CPE 315-01 Computer Architecture

Lab 4

Submitted 2/9/2018

Samuel Sachnoff and Michael Silin

Professor David Retz

Cal Poly San Luis Obispo

1. Questions

1a) **Q: The designers of the MIPS instruction set did not make push and pop basic instructions, yet they are frequently used. How might a push or pop be implemented as a pseudoinstruction? Explain each.**

The push and pop commands could be simply implemented using just the $ra as the only argument. So it would look like

push $ra

addi $sp, $sp, -4# $sp—

sw $ra, 0($sp) # push entry on stack

pop $ra

lw $ra, 0($sp) # pop enry from stack

addi $sp, $sp, 4 # advance stack pointer

If we used a pseudo instruction, we wouldn’t need to include any reference to the stack pointer and we could just let the machine handle it.

1b) **Q: If push and pop were to be included in the instruction set, what format would they take? Suggest the use of machine registers, instruction formation, and describe the operation.**

If we included the push/pop command in our instruction we would use an immediate command.

Our instructions would look like this : push Rd, pop Rd. An offset could be used if we wanted to perform a more advanced command. We could still use the stack pointer register. The function would only need one argument.

**• Introduction**

In this lab we learned how to use Frame Pointers and Stack Pointers to handle our return addresses, function parameters, and return values. We also practiced with a recursive fibonacci program, and a binary to hex translator.

**• Functional Requirements**

We accomplished this by creating 5 different programs to handle each individual question. We reused our bin2hex function from part 2 in order to check our results for part 4. In part 3 we had a brief introduction to stack pointers and return addresses, while part 6 required us to implement frame pointers.

**• Approach**

Part 2: For our Binary to Hex function, we used a different approach when compared to the instructor solution. Some of our solution did share the same methods as the instructor solution. For example, we split our number up into 8 hex digits, and we used a counter to iterate through each digit. To convert each 4 bits to one hex digit, we used a conditional add code to add x30 to each decimal number if a 1-9 was detected. If the binary number was 10-15, we added x39 in order to skip the ascii characters in between 1-9 and A-F

Part 3: This was a simple recursive program we wrote. We used t0-t2 for temporary storage, and we pushed our $ra onto the stack in order to recursively call the function. We kept track of the total number of recursive functions to call with a counter.

Part 4: For this function, we followed the instructions and we used an addu command to add the low bits. We then followed that with two separate setl commands, in order to detect if the result was less than the original number. If it was, then a carry occured. We ORed the two carry detection instructions in order to create our final carry detect result. We then added this to the result of the unsigned add of the two high bit numbers. Finally , we printed both the high result and the low result using our bin to hex function.

Part 5: For this code, we used a temporary register to capture the bits from the first register that were pushed off after shifting twice. We then shifted the second register by two, and added the temporary register in order to restore the first two bits that were pushed off.

Part 6: We created a simple set of instructions to push the different sets of bits over. In order to conform to the requirements, we stored our stack pointer, along with the frame pointer, and parameters onto our stack. We used $v0 as our result, and $v1 as a condition code (0 if valid input, 1 if invalid input). We checked the user input by using a mask, setting all the bits that were supposed to be 0 to 1 and anding our mask with the user input. If the result was greater than 0, we detected an invalid input.

**• Discussion of any difficulties encountered in the implementation, and information relative to issues of Reliability, Maintainability, and Security.**

The only part we found confusing was for part 4) when adding two 64 bit numbers. We didn’t think the instructions were clear on what to do when there was an overflow or carry from the high order bits. It was fairly easy to implement the carry bit from the low order bits and to move that over to the high order bits.

Also for Part 6, it was sort of challenging to write a test driver program / main in order to run against our program since we weren’t familiar with the syscall condition codes.

**• A summary of what you learned from the lab.**

In this lab we learned how to use the stack in mips. At first we started off by simply using it to call a function recursively. By the end, we were pushing the parameters and return values, along with the previous frame pointer. This gave us a great understanding of how functions actually work.

We also became more familiar with the MIPS instructions, such as syscalls and set less than. This lab also gave us a lot of experience when handling a 64 bit number as two 32 bit numbers. The BIN to Hex helped us learn how to output our MIPS code to the end user, as there doesn’t seem to be a direct way to print binary or hex numbers.

**\*\*\*Source Code \*\*\***

**Please see the attached zip file. Thank you**