ECSE 425
Computer Organization and Architecture

<u>Assignment 3</u>

<u>Due: March 1, 2007 at 4:00 PM in the Trottier assignment box.</u>

- You may make use of reasonable assumptions of your own for data that might be missing in the problem texts, provided that they are explicitly and clearly stated.

- You may submit a partial answer to a problem. The grading will account for this.

- Their main purpose of this assignment is to help you attain an understanding of the course material (the primary goal) and, in addition, to help you prepare for the exams.

Question 1: Code Templates - (20%)

Compilers use libraries of "code templates" to translate individual statements of a high-level language into sequences of machine instructions which can be strung together to form a complete program. The purpose of this exercise is to design such templates to translate simple integer constructs into MIPS-64 assembly.

Register allocation is not performed at the template level. To deal with this, we assume that all scalar integer variables are stored in "pseudo-registers." A pseudo register is used in the assembly code template as if it was a real register, but its final identity is left unspecified. It is the job of the "register allocator" to later associate pseudo-registers with real registers or memory locations.

We use the following notation:

- An integer variable *x* in C is represented in the templates by a pseudo-register called R*x* (i.e., R*x* is the register that holds the value of *x*).
- Temporary integers are represented by numbered pseudo-registers T0, T1, etc.
- Labels are given generic names like L0, L1, etc.
- An unspecified C statement *stmt1* is represented by S1, *stmt2* by S2, etc.
- An unspecified C expression *expr1* is represented by E1, etc.

For example:

```
        if (expr1) stmt1;
        else stmt2;
```

could yield the following template:

```
        beqz E1, L0
        S1
        j    L1
L0:     S2
L1:
```

Now, using MIPS64 instructions, write one template for each of the following C statements. Assume that all variables are 32-bit integers, and that all variables are already loaded into registers. Comment your code.

1. `*a`

2. `a <<= (b == c) // shifts a by one bit if b equals c`

3. `a = expr1 || expr2`

4. `a[expr] = a[expr-1] + b[expr+1];`
   Assume that the base addresses of arrays *a* and *b* are in R*a* and R*b*, respectively. Be efficient with indexing the arrays.

5. `a -= (++a)`

6. `a -= (a++)`

7. `a & 0xAAAABBBBCCCCDDDD // this is a **64-bit** quantity`

8. `do stmt while(expr);`

9. `while (expr) stmt;`

10. `!a`

Notes:

- Use only the MIPS64 instructions listed at the back of the textbook.
- In C, an expression is false if it evaluates to zero; it is true otherwise.
- Be careful with the increment (++) operator. Note how it changes the values of variables.

Question 2: Pipelining and Code Scheduling (20%)

Consider the three functions `foo1()`, `foo2()`, `foo3()` listed below. Notice the different ways they allocate storage for the variables.

```
extern float  glob_A[], glob_B[];
       int    glob_i, glob_c = 0;

foo1()
{
    for (glob_i = 0; glob_i < 100; ++glob_i) {
        if (glob_A[glob_i] > glob_B[glob_i + 1]) {
            glob_c *= glob_i + 1;
        }
    }
    return (glob_c);
}

foo2()
{
    register int rloc_i;

    for (rloc_i = 0; rloc_i < 100; ++rloc_i) {
        if (glob_A[rloc_i] > glob_B[rloc_i + 1]) {
            glob_c *= rloc_i + 1;
        }
    }
    return (glob_c);
}

foo3()
{
    float  loc_A[100], loc_B[100];
    int    loc_c = 0;
    register int rloc_i;

    for (rloc_i = 0; rloc_i < 100; ++rloc_i) {
        if (loc_A[rloc_i] > loc_B[rloc_i + 1]) {
            loc_c *= rloc_i + 1;
        }
    }
    return (loc_c);
}
```

Using a SPARC machine, compile this code with the gnu compiler `gcc` using the `-S` option to get the assembly code output. Compile it a second time with the `-O2` optimizer option and get the assembly code output.

For each version of the foo functions (both unoptimized and optimized), isolate its code in the assembly output: the code of each function starts at the entry point "_foo?" and finishes at the return instruction "ret". Try to understand the assembly code by referring to the SPARC manual (available on WebCT).

(a) In the assembly code for each version of the foo functions, mark the assembly instructions that correspond to the following C statements:

- Initialization of the for loop (`glob_i = 0, rloc_i = 0`)
- Start of the loop iterations
- Evaluation of the loop condition (`glob_i < 100, rloc_i < 100`)
- Evaluation of the if statement (`glob_A[glob_i] > glob_B[glob_i+1]`, …)
- Body of the if statement (`glob_c *= glob_i+1`, …)
- Update of the index (`++glob_i`, …)
- End of the loop iterations

The goal of doing this is to demonstrate that you have understood the assembly code and know how it corresponds to the original C statements. Don't worry that your answer isn't 100% "exact."

