Last Name: SAMPLE First Name: ____________________

Student ID: ____________________ Signature: ____________________

Course 304-425B -- Computer Organization and Architecture

Final examination

April 18, 2000, 9:00 -- 12:00

Examiner: Prof. V. Hayward

Associate Examiner: Prof. K. Khordoc

INSTRUCTIONS

- **This is a closed book examination.** Calculators and up to two sheets of notes are allowed.
- Explain every result concisely **when asked**. Marks will be given for clear, concise solutions.
- State any assumption required for an answer if it is not clear in the text of the question.
- This exam has 12 pages including this one. It has 7 sections for 24 questions (including a bonus question) indicated by the bullet sign (•). The marks add up to 100.
- 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** and keep all the pages together.

PLEASE NOTE CAREFULLY

- Make sure that the signed paper in its entirety is handed in (along with all signed exam books) at the end of examination.
- Make sure that the answers are put in the space provided, **answers in any other location will not be marked.**
- You have approximately 180 minutes to complete the exam.

Section 1: Performance (12 points)

- Apply Amdahl's law to compute the speed-up factor for a machine to which an enhancement is added to improve some mode of execution by a factor 10. This mode is used 50% of the time, measured as a percenta of the original exec time. (4 points)

$$T_e = T_u \left[(1 - FE) + \frac{FE}{SU_E} \right]$$

$$SU = \frac{T_u}{T_e} = \frac{1}{(1-FE) + \frac{FE}{SU_E}} = \frac{1}{.5 + \frac{.5}{10}} = \frac{1}{.55} = 1.82$$

- Derive a variant of Amdahl's law to compute the speed-up factor for a machine to which an enhancement is added to improve some mode of execution by a factor 10. However in this question, the mode is used 50% the time measured as a percentage of the *enhanced* exec time. (4 points)

Hint: start from the definition of speed-up: $Speed\_up = \frac{ExecTime_{unenhanced}}{ExecTime_{enhanced}}$, in short: $SU = \frac{T_u}{T_e}$.

$$SU = \frac{T_u}{T_e} = \frac{1}{T_e} T_e \left[(1 - F_E) + F_E \, SU_E \right]$$

$$= .5 + 5 = 5.5$$

- Assume that we have a Load/Store machine which behaves with a perfect cache as follows:

ALU ops	40%	1 clock cycle
Load/Stores	30%	2 clock cycles
Branches and others	30%	2 clock cycles

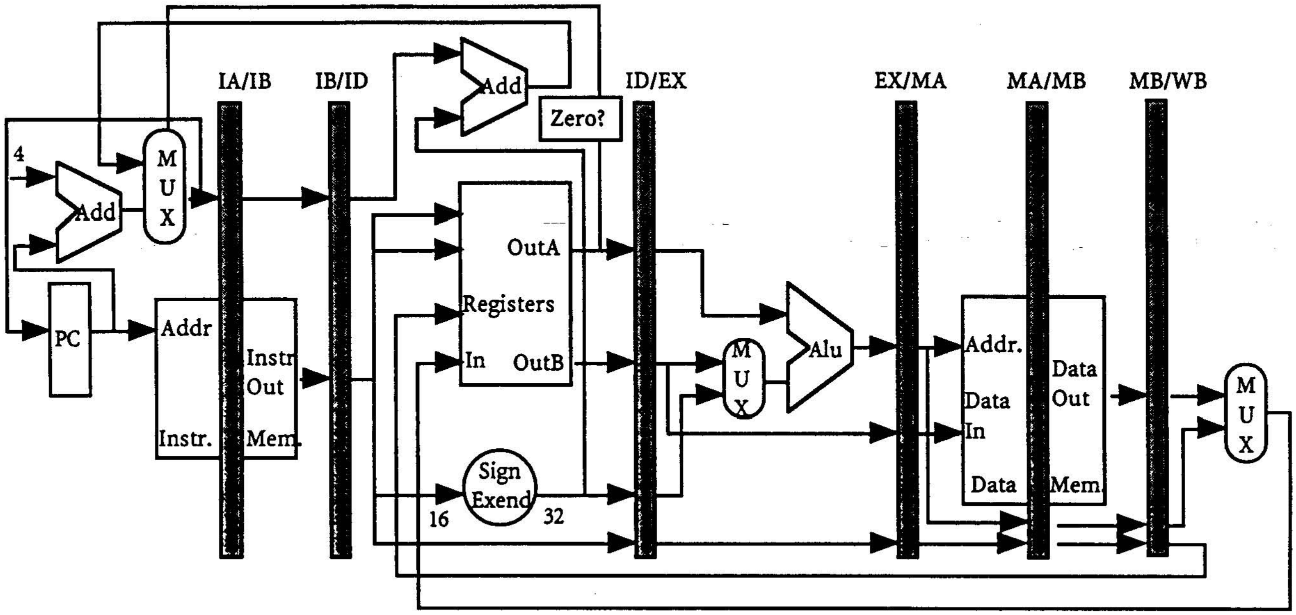
The machine is modified to add new ALU instructions that have one source operand in memory. These new *register-memory* instructions have a clock cycle count of 2. The total number of ALU operations, branches, ar others instruction remains the same, of course, but the number of loads and stores is divided by two. Is this enhancement worth implementing? (4 points)

