

## CS 110 Midterm Solutions

---

### Problem 1 – Hand Assembly (16 points)

Convert the following instructions from assembly to machine language.

Assembly	Machine Code (in HEX)
MOVE.L (A0)+, D3	\$2618
EORI.L #-1, D3	\$0A83
ADDQ.L #1, D3	\$5283
SUB.L D3, -4(A0)	\$97A8 FFFC

Bonus (+1 point): What single-word instruction will produce essentially the same effect as executing the preceding series of instructions?

*...there is nothing exact. The instructions multiply the long stored at (A0) by 2 and add 4 to A0. The closest instruction is LSL (A0)+, which multiplies the word stored at (A0) by 2 and adds 2 to A0. Because this solution stretches the “essentially” part of the question a bit far, I accepted any answer that showed knowledge that the “OR” and “ADDQ” did 2’s complement.*

### Problem 2 – Hand Disassembly (16 points)

Convert the following instructions from machine language to assembly.

Machine Code	Hint	Assembly
\$0000 0000	ORI	ORI.B #0, D0
\$3585 D803	MOVE	MOVE.W D5, 3(A2, A5.L)
\$BEEF AAAA	CMPA	CMPA.W \$AAAA(A7), A7
\$4498	NEG	NEG.L (A0)+

### Problem 3 – Condition Codes and Branching (20 points)

Given the following initial instructions:

```
CLR.L D0
CLR.L D1
MOVE.W #$65, D0
MOVE.W #$FC, D1
```

a) Give the values of D0 and D1 after the instructions execute (give a 32-bit value in hex):

D0: 0000 0065

D1: 0000 00FC

b) Show the condition codes resulting after the following instructions (each should be either a “0” or a “1”) and tell whether the given branches would be taken (“y” or “n”):

Instruction	N	Z	V	C	BHI	BGT
CMP.L D0, D1	0	0	0	0	y	y
CMP.W D0, D1	0	0	0	0	y	y
CMP.B D0, D1	1	0	0	0	y	n
CMP.B D1, D0	0	0	0	1	n	y
ADD.B D0, D0	1	0	1	0	y	y

### Problem 4 – Performance and Tables (28 points)

The following piece of code sets up a table of the first 16 powers of three in order to try to speed up calculations in another part of the program (thus, the “0<sup>th</sup>” element in the table is  $3^0 = 1$ , and the “15<sup>th</sup>” element is  $3^{15} = 14,348,907$ ).

a) Fill in the following:

- Fetch reads – the number of reads to fetch the instruction.
- Execute reads – the number of reads to load the operands of the instruction.
- Execute writes – the number of writes to store the result of the instruction.

			Fetch reads	Exec reads	Exec writes
ThreesTable	DS.L	15			
	MOVE.L	#0, D0	3	0	0
	MOVE.L	#1, D1	3	0	0
	LEA	ThreesTable(A5), A0	2	0	0
Loop	MOVE.L	D1, 0(A0, D0)	2	0	2
	ADD.L	#4, D0	3	0	0

```

MULU.L #3, D1           3           0           0
CMP.L #60, D0           3           0           0
BNE Loop               1 or 2       0           0
RTS

```

**b)** Given that some value  $n$  ( $0 \leq n < 15$ ) is stored in D0, write the code to access  $3^n$ . In other words, if D0 contains 1, you want to retrieve the “1<sup>st</sup>” element in the table (which should be 3). If D0 contains 0, you want to retrieve the “0<sup>th</sup>” element. Place the result in D1. You need not worry about performance in this problem (though you shouldn’t need more than a few instructions).

```

LEA ThreesTable(A5), A0
LSL.L #2, D0
MOVE.L 0(A0, D0), D1

```

**c)** The code given to initialize the table is quite inefficient both in terms of speed and in terