Project 1 – Due October 17 (midnight)
(To be completed by groups of 3 students)

One benefit of a dynamically scheduled processor is its ability to tolerate changes in latency or issue capability in out of order speculative processors.
The purpose of this project is to evaluate this effect of different architecture parameters on a CPU design by simulating a modified (and simplified) version of the PowerPc 604 and 620 architectures. We will assume a 32-bit architecture that executes a subset of the RISC V ISA which consists of the following instructions: *lw, sw, fld, fsd, and, andi, or, ori, slt, slti, add, addi, sub, subi, mul, fadd, fsub, fmul, fdiv, beq* and *bne.* See Appendix A of the textbook for instructions' syntax and semantics.

Your simulator should take an input file as a command line input. This input file, for example, **prog.dat**, will contain a RISC V assembly language program (code segment). Your simulator should read this input file, recognize the different fields of the instructions and simulate their execution on the architecture describe below in this handout. In order for you to concentrate on simulating the architecture, the TA will provide you with a parser that reads the input file (for example prog.dat) and generates records with the different fields for each instruction. Your task will be to implement the functional+timing simulator.

The simulated architecture is a speculative, multi-issue, out of order ALU where:

a) The fetch unit fetches up to NF=4 instructions every cycle (as long as they are in a single cache line)
b) A 1-bit branch predictor with 16-entry branch target buffer (BTB) is used. It hashes the address of a branch, L, to an entry in the BTB using bits 7-4 of L.
c) The decode unit decodes (in a separate cycle) the instructions fetched by the fetch unit and stores the decoded instructions in an instruction queue which can hold up to NI=8 instructions.
d) Up to NW=4 instructions can be issued every clock cycle to reservation stations. The architecture has the following functional units with the shown latencies and number of reservation stations.

Unit	**Latency for operation**	**Reservation stations**	**Instructions executing on the unit**
INT	1 (integer and logic operations)	4	*and, andi, or, ori, slt, slti, add, addi, sub, subi*
MULT	4 (pipelined integer multiply)	2	*mul*
Load/Store	1 for address calculation	2 load buffer + 2 store buffer	*lw, fld* *sw, fsd*
FPadd	3 (pipelined FP add)	3	*fadd, fsub*
FPmult	4 (pipelined FP multiply)	4	*fmul*
FPdiv	8 (non-pipelined divide)	2	*fdiv*
BU	1 (condition and target evaluation)	2	*beq, bne*

e) A circular reorder buffer (ROB) with NR=16 entries is used with NB=4 CDB busses connecting the WB stage and the ROB to the reservation stations and the register file. You have to design the policy to resolve contention between the ROB and the WB stage on the CDB busses.

f) The size of the general purpose register file is 64 (32 integer registers and 32 FP registers, each 32-bits long). In addition, 8 integer renaming registers and 8 FP renaming registers are used.
g) A dedicated ALU is used for the effective address calculation in the branch unit (BU) and simultaneously, a special hardware is used to evaluate the branch condition. Also, a dedicated ALU is used for the effective address calculation in the load/store unit.

The simulator should be parameterized so that one can experiment with different values of NF, NI NW, NR and NB. To simplify the simulation, we will assume that the instruction cache line contains NF instructions and that the entire program fits in the instruction cache (it always takes one cycle to read a cache line). Also, the data cache (single ported) is very large so that writing or reading a word into the data cache always takes one cycle.

Note that when you read your code segment, you need to emulate a loader which will load each instruction to a given memory location (in the instruction cache) and evaluate the labels in the branch instructions accordingly. Caches can be implemented using "dictionary" data structures. In addition to the code segment, the input file **`(prog.dat)`** will contain a data segment that starts with the word "**`DATA`**" on a separate line. Each line in this segment indicates the initial content of a memory location:

```
Mem(10000)=34567
Mem(10008)=756.98
```

Note also that for the functional part of the simulator, you need to keep track of the actual values in all the registers, but you only need to keep track of the contents of the memory locations specified in the **DATA** segment of **prog.dat** and the ones used by the store instructions.

Results reporting:

The TA will indicate shortly to you the project submission requirements. For now, however, keep in mind that your simulation should keep statistics about the number of cycles needed for execution, the number of times computations has stalled because 1) the reservation stations of a given unit are occupied, 2) the reorder buffers are full. You should also keep track of the utilization of the CDB busses. This may help identify the bottlenecks of the architecture.

Comparative analysis:

After running the benchmarks with the parameters specified above, perform the following analysis:

1) Study the effect of changing the issue and commit width to 2. That is setting NW=NB=2 rather than 4.
2) Study the effect of changing the fetch/decode width. That is setting NF = 2 rather than 4.
3) Study the effect of changing the window width from which the instructions are issued. That is setting NI = 4 and 10 rather than 8.
4) Study the effect of changing the number of reorder buffer entries. That is setting NR = 4, 8 and 32.

Part 2:

In part 2 of this project, you will add SMT capabilities to your architecture so that it executes two programs (from two prog.dat files) simultaneously. More specifically, your architecture will have two PCs, two instructions queues (each with NI entries), two branch predictors, two sets of registers and two ROBs. Each cycle, the fetch unit will fetch NF instructions from the thread that has fewer entries in its instruction queue. Moreover, up to NW instructions can be issued in each cycle, with priority given to the first thread in even cycles and to the second thread in odd cycles (you may use the same logic to resolve the conflict between the two threads on the CDB).

The API for your simulator should now take two arguments to specify the names of the two **prog.data** files. If only one argument is supplied in the command line, then the simulation will not apply SMT and will revert to the single thread simulation of Part 1. In other words, the command to run your simulator should be:

```
java ./TomasuloSimulator ../Benchmarks/Test1.dat  ../Benchmarks/Test2.dat
```

Moreover, the parameters for the simulator should be set in a configuration file, TomasuloSimulator.config, as follows:

```
NF=4
NI=8
NW=4
NR=16
NB=4
```

After testing your simulator, repeat the experiments that you conducted in Part 1 and analyze the effect of SMT on the performance of the different instances of the architecture (the different values of the architecture parameters).

Test Bechmark:

Use the following as an initial benchmark (i.e. content of the input file **prog.dat**).

```
        ori     R1, R0, 24
        ori     R2, R0, 124
        fld     F2, 200(R0)
loop:   fld     F0, 0(R1)
        fmul    F0, F0, F2
        fld     F4, 0(R2)
        fadd    F0, F0, F4
        fsd     F0, 0(R2)
        addi    R1, R1, -8
        addi    R2, R2, -8
        bne     R1,$0, loop

DATA
     Mem(200) = 2.0
     Mem(24) = 100.0
     Mem(16) = 200.0
     Mem(8) =  300.0
     Mem(124) = 300.0
     Mem(116) = 200.0
     Mem(108) = 100.0
```