$$\frac{CPU_{TIME}}{CC} = IC \sum F_i \, CPI_i \quad s.t. \ \sum F_i = 1 \qquad IC, \ CLASSES, \begin{cases} F_i \\ CPI's \end{cases} \ CHAN$$

NEW CLASSES	:	ALU_{OP_1}	ALU_{OP_2}	L/S	BO
NEW CPI's	:	1	2	2	2
NEW F_i's	:	$\frac{.4-.15}{.85}$	$\frac{.15}{.85}$	$\frac{.15}{.85}$	$\frac{.3}{.85}$
NEW IC	:	$.85 \, IC_{OLD}$			

$$\frac{TIME_{NEW}}{TIME_{OLD}} = \frac{\cancel{.85}}{\cancel{.85}} (.25 + .15 \times 2 + .15 \times 2 + .3 \times 2) = 1.45$$

Timing analysis reveals that the memory cycles in the standard DLX pipeline are the limiting factor for clock c time improvement. One design option is to split the memory cycles in an attempt to increase the clock rate. Th often called super pipelining and is illustrated in the diagram below. Complete instruction fetches takes two st: `IA` and `IB`. In the first stage, the memory addressed is specified, in the second, the instruction is read out. same technique is applied to the `MEM` stage, now split into a `MA` and a `MB` stage. The new design is fully pipeli This is symbolically represented by introducing two new pipe registers.

