Table of Contents

Introduction 3
Description 3
Instructions 3
Description 3
Design 3
Implementation 4
Testing 5
Component Testing 5
Implementation Testing 5
Datapath Testing 6
Unique Features 8
Single Instruction Format 8
Single Programmer Accessible Register 8
Subroutine Return Values and Arguments Placed on Stack 8
Extra Features 9
Assembler 9
Assembler Pseudo-instructions 10
Assembler Arguments 10
Design Document 11
Design Journals 11
Instruction Set 12
RTL 14
Datapath 16
Components 16

Introduction

Description

Our processor implements an accumulator design, which includes a register which is the basis for most of our instructions. We included special registers such as the BA, SA, stack pointer and register address, which all serve to make instructions easier to both run and understand. These registers alongside memory will be able to perform conditional, algorithmic, and logical operations in order to support any type of program written for it. Our design excels at having lots of pseudo instructions, so programmers using our processor are not confused by the more complicated instructions.

Instructions

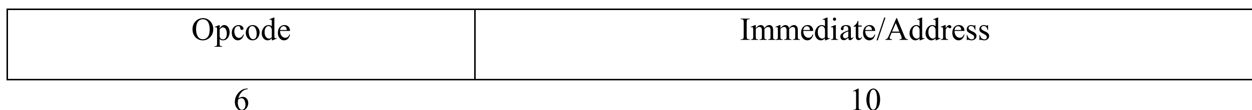
Description

Our design boasts a total of 27 instructions, which all serve to make any program imaginable work on our processor. With these 27 instructions we've made our processor extremely versatile for both programs and even pseudo instructions

Design

We decided to go with only one instruction type, which we named the Universal Type (U-Type) and it takes in a 6-bit opcode and a 10-bit Immediate and/or Address. We decided to go with only one because it makes it much easier to understand exactly what each instruction does, which makes programming in our processor very easy.

Here is a visualized model of our U-Type instruction.



Opcode	Immediate/Address
6	10

To help better understand our instruction design, I have included a couple of instructions with explanations below:

ADD m	This instruction would Add the value in the address m and put the result into the accumulator
BEZ Test	This instruction would check if the accumulator equaled zero and if it did it would branch to the label Test
SL 2	This instruction would shift the value in the accumulator by 2
JUMP GCD	This instruction would set the PC to the address at the label GCD

Implementation

The implementation for each instruction took a lot given that each instruction did something different. So, we decided to build a **Datapath** and then spilt it up into parts that we could implement and test separately. We had 5 implementation steps for our Datapath, which were

1. Control	Control was our basis for controlling every part of the data path, which is why this was one of the few separate implementation plans
2. ALU/ALUOut	ALU and the register ALUOut were combined because ALUOut heavily depending on whatever the ALU outputted
3. Instruction Memory/PC	PC and Instruction were combined because PC determined the next instruction that would run through the Datapath
4. Data Memory	Data memory was the other implementation step that only included one component. This was because Data Memory was fairly large and our team needed it to work alone before it was connected to anything else.
5. Combing Factors	This step of the implementation plan was mostly combining sign extenders, shifters, concatenation components, etc. to the other implementation plans

Testing

Testing was extensive when it came to our Datapath given the fact that we would like for every part of the Datapath to work. We tested each component individually. Once every component worked, we tested the implementation plans and lastly after those were done being tested we fully tested the Datapath with the instructions and their translations below.

Component Testing

Here we will provide some examples of some of the components tested and how exactly we tested them

Sign Extender	With the sign extender we made sure to test putting in both a negative and positive value to ensure that the extension worked with both zeros and one's
ALU	With the ALU we decided to test all the ALU's opcodes to make sure it functioned as it should.
Instruction Memory	With instruction memory we made sure that if we passed in instructions address it would output that instructions' bits
Control	With control we tested different opcodes and ensured that they all outputted the correct control bits

Implementation Testing

Each part of the implementation was tested to make sure connecting different components together didn't break anything significant

1. Control	As pointed out in the above control test, an opcode was passed through, and we ensured that each opcode corresponded to the correct control signals
2. ALU/ALUOut	This implementation plan test was basically testing the ALU to make sure its opcodes worked, and then using that output and storing it inside of ALUOut
3. Instruction Memory / PC	With instruction memory we made sure to make tests of code snippets, and then ensure that the instruction and PC matched.

