CSE 141L Lab 1 Report

Anahita Afshari, Meron Asfaw

Introduction

Our ISA is an accumulation instruction set architecture that minimizes the number of available R-type and I-type binary-encoded, 9-bit instructions without losing the ability to execute the given programs. Our design philosophy is maximizing the amount of registers to use efficiently within our small set of developed instructions, so that we can minimize the number of bits that our op-codes use. By minimizing the number of bits our opcodes take, we can allow for more registers in the architecture.

Instruction Formats & Operations + Examples of Assembly to/from Machine Code

<table>
<tr><th>Type</th><th>8th bit</th><th>7nd bit</th><th>6rd bit</th><th>5th bit</th><th>4th bit</th><th>3th bit</th><th>2th bit</th><th>1th bit</th><th>0th bit</th></tr>
<tr><td rowspan="8">R</td><td rowspan="8">0</td><td colspan="2">2-bit opcode</td><td colspan="2">2-bit destination register</td><td colspan="2">2-bit target register</td><td colspan="2">2-bit source register</td></tr>
<tr><td>Function</td><td colspan="2">Opcode</td><td colspan="2">Example</td><td colspan="3">Explanation</td></tr>
<tr><td><code>add</code></td><td colspan="2"><code>00</code></td><td colspan="2"><code>R2 = R1 + R0</code>
<code>add R2,R1,R0</code>

<code>R2 = R1 + 2’R0 + 1</code></td><td colspan="3">Adds/Subtracts operand register from source register</td></tr>
<tr><td colspan="2">Example:</td><td colspan="2"><code>0_00_100100</code>
<code>0x024</code></td><td colspan="3"><code>add</code> <code>R1</code> to <code>R0</code> and store in <code>R2</code></td></tr>
<tr><td><code>xor</code></td><td colspan="2"><code>01</code></td><td colspan="2"><code>R2 = R0 ^ R1</code></td><td colspan="3">Write the result of the source and target registers into the destination register</td></tr>
<tr><td colspan="2">Example:</td><td colspan="2"><code>0_01_100100</code>
<code>0x064</code></td><td colspan="3"><code>xor</code> <code>R1</code> to <code>R0</code> and store in <code>R2</code></td></tr>
<tr><td><code>shift</code></td><td colspan="2"><code>10</code></td><td colspan="2"><code>R3 = (R3<<1) | R3</code></td><td colspan="3">Shifts the destination register by a value; has 2-bit destination register, 1-bit direction determinator (0 for left, 1 for right) and last 3 bits for the value to be shifted by</td></tr>
</table>

CSE 141L Lab 1 Report

Anahita Afshari, Meron Asfaw

		Example:		`0_10_110010`
				`0x0B2`
				
				`0_10_111011`
				`0x0BB`
					`shift left` `R3` by #2 positions
					
					`shift right` `R3` by #3 positions

		`and`	`11`	`R2 = R0 & R1`	**Write** the result of the source and target registers into the destination register
		Example:		`0_11_100100` `0x0E4`	**`and`** `R1` with `R0` and store in `R2`
I	1	2-bit opcode		6-bit source register or two 3-bit registers	
		Function	*Opcode*	*Example*	*Explanation*
		`lw`	`00`	`lw R2, R0`	**Loads** value at the address of the source register
		Example:	`1_00_010000` `0x110`		**`load`** value of `R0` into `R2`
		`sw`	`01`	`sw R2, R3`	**Stores** value at the address of source register
		Example:	`1_01_011010` `0x15A`		**`store`** value of `R3` in `R2`
		`beq`	`10`	`br somewhere`	**Branches** to some address
		Example:	`1_10_111111` `0x1BF`		**`branch`** `to stp`
		`stp`	`11`	`stp program`	Our **stop** signal so that our program execution is stopped at the right time
		Example:	`1_11_111111` `0x1FF`		**`end operation`**

CSE 141L Lab 1 Report

Anahita Afshari, Meron Asfaw

Internal Operands

R0	Quick arithmetic and logic operations like ADD and XOR
R1	
R2	
R3	
R4	General storage
R5	
R6	
R7	
R8	
R9	
R10	
R11	Branching addresses
R12	
R13	
R14	
R15	Accumulator

Control flow branches

We support traditional branching, denoted with the instruction **beq.** Target addresses are calculated based on their current bit location, and the maximum branch distance has a 6-bit length, meaning the maximum branch distance that we support is #63 or 0x3F.

Addressing modes

Memory addressing is supported directly in a 6-bit length relative to the register's location.

CSE 141L Lab 1 Report

Anahita Afshari, Meron Asfaw

Machine classification

Our machine can be classified as a hybrid between an accumulator, load-store, and register machine, because we use elements of all three. We utilize many registers and denote specific registers for specific functions, we load and store with instructions based on the `rd/rs/rt` load-store architecture standard, and the accumulator is used when we do destructive instructions like for example A = A + B, which we have shown below using our instructions:

```
load R0 // A
load R1 // B
add R0, R0, R1
 store R0
```