Last Name: Sample

First Name: ____________________

Student ID: ____________________

Signature: ____________________

Course 304-425B -- Computer Organization and Architecture

Final examination

April 19, 2002, 09:00 -- 12:00

Examiner: Prof. V. Hayward [illegible] Associate Examiner: Prof. F. Ferrie [illegible]

FIRST READ THESE INSTRUCTIONS

◇ **This is a closed book examination**. Calculators and one or two sheets of notes are allowed.
◇ Hand in your signed paper in its entirety (along with all signed exam books) at the end of the examination.
◇ Explain every result concisely **when asked**. Marks will be given for clear, concise solutions.
◇ When a numerical answer is asked, **always write the formula(s)** and then evaluate using the givens.
◇ State any assumption required for an answer if it is not clear in the text of the question.
◇ Please sign this paper at the top of the page, write your name and student number legibly there.
◇ **Put your answers in the space provided, answers in any other location will not be marked**.
◇ This exam has 12 pages, 7 sections for 25 questions. Each question is indicated by a bullet sign (•) and carries four (4) marks which adds up to 100. You have approximately 180 minutes to complete the exam.

PROBLEMS

SECTION 1: Performance (5 questions)

• Consider that the CPI of an ideal machine is 2 when all the memory references, including fetches are cache hits. The real machine version A has a unified cache that misses at a rate of 6%. Another real machine version B has a split cache that misses at a rate of 5% for instructions and 8% for data. The only data access are by load/stores instructions and these form 30% of all instructions. The miss penalty is 20 clock cycles in all cases. Evaluate the performance ratio between these two machines, all other parameters being equal.

MACHINE A: $\text{MISS RATE}_{DATA} = \text{MISS RATE}_{FETCHES} = 0.06$

MACHINE B: $\text{MISS RATE}_{DATA} = 0.08$; $\text{MISS RATE}_{FETCHES} = 0.05$

$CPI_{IDEA} = 2$

$\text{MISS PENALTY} = 20$

$\%\ LD/ST = 30\%$

$\textcircled{A}\ CPI_{REAL} = 2 + 0.06\ 20 + 0.3\ 0.06\ 20 = 3.56$

$\textcircled{B}\ CPI_{REAL} = 2 + 0.05\ 20 + 0.3\ 0.08\ 20 = 3.48$

$\frac{CPI_A}{CPI_B} = 1.023$

• The designers of a processor consider increasing the depth of its single pipeline from 6 to 9 stages and this could increase the clock rate by 10%. When the processor had 6 stages, the average number of stalls per instruction was 0.5, with a deeper pipeline this number increases. What is maximum tolerable average number of stalls per instruction with the 9 stage pipeline to make the effort worthwhile?

$CC_9 = CC_6\ 1.10$ $\text{STALL RATE}_6 = 0.5$ PER INSTR.

SINGLE PIPE : $CPI_{IDEAL} = 1$, FOR A SAME INSTRUCTION COUNT, WRITE:

$$(1 + 0.5)\ CC_6 = (1 + \text{STALL RATE}_9)\ CC_6 \frac{1}{1.10}$$

$$1.5 \geq (1 + \text{STALL RATE}_9)\frac{1}{1.10} \Rightarrow \text{STALL RATE}_9 \leq 1.5 \times 1.10 - 1$$

$$\leq 0.65$$

- A floating point application was compiled for a RISC processor having floating point instructions. One run required 20,000 loads/stores, 5,000 branches, 10,000 ALU instructions, 5,000 fixed-point multiplies, 5,000 floating point additions, and 5,000 floating point multiplies. The count of all other instructions was negligible. The application was used to benchmark a machine with floating point hardware. The following CPI's were found: load/stores=2, branches=1.5, ALU=1.2, fixed-point mults=4, FP adds=4, FP mults=8. The same application run on a machine without floating point hardware but which emulated it with integer routines gave: 20,000 load/stores, 8,000 branches, 60,000 ALU instructions, 10,000 fixed-point multiply instructions. The CPI's were almost the same. What is the speed-up provided by the floating point hardware assuming that the clock rates were the same?

FOR A GIVEN SINGLE BENCHMARK, WE CAN SIMPLIFY BY FINDING THE TOTAL NUMBER OF NEEDED CLOCK CYCLES:

$$\#CC_{FP} = \underbrace{20 \times 2}_{LD/ST} + \underbrace{5 \times 1.5}_{BR} + \underbrace{10 \times 1.2}_{ALU} + \underbrace{5 \times 4}_{FIX*} + \underbrace{5 \times 4}_{FP+} + \underbrace{5 \times 8}_{FP*} = 139.5\ 10^3$$

$$\#CC_{FIX} = \underbrace{20 \times 2}_{LD/ST} + \underbrace{8 \times 1.5}_{BR} + \underbrace{60 \times 1.2}_{ALU} + \underbrace{10 \times 4}_{FIX*} = 164\ 10^3$$

$$SPEEDUP = \frac{164}{139.5} = 1.175$$

- The compiler for this same processor is now enhanced by a strength reduction optimization feature (*e.g.* replacing mults by shifts). Now, half of the fixed-point multiplication are replaced by a sequence of 2 ALU instructions. The other effects are negligible. What is the now speed-up provided by the floating point hardware when using this new compiler?

