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.

Step 0.0:A: 1011(-5) | B: 0011(3)

// First, the multiplier and multiplicand are loaded into

// memory/registers

Step 0.1:-B: 1101 | 2B: 0110 | -2B: 1010

// Next, the inverse, double, and double inverse of the

// multiplicand are calculated and loaded into memory/registers

Step 1:0000 10110

// At this point, the algorithm actually gets going.

// First, the multiplicand is padded with a 0, and loaded into

// the lower half of the result register. We look at the bottom

// three digits, and identify the action we should take next

// before shifting, which here is to add the inverse

Step 2:1101 10110

// Now that the inverse has been added to the upper half of the // result register, we prepare to shift right twice, while

// preserving the sign bit

Step 3:1111 01101

// Once again, we look at the bottom three digits, and identify // the next step, which once again is to add the inverse to the // upper half of the result register

Step 4:1 1100 01101

// When we added in the last step, we g0t a carry bit, which

// we’ll drop before the sign-preserving double shift

Step 5:1111 00011

// At this point, we’ve cycled through every bit in the

// multiplier, and we’re pretty much done. All that’s left is to

// drop the pad bit.

Step 6:1111 0001

// Here’s our result, which should be -15. Using Two’s

// Complement, we can verify that this is indeed a -15.

Verify :0000 1111

// Yup, sure enough, that’s a 15, which means our result was -15

**Instructions**

After working through several problems using the radix-four Booth’s algorithm, we felt we had a solid understanding of the process. The next step was to break down the algorithm into its absolute simplest form, and then develop an instruction set from that simplification. Below is the full listing of that set, starting with the instructions that carry over from the standard MIPS instruction set.

* **Register Load/Store**

Straightforward register loading and storing, working from memory addresses or register contents.

* **Register Load Immediate**

Another instruction for working with registers, that reads in immediate values.

* **Register Store Upper/Lower**

A pair of instructions for dealing with our oversized result register, by splitting it into an upper and lower half, and providing an instruction for working with each half.

* **Syscall**

Syscall is a special instruction that prepares the simulator or processor for a variety of actions, such as taking input digits, outputting register contents, and telling the simulator/processor to quit.

* **Variable Shift (by 2 bits)**

A shift function that can be logical or arithmetic, that shifts the contents at a source address by n bits, before storing it into a destination address.

* **And**

Stores the Bitwise AND of two source addresses into a destination address.

* **Or**

Stores the Bitwise OR of two source addresses into a destination address.

* **Branch on equal/not equal**
* **Add**

Add two unsigned registers together (this is the instruction that gets used in booth’s algorithm.)

After listing out all the necessary instructions, we took a step back and considered pieces of hardware that would be able to take care of some of the large clumps of repetitive instructions by replacing them with simple custom instructions. Those custom instructions are as follows:

* Two’s Complement

Store the two’s complement of address1 in address2

A function for inverting then adding one to the bits. Bitwise invert, add 1 unsigned

* Booth-Load

Loads the value of A and B into predetermined registers

-B (two’s complement B)

2B (shift left logical B)

-2B (shift left logical –B)

* Booth-Add

This function looks at the last three bits of A and performs the appropriate operation according to the function. Use last three bits of A as select for mux to 0, B, 2B, -2B, and –B. Add selected register to upper of result

**Development Process**

After coming up with basic instruction set definitions, we spent some time going over old homework that focused on instruction sets, specifically targeting our Radix-Two Booth’s Algorithm homework. To confirm that we’d done our due diligence in designing a full-featured instruction set, we redesigned the Radix-Two program with our instruction set, and then adapted that program to work with the Radix-Four version of the algorithm. Once this was taken care of, we moved on to the biggest part of our assembler, the mappings between instruction sets and machine code.