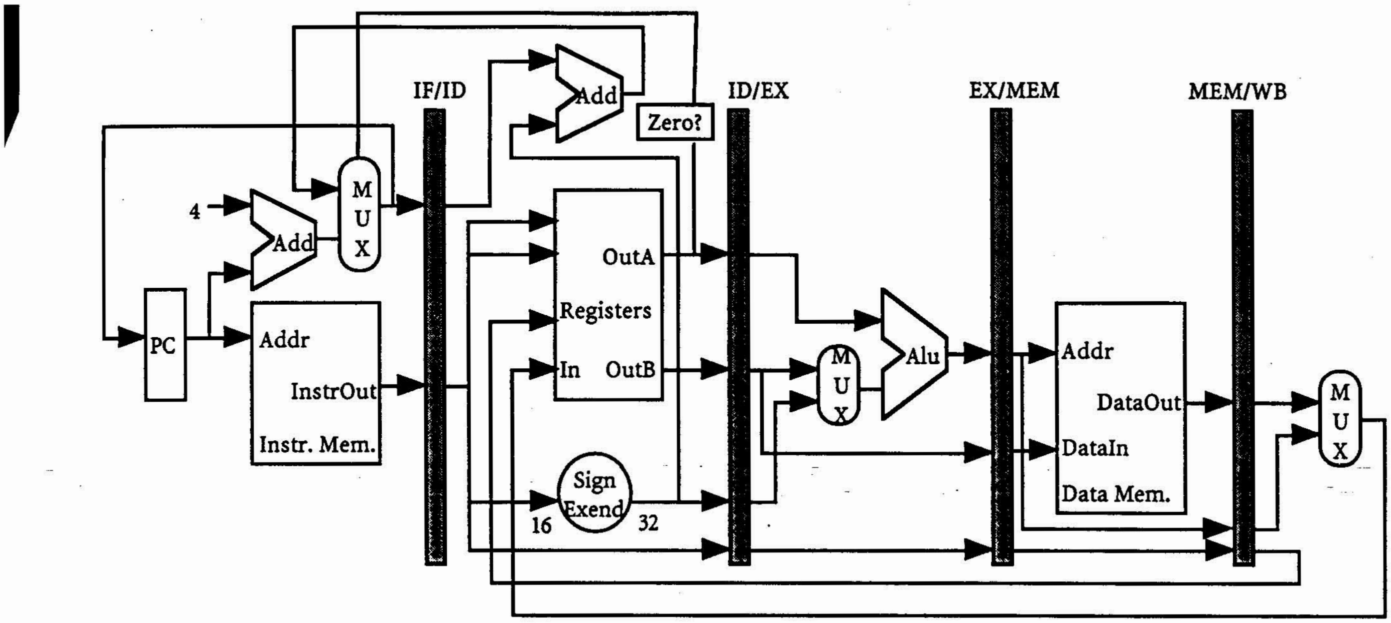


SUPERPIPELINED DLX

- Assuming full bypassing/forwarding (including to and from the memory) use the chart below to repon timing diagram for this code. . Also note that the branch must stall. Show the branch behavior to be del: branch. Suppose that the jump instructions benefit from branch folding and that there is a hit. (10 points)

```
LOOP:     LW       R1, 0(R2)
          SW       R1, 0(R3)
          BEQZ     R1, OUT
          ADDI     R2, R2, 4
          ADDI     R3, R3, 4
          J        LOOP
OUT:      .....
```

LW	IA	IB	ID	EX	MA	MB	WB															
SW		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									
J								IA	IB	ID												
LW								~~IA~~	~~IA~~	ID	EX	MA	MB	WB								
SW									IA	IB	ID	EX	S	MA	MB	WB						



STANDARD DLX PIPELINE

Recall that there are four basic techniques to handle branches in a pipeline like DLX's:
(A) flush (or freeze) a number of instructions after the branch; (B) static prediction such as "predict-not-taken
(C) delayed branch which creates "delay slots"; (D) delayed branch with canceling.

Consider now the following sequence to compute the double of the absolute value of a number in memory:

```
1.              LW   R2, 0(R3)        \\ load number
2.              SLTI R1, R2, 0        \\ R1 <-- 1 if a < 0
3.              BEQZ R1, SKIP         \\ skip if a > 0
4.              SUB  R2, R0, R2       \\ negate
5.   SKIP:      ADD  R2, R2, R2       \\ double
6.              SW   0(R3), r2        \\ store back
```

Show the timing of this sequence for the DLX pipeline assuming full forwarding and bypassing hardware assuming a register read and a write in the same clock cycle implicitly "forwards" through the register file (' first and then read). Use the chart to show the timing of instructions starting at instruction SLTI *when the br is taken.* Fill-in the two blank entries according to the case. (note: a similar question was given last term, howe is NOT the same question).

- (B)"predict-not-taken": (5 points)

LW	IF	ID	EX	ME	WB														
SLTI		IF	ID	S	EX	ME	WB												
BEQZ			IF	S	S	ID	Ex	ME	WB										
SUB						IF	ID												
ADD							IF	ID	Ex	ME	WB								

- Assuming now that the machine can detect hazards, has forwarding hardware, and uses delayed branches (c C). Schedule the following code, to minimize the stalls. (5 points):

```
LOOP: Ⓐ → SGT  R4, R1, R6     \\ compare R1 with R6
  Ⓑ ────→ BNEZ R4, OUT        \\ if R1 > R6
      Ⓒ → LW   R2, 0(R3)      \\ Load number
  Ⓓ ────→ SLTI R4, R2, 0      \\ R4 <- 1 if a < 0
      Ⓔ → BEQZ R4, SKIP       \\ skip if a > 0
          SUB  R2, R0, R2     \\ negate
SKIP:     ADD  R2, R2, R2     \\ double
          ADDI R3, R3, 4      \\ increment pointer
          SW   0(R3), r2      \\ store back
          ADDI R1, R1, 1      \\ increment counter
          J    LOOP           \\ loop back to while test
OUT:      AND  R2, R0, R0     \\ clear R2
          ...
```

Ⓐ/Ⓓ DATA HAZARD

Ⓑ/Ⓔ BRANCH DELAY (DELAYED BRANCH): Ⓑ OK , Ⓔ NEEDS (

Ⓒ LOAD HAZARD

MANY STRATEGIES TO FILL THE SLOTS, E.G.

