Nicholas Allaire, A10639753
Nathaniel Perkins, A11727588
CSE 141L
4/24/17

Lab 1 Write-up

1. **Introduction**
    a. Architecture Name: PLZ
    b. We wanted to create an accumulator architecture that is able to take in more than 1 operand and support 16 registers. We did this with a 3 bit opcode and 3 bit func code, for a total of 15 instructions and used an implicit r1 register to be able to access 16 registers with only 3 bits.

2. **Instruction Formats**
    a. Type 1: 3 bits for opcode, 3 bits for register, 3 bits for immediate
        i. Example: blt r2, r3 -> 001 010 011
    b. Type 2: 3 bits for opcode, 6 bits for immediate
        i. Example: set 46 → 110 101 110
    c. Type 3: 3 bits for opcode, 3 bits for reg, 3 bits for immediate
        i. Example: sti r4, 5 → 111 100 101
    d. Type 4: 3 bits for opcode, 3 bits for reg, 3 bits for func code
        i. Example: add r2 → 000 010 011

3. **Operations**

Instruction Name	opcode	func code	Instruction Format	9 bit distribution	Description of implementation	Example
blt	001	-	Type 1	3 opcode, 3 reg, 3 reg1	if reg < reg1, go to label in r0	blt reg, reg1
beq	010	-	Type 1	3 opcode, 3 reg, 3 reg1	if reg == reg1, go to label in r0	beq reg, reg1
sl	011	-	Type 1	3 opcode, 3 reg, 3 reg1	reg = reg << reg1	sl reg, reg1
sr	100	-	Type 1	3 opcode, 3 reg, 3 reg1	reg = reg >> reg1 (pad with signed bit)	sr reg, reg1
sro	101	-	Type 1	3 opcode, 3 reg, 3 reg1	reg = reg >> reg1 (pad wih ov bit)	sro reg, reg1
set	110	-	Type 2	3 opcode, 6 imm	r0 = imm	set 46
sti	111	-	Type 3	3 opcode, 3 reg, 3 imm	reg = imm	sti reg, 5
halt	000	000	Type 4	3 opcode, 3 free, 3 func	done	halt
lw	000	001	Type 4	3 opcode, 3 reg, 3 func	reg = M[r0]	lw reg
sw	000	010	Type 4	3 opcode, 3 reg, 3 func	M[r0] = reg	sw reg
add	000	011	Type 4	3 opcode, 3 reg, 3 func	r0 = r0 + reg + ov	add reg
comp	000	100	Type 4	3 opcode, 3 reg, 3 func	reg = -reg	comp reg
mov	000	101	Type 4	3 opcode, 3 reg, 3 func	reg = r0	mov reg
j	000	110	Type 4	3 opcode, 3 free, 3 func	jump to r0	j
clr	000	111	Type 4	3 opcode, 3 free, 3 func	ov = 0	clr

4. **Internal Operands**
    a. We are an accumulator architecture, and r0 is our implicit accumulator register. To be able to access 16 registers with 3 bits, we designated r1 to be an implicit register that specifies which block of registers we are accessing. It can store value 0,1, and 2 to access different sections of our register file. We wanted to be able to access r0 and r1 no matter which block we were in, and those are addressed by 000 and 001. The rest of the registers are divided into separate blocks as follows:
        i. With r1 = 0, codes 010 - 111 access registers r2-r7
        ii. With r1 = 1, codes 010 - 111 access registers r8 - r13
        iii. With r1 = 2, codes 010 and 011 access registers r14 and r15

5. **Control Flow (branches)**
    a. We support two types of branches: beq and blt
        i. Beq: The target addresses are calculated by using the 6 bit signed value stored in r0. The instruction will check if reg1 == reg2, then go to the address stored in r0, else continue with next instruction. The maximum branch distance supported is 31 instructions forward and 32 instructions backward.
        ii. Blt: The target addresses are calculated by using the 6 bit signed value stored in r0. The instruction will check if reg1 < reg2, then go to address stored in r0, else continue with next instruction. The maximum branch distance supported is 31 instructions forward and 32 instructions backward.
        iii. J: The target addresses are calculated by using the 6 bit signed value value stored in r0, i.e. set 27; j;

    We want to later use a LUT and switch to absolute branching rather than relative branching, supporting up to 63 different labels.

6. **Addressing Modes**
    a. We use a word accessible memory. In order to store or load values from memory, we first load the address into r0, then specify which register to load from/store to in the instruction. For example, to store a value at 0x08 from r3, we would set 8, sw r3;

7. **Main Memory Size**
    a. Main memory is 256 bytes.

8. **Optimize For Dynamic Instruction Count**
    a. Because of the way we designed our ISA, we strayed away from the traditional accumulator architecture by not always storing values in r0 (like in our shift instructions) which takes away the need to move values from r0 to different registers. We also support 15 instructions by utilizing a func code for type 4 instructions, which allowed us to create more specialized instructions overall, which in turn, decreases dynamic instruction count. For example, instead of always shifting left by 1, we added functionality to be able to include a shamt in another register. If we wanted to shift left by 4, instead of having to shift left four times in a row, we can shift left 4 in one instruction.

9. **Optimize For Short Cycle Time**
    a. To optimize for a short cycle time, we designed our ISA to take many instructions(15 of them) that are all very simple. For example, in MIPS there are multiply and divide instructions that would increase the cycle time, as those instructions are fairly complex when compared to shifts, stores, etc. Our most complex instruction is an add, so our cycle time should be fairly short.

10. **Why Our ISA Is Better**
    a. Our competitor may have more functionality with their ISA because they have 32 instructions, but is very limited on the size of data they can work with. While we support many less instructions than they do, our 15 instructions were all we needed to be able to complete the 3 different tasks at hand. We are also able to access 16 different registers, which minimizes the amount of time we would have to write and read to memory. Because we can read from as many registers as we want in 1 clock cycle, we have access to 16 different values at any one time, while they would only be able to access 2 different values per clock cycle.

11. **2 More Bits**
    a. If we had 2 more bits for instructions, for type 4 instructions we would keep 3 bit opcode and have 4 bit registers and 4 bit function code. This increases the number of instructions with function code from 8 to 16, allowing us to have specialized instructions like increment, decrement, because we only need 1 register to do this. Type 1 instructions would have 3 bit opcode, 4 bit for each register, allowing us to access all 16 registers at one time because with 3 bits for registers we have to switch between different blocks to access certain registers. Type 2 instructions would have 3 bits for opcode and 8 bits for immediate, which would allow us to branch and jump further in our program. Type 3 instructions would have 3 bits for opcode, 4 bits for register, and 4 bits for immediate because would be able to access all 16 registers and have a maximum immediate value of 15 instead of 7.

12. **Classical Classification**
    a. Has the main concept of an accumulator by having r0 as the implicit register. It strays away from the traditional accumulator architecture, however as we are able to take in more than 1 operand for many of our instructions. As we base our instructions on computations between registers, it behaves like a register-register type as well. I would classify our ISA as an accumulator, reg-reg hybrid.

13. **Assembly Instruction Example:**
    a. blt r2, r3 → 001 010 011

** Assembly code on next page, read page by page (1st column 1st page → 2nd column 1st page → 1st column 2nd page, etc.)**

