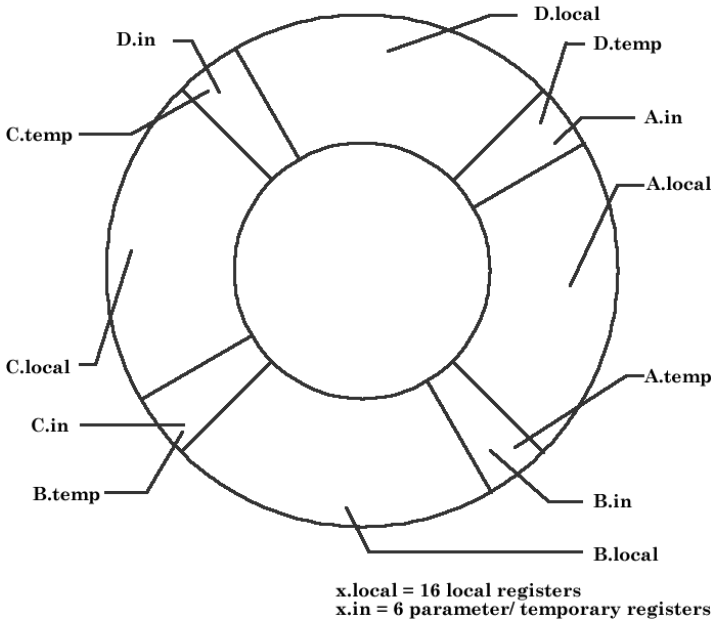
Fall 2006 CSCI 4717 – Computer Architecture
RISC Homework

Due: Tuesday, December 5, 2006

1.) Assume that a RISC processor uses register windows configured like that shown in the figure to the right. To what depth can a process call procedures before having to push a window of registers to the stack, i.e., how many function calls can be present at one time? (Assume that the process in window A is at a depth of 1.)

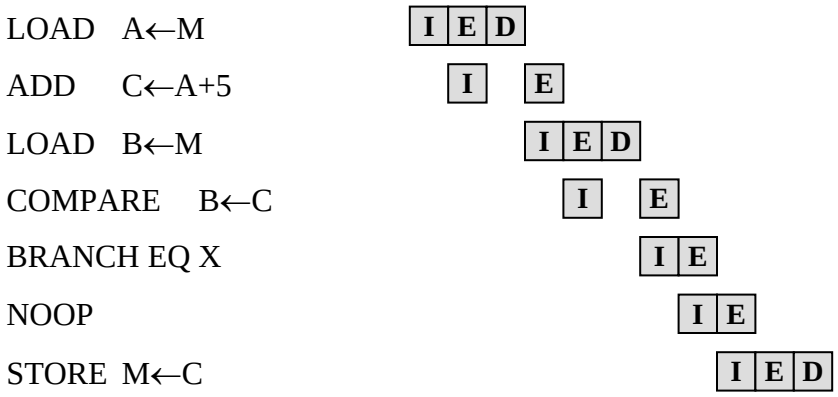


Since each procedure loaded into a register window must have its input parameters (x.in), its local parameters (x.local), and the temporary registers to pass as parameters to the next procedure (x.temp), there must be at least one empty window between the oldest and the newest procedure. Since there are 4 windows, then only 3 can be used at a time. ***Therefore, the answer is 3***.

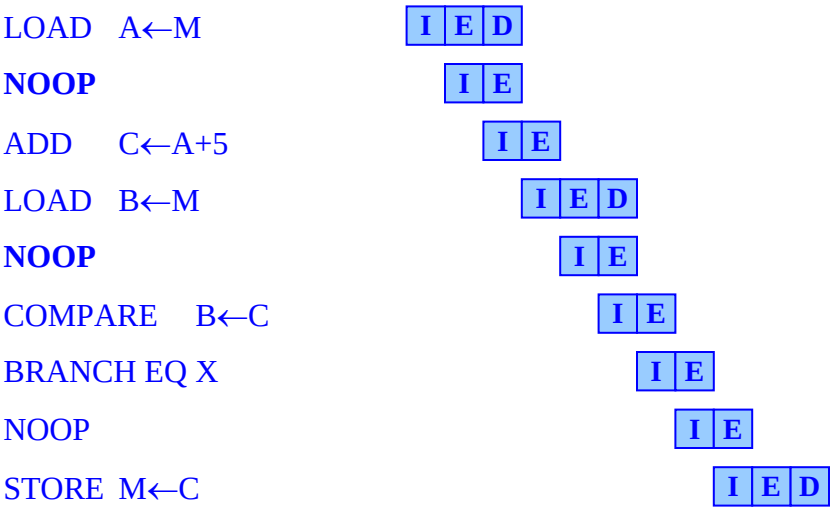
2.) For the figure in problem 1, if a process is called that requires pushing a window to the stack, how many registers must be pushed?

When a register window needs to be freed up, and the procedure that is currently in that window must be pushed to the stack, then the input parameters (x.in) and the local parameters (x.local) must be pushed. The temporary registers used to pass as parameters to the next procedure (x.temp) do not need to be pushed as they are the same thing as the lower procedures input parameters (x+1.in). ***Therefore, 16 + 6 = 22 registers must be pushed.***

3.) Below is the timing diagram for the execution of 5 instructions on a RISC processor with 2-way pipelined timing, i.e., I and E stages are executed simultaneously, but bus limitations require D to be executed alone. How many cycles would be saved if a dual-port RAM is used permitting two memory accesses at one time?

