Project 1 – Due October 21(midnight)
(To be completed by groups of up to 4 students)

The purpose of this programming assignment 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 (see this web site for more details). We will assume that the architecture executes a subset of the MIPS64 ISA which consists of the following instructions: AND, ANDI, OR, ORI, SLT, SLTI, DADD, DADDI, DSUB, DMUL, LD, SD, L.D, S.D, ADD.D, SUB.D, MUL.D, DIV.D, BEQZ, BNEZ, BEQ, BNE. See the back cover of your textbook for instruction's 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 MIPS assembly language program (code segment). Your simulator should read this input file, recognize the different fields of the instructions and simulate their execution. To simplify the architecture, we will assume that

(a) The entire program fits in the instruction cache, which is separate from the data cache. The cache line is 4 words long.
(b) The load and store operations always result in cache hits and take one cycle to access the cache.
(c) A 32-entry branch target buffer. A branch instruction at location L (word address) uses entry "L mod 32" in the buffer to hold the outcome and possibly the target address of the last execution of the branch.

The Fetch unit fetches up to NF=4 instructions every cycle (as long as they are in a single cache line) and puts them in an instruction queue which can hold up to NQ=8 instructions. The decoding unit decodes up to ND=4 instructions every clock cycle and puts them in another queue (of length NI=8) of instructions waiting to be issued. Up to NW=4 instructions can be issued every clock cycle to reservation stations. Use a Tomasulo-like algorithm to control dynamic scheduling. The simulator should be parameterized so that one can experiment with different values of NF, NQ, ND, NI and NW.

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**
INT0 and INT1	1 (integer and logic operations)	2 + 2	AND, ANDI, OR, ORI, SLT, SLTI, DADD, DADDI, DSUB
MULT	4 (integer multiply)	2	DMUL
Load/Store	1 for address calculation	3 load buffer + 3 store buffer	LD, L.D, SD, S.D
FPU	4 (pipelined FP multiply/add) 4 (non-pipelined divide)	5	ADD.D, SUB.D, MUL.D, DIV.D
BU	1 (condition and target evaluation)	2	BEQZ, BNEZ, BEQ, BNE

All the functional units are pipelined, except the FPU unit, in which the division operation is not pipelined. The architecture also has:

(a) 32 Integer registers and 32 FP registers.
(b) NR=16 reorder buffers (ROB)

(c) The architecture does not use renaming registers. Rather is uses the ROB entries for resolving the name dependences.
(d) 19 reservation stations as shown in the above table. However, only one instruction may be issued to a reservation stations associated to a specific functional unit every cycle and one instruction can start execution in each functional unit every cycle.
(e) NB=4 busses connecting the execution stations to the ROB. That is, up to 4 instructions can finish execution in one cycle.
(f) NC=4 CDB busses connecting the ROB to the registers. That is, up to 4 instructions can commit every clock cycle.
(g) A single port data cache. That is, at most one cache read/write operation can be performed every cycle (priority is given to writes).

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 (test a register or compare two registers).

A dedicated ALU is also used for the effective address calculation in the load/store unit. A load operation takes one cycle to calculate the memory address and one cycle to read the cache (if the cache is not used for a write operation). A store operation uses the Load/Store unit to calculate the address, and then moves to the reorder buffer. A store instruction commits by writing into the cache.

When you read your code segment, you need to emulate a loader which will load each instruction to a given memory location and evaluate the labels in the branch instructions accordingly.

In addition to the code segment, the input file **`(prog.dat)`** should also 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
```

You need to keep track of the actual values in all the registers. As for memory, you may only 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 on the execution time of changing the issue and commit width to 2. That is setting NW=NB=NC=2 rather than 4.

2) With NW=NR=NC=4, study the effect on the execution time of setting NF=ND=2 and NQ=NI=4. That is limiting the fetch/decode width.
3) With NW=NB=NC=NF=ND=4 and NQ=NI=8, study the effect of limiting NR to 8.

Test Bechmark;

Use the following as an initial benchmark (i.e. content of the input file **prog.dat**). More benchmarks will be provided for your testing later.

```
        ORI     R1, R0, 24
        ORI     R2, R0, 124
        L.D     F2, 200(R0)
loop:   L.D     F0, 0(R1)
        MUL.D   F0, F0, F2
        L.D     F4, 0(R2)
        ADD.D   F0, F0, F4
        S.D     F0, 0(R2)
        DADDI R1, R1, -8
        DADDI R2, R2, -8
        BNEZ    R1,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