YAMS: Awesome MIPS Server

Stephen Brennan, Katherine Cass, Jeffrey Copeland, Andrew Mason, Thomas Murphy, Aaron Neyer

***Abstract*—We set out to build a simple web server in MIPS. Mission Accomplished.**

***Index Terms*—MIPS, computer architecture, HTTP, server, MARS, ISA simulation.**

I. Problem Statement

THE goal of this project was to write a static HTTP server in MIPS assembly running in the MARS simulator. The server would be able to serve a website, which is contained in the `html/` directory on port 19001. The content would then be viewable in a browser. Sockets would be used for networking and implemented in extending MARS syscalls, making the high-level networking features accessible from simulated MIPS assembly.

II. Major Challenges

The first major challenge in YAMS was implementing socket syscalls. MARS was chosen early on because it allowed for custom syscalls; however, we only found sparse documentation. This required us to learn about MARS through trial and error. Additionally, if one stops MARS while waiting on a socket syscall, the simulator enters an error state, forcing us to either 1) double-reassemble-run the program, or 2) restart all of MARS. These additional steps made debugging much slower than simulating a program without the unexpected behavior of the Java/MARS sockets.

The larger challenge in implementing YAMS was parsing HTTP requests. RFC 2616[1], our HTTP (Hypertext Transfer Protocol) reference for this project, is 175 pages long. While it is possible to implement a full HTTP stack in assembly, it is difficult to do in one or two months. Thus, we focused on what was required to get web pages to display in a browser: GET, POST, Expect, and only identity encoding or Content-Length. Additionally, because the request parser had to interact with the socket differently depending on the request header (e.g. Transfer-Encoding, Content-Length, and Expect), our testing and debugging revolved around trial-and-error requests with the Unix program `curl` and browsers.

III. System Components

After submission of the report, the entirety of the implementation is available at [2]. The breakdown of the system components is as follows.

All authors are with the Department of Electrical Engineering and Computer Science, Case Western Reserve University, Cleveland, OH, 44106 USA

Final Report submitted May 1, 2015, typographical revisions and clarifications May 28, 2015.

A. String Operations

In order to parse HTTP requests, we needed to implement much of the C standard string library. In `mips/string.asm`, we implemented `strlen`, `strncpy`, `memcpy`, `strcmp`, `strncmp`, `atoi`, `htoi`, `strcat`, `strncat`, as well as two functions for identifying the index of characters and substrings, `str_index_of` and `substr_index_of` respectively. To verify their correctness, we wrote tests in `mips/test_string.asm`. They can be run in MARS by loading that file as the main file, instead of `mips/main.asm`.

B. HTTP Request Handling

As mentioned in Major Challenges, our focus with request parsing was on serving a page that a browser could render. Therefore, we chose to support a subset of HTTP request statuses (e.g. GET, POST), look for a few headers (e.g. Content-Length, Expect), and read or write through the socket accordingly. The logic for request parsing is in `http-requests.asm`, and is reached through the `get_request` method from `main`, returning an internal HTTP request type, and, as applicable, request URI, request body, request body length, and content type. The main method takes this data and hands it off to the file loader and response builder.

The request parser functionality was validated through trial-and-error: seeing if debug prints contained the right information and if the browser loaded the page properly. The mixture of MIPS assembly and MARS syscall extensions precluded the use of unit tests to verify the correctness of the parser's behavior in the given time.

C. File Access

File access is already implemented in the MARS default syscalls, saving us the trouble of adding this feature to the simulator. These calls were wrapped in function-like macros for ease of use.

Additionally, the HTTP handling receives URIs (uniform resource identifiers), not system paths. These strings needed processing to produce a valid file path that could be consumed by the MARS syscall. Our processing, implemented in `mips/file.asm`, is composed of substring-blacklisting to prevent the `../` directory from being used to access arbitrary system resources and string operations to modify the URI into a useful file path. The file path is created using the default `html/` directory (where our web resources are stored) as the base relative file path and URIs with a trailing `/` have the file name `index.html` appended. The processing operation has tests implemented in `mips/test_file.asm`.

