JavaScript ARM Emulator

William Koch
wfkoch at gmail dot com

April 27, 2012

Abstract

There has been a recent trend to migrate applications to the cloud. There many benefits in doing so such as centralizing data, allowing for the application to be always up to date, allowing for access across multiple devices and not requiring the end user to install additional software on the host machine. The majority of web applications contain JavaScript. Support for JavaScript is native to all browsers and allows for client side programming. ARM processors are used in a wide variety of embedded systems across many industries and provide a platform for mobile operating systems. I have developed a framework for an ARM based emulator written in JavaScript and a front end user interface written in HTML5. This architecture will allow the emulator to be run locally without an Internet connection or be deployed on a server as a web application.   Theoretically the JavaScript ARM emulator could allow for a web browser to run a mobile operating system as done in ARM based devices. This could provide a solution to online demonstration of running live ARM software. JavaScript is typically event driven, a client based processor will provide for a different type of programming style in which instructions are processed in a simulated parallel execution in a time dependent manor.

Table of Contents

List of Figures

List of Tables

Introduction

As devices become smaller, so does the hardware. This restrictions prevent devices such as smartphones to be able to handle computationally expensive tasks. For example, the Maps application on the Google Android operating system will request parts of the map from a server to be rendered rather than storing the entire world map on each phone. This has lead many developers to convert there applications into webapps and deploy them on a single server to be run from a browser.

This model provides many benefits to the developer and the end user. Rather than having to support multiple versions of software, a webapp will always be at its current version relieving the developer of maintaining support and bug fixes for multiple versions. Crucial software changes that need to be made, such as security patches, can go into affect immediately. The end user can be confident knowing there application is always up to date and does not have to worry about performing and/or checking for updates. Webapps provide the ability for the end user to be able to run software right from the browser rather than needing to install software on their machine. Furthermore this provides a cross device/platform experience as the only requirement is to have a browser with access to the Internet.[1]

It is far to easy to lose sentimental electronic data due to hard-drive crashes or other hardware malfunctions. Webapps provide for your data to be in a centralized location which allows access across multiple devices and provides peace of mind that data protection and backup is being performed on a regular basis. Some example webapps include Google Docs, a desktop word processor replacement, Dropbox, providing web storage and file staring, Turbo Tax for managing and filing taxes and Facebook which has created an entire platform and API for developers. Although there are large number of benefits to cloud migration as we have seen, there is still one underlying major requirement; you must have an Internet connection.

A JavaScript emulator provides the ability to be deployed on a server to run as a webapp and also allows for local execution. Web browsers come pre-installed in all modern operating systems provide a JavaScript interpreter, thus allowing for a JavaScript emulator to be run without an additional changes to the host machine. Many other programming languages need supporting software in order to run such as Java requiring a Java Virtual Machine (JVM), Perl requiring a Perl interpreter and PHP needing a PHP interpreter.

Primary motivation to develop an JavaScript based ARM emulator was based on Fabrice Bellard's JavaScript PC Emulator[2]. Bellard created a Linux operating system emulator completely in JavaScript. The home page of the webapp is a Linux terminal where you are able to navigate and execute commands just as you would in any Linux terminal. I wanted to take this emulation one step lower creating the ARM processor emulator in JavaScript which would theoretically allow for the assembly code of a Linux operating system to be installed. Not only could the processor install Linux but would be providing the underlying architecture to deploy any operating system written in the ARM instruction set architecture(ISA). This will open new doors to what is possible to be done client side on a browser. Additionally the JavaScript ARM emulator as a webapp will allow for collaboration between individuals spread across different locations in the world to discuss and test assembly code right from

1 This should be considered best practice because it is one of the primary advantages and reasons to use webapps however this is not always the case in the real world. Some companies such as Netflix do not support operating systems such as Linux [1] due to a Microsoft Silverlight plugin required to view video.

there browser. The emulator can also act as a teaching tool providing for a graphical user interface (GUI) of the processor internals that can not be seen.

Technical Approach

To create a JavaScript ARM emulator a processor first had to be selected to use as a guide as to how the emulator should be constructed. The ARM7TMDI processor was chosen because of its simple 3-stage pipeline design which can be seen in Figure 1 [3]. The ARM7TMDI was not replicated exactly due to information on control signaling being scarce. The approach was similar to Qualcomms Snapdragon Scorpion and Krait processors [4]. Qualcomm decided to design there own processor based on the ARM architecture. This allowed them to optimize their core while still allowing for the processor to be compatible with software implementing the ARM ISA. The ARM7TMDI implements the ARMv4T architecture. In order to simplify testing and logic in the decode stage, only the 16-bit Thumb ISA was decided to be implemented.

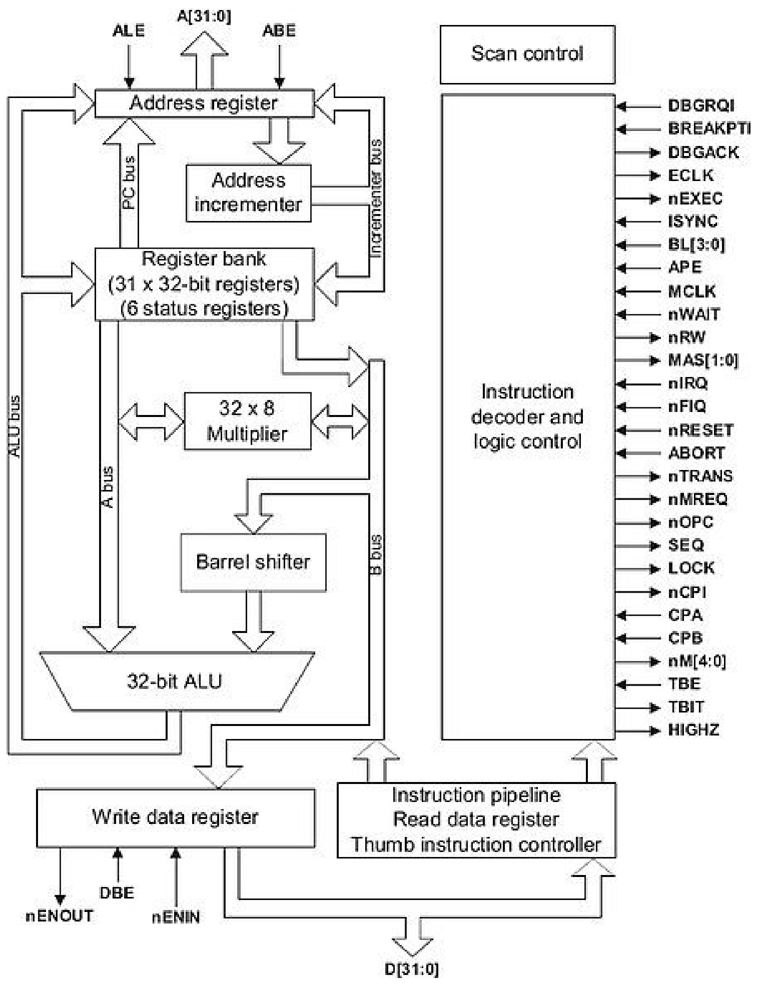


Figure 1: ARM7TMDI pipeline and processing elements.

Developing a processor emulator is large task with many subsystems which need to be integrated correctly for the system as a whole to properly function. The approach to how the subsystems are to be implemented and in which order will determine how easily the system can be tested and debugged. The major subsystems can be displayed in Figure 2. The GUI will provide the interface for the user to interact with the processor and memory. It supports a method to load the binary program into the memory. It also provides a display into memory so the user can view any changes that the code may have made in memory. Similarly the GUI displays the state of registers that may change during program execution. These systems have been designed to be as decoupled as possible so they can be updated or replaced by different implementations easily. The sandboxed subsystems also allows for testing between subsystems to be conducted.

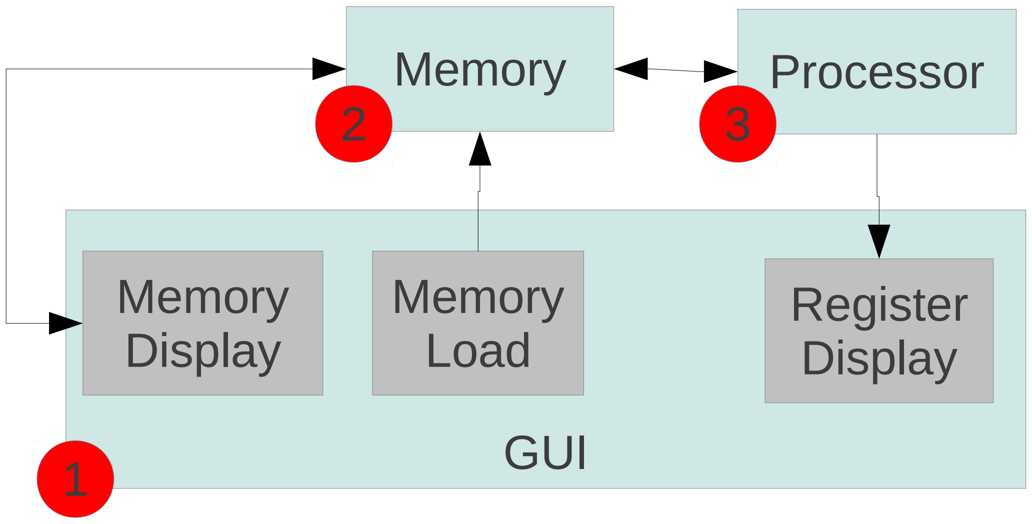


Figure 2: Primary subsystems in the emulator.

Since the GUI is the primary way to interact with memory and the processor it was selected to be implemented first. Once this interface was in place to provide the ability to load programs and display the activity in the processor and memory, the memory emulator was implemented. At this point testing between the GUI and Memory could be conducted to make sure they were operating correctly. The processor was implemented last and wired up to the rest of the subsystems. The red circles in Figure 2 denote the order in which subsystems were implemented.

The approach to designing the processor was to focus on the infrastructure rather than instruction implementation. The primary subsystems in the processor are clock generation and the 3-stage pipeline. It was known up front that these would be the most problematic due to there complexity, thus the most time was designated towards designing them. Once these subsystems are operational, expanding the ISA is just a matter of setting up signals in the decode stage.

Implementation

The implementation can be broken down into three components as outlined in Figure 2. These components are the GUI, memory and the ARM processor. The GUI is implemented mostly in JavaScript using the Document Object Model (DOM) to dynamically build the interface. The memory and processor must be initialized before the GUI so the size of the memory and the number of registers can be used to dynamically create the correct number of cells. Some pieces of the GUI are in raw HTML5 such as the layout and toolbar which we will go into more detail later. The memory and processor are implemented in raw JavaScript. JavaScript is a interpreted prototyping programming language. An object can be created with the *new* keyword. Additional properties such as variables, arrays and functions can be added to the object through the *prototype* keyword which creates a clone of the object and then adds the new definitions. This technique coupled with scope closures allow for classes to be constructed in JavaScript. Both the memory and processor implement this technique. No third-party libraries are used to limit the delivery of the emulator to two files. A HTML file with embedded JavaScript for the font end and a JavaScript file for the memory and processor implementation which can both be found in the Source Code section of this document.