(b) For the *unoptimized* code, estimate the total number of memory transactions in each function (recall that memory transactions consist of loads for instructions, loads for data, and stores for data). For your estimation, ignore instructions outside of loop iterations, and assume that the if statement always evaluates to true. Explain the differences in the number of memory transactions among the functions (how are the two arrays and two integers handled differently?).

(c) For the *optimized* code, estimate the total number of memory transactions in each function. Explain the reduction in the number of memory transactions after optimization (give examples). In addition, give examples of where the compiler accounted for the fact that the target CPU is pipelined (watch for delay slots and certain code scheduling techniques).

Question 3: Integer Pipeline (40%)

For all parts of this question, assume that all instruction and data memory accesses are cache hits (i.e., all memory accesses take one clock cycle to complete).

(a) Consider the code sequence

```
DADD  R1, R2, R3
DADD  R3, R4, R5
DADD  R5, R3, R4
DADD  R6, R7, R8
```

This code is executed in the 5-stage MIPS pipeline with forwarding hardware. Show the timing of this code. At the end of the 5th cycle, which registers in the register file will be read, and which registers in the register file will be written?

(b) For the code in (a), what useful data (if any) is being forwarded during cycle 4? What about cycle 5?

Consider:

```
1.  LOOP:   LD    R2, 0(R3)     \\ load number a
2.          SLT   R1, R2, R0    \\ R1 = 1 if a < 0, R1 = 0 otherwise
3.          BEQZ  R1, OUT       \\ exit if a >= 0
4.          NOP                 \\ delay slot
5.          DSUB  R2, R2, R0    \\ negate
6.          SD    R2, 0(R3)     \\ store back a
7.          DADDI R3, R3, #8    \\ increment pointer
8.          J     LOOP          \\ jump back
9.  OUT:    NOP                  \\ delay slot
```

Assume that this loop will iterate many times.

(c) For the above code, show the timing of the first loop iteration for the 5-stage MIPS pipeline *without any forwarding hardware*. Hazard detection hardware still exists so the instructions would know whether they should stall and how long they should stall. Recall that hazard detection is done in the ID stage. Note the following for this pipeline:

- Once a hazard is detected, the instruction currently in the ID stage stalls until the hazard is cleared and then restarts its ID stage.
- Recall that a register can be written and read in the *same* clock cycle (the write is done in the first half of the cycle, and the read is done in the second half of the cycle).
- Conditional branches and jumps use delayed branches with one delay slot.
- The jump instruction computes its target address in the ID stage.

(d) List the pairs of instructions that cause data hazards in part (c), and explain for each pair why the data hazard exists.

(e) Show the timing of the same sequence for the MIPS pipeline *with* forwarding hardware (as before, show the first loop iteration).

(f) Schedule this code to minimize the number of stalls and NOPs. Show the timing of the scheduled code.

(g) Consider this frequent instruction sequence (it can be used to implement "a = b"):

```
LD  T1, d1(I1)
SD  T1, d2(I2)
```

Draw additional forwarding path(s) and multiplexer(s) in Figure A.23 of the textbook so SD does not stall in this instruction sequence.

(h) Consider:

```
DADD R2, R4, R5
DADD R4, R2, R5
LW   R5, 20(R2)
DADD R3, R5, R4
```

Show the timing of this code when it executes on the *R4000* pipeline. Note that this is not the same as the 5-stage MIPS pipeline (a description of the R4000 pipeline is given in the textbook section A.6). What forwarding is done during execution? (Name the instruction pairs and the pipeline stages involved.)

Question 4: FP Pipeline (20%)

Consider:

```
LOOP:  L.D    F0, 0(R2)
       L.D    F4, 0(R3)
       MUL.D  F0, F0, F4
       SUB.D  F2, F0, F2
       DADDI  R2, R2, #8
       DADDI  R3, R3, #8
       DSUB   R5, R4, R2
       BNEZ   R5, LOOP
       NOP
```

The initial value of R4 is R2 + 100. This code is executed on the 5-stage MIPS integer pipeline and the MIPS floating-point pipeline shown in textbook Figure A.31. The pipelines have full forwarding hardware. When write-back contention occurs, the earliest instruction gets priority and other instructions are stalled. Branches have one delay slot.

(a) Show the timing of the first iteration of this instruction sequence. Do not schedule the instructions in the loop.

(b) Schedule this code in order to minimize the number of stalls and NOPs. Show the timing of the scheduled code.