D. Turing Tape Language Interpreter

As an additional stretch feature, we implemented an interpreter for the esoteric programming language Brainfuck[3]. This simple language simulates a Turing machine's operations and has only 8 commands (each of which is a single character). The interpreter is implemented in `mips/brainfuck.asm`, and tested in `mips/test_brainfuck.asm`. When the server is running, code can be loaded into the interpreter by sending a POST request to the URL `/load`. The code can then be run by sending input in a POST request to the URL `/run`, which will return the program output in its response. For a more accessible interface, the URL `/brainfuck.html` provides a simple page that uses JavaScript to transfer the code and input to the interpreter and receive the resultant output.

IV. Component Integration

In order to integrate these different components into a cohesive project, we had to define and follow a strict calling convention. We decided that all functions would be called using the `JAL` instruction. Only the `$s0-$s7` saved registers and global pointers would be preserved across function calls. Arguments were passed (and not necessarily preserved) in the `$a0-$a3` registers and return values were placed in the `$v0-$v1` registers. Any additional arguments or return values would be placed on the stack. It was the caller's responsibility to push the return address of its scope and any other registers which it wanted preserved to the stack prior to calling the next function. To facilitate conforming to these rules, we defined macros in `mips/util-macros.asm` for pushing to and popping from the stack to make those operations semantically clear in our code.

V. User Interface

We have implemented an HTTP server, so the functional interface is through a web socket while the debug interface is the standard output of the MARS simulator. To compile MARS with our implemented syscalls, run `make` from the project root. Then run `java -jar Mars4_5-SockMod.jar` or directly open the JAR to use the modified version of MARS. Load the `mips/main.asm` file into the simulator. Assemble and run that to start a server listening on port 19001 (`http://localhost:19001/`). Now, the content in the `html/` directory is hosted at the root of the server. The content we created for this project is rendered below as our interface examples. Alternatively, the server can be accessed using the Unix tool `curl`.

VI. Documentation of Runs

A. Static Page Content

The static page in Fig. 1 is hosted as the `/index.html` document of the server. This is the document the user would expect to load into their browser upon access to the root page of the server. This demonstrates the ability to serve multiple resources for a page: images, fonts, and the HTML.

B. Dynamic JavaScript Content

The dynamic page in Fig. 2 is the presentation given to the class on April 23, 2015. This document uses HTML, JavaScript, and images. All resources are loaded from YAMS and not from external internet hosts.

C. Interactive AJAX Content

The final dynamic-content page, shown in Fig. 3, is the front end interface for the YAMS server's Brainfuck[3] interpreter. The page interacts with the interface through POST requests and updates the text field on the page when it receives the result of running the user-supplied Brainfuck code and input.

VII. Group Member Contributions

- Stephen Brennan
  1) String functions
  2) Brainfuck interpreter
- Katherine Cass
  1) Static site
  2) Report content
- Jeffrey Copeland
  1) Socket syscalls
  2) HTTP request handling
- Andrew Mason
  1) HTTP response building
  2) Project presentation
- Thomas Murphy
  1) File access
  2) Code formatting and style
  3) Report organization and typesetting

Aaron Neyer planned to build a website documenting our project, but could not complete it due to personal circumstances.

VIII. Conclusion

We started this project to build a simple web server in MIPS. Despite the high upfront workload, our collective components merged into what has been called "a surprsingly robust web server." Thus, we have achieved our goal: writing a static webserver in MIPS and MARS.

