Problem 1 on Recursion (24 points): For the C function and assembly below, fill the table with as many rows as you can when **pcount_r(5)** is called. Do not fill the columns with N/A. Write your answers in hexa. **Write values only. Do not write 0x.**

```
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x & 1) + pcount_r(x >> 1);
}
```

```
  120: pcount_r:
  120:   movl    $0, %eax
  130:   testq   %rdi, %rdi
  140:   je      .L6
  150:   pushq   %rbx
  160:   movq    %rdi, %rbx
  170:   andl    $1, %ebx
  180:   shrq    %rdi
  190:   call    pcount_r
  1A0:   addq    %rbx, %rax
  1B0:   popq    %rbx
  1C0:.L6:
  1C0:   rep; ret
```

stack address in hexa	**stack valuein hexa**
0x80	N/A
0x78	**U**
0x70	**V**
0x68	**W**
0x60	**X**
0x58	**Y**
0x50	**Z**
0x48	
0x40	
0x38	
0x30	
0x28	

iteration	**rsp inhexa**	**rbx in hex**	**rdi in hex**	**eax in hex**
when entering	0x80	9	5	0
0		**A**		**M**
		B		
1		**C**		**N**
		D		
2		**E**		**O**
		F		
3		**G**		**P**
		H		
ret, pop		**I**		**Q**
ret, pop		**J**		**R**
ret, pop		**K**		**S**

Problem 2 on Stack Overflow (22 points): The C code shown below is compiled with and without *stack proctector* to generate the assembly code listed below.

```
int len(char *s) {   return strlen(s); }

void lptoa(char *s, long *p) {
  long val = *p;
  sprintf(s,"%ld",val);
}
int main() {   longlen(INT_MAX-1); }
```

```
long longlen(long x) {
  long v;
  char buf[8];
  v = x;
  lptoa(buf,&v);
  return len(buf);
}
```

common code

```
len:
     movl   $0, %eax
     movq   $-1, %rcx
     repnz scasb
     movq   %rcx, %rax
     notq   %rax
     subq   $1, %rax
     ret
```

```
lptoa:
     subq   $8, %rsp
     movq   (%rsi), %rdx
     movl   $.LC0, %esi
     movl   $0, %eax
     call   sprintf
     addq   $8, %rsp
     ret
```

```
main:
     subq   $8, %rsp
     movl   $2147483647, %edi
     call   longlen
     movl   $0, %eax
     addq   $8, %rsp
     ret
```

Without stack protector

```
longlen:
     subq   $32, %rsp
     movq   %rdi, 8(%rsp)
     leaq   8(%rsp), %rsi
     movq   %rsp, %rdi
     call   lptoa
     movq   %rsp, %rdi
     call   len
     addq   $32, %rsp
     ret
```

With stack protector

```
longlen:
     subq   $48, %rsp
     movq   %fs:48, %rax
     movq   %rax, 24(%rsp)
     xorl   %eax, %eax
     movq   %rdi, 8(%rsp)
     leaq   8(%rsp), %rsi
     leaq   16(%rsp), %rdi
     call   lptoa
     leaq   16(%rsp), %rdi
     call   len
     movq   24(%rsp), %rdx
     xorq   %fs:48, %rdx
     jne    .L7
     addq   $48, %rsp
     ret
.L7:
     call   __stack_chk_fail
```

Fill the table with appropriate values without and with stack protector. Leave the entries empty if not applicable.

	gcc flag	no stack protector	stack protector
len	assembly for allocating stack		N/A
	stack size in decimal		N/A
	assembly for freeing stack		N/A
lptoa	assembly for allocating stack		
	stack size in decimal		
	assembly for freeing stack		
	"char *s" address relative to rsp after entering lptoa		
	"long *p" address relative to rsp after entering lptoa		
	"val" address relative to rsp after entering lptoa		N/A
longlen	assembly for allocating stack		
	stack size in decimal		
	assembly for freeing stack		
	"x" address relative to rsp after entering longlen	N/A	N/A
	"v" address relative to rsp after entering longlen		
	"buf" address relative to rsp after entering longlen		
	canary register name	N/A	
	canary address relative to rsp	N/A	
	canary value	N/A	
	assembly for erasing canary value	N/A	
	assembly for canary cross check	N/A	