To get a handle on how these mappings work, we spent a good deal of time going over MIPS instruction set documentation. Our justification for this was that with the exception of our custom Booth hardware, our hypothetical processor would be similar to the MIPS processor. During our time focusing on MIPS’ machine code, we paid special attention to op-codes, function codes, and the parameters for each instruction.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Instruction** | **Instruction Type** | **Function Code (If R type)** | **Opcode** | **Description** | **Example** |
| 2's Comp | R | 0x01 | 0x00 | A function for inverting then adding one to the bits. (Bitwise invert Add 1 unsigned) | 2-com $r1, $r2 |
| booth-load | R | 0x04 | 0x00 | Loads the value of A and B into predetermined registers | booth-load $r1, $r2 |
| Booth-Add | R | 0x09 | 0x00 | This function looks at the last three bits of A and performs the appropriate operation according to the function. | booth-add |

**Assembler Overview & Class**

From a general perspective the assembler needs to have a couple characteristics. 1) The input file needs to be specified we opted to use built in parameter functions when calling a python script. For example the following command can be used to run the script and compile an input assembly file to an output.

python main.py -ifile example.asm -ofile example.mntddw

A custom built python script for this handles the input parameters, however it was omitted from this report. To see the full source code take a look at the Github page, <https://github.com/asteroidice/MD-MARRFB-Assembler>.

The larger, more interesting, part of the assembler is the portion that translates the instructions to machine code. This part of the code is created as a python class. This leads to more organized code and easier expendability and debugging. The python class has three public methods to facilitate the control of assembling code. These methods are as follows.

The initializer (\_\_init\_\_(self, inputfile)) is passed the input parameter *inputfile*. This is a string of the path to the file. The file is opened and every line of the file is added to the *input\_lines* list.

The assemble function (assemble(self)) calls two other private method, *remove\_whitespace()* and *parse()*. These two functions are used for filtering and parsing (or mapping) the assembly code.

Lastly the saveFile function (saveFile(self, outputfile)) is passed a output file path, *outputfile,* and then turns compiled machine code form the *assemble* method to create the output file.

A skeleton of the *Parser* class can be seen on the next page.

class Parser():  
    """  
    Class Variables:  
        file - A file object of the input file.  
        outputfile - A file object of the output file.  
        input\_lines - An array of strings that contain every line of the input file  
        instruction\_lines - A tuple of every instruction and the original line index it was on.  
        mntdw\_lines - An array of strings that are compiled machine code  
        labels - A dictionary of labels and their index in instruction\_lines  
    """  
    mntdw\_lines = []  
    instruction\_lines = []  
    labels = {}  
  
    def \_\_init\_\_(self, inputfile):  
        self.file = open(inputfile, "r")  
        self.input\_lines = self.file.read().split("\n")  
        self.file.close()  
  
    *# A function that handles the assembly of all the code.*  
    def assemble(self):  
        self.\_\_remove\_whitespace()  
        self.\_\_parse()  
   
    *# Save the file to `outputfile`.*  
    def saveFile(self, outputfile):  
        try:  
            os.remove(outputfile)  
        except OSError:  
            pass  
        self.outputfile = open(outputfile, 'w')  
        self.outputfile.write("\n".join(self.mntdw\_lines))  
        self.outputfile.close()  
  
    *#This function creates labels and removes white space from the inputfile.*  
    def \_\_remove\_whitespace(self):  
       *# ...*  
  
    *# This function does some preprocessing and then calls `\_\_parseLine()`.*  
    def \_\_parse(self):  
       *# ...*  
  
    *# This function parses an individual line. It is responsible for calling*  
    *# the instruction parser associated with the line.*  
    def \_\_parseLine(self, instruction\_tuple, address):  
       *# ...*

**Parsing Rules**

With rules in place to translate from instructions to machine code, we set about developing the assembler. The base of this software would be some sort of *for loop* that reads in a text based file, line by line. During this parsing, some text, such as comments and white space (tabs and spaces), should be ignored. The easiest way to do this is with a basic parsing function that uses regular expressions, so we used an online resource to specify and develop the necessary regular expressions. Once these basic assembler-ignore rules had been put in place, and after some deliberation, we decided that we’d also add some basic support for jumps. Jumps require a couple things: a way to parse out labels, and a way to reorder and repeat labeled code sections during assembly. To parse out the labels, we’d need to delimit labels by putting them on their own lines, and marking them with colons. By using some more regular expressions, we’d be able to grab section labels, and hold their identifiers and addresses in program memory. After a file has been parsed for ignorable text and labels, all that should be leftover is lines of instructions and their associated parameters. We came up with some basic design for these parsing rules, regular expressions, and control functions and then set up a basic python program that would be able to execute these rules on instruction files. With this foundation in place, we set to work developing functions to process the leftover lines of instructions and parameters.