4. Data Memory — Testing Data memory consisted of ensuring that passing in a memory address and data would write that passed in data into the specified memory address.

5. Combing Factors — The combining factors were mostly tested with Datapath, and Component based testing.

Datapath Testing

With Datapath testing, after each step of our implementation passed testing, we passed code snippets of an array of our instructions to make sure each instruction worked both individually and in combination with others.

The code snippets used to test the Datapath are provided below:

Datapath Testing		
Arithmetic	Memory	Subroutine
ADDIMM 5 # AccOut = 5 SUBIMM 2 # AccOut = 3 ORIMM 50 # AccOut = 51 ANDIMM 30 # AccOut = 18 CMPE 18 # AccOut = 1 ORIMM 35 # AccOut = 35 CMPLT 100 # AccOut = 1 ORIMM 35 # AccOut = 35 CMPLT 30 # AccOut = 0 ORIMM 8 # AccOut = 8 SL 1 # AccOut = 16 SR 2 # AccOut = 4	DEFINE x, y ANDIMM 0 ORIMM 4 STORE x # AccOut = 4 # x = 4 ADDIMM 2 # AccOut = 6 STORE y # y = 6 LOAD x # AccOut = 4 ADD y # AccOut = 10 SUB x # AccOut = 6 AND x # AccOut = 4 OR y # AccOut = 6 # AccOut-- until AccOut = 1 LOOP:	ALLOCATE 2 ANDIMM 0 ORIMM 55 # AccOut = 55 PUSH 0 # Mem[0] = 55 # AccOut = 55 ANDIMM 0 ORIMM 45 # AccOut = 45 PUSH 1 # Mem[1] = 45 # AccOut = 45 JUMPL ADD_FUNCTION # PC -> ADD_FUNCTION PULL -1 # AccOut = 100 ANDIMM 0 ORIMM 50 DEFINE b ADD_FUNCTION: PULL 1 # AccOut = 45 STORE b # b = 45

	CMPE 1 BNEZ BREAK LOAD y SUBIMM 1 STORE y JUMP LOOP BREAK: # AccOut = 1 ALLOCATE 3 PUSH 0 # Mem[0] = 1 ADDIMM 10 # AccOut = 11 PUSH 1 # Mem[1] = 11 ADDIMM 10 # AccOut = 21 PUSH 2 # Mem[2] = 21 PULL 0 # AccOut = 1 PULL 1 # AccOut = 11 PULL 2 # AccOut = 21 DEALLOCATE 3	PULL 0 # AccOut = 55 ADD b # AccOut = 100 ALLOCATE 2 # AccOut = 100 PUSHRA 0 # Mem[0] = RA # AccOut = 100 PUSH 1 # Mem[1] = 100 PULL 0 # AccOut = 0x10 DEALLOCATE 2 # SP = 0x3fb JUMPACC # PC = 0x10

Unique Features

Single Instruction Format

Our programming language utilizes a single instruction type U (for Universal). This means that all instructions have a 6-bit opcode followed by 10-bits of an immediate or an address. It adds to the simplicity, programmability, and ease of translation to our programming language. Given a file with machine code, a user can easily interpret the instruction, which helps with debugging immensely. The 6-bit opcode also allows for easy future modification of our programming language and the addition of many pseudo-instructions to aid with programmability.

Single Programmer Accessible Register

In order to keep our processor as close to pure accumulator-based design as possible, the accumulator is the only programmer accessible register. Other registers are implemented in the data path to accommodate a multicycle process (e.g. IM, ALUOut, and PC registers) but are not available to the user.