APPENDIX A
SCREENSHOTS OF CODE EXECUTION RESULTS

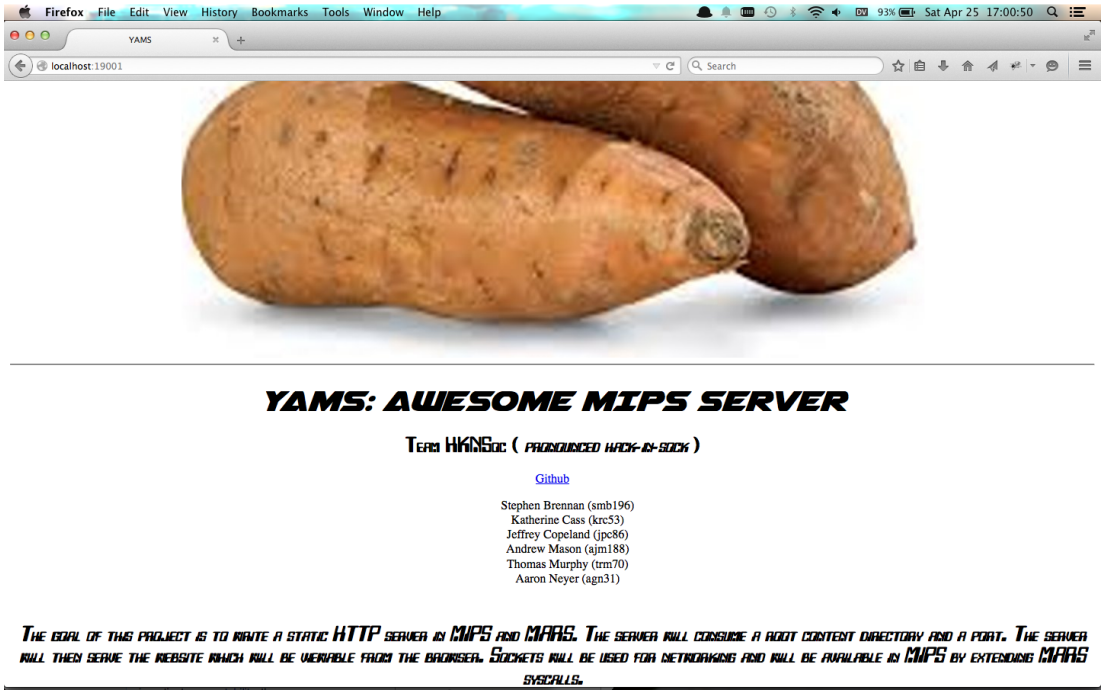

Fig. 1. The static `index.html` served by YAMS by default.

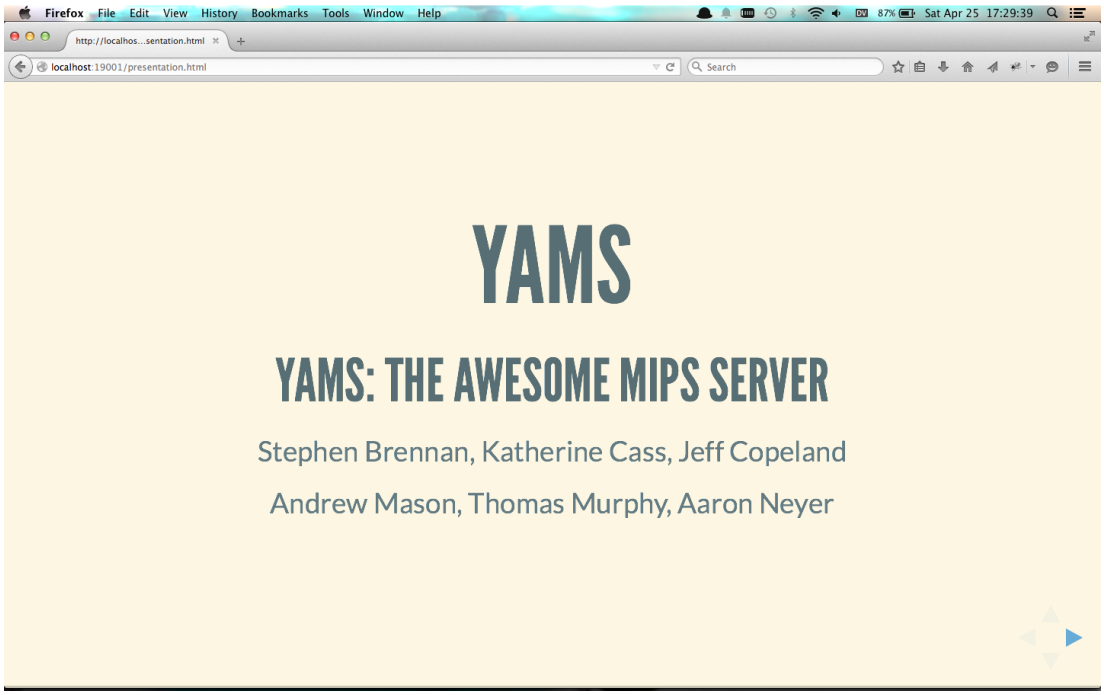

Fig. 2. The first view of a dynamic HTML/JavaScript page containing the material presented to the class.