WITH NEW COMPILER, THESE NUMBER BECOME:

$$\#CC_{FP} = \underbrace{20 \times 2}_{LD/ST} + \underbrace{5 \times 1.5}_{BR} + \underbrace{12.5 \times 1.2}_{ALU} + \underbrace{2.5 \times 4}_{FIX*} + \underbrace{5 \times 4}_{FP+} + \underbrace{5 \times 8}_{FP*} = 132.5\ 10^3$$

$$\#CC_{FIX} = \underbrace{20 \times 2}_{LD/ST} + \underbrace{8 \times 1.5}_{BR} + \underbrace{70 \times 1.2}_{ALU} + \underbrace{5 \times 4}_{FIX*} = 156\ 10^3$$

$$SPEEDUP = \frac{156}{132.5} = 1.177$$

- We focus now on the performance of a machine with virtual memory and one level of blocking cache. The performance figures are 1 cc hit time and a 5% miss rate for the cache. It takes 30 cc to replace a block. The virtual memory system misses at the rate of 0.0002% and it takes 5,000,000 cc to resolve a page fault. To improve the system, a second level cache L2 is introduced that can replace a L1 block in 10 cc. L2 misses at a rate of 1% and it takes 50 cc to replace a block. What is the speed up introduced by the L2 cache?

BEFORE

	~~HIT~~TIME	RATE	PENALTY
CACHE	1	0.05	SEE MM
MM	30	$2\ 10^{-6}$	$5\ 10^6$

AFTER

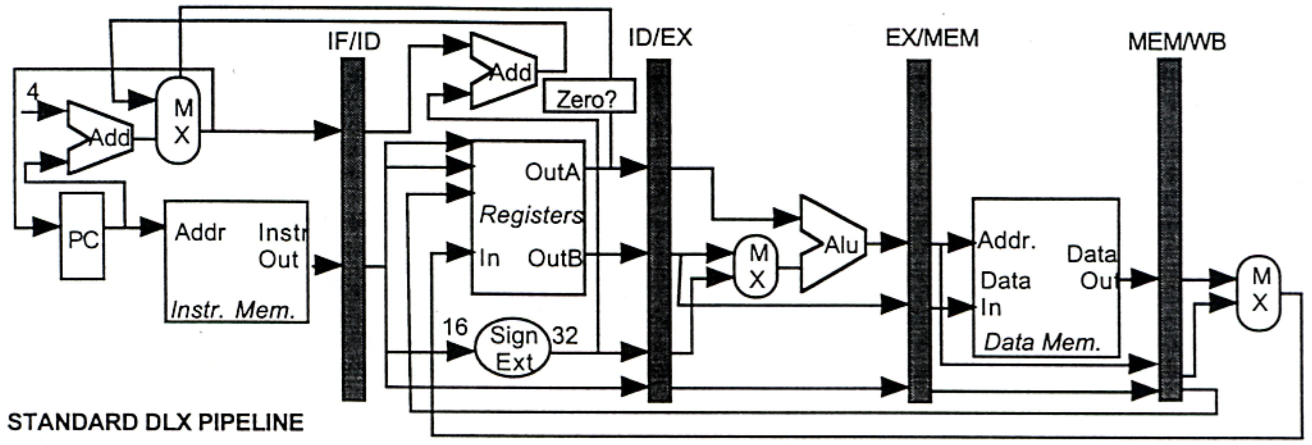
	HITTIME	RATE	PENALTY
L1	1	0.05	SEE L2
L2	10	0.01	SEE MM
M	50	$2\ 10^{-6}$	$5\ 10^6$

$$AMAT = 1 + 0.05(30 + 10) = 3$$

$$AMAT = 1 + 0.05(10 + 0.01(50 + 10)) = 1 + 0.05(10 + 0.6) = 1.53$$

END OF SECTION 1 (5 questions, 5/25 so-far)

SECTION 2: Integer Pipelining (4 questions)



STANDARD DLX PIPELINE

Consider the standard DLX pipe as above and consider it fully bypassed. Recall that basic techniques to handle the branch are "pure delayed branch" and "delayed branch with canceling". Now consider the code sequence:

```
LOOP LW     R2, 0(R3)     \\ load word w
     SEQI   R1, R2, 0     \\ R1 <- 1 if w == 0
     ORI    R4, R2, x0F   \\ set least significant nibble
     BNEZ   R1, OUT       \\ exit if w == 0
     SW     0(R3), R4     \\ store modified w
     ADDI   R3, R3, 4     \\ next item
     J      LOOP          \\ jump back
```

- Indicate the 2 instructions after BNEZ and show timing for "delayed branch with canceling" when:

The branch is taken:

LW	IF	ID	EX	MEM	WB								
SEQI		IF	ID	S	EX	MEM	WB						
ORI			IF	S	ID	EX	MEM	WB					
BNEZ					IF	ID							
SW						IF	CANCEL						
INST. @ OUT							IF	ID	EX	MEM	WB		

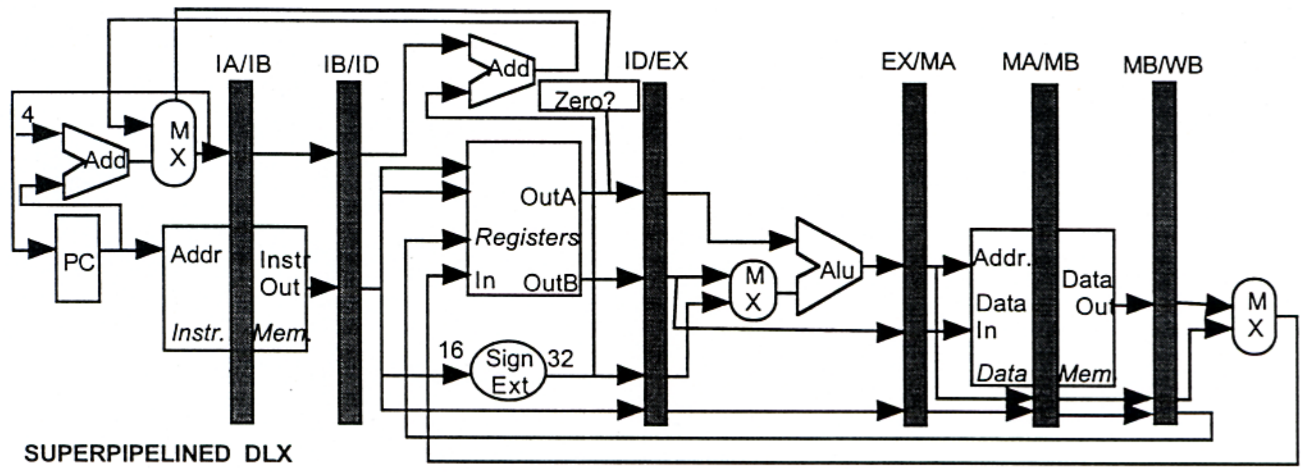
The branch is not taken:

LW	IF	ID	EX	MEM	WB								
SEQI		IF	ID	S	EX	MEM	WB						
ORI			IF	S	ID	EX	MEM	WB					
BNEZ					IF	ID							
SW						IF	ID	EX	MEM	WB			
ADDI							IF	ID	EX	MEM	WB		

- For pure branch delay, does the code execute correctly? If yes, state why. If not, place NOP(s) to restore correctness. Ignore the J instruction and apply no other transformation.

NO, IT WOULD NOT. "SW" WILL BE FETCHED AND WOULD COMPLETE EXECUTION, WHEN IT SHOULD NOT.

LW
SEQI
ORI
BNEZ
NOP
SW
ADDI
J



SUPERPIPELINED DLX

Consider now a superpipelined version of DLX designed in an effort to increase the clock rate (memory accesses are performed in two cc instead of one). Like the standard version, it is fully bypassed, including the data memory, and branches are handled by delayed branches with canceling.

- Using timing diagrams, determine by how much the clock rate should be increased to make the enhancement worthwhile for this sequence of code (it does string copy):

```
LOOP: LB    R1, 0(R2)
      SB    0(R3), R1
      BEQZ  R1, OUT
      ADDI  R2, R2, 1
      ADDI  R3, R3, 1
      J     LOOP          // J is folded, so its CPI=0
```

Standard DLX

	1	2	3	4	5	6								
LB	IF	ID	EX	MEM	WB									
SB		IF	ID	EX	MEM	WB								
BEQZ			IF	S	ID									
ADDI					IF	ID	EX	MEM	WB					
ADDI						IF	ID	EX	MEM	WB				

Superpipelined DLX

							7							
LB	IA	IB	ID	EX	MA	MB	WB							
SB		IA	IB	ID	EX	S	MA	MB	WB					
BEQZ			IA	IB	S	S	ID	EX	MA	MB	WB			
ADDI				IA	S	S	IB	ID	EX	MA	MB	WB		
ADDI							IA	IB	ID	EX	MA	MB	WB	

Calculate here the minimum required clock rate improvement to make the superpipelined design worthwhile?

7/6 SINCE SUPERPIPELINING TAKES 7CC TO EXECUTE VS. 6 CC.

- Schedule this code for the superpipelined case and state why you could use speculative execution, or why you could not:

IT'S POSSIBLE TO SPECULATIVELY "ADDI R2" AND FILL THE LOAD DELAY SLOT. THE OTHER ADDI CAN FILL THE BRANCH DELAY SLOT (COULD USE SB TOO).

```
LOOP: LB    R1, 0(R2)
      ADDI  R2, R2,1
      SB    0(R3), R1
      BEQZ  R1, OUT
      ADDI  R3,R3,1
      J     LOOP.
```

IT CAN BE VERIFIED THAT THIS WORKS, BUT THE TWO BRANCH DELAY SLOT CAN BE ARGUED TO BE PROBLEMATIC.

