Name, SURNAME and ID ⇒

Middle East Technical University
Department of Computer Engineering

CENG 331

Section 2,3
Fall '2009-2010

FINAL

- **Duration: 120** minutes.
- **Exam:**
  - This is a **closed book**, **closed notes** exam.
  - No attempts of cheating will be tolerated. In case such attempts are observed, the students who took part in the act will be prosecuted. The legal code states that students who are found guilty of cheating shall be expelled from the university for **a minimum of one semester**!
  - **This booklet consists of *8* pages including this page. Check that you have them all!**

Question 1	
Question 2	
Question 3	
Question 4	
Question 5	
Total ⇒	

1 **(10 pts)**

Answer the following short questions.

- (4 pts) Your embedded processor has a one byte floating point representation with 3 bits for significand and 4 bits for exponent. What are the largest positive decimal number that can be represented? .
- (3 pts) Cache A is set-associative, Cache B is direct-mapped. Both A and B have data stores that are the same size. The total size of the tag's for A is _____ B s.
  - Same size as
  - Smaller
  - Larger
  - Depends on degree of associativity
- (3 pts) Consider a virtual memory system with a byte-addressable address space of 2^{32} locations. If the size of a page table entry is 4 bytes and each page is 4KB, how large is the page table?

2 **(10 pts)**

Given the following Y86 code, find all cases of data and control dependencies and tell which ones can be solved by forwarding unit available in the PIPE+ architecture.

```
        pushl   %ebp
        rrmovl  %esp, %ebp
        irmovl  $20,  %eax
        subl    %eax,  %esp
        pushl   %ebx
        mrmovl  8(%ebp),  %ebx
        mrmovl  12(%ebp),  %eax
        andl    %eax,  %eax
        jle     L38
        irmovl  $-8,  %edx
        addl    %edx,  %esp
        irmovl  $-1,  %edx
        addl    %edx,  %eax
        pushl   %eax
L38:    ...
        ...
        ...
```

3 **(30 pts)**

Consider the following C code:

```
int main(){
  return funny(3);
}
int funny(int n){
        if (n<=2)
                return 1;
        else
        return (funny(n-1)+funny(n-2));
}
```

The *funny* function compiles into the following executable code

```
0x08048367 <funny+0>:     push   %ebp
0x08048368 <funny+1>:     mov    %esp,%ebp
0x0804836a <funny+3>:     push   %ebx
0x0804836b <funny+4>:     sub    $0x4,%esp
0x0804836e <funny+7>:     cmpl   $0x2,0x8(%ebp)
0x08048372 <funny+11>:    jg     0x804837d <funny+22>
0x08048374 <funny+13>:    movl   $0x1,-8(%ebp)
0x0804837b <funny+20>:    jmp    0x80483a6 <funny+63>
0x0804837d <funny+22>:    sub    $0xc,%esp
0x08048380 <funny+25>:    mov    0x8(%ebp),%eax
0x08048383 <funny+28>:    dec    %eax
0x08048384 <funny+29>:    push   %eax
0x08048385 <funny+30>:    call   0x8048367 <funny>
0x0804838a <funny+35>:    add    $0x10,%esp
0x0804838d <funny+38>:    mov    %eax,%ebx
0x0804838f <funny+40>:    sub    $0xc,%esp
0x08048392 <funny+43>:    mov    0x8(%ebp),%eax
0x08048395 <funny+46>:    sub    $0x2,%eax
0x08048398 <funny+49>:    push   %eax
0x08048399 <funny+50>:    call   0x8048367 <funny>
0x0804839e <funny+55>:    add    $0x10,%esp
0x080483a1 <funny+58>:    add    %eax,%ebx
0x080483a3 <funny+60>:    mov    %ebx,-8(%ebp)
0x080483a6 <funny+63>:    mov    -8(%ebp),%eax
0x080483a9 <funny+66>:    mov    -4(%ebp),%ebx
0x080483ac <funny+69>:    leave
0x080483ad <funny+70>:    ret
```

When the program is run, the *funny* code shown above will be executed with arguments 3, 2 and 1. We call these instances of the function as *funny(3), funny(2)* and *funny(1).*

Assume that during the execution of *funny(3)*, before the execution of instruction at 0x08048367, %eax = 0, %ebx = 0, %ebp = 0x0fffffff, %esp = 0x0fffff7 and %eip = 0x08048367.

- (20 pts) During the execution of *funny(1)* what would be the state of the stack just before the instruction at 0x080483a6? Fill in the stack values as instructed in the previous item.

	Value	Description
0x0fffffff		
0x0ffffffb	3	argument for fib(3)
0x0ffffff7	0x08000000	return address from fib(3)
0x0ffffff3		
0x0fffffef		
0x0fffffeb		
0x0fffffe7		
0x0fffffe3		
0x0fffffdf		
0x0fffffdb		
0x0fffffd7		
0x0fffffd3		
0x0fffffcf		
0x0fffffcb		
0x0fffffc7		
0x0fffffc3		
0x0fffffbf		
0x0fffffbb		
0x0fffffb7		
0x0fffffb3		
0x0fffffaf		
0x0fffffab		
0x0fffffa7		
0x0fffffa3		
0x0fffff9f		
0x0fffff9b		
0x0fffff97		
0x0fffff93		
0x0fffff8f		
0x0fffff8b		
0x0fffff87		
0x0fffff83		

- (10 pts) During the execution of *funny(1)* what would be the state of the registers just before the instruction at 0x080483a6?

	Value
%eax	
%ebx	
%ebp	
%esp	
%eip	

4 **(20 pts)**

The following problem concerns the way virtual addresses are translated into physical addresses.