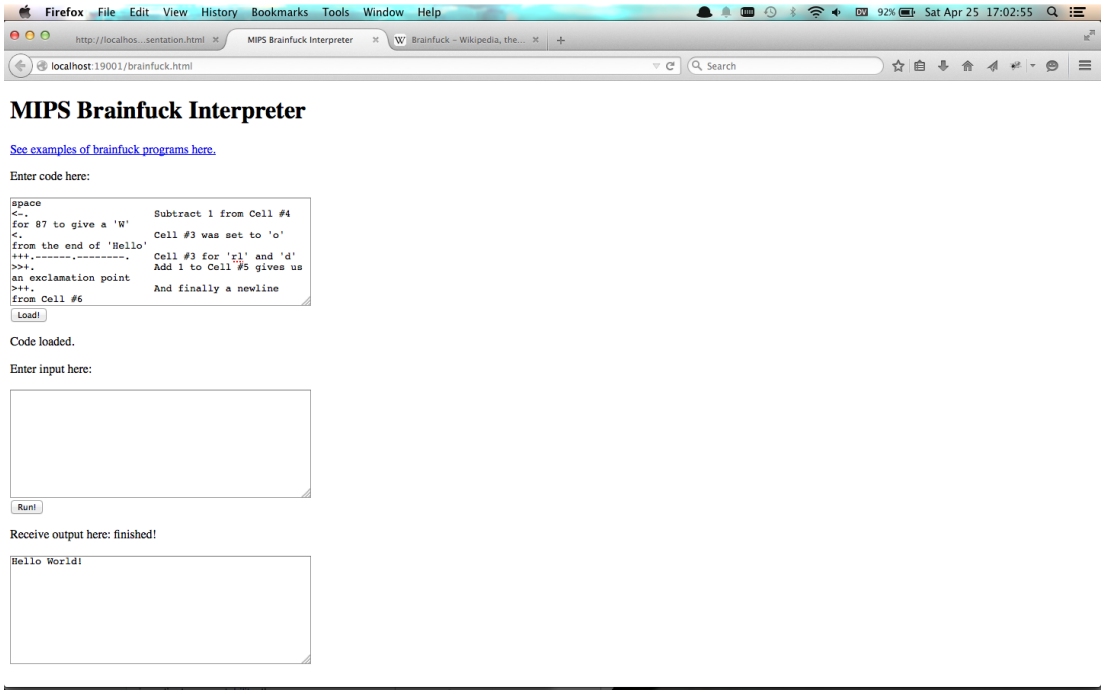

Fig. 3. A web interface to an implementation of the Brainfuck interpreter contained within YAMS.

ACKNOWLEDGMENT

The authors would like to thank Cameron Gutman (`https://github.com/cgutman`) for providing information about the usage of identity transfer encoding.

REFERENCES

[1] P. J. Leach, T. Berners-Lee, J. C. Mogul, L. Masinter, R. T. Fielding, and J. Gettys. Hypertext transfer protocol – HTTP/1.1. [Online]. Available: https://tools.ietf.org/html/rfc2616

[2] S. Brennan, K. Cass, J. Copeland, A. Mason, T. Murphy, and A. Neyer. yams - YAMS: Awesome MIPS server. [Online]. Available: https://github.com/brenns10/yams

[3] U. Müller, "Brainfuck: An eight-instruction turing-complete programming language." [Online]. Available: http://www.muppetlabs.com/~breadbox/bf/

[4] "List of HTTP status codes," page Version ID: 659966084. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_HTTP_status_codes&oldid=659966084

[5] R. Fielding and J. Reschke. Hypertext transfer protocol (HTTP/1.1): Semantics and content. [Online]. Available: https://tools.ietf.org/html/rfc7231