ARM Processor

The ARM processor is implemented in raw JavaScript. The implementation is in arm.js which can be viewed in the Source Code section of this document. The ARM processor is represented as a single JavaScript class called ARM. Due to all of the properties and functions being prototyped from the ARM object, everything is accessible from a ARM instance variable. The memory must be initialized before the processor because the processor is dependent on a memory object and at the moment is the only configurable option in the ARM constructor.

Architecture Overview

The processor has been designed in a way in which a high level language has mimicked an electronic circuit. JavaScript must be used to act as a hardware description language (HDL). This was a major challenge to overcome. An input to a logic circuit will propagate through all elements until it reaches the end. These logic circuit elements may be sequential, or in parallel. In JavaScript programming there are no threads so parallel execution is not possible. A simulated parallel execution is creating in the processor using a clock which can signal a group of processing elements at once.

Each processing element in the processor is represented by a function while the buses connecting the processing elements together are represented by global variables. The interaction between processing elements and buses can be seen in Figure 3. These interactions are directly mimicked from the ARM7TDMI illustrated in Figure 1. Each processing element function acts as an electronic latch. When signaled by the clock the processing element will only execute its intended function if unlatched. For example if the ALU is unlatched it will have access to data on the A bus and the B bus and will be able to output onto the ALU bus. Typically this programming style is bad practice as it would be far more elegant to have the ALU function signature defined as *int alu(int value1, int value2)* rather than *void alu().* However we are trying to mimic an electronic circuit where multiple elements may be attached to a single bus, thus the decision was to have functions point to shared global variables to provide for a more realistic implementation.

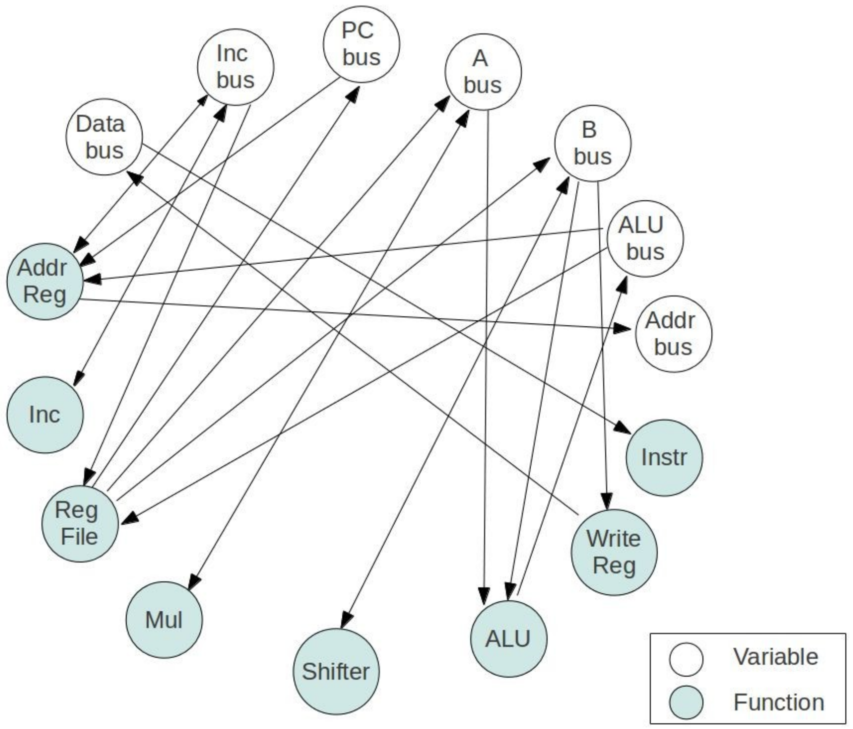


Figure 3: Graph describing relationships between processing elements and buses.

Clock Generation

A clock must be implemented to provide a way for state change. The clock is the heart of the processor. If the clock stops, the processor is halted, if the clock increases in speed, so will the number of processed instructions.

The clock is implemented by calling the JavaScript *setTimeout* function in an infinite loop. The *setTimeout* function will execute a callback function after a specified number of milliseconds. The delay is essentially the clock frequency which is configurable from a static variable. A minimum delay of 4ms defined in the HTML5 spec [5] restricts the maximum operating speed to 250Hz. There is a possibility to increase the operating frequency using the JavaScript *postMessage* function [6] [7] however this has not been full investigated and may be addressed in future work. The *setTimeout* callback function is where the clock broadcaster is implemented.

The processing elements are signaled by the clock using a technique in JavaScript known as broadcasting and subscribing. Processing element functions can subscribe to the clock broadcaster. Everytime *setTimeout* calls the clock broadcaster each function that is subscribed will be executed. The

ARM processor has two phases for every clock cycle [8]. These phases are internal but allow for more processing to be executed per clock cycle. The JavaScript ARM processor takes a similar approach by defining four clock states in which a function can subscribe. The clock states are rising, high, falling and low. Elements such as the address register and registry file subscribe to every state so if any other element makes a change to the attached bus it can be made sure to be updated. The other processing elements only subscribe to the rising and falling states. There was no need at the time to subscribe to all states. Although each function is being called at every clock cycle, as described in the previous section, the element will only process if unlatched. The Control Unit is responsible for latching and unlatching processing elements as well as setting control signals to each element. Figure 4 illustrates the processing elements that are subscribed to the clock.