END OF SECTION 2 (4 questions, 9/25 so-far)

SECTION 3: Loop Transformations With FP Code (3 questions)

The FPUs are fully pipelined and bypassed. In case of write contention (one write per clock cycle) the earliest instruction has priority and stalls the contending instruction(s). Assume the following data:

Instruction producing result	Instruction using result	Latency in clock cycles
Load double	any FPU operation	1
completion of any FPU operations	Store Double	0
FP ADD	FPU operation	3
FP MULT	FPU operation	4

DELAYS

```
LOOP: LD    F2, 0(R1)   ← 1         A[I] = A[I] * c
      MULTD F2, F2, F0
      SD    F2, 0(R1)   ← 4
      LD    F4, 0(R2)               B[I] = (A[I] + B[I])
      ADDD  F2, F2, F4  ← 1
      SD    F2, 0(R2)   ← 3
      ADDI  R1, R1, 8               INTEGER
      ADDI  R2, R2, 8               INDEX AND
      ADDI  R5, R5, -1              LOOP
      BNEZ  R5, LOOP    9 TOTAL     CONTROL
```

- Schedule this loop taking advantage of software pipelining to minimize stalls, but use no other transformation. For conciseness, ignore the init and cleanup code. Assume that, due to prediction, there is no branch delay.

USE SD FROM PREVIOUS ITERATION AND INTEGER CODE TO FILL DELAYS.

ABSENCE OF RENAMING LIMITS THE OPTIONS.

OTHER POSSIBLE SOLUTION STARTS LIKE:

```
SD F2
LD F2
LD F4
:
```

```
SD    F2, -8(R2)
LD    F2, 0(R1)
MULTD F2, F2, F0   - 1
ADDI  R1, R1, 8
ADDI  R2, R2, 8            3 DELAYS
ADDI  R5, R5, -1           LEFT.
LD    F4, -8(R2)
SD    F2, -8(R1)
ADDD  F2, F2, F4
BNEZ  R5, LOOP
                   - 2
```

- Unroll this loop twice (that is, one interation for `I` and `I+1`), and schedule it to minimize stalls.

FIRT UNROLL AND RENAME

```
I    LD    F2, 0(R1)
     MULTD F2, F2, F0
     SD    F2, 0(R1)
     LD    F4, 0(R2)
     ADDD  F6, F4, F2
     SD    F6, 0(R2)   //FREE F2
I+1  LD    F8, 8(R1)
     MULTD F8, F8, F0
     SD    F8, 8(R1)
     LD    F10, 8(R2)
     ADDD  F12, F8, F10
     SD    F12, 8(R2)
     ADDI  R1, R1, 16
     ADDI  R2, R2, 16
     ADDI  R5, R5, -2
     BNEZ  LOOP
```

WE NOW HAVE PARALLEL CODE SO WE CAN OVERLAP TWO ITERATIONS

LOTS OF OTHER OPTIONS FOR A GOOD SCHEDULE!

THEN SCHEDULE

```
LD    F2, 0(R1)
LD    F8, 8(R1)
MULTD F2, F2, F0
MULTD F8, F8, F0
LD    F4, 0(R2)
LD    F10, 8(R2)
ADDI  R5, R5, -2
ADDD  F6, F2, F4    // MULTD'S DONE
ADDD  F12, F8, F10  // START ADDD'S
SD    F2, 0(R1)     // STORE MULTD'S
SD    F8, 8(R1)
SD    F6, 0(R2)     // STORE ADDD'S
SD    F12, 0(R2)
ADDI  R1, R1, 16    // POINTERS
ADDI  R2, R2, 16
BNEZ  R5, LOOP      // NO DELAY!
```

• Consider computing the dot product of vectors stored in two arrays as in:

```
Loop  LD     F2, 0(R1)          // A[I]
      LD     F4, 0(R2)  -1      // B[I]
      MULTD  F2, F2, F4         // A[I] * B[I]
      ADDD   F0, F0,F2  -4      // P += A[I] * B[I]
      ADDI   R1, R1, 8          // I = I + 1
      ADDI   R2, R2, 8
      ADDI   R3, R3, -1
      BNEZ   R3, Loop
```

• Schedule it for a VLWI processor that has the following structure (use the same latencies).