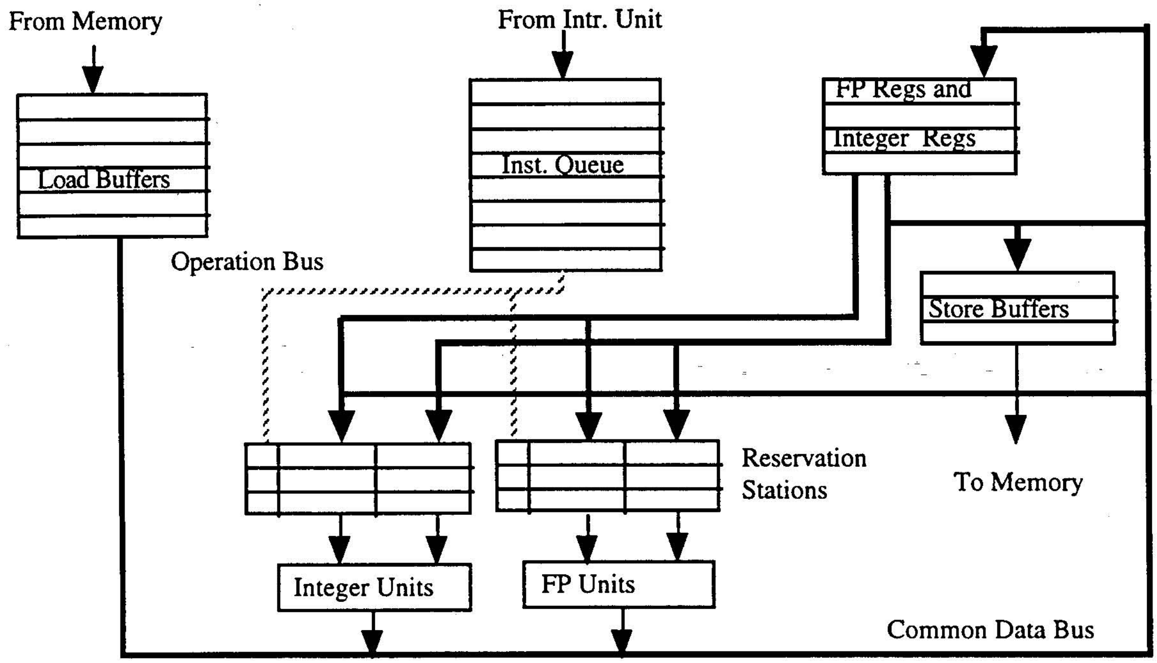
- MOVE THE "ADDI'S" UP
- MOVE THE "LOAD" BEFORE BNEZ : FILLS TWO SLOTS.
- MOVE THE "AND" BEFORE BNEZ
- DOUBLE NUMBER IN DELAY SLOT (KNOWLEDGE OF SEMANT

EXAMPLE

```
LOOP:  SGT   R4, R1, R6
       LW    R2, 0(R3)      LOAD NUMBER REGARDLESS OF
                            BRANCH OUTCO
       BNEZ  R4, OUT
       SLTI  R4, R2, 0      DO TEST IN DELAY SLOT
       ADDI  R3, R3,        FILL WITH ADDI
       BEQZ  R4, SKIP
       ADDI  R1, R1, 1      FILL SLOT WITH ADDI
       SUB   R2, R0, R2
SKIP:  ADD   R2, R2, R2
       SW    0(R3), R2
       J     LOOP
OUT:   AND   R2, R0, R0
```

[illegible]

Consider the pipeline below. The integer units can be controlled to carry out any types of integer instructions an the FP units any types of floating point operations.



```
LOOP: LD      F2,0(R1)
      MULTD   F4,F2,F0
      LD      F6,0(R2)
      ADDD    F6,F4,F6
      SD      0(R2),F6
      ADDI    R1,R1,#8
      ADDI    R2,R2,#8
      SGTI    R3,R1,DONE
      BEQZ    R3,LOOP
```

Consider the code at the left which implement the vector operation: Y = a * X + Y where X and Y are vector arrays.