*# This function creates labels and removes whitespace from the input.*  
def \_\_remove\_whitespace(self):  
    for index, line in enumerate(self.input\_lines):  
        *# Remove white space characters*  
        new\_line = re.sub(r'\s+', '', line)  
  
        *# REMOVEs COMMENTS from input\_lines*  
        new\_line = re.sub(r'(#|;).\*$', '', new\_line)  
        *# Check if new line is a label.*  
        *# TODO: Allow labels to be on the same line as an instruction.*  
        if re.match(r'^[a-zA-Z]+\w+:+$', new\_line):  
            *# store the name and index of the label*  
            self.labels[new\_line[:-1]] = len(self.instruction\_lines)  
            *# labels don't need to be assembled them.*  
            continue  
        *# Ignore lines that are blank*  
        if new\_line == "":  
            continue  
  
        *# New line should be ready to do*  
        self.instruction\_lines.append((new\_line, index))

def \_\_parseLine(self, instruction\_tuple, address):  
    instruction = instruction\_tuple[0]  
    *# go through every instruction pattern and see if it's a match.*  
    for pattern in INSTRUCTION\_PATTERNS:  
        *# create a regex object and matches it with the instruction.*  
        regex = re.compile(pattern[0])  
        if regex.match(instruction):  
            *# remove the instruction word from the instruction.*  
            instruction\_params = re.sub(regex, '', instruction)  
            *# If instruction parameters exist split them by commas.*  
            if instruction\_params:  
                instruction\_params = instruction\_params.split(',')  
            else:  
                instruction\_params = []  
            *# Call the instruction parser function and give it the*  
            *# instruction object.*  
            return pattern[1]({  
                'params': instruction\_params,  
                'address': address,  
                'line': instruction\_tuple[1],  
                'complete\_instruction': instruction,  
                'labels': self.labels,  
            })  
    raise SyntaxError