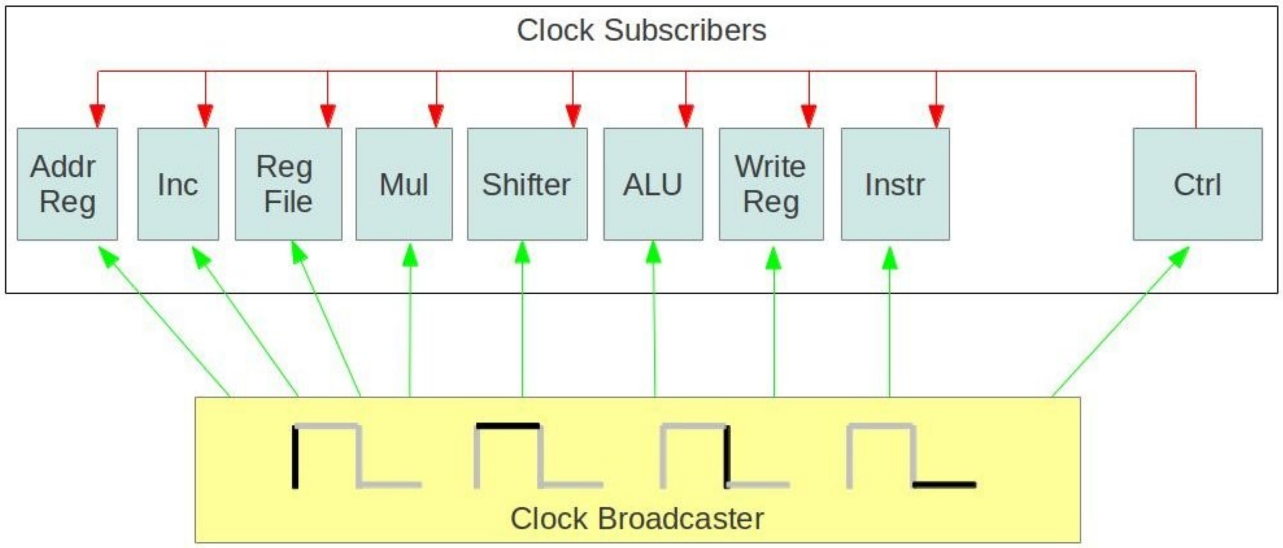


Figure 4: Clock model displaying broadcaster and subscribers.

Pipelining

The JavaScript ARM emulator implements a 3-stage pipeline. The stages are fetch, decode and execute. The fetch and decode stages are implemented in their own function. The Control Unit determines if and when the stages should execute based on whether there are any current hazards. The execution stage is not in its own function. It is just a point in time in which the processing elements are enabled by control signals to execute the current instruction.

Fetch Stage

The fetch stage is the simplest of the stages. It subscribes to the rising and falling clock events. During the falling edge the next instruction is read from memory, the address in the address register is incremented and then stored in the PC register[9].

Decode Stage

The decode stage decodes the binary instruction from memory and prepares all control signals depending on the type of instruction. The decode state is also subscribed to the rising and falling clock events.

During the rising edge of the clock, control signals cached from the previous cycle are added to the global signal queue. The control signals are cached for a cycle because the execution for the currently decoded instruction will not happen still the next cycle.

On the falling edge of the clock the instruction is decoded. There are 23 different possible encodings[10]. The encoding format is determined by using bit masks and comparison. When the instruction matches an encoding format the operands defined by the encoding format are extracted using a combination of bit masking and bit shifting. Next the control signals are prepared. As described earlier, the control signals are stored in a temporary cache so they can be added to the global signal queue during the next cycle. The following psuedo code adds a signal callback function to the queue.

```
1   this._pSignals.push(
2       function(e) {
3           if (e==”rising”){
4           //Set signals to be executed during rising clock edge
5           } else if (e==”falling”) {
6           //Set signals to be executed during falling clock edge
7           //If this is the last or only callback function remove myself from global queue
8           }
9       }
10  )
```

Line 1 pushes the callback function to the end of the temporary queue. Prior to the execution stage, this callback function is executed. A reference to the state of the clock will be passed in named variable *e*. Since this callback function will keep executing on every cycle until it is removed from the queue, it is able to be triggered for multiple clock states. Depending on the clock state the appropriate control signals can be set.

The data structure to store the control signals is a queue because some instructions use more than one execution cycle. For every execution cycle a callback function must be push onto the queue. STR instructions take 2 execution cycle thus after the decode stage, 2 callback functions will be cached to be added to the global signal queue on the next clock cycle. A LDR and some B instructions need 3 execution cycle. Likewise 3 callback functions will be cached for a cycle[11].

Execute Stage

Before the processing elements are triggered by the clock the first callback function in the queue is executed. This is done using the JavaScript *apply* function which allows for a function to be called and allow for parameters to be passed to the callback function. In our case the current clock state is passed to the function. Once the function has set all of the control signals the processing elements can be called by the clock. Referring to Figure 4, once the clock broadcasts a signal to the subscribed functions, they will each execute and if unlatched by a control signal shall perform the intended function. For example, the ALU is latched with the *this._signals.alu.fEnable* variable. This is boolean flag that indicates if the ALU is enabled or not. If the ALU is enabled by the callback function in the signaling queue it will access the global variables *this._wABus* and *this._wBBus* which will hold the data on the A and B buses. The variable *this._signals.alu.bFn* is a byte value indicating the function of the ALU to perform. The value of this flag is based on the ARM2 ALU functional codes[8]. Once the ALU has processed its function it is added onto the ALU bus by setting the result to the global variable *this._wALUBus.*

Hazards

In order to avoid pipeline hazards the processor takes a simple approach by stalling the pipeline for instructions that take use more than one execution cycle. Stalling prevents collisions and irregularities to occur in the datapath. With out stalling, if control signals are set while the previous instruction is being executed and still using the datapath unexpected behavior will occur. The JavaScript processor provides a very simple stalling mechanism. Before fetch and decode functions are executed the size of the signaling queue is read to get an idea of how many cycles the datapath will be used in the execution phase from the previously decoded instruction. If the queue size is greater than one then it is known that the instruction executing is going to take more than one cycle to complete. A display of the state of the signally queue per clock cycle can be seen in Figure 5.