14. **Cordic**
    a. Dynamic Instruction Count: 1712

15. **String Match**
    a. Dynamic Instruction Count: 9382

16. **Signed Integer Division**
    a. Dynamic Instruction Count: 686

Cordic Code:

```
// mem 0x01 MSW X
// mem 0x02 LSW X
// mem 0x03 MSW Y
// mem 0x04 LSW Y
// mem 0x05 MSW R
// mem 0x06 LSW R
// mem 0x07 MSW Theta
// mem 0x08 LSW Theta

// Registers initially 0
// ov - overflow
// r0 - Implicit accumulator

// BLOCK 0
// r1 - Implicit reg index
// r2 - index of loop
// r3 - X MSW
// r4 - X LSW
// r5 - X_temp MSW
// r6 - X_temp LSW
// r7 - temp

// BLOCK 1
// r8 - Y MSW
// r9 - Y LSW
// r10 - Y_temp MSW
// r11 - Y_temp LSW
// r12 - temp
// r13 - temp

// BLOCK 2
// r14 - T MSW
// r15 - T LSW
// others - temp

// load in X
set 1
lw r3
lw r5
set 2
lw r4
lw r6

// load in Y
sti reg1, 1
set 3
lw r8
lw r10
set 4
lw r9
lw r11

mainloop:
// y >= 0
sl r8
sro r8

set 1
mov r12
set 0
add r13
mov r13
clr
set else
beq r12, r13

sti r1, 0
comp r5
sti r7, 1
comp r7
set 0
add r5
add r7
mov r5
clr

comp r6
set 0
add r5
mov r5
clr
set done_if
j
```

```
else:
sti r1, 1
comp r10
sti r12, 1
comp r12
set 0
add r10
add r12
mov r10
clr

comp r11
set 0
add r10
mov r10
clr

// on the way back up to mainloop
set done_if
j
goUp3:
set mainloop
j

done_if: // let r12 be the index
// y_temp >>> i
sti r1, 0
set 0
add r2

sti r1,1
mov r12

set 5
mov r13
set ltfive1
blt r12, r13

// i >= 5
loop1:
set 1
mov r13
sr r10, r13
sro r11, r13
clr
comp r13
set 0
add r12
add r13
mov r12
clr

sti r13, 0
set loop1
blt r13, r12
set done1
j

// i < 5
ltfive1:
sr r10, r12
sro r11, r12
clr

done1:
// x_new = ......
set 0
add r11
sti r1, 0
add r4
mov r4
clr
sti r1, 1
set 0
add r10
clr
sti r1, 0
add r3
mov r3
clr

// x_temp >> i
// store i in r12
set 0
add r2
sti r1, 1
```

```
mov r12
sti r1, 0

set 5
mov r7
set ltfive2
blt r2, r7

// Going up to the mainloop
set loop2
j
goUp2:
set goUp3
j

// i >= 5
loop2:
set 1
mov r7
sr r5, r7
sro r6, r7
clr

// decrement
comp r7
set 0
add r2
add r7
mov r2
clr

sti r7, 0
set loop2
blt r7, r2
set done2
j

// i < 5
ltfive2:
sr r5, r2
sro r6, r2
clr
```

```
done2:
// y_new = ......
set 0
add r6
sti r1, 1
add r9
mov r9
clr
sti r1, 0
set 0
add r5
clr
sti r1, 1
add r8
mov r8
clr

// t_new = ...
// put index back inro r2
set 0
add r12
sti r1, 0
mov r2

// load in T to Y_Temp
sti r1, 2
set 0
add r14
sti r1,1
mov r10
sti r1,2
set 0
add r15
sti r1, 1
mov r11

set 8
mov r13
set if
blt r12, r13
// i >= 8
set 16
mov r13
```

```
// on the way up to the main loop label...
set after1
j
goUp1:
set goUp2
j

after1:
set 11
comp r12
clr
add r12

sl r13, r0
comp r13
clr

// addition
set 1
mov r12
comp r12

set 0
add r11
add r13
mov r11
set 0
add r12
clr
add r10
mov r10
clr

set end_if1
j

if: // i < 8
set 1
mov r13

set 7
comp r12
clr
add r12

sl r13, r0
comp r13
clr

// addition
set 0
add r13
add r10
mov r10
clr

end_if1:
// put t back
sti r1, 2
mov r14
sti r1, 1
set 0
add r11
sti r1, 2
mov r15

// fill y_temp
sti r1,1
set 0
add r8
mov r10
set 0
add r9
mov r11

// fill x_temp
sti r1,0
set 0
add r3
mov r5
set 0
add r4
mov r6

// branch back
```

```
set 14
mov r7
set goUp1
blt r4, r7

// load values into memory

// r = x_temp
set 5
sw r5
set 6
sw r6

// t = t_temp
sti r1, 2
set 7
sw r14
set 8
sw r15

halt
```