**Instruction Processing**

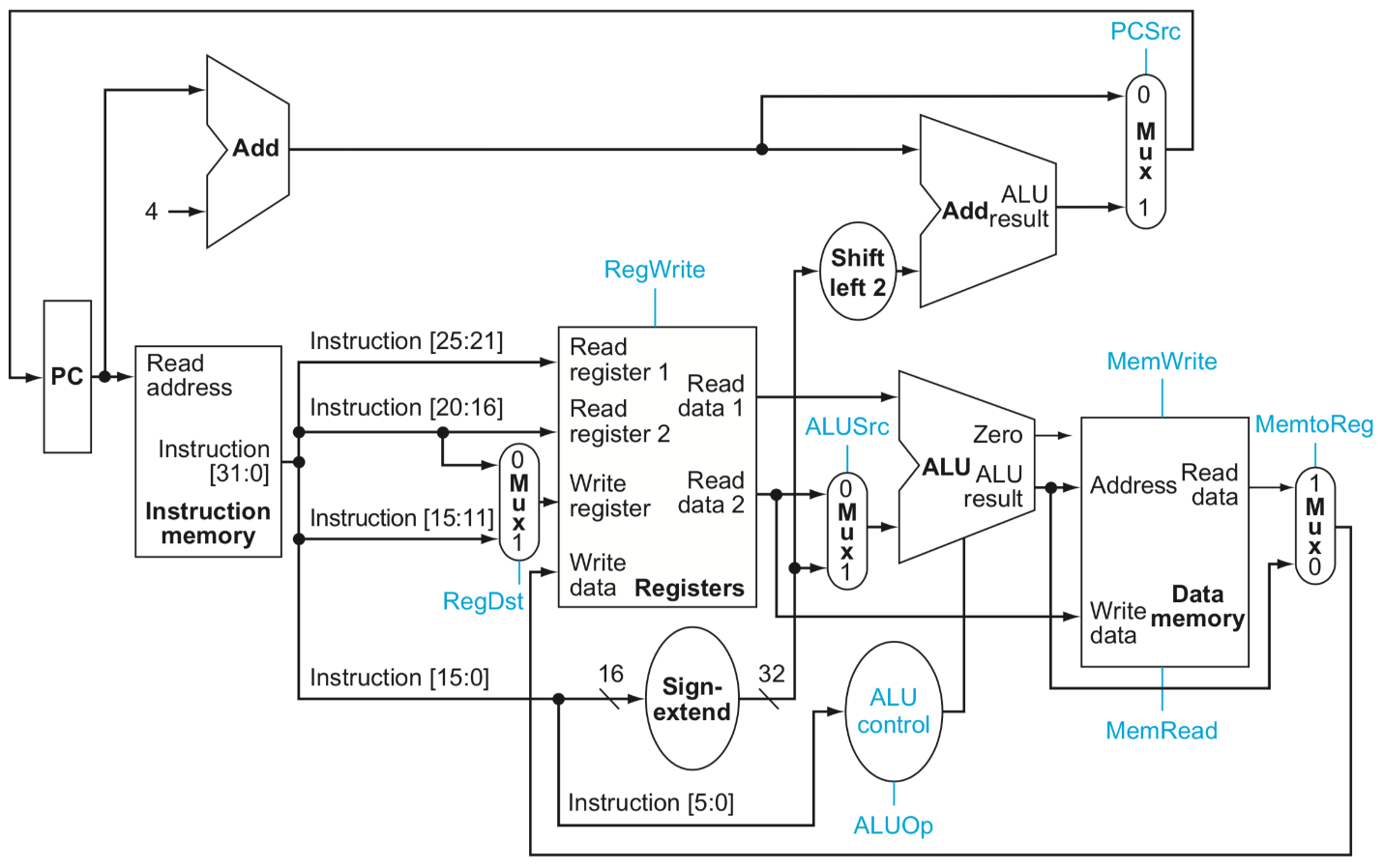
At this point, it’s important to remember that our goal has always been to read through a file of instructions and parameters, and convert those instructions into machine code. Now that we’ve gotten to the point where we can deal with individual instructions, all we have to do is follow the table of translations that we came up with during our assembly code research. This is easy enough, and would be super easy to just whip up a hardcoded python script to do these translations. But, if we use this quick and dirty hardcoded method, and then somewhere down the road, have to change or extend our instruction set, we’d be stuck. Whether we went with dozens of nested if statements, or a switch statement, extending hardcoded instruction set processing would be a massive pain. To allow for easy extensions, we instead opted for a pattern matching function that would identify instruction identifiers and match them to a specific instruction parser in a separate library. Here’s how each of these things work.

The instruction pattern function, which does the matching, uses a regular expression to pull instruction identifiers from each line. The function then cycles through a list of defined identifiers, looking for a pattern match. (It’s important to note here that similarly named instructions like ADDI and ADD cause a bit of an issue, because ADD will match to both of these identifiers.) These match patterns are defined in tuples, along with a string that matches the function code to its associated instruction parser function, which is then called to convert the associated parameters and identifier into machine code.

