Full Name: James Le

CS-281, Introduction to Systems
Fall 2015
Take Home Final Exam

Due by Thursday, December 17, 2015, 3 p.m.

Instructions:

- Make sure that your exam is not missing any pages, then write your full name on the front.
- Write your answers in the space provided below the problem. If you make a mess, clearly indicate your final answer.
- Self-imposed *strict* time limit of 3 contiguous hours. You should not read any of the questions until you are ready to begin and you should stop after three hours of work.
- Open textbook, and open professor-provided handouts (Y86 Instruction set and Instruction semantics).
- Calculators may be used, but **only** for addition, subtraction, multiplication or division of decimal numbers.
- **You may not** use any other electronic aids in the completion of this test.
- The problems are of varying difficulty. The point value of each problem is indicated. If you feel any problem has multiple reasonable interpretations, clearly state your assumptions and solve it under those assumptions.
- After reading these instructions, affix your signature to this front page below to indicate your understanding and compliance with the above instructions and the academic integrity statement that follows.

On my honor, I acknowledge the above instructions and assert that I will follow these instructions and limit myself to three contiguous hours, to not speak with other students about any questions, and to otherwise act with honor and integrity in the completion of this exam.

Khanh

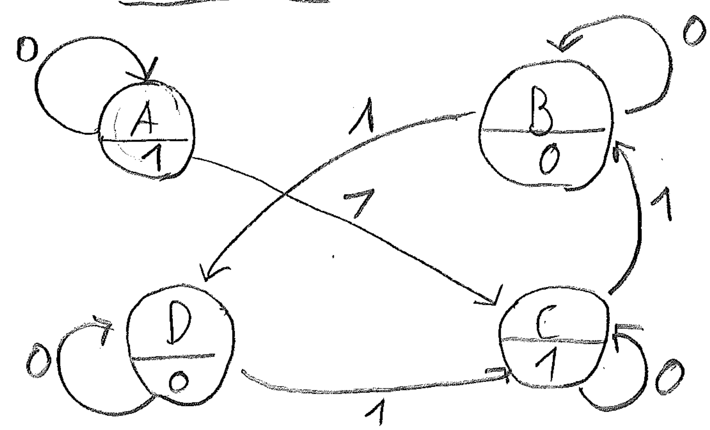
Khanh Nam "James" Le

Problem 1. (15 points):

Suppose, as a digital logic hardware designer, you are given the table below, which describes state transitions and outputs for a desired sequential circuit. In the table, Q represents the current state, the next state (Q') is given by the next two columns based on the value of binary input X, and the output Z associated with each current state is given in the final column. The four states of the system are represented symbolically as A, B, C, and D. Your goal is to design a sequential circuit for this system using 2 D flip-flops to store the current state. Note that, since the output Z is determined solely by the current state, this state transition table describes a Moore Machine.

Q	$X=0$	$X=1$	Z
A	A	C	1
B	B	D	0
C	C	B	1
D	D	C	0

A. Begin by drawing the Finite State Machine (Moore style) that corresponds to the described sequential circuit.



B. Assume that the state assignment for our two D flip flops (D_1D_0) is $A=00$, $B=01$, $C=10$, $D=11$. Fill in the truth tables below for both the output and the "next state" combinational logic:

D_1	D_0	Z
0	0	1
0	1	0
1	0	1
1	1	0

Z

D_0 \ D_1	0	1
0	1	1
1	0	0

$\rightarrow Z = \overline{D_0}$

D_1	D_0	X	D_1'	D_0'
0	0	0	0	0
0	0	1	1	0
0	1	0	0	1
0	1	1	1	1
1	0	0	1	0
1	0	1	0	1
1	1	0	1	1
1	1	1	1	0

D_0'

D_1 \ $D_0 X$	00	01	10	11
0	0	0	1	1
1	0	1	1	0

$\rightarrow D_0' = \overline{X} D_0 + X$

C. Using K-maps (and circling necessary *prime* implicants), design a minimal circuit for **only** next state variable D_0' and output Z, giving the minimal boolean expression in the resultant sum-of-products form. Show your work to the right of the truth tables above. Just give Boolean algebra expressions for D_0' and for Z, you **do not** have to draw the gate diagram.