Assume the latencies are 0 for all integer operations including loads, 4 for additions, 6 for the multiplication regardless of the instruction using result. The Common Data Bus is written and read on the same cycle and (support multiple data transfers (so there is no structural hazard there).

Use "`Mem[10+Reg[R1]]`" to denote, for example, the *value* fetched by first load, "`Reg[R1]`" to denote the *value* to be held in register `R1`, and to denote the value 8.

To illustrate operation, the table below indicates the status of the pipeline once the instructions of the first iterati have issued (that is at clock cycle 8), starting from a blank state.

Instruction Status including the CC counts spent in each stage Instruction	Issue	Execute	Write Result
LD	1	2	3
MULTD	2	3--7	8
LD	3	4	5
ADDD	4	8--?? 9 - 10	?? 11
SD	5	?? 6	??
ADDI	6	7	8
ADDI	7	8	?? 9
SGTI	8	? 9	?? 10

CALL FPU1
INT 1 INT 2 INT 3

- Indicate in the table below the state of the reservation stations. (4 points)

Reservation Stations						
Name	Busy	Op	Vj	Vk	Qj	Qk
FPU 1	Y	MULTD	MEM[0+REG[R1]]	REG[F0]	–	–
FPU 2	Y	ADDD	–	MEM[0+REG[R2]]	FPU1	–
INT 1	Y	ADDI	REG[R1]	#8	–	–
INT 2	Y	ADDI	REG[R2]	#8	–	–
INT 3	Y	SGTI	–	DONE	INTU1	–

- Indicate in the table below the status of the registers. (2 points)

Register Status									
Field	F0	F2	F4	F6	....	R1	R2	R3	...
Qi			FPU1	FPU 2		INT1	INT2	INT3	

- Indicate in the table below the state of the store buffers (2 points)

Store Buffers			
Field	Store 1	Store 2	Store 3
Qi	FPU 2		
Busy	Y		
Address	0 + [R2]		

Ignoring the branch delay, now show the new state of the machine, **one clock cycle later** (this means a new load has been issued).

- Indicate in the table below the state of the reservation stations. (4 points)

Reservation Stations						
Name	Busy	Op	Vj	Vk	Qj	Qk
FPU 2	Y	ADDD	REG[F4]	MEM[0+REG[R2]]	–	–
INTU2	Y	ADDI	REG[R2]	# 8	–	–
INTU3	Y	SGTI	REG[R1]	DONE	–	–

- Indicate in the table below the status of the registers. (2 points)

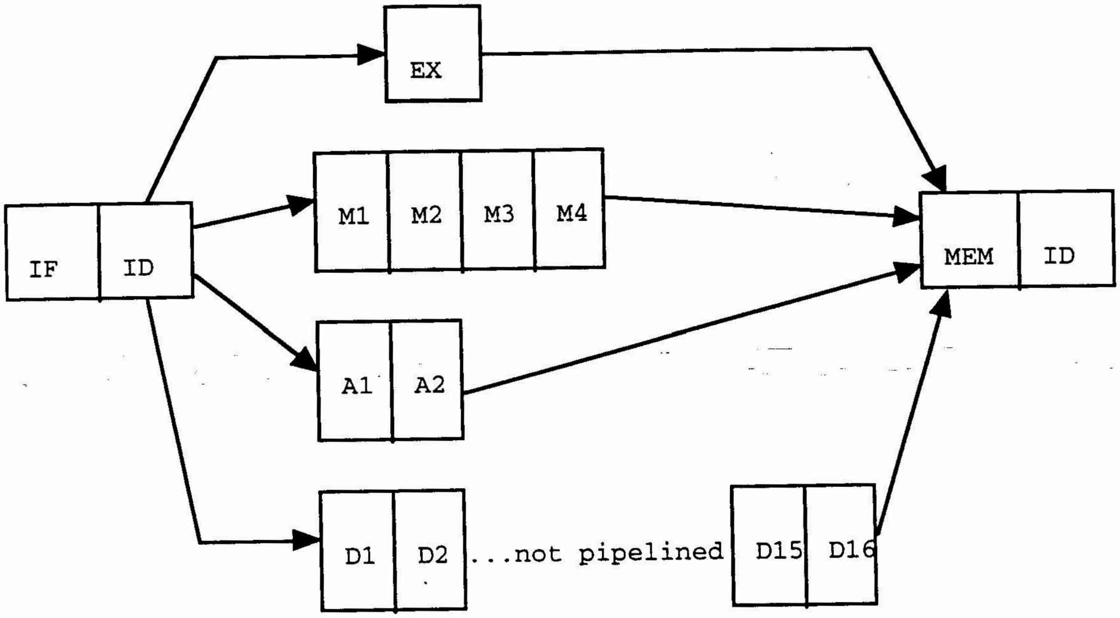
Register Status									
Field	F0	F2	F4	F6	...	R1	R2	R3	...
Qi		LD1		FPU2			INT2	INT3	