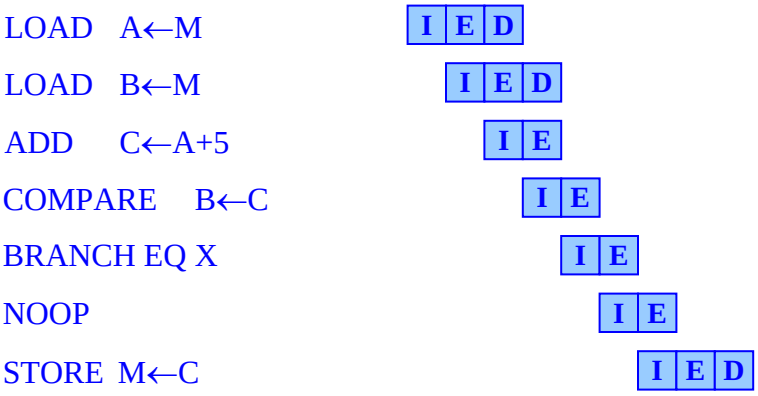


Before we can remove the delays inserted between the I and E cycles of the instructions following memory operations, we need to examine the code to see if any NOOPs need to be inserted for delayed loads or delayed branches. In other words, the delays between the I and the E cycles allow the processor time to load (which is a slower operation) before using the value loaded. It turns out that NOOPs will be needed after both the LOAD A←M and the LOAD B←M operations because the loaded values are used immediately after the loads. This gives us the timing diagram shown on the next page.



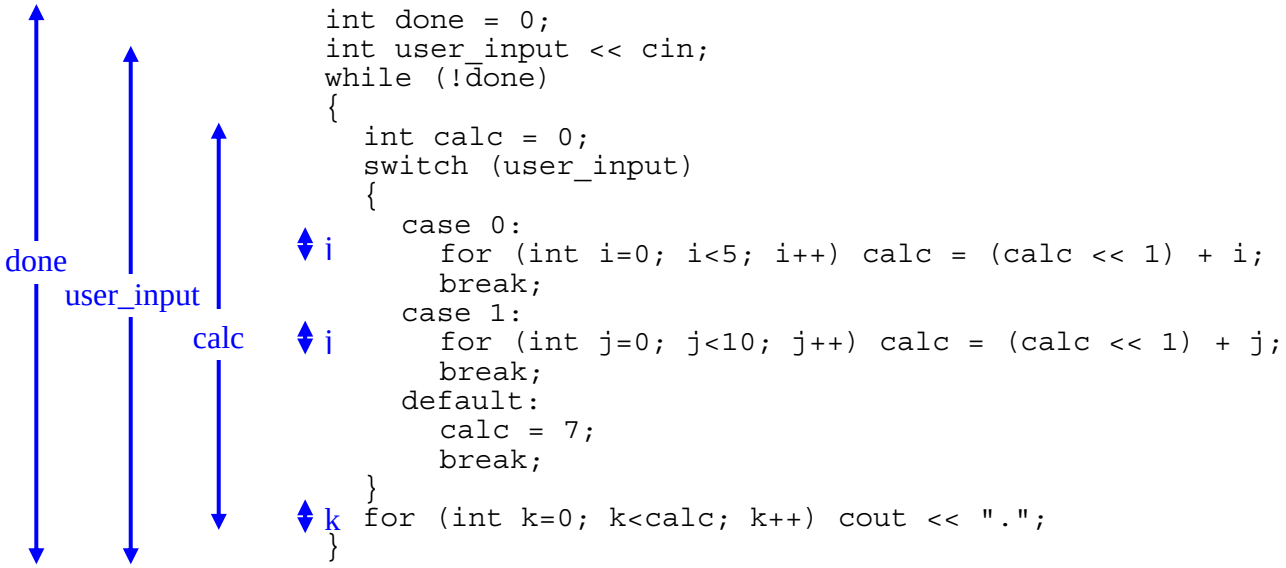
This means that by using a dual port memory, this segment of code takes 11 cycles, which is unfortunately exactly how many cycles the code without dual port memory takes. ***So no cycles are saved.***

Alternatively, if you swapped the ADD C←A+5 and the LOAD B←M functions, you wouldn't need the NOOPs after the LOAD A←M and LOAD B←M instructions. In this case, 2 cycles would be saved.



As for the answer, if you addressed the need for NOOPS/delayed loads, I gave you full credit. (You didn't need to see the instruction swapping trick.) If you simply collapsed all of the delays which could be done with a dual port memory, but not with the write-read dependency, then I took off a point.

4.) Find the absolute minimum number of registers required to execute the code below. (Hint: Think like an assembly language programmer trying to minimize the number of registers needed for the code.)




```
int done = 0;
int user_input << cin;
while (!done)
{
  int calc = 0;
  switch (user_input)
  {
    case 0:
      for (int i=0; i<5; i++) calc = (calc << 1) + i;
      break;
    case 1:
      for (int j=0; j<10; j++) calc = (calc << 1) + j;
      break;
    default:
      calc = 7;
      break;
  }
  for (int k=0; k<calc; k++) cout << ".";
}
```

The arrows identify the periods during which each of the variables is active. By aligning arrows that do not overlap in time, we can see that a minimum of 4 columns of arrows can be created. ***This means that a minimum of 4 registers will be needed.***