- The memory is byte addressable.
- Memory accesses are to 4-byte words.
- Virtual addresses are 20 bits wide.
- Physical addresses are 16 bits wide.
- The page size is 4096 bytes.
- The TLB is 4-way set associative with 16 total entries.

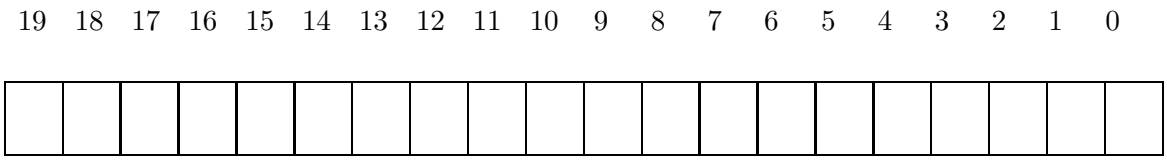
In the following tables, **all numbers are given in hexadecimal**. The contents of the TLB and the page table for the first 32 pages are as follows:

TLB			
Index	Tag	PPN	Valid
0	03	B	1
	07	6	0
	28	3	1
	01	F	0
1	31	0	1
	12	3	0
	07	E	1
	0B	1	1
2	2A	A	0
	11	1	0
	1F	8	1
	07	5	1
3	07	3	1
	3F	F	0
	10	D	0
	32	0	0

Page Table					
VPN	PPN	Valid	VPN	PPN	Valid
00	7	1	10	6	0
01	8	1	11	7	0
02	9	1	12	8	0
03	A	1	13	3	0
04	6	0	14	D	0
05	3	0	15	B	0
06	1	0	16	9	0
07	8	0	17	6	0
08	2	0	18	C	1
09	3	0	19	4	1
0A	1	1	1A	F	0
0B	6	1	1B	2	1
0C	A	1	1C	0	0
0D	D	0	1D	E	1
0E	E	0	1E	5	1
0F	D	1	1F	3	1

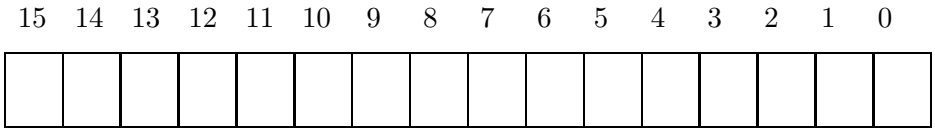
The box below shows the format of a virtual address. Indicate (by labeling the diagram) the fields (if they exist) that would be used to determine the following: (If a field doesn't exist, don't draw it on the diagram.)

VPO The virtual page offset
VPN The virtual page number
TLBI The TLB index
TLBT The TLB tag



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:

PPO The physical page offset
PPN The physical page number

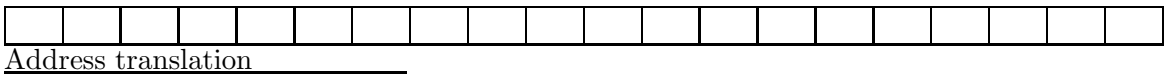


For the given virtual addresses, indicate the TLB entry accessed and the physical address. Indicate whether the TLB misses and whether a page fault occurs.

If there is a page fault, enter "-" for "PPN" and leave part C blank.

Virtual address: `7E37C`

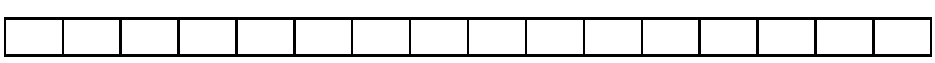
Virtual address format (one bit per box)



Address translation

Parameter	Value
VPN	0x
TLB Index	0x
TLB Tag	0x
TLB Hit? (Y/N)	
Page Fault? (Y/N)	
PPN	0x

Physical address format (one bit per box)



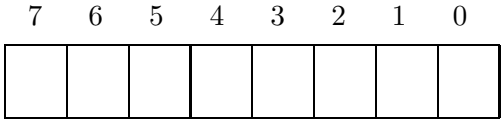
5 **(30 pts)**

We have the following cache system:

- The memory is byte addressable.
- Memory accesses are to **1-byte words** (not 4-byte words).
- Physical addresses are 8 bits wide.
- The cache is 2-way set associative, with a 2 byte line size and 8 total lines.
- The cache uses the LRU (Least Recently Used) replacement policy.
- Initially the cache is 'cold'.
- When all lines in a set are 'cold', the cache uses the first (or top) line of the set.
- Physical adresses are split into three in the order (cache tag, cache index, byte offset).

(a) (1 pt) The box below shows the physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

CO The block offset within the cache line
CI The cache index
CT The cache tag



(b) (14 pts) Draw the cache on the next page. Note that you will be filling the cache next, therefore read **(c)** before answering.

(c) (15 pts) The memory addresses accessed and the values they contain are shown below. Adresses are shown in binary, whereas values are shown in hexadecimal. The access order specifies the order an address is accessed. For instance, memory address 1100 0100 which contains the value 11 is accessed first. The second memory address accessed is 1100 1001 contained the value 66.

Access order	Address	Value
1	1100 0100	11
13	1100 0101	22
9	1100 0110	33
3	1100 0111	44
6	1100 1000	55
2	1100 1001	66
8	1100 1010	77
7	1100 1011	88
4	1100 1100	99
10	1100 1101	AA
11	1100 1110	BB
5	1100 1111	CC
12	1101 0000	DD
	1101 0000	EE

What will be the state of the cache after the 12'th memory access? That is what will be the values (tag, valid, data, etc.) in the cache? Show them on the cache structure drawn in the previous part.

Write down you answer below..