Problem 3 on Variable Frame (24 points): The code in (a) gives a function containing a variable-size array while the one in (b) shows portions of the code gcc generates for function vframe. Fill the table and answer the questions.

```
(a) C code
long vframe(long n,long idx,long *q){
  long i;
  long *p[n];
  p[0] = &i;
  for (i=1; i<n; i++) p[i] = q;
  return *p[idx];
}
(b) Portions of generated assembly code
     long vframe(long n, long idx, long *q)
     n in %rdi, idx in %rsi, q in %rdx
     Only portions of code shown
1 vframe:
2       pushq %rbp                          Save old %rbp
3       movq %rsp, %rbp                     Set frame pointer
4       subq $16, %rsp                      Allocate space for i (%rsp = S1)
5       leaq 28(,%rdi,8), %rax
6       andq $-16, %rax
7       subq %rax, %rsp                     Allocate space for array p (%rsp = s2)
8       leaq 15(%rsp), %rax
9       shrq $3, %rax
10      leaq 0(,%rax,8), %r8                Set %r8 to &p[0]
11      movq %r8, %rcx                      Set %rcx to &p[0] (%rcx=p)
        ...
     Code for initialization loop
     i in %rax and on stack, n in %rdi, p in %rcx, q in %rdx
12 .L3:                                  loop:
13      movq %rdx, (%rcx,%rax,8)            Set p[i] to q
14      addq $1, %rax                       Increment i
15      movq %rax, -8(%rbp)                 Store on stack
16 .L2:
17      movq -8(%rbp), %rax                 Retrieve i from stack
18      cmpq %rdi, %rax                     Compare i:n
19      jl .L3                              If <, goto loop
        ...
Code for function exit
20      leave                               Restore %rbp and %rsp
21      ret                                 Return
```

Write the answers in decimal, NOT in hexa decimal.

n	s1	s2	p	e1=s1-p	e2=p-s2
5	401	**A**	**B**	**C**	**D**
6	403	**E**	**F**	**G**	**H**

Line 6 indicates alignment by _____**I**_____ by setting the lowest _____**J**_____ bits to _____**K**_____.
The code guarantees alignment by _____**L**_____ for the values of s2 and p .

Problem 4 on Sequential Computer (20 points): For the instructions below, fill the table with a sequence of numbers necessary to execute each instruction. List numbers in the order in which they flow. Assume the PC is filled with a correct address for fetching. ***Refer to the textbook and lecture notes for icode, ifun, and register assignment.***