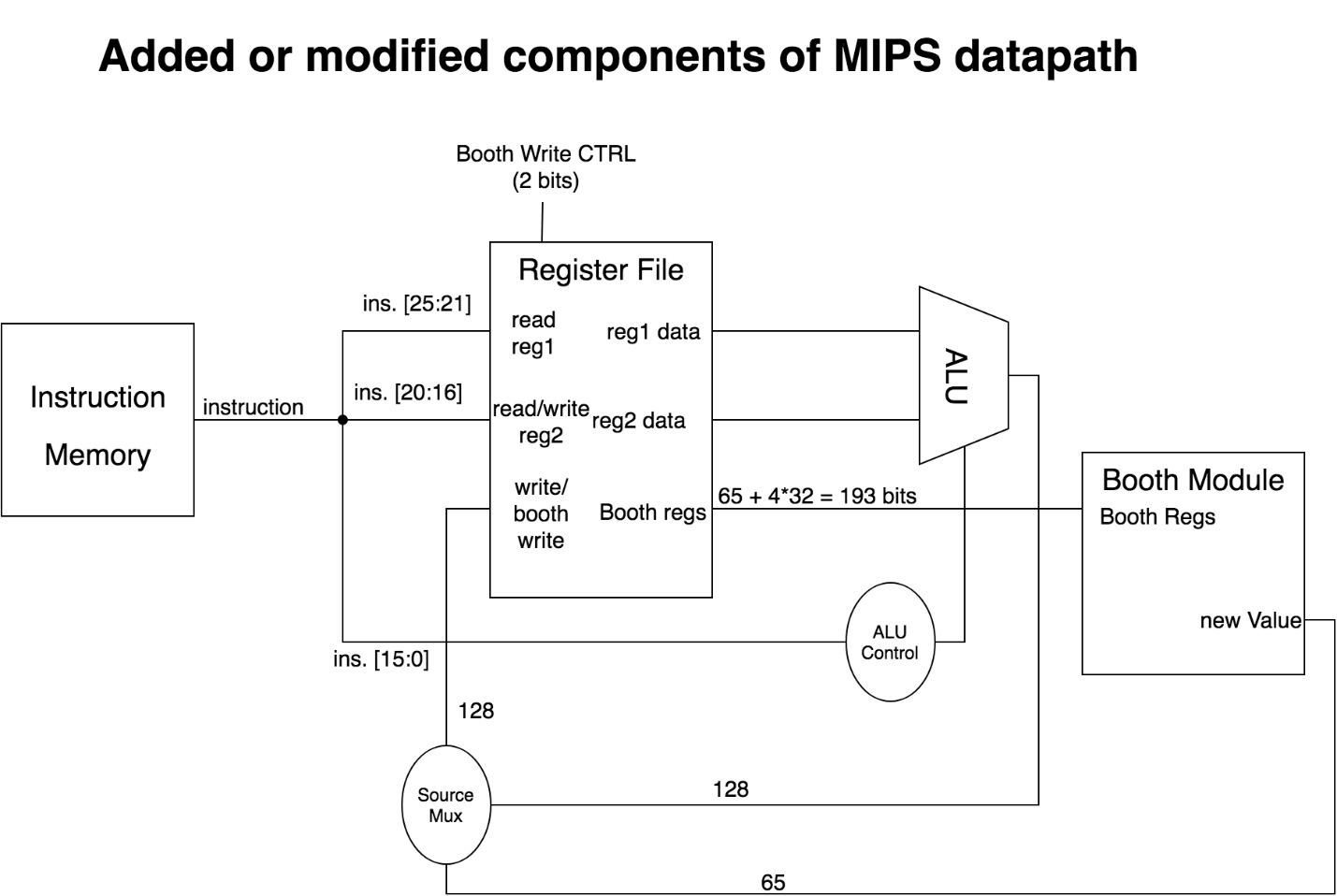
Instruction parsers are all held in a separate library that can be easily changed, or extended. Each instruction identifier matches to or aliases to a parser function. This parser function contains a few helper functions that throw exceptions for syntax errors (for example, if a function requires two registers but the user lists three, or if the function expects immediate terms but the user lists registers) back to the main syntax error object, where the error and associated address/line will e printed to the console.

