CDA 5155 – Homework #3

Submission:
1. An email submission with a word or PDF document to the instructor (yuw@cs.fsu.edu).

Due Date:
1. Thursday, April 7th, 2016, 11:50pm for all students

Late Penalty:
1. 10% per day (5% if within 3 hours).
2. Maximum two days

Questions:

1. (25 points) **Dynamic Scheduling**. The following stream of instructions is executed on a processor that supports Tomasulo's algorithm (with reservation stations and without reorder buffer).

```
L.D         F1, 8(R1)       ; F1 = MEM[R1+8]
L.D         F2, 8(R2)       ; F2 = MEM[R2+8]
MUL.D       F4, F1, F2      ; F4=F1*F2
S.D         F4, 8(R2)       ; MEM[R2+8] = F4
ADD.D       F4, F1, F2      ; F4 = F1+F2
ADD.D       F6, F6, F4      ; F6 = F6+F4
```

   (a) Please list the number of dependences in the code including RAW, WAW and WAR.
   (b) Based on the example from Figure 3.7 in the textbook, show what the information tables look like when only the first load has completed and written its result F1. Be sure to include three tables for instruction status, reservation stations and register status.
   (c) Then show what the information tables look like when the MUL.D instruction is ready to write its result F4. Be sure to include three tables for instruction status, reservation stations and register status.
   (d) Finally, assume that the execution of instructions take one cycle for integer ALU operation, 2 cycles for Load/Stores (one for address calculation; another for data access), 2 cycles for ADD.D, and 4 cycles for MUL.D. Fill in the following table for the execution timing (in terms of cycle) of all instructions.

Instruction	**Issue at**	**Execute start at**	**Execute end at**	**Data at**	**Write CDB at**
`L.D      F1, 8(R1)`	1	2	2	3	4
`L.D      F2, 8(R2)`					
`MUL.D    F4, F1, F2`					
`S.D      F4, 8(R2)`					
`ADD.D    F4, F1, F2`					
`ADD.D    F6, F6, F4`					

2. (25 points) **Speculative Tomasulo and Speculative Superscalar**. The following loop of instructions is executed. Assume that the execution takes one cycle for integer ALU operations and branches, 2 cycles for Load/Stores (one for address calculation; another for data access), 2 cycles for ADD.D, and 4 cycles for MUL.D.

```
LOOP:
        L.D     F2, 8(R2)       ; F2 = MEM[R2+8]
        MUL.D   F4, F2, F2      ; F4=F2*F2
        S.D     F4, 8(R2)       ; MEM[R2+8] = F4
        ADD.D   F4, F2, F2      ; F4 = F2+F2
        ADD.D   F6, F6, F4      ; F6 = F6+F4
        DSUBUI  R4, R4, #8      ; R4 = R4 - 8
        BNEQ    R4, LOOP        ; if R4 != 0, jump to LOOP
```

(a) When the code above is executed on an in-order issue processor that supports speculative Tomasulo's algorithm (with reservation stations and reorder buffer), fill in the following timing table for the first 2 iterations of the loop.

Instruction	**Issue at**	**Execute start at**	**Execute end at**	**Data at**	**Write CDB at**	**Commit at**
`LOOP: L.D     F2, 8(R2)`	1	2	2	3	4	5
`MUL.D    F4, F2, F2`						
`S.D      F4, 8(R2)`						
`ADD.D    F4, F2, F2`						
`ADD.D    F6, F6, F4`						
`DSUBUI   R4, R4, #8`						
`BNEQ     R4, LOOP`						
`LOOP: L.D     F2, 8(R2)`						
`MUL.D    F4, F2, F2`						
`S.D      F4, 8(R2)`						
`ADD.D    F4, F2, F2`						
`ADD.D    F6, F6, F4`						
`DSUBUI   R4, R4, #8`						
`BNEQ     R4, LOOP`						

(b) If the code is executed on a dual-issue processor that supports dynamic scheduling, but without speculation, using Figure 3.19 as an example, provide a timing table for the first two iterations of the loop.