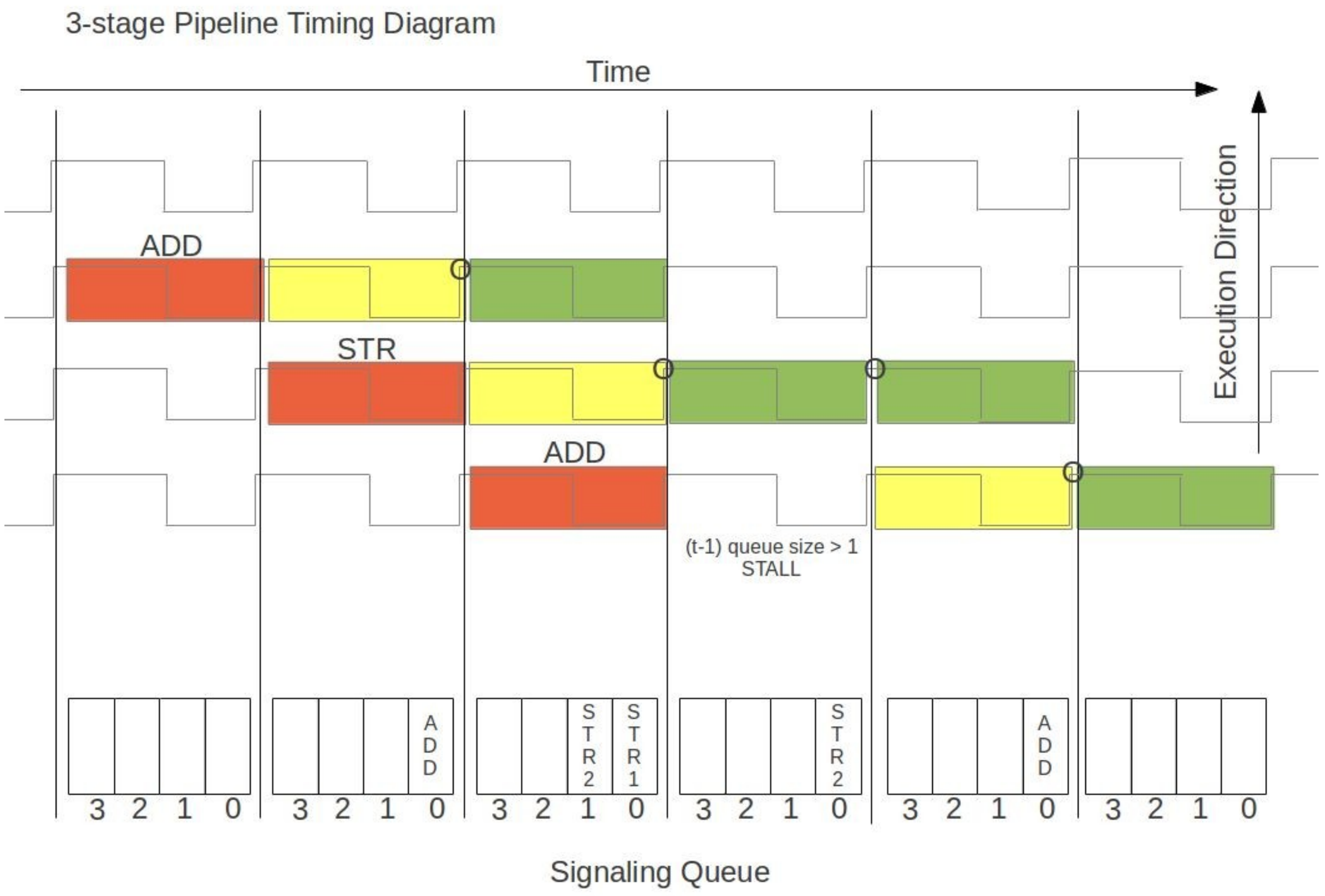


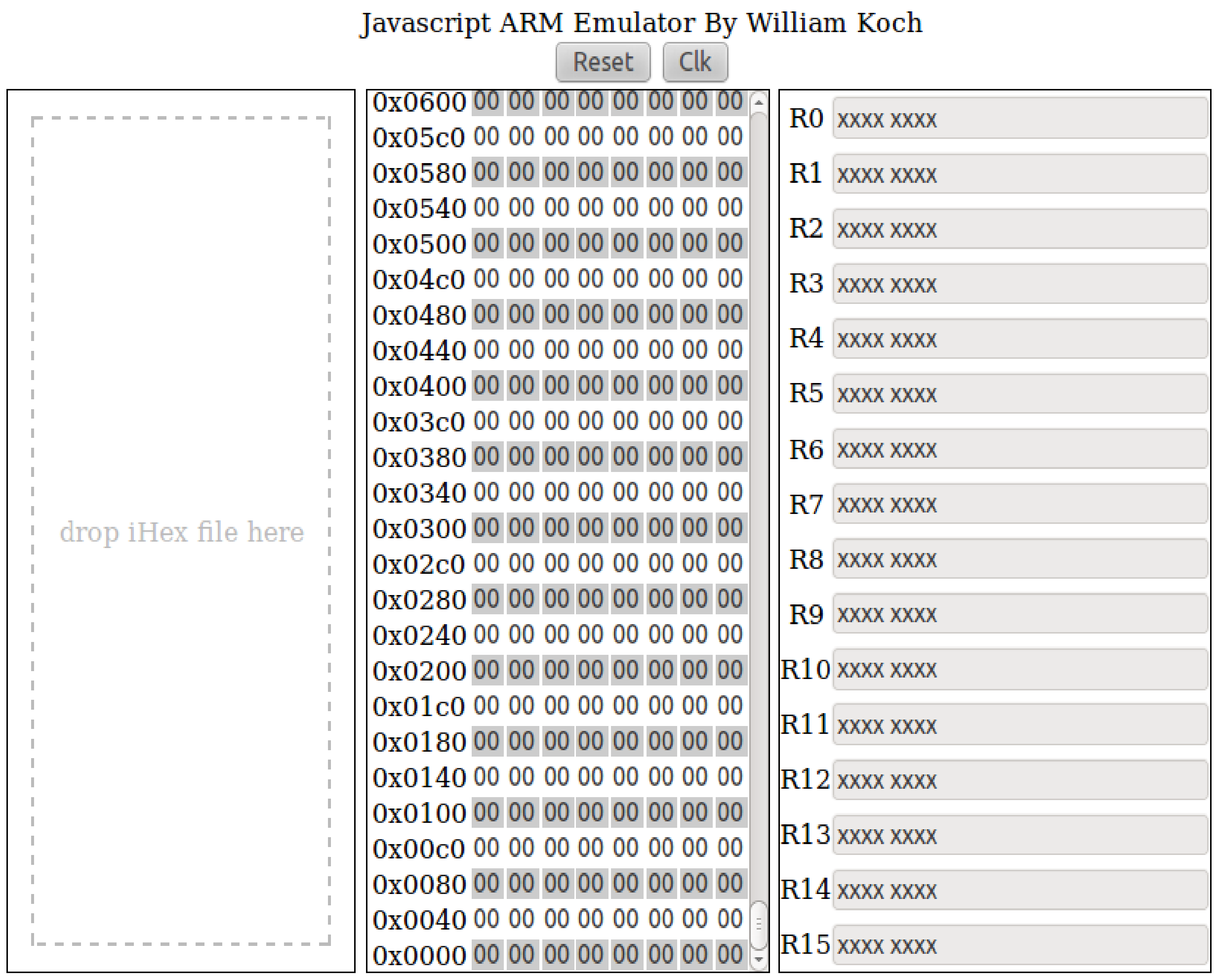
Figure 5: Example of pipeline signaling queue

Memory

The memory is defined in its own class called Mem. The Mem class resides in arm.js with the ARM class. An instance of Mem must be initialized before the processor in order to be passed into the processors constructor. Unlike the ARM processor, the implementation of the memory is quite high level and the electrical circuitry is abstracted. The primary focus of this project was the ARM processor so effort to create a realistic hardware emulation was designated for the processor rather than memory. The currently implemented memory is tightly coupled with the front end. When an instruction or data is fetched from memory, the data is obtained directly from the HTML input box representing the memory cell in the DOM. An address is passed to the *load* function in the memory class. The input box ID in the DOM is calculated using the the address and then used to access the value set in the input box. Likewise the *store* function calculates the input box ID in the same way and then sets the input box value with a second parameter of the *store* function.

User Interface

The emulator interface is constructed with HTML5, CSS and JavaScript and is implemented in arm.html which can be found in the Sources Code section of this paper. A screenshot of the emulator running in a Firebox browser can be seen in Figure 6. The layout and styling is achieved fully in CSS, the static content including the button bar and drag and drop area is in HTML and the memory and registers and built in JavaScript. The memory and register displays are dynamic and dependent on the configured size in the Mem class and the number of registered defined in the ARM class.



```
0x0600 00 00 00 00 00 00 00 00
0x05c0 00 00 00 00 00 00 00 00
0x0580 00 00 00 00 00 00 00 00
0x0540 00 00 00 00 00 00 00 00
0x0500 00 00 00 00 00 00 00 00
0x04c0 00 00 00 00 00 00 00 00
0x0480 00 00 00 00 00 00 00 00
0x0440 00 00 00 00 00 00 00 00
0x0400 00 00 00 00 00 00 00 00
0x03c0 00 00 00 00 00 00 00 00
0x0380 00 00 00 00 00 00 00 00
0x0340 00 00 00 00 00 00 00 00
0x0300 00 00 00 00 00 00 00 00
0x02c0 00 00 00 00 00 00 00 00
0x0280 00 00 00 00 00 00 00 00
0x0240 00 00 00 00 00 00 00 00
0x0200 00 00 00 00 00 00 00 00
0x01c0 00 00 00 00 00 00 00 00
0x0180 00 00 00 00 00 00 00 00
0x0140 00 00 00 00 00 00 00 00
0x0100 00 00 00 00 00 00 00 00
0x00c0 00 00 00 00 00 00 00 00
0x0080 00 00 00 00 00 00 00 00
0x0040 00 00 00 00 00 00 00 00
0x0000 00 00 00 00 00 00 00 00
```

```
R0  xxxx xxxx
R1  xxxx xxxx
R2  xxxx xxxx
R3  xxxx xxxx
R4  xxxx xxxx
R5  xxxx xxxx
R6  xxxx xxxx
R7  xxxx xxxx
R8  xxxx xxxx
R9  xxxx xxxx
R10 xxxx xxxx
R11 xxxx xxxx
R12 xxxx xxxx
R13 xxxx xxxx
R14 xxxx xxxx
R15 xxxx xxxx
```

Figure 6: ARM emulator graphical user interface (GUI)

iHex Import

Referring to Figure 6, the first column provides the user the ability to directly import an Intel iHex file. When the user drags and drops a file in this region, the dragover and drop events are fired calling the subscribed callback function. The callback function uses a new HTML5 feature called FileReader which allows for reading a file client side. Following the specs of the Intel Hex format [12] a parser was created in order to properly store the encoded instructions into memory.