*# booth-load $t1, $t2*  
def parseBoothLoad(instruction):  
    *# Ensure parameters are valid*  
    check\_params(instruction, ("register", "register"))  
      
    *# instruction['params']: ["$t1", "$t2"]*  
    a\_param = instruction['params'][0]  
    b\_param = instruction['params'][1]  
  
    a\_reg = REGISTERS[a\_param]  
    b\_reg = REGISTERS[b\_param]  
  
    opcode = "000000"  
    func\_code = to\_bin\_string(0x04, 6)  
    source\_reg = to\_bin\_string(a\_reg, 5)  
    target\_reg = to\_bin\_string(b\_reg, 5)  
    shift = "00000"  
    dest\_reg = "00000"  
  
    return(opcode + source\_reg + target\_reg + dest\_reg + shift + func\_code)

**Hardware Implementation**

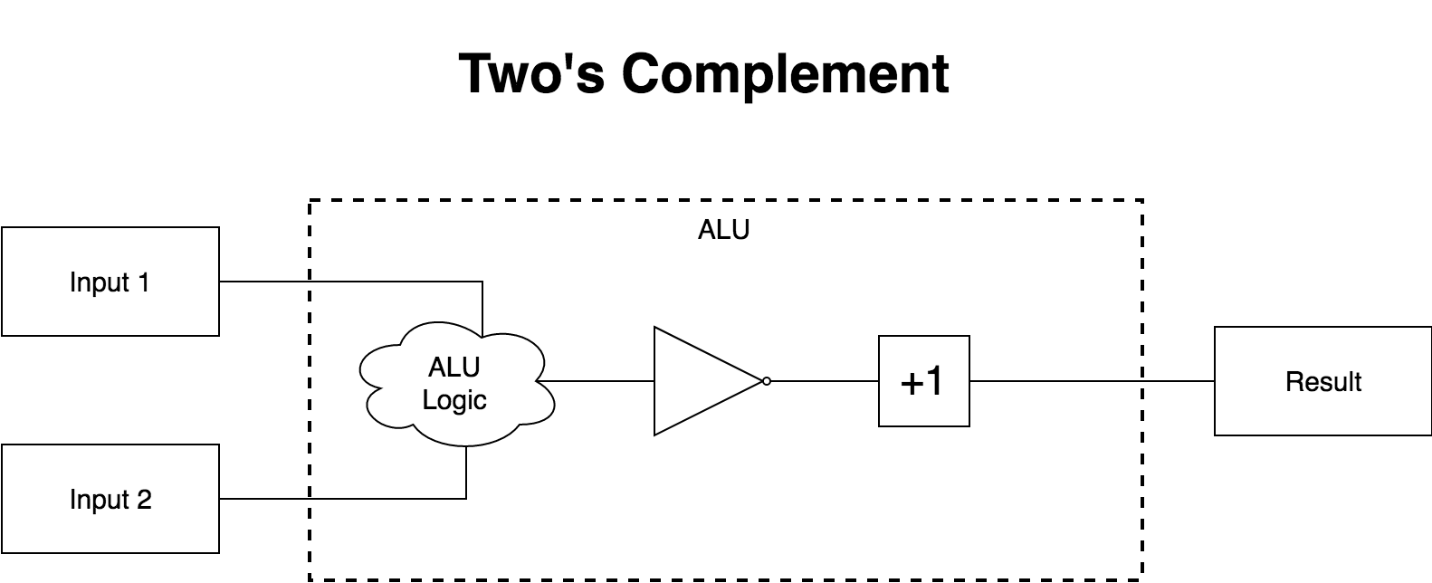


This is a figure of the original MIPS datapath taken from the book. The next figure will focus on some key differences between the standard MIPS and our extended MIPS datapath.

