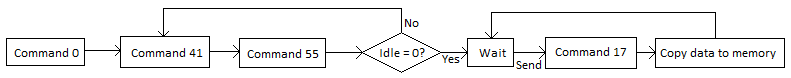


Figure 6: Flowchart of SD card initialization, reading blocks

The SD card module uses a similar style of command as the VGA for the processor to communicate with it.

|  |  |
| --- | --- |
| Bits 31-30: Command | Bits 29-0: Address |

Command:

0: Sets the block number to the given value. This address is not the physical address, but the physical address shifted right 9 bits.

1: Sets the address in memory to save to.

2: Starts a read from the SD card

#### Seven-Segment Display and Switches Peripheral

The seven-segment display is a simple display which outputs a hexadecimal value to the user. The display by default only displays the lower 16 bits, but it will display the upper 16 bits when the seven-segment display button is pressed. The programs can also access the switches on the FPGA board. These peripherals are mainly to be used for debugging purposes.

#### BitEpicness Assembler

The assembler’s design is very simplistic, in order to reflect the simplistic nature of its purpose. Most of its syntax is C syntax, but it is compiled using the Microsoft C++ compiler, in order to easily integrate with the MiniC Compiler, which does use C++ syntax. The assembly language itself is described in a separate document, submitted with this report. This section serves only to describe the design and architecture of this particular implementation of the assembler.

The three header files are the Assembler.h, Constants.h, and DataStructures.h files. Assembler.h includes the function declarations for all functions. The Constants.h file includes all of the #define constants which are used throughout the code, including error messages, processor specifications, and string lengths. The last header file, DataStructures.h, includes the enumeration defining the different type of operations available, as well as the ErrorHandler struct, which is created in order to collect any errors found during processing.

The source code file contains the majority of code for the Assembler. At the top, are initializers for many different arrays. These arrays define the operations allowed, the op code for these operations, the func code (where valid) of these operations, as well as registers and their translations. This design allows easy modification of operations, as long as the format is the same as an existing operation. Note the size of these arrays is controlled by constants defined in the Constants.h header file.

The next section (as defined by #pragma region preprocessor directives) contains functions which can be used with little or no modification in other programs, and are generally simplistic in nature. Each of these functions have descriptions above them, so not much will be described here. The functions in this section include functions for displaying the help output, checking the heap, replacing extensions in file names (for automatic output file creation), getting the size of a file, reading a coe file for unit testing, and translating a character to a string of zeros and ones.

The next section contains functions which focus on reading and writing assembly files. These include functions to get the number of instructions in the assembly file, to separate all of the instructions into an array of strings containing an instruction in each element of the array, to clean those instructions, eliminating white space which can cause issues, to write the assembled byte code to file and to write raw characters to a binary file for use with the SD card, These functions would be useful in other programs, although not as much as the utility functions.

The next section is the basis for the assembler as a whole. The functions focus on interpreting input, and almost exclusively are the users of the arrays described at the top of the source code file. They function to retrieve op codes, function codes and op types depending on the operation name, translate the register input into binary, create a list of labels and addresses which those labels refer to, remove addresses from instructions for translation and convert decimal to a string of binary numbers. These functions are called often, but are not modified often due to the arrays, but some are modified at times for additions to the assembly language.

The largest, and most important, section is the Processing section. This contains two functions: ProcessPseudoInstructions and ProcessInstructions. ProcessPseudoInstructions currently replaces the instruction:

la $k0, L1

with:

ldl $at, L1[15:0]

ldl $k0, L1[31:16]

slli $k0, 16

add $at, $rr

move $k0, $rr

Because there are more instructions in the result, the space is made while gathering addresses for these extra instructions. This function then extracts the data needed, checks for errors, and then replaces the instructions needed. The second function is the heart of the Assembler, and calls all of the above functions in order to interpret the code as best as possible. It then goes through each operation, interpreting it based on the type passed back from interpreting the operation name. The bit code is then written to a string, and concatenated together correctly. Once the entire coe file has been constructed, it is returned.

The next section contains functions which can be called by a series of C# unit tests built on the MS-Test unit testing framework. This requires the assembler to be built as a DLL (which is configured to be easy to do), and is then called numerous times to test the assembler as best as possible. The tests include testing translation of assembly operations, registers and constants to binary, translating a series of instructions, and translating an entire program.

The last section contains the main function, as well as a function which parses the input flags. The main function calls the input flag function, and then determines whether to display the help output, whether to assemble, and what to do with the result once the file has been assembled.

#### MiniC Compiler

The MiniC compiler implementation follows the specs for ANSI C, along with a few additions in order to help facilitate work for this particular hardware system. These additions included a primitive Boolean type, and an asm keyword which allows inline assembly with input and output values assigned to different registers.

The implementation itself consisted of 42 classes, 36 of which made up the Abstract Syntax Tree (AST), which is discussed below. The general flow is shown below. Due to the low-priority nature of this portion of the system, not every function will be described, but instead every step shown below. Comments exist for the code source (which should also be attached), which should clarify any particular questions.