- Indicate in the table below the state of the store buffers (1 points)

Store Buffers			
Field	Store 1	Store 2	Store 3
Qi	FPU 2		
Busy	Y		
Address	0 + [R2]		

Section 4. Unrolling (5 points)

Consider a standard FP pipeline as in the mid-term:



Consider again the loop of the previous question:

```
LOOP: LD        F2,0(R1)  → 1cc
      MULTD     F4,F2,F0          → 3 cc
      LD        F6,0(R2)  → 1cc
      ADDD      F6,F4,F6          → 1 cc
      SD        0(R2),F6
      ADDI      R1,R1,#8
      ADDI      R2,R2,#8
      SGTI      R3,R1,DONE
      BEQZ      R3,LOOP
```

- Unroll this loop twice and schedule it for minimal execution time on average when run on the pipeline above. Ignore the branch delay and assume that all branches are correctly predicted. (5 points)

TWO STEPS

```
LOOP: LD    F2, 0(R1)
      MULTD F4, F2, F0
      LD    F6, 0(R2)
      ADDD  F6, F4, F6
      SD    0(R2), F6
      LD    F8, 8(R1)
      MULTD F10, F8, F0
      LD    F12, 8(R2)
      ADDD  F12, F10, F12
      SD    8(R2), F12
      ADDI        #16
      ADDI        #16
      SGTI
      BEQZ        LOOP
```

UNROLL

```
LD    F2, 0(R1)
LD    F8, 8(R1)
MULTD F4, F2, F0
MULTD F10, F8, F0
LD    F6, 0(R2)
LD    F12, 8 (R2)
ADDD  F6, F4, F6
ADDD  F12, F10, F12
SD    0(R2), F6
SD    8(R2), F12
{ INTEGER CODE }
```

START M
AS EARLY
POSSIBLE

COULD BE
FOR MORE
SCHEDULIN

SCHEDULE

Section 5: Branch predictors (15 points)

Consider this infinite loop and its assembly code translation

```
a = 1;                                         ADDI  R1, R0, 1   // init a
b = 1;                                         ADDI  R2, R0, 1   // init b
while (1) {      /* for ever */          B1:   BNEZ  R1, ELSE
    if (a == 0)                                ADDI  R1, R0, 1
        a = 1;                                 J     B2
    else                                ELSE:  ADDI  R1, R0, 0
        a = 0;                           B2:   BEZ   R1, B3
    if (a != 0)                                ADDI  R2, R0, 0
        b = 0;                           B3:   BNEZ  R2, B1
    if (b == 0)                                ADDI  R2, R0, 1
        b = 1;                                 J     B1
}
```

In the table below, the successive values of a and b are listed. Notice the period two. The sequence of taken (T) an not taken (N) branch outcomes is also given in the table below.

a b	
1 1	B1 outcome: T
0 1	B2 outcome: T
0 1	B3 outcome: T
0 1	B1 outcome: N
1 1	B2 outcome: N
1 0	B3 outcome: N
1 1	B1 outcome: T
0....1 And so-on...	B2 outcome: T

- A machine has a 2-bit branch predictor mechanism. What is the performance of this predictor while executin this code in the steady state in terms of correct predictions(s) per iteration? A concise explanation must be given to get the marks. (5 points)

EACH BRANCH (B1, B2, B3) HAS A TWO BIT PREDICTOR

B1 SEQUENCE : T, N, T, N ...