(c) If the code is executed on a dual-issue processor that supports dynamic scheduling and speculation, using Figure 3.20 as an example, provide a timing table for the first two iterations of the loop.

3. (15 points) **Very long instruction word**. The following loop is the so-called DAXPY loop (double-precision aX plus Y) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, Y = aX + Y, for a vector length 100. Initially, R1 is set to the base address of array X and R2 is set to the base address of Y:

```
        DADDIU    R4, R1, #800   ;    R1 = upper bound for X
foo:    L.D       F2, 0(R1)      ;    (F2) = X(i)
        MUL.D     F4, F2, F0     ;    (F4) = a*X(i)
        L.D       F6, 0(R2)      ;    (F6) = Y(i)
        ADD.D     F6, F4, F6     ;    (F6) = a*X(i) + Y(i)
        S.D       F6, 0(R2)      ;    Y(i) = a*X(i) + Y(i)
        DADDIU    R1, R1, #8     ;    increment X index
        DADDIU    R2, R2, #8     ;    increment Y index
        DSLTU     R3, R1, R4     ;    test: continue loop?
        BNEZ      R3, foo        ;    loop if needed
```

Assume the functional unit latencies as shown in the table below (note that these numbers have changed from Homework #2). Assume a one-cycle branch delay can be in the ID stage.

Instruction producing result	**Instruction Using result**	**Latency in clock cycles**
FP multiply	FP ALU op	5
FP add	FP ALU op	3
FP multiply	FP store	4
FP add	FP store	3
Integer operations and all loads	Any	1

a) Assume a VLIW processor with instructions that contain five operations, as shown in Figure 3.16 in the textbook. We will compare two degrees of loop unrolling. First, unroll the loop 5 times to extract ILP and schedule it without any stall or the least number of stalls (a stall is an empty issue cycle), collapsing the loop overhead instructions, and then repeat the process but unroll the loop 8 times. Ignore the branch delay slot. Show the two schedules using figures similar to Figure 3.16.

b) What is the execution time per element of the result vector for each schedule? What percent of the operation slots are used in each schedule? How much does the size of the code differ between the two schedules? What is the total register demand for the two schedules?

4. (20 points) **Programming with vector instructions**.

Assume that we have a vector processor (64 elements in a vector) that supports VMIPS code. It has a load/store unit with a start-up overhead of 15 cycles; a multiply unit, 8 cycles; and an add/subtract unit, 5 cycles. We have the following loop that computes two new arrays based on two old arrays.

```
for (k=0; k<64; k++) {
        C[k] = A[k] * B[k];
        D[k] = A[k] + B[k];
}
```

a. Based on the VMIPS code from Page 270, convert the loop into VMIPS code.
b. With chaining and a single memory pipeline (one load or store in a chime), how many chimes are required for this loop? How many clock cycles are required per complex result value, including start-up overhead (assuming startup overhead cannot be overlapped)?
c. If we have two memory pipelines (two loads or stores in a chime), how many chimes are required for this loop? How many clock cycles are required per complex result value, including start-up overhead (assuming startup overhead cannot be overlapped)?

5. (15 points) **Loop Dependence Analysis (GCD test and loop parallelization)**.

a. Please use GCD test to examine if the following loop has a loop-carried dependency:

```
for (k=0; k<100; k++) {
        A[k] = B[2*k+3];
        B[3*k+4] = A[k];
}
```

b. In the following loop, find all the true dependences, output dependences, and antidependences. Eliminate the output dependences and antidependences by renaming.

```
for (i=0; i<100; i++) {
        A[i] = a * B[i];        /* S1 */
        B[i] = A[i] + b;        /* S2 */
        A[i] = C[i] * c;        /* S3 */
        C[i] = D[i] * A[i];     /* S4 */
}
```