Figure 7: Compiler Dataflow

The first two steps of the compiler, the Scanning and Parsing stages, were both generated using a tool called the Coco Parser-Generator, and thus will be discussed together. Coco uses an atg file, which describes the grammar that should be found, and what to do if a given piece of the language’s grammar is found, in order to generate six files which scan and parse the program automatically. The atg file was modeled after the C90 specification of ANSI C, and was optimized to avoid conflicts between multiple states matching a single input.

Out of this stage, an AST is created. An AST describes the structure of the program, taking into account things like order of operations. Each node in the tree represents a different part of the language, and has its own rules for creation, type-checking, optimization and flow graph generation, as will be described below. It also contains any fields which fully describe that part. As mentioned before, this structure took up the majority of the compiler’s code, and the next three steps all involve the AST heavily. Below is a heavily-simplified example of an AST for a program which calculates a Fibonacci number.

The next stage then goes through this AST, and checks that all types are correct, and annotates the AST with information that will be needed later (such as type size). After this point, the code is assumed correct by the compiler, and is passed on to the optimizer.

In the MiniC compiler, we have a few different optimizations. This first optimization step evaluates constant expressions and removes all unused functions, making the code much smaller and manageable. Once this is done, all nodes are converted to intermediate representation (IR) format, which resembles a high level assembly language. Optimizations then are made to take care of inefficiencies in this generated code. Once all this is done, the IR is taken and each IR operation is converted to one or more assembly operations. This creates the actual result assembly, which can be passed on to the assembler for more processing.

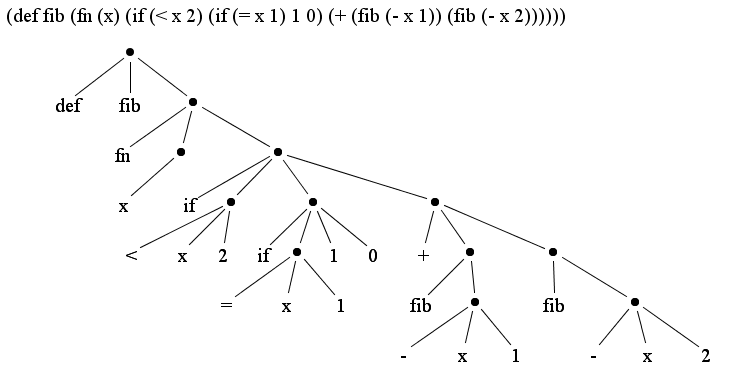


Figure 8: Sample Abstract Syntax Tree of Fib function (Courtesy of http://lifeofaprogrammergeek.blogspot.com/2009\_04\_01\_archive.html)

#### McOS Operating System

McOS was designed to be as simple as possible, in order to reduce its memory footprint, as well as to make it easy to implement. The OS can run one other program at the same time, placing it in memory half way through the memory stack.

McOS launches programs by loading the program from memory, adjusting the base register to point to the top of the program, and then to jump to the address 0, which will be adjusted according to the base register, starting the program correctly. Programs can end through software interrupts only.

The other program has its own stack, heap and instructions, but must use software interrupts to interact with the OS when requesting I/O or returning. An interrupt handler in the OS (using the custom “interrupt” keyword in MiniC) processes these, separating hardware interrupts from software interrupts. If it is a software interrupt, the code is processed, and the proper task is taken. If any arguments are needed, the register $a will have a pointer to where the arguments are. This pointer will need to be added to the base register of the program, in order to enter the program’s memory space. Return values will be pushed on top of the stack in the program, allowing it easy access without violating memory rules.

Hardware interrupts will be handled differently, processing them, and checking if any I/O requests are waiting for those interrupts (such as a call to getc). If this is the case, that function is allowed to continue. Otherwise, if there are no overall actions (such as reset or end program) that are assigned to that interrupt, the OS returns to the place the code was at before being interrupted.

## Discussion

**Problems Encountered**

Many different problems were encountered in the development of this system. Below is just a taste of the major problems we encountered:

#### Finding The Problem

Because this system had so many levels to it, sometimes half the battle was finding where the error would occur. Especially near the end, when everything was working together, a problem might look as if the processor was not branching correctly, when, in fact, the assembler would not be outputting the right address, or the compiler would not be outputting the right label to load. Once the SD card was loading, this problem was exacerbated by the fact that now the utilities we created to write the SD card image or the writing of the binary file itself would not work, which required the evaluation of the code in a hex editor, comparing it to the coe file output. These types of errors would appear to be an assembler issue, until either the coe file was run on the processor, or comparisons were made between the two.

Over time, these bugs became more recognizable as coming from different areas of the project, making it go faster. As things were added however, other parts would often crash due to being tested in new ways, which we had not thought of at that point.

#### Debugging the System

Due to the heavy use of peripherals, testing using a waveform was not practical for most purposes. Often, a seven-segment display of a single 32-bit value was all the information that could be gleaned at a time. Before the SD card was finished, each remake of the assembly for handling peripherals meant a 5 minute wait for it to compile in Xilinx. Once the SD card was done, this improved for assembly programs, but the actual hardware handling the peripherals still take 5 minutes or so just to compile and get on the board, making a very long and boring testing process.