B2 SEQUENCE : T, N, T, N ---

B3 SEQUENCE : T, N, T, N ---

ONE CORRECT PREDICTION OUT OF TWO : 50% 1/2

- A machine has a (1,1) correlating branch predictor. What is its performance while executing the same code the steady state in terms of correct prediction(s) per iteration? Fill the table below to get the marks. (5 point

LAST BRANCH NOT TAKEN / LAST BRANCH TAKEN

B1 prediction bits: NN	B1 prediction: N	B1 outcome: T	UPDATE NO
B2 prediction bits: TT	B2 prediction: T	B2 outcome: T	NO UPDATE
B3 prediction bits: NN	B3 prediction: N	B3 outcome: T	UPDATE
B1 prediction bits: TN	B1 prediction: N	B1 outcome: N	NO UPDAT
B2 prediction bits: TT	B2 prediction: T	B2 outcome: N	UPDATE N
B3 prediction bits: NT	B3 prediction: N	B3 outcome: N	NO UPDA
B1 prediction bits: TN	B1 prediction: T	B1 outcome: T	NO UPD
B2 prediction bits: NT	B2 prediction: N	B2 outcome: T	UPDATE
B3 prediction bits: NT	B3 prediction: T	B3 outcome: T	NO UPD

Average number of correct predictions? 2/3 for the code below, supposing that there is a hit in the buffer (that is: predicted taken), but the prediction is incorrect. (5 points)

```
1.            SLTI R5, R1, 0       \\ compare R1 with 0
2.            BNEZ R5, SKIP        \\ if R1 >= 0 skip
3.            SUBI R1, R0, R1      \\ negate
4.  SKIP:     MULT R1, R1, R1      \\ double
5.            SW   R1, 0(R7)       \\ store it
6.            AND  R1, R0, R0      \\ clear R1
```

SLTI	IF	ID	EX	ME	WB														
BNEZ		IF	S	ID	EX	ME	WB												
MULT				IF	ID*														
SUBI						IF													
MULT							IF												

SEND PC HIT, F

KILL MULT

Section 6: Loop level parallelism (15 points)

Consider this loop:

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

- List all the dependencies: output dependencies, anti-dependencies, and true data dependencies and indicate : each dependency the pair of statements and which are "loop carried" (5 points)

Output Dependencies: S1 - S3 LOOP CARRIED a[i] → a[i-1]

SAME

Anti Dependencies: S3 - S1 LOOP CARRIED

Data Dependencies S2 - S3

Rewrite the loop so it becomes parallel. Solve this problem in two different ways:

- First use software renaming, not changing the structure of the loop: (5 points)

LOOP CARRIED DEPENDENCIES INVOLVE a[i]. THAT IS a[i-1] OF NEXT ALWAYS WRITTEN, NEVER READ. CALL IT TEMP[i]

```
for (————){
   TEMP[i-1] = c[i-1] +n
   b[i] = m + c[i]
}  a[i] = a[i]+ b[i]
```

- Second transform the loop without renaming so it becomes parallel: (5 points)

```
a[0]= c[0] +n
for(i=1 ; i< 99 ; ++i) {
        b[i] = m+ c[i];
        a[i] = a[i] + b[i];
        a[i] = c[i] + n;
}
b[99] = m+ c[99];
a [[illegible]]= a[ ]+ b[99];
```

Section 7: Memory Hierarchy (18 points)

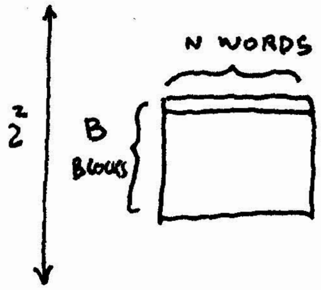
A cache system has B blocks of N words and total storage capacity L (for valid bits, tags, and data) measured in bits. Recall that the degree of associativity A is defined as the number of blocks per set. Assume further that the memory address space is 2^Z and that the memory is word addressed (each word has W bits). Call H the hit time, M the miss rate, and P the miss penalty measured in clock cycles.

