Due: Monday, December 12, 2005, 5 pm.

Guidelines:

(1) All work during the project must be performed by each student independently and reflect his/her own contribution only.

(2) Any help or support which has been received during the project must be cited and explicitly credited.

(3) All tasks have equal weight. The last three problems are for extra credit only.

(4) The tasks must be reported in a project document and submitted in pdf format to yavuz@eng.umd.edu with a copy to rahulr@eng.umd.edu.

(5) All programs and program outputs must be attached to the project in separate ordinary text files.

(6) Late projects will not be accepted.

The goal of this project is to design a floating-point co-processor which can be used to carry out basic arithmetic operations on floating-point numbers in connection with a central processor or on a standalone basis. The operands for the co-processor are placed in a floating-point register file while the instructions to be executed by it are removed from the instruction stream of a central processor unit and buffered into an instruction buffer. The co-processor consists of two major sub-systems as shown below. The number of instructions, $m$, which can be held in the instruction buffer, and the number of registers which can be used to keep operands and results are design parameters which will need to be determined during the project.
Part (1): In the first part of this project, you will design a floating-point arithmetic unit to perform addition, subtraction, multiplication, and division using 32-bit floating-point numbers. As described in the lecture notes, floating-point numbers are stored in the registers of the floating-point arithmetic unit in single precision IEEE-754 format.

Task 1: Describe an algorithm to perform either addition or subtraction on two single precision IEEE-754 floating-point numbers. Your algorithm must explicitly handle exponent and mantissa alignment and post-operation normalization of the result. In particular, the following issues must be resolved carefully:

(a) The result of the operation must be displayed in binary, hexadecimal, decimal and scientific notations as shown below:

Enter u (in 8 hex digits): 58400000
Enter v (in 8 hex digits): 57200000
Type a to add and s to subtract: a
Type b to view operands and sum in binary, h to view it in hexadecimal, d to view it in decimal, s to view it in scientific notation, and a to view it in all: a

In binary (before alignment)
\[ u = 0 \text{ 10110000 100000000000000000000000} \]
\[ v = 0 \text{ 10101110 010000000000000000000000} \]

In binary (after alignment)
\[ u = 0 \text{ 10110000 100000000000000000000000} \]
\[ v = 0 \text{ 10110000 010100000000000000000000} \]
\[ \text{sum} = 0 \text{ 10110000 110100000000000000000000} \]

In hexadecimal
\[ u = \text{58400000} \]
\[ v = \text{57200000} \]
\[ \text{sum} = \text{58680000} \]

In decimal
\[ u = +844424930131968 \]
\[ v = +175921860444160 \]
\[ \text{sum} = +1020346790576128 \]

In scientific notation
\[ u = +1.5 \times 2^{49} \]
\[ v = +1.25 \times 2^{47} \]
\[ \text{sum} = +1.8125 \times 2^{49} \]

Enter u (in 8 hex digits):

(b) The sign of the result must agree with the sign of the operand with the larger magnitude and according to the operation performed.

(c) Hidden bits of both operands must be set to 1 when they are read in and the hidden bit of the result must be set to 1 after normalizing it unless the number is de-normalized. Note that it is possible for the result to exceed 1 or be less than 1. The normalization must handle both cases.

Task 2: Implement your algorithm in C.

Task 3: Give a block diagram of a hardware system whose operation follows the steps as described in the algorithm you developed in Task 1, clearly indicating what each subsystem does in your block diagram.

Task 4: Describe an algorithm to perform either multiplication or division on two single precision IEEE-754 floating numbers. Your algorithm must explicitly handle exponent alignment and post-operation normalization of the result.
Task 5: Implement your algorithm in C.

Task 6: Give a block diagram of a hardware system whose operation follows the steps as described in the algorithm you developed in Task 4, clearly indicating what each subsystem does in your block diagram.

Part (2): In this second part of the project, you will design an instruction dispatcher unit which is supposed to schedule one of 4 instructions to pair of adder/subtractor units and a pair of multiplier/divider units as designed in Part (1) of the project. The instruction dispatcher must operate on the datapath shown below when it is provided with one of the floating-point instructions in the instruction stream of a central processor unit. The floating-point processor must ensure that the instructions are scheduled to the four operational units with as much parallelism as possible but without causing any structural and data hazards between the instructions. In particular, it must be designed as a scoreboard to avoid structural, WAW, RAW, and WAR hazards.

Instructions to be executed are:

LOAD Floating-point register \( F_x \) from address stored in register \( R_y \)

STORE Floating-point register \( F_x \) into address stored in register \( R_y \)

ADD \( F_x, F_y, F_z, \Rightarrow F_x = F_y + F_z \)
SUB $F_x$, $F_y$, $F_z$, \( \iff F_x = F_y - F_z \)

MPY $F_x$, $F_y$, $F_z$, \( \iff F_x = F_y \times F_z \)

DIV $F_x$, $F_y$, $F_z$, \( \iff F_x = F_y / F_z \)

**Task 7:** Describe how you would pipeline each of the four arithmetic operations. Specify the latencies which are needed for each pair of instructions in case of any RAW data dependencies. Your latency figures must be based on how you would pipeline the four arithmetic operations.

**Task 8:** Design a scoreboard algorithm to implement the instruction dispatcher and implement it in C. Your C implementation must be able to recognize all six instructions using the following op-code assignments:

<table>
<thead>
<tr>
<th>op-code</th>
<th>instruction</th>
<th>op-code</th>
<th>instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>LOAD</td>
<td>011</td>
<td>SUB</td>
</tr>
<tr>
<td>001</td>
<td>STORE</td>
<td>100</td>
<td>MPY</td>
</tr>
<tr>
<td>010</td>
<td>ADD</td>
<td>101</td>
<td>DIV</td>
</tr>
</tbody>
</table>

The output from the C program must indicate the state of each operation unit with a flag as well as the state of each instruction during the execution of a program.

**Task 9 (Bonus Problem):** Suppose that the following assumptions hold:

1. The probability for any two instructions which are $x$ instructions apart to have a RAW hazard is \( \frac{1}{2^x} \), $x = 1, 2, \ldots$
2. The probability for any two instructions which are $x$ instructions apart to have a WAW or WAR hazard is \( \frac{2}{3^x} \), $x = 1, 2, \ldots$
3. All instructions are fetched with equal probability.
4. No two instructions can be scheduled for execution unless they do not have a data or structural hazard.

How big an instruction buffer would be needed to ensure that all four units are scheduled for an instruction 90% of the time? Generalize your answer assuming that the number of add/sub units is increased to $s$ while the number of mpy/div units is increased to $t$ and all four assumptions remain the same.

**Task 10 (Bonus Problem):** If the same four assumptions in Task 9 hold and furthermore all registers are equally likely to be used as the destination of an operation, what is the probability that, with an instruction buffer of size $m$ and a floating-point register file of size $k$, at least two instructions will have a WAW or WAR hazard?
**Task 11 (Bonus Problem):** Verify that the estimates of probabilities you computed in Tasks 9 and 10 hold experimentally by running a set of sample programs on this floating-point processor.