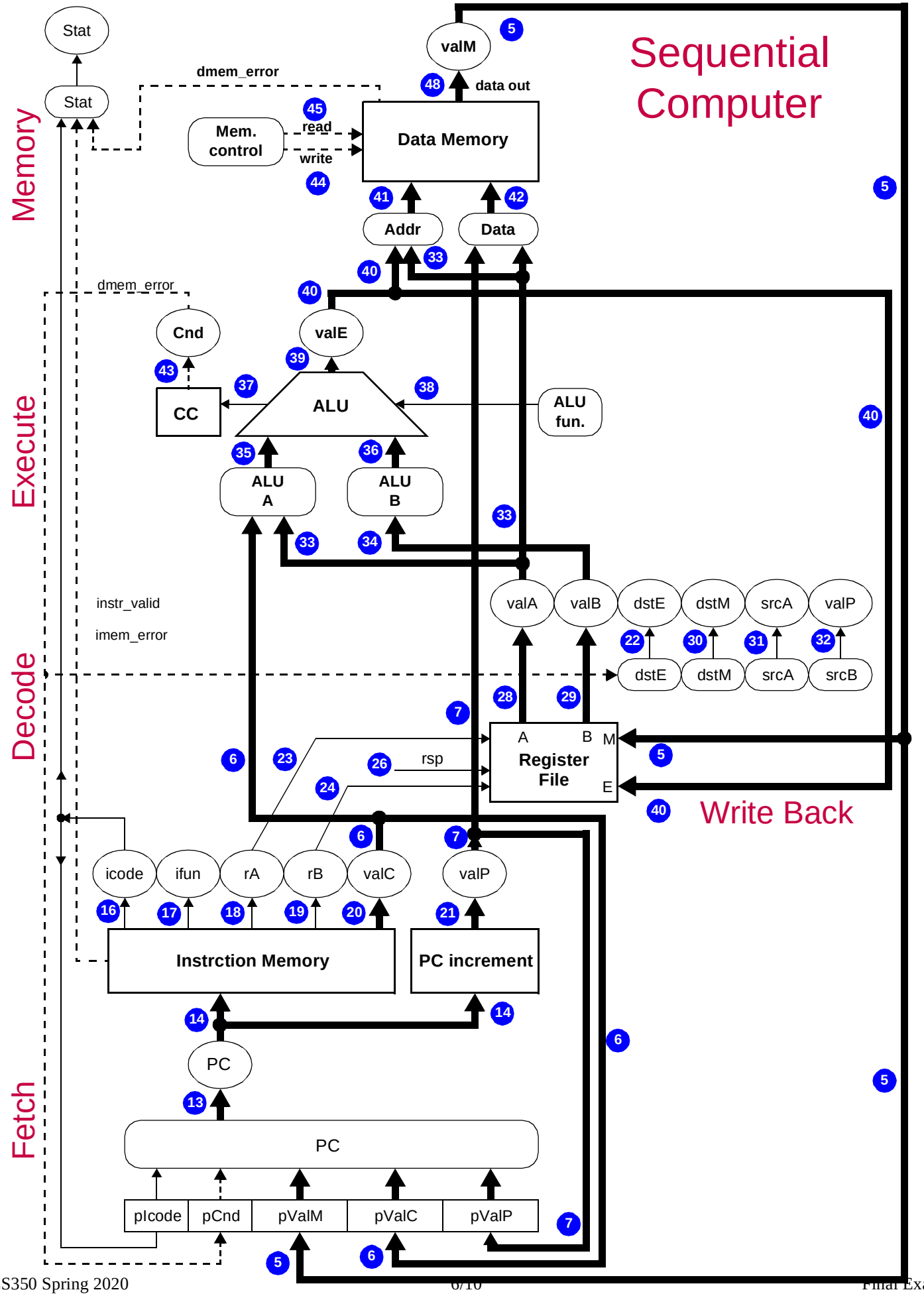
Instruction	Fetch	Decode	Execute	Memory	Write Back
addq %rax,%rbx					
mrmovq 5(%rdx),%rax					
push %rax					
jne loop					
call func					

Assuming the following values, fill the table below with values. If not applicable, leave it blank. See the table attached for register assignment.

Memory addresses each are 8 bytes (64 bits).
All instructions are at memory address 0x0000.
memory 0x1000: 1 (64 bits)
memory 0x2000: 2 (64 bits)
memory 0x3000: 3 (64 bits)
memory 0x4000: 4 (64 bits)

rax=1, rbx=2, rdx=0x1000, rsp=0x2000
loop is at 0x3000
func is at 0x4000
previous instructions resulted in greater than zero.

