6.823 Computer Architecture	DUE: 2/27/2002, EXTENSION: 3/4/2002
Timothy Danford, Killian Pohl, Bryan Russell	
Problem Set 1	*Instructors: Krste Asanovic, Srinivas Devadas*

Problem 2

(a) In the DLX pipeline, we need to modify the Execution (EX) and Memory access (MEM) stages. In the EX stage, we need to carry the information from the "A" register to the MEM stage for the PUSH instruction. Hence, there needs to be an extra datapath going from the "A" register (between the pipe stages ID and EX) to the set of registers between the EX and MEM pipe stages. In the MEM pipe stage, we need to add two multiplexors (MUX) and an ALU. The ALU is added so that we can update the stack pointer register $R29$. The output of this ALU is of course put into register $R29$. The two MUXes control the data and address input to the Data Memory. For the data input, we must choose between the output from the ALU in the previous pipe stage and the data in the register from the PUSH instruction. For the address input, we must choose between register "B" from the previous pipe stage, register $R29$ (for the POP instruction), and the output from the added ALU in this pipe stage (for the PUSH instruction). We would not want to add anything beyond this because we have the needed circuitry to do everything else (except handle hazards). We note here that for the POP instruction, R_{DEST} is forwarded through the pipeline and eventually gets back to the Registers for the Write Back stage.

We would not want to allow one to call the pop instruction with the R_{DEST} register. If we allowed this, one could pop the top element and replace the stack pointer with the "popped" contents. In short, we would overwrite the stack pointer and lose track of the stack in memory.

(b) Making the push and pop instructions 11 bits long might not be a good idea because we would have to completely change the design of the memory storing/fetching hardware. In the current implementation of the memory storing/fetching, we are allowed to store/fetch 1 word (32 bits) at a time. With an 11 bit instruction in the current memory implementation, we would have major alignment issues when fetching and storing instructions from/in memory. A better length would be 16 bits since it would allow the instruction to be better-aligned in memory with 2 push/pop instructions per word. Even with this, we have to add additional hardware to implement these fetches/stores which may decrease the clock speed. Also, if we allow the instruction to be 16 bits we still have to be careful with the branch delay and jump instructions. When calculating the address to branch/jump to, we have to take into account these shorter instructions.

(c) We have 22 instructions with each instruction requiring 4 bytes of space in memory. Hence, we would need to fetch 88 bytes of instructions.

(d) Here's the code using push and pop instructions for any stack loads and stores:

```
bar:
        ADDI    sp,sp,#12       # Set the stack pointer to start position
        PUSH    R6              # Push f
        ADD     R6,sp,R0        # Put address of f into R6
        PUSH    R5              # Push e
        ADD     R5,sp,R0        # Put address of e into R5
        PUSH    R4              # Push d
        ADD     R4,sp,R0        # Put address of d into R4
        PUSH    R31             # Push link register
        PUSH    R16             # Save callee-saved register R16
        SUBI    sp,sp,#16       # Adjust stack pointer for subroutine
        ADD     R16,R7,R0       # Put g into R16
        JAL     foo             # Jump and link to foo
        ADDI    sp,sp,#16       # Set stack pointer to last item in stack
        ADD     R6,R16,R0       # Put g into R6
        POP     R16             # Restore callee-saved register R16
        POP     R31             # Restore link register
        POP     R3              # Put d into R3
        POP     R4              # Put e into R4
        POP     R5              # Put f into R4
        ADD     R3,R3,R4        # (d+e+f+g+h)
        ADD     R3,R3,R5
        ADD     R3,R3,R6
        ADD     R2,R2,R3
        SUBI    sp,sp,#12       # Restore stack
        J       R31             # Return to caller
```

For this implementation, we have 15 instructions occupying 4 bytes in memory and 10 instructions (push/pop) occupying 2 bytes in memory. Hence, we need to fetch 80 bytes from memory, which is a slight improvement from the original implementation. From this, we see that using push and pop does not give us much savings in memory.

(e) The problem here is encoding the instructions so that we do not have alignment issues as in problem **2.B**. To encode, let us allow the first 6 bits of the instruction for the op-code and the remaining bits to represent the registers. We can represent the registers in one of two ways. The first way is to encode each register using the standard 5 bit representation as in the ADD instruction. The problem with this is that we can only allow 4 registers to be specified in the register list for the 32 bit representation (6 bits for the op-code, 5 bits for the R_{BASE}, 20 bits for the register list, and 1 bit left over). We can allow more registers to be represented with this representation, but we would have to extend the length of the instruction (of course taking into account the issues raised in problem **2.B**). So, in short we would either have to allow the instruction to vary in size or restrict the number of registers that can be listed in the register list. The second way to encode the registers is the allow the remaining bits to represent each of the registers. To encode the register list, we just need to flip the corresponding bits to 1 (keeping those registers not listed as 0). The problem with this representation is that we only have 26 bits to work with in the 32 bit encoding scheme, which is less than the total number of registers that we have. Again, we could extend the length of the instruction so that we can include all of the registers. Else, we could just choose the registers that can be listed in the instruction.

(f)

```
bar:
        ADDI    sp,sp,#12               # Set the sp to start position
        STMFD   sp,R4,R5,R6,R16,R31     # Store d,e,f,R16, and link register
        ADDI    R6,sp,#8                # Put address of f into R6
        ADDI    R5,sp,#12               # Put address of e into R6
        ADDI    R4,sp,#16               # Put address of d into R6
        SUBI    sp,sp,#16               # Adjust sp for subroutine
        ADD     R16,R7,R0               # Put g into R16
        JAL     foo                     # Jump and link to foo
        ADD     R6,R16,R0               # Put g into R6
        ADDI    sp,sp,#16               # Set sp to last item in stack
        LDMFD   R3,R4,R5,R16,R31,sp     # Restore d,e,f,R16,link reg
        ADD     R3,R3,R4                # (d+e+f+g+h)
        ADD     R3,R3,R5
        ADD     R3,R3,R6
        ADD     R2,R2,R3
        SUBI    sp,sp,#12               # Restore stack
        J       R31                     # Return to caller
```

In this implementation, we have 15 instructions requiring 4 bytes and 2 instructions requiring either 4 bytes (32 bits) or 8 bytes (64 bits). Hence, for the 32 bit encoding, we need to fetch 68 bytes and for the 64 bit encoding, we need to fetch 76 bytes. Compared to the original implementation, this new implementation reduces the size of the instructions by 14% or 23% respectively for the 64 bit and 32 bit encodings. Compared to the push/pop implementation, we only get 5% or 15% savings respectively for the 64 bit and 32 bit encodings. Hence we get quite a bit of savings compared to the original implementation and not so much for the push/pop implementation. This says that with the load and store multiple instructions, we can get varied savings depending on the instruction set that we are comparing it to.