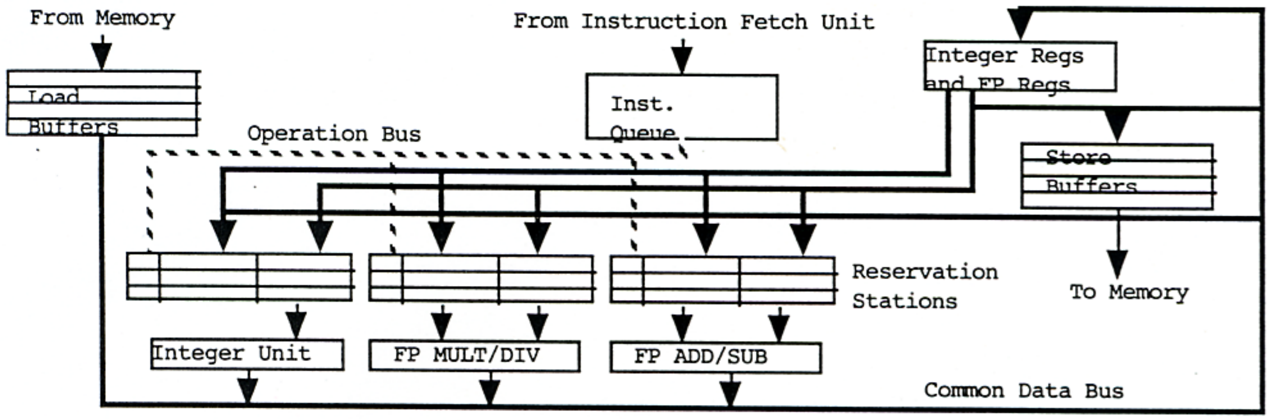
Memory Ref 1	Memory Ref. 2	FP Op 1	FP Op 2	Integer Op/branch
LD F2, 0(R1)	LD (F4, 0(R2)			
				ADDI R1, R1, 8
		MULTD F2,F2,F4		ADDI R2, R2,8
				ADDI R3, R3, -1
			ADDD F0,F0,F2	BNEZ R3,LOOP

COMMENT: AS EVIDENCED BY THIS EXAMPLE, VLWI PROCESSORS PERFORM POORLY WITHOUT HEAVY CODE TRANSFORMATION.

END OF SECTION 3 (3 questions, 12/25 so-far)

SECTION 4: Tomasulo's Algorithm (4 questions)

Consider the pipeline below. The integer unit can be controlled to carry out any type of integer instructions. It has one FP ADD/SUB unit and one MULT/DIV unit. The load and store buffers have four entries each. The reservation stations have three entries.



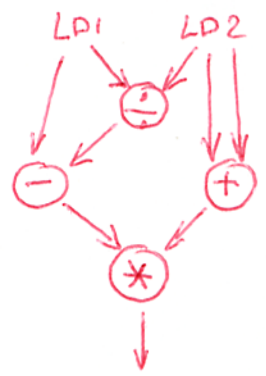
Assume that the latencies are 0 for all integer operations, 1 for loads, 4 for the FP add/sub's, 6 for the multiplication and 12 for the divides, regardless of the instruction using the result. All units are fully pipelined. The Common Data Bus is written at the end of the last clock cycle of an operation. An operation that depends on the written value starts at the next clock cycle. The Common Data Bus can support multiple data transfers. Assume that a new instruction is fetched at each clock cycle and is issued at the next clock cycle if there is free entry in the reservations or in the buffers.

- By inspection of the code sequences A and B (watch for the SUBD) and by completing the timing tables, determine the case in which the second SD completes first and state why, assuming that all the reservation stations and the store buffers are initially free.

Fetch	Code A	Issue	Exec	Write CDB
CC1	LD F2, 0(R1)	CC2	3-4	CC4
CC2	LD F4, 0(R2)	CC3	4-5	CC5
CC3	DIVD F6, F2, F4	CC4	6-18	CC18
CC4	SD 0(R1), F6	CC5	19	
CC5	SUBD F6, F2, F6	CC6	19-23	CC23
CC6	ADDD F4, F4 F4	CC7	8-12	CC12
CC7	MULTD F2, F4, F6	CC8	24-30	CC30
CC8	SD 0(R2), F2	CC9	31	

Fetch	Code B	Issue	Exec	Write CDB
CC1	LD F2, 0(R1)	CC2	3-4	CC4
CC2	LD F4, 0(R2)	CC3	4-5	CC5
CC3	DIVD F6, F2, F4	CC4	6-18	CC18
CC4	SD 0(R1), F6	CC5	19	
CC5	SUBD F2, F2, F6	CC6	19-23	CC23
CC6	ADDD F4, F4 F4	CC7	8-12	CC12
CC7	MULTD F2, F4, F6	CC8	19-25	CC25
CC8	SD 0(R2), F2	CC9	26	

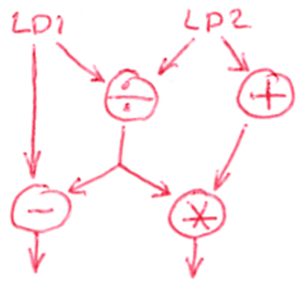
Comment DATA DEPENDENCIES ARE SUCH THAT DIVD → SUBD → MULTD, SO ALL THE LATENCIES ADD UP. THE COMPLETE DATA FLOW DIAGRAM IS:



Comments HERE, { DIVD → SUBD & MULTD ; ADD → MULTD }

SO THERE IS MORE PARALLELISM, THE MULTD CAN START 5 CC EARLIER (1 CC ISSUE + 4 CC LATENCY)

AT	CC8
DIVD	EXECS
SUBD	PENDING DIVD
ADD	EXECS
MULT	PENDING ADDD, DIVD,



Use "`Mem[Reg[R1]]`" to denote, for example, the *value* fetched by the first load, "`Reg[R1]`" to denote the *value* held in register `R1`, and `#8` to denote the value 8.