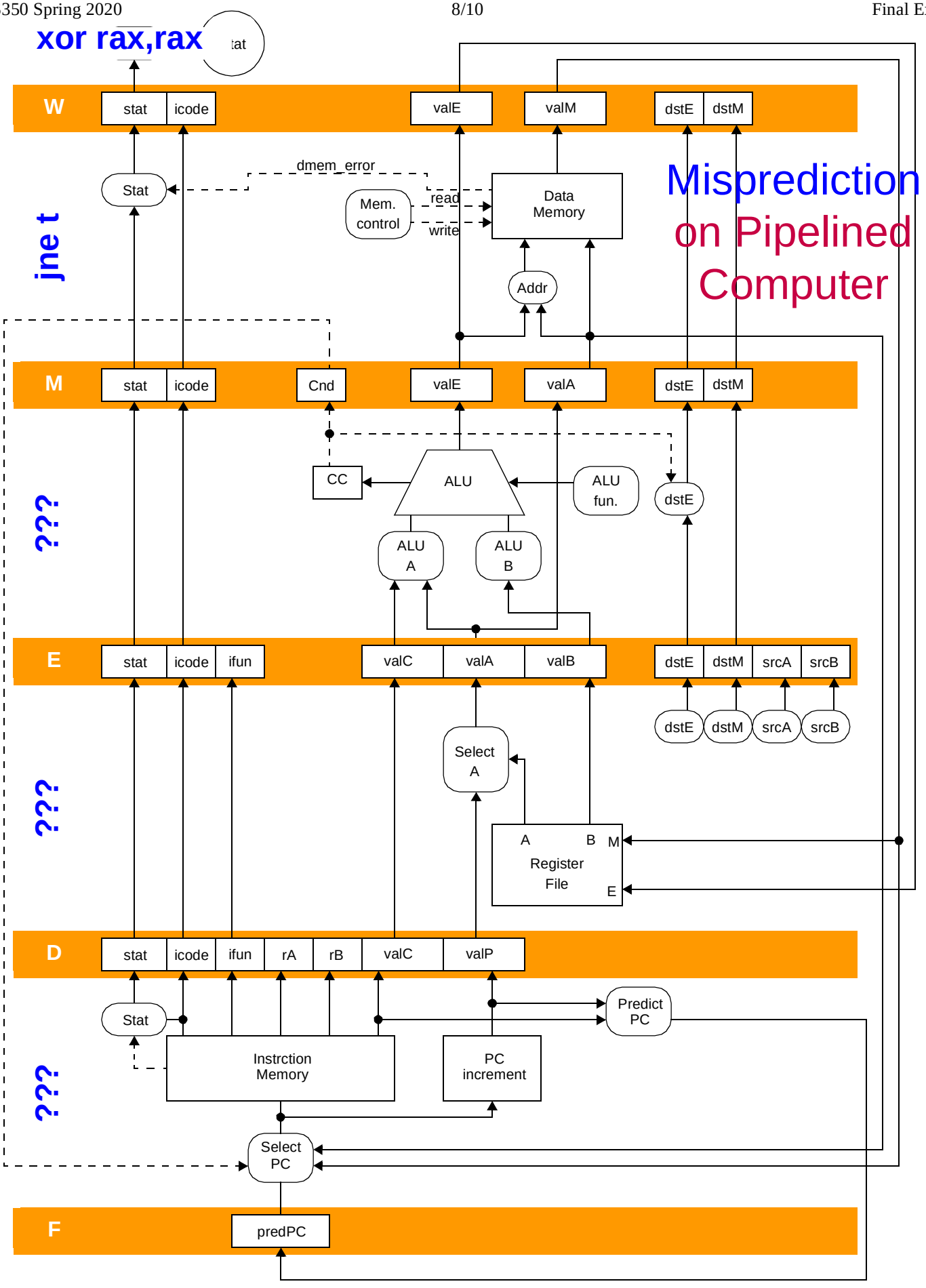
Instruction	icode	ifun	rA	rB	valC	valA	valB	dstE	dstM	srcA	valE	valM	Cnd
addq %rax,%rbx													
mrmovq 5(%rbx),%rax													
push %rax													
jmp loop													
call func													


Sequential Computer
Memory
Execute
Decode
Fetch
Write Back
Stat
Stat
dmem_error
valM
data out
Mem. control
read
write
Data Memory
Addr
Data
dmem_error
Cnd
valE
CC
ALU
ALU fun.
ALU A
ALU B
instr_valid
imem_error
valA
valB
dstE
dstM
srcA
valP
dstE
dstM
srcA
srcB
A
B
M
Register File
E
rsp
icode
ifun
rA
rB
valC
valP
Instrction Memory
PC increment
PC
PC
pIcode
pCnd
pValM
pValC
pValP

