**Draft Outline**

1. Intro
2. Instruction Objectives + Opcode Objectives
3. Assembler Objectives
4. Radix-Four Booth’s Algorithm (Theory)
5. Radix-Four Booth’s Algorithm (Example)
6. Instruction Set Design
7. Pre-existing Instructions
8. More Instruction Set Design
9. Datapath design
10. More datapath design

Processor Design:

A Report on the Design and Development of a Software Assembler for a Custom Instruction Set Architecture

By Matt Fennell and Ryan Rabello

Submitted in Partial Fulfilment of the Requirements of CPTR 380

3/15/18

**Intro**

As part of the coursework for CPTR 380 – Computer Architecture, we were tasked with developing a project that would give us further experience with instruction set architectures, machine code, processor design, and processor operation. After some consideration, we decided to choose a project focusing on the assembly of processor instructions into processor machine code. In order to add an element of creativity, and to expand the scope of the project, we decided to also develop a custom instruction set that would focus on the radix-four version of Booth’s Multiplication algorithm. Using our new, simplified instruction set, along with its own machine code, we’d then develop a software-based assembler. Once the scope and direction of our project had been decided, we set about defining specific goals for each of the three areas of our project, which are as follows.

First, our goals regarding instruction set design. Our guiding principles in this area were **simplicity** and **clarity**. Our aim was to develop a clean, creative, and readable set of instructions for implementing the radix-four Booth’s algorithm. On top of standard forms of documentation, we planned include an explanation of the process that went into the design of our instruction set, which we figured would aid in comprehension and usability.

Next, our goals regarding the machine code design. Once again, simplicity was a guiding light, with our main goal in this area being the re-use of existing MIPS machine code. By saving time not reinventing the wheel, we’d be able to dedicate our resources to developing full data path designs for any of the new hardware components our processor might entail. From the preexisting instructions and our new data path designs, we’d then be able to have a simple 1-to-1 translation from our custom instruction set to machine code.

Finally, our goals for the assembler itself. These were a bit more complex, as we were able to come up with all sorts of cool additional features beyond simple instruction set assembly. These bonus goals include support for output processing logs, to help with debugging faulty instructions. Additionally, we planned to implement a simple GUI for interfacing with the assembler, and basic hazard detection that would trigger warnings for basic stall and control hazards.

With these goals in mind, we set up a Google Doc for documenting our work throughout the project, and began our research into the specifics of the radix-four version of Booth’s algorithm.

**Radix-Four Booth’s Algorithm**

While the standard, radix-two version of Booth’s Algorithm relies on individually shifting through each bit of the multiplier to determine the next action, the radix-four version works a little bit differently, in order to speed up execution time. By taking in bits of the multiplier three at a time, shifting twice, and allowing for an overlap of one bit in the next set of bits, we’re able to recode the multiplier to allow for one of five possible actions: do nothing, add the multiplicand, add the inverse of the multiplicand, add the multiplicand’s double, or add the inverse of the multiplicand’s double. This radix-four recoding splits processing time in half, and as long as we remember to preserve the sign bit, gives us the exact same result as the standard Booth’s algorithm.

During our research, we stumbled upon a few different methods for running through the radix-four version of the algorithm, some with shortcuts for even faster execution, but in the end, we chose a simple, reliable version to base our instruction set off of. Included below is a short example of the algorithm we’ve used, with documentation of each step taken along the way.