- Indicate in the tables the state of the reservation stations and of the registers at clock cycle 8 for code **B** only:

Name	Busy	Op.	Vj	Vk	Qj	Qk
MULT/DIV-1	✓	DIVD	MEM [REG[R1]]	MEM[REG[R2]]	–	–
MULT/DIV-2	✓	MULTD	–	–	ADD/SUB-2	MULT/DIV-1
MULT/DIV-3						
ADD/SUB-1	✓	SUBD	MEM[REG[R1]]	–	–	MULT/DIV-1
ADD/SUB-2	✓	ADDD	MEM[REG[R2]]	MEM[REG[R2]]	–	–
ADD/SUB-3						
INT-1						
INT-2						
INT-3						

Field	F0	F2	F4	F6	F8	R1	R2	R3	R4
Qi		ADD/SUB-2	ADD/SUB-2	MULT/DIV-1					

- What is the estimated throughput measured in number of floating point instructions per second when the architecture on the previous page repeatedly executes the code below (a loop unrolled a large number of times)? Assume same behavior, perfect cache performance, and absence of structural hazards such as shortage of reservation station entries.

```
        ...
        ...
1       LD     F2, 0(R1)
2       LD     F4, 0(R2)
3       SUBD   F6, F2, F4
4       ADDD   F8, F2 F6
5       ADDI   R1, R1, 8
6       MULTD  F2, F4, F4
        ADDI   R2, R2, 8
        SD     0(R1), F8
        SD     0(R2), F2
        LD     F2, ...           \\ again and again
        LD ...
        ...
```

NOTE: IF THE ADDI'S WERE BETWEEN THE SD'S AND LD'S, THEN THE CODE WOULD BE PARALLEL AND HENCE LIMITED BY STRUCTURAL LIMITATIONS SUCH AS FETCH RATE HOWEVER, EACH NEW CALCULATION DEPENDS ON THE PREVIOUS ONE SO WE MUST ESTIMATE THE TOTAL LATENCY.

THERE ARE TWO PARALLEL CALCULATIONS, MULTD AND SUBD → ADDD, BOTH TAKE ABOUT 14 CLOCK CYCLES TO COMPLETE BEFORE A NEW SEQUENCE CAN START. SO WE HAVE ABOUT 3 FLOP'S PER 14 CC OR 1 FLOP PER 4.6 CC. CALL CC_{TIME} THE CLOCK CYCLE TIME:

$$\text{THROUGHPUT} = \#\ \text{FLOP}/s = \frac{1}{4.6\ \text{CCTIME}}$$

END OF SECTION 4 (4 questions, 16/25 so-far)

SECTION 5: Branch predictors (2 questions)

- A machine has a 2-bit branch predictor. Estimate how many correct and how many incorrect predictions are made while executing the code below per outer iteration, assuming a 100% hit rate and no clashes in the predictor table (Giving just a number will earn you no marks, do not mind the nonsensical computation).

```
for(;;) {                               /* begin for ever */
        i = 0;
        do {
                ++i;
                j = 0;
                do {
                        ++j;
                        if (j == 10 || j == 20) /* call it B1 */
                                A[i] = j;
                } while (j < 100);              /* call it B2 */
        } while (i < 4);                        /* call it B3 */
}                                       /* end for ever */
```

B1 BEHAVIOR $\underbrace{\text{T...TNT...TNT...T}}_{100}\underbrace{\text{T...}}$ 98 CORRECT 2 INCORRECT

B2 BEHAVIOR $\underbrace{\text{T.............N}}_{100}\underbrace{\text{T...}}$ 99 CORRECT 1 INCORRECT

$\left.\right\}\times 4$

B3 BEHAVIOR $\underbrace{\text{TTTN}}_{4}\underbrace{\text{T...}}$ 3 CORRECT 1 INCORRECT

TOTAL: 791 CORRECT PREDICTIONS AND 13 MISPREDICTIONS

- Assume now that a machine as a (1,1) correlating predictor. What is its performance when executing the code below? Report performance in the same manner as above (same remarks).

```
for(;;) {                               /* begin for ever */
        i = 0;
        while (i < 100) {               /* call it B1 */
                if (i % 2 == 0)         /* call it B2 */
                        A[i] = 0;
                else
                        A[i] = 1;
                ++i;
        }
}                                       /* end for ever */
```

EACH BRANCH HAS A PAIR OF 1 BIT PREDICTORS, CALL THEM P0/P1. P0 IS USED WHEN THE LAST BRANCH WAS TAKEN AND P1 WHEN IT WAS NOT. WORK A FEW ITERATIONS TO TEST ALL CASES:

	B1	B2	B1	B2	B1	B2	B1	B2	
OUTCOME:	N	N	N	T	N	N	N	T	---
PREDITOR:	N/N	N/N	N/N	N/N	N/N	N/T	N/N	N/N	---
NEW STATE:	N/N	N/N	N/N	N/T	N/N	N/N	N/N	N/T	---
	INIT.				REPEATING BEHAVIOR				

COMMENT: B1 ALWAYS CORRECTLY PREDICTED AND B2 ALWAYS MISPREDICTED. THE IS NO CORRELATION, SO PERFORMS NO BETTER THAN 1 BIT PREDICTOR.

END OF SECTION 5 (2 questions, 18/25 so-far)

SECTION 6: Loop level parallelism (3 questions)

• Consider this loop and list all the dependencies: true data dependencies and name dependencies; and whether they are loop carried.

```
for (i = 0; i < 100; ++i) {
    c[i] = a[i] + c[i];                 /* S1 */
    b[i] = c[i + 1] + a[i];             /* S2 */
    c[i + 1] = a[i + 1] + a[i + 1];     /* S3 */
    a[i + 1] = b[i + 1] + b[i + 1]      /* S4 */
}
```

DATA DEPENDENCIES

S1 DEPENDS ON S3 (LC)
S1 " " S4 (LC)
S2 " " S4 (LC)

ANTI-DEPENDENCIES

S2 DEPENDS ON S3
S3 " " S4
S2 " " S1 (LC)
S4 " " S2 (LC)

OUTPUT-DEPENDENCIES

S3 DEPENDS ON S1 (LC)

LC: LOOP CARRIED

• Use software renaming and/or other transformation(s) so that it becomes parallel:

BRUTE FORCE SOLUTION:
RENAME ALL ~~V~~ARIABLES INVOLVED IN NAME DEPENDENCIES, THEN START LOOP AT S3 SINCE NO DATA DEPENDENCY BETWEEN S3 AND S2.

```
X[i] = T[i] + Z[i]
Y[i] = C[i+1] + T[i]
Z[i+1] = a[i+1] + a[i+1]
T[i+1] = b[i+1] + b[i+1]
```

```
X[0] = T[0] + Z[0]
Y[0] = C[1] + T[0]
FOR(i=1; i<100; ++i){
    Z[i] = a[i] + a[i];
    T[i] = b[i] + b[i];
    X[i] = T[i] + Z[i];
    Y[i] = C[i+1] + T[i];
}
Z[100] = a[100] + a[100];
T[100] = b[100] + b[100];
```

LEAST EFFORT SOLUTION: NOTICED THAT THE ONLY NAME DEPENDENCY IS BETWEEN S3 AND S2 AND RENAME JUST C.

```
c[0] = a[0] + c[0]
b[0] = X[1] + a[0]
FOR (i=1; i<100; ++i){
    c[i] = a[i] + a[i]
    a[i] = b[i] + b[i]
    c[i] = a[i] + c[i]
    b[i] = X[i+1] + a[i]
}
c[100] = a[100] + a[100];
a[100] = b[100] + b[100];
```

• Can this loop be transformed to become parallel. Why or why not?:

```
for (i = 1; i < 100; ++i) {
    a[i] = a[i - 1] + a[i];          /* S1 */
    b[i] = a[i] + a[i + 1];          /* S2 */
}
```

S2 IS DATA DEPENDENT ON S1, SO THEY CANNOT RUN IN PARALLEL. S1 IS LOOP-CARRIED DEPENDENT ON S1, SO ONE ITERATION MUST TERMINATE BEFORE ANOTHER ONE CAN START. NO PARALLELISM.

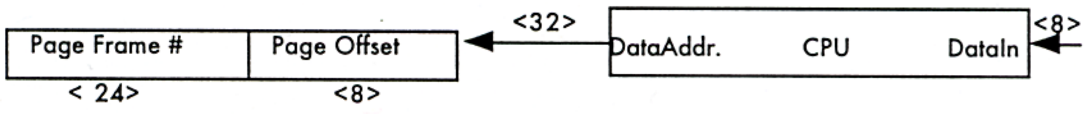
END OF SECTION 6 (3 questions, 21/25 question so-far)

SECTION 7: Memory Hierarchy (4 questions)

Consider the organization of the Cheap'O computer memory hierarchy. Notice that the CPU produces ~~16~~ 32 bit virtual addresses and the memory is byte addressed. A "load byte" instruction requests the byte stored at address "`0A1B 2C08`". From the content of the TLB, and of the cache, determine the value of the byte that is returned.

