CMPE 12 Homework #3

John Allard

November 20th, 2014

1. The following operations are performed on the stack

   `PUSH A, PUSH B, POP, PUSH C, PUSH D, POP, PUSH E, POP, POP, PUSH F`

   (a) What does the stack contain after the `PUSH F` operation? **Answer :** `A, F`

   (b) At which point does the stack contain the most elements?
   Answer : After `PUSH D`, it contains 3 elements.

   Without removing the elements left on the stack from the previous operations, we perform:

   `PUSH G, PUSH H, PUSH I, PUSH J, POP, PUSH K, POP, POP, POP`
   `PUSH L, POP, POP, PUSH M`

   (c) What does the stack contain now?
   Answer : `A, L, M`

2. What are some disadvantages in programming in a higher level language?
   Answer : First and formost, you don't have as much control over the operation of the computer. Higher level languages most of the time don't allow you to access memory locations directly. Also, higher level languages consist of operations that wrap around the opcodes for a given computers ISA, which means that you aren't directly able to call certain operations on a computer, you have to use the interface for those operations provided by a given language. This can restrict the user from using optimizations specific to certain architectures. Generally using a higher level language will compromise performance because of all of the extra computation that is going on in the background (garbage collection, reference counting, etc.).

3. Compare and contrast the execution process of an interpreter versus the execution process of compiled binary. What implications does interpretation have on performance?
   Answer : An interpreter will take in a given source file, and break it up into a finite sequence of discrete chunks (lines, paragraphs, functions, etc). It will then read in each 'chunk' of data, and interpret this chunk in a way to produce the desired results by the user. It then moves on to the next 'chunk' of code, and repreats the same process. This is how a virtual machine acts. A compiler will take an entire source file, and translate it directly into a computers native language (it's ISA). Once this is done, we will then have a given executable file that can be run directly on the computers hardware without any further translation needed. Using an interpreter means that each 'chunk' of commands will have to processed and translated into machine code during each run of the program. This means if you want to run a program many times it could be slower than using a compiler which will yield a single file of instructions in the computers native language.

4. For this question refer to Figure 11.2 (Page 295 in the textbook).

   (a) Describe the input to the C preprocessor. **Answer :** The input the C preproceser are the source and header files for a given C project. This means the raw source files as the user tpyed them in their editor.

   (b) Describe the input to the C compiler. **Answer :** The compiler obtains the output of the C preprocessor. This means it gets almost the same text as the raw source files, except any of the preprocessor directives (`#define`, `#pragma`, etc.) have been converted into actual C commands that can be compiled.

   (c) Describe the input the linker. **Answer :** The linker recieves the output of the C compiler, which has turned nearly-raw C source code into individual object modules. The modules represent bits of a machine code for single sections of a program. These individual object modules are linked with object files from other libraries and from code that comes with the operating system to produce an executable program.

5. What happens if we change the second-to-last line of the program in Figure 11.3 from `printf("%d\n", counter);` to:

   (a) `printf("%c\n", counter + 'A');`
   Answer : This code would take counter, and add to it the value for the ASCII character A (65). This would give us counter'th letter in the alphabet if counter is less than 26 and greater than zero, then output this to the screen. Example, counter = 2, then this would print the letter B to the screen.

   (b) `printf("%d\n%d\n", counter, startPoint + counter);`
   Answer : This would output to integers next to eachother. The first one will be the numbers counting down from counter to zero. The second number will be counting down from 2 times startpoint to startpoint before ending.

   (c) `printf("%x\n", counter);`
   Answer : This would print out the hexidecimal numbers starting from counter to 0.

6. The `scanf` reads in a character from the keyboard and the function `printf` prints it out. What do the following two statements accomplish?

   (a) `scanf("%c", nextChar");` **Answer :** This code reads in a single character from the keyboard and saves it into the nextChar variable.

   (b) `printf("%d\n", &nextChar");` **Answer :** This takes the char from the nextChar variable and outputs it's ASCII value.

7. The following lines of C code appear in a program. What will be the output of each printf statement?

```
#define LETTER '1'
#define ZERO 0
#define NUMBER 123
printf"%c", 'a');
printf("x%x", 12288);
printf("$%d.%c%d\n", NUMBER LETTER, ZERO);
```

   Answer :

```
a
x3000
$123.10
```

8. a) The following code is intended to jump to k0+4112, but it doesnt. [1 pts]

```
addiu k0,k0,4112
```

```
jr k0
addiu k0,k0,-4112
```

(a) Why not? **Answer :** The k_n registers are only for use by the Operating System, so a user is not allowed to make a call like this. More importantly, a `jr` routine always executes the next command, which means that we would be setting k0 back to it's original value before jumping there.

(b) How can we fix the problem and ensure that the jump is taken and the second addiu is not performed? We could simply insert a `NOP` command on the line after the `jr` command.

9. The unconditional jump instruction (J) performs PC = PC[31:28] offset<< 2. How can you jump if you want to modify PC[31:28] as well? [1 pts]

10. How is the stack on the PIC32 organized (where is it and which way does it grow)? [1 pt]

11. How do you...[2 pts]

    (a) PUSH register 16 onto the stack?

    (b) POP register 16 from the stack

12. . What is the purpose of the $fp register? **Answer :** The frame pointer always points to the base of the stack frame and stays at that position for the duration of the subroutine call. This allows the stack to grow and shrink inside of a subroutine while still knowing what was on there when the routine was started.

13. What is the difference between register $v0 and $r2? **Answer :** They refer to the same register, just $v0 is the symbolic name for register 2. Using symbolic names helps make the programmers life easier but does not change much of anything in the final code.

14. How are branches in PIC32 different than branches in LC-3? **Answer :** For one, the instruction after the branch routine is always read in in the PIC32 system. If you use the branch likely operation, this next instruction is only executed if the branch operation succeeds. The LC3 would not implement the instruction after a branch command unless the branch failed or the program returned from the branch some how. Also, one can branch to a much further location on the PIC32 system than on the LC3. Lastly, there are more cases to branch on in the MIPS ISA that runs on PIC32, such as Branch on Equal, Branch and Link, and others. LC3 just let you branch based on any of the 3 binary flags available to the user (negative, zero, and positive).

15. . Describe the operation of each instruction: [2 pts]

    (a) eret **Answer :** Error Return, sets the PC equal to the value in $14. Used by exception handlers to return control of the program.

    (b) bne at,r0,0x9d0000c0 **Answer :** This branches if the values held in the two registers, at and r0 are not equal. If so, it will increment the PC by $0x9d0000c0 << 2$.

    (c) lui t1,0x0 **Answer :** The immediate value (0x0) is left shifted by sixteen bits and stored in the register t1. The lower 16 bits are all zeroes.

    (d) sw t1,0(t2) **Answer :** Store the value in register t1 into the address that is 0 words passed the address help in register t2.

16. In PIC32, how do you: [2 pts]

    (a) Add an signed immediate value of -2 to value in register 1. **Answer :** `addi r1, r1, -2`

    (b) Enable interrupts.

    (c) Branch forward 5 instructions if register 1 is less than register 0. `BLT r1 0x0005`

(d) Store the unsigned byte 7 to a memory location that is 5 instructions beyond register 7.
Answer : `addi \$t1 \$t1 0x0007`
`sb \$t1, 5(\$r7)`