Jennings Fairchild

Nicholas Warner

4 Nov 14

CSCE 312-503

**Lab 5**

**Problem 1:**

1. See prob1.ys

|  |  |
| --- | --- |
| int i,j;  …..  if (i >j) {  i= i+5;  }  else {  i=0;  j++;  } | Main:  # set up int i and int j, per our assumption that i and j were initialized already in the c code fragment somehow  irmovl $4, %eax # put 4 into eax (int i)  irmovl $6, %ecx # put 6 into ecx (int j)  irmovl $1, %edx # put 1 in edx, for incrementing (a constant, basically)  irmovl $5, %ebx # put 5 into ebx, to add in the else block  rrmovl %eax, %esi # make a copy of i into esi (temp)  subl %ecx, %esi # is 4 > 6? will set sign bit to positive if true  jg IF # if 4 > 6, then go to IF, else...  irmovl $0, %eax # replace i's value with 0  addl %edx, %ecx # increment j by 1  pushl %eax # push i onto stack  pushl %ecx # push j onto stack  jmp End # end our control structure  IF:  addl %ebx, %eax # add 5 to int i (%eax)  jmp End # end our control structure  #move on to other operations, but for this fragment, halt  End:  halt |

1. See prob1.ys and prob1.yo
2. We had to assume, at a minimum, that ints i and j were initialized somehow before the if-else control structure was reached. I just chose values in my init: for the purpose of compilation

**Problem 2:**

1. See prob2.ys

|  |  |
| --- | --- |
| int j,k;  …..  for (int i=0; i <5; i++) {  j = i\*2;  k = j+1;  } | Main:  # set up int i and int j, per our assumption that i and j were initialized already in the c code fragment somehow  irmovl $0, %eax # put 0 into eax (int i)  irmovl $0, %ecx # put 0 into ecx (int j)  irmovl $0, %edx # put 0 into edx (int k)  call Loop # go into for loop  halt  Loop:  irmovl $5, %edi # put 5 into edi for condition testing every loop  rrmovl %eax, %esi # put eax (int i) into esi (temp)  addl %eax, %esi # add temp (esi, int i) and int i (eax), store into temp (esi)  rrmovl %esi, %ecx # copy temp (esi) into int j (ecx)  irmovl $1, %esi # change temp (esi) to 1  addl %ecx, %esi # add 1 (temp, esi) to int j (ecx), store in temp (esi)  rrmovl %esi, %edx # copy temp (esi) into k (edx)  irmovl $1, %esi # change temp (esi) back to 1 to prepare to increment  addl %esi, %eax # ++i  rrmovl %eax, %ebx # copy i to ebx to test it against 5  subl %edi, %ebx # copy of i - 5 (edi), store in edi. sign bit will be 1 for i = 1...4  jl Loop # loop so long as i < 5  ret # return when i >= 5 |

1. We assume that j and k were initialized somehow before we got into the loop. For my testing, I set them to 0 and ran the loop.

**Problem 3:**

1. Commands:
   1. gcc –S lab5\_prob3\_1.c –o lab5\_prob3\_1.s
   2. gcc –S lab5\_prob3\_2.c –o lab5\_prob3\_2.s
2. Commands:
   1. gcc lab5\_prob3\_1.s –o lab5\_prob3\_1
   2. gcc lab5\_prob3\_2.s –o lab5\_prob3\_2
3. Assembler with the required descriptive comments included for both files.

**Problem 4:**

1. Commands:
   1. gcc –S lab5\_prob4\_main.c –o lab5\_prob4\_main.s
2. Explanations detailed in the assembler file lab5\_prob4\_main.s where the code is different (i.e. to support the function call for print\_hello() ).

**Problem 5:**

1. Commands:
   1. gcc –S lab5\_prob5\_main.c –o lab5\_prob5\_main.s
   2. gcc –S lab5\_prob5\_print.c –o lab5\_prob5\_print.s
   3. gcc lab5\_prob5\_main.s lab5\_prob5\_print.s –o lab5\_prob5
2. See associated assembly codes for noted differences. Note: comments are only added where differences are apparent, and explanations are given at the head of each file. A good bit of code is the same as problems 3 and 4.

**Problem 6:**

1. See lab\_prob6.ys
2. The only worry I had for register use was accidently overriding the data I had in one. So, keeping track of how values moved between functions became imperative. One trick I discovered a few problems back that I implemented here was updating the register with the variable I was using for comparison every loop cycle so I could use it in each cycle for something else. With the 64 being the delimiter, I was able to use that register for temporary integer operation work and then load 64 every cycle right before I needed it. Using techniques like that, you can typically use just registers for functions instead of going to the stack, leaving it for more permanent things like important variables and return addresses.

**Problem 7:**

1. The Intel Core i7 processors as well as the AMD Phenom II processors all have Time Stamp Counters which uses the EDX:EAX register.
2. IA-32, PDP-11, PowerPC
3. See prob7.ys
   1. The clock counter has an overflow, so as soon as it hits that overload point, it will reset to 0, so if the execution is that large, it will not accurately count it since it will overload.
   2. An alternate method we were able to find through research was using a low frequency timer that periodically interrupts the processor. The spacing between these interrupt signals is called the interval time and can last between 1 and 10 milliseconds. When a timer interrupt occurs, the operating system scheduler can choose to either resume the currently executing process or to switch to a different process.
4. See prob7\_5.ys
5. Counting clock ticks across multiple cores could become unsynchronized where one core is ahead of the other. There is also the limitation of faster overflow since there are four cores being timed as opposed to just one.