- As the request works its way through the hierarchy, specify the values of

1. Data Address : `0A1B 2C08`
2. Page frame # : 0A 1B2C
3. Page Offset : 08 (PHYS ADDR. EF08)
4. Cache Tag : EF0
5. Cache Index : $(10)_2 = 2$
6. Cache Block Offset : $(00)_2 = 0$
7. Byte returned : CC (ASSUMING THIS BYTE NUMBERING: $\overset{0}{CC}$ $\overset{1}{DD}$ $\overset{2}{EE}$ $\overset{3}{FF}$)



V	R	W	Tag	Phys. Frame #
1	1	1	987654	23
1	1	1	FABBAF	89
1	1	1	0A1B2C	EF
1	1	1	B0AC55	01
			<24>	<8>

Translation Lookaside Buffer

Cache Address Format

Tag	Index	Block Offset
<12>	<2>	<2>

	V	Tag	Data	V	Tag	Data	V	Tag	Data	V	Tag	Data
Set-0	1	012	00112233	1	CDE	CCDDEEFF	1	89A	8899AABB	1	456	44556677
Set-1	1	345	44556677	1	F01	8899AABB	1	BCD	00112233	1	789	CCDDEEFF
Set-2	1	678	8899AABB	1	234	44556677	1	EF0	CCDDEEFF	1	ABC	00112233
Set-3	1	9AB	CCDDEEFF	1	567	00112233	1	123	44556677	1	DEF	8899AABB

Set Associative Cache (4 sets of 4 blocks)

"Cheap'O Computer" micro-architecture, memory hierarchy, Version 2, Rev. 5

In robotics and graphics, a common case is the product of a vector by a 4x4 matrix that can be coded as follows. Whether this is coded inline as shown, or as nested loops, has no bearing on the answer since, in this question, we are interested in cache operation. (p, a and T are floats).

```
p[0] = T[0][0] * a[0] + T[0][1] * a[1]+ T[0][2] * a[2] + T[0][3] * a[3];
p[1] = T[1][0] * a[0] + T[1][1] * a[1]+ T[1][2] * a[2] + T[1][3] * a[3];
p[2] = T[2][0] * a[0] + T[2][1] * a[1]+ T[2][2] * a[2] + T[2][3] * a[3];
p[3] = T[3][0] * a[0] + T[3][1] * a[1]+ T[3][2] * a[2] + T[3][3] * a[3];
```

The compiler arranges the elements of the matrix row-wise such that `T[i][j]` is contiguous to `T[i][j+1]`. Suppose also that the cache blocks are just four floats long and that data is properly aligned. Assume that:

1. there is no conflict replacements,
2. the cache is "write through",
3. it is "no-allocate",
4. it is blocking,

- The hit-time is 1 CC and the penalty ($b * 4$), where b is the block size in floats. The cache is just big enough to hold 32 floats. Would it better to have 8 blocks of 4 floats, or 4 blocks of 8 floats? Justify your answer quantitatively.

CASE 8x4: WHEN T[0][0] IS LOADED, A ROW IS LOADED IN ONE BLOCK
WHEN ~~a~~ a[0] IS LOADED, VECTOR a IS LOADED IN A SECOND BLOCK
THEN, T[1][0], T[2][0], AND T[3][0] ALLOCATE 3 MORE BLOCKS.
TOTAL 32 ACCESSES ~~WITH~~ 5 PENALTIES: 32+5x16 = 112 CC TO PERFORM ACCESS TO MULTIPLY

CASE 4x8: WHEN T[0][0] IS LOADED, TWO ROWS ARE LOADED IN ONE BLOCK
WHEN a[0] IS LOADED, VECTOR a IS LOADED IN A BLOCK
THEN T[2][0] ALLOCATE ONE MORE BLOCK FOR TWO ROWS
TOTAL 32 ACCESSES WITH 3 PENALTIES: 32+3x32 = 128CC TO PERFR SAME NUMBE OF ACCESSES : CONCLUSION, SMALL BLOCKS BETTER.

- For the case of 8 blocks of 4 floats, which of these two improvements would be preferable: reducing the penalty to 6 CC by adding a non-blocking feature, or making the cache "write allocate"?

FIRST ENHANCEMENT : 32 + 5 x 6 = 62 CC FOR READS

SECOND ENHANCEMENT CAUSES AN EXTRA PENALTY WHEN P[0] IS WRITTEN, SO COUNTING THE WRITES : 36 + 6 x 16 = 132 CC

IT IS UNLIKELY THAT 4 WRITES ARE WORTH 70 CC. THIS IS DUE TO THE FACT THAT THERE ARE MANY MORE READS THAN WRITES.

- Answer these yes/no questions

◇ "A reorder buffer" is a method to reduce the miss penalty in caches [yes/no]: NO

◇ "Blocking in matrix computations" is used to minimize the number of page faults [yes/no]: ~~NO~~ YES (BLOCKING AT CACHE BLOCK SIZE) (BUT APPLIES TO VM AS WELL)

◇ "Protection" is a software technique to speed up recursive code [yes/no]: NO

◇ "A write buffer" makes it possible to pipeline block replacement in caches [yes/no]: NO

END OF SECTION 7 (4 questions) and END OF EXAM (and end of the 25 questions)