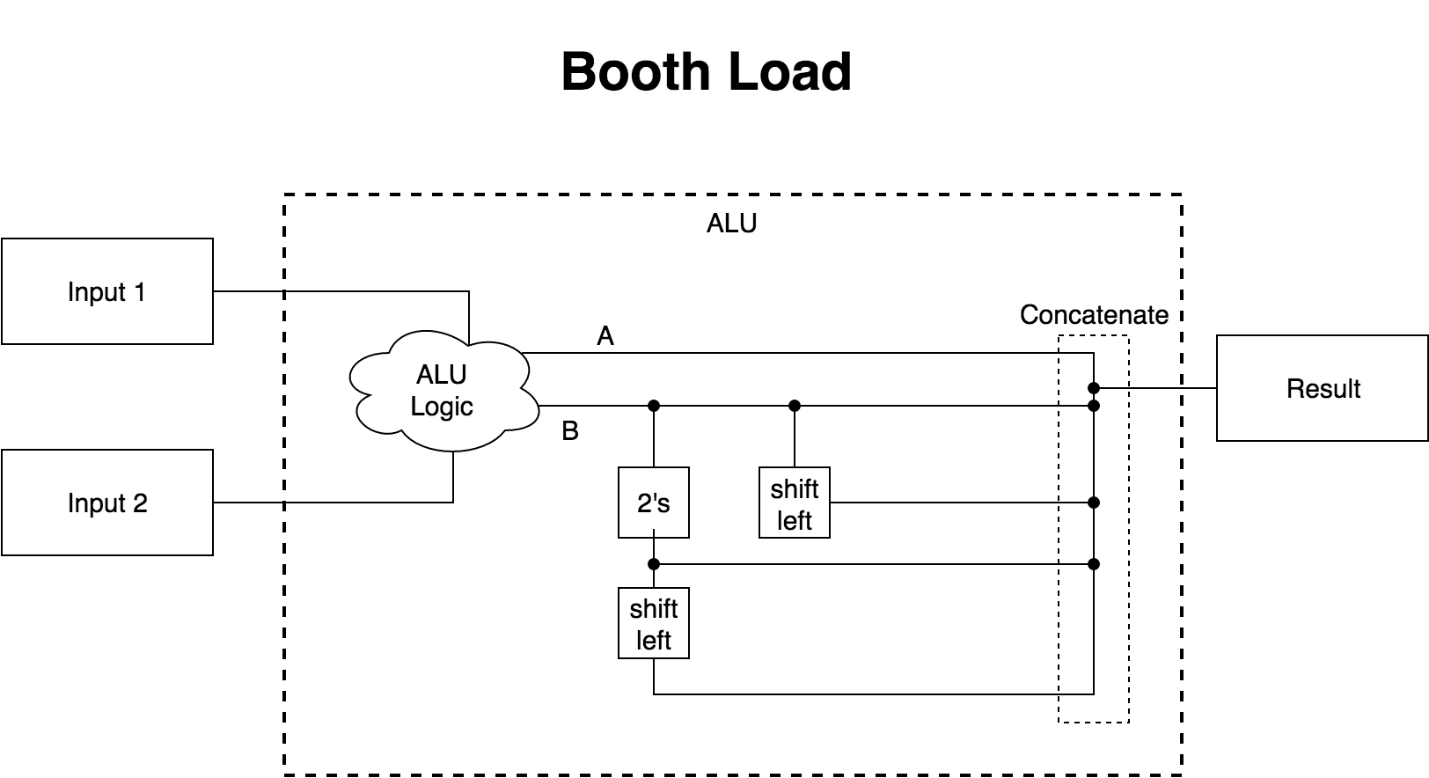


This datapath contains a couple of key differneces from the standard MIPS datapath. Most notably is the addition of another module called the *Booth Module*. This module is responsible for the boot-add command. There are also some minor changes to the ALU. These changes will be covered in more detail on the next couple of pages.

