CSCI 350
Chapter 4 CPU Design
Comments for Register Transfer Examples

These examples will show what happens at various steps in the instruction cycle

FETCH: This is not an instruction but rather what is to be done at the start of the "fetch" phase of the instruction cycle. At the time when we perform this, which we are going to be calling time T0 of the instruction cycle, the op-code for the instruction that is to be executed had already been retrieved from memory and is in the MBR. The PC register contains the address in memory where this instruction was stored. What this operation requires us to do is to increment the value of the PC and to issue a ***fetch*** signal. What will happen as a result is that during the clock cycle after T0, which we'll call T1, the memory address for the op-code of the presumed next instruction will be received by memory, the memory byte at this address will be accessed, its value placed on the memory bus, and before clock cycle T2 commences, this value will have been placed in the MBR.

- If during clock cycle T1 the instruction being executed does not require the use of the memory bus, or does not require the use of the value being retrieved in the T0 cycle, then its implementation can actually begin during cycle T1. For the implementations of all of the machine language instructions for our architecture this can, in fact, be the case.

POP: As described on page 5 of the notes, we are to "Pop the word on top of the operand stack." Implementing this instruction is simply a matter of decrementing the value of SP by 1, as well as placing this new SP value in the MAR. We will then do a ***read*** operation, and once the ***read*** completes, place the value of the MDR (which holds this value read) into TOS.

IADD: As described on page 4 of the notes, we are to "Pop two words from operand stack and push their sum onto the stack". In actuality we only need to go to memory for the second of the values since by convention the TOS register contains a copy of the value atop the stack. the SP register gives us the address of the top of the stack. We are assuming the operand stack grows in the order of increasing addresses, so the addresses of the two values atop the operand stack are [SP] and [SP-1]. To actually push the value one must place the address -- given by the current value of SP -- into the MAR, place the value to be stored in the MDR, and issue a ***write*** signal.

DUP: As described on page 4 of the notes, we are to "Copy top word on stack and push onto stack". Since we already have this value in the TOS, implementing this instruction is simply a matter of incrementing the value of the SP and placing it in the MAR, placing the value of TOS in MDR and starting a ***write*** operation..

ILOAD: As indicated on page 4 of the notes, this instruction has a parameter ***varnum*** whose value (8 bits, unsigned) is stored in memory immediately after the op-code in the "method area" region of memory (which is byte-addressable). The PC register already contains the address of this parameter and the MBR already contains the value of ***varnum***. Part of implementing this instruction therefore is incrementing PC by 1 and performing a ***fetch*** operation so that when this operation completes, the MBR will contain (presumably) the op-code of the next program instruction. For the rest of the instruction, *varnum* gives the displacement from the current value of LV. One must look in the local variable area of the currently executing program to find the value to be retrieved. Retrieveing it requires adding the value of *varnum* to the value of LV, placing this sum in the MAR register, and initiating ***read*** operation. When the read operation completes that value being loaded will, be in the MDR. While the read is taking place, however, we can then increment the value of SP by 1, place this value in both SP and the MAR. Once the read completes the value in the MDR can then be written onto the stack and must also be copied into the TOS register. In addition, since PC is currently referencing the location containing ***varnum***, we must increment it past this location to that of the op-code of the next instruction to be executed and start a ***fetch*** operation so this op-code goes into the MBR.

GOTO: As indicated on page 4 of the notes, this instruction has as parameter a two-byte value ***offset*** whose value (16 bits, signed) gives the displacement relative to the address of the current instruction where one finds the location of the op-code for the next instruction to be executed.. The PC register already contains the address of this parameter and the MBR already contains the value of ***varnum***. Part of implementing this instruction therefore is incrementing PC by 1 and performing a ***fetch*** operation so that when this operation completes, the MBR will contain (presumably) the op-code of the next program instruction

IFLT: As indicated on page 4 of the notes, this instruction has as parameter that has the same purpose as that for GOTO, however the difference here is that one must first pop the next value from the operand stack and only make the branch if the value (assumed to be 32-bit twos-complement) is negative. The way that we can determine if the value on the operand stack was negative is to run the value of TOS thorough the ALU and check to see if the output signal N has a value of 1, indicating that this value has a 1 in the sign-bit, and hence was that of a negative integer.

INVOKEVIRTUAL: This one is monstrously long and represents all of the operations one must go through to invoke a method, whose the address of whose first instruction is stored is in the constant pool region of memory at a distance ***disp*** (2 bytes, signed) from the current value of CPP. To see everything that is involved here, please read pages 263 - 264 of your textbook. You should have seen the basic idea behind what is going on here in either CSCI220 or CSCI 221.