MD-MARRFB Assembler

Matthew Fennell & Ryan Rabello

What are we going to talk about?

• What is the MD-MARRFD Assembler?
• Project Scope
• Project Objectives
• Project Constraints
• Instruction Set Design
• Machine Code and Hardware Planning
• Assembler Development
What is the MD-MARRFB Assembler?

- Acronym stands for "Mountain Dew-Matt and Ryan Radix Four Booth"
- Our project is based on a hypothetical processor optimized for the Radix-4 Booth’s Multiplication algorithm
- Developing this assembler gave practical experience with wide range of computer architecture topics

Three Big Parts - Scope

- **Instruction Set**: Custom assembly instructions for Radix-4 Booth’s Algorithm
- **Assembler**: Software utility that converts instruction set programs into machine code programs
- **Machine Code**: Hardware inputs for Radix-4 Booth’s Algorithm that are loaded into the processor
Objectives

- Instruction Set Objectives
  - Develop instructions for Radix-Four Booth's algorithm
  - Explain development motivations
  - Provide documentation for instruction set
- Machine Code Objectives
  - Develop format for translation from instruction set to machine code
  - Design simple data path/hardware layout for new components
- Assembler Objectives
  - Build a program to convert from plaintext instructions to machine code
  - Output syntax errors to help with debugging

Constraints

What are we working with?

- Booth’s Radix 4
- Python
- MIPS
Booth’s Radix 4

Step 0.0: A: 1011(-5) | B: 0011(3)
Step 0.1: -B: 1101 | 2B: 0110 | -2B: 1010
Step 1: 0000 10110 (Pad w/ 0, -B)
Step 2: 1101 10110 (Sign-Preserve-Shift Right 2)
Step 3: 1111 01011 (-B)
Step 4: 11100 01101 (Sign-Preserve-Shift Right 2, Ignore Carry)
Step 5: 1111 00011 (Drop Pad)
Step 6: 1111 0001 (Final Result)
Verify: 0000 1111 (Two’s Complement is 15, confirms result is -15)

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>NA</td>
</tr>
<tr>
<td>001</td>
<td>+B</td>
</tr>
<tr>
<td>010</td>
<td>+B</td>
</tr>
<tr>
<td>011</td>
<td>+2B</td>
</tr>
<tr>
<td>100</td>
<td>-2B</td>
</tr>
<tr>
<td>101</td>
<td>-B</td>
</tr>
<tr>
<td>110</td>
<td>-B</td>
</tr>
<tr>
<td>111</td>
<td>NA</td>
</tr>
</tbody>
</table>

Design Process

- Used HW and Radix-4 examples as starting point
- Grouped instructions to simplify
- Recognized the need for 3 custom instructions
  - Booth Load
  - Booth Add
  - Two’s Complement (not actually needed)
### Machine Code

- Extend MIPS architecture where room is available
- Develop hardware for custom instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Type (I,R,J)</th>
<th>Function Code</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2's Comp</td>
<td>R</td>
<td>0x01</td>
<td>0x00</td>
<td>A function for inverting then adding one to the bits. (Bitwise invert Add 1 unsigned)</td>
</tr>
<tr>
<td>booth-load</td>
<td>R</td>
<td>0x04</td>
<td>0x00</td>
<td>Loads the value of A and B into predetermined registers</td>
</tr>
<tr>
<td>Booth-Add</td>
<td>R</td>
<td>0x09</td>
<td>0x00</td>
<td>This function looks at the last three bits of A and performs the appropriate operation according to the function.</td>
</tr>
</tbody>
</table>

### Hardware

**How does it work?**

- Basic MIPS datapath
- New Datapath
- Booth-Load
- Booth-Add
- Two's Complement
Basic MIPS datapath

Added or modified components of MIPS datapath
Booth Load

Booth Add
Assembler

Python functions that do the heavy lifting.

- Parsing
- Mapping
- Helpers/Checkers
- Sample IO

Two's Complement

Input 1

ALU Logic

ALU

+1

Result

Input 2
Parsing (self.parse())

- Things to ignore
  - White Space
  - Comments
- Tracking Labels

```python
def __remove_whitespace_and_labels(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'(\#)|(^$)', '', new_line)
        # Check if new line is a label.
        if re.match(r'^[a-zA-Z]+w+:$', 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.
            continue
        # Ignore lines that are blank
        if new_line == '':
            continue
        # New line should be ready to do
        self.instruction_lines.append((new_line, index))
```

Mapping Instructions (self.__parseLine())

- Instruction keyword removed
- Instruction Parameters parsed
- Instruction parser called
- Instruction parser returns assembled machine code

```python
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
```
Example Instruction Parser

- check_params() Helper
- to_bin_string() Helper

```python
# 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 = "00000"
    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)
```

Sample Input/Output

```
li $t0, 0               # initialize counter
li $t1, 4               # Load A ($t1) and B ($t2)
li $t2, 6

booth-load $t1, $t2    # Setup A, B, and -b
start:
  booth-add
  sra $b0, $b0, 2       # Shift $b0 by 2
  add $t0, $t0, 1      # Increment the counter
  slti $a0, $t4, 16    # set if $t4 is less than 16
  bne $a0, $zero, start # If a0 = 0 then continue

console
  li $v0, 1
  move $a0, $t1
  syscall

  li $v0, 10             # exit the program
```
Extendability

- Registers and functions can be easily changed for use in different processors or with different instruction sets
- Instructions and registers are defined dynamically in separate files

```
REGISTERS = {
    "$zero": 0,
    ...
    "$t0": 8,
    "$t1": 9,
    ...
    "$b0": 16
}

INSTRUCTION_PATTERNS = [
    ('addi', parseAddImmediate),
    ('add', parseAdd),
    ('booth-load', parseBoothLoad),
    ('booth-add', parseBoothAdd),
    ...
]
```

Conclusion

Respect your assembler

- Datapath Architecture is challenging
- Python is great!
- Regular Expressions are VERY useful and VERY magical.
- Learned a lot about opcodes and pseudo instructions
- Teaching a computer to do something is a great way to learn it yourself
Questions?

References

https://docs.python.org/3/

https://en.wikibooks.org/wiki/MIPS_Assembly/Instruction_Formats

https://en.wikibooks.org/wiki/MIPS_Assembly/Pseudoinstructions

https://stackoverflow.com/questions/1604464/twos-complement-in-python

http://www.cs.uwm.edu/classes/cs315/Bacon/Lecture/HTML/ch05s03.html (regs)