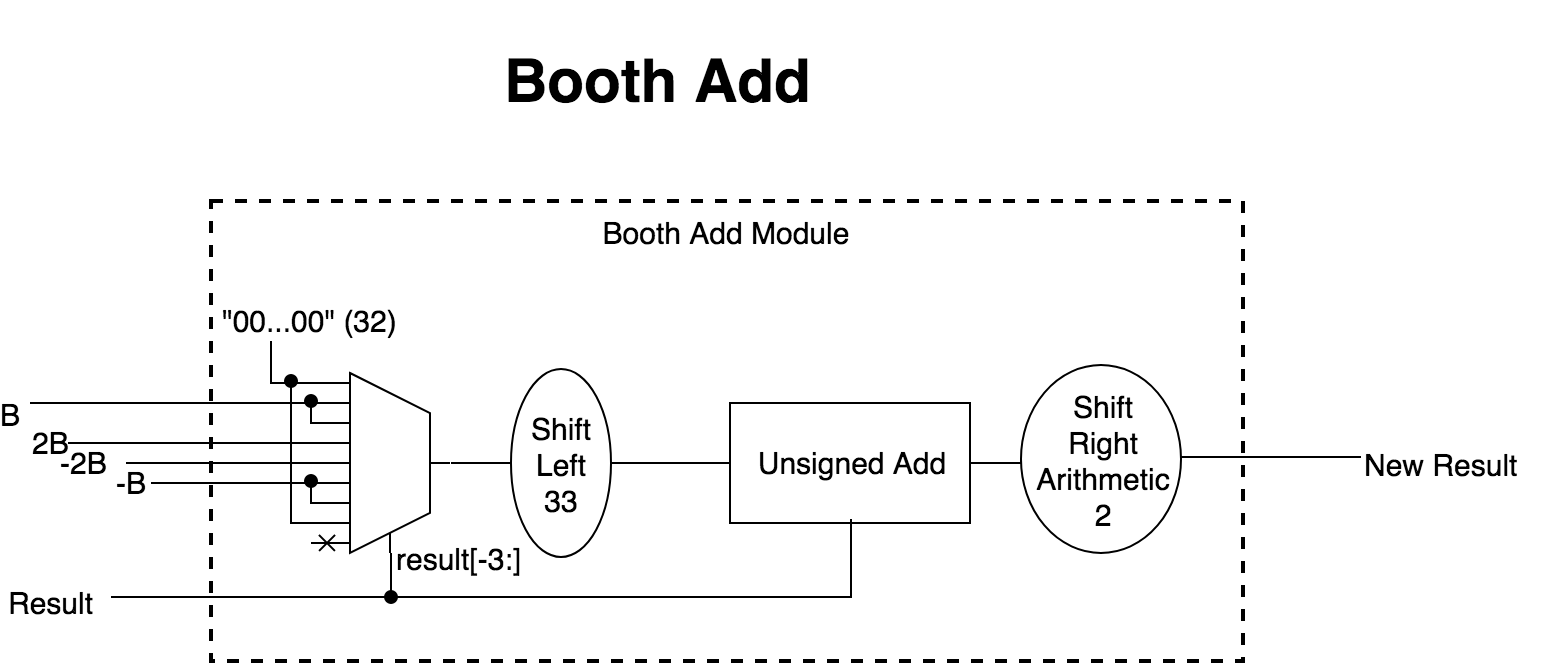
One of the less notible differences between the two datapaths is the Register file. Our design of MIPS requires that some registers be written to at the same time. We also have one register that is 65 bits. For this reason we needed to modify the register file to support read and write operations for these various schemes.



The two’s complement functionality of the ALU is activated when the instructions function code is present. The two’s complement is calculated using the traditional method of inverting all the bits and adding one. Note that if there is an overflow the + 1 module creates an exception.



The booth portion of the ALU writes four values at once back to the register file. This is accomplished calculating and concatenating the appropriate multiples of B. The shift left is used to multiply by two and the 2’s complement module is much like the above two’s complement module.



**Conclusion**

This project was a great way to explore a couple layers of computer architecture. We’ve learned how to build an assembler and moreover learned how to implement custom instructions in every layer in computer architecture. We learned how to decide which assembly instructions would be best to add to our instruction set and how those instructions would be implemented in hardware. We believe that this is the very essence of Computer Architecture and that in this regard our project lead to a greater understanding of Computer Architecture as a whole.

**References**

Python Documentation - <https://docs.python.org/3/>

Instruction Formats - <https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats>

Pseudo Instructions - <https://en.wikibooks.org/wiki/MIPS_Assembly/Pseudoinstructions>

Two’s Comp. - <https://stackoverflow.com/questions/1604464/twos-complement-in-python>

MIPS registers - <http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html>