Note: This homework assignment is in two parts, Part I and Part II. Part I consists of problems from the textbook (Hennessy/Patterson 3rd ed., or “HP3”) whose solutions appear at the end of the book. These problems are for self-study only, and will not be graded, and therefore should not be submitted as part of your written work. Part II consists of problems that will count towards your grade for this homework.

Part I (not graded):

1. Basic Tomasulo Algorithm. Do HP3 Problem 3.5.
7. Loop Unrolling. Do HP3 Problem 4.7.

Part II (to be graded):

1. (25 points) Tomasulo’s Algorithm/Superscalar/Speculation. Do HP3 Problem 3.6, but use the following code fragment, instead of the DAXPY loop:

```
loop:  L.D F2, 0(R1)
       L.D F0, 0(R4)
       MULT.D F4, F2, F0
       LD.D F6, 0(R2)
       ADD.D F6, F4, F6
       S.D 0(R2), F6
       DADDUI R1, R1, #8
       DADDUI R2, R2, #8
       DADDUI R4, R4, #8
       DSGTUI R3, R1, exit
       BEQZ R3, loop
```

Effectively, the above code fragment computes $Y_i = A_i X_i + Y_i$, where $A$, $X$ and $Y$ are vectors of length 100.

Here are some assumptions/clarifications:

- Initially, R1, R2 and R4 contain the start address of $X$, $Y$ and $A$, respectively.
- You may assume that the three vectors are non-overlapping.
- The instruction mnemonic DSGTUI stands for “set if greater than unsigned immediate.” In particular, DSGTUI R3, R1, exit does the following: set R3 to one if the value in R1 is greater than the immediate constant “exit”; otherwise, set R3 to zero. The next instruction then checks the value of R3 and branches to the beginning of the loop if R3 has been set to zero. Note: “exit” is a constant integer.
Function units are not pipelined in part (a), but are fully pipelined (i.e., can accept a new set of operands every clock cycle) in parts (b) and (c).

The pipeline is single-issue for part (a), but two-issue for parts (b) and (c).

There is no forwarding between function units; results are communicated via the CDB.

The execution stage (EX) does both the effective address computation and the memory access for loads and stores. Thus, there is no explicit memory stage (MEM); instead, the pipeline is: Fetch (IF), Decode (ID), Issue (IS), Execute (EX), and CDB Write. Each of these pipeline stages, except for the EX stage, takes 1 clock cycle.

For the EX stage, assume the latencies of HP3 Figure 3.62. Loads take 1 clock cycle. Note: Do NOT use latencies from Figure 3.63!

There are five load buffer slots and 5 store buffer slots.

Assume that at most one instruction can write the CDB in one clock cycle.

For architectures with reorder buffer, assume at most one instruction can commit per clock cycle.

Follow the rest of the instructions for parts (a), (b) and (c) in the book.

2. (15 points) **Dynamic Scheduling vs. Speculation.** Do HP3 Problem 3.18, but with the following modifications and clarifications. For the first part (i.e., dynamic scheduling), assume the original Tomasulo’s algorithm, without speculation, and present your answer in the same format as the table in Figure 3.33. For the second part, assume Tomasulo’s with reorder buffer, and use the format of the table in Figure 3.34. **Assume that loads and stores require 3 clock cycles to access memory.**

3. (15 points) **Dynamic Branch Prediction.** Do HP3 Problem 3.10, but assume that the 1-bit predictors are initialized to T, the correlation bit is initialized to T, and the value of d cycles as 2, 1, 0, 2, 1, 0, ...

4. (15 points) **Multiple-Issue: VLIW.** Suppose we have a VLIW processor that could issue three memory references, one floating-point operation, and one integer operation or branch in every clock cycle. Assume the following:

- **Load to add latency:** If a L.D instruction is issued in cycle 1, then a dependent ADD.D instruction, which needs the value just loaded, must be issued in cycle 3 or later.
- **Add to store latency:** If an ADD.D instruction is issued in cycle 1, then a dependent S.D instruction, which saves the result to memory, must be issued in cycle 4 or later.
- There is one branch-delay slot.
- An unlimited number of registers are available.

Show an unrolled version of the following loop for such a processor, and schedule it to execute efficiently. Unroll the loop the minimum number of times necessary to eliminate all stalls, but no more than six times. **For full credit, your unrolled code must execute as fast as possible. However, do not perform any software pipelining.**

```
Loop:  L.D  F0, 0(R1)
       L.D  F2, 1000(R1)
       ADD.D F4, F0, F2
       S.D  0(R1), F4
       SUBI R1, R1, #8
       BNEZ R1, Loop
```
How many clock cycles does your unrolled loop require for each iteration of the original loop?

5. (10 points) **Loop-Carried Dependencies.** Consider the following program fragment:

```c
for (i=1; i<100; i=i+1) {
    a[i] = b[i-1] + c[i];
    b[i] = c[i] + d[i];
    b[i+1] = a[i] + e[i];
}
```

(a) List all of the dependencies for the above fragment, and for each dependency indicate its class (i.e., RAW, WAW or WAR) and whether it is loop-carried (i.e., whether it is a dependency between instruction instances across loop iterations).

(b) Rewrite the loop so that it is “parallel,” i.e., rewrite so as to eliminate all loop-carried dependencies. Indicate and classify all of the dependencies in the rewritten loop.

*Hint:* This is an easy question!

6. (20 points) **Software Pipelining.** Do HP3 Problem 4.11, but assume that the latency of the ADD.D instruction is 7 cycles, instead of 5. This problem refers to the example in HP3 on page 330 and/or the example on slide 19 of Lecture 15.

Note that the phrase “latency of the ADD.D is 7 cycles” means the following: if ADD.D is issued at clock cycle 1, then the subsequent consuming S.D must be issued at clock cycle 9 or later. Due to such a long latency, software pipelining is not enough to ensure that dependent instructions are placed far enough apart so as to not require stalls.

In this question, you are to first apply software pipelining to expose loop-level parallelism, and then unroll the resulting loop and schedule it to eliminate stalls. Be sure to show the loop start-up and clean-up code for achieving software pipelining. You may assume that the number of loop iterations is a multiple of a convenient number.