Memory

Referring to Figure 6, the center column is a visual display of the attached memory. The memory display is created dynamically using JavaScript because of the configurable size. The line address is to the left and the memory cells to the right. Each memory cell is a modifiable HTML text input element with its value holding a byte of data represented in hex format. The decision to make the memory cells editable was to provide the flexible to enter the hex of an instruction manually for testing without having to load it through the iHex Import. Each memory cell input box has an unique ID which can be accessed during load and stores. The ID format is *mem-<address_base_16>* where *<address_base_16>* is the address location of the memory cell in hex. Due to the memory being tightly coupled with the memory display the contents displayed are always up to date since the memory is loading and storing its values straight into the HTML input elements. This was chosen as an optimized way to display and store memory instead of storing values in an array and displaying in a table which would result in redundant data.

Registers

Referring to Figure 6, the processor registers are displayed in the right column. Similar to the memory display the register file display is created dynamically using JavaScript due to the configurable number of registers. The register number is displayed to the left while the register value is displayed to the right. Each register value is set in a HTML readonly text input element. The decision was made to make the registers readonly to maintain the integrity of values in the register. There were instances during testing where it was convenient to set register values but there is a work around to inject new values into registers which is discussion in the Testing section of this document.

The register display is completely decoupled from the processor. The register values are updated using a callback function subscribed to the ARM processor clock. Implementing a custom decoupled user interface is a perfect example of the flexibility the clock broadcasting model. On every clock cycle, the register display callback function is called. The scope of the callback function is set to the ARM processor instance so the callback function has full access to all properties including the values in the registers. The callback function updates the register file display with each new register value.

Button Bar

Referring to Figure 6, the button bar consists of a Reset and Clk button. The Reset button sets the processor to re-initialize. The Clk button was added to help during testing and debugging. The Clk button allows the user to manual trigger the clock. The state of the processor is frozen until the user fires the next clock cycle. This provides the user the ability to inspect the state of the pipeline, processing elements and buses for detailed debugging. Once implementation of the processor began it became obvious that it would be quite complex and time consuming troubleshooting issues without this feature.

Testing

The development for the ARM JavaScript emulator was done on a 64-bit Ubuntu 11.10 operating system. The software used to develop the emulator is listed in Table 1.

Software	**Version**	**Description**
Firefox	11.0	Internet browser emulator was test on.
Firebug	1.9.1	Firefox plugin used for DOM inspection and JavaScript debugging of the emulator.
GNU ARM Toolchain	4.0.2	Group of tools for compiling ARM assembly code.
Eclipse Indigo	3.7.1	Integrated development environment (IDE)
Web Tools Platform (WTP)	3.3.2	Eclipse plugin providing features for developing web applications.

Table 1: Software used in development

Testing of the emulator was achieved by writing the assembly code for the instruction to be tested in a file called *testing.s.* The GNU ARM Toolchain was then used to produce an iHex file to be imported into the emulator. The programs were executed in the following order:

1. ./arm-elf-as -mcpu=arm7tdmi -mthumb -o testing.o testing.s
2. ./arm-elf-ld -o testing.elf testing.o
3. ./arm-elf-objcopy -O ihex testing.elf testing.hex

Line 1 executes the arm-elf-as program which assembles the assembly code into an object file. The target platform is defined using the *-mcpu* flag and to indicate that the source should be assembled into the Thumb ISA the *-mthumb* flag is set [13]. Line 2 executes the arm-elf-ld program. The assembled object file is linked to produce the elf file. Line 3 executes the arm-elf-objcopy program which converts the elf file into an iHex formated hex file.

After the iHex file is dropped into the emulator and the memory is populated with the instructions, and testing can begin.

The Firebug plugin is enabled for JavaScript debugging. Breakpoints are set in the code at locations of interest. The manual clock was then used so the state of the processor could be inspected at each clock cycle and to confirm the behavior of the instruction functioned correctly. Additionally Firebug allows for modification of the DOM. Properties in the ARM emulator could be dynamically changed at run time such as values in the register.

Results

Once implementation of the pipeline began, a better understanding of the complexity was established. The difficulty of programming the main processing elements was underestimated and it was clear from the beginning it was unrealistic to implement the full Thumb ISA. As previously mentioned, the primarily focus was to produce functional processing elements and a stable architecture. Once in place, instructions one by one would be implemented in the decode stage. Early on it was a goal to be able to at least get an instruction from every main category (arithmetic, memory and repetition) tested and implemented to show the processor was functional. The ADD immediate instruction was tackled first since it could be executed in three clock cycles and had the simplest datapath. This implementation was straight forward and was completed without much trouble. Next the STR instruction was approached to provide a way to store into memory. This is when the complexity started to increase due to the STR instruction needing two execution cycles[11]. While implementing the ADD immediate instruction I did not take into account multiple execution cycles. Solving this problem ended up being the biggest challenge because of the way the signals for every execution cycle needed to all be determined in the decode stage and executed before every execution cycle. This essentially became a timing issue. The problem was eventually solved using the signaling queue as discussed previously. With this problem solved the changes needed for the LDR instruction was minimal. Next the B instruction was integrated to provide repetition and looping which like the LDR instructions needs 3 execution cycles[11] however this instruction has not been fully tested.