The solution to this was to make smaller modules which could be compiled and loaded faster, and test those to make sure they work. Once these work, they were added into the processor and tested there, reducing the amount of time we needed to debug the system.

#### Testing the System

The final issue was testing the system. When one change was made, inevitably it broke something else that had been assumed fixed. For software, unit tests helped this, but they depended on the output being the exact same as the old output, which is often not the case, forcing by-hand comparisons. Hardware-wise, the only way to test was to run it and see what happened.

Throughout the development process, a program named Euclids, which determined the smallest relative prime number to the number indicated on switches on the board, was used as the defacto test. This tested loops, if statements, input and output, which was most of the needed features for a while. More programs are needed now, in order to test things like the VGA output, the SD card reader (which, incidentally, is mostly tested by the bootloader that loads every program), and the touch screen.

**Engineering Resources Used**

For this system, the two developers spent around 500 hours on the project, with an average of 10 hours per week per developer. About half of this time was testing, coming out to 250 hours of actual development, and 250 hours of testing.

## References

#### Tools Used

See tools required section above.

#### Contributors

|  |  |
| --- | --- |
| Name | Role |
| Kyle Green | Hardware Designer |
| David McGinnis | Software Developer |

#### Design Source Files

#### Because of the size of all of the source files needed, they have been included in the zip file which this report should have come with.

#### Reference Materials

See reference materials list above.

## Appendix 1: Descriptions of Verilog files

* **Main.v**: Main module that connects the processor to external modules and connects external modules to the outside world
  + **BitEpicness.v**: The top module for the processor. Contains the main datapath
    - **PCControl.v**: Determines if the program counter should branch or jump to a new instruction
    - **InstructionTranslate.v**: Splits the 16 bit instruction into an opcode, register numbers, immediates, and function codes
    - **Control.v**: Sets the control bits for each part stage of the pipeline for each instruction
    - **StallDetector.v**: Detects if a stall is necessary, and if so sends a signal to the PC control
    - **RegisterFile.v**: Contains sixteen 32-bit registers, with two outputs
    - **SignExtend7.v**: Sign extends the seven bit immediate to a 32 bit value
    - **ALU.v**: A simple 32 bit ALU which provides add, subtract, or, and, negate, shift, and comparisons to two values A and B
      * **Negator.v**: Negates the inputted value
      * **Shifter.v**: Shifts A left by B. If B is negative it shifts right.
      * **Comparator16b.v**: Compares A and B and outputs if A > B, A = B, or A < B
    - **Coprocessor.v**: Handles operating system type calls, such as reading and writing to external modules, generating software interrupts, and reading from/writing to interrupt related registers
    - **InterruptHandler.v**: Waits for hardware interrupts and stores them in a temporary buffer.
      * **Encoder32bit.v**: Converts 32 inputs into a 5 bit binary value
  + **MemoryManager.v**: Controls the memory, and outputs the instruction and 32 bits of data for each clock cycle
    - **InstructionMem.xco**: A block memory ROM containing the bootloader
  + **Decoder32bit.v**:
  + **SevenSegDisplay.v**: Stores a 32 bit value to be displayed on the seven segment display
    - **LEDDriver.v**: Converts a 16 bit value to a hex representation and displays it
  + **MillisecondTimer.v**: Counts the number of milliseconds that have elapsed since the last reset
  + **SwitchInput.v**: Reads the value from the switches, generates a user interrupt
    - **Debouncer.v**: Debounces the signal
    - **Pulser.v**: Generates a pulse from a debounced signal
  + **VGA.v**: Manages all the graphics signals
    - **VGAcontroller.v**: Generates the timing and screen coordinates to display to for the VGA interface
    - **GPU.v**: Writes simple graphics, such as fonts and filled blocks to the screen memory
      * **FontMem.xco**: Stores a simple font
    - **ScreenMem.xco**: Block memory RAM which stores the picture on the screen
  + **SDCard.v**: Stores addresses and starts the SD reader. Transfers data back to main memory when a block read is called
    - **SDCardReader.v**: Initializes the SD card and sends the block read command
      * **Execute Command.v**: Coordinates sending data, receiving the response, then reading the block of data (if necessary)
        + **sendSD.v**: Sends a command over the MOSI port
        + **receiveSD.v**: Reads the response from a command
        + **blockReceiveSD.v**: Reads a 512 byte long block and saves in the SDtempRAM
    - **SDtempRAM.xco**: Stores the values received by the SD card during transmission
  + **Keyboard.v**: Receives keyboard data from ReadPS2 and converts it to ASCII
    - **ScancodeDecoder.xco**: A scancode to ASCII lookup table
    - **ReadPS2.v**: Reads values from the keyboard
      * **OddParity.v**: Determines the parity bit for an 8 bit value
* **ucf.ucf**: Contains the configuration information