Consider now this contrived but interesting example (read the whole section before starting). The benchmark test is to visit (read only) all the addresses in the address space exactly once.

- Calculate the AMAT of the cache system for this test as a function of B, N, W, Z, H, and P starting from a blank cache (all the valid bits are off). In developing the formula, take the case of a direct mapped cache, or equivalently $A = 1$, that is M sets. (6 points)

AMAT= $H + MP$ 2^Z WORDS VISITED

↳ MISS RATE ?



NUMBER OF MISSES : $\frac{2^Z}{N}$

NUMBER OF ACCESSES : 2^Z

} RATE = $\frac{1}{N}$

$$AMAT = H + \frac{P}{N} = 2 + \frac{20}{4} = 7$$

LOCALITY DOES NOT APPLY SINCE ALL LOCATIONS VISITED ONCE

- Work out the result for $B = 16, N = 4, A = 1, Z = 32, W = 32, H = 2$, and $P = 20$. (4 points)

1 MISS , 3 HITS IN CLOCK CYCLES

$4 \times 2 + 20$ FOR EACH BLOCK OF 4 WORD

AVERAGE ACCESS TIME = $\frac{28}{4} = 7$ CC .

Note that these last two questions are independent. You can solve the numerical example by reasoning it out and then derive the formula, or you can develop the formula first and then plug the numbers in.

- Bonus question!: Solve the same problems for $A = 2$ (5 + 5 points

MORE ASSOCIATIVITY DOES NOT CHANGE RESULT .

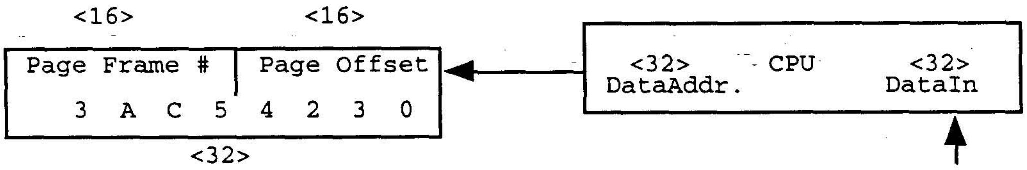
ice the missing field sizes in each place ...

CPU requests this sequence of addresses:3AC54230, A35C2340, and 57BF2344. If there is a miss, ate a replacement by showing which tag gets changed and assume the blocks continue to hold the same In any case, indicate below the values returned to the CPU. (4 points)

30: PHYS. ADDR. 3AC5244; INDEX : 2 ; TAG : 3AC52 ; MISS; RETURN 2..2 OR D.

40: PHYS. ADDR. A35C123; INDEX: 1; TAG : A35C1 ; MISS; RETURN 1..1 OR E..E

44: PHYS. ADDR. 57BF244; INDEX: 2; TAG: 57BF2; MISS; RETURN D..D OR 2..2 ACCORDING TO LRU



V	R	W	<16> Tag	<12> Phys.
0	0	0	2 D B 0	2 B A
1	1	0	3 A C 5	2 4 4
1	1	0	A 3 5 C	1 2 3
1	1	0	5 7 B F	2 4 4

Translation Lookaside Buffer

<20>	<3>	<5>
Tag	Index	Block Offet
<28>		

Cache Addresses Format

		<20>	<1024>			<20>	<1024>
V	D	Tag	Data	V	D	Tag	Data
1	0	CB442	0..............0	1	0	24542	F..............F
1	0	~~98765~~**	1..............1	1	0	24442	E..............E
1	0	~~76AB2~~*	2..............2	1	0	~~12342~~***	D..............D
1	0	29831	3..............3	1	0	24442	C..............C
1	0	13542	4..............4	1	0	24CC2	B..............B
1	0	F4B42	5..............5	1	0	2AA42	A..............A
1	0	00000	6..............6	1	0	22342	9..............9
1	0	21212	7..............7	1	0	2FFFF	8..............8

Note: this means that all the bytes in each block have the same value. In this case: "8"

Set Associative Cache
(8 sets of 2 blocks)
Write Back, Write Allocate

52
5C1
F2