Discussion

Lessons Learned

This project has been a tremendous learning experience. I have always wanted to write an emulator so this provided the perfect opportunity. It has taught me a different way to approach software problems using a high level language, specifically developing a programming technique to mimic an electrical circuit. Typically JavaScript is event driven. In a web page, HTML elements will subscribe to JavaScript callback functions sequentially for processing. The approach of having to use JavaScript as a continually executing program and mimic ways of parallelization also showed me a different way JavaScript could be used. The only way this was achievable was through the processing elements being powered by the clock to produce concurrent like processing.

Additional this project gave me the opportunity to learn the HTML5 FileReader. Without the ability to conveniently have a large program automatically load into memory this emulator would remain utterly useless outside of demonstrating the execution of a few instructions. Although loading a large program could have been achieved by pasting iHex code into an HTML text area element, convenience is everything.

Future

This is just the beginning for the emulator. There is a significant amount of work needed to be done to get to the point of of running a operating system. Further testing across multiple browsers is also needed to observed the differences in how the browser renders the front end. Benchmarking of the execution speed and max operating speed also needed to be conducted. The future work can be divided into three categories. Improvements made to the processor, to the front end and finally what sort of creative projects can be based off of this emulator.

Processor Improvements

The first thing that needs to be finished up is the implementation of the Thumb ISA. Once this is complete a version can be released of the emulator. Next support for exception handling needs to integrated and a look at how to increase the running speed of the processor clock. These improvements can produce another minor release of the software.

Implementation of the full ISA will be a very large task. Support for Thumb decompression and context switching are a couple of the additions that will need to be made. These changes will essentially complete the processor thus releasing a major version of the emulator.

Apart from additional features, as the emulator evolved and started taking shape I found that it would be organized better to represent each processing element as its own separate class so it could have its own unique control signals prototyped off of it. As discussed earlier, processing elements are represented as functions however their control signals are also global variables. As the project expands it would be more manageable to move these global variable control signals into their own class.

Visual Improvements

Programmers may wish to customize their own interface for the processor and due to the decoupled nature of the architecture this is easy to do. However, to improve upon the current font end it would be beneficial to add a visual clock and debug console.

The visual clock would provide the user an indication as to whether the clock was rising, high, falling or low. This would show which stages or processing elements may be executing at that moment.

In certain situations errors may arise and the user should be notified. An example may be a file trying to be loaded into memory that is not an iHex file. A console output will be added to notify the user of particular events. Additionally this would provide a perfect place for log messages to be displayed for events happing in the processor during run time.

Possibilities

It is very exciting to brainstorm project possibilities to be used with this architecture. An entire community could be based on creating interfaces and plugins for the emulator to achieve specific goals.

JavaScript has built in functions for making Asynchronous JavaScript and XML (AJAX) calls to servers. AJAX allows data to be sent and received from remote servers. An AJAX memory interface could be created to allow for the processor memory to be stored on a remote server rather than embedded in the client. If this emulator were to actually be used to processes client side tasks, this is probably the approach that would be taken. The program stored on the remote server would allow it to be more decoupled from the processor allowing updates to be made with out other systems being affected. Since an AJAX request takes time to retrieve or store data it would demonstrate a similar memory hierarchy as seen in real processors. The information on the remote server could be thought of as the main memory.

The continuous frequent requests to the server for information may create a bottle neck in which the server may not be able to attend to all clients making this improbable to actually be used in production environments. However it may be possible by integrating cache controllers and having a large client side caches so less frequent requests need to be made to the server.

The HTML5 Canvas allows for modification of a bitmap. This provides a way for dynamic drawing. A Canvas Interface Controller could theoretically be created to mimic a VGA controller allowing for the processor to interact with a display. This could open up an entire realm of video game emulators which could be run client side on a browser with out any additional dependencies.

References

1: Microsoft, Get Microsoft Silverlight, , http://www.microsoft.com/getsilverlight/Get-Started/Install/Default.aspx
2: Fabrice Bellard, Javascript PC Emulator, 2011, http://bellard.org/jslinux/
3: Erich Silvestre e Pedro Bachiega, Diagrama de Blocos da Arquitetura ARM7, 2009, http://en.wikipedia.org/wiki/File:Diagrama_ARM7.jpg
4: Ryan Whitwam, How Qualcomm's Snapdragon ARM chips are unique, 2011, http://www.extremetech.com/mobile/94064-how-qualcomms-snapdragon-arm-chips-are-unique
5: Mozilla, window.setTimeout, , https://developer.mozilla.org/en/DOM/window.setTimeout
6: Mozilla, window.postMessage, , https://developer.mozilla.org/en/DOM/window.postMessage
7: David Baron, setTimeout with a shorter delay, 2009, http://dbaron.org/log/20100309-faster-timeouts
8: Steve Furber, ARM system-on-chip architechure, 2000, Addison-Wesley Professional
9: Leonid Ryzhyk, The ARM Architechure, 2006, University of New South Wales
10: ARM, ARM Architechure Reference Manual, 2000
11: ARM, Performance of the ARM9TDMI and ARM9E-S cores compared to the ARM7TDMI core, 2000
12: SB-Projects, Intel HEX format, 2011, http://www.sbprojects.com/knowledge/fileformats/intelhex.php
13: Dean Elsner, Jay Fenlason & friends, Using as, 2002