$\underbrace{s}_{1}\ \underbrace{\text{exp}}_{k=4}\ \underbrace{\text{frac}}_{n=3}$

Problem 2. (12 points):

Assume you have an 8 bit floating point format with a sign bit ($s = 1$), $k = 4$, and $n = 3$. Fill in the required values below based on the following instructions for each field. Be careful to give me M and E, not exp and frac.

significand

exponent

- **Binary:** The 8 bit binary representation.
- **M:** The value of the significand. This should be a number of the form x or $\frac{x}{y}$, where x is an integer, and y is an integral power of 2. Examples include 0, $\frac{5}{4}$.
- **E:** The base 10 integer value of the exponent.
- **Value:** The base 10 numeric value represented.

  Note: you need not fill in entries marked with "—".

Description	Binary	M	E	Value
—	0 | 1 0 0 0 | 1 0 1	13/8	1	13/4
Smallest positive nornalized number	1 | 0 0 0 1 | 0 0 0	1	-6	1/64
Largest positive normalized number	1 | 1 1 1 0 | 1 1 1	15/8	7	–
Largest positive denormalized number	0 | 0 0 0 0 | 1 1 1	7/8	-6	–
—	1 | 0 1 1 1 | 0 1 1	11/8	2	$\frac{11}{2}$

$\text{Bias} = 2^{k-1} - 1 = 2^3 - 1 = 7$

exp $\in \{0, \dots, 15\}$

Normalized (exp $\in \{1, \dots, 14\}$)

Smallest E : -6 (exp $= 1$)

Largest E : 7 (exp $= 14$)

Smallest M : 1.000

Largest M : 1.111

Smallest nor. : $1.0 \times 2^{-6} = 0.015625$

Largest nor. : 1.111×2^{7}

Denormalized

- Exp = 0
- E = 1 − Bias = 1 − 7 = −6
- Largest M : 0.111
- Largest Denormalized :

0.111×2^{-6}

The following problem concerns the following, low-quality code:

```
void foo(int x)
{
  int a[3];  ✓
  char buf[4]; ✓
  a[0] = 0xF0F1F2F3; ✓
  a[1] = x; ✓
  gets(buf);
  printf("a[0] = 0x%x, a[1] = 0x%x, buf = %s\n", a[0], a[1], buf);
}
```

In a program containing this code, procedure foo has the following disassembled form on an IA32 machine:

```
foo:
  pushl   %ebp                 save %ebp on stack
  movl    %esp,%ebp
  pushl   %ebx                 save %ebx
  subl    $32,%esp             allocate 32 bytes on stack

  movl    8(%ebp),%eax         retrieve canary
  movl    $0xf0f1f2f3,-16(%ebp)  store on stack // a[0]
  movl    %eax,-12(%ebp)       store on stack

  leal    -20(%ebp),%ebx       compute buf as %ebp-20
  movl    %ebx,(%esp)          store buf at top of stack
  call    80484a4 <_init+0x54>   # gets   call "gets"

  movl    %ebx,12(%esp)        store buf at top of stack
  movl    -12(%ebp),%eax       retrieve canary
  movl    %eax,8(%esp)
  movl    -16(%ebp),%eax       retrieve canary
  movl    %eax,4(%esp)
  movl    $0x80490ec,0(%esp)     # address of format string
  call    8048494 <_init+0x44>   # printf   call "printf"

  addl    $32,%esp             deallocate stack space
  popl    %ebx                 restore %ebx
  movl    %ebp,%esp
  popl    %ebp                 restore %ebp
  ret                          return
```

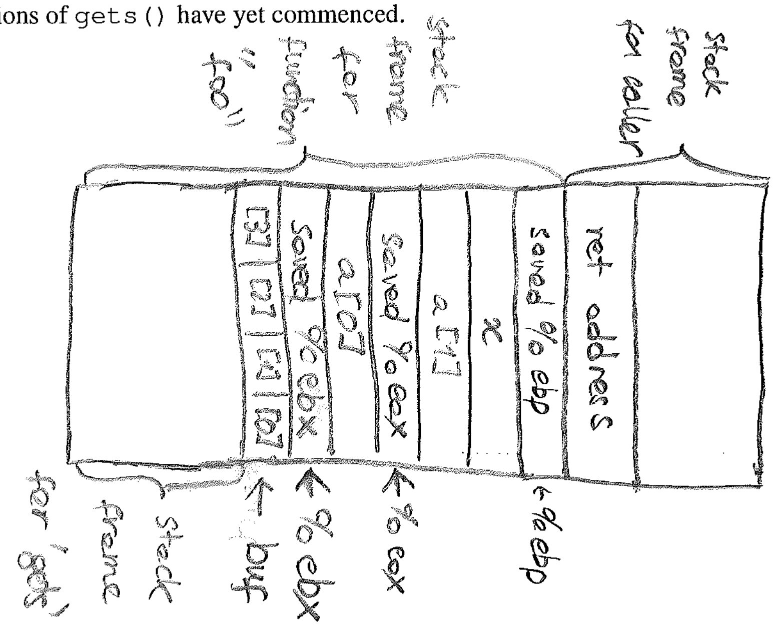
For the following questions, recall that:

- gets is a standard C library routine, like we saw in the buffer lab, with no bounds checking.
- IA32 machines are little-endian.
- C strings are null-terminated (i.e., terminated by a character with value 0x00).
- Characters '0' through '9' have ASCII codes 0x30 through 0x39.

Problem 3. (12 points):

Consider the case where procedure `foo` is called with argument `x` equal to `0xE3E2E1E0`, and we type "`123456789`" in response to `gets`.

A. Draw a detailed picture of the stack (using a 4-byte wide representation) with a frame for foo labeled with any known locations for variables and/or registers and values, up to the point where `gets()` has *just been called*, but no instructions of `gets()` have yet commenced.



B. Fill in the following table indicating which program values are/are not corrupted by the response from `gets`, i.e., their values were altered by some action within the call to `gets`.

Program Value	Corrupted? (Y/N)
`a[0]`	~~N~~ Y
`a[1]`	N
`a[2]`	N
`x`	N
Saved value of register `%ebp`	N
Saved value of register `%ebx`	Y

C. What will the `printf` function print for the following:

- `a[0]` (hexadecimal): 0x f0 f1 f2 39
- `a[1]` (hexadecimal): 0x e3 e2 e1 e0
- `buf` (ASCII): 34 33 32 31

$S = \frac{C}{B \times E}$ $t = m - (s + b)$ $s = \log_2(S)$ $b = \log_2(B)$

Problem 4. (6 points):

The following table gives the parameters for a number of different caches, where m is the number of physical address bits, C is the cache size (number of data bytes), B is the block size in bytes, and E is the number of lines per set. For each cache, determine the number of cache sets (S), tag bits (t), set index bits (s), and block offset bits (b).

Cache	m	C	B	E	S	t	s	b
1.	32	1024	4	4	64	24	6	2
2.	32	1024	4	256	1	30	0	2
3.	32	1024	8	128	1	29	0	3

Problem 5. (6 points):

In our Y86 programs, we encounter many cases in which we want to add a constant value to a register. This requires two instructions, an `irmovl` followed by an `OPl` where the ifun indicates addition. Suppose we want to add a new instruction `iaddl` with the following format:

`addl` V, rB

C	0	F	rB	← V →

This instruction adds the constant value V to register rB. Using Register Transfer Language, specify the six stages, from *Fetch* to *PC Update*, to implement this instruction.

irmovl V, rB

3	0	F	rB	← V →

addl rA, rB

6	0	rA	rB

⇒ Dataflow by Stages for "iaddl"

Stage	
Fetch	icode : ifun ← $M_1[PC]$ rA : rB ← $M_1[PC+1]$ valC ← $M_4[PC+2]$ valP ← PC + 8
Decode	valA ← R[rA] valB ← R[rB]
Execute	valE ← valC + valB + valA CC: ZF ← 0, SF ← 0, OF ← 0
Memory	N/A
Write Back	R[rB] ← valE
PC Update	PC ← valP

addl V, rB

C	0	F	rB	← V →

Problem 6. (14 points):