Subroutine Return Values and Arguments Placed on Stack

Due to our desire to keep our processor close to pure accumulator, we place return values and arguments on the stack during subroutines instead of into special registers. In our calling conventions, you can see that we store returned values at the stack pointer –1 and –2 and the return address at the stack pointer. This makes our processor unique from Load Store or less pure accumulator-based designs.

Extra Features

Assembler

The assembler for NLS is highly efficient, flexible, extensible, and easy to use. It is written in python and works as a command-line program. Assembly files can contain blank lines and comments that begin with '#'. To run the assembler, the user must run it from the command line with their assembly file as an argument, like so:

```
>python assemble.py ./my_assembly_file.asm
```

The assembler will automatically assemble it and write the binary result to a .mif file in the same directory as the assemble.py file. The result of the above command would be the file "my_assembly_file.mif".

The user can also add the flag "--debug-mode" if they want to see the PC value and instruction next to each line of machine code output. For example:

```
>python assemble.py relprime.asm --debug-mode
Assembling "relprime.asm"...

LABELS: {RESULTLOOP: 0x8, RELPRIME: 0xc, WHILE: 0x1a, RELPRIMEDONE: 0x32, GCD: 0x3c,
GCDWHILE: 0x56, ELSE: 0x68, RETURN_A: 0x70}
VARS: {m: 0x200, n: 0x202, a: 0x204, b: 0x206}

   0x0  0100010000000010   |   ALLOCATE 2
   0x2  0100110000000001   |   PUSH 1
   0x4  0011000000000110   |   JUMPL RELPRIME
   0x6  0101001111111111   |   PULL -1
   0x8  0001010000000000   |   ORIMM 0
   0xa  0010110000000100   |   JUMP RESULTLOOP
   0xc  0000110000000000   |   ANDIMM 0
   0xe  0001010000000010   |   ORIMM 2
  0x10  0011101000000000   |   STORE m
  0x12  0101000000000001   |   PULL 1
  0x14  0011101000000010   |   STORE n
  0x16  0100010000000011   |   ALLOCATE 3
  0x18  0101010000000000   |   PUSHRA 0
  0x1a  0011011000000010   |   LOAD n
  0x1c  0100110000000001   |   PUSH 1
  0x1e  0011011000000000   |   LOAD m
  0x20  0100110000000010   |   PUSH 2
  0x22  0011000000011110   |   JUMPL GCD
                    . . .
```

As seen above, the debug flag makes it easy for the user to see how their instructions correlate to machine code, making it easy for them to work through problems on the machine.

Assembler Pseudo-instructions

As stated above, the assembler is highly extensible. New pseudo-instructions can be added to the assembler with ease. In the inst.json file, the user can add a new json entry that contains the new pseudo-instruction's name and sub-instructions.

For example, this pseudo-instruction 'LOADIMM' will set the accumulator to 0 using ANDIMM 0, then use ORIMM to fill the accumulator with the desired immediate:

```
"LOADIMM": {
    "is_pseudo": true,
    "parts": [
        "ANDIMM 0",
        "ORIMM {0}"
    ]
},
```

The {0} is the first argument of the pseudo-instruction, which gets replaced with the real argument automatically. 'LOADIMM 5' means that the 'ORIMM {0}' will then become 'ORIMM 5'. The user can have as many arguments as they want, and use {1}, {2}, and so on to fill them in to the sub-instructions.

Assembler Arguments

Required

Argument name	Argument type	Description
(None)	String	The path of the input assembly file

Optional

--output-file-path -o	**String**	**The path of the output**
--debug-mode -d	Flag (on/off)	Enable debug mode
--hex-mode	Flag (on/off)	Output hexadecimal machine code instead of binary

Conclusion

While working on this processor, our greatest issues arose from the single-register nature of the accumulator architecture and our planning of our testing stages. To address those, our processor has special registers such as a stack pointer and register address, we were able to get thorough tests in the late stages of implementation that helped us track down small bugs. After debugging our entire Datapath, our processor is able to fully run all of our instructions, and programs. The biggest part about our processor is the highly efficient assembler as well as our single instruction format and subroutine calls placed on the stack.