CE 12 Lab Report 5

PIC32 Function Calls and IO

Brandon Gomez

Lab Section: 2

11.26.14

Procedure:
The purpose of this lab is to get familiar with the PIC32 function calls and the IO using MIPS. Our program is going to print out, “hello world! #” where “#” is an incremented number starting from 0. The LEDs on the chips will blink accordingly to the print procedure. We will then slow or speed up the print procedure.

Materials:
Hardware:
- PICkit3
- Chipkit Uno32
- Chipkit Basic IO Shield
- Two USB cables (one for PICkit3 and one for Uno32)

Software:
- MPLAB (to code)
- CoolTerm (to see output)

Instructions:
- add: $d = $s + $t
- addiu: $t = $s + SE(i)
- addi: $t = $s & ZE(i)
- and: $d = $s & $t
- sw: MEM [$s + i]:4 = $t
- lw: $t = MEM [$s + i]:4
- la: ui $rd, LabelAddr[31:16]; ori $rd,$rd, LabelAddr[15:0]
- jal: $31 = pc; pc += i << 2
- nop: Do nothing
- move: add $rt,$rs,$zero
- beq: if ($s == $t) pc += i << 2
- bne: if ($s != $t) pc += i << 2

Debugging:
Here is what is done inside the c printf statement:
- JAL print_cdnopuxX // contains the address 9D00_2484
- JAL vprintf_cdnopuxX // contains the sddress 9D00_0000
- JAL flsbuf // contains the address 9D00_1BEC
- JAL write // contains the address 9D00_2164
- JAL mon_putc // contains the address 9D00_20B0
- JR RA // exists mon_putc contains 9D00_21EA
- JR RA // exits write contains 9D00_1C98
- JR RA // exits flsbuf contains 9D00_1AA4
- JR RA // exits vprintf_cdnopuxX
- … (repeat for each letter)
- JR RA // exits print_cdnopuxX

The printf function goes five levels deep also there are five returns in order to get out of printf.

Subroutines in MIPS:

Once we call our subroutine inside the c program, we will want to print out “hello, world! #” and return 0x80000 for the delay loop. We are going to use a stack. We are then going to store the message address inside a few arguments, we are going to store inside the counter inside S4, and every time the program prints, S4 will increment. We then call printf to print the message.

```
/*
    push into stack
    store word
    load address of hello world
    incremenet counter then print
    pop out of stack
*/

push: addi $sp, $sp, -4
sw $ra, 0($sp)

la $a0, myStr
jal puts
nop

move $a2, $zero
addiu $s4, $s4, 1
move $a1, $s4
la $a0, myStr2
nop
jal printf

pop: lw $ra, 0($sp)
addi $sp, $sp, 4
```

IO in MIPS:

```
/* s0 contains the input */

lw $s0, PORTD
```

In order to get the data of the switches we are going to use PORTD. For example, if all the switches are on then the result will be 0xF10, and if all are off then the result will be 0x110. Once the program gets the result of PORTD, the program will print the message and v0 will contain the returning value for delay. The delay has to correspond to the PORTD value so I used 16 branches to compare the results of PORTD to all the switch values and if it matches, the program will add values to v0 to return to delay. I computed the delay values by guessing and checking. If all the switches are on the delay equals 16,384,000, which is about 15 seconds. I then deducted until I was able to reach my desired delay.

```
/*
   store PORT values into t
   and compare it to s0 then put
   sec values inside v0
*/

and $t0, $t0, 0
addi $t0, $t0, 0x110
beq $s0, $t0, doit1
bne $s0, $t0, skip1
doit1:
addi $v0, $v0, 24000
add $v0, $v0, $v0
add $v0, $v0, $v0
add $v0, $v0, $v0
add $v0, $v0, $v0
add $v0, $v0, $v0
skip1:
```

Results:
The results of the program depend on the amount of switches that are on. If all the switches are off then the program will print "Hello World #" as fast as possible and the LEDs are all on with none of them moving. If SW1 is on the others are off then it will take 1 second for all the LEDs the light up and the message will appear accordingly to the LEDs. If all the LEDs are on then one LED will be moving roughly every 2 seconds and the message will appear accordingly.

Conclusion:
In conclusion, it was quite interesting to see the different types of machine code from LC-3 and MIPS. MIPS are quite complex with the bigger values and more operation codes. However, there were pseudo operations that made life a little bit easier like la. It was also quite interesting how the c code can translate to assembly code. It is making me see code and pointers quite differently. LI is a pseudo instruction that actualy translates to lui $8, 0x0003; ori $8, $8, 0xBF2. Mask = mask << 1gets complied to: LW V0, 16(S8); SLL V0, V0, 1; SW V0, 16(S8). The second LED is one before the operation PORTECLR = 0x00FF. Mask is store inside v0 because this is the value of the expression results.