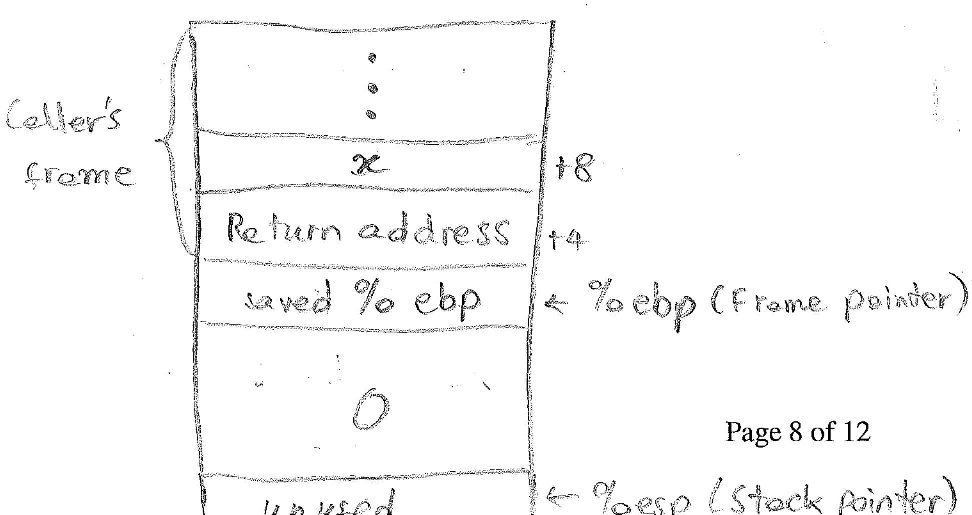
Consider the function baz disassembled below. Assume that baz is invoked from a main function in a manner consistent with the C code:

```
int main() {
  int x = 5; ✓
  int y = baz(5); ✓
  printf("baz(%d) = %d\n", x, y); ✓
  exit(0);
}
```

```
Dump of assembler code for function baz:
=> 0x0804844f <+0>:   push   %ebp   save old %ebp
   0x08048450 <+1>:   mov    %esp,%ebp   set %ebp as frame pointer
   0x08048452 <+3>:   sub    $0x8,%esp   allocate 8 bytes on stack
   0x08048455 <+6>:   cmpl   $0x0,0x8(%ebp)   compare 0 : x (jump if x > 0)   main's (8+%ebp)
   0x08048459 <+10>:  jg     0x8048462 <baz+19>
   0x0804845b <+12>:  mov    $0x0,%eax   get x' = 0 (Jump anyway)
   0x08048460 <+17>:  jmp    0x8048479 <baz+42>
   0x08048462 <+19>:  mov    0x8(%ebp),%eax   get x'
   0x08048465 <+22>:  sub    $0x2,%eax   x'-2
   0x08048468 <+25>:  mov    %eax,(%esp)   store (x'-2) on stack (on esp)
   0x0804846b <+28>:  call   0x804844f <baz>   call "baz"
   0x08048470 <+33>:  mov    %eax,-0x4(%ebp)   store x' on stack (4 down of ebp)
   0x08048473 <+36>:  mov    0x8(%ebp),%eax   get x'
   0x08048476 <+39>:  add    -0x4(%ebp),%eax   return value = x'+(x'-2)
   0x08048479 <+42>:  add    $0x8,%esp   deallocate 8 bytes from stack
   0x0804847c <+45>:  pop    %ebp   restore %ebp
   0x0804847d <+46>:  ret   return.
```

Your job is to draw, in detail, all stack frames corresponding to the execution of this program, beginning with the stack frame for main at the time of the call to baz. Also indicate the result of the printf in main at the conclusion of the program. Do not worry about indicating the completion of stack frames as functions complete. Show the limits of each stack frame and, within the boxes in the stack, indicate the words used in the stack frame. If there is a known value, enter that. If the value is an address, give a short qualitative description (e.g. "ret addr to main", or "main's %ebp").

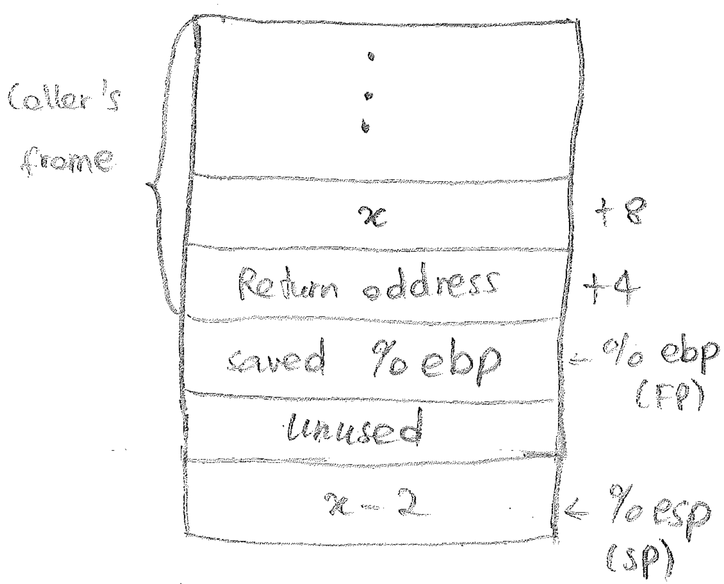
Case I - The argument ~~for baz~~ x is less than 0. There will be no call to baz, and there is only 1 stack frame of the main function.



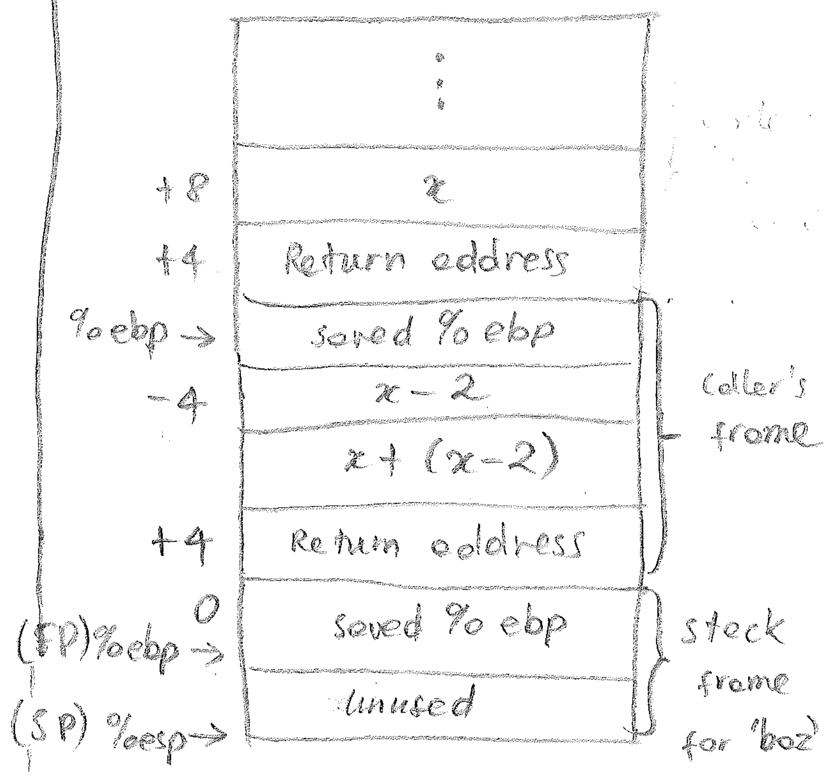
Additional space for Problem 6.

Case II – The argument x is bigger than 0. There are 2 displayed frames:

* Frame 1 – Just before call to 'boz'



* Frame 2 – In body of 'boz'



Main function will print out:

(5, 8)

to the screen

Problem 7. (6 points):

Consider the following C program:

```
#include <sys/wait.h>

main() {
  int status;
  int pid;

  printf("%s\n", "Hello");
  pid = fork();
  printf("%s\n", (pid != 0) ? "parent" : "child"); // !fork()

  if (wait(&status) != -1)
    printf("%d\n", WEXITSTATUS(status));

  printf("%s\n", "Bye");

  exit(2);
}
```

Recall the following:

- Function `fork` returns 0 to the child process and the child's process Id to the parent.
- Function `wait` returns -1 when there is an error, e.g., when the executing process has no child.
- Macro `WEXITSTATUS` extracts the exit status of the terminating process.

Give two valid output results of this program. *Hint: there are several correct solutions.*

Output 1: Hello 0 1 Bye 2 Bye

Output 2: Hello 1 Bye 0 2 Bye

Problem 8. (15 points):

Consider a memory system with a main memory where $w = 8$ so there are $2^8 = 256$ bytes of main memory. Further assume that the memory contents have the values as given in the table on the next page, wherein the left hex digit of an address gives the row of the table and the right hex digit of an address gives the column.

We introduce a level-1 cache where there are four bytes per cache block, the cache has 4 sets, and there are two lines per set (aka, a 2-way set associative cache), and the cache uses an LRU replacement policy on evictions. The cache has initial contents as given below. Your task is to simulate operation of the cache by translating the sequence of address references below, noting *Hit* or *Miss* below the address reference, and updating the cache picture with new values for any updated cache line by writing the new values below any previous cache line values in the cache contents picture below. Assume all address references are *load* operations and access a single, so you do not need to worry about write policies, nor an access crossing a cache line boundary.

