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 3

(a)

Note: R1: a, R2: h, R3: sum, R4: I, R5: N, R6: Condition register, R7:F0: a[I], R8:F1: h[N-1-I], R9:F2: a[I]*h[N-1-I]

Also: I am assuming that the original C code is the following:

```
sum = 0;

for(I = 0; I < N; I++) {
        sum += a[I]*h[N-1-I];
}
```

Now, the DLX code:

```
Foo:
        ADD     R3,R0,R0        # sum = 0;
        ADD     R4,R0,R0        # I = 0;
        SLLI    R6,R5,#2        # Store 4*N...
        ADD     R2,R2,R6        # ...use to set h to start position
        SLT     R6,R4,R5
        BEQZ    R6,End
Loop:
        LW      R7,0(R1)        # Get a[I]
        ADDI    R1,R1,#4        # a += 4;
        SUBI    R2,R2,#4        # h -= 4;
        LW      R8,0(R2)        # Get h[N-1-I]
        MOVI2FP R7,F0           # Store a[I] in F0
        MOVI2FP R8,F1           # Store h[N-1-I] in F1
        MULT    F2,F0,F1        # a[i]*h[N-1-I]
        MOVFP2I F2,R9           # Get result
        ADD     R3,R3,R9        # sum += result
        ADD     R4,R4,#1        # I++;
        SLT     R6,R4,R5
        BNEZ    R6,Loop
End:
```

(b)

This will change with the updated number of instructions, after doing the mulitiplication in the floating point registers.

STARTUP: 6 cycles/6 instructions LOOP: 20 cycle loop/12 instruction

100 loops: 2006 cycles 1206 instructions

(c) Assume that Rt_1, Rt_2, and Ft_1 are temporary registers:

```
MACC Rd, Rs1, Rs2, Im1, Im2 ==

  LW       Rt1, (Rs1)
  LW       Rt2, (Rs2)
  MOVI2FP  Rt1, Ft1
  MOVI2FP  Rt2, Ft2
  MULT     Ft1, Ft1, Ft2
  MOVFP2I  Ft1, Rt1
  ADD      Rd, Rd, Rt1
  ADDI     Rs1, Rs1, Im1
  ADDI     Rs2, Rs2, Im2

REPEAT Rs
<instruction> ==

<instruction>
...
<instruction>  (Repeat Rs times)
```

(In other words, a REPEAT instruction followed by a second instruction repeats the second instruction Rs times).

There are two problems:

Problems: MACC needs three additional temporary registers, Rt1, Rt2, and Ft1, to handle the contents of the memory addresses while avoiding overwriting Rs1 and Rs2 during its update. We can't use existing integer and floating-point registers since doing so would possibly obliterate meaningful values left there by other parts of the program. MACC needs its own memory. REPEAT also needs additional hardware support: it is only "half" of an instruction since the exact sequence of real DLX instructions that it is translated into depends on the instruction that follows it. The machine needs some way of keeping track of how many instructions to generate.

Hardware Solutions: The end result, in terms of hardware, is that three additional registers and a state bit are necessary to implement both MACC and REPEAT. For MACC, the three additional registers (two integer, one floating point) are labeled Rt1, Rt2, and Ft1, and used as in the code above. REPEAT could be implemented in a variety of ways... but one way that it could be implemented would be by storing the contents of Rs in one of the temporary registers that we need for MACC (say, Rt1). We could then set this extra “state bit” (signifying that we're in the middle of a REPEAT) instruction. After fetching the subsequent instruction, we could store its encoding in the second temporary register (Rt2), and emit that value Rt1 number of times.

In other words, the only additional hardware needed for REPEAT beyond what's needed for MACC is the state bit and the necessary data path to implement a loop (decrement and equality check) in hardware for the Rt1 register.

(d)

DSP core code rewritten using MACC only:

```
       ADD R3, R0, R0             ;; sum = 0
       ADD R4, R0, R0             ;; i = 0
       SLLI R5,R5,#2              ;; N = 4*N
       SUBI R7,R5,#4              ;; R7 = h + N-4
       ADD  R7,R7,R2              ;;
       SLT  R6,R4,R5              ;; I < N???
       BEQZ R6,end_loop
loop:
       MACC R3, R4, R7, #4, #-4
       SLT  R6,R4,R5              ;; I < N???
       BNEZ R6,loop
end_loop:                         ;; R3 contains sum...
```

DSP core code rewritten using MACC and REPEAT:

```
ADD R3, R0, R0     ;; sum = 0
ADD R4, R0, R0     ;; i = 0
SLLI R6,R5,#2      ;; R7 = h + N-4
SUBI R6,R6,#4      ;;
ADD  R6,R6,R2      ;;

REPEAT R5
MACC R3, R4, R6, #4, #-4
```

For $N = 100$, in the MACC only case the code takes 307 instructions and 1207 cycles. The MACC and REPEAT case takes 7 instructions and 1907 cycles.

Even if the MACC and REPEAT instructions are implemented with the translation mechanism from 3C, there's a definite performance benefit: MACC and REPEAT are compact encodings of multiple instructions. A single instruction fetch for a MACC instruction actually fetches 9 full instructions. A REPEAT instruction can save Rs number of instruction fetches.

(e) For the program that uses both MACC and REPEAT, a loop with N=100 would take 700 cycles for the MACC instructions and an additional 4 cycles startup (set up the variables, and read the REPEAT instruction).

In this specialized implementation of MACC, the hardware to do the 5-cycle read and add sits idle while the sequential multiply and add instructions (2 cycles) execute. We can improve the performance in the situation where multiple MACC instructions are executed sequenctially (as in the MACC + REPEAT example above) by pipelining the "microcode" (or what is essentially microcode, that is, the actions of the datapath in the execution of the MACC instruction) for the MACC instruction. In essence, while the multiply and add instructions are executing, the next parallel load and add has begun. This means that two cycles of the parallel instructions overlap with the two cycles of sequential instructions, and a MACC instruction is executed every 5 cycles instead of 7. If this was the case, the MACC + REPEAT code would take 506 cycles for N = 100 (5 * 100 in the loop + 4 cycles startup instructions + 2 wasted cycles for the first MACC instruction that isn't pipelined).