String Pattern Code:

```
// mem 0x20 - 0x5F words
// mem 0x09 pattern
// mem 0x0A count

// Registers initially 0
// ov - overflow
// r0 - implicit accumulator reg

// BLOCK 0
// r1 - implicit reg index
// r2 - pattern
// r3 - word
// r4 - temp reg to compare to pattern
// r5 - count

// BLOCK 1
// r8 - address of next word
// r9 - index i
// r10 - index j
// others - temp

set 9 // load in pattern
lw r2
set 32 // load in 1st word
lw r3

sti r1, 1
mov r8

st1, r1, 0
// 1st iteration
// load 4 MSB into temp reg
set 4
sl r2, r0
set 0
add r4
mov r4
clr

// count++ if match
set inc
beq r2 r4

set else
j

inc:
set 1
add r5
mov r5

// j = 4, continue matching pattern in
innerloop for 1st word
else:
sti r1, 1
set 4
mov r10
set innerloop
j

outerloop:
// load in next word
sti r1, 1
set 8
add r8, r0
mov r8
sti r1, 0
lw r3

////

// do all compares for 1 word
innerloop:
// sl temp and word by 1
sti r1, 0
set 1
sl r4, r0
sl r3, r0

// add MSB word to LSB temp
add r4
mov r4

// clear leftmost 4 bits of temp
```

```
set 4
sl r4, r0
clr
sro r4, r0

// compare the temp and pattern
set inc2
beq r2, r4

endinner:
// j++
sti r1,1
set 1
add r10, r0
mov r10

// loop if j < 8
set 8
mov r11
set innerloop
blt r10, r11
set end
j

inc2:
// count++
set 1
add r5, r0
mov r5
clr
set endinner
j

end:
// i++
set 1
add r9
mov r9

// i < 63, outerloop again
set 63
mov r11
set outerloop
blt r9 r11

sti r1, 0
set 10
sw r5
halt
```

Signed Integer Division Code:

```
// Signed Integer Division

// mem 0x7E - 0x7F quotient
// mem 0x80 - 0x81 dividend
// mem 0x82 divisor

// Registers initially 0
// ov - overflow
// r0 - implicit accumulator reg
// r1 - Implicit register index

// BLOCK 0
// r2 - dividend MSW
// r3 - divident LSW
// r4 - divisor
// r5 - temp
// r6 - temp MSW
// r7 - temp LSW

// BLOCK 1
// r8 - loop index
// r9 - quotient MSW
// r10 - quotient LSW

set 7
mov r7

set 1
sl r0, r7
lw r2

mov r5
clr
set 1
mov r7
add r5
lw r3

add r7
lw r4

set 0
mov r7

loop:

// shift div left by 1
set 1
mov r5
sl r6, r5
sl r7, r5
set 0
add r6
mov r6
clr

// div << 1, dividend << 1, div +
MSB_dividend
sl r2, r5
set 0
add r7
mov r7
sl r3, r5
set 0
add r2
mov r2
clr

// Check if MSW of div == 0
set 0
mov r5
set if
beq r6, r5

// Check if divisor < div LSW
set else
blt r4, r7

if:
// div = div - divisor

// sign extend  -divisor
comp r4
set 2
mov r5
```

```
set 63
sl r0, r5
sti r5, 3
add r5
mov r5

set cont
j
goup:
set loop
j

cont:
// div - divisor
set 0
add r4
add r7
mov r7
set 0
add r6
add r5
mov r6
clr
comp r4

// quotient << 1, add 1
set 1
mov r1
sl r9, r0
sl r10, r0
set 0
add r9
mov r9
clr
set 1
add r10
mov r10
clr
j endif

else:
set 1
mov r1
sl r9, r0
sl r10, r0
set 0
add r9
mov r9
clr

endif:
// increment index
set 1
add r8
mov r8
set 16
mov r11
set goup

// Check index
blt r8, r11

// store quotient
clr
set 63
add r0
sw r9
add r1
sw r10

halt
```