Problem 5 on Pipelined Computer with branch misprediction (26 points): Fill the table with values in **hexa** when the instructions below are executed on the pipepline. Do not write 0x. ***Refer to the textbook/lecture notes for icode, ifun, and register assignment.*** Assume branch is always taken.

```
0x000:  xorq %rax,%rax
0x002:  jne  t                  # Not taken
0x00b:  irmovq $1, %rax         # Fall through
0x015:  nop
0x016:  nop
0x017:  nop
0x018:  halt
0x019: t: irmovq $3, %rdx       # Target (Should not execute)
0x023:  irmovq $4, %rcx         # Should not execute
0x02d:  irmovq $5, %rdx         # Should not execute
```

clock	1		2		3		4		5	
xor rax,rax	**fetch**		**decode**		**execute**		**memory**		**writeback**	
	predPC	**A**	icode	**B**	icode		icode		icode	
			ifun	**C**	ifun		Cnd		valE	
			rA	**D**	rA		valE		valM	
			rB	**E**	rB		valA		dstE	
			valC		valC		dstE		dstM	
			valP	**F**	valA	**G**	dstM			
					valB	**H**				
					dstE	**I**				
					dstM					
					srcA	**J**				
					srcB	**K**				
jne t			**fetch**		**decode**		**execute**		**memory**	
			predPC	**Y**	icode	**L**	icode		icode	
					ifun	**M**	ifun		Cnd	
					rA	**N**	rA		valE	
					rB	**O**	rB		valA	
					valC	**P**	valC		dstE	
					valP	**Q**	valA		dstM	
							valB			
							dstE	**R**		
							dstM			
							srcA			
							srcB			
Z					**fetch**		**decode**		**execute**	
					predPC		icode	**S**	icode	
							ifun	**T**	ifun	
							rA	**U**	rA	
							rB	**V**	rB	
							valC	**W**	valC	
							valP	**X**	valA	
									valB	
									dstE	
									dstM	
									srcA	
									srcB	


xor rax,rax
jne t
???
???
???
W
stat
icode
valE
valM
dstE
dstM
Stat
dmem_error
Mem.
control
read
write
Data
Memory
Addr
Misprediction
on Pipelined
Computer
M
stat
icode
Cnd
valE
valA
dstE
dstM
CC
ALU
ALU
fun.
dstE
ALU
A
ALU
B
E
stat
icode
ifun
valC
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
Select
A
A
B
M
Register
File
E
D
stat
icode
ifun
rA
rB
valC
valP
Stat
Predict
PC
Instrction
Memory
PC
increment
Select
PC
F
predPC

Definition of Instructions

		ALU Op	reg to reg	imm to reg	load	store
		OPq rA, rB	rrmovq rA, rB	irmovq V, rB	mrmovq D(rB), rA	rmmovq rA, D(rB)
F	icode,ifun	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]
	rA,rB	rA:rB <- M1[PC+1]	rA:rB <- M1[PC+1]	rA:rB <- M1[PC+1]	rA:rB <- M1[PC+1]	rA:rB <- M1[PC+1]
	valC			valC <- M8[PC+2]	valC <- M8[PC+2]	valC <- M8[PC+2]
	valP	valP <- PC+2	valP <- PC+2	valP <- PC+10	valP <- PC+10	valP <- PC+10
D	valA, srcA	valA <- R[rA]	valA <- R[rA]			valA <- R[rA]
	valB, srcB	valB <- R[rB]			valB <- R[rB]	valB <- R[rB]
EX	valE	valE <- valB OP valA	valE <- 0 + valA	valE <- 0 + valC	valE <- valB + valC	valE <- valB + valC
	Cond code	Set CC				
M	valM				valM<- M8[valE]	M8[valE] <- valA
WB	dstE	R[rB] <- valE	R[rB] <- valE	R[rB] <- valE	R[rA] <- valM	
	dstM					
PC	PC	PC <- valP	PC <- valP	PC <- valP	PC <- valP	PC <- valP

		push	pop	jmp	call	ret	cmov
		pushq rA	popq rA	jXX Dest	call Dest	ret	cmovXX rA, rB
F	icode,ifun	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]	icode:ifun <- M1[PC]
	rA,rB	rA:rB <- M1[PC+1]	rA:rB <- M1[PC+1]				rA:rB <- M1[PC+1]
	valC			valC <- M8[PC+1]	valC <- M8[PC+1]		
	valP	valP <- PC+2	valP <- PC+2	valP <- PC+9	valP <- PC+9	valP <- PC+1	valP <- PC+2
D	valA, srcA	valA <- R[rA]	valA <- R[%rsp]			valA <- R[%rsp]	valA <- R[rA]
	valB, srcB	valB <- R[%rsp]	valB <- R[%rsp]		valB <- R[%rsp]	valB <- R[%rsp]	valB <- 0
EX	valE	valE <- valB + – 8	valE <- valB + 8		valE <- valB + –8	valE <- valB + 8	valE <- valB + valA
	Cond code			Cnd <- Cond(CC,ifun)			If ! Cond(CC,ifun) rB <- 0xF
M	valM	M8[valE] <- valA	valM<- M8[valA]		M8[valE] <- valP	valM<- M8[valA]	
WB	dstE	R[%rsp] <- valE	R[%rsp] <- valE		R[%rsp] <- valE	R[%rsp] <- valE	R[rB] <- valE
	dstM		R[rA] <- valM				
PC	PC	PC <- valP	PC <- valP	PC <- Cnd ? valC : valP	PC <- valC	PC <- valM	PC <- valP

Problem 6 on Cache Memory (18 points): The following problem concerns basic cache lookups.
- The memory is byte addressable.
- Memory accesses are to 1-byte words (not 4-byte words).
- Physical addresses are 13 bits wide.
- The cache is 2-way set associative, with a 4 byte line size and 16 total lines.

In the following tables, all numbers are given in hexadecimal. The contents of the cache are as follows:

2-way Set Associative Cache

	Line 0						Line 1					
Index	Tag	Valid	Byte0	Byte1	Byte2	Byte3	Tag	Valid	Byte0	Byte1	Byte2	Byte3
0	09	1	86	30	3F	10	00	0	99	04	03	48
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	2F	81	FD	09	0B	0	8F	E2	05	BD
3	06	0	3D	94	9B	F7	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	B0	39	D3	F7
6	91	1	A0	B7	26	2D	F0	0	0C	71	40	10
7	46	0	B1	0A	32	0F	DE	1	12	C0	88	37

The box below shows the format of a physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following: **CO** (block offset within the cache line), **CI** (cache index), **CT** (cache tag)

Bit 12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
Tag								**Index**			**Offset**	

For the given physical addresses **0x0123** and **0x08FC**, indicate the cache entry accessed and the cache byte value returned in hex. Indicate whether a cache miss occurs. If there is a cache miss, enter "**NA**" for "Cache Byte returned".

Write answers in hexa decimal.

Physical address **0x0123**												
Bit 12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
A	**B**				**C**				**D**			

Physical address **0x08FC**												
Bit 12	11	10	9	8	7	6	5	4	3	2	1	Bit 0
E	**F**				**G**				**H**			

Write the answers in hexa decimal.

Cache block offset:	0x	**I**
Cache set index:	0x	**J**
Cache tag:	0x	**K**
Cache hit? **0** for NO and **1** for YES:		**L**
Cache byte returned:	0x	**M**

Cache block offset:	0x	**N**
Cache set index:	0x	**O**
Cache tag:	0x	**P**
Cache hit? **0** for NO and **1** for YES:		**Q**
Cache byte returned:	0x	**R**