$E = 2$

$B = 4 \rightarrow b = 2$

$S = 4 \rightarrow s = 2$

$t = 4$

CT	CT	CT	CT	CI	CI	CO	CO
7	6	5	4	3	2	1	0

2-way Set Associative Cache						
Index	Tag	Valid	Bytes(0-3)	Tag	Valid	Bytes(0-3)
0	0	0	00 00 00 00	0	0	00 00 00 00
1	3	1	02 B3 8F B6	~~0~~ ~~8~~ A	~~0~~ 1	~~00 00 00 00~~ A2 4D
2	~~0~~ A	~~0~~ 1	~~00~~ 00 00 00 5D	8	1	EF 09 86 2A
3	B	1	C5 EC 76 4E	~~0~~ 3	~~0~~ 1	00 00 ~~00~~ 00 19

Address Reference Sequence						
Addr	0x87	0x88	0xA5	0xA8	0x36	0x3E
Hit?	N	Y	N	N	Y	N

Memory Contents																
Address	Bytes x0 to xF relative to address															
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x00	AA	03	3E	CD	38	16	7B	ED	5A	8E	4C	DF	18	FB	B7	12
0x10	A9	76	2B	EE	BC	91	D5	92	80	BA	9B	F6	48	16	81	0A
0x20	75	F7	3F	C6	9E	3A	0F	DA	00	4C	B6	A8	92	04	E5	2E
0x30	E0	22	$\boxed{19}$	3A	02	B3	8F	B6	25	31	E1	02	18	09	73	02
0x40	86	B8	F0	C6	AA	29	AE	16	76	46	80	6E	13	EA	A8	66
0x50	04	2A	32	6A	B1	86	56	0E	96	30	47	F2	F8	1D	42	30
0x60	2F	7E	3D	A8	27	95	A4	74	07	11	6B	D8	C7	B7	AF	C2
0x70	D6	A4	89	92	FD	FE	D6	DA	DE	D5	CD	4A	7C	68	3A	1A
0x80	ED	32	0A	$\boxed{A2}$	BF	80	1D	FC	EF	09	86	2A	25	44	6F	1A
0x90	1E	C2	AE	60	5C	3E	DF	F2	25	CF	84	DA	F1	6B	DC	DE
0xA0	$\boxed{5D}$	$\boxed{4D}$	F7	DA	69	C2	8C	74	A8	CE	7F	DA	FA	93	EB	48
0xB0	61	C6	5E	74	03	97	BA	62	F8	11	72	12	C5	EC	76	4E
0xC0	17	52	75	2C	62	89	EF	18	BB	7D	8C	7C	26	57	7F	C2
0xD0	54	9E	1E	FA	DC	81	B2	14	B6	1F	7B	44	10	F5	B8	2E
0xE0	14	9A	0D	4A	C8	1D	E6	6E	F3	38	F3	5C	6C	8F	BD	A8
0xF0	32	21	1C	2C	22	C2	DC	34	BA	DD	37	D8	E7	A2	39	BA

Add. 0x87:

7	6	5	4	3	2	1	0
1	0	0	0	0	1	1	1

- CO: 0x3 (11)
- CI: 0x1 (01)
- CT: 0x8 (1000)
- Cache Hit: N

Add 0x88:

7	6	5	4	3	2	1	0
1	0	0	0	1	0	1	0

- CO: 0x2 (10)
- CI: 0x2 (10)
- CT: 0x8 (1000)
- Cache Hit: Y
- Cache byte returned: 0x86

Add. 0xA5:

7	6	5	4	3	2	1	0
1	0	1	0	0	1	0	1

- CO: 0x1 (01)
- CI: 0x1 (01)
- CT: 0xA (1010)
- Cache Hit: N

Add. 0xA8:

7	6	5	4	3	2	1	0
1	0	1	0	1	0	0	0

- CO: 0x0 (00)
- CI: 0x2 (10)
- CT: 0xA (1010)
- Cache Hit: N

Add. 0x 36:

7	6	5	4	3	2	1	0
0	0	1	1	0	1	1	0

- CO: 0x2 (10)
- CI: 0x1 (01)
- CT: 0x3 (0011)
- Cache Hit: Y
- Cache Byte returned: 0x8F

Add. 0x 3E:

7	6	5	4	3	2	1	0
0	0	1	1	1	1	1	0

- CO: 0x2 (10)
- CI: 0x3 (11)
- CT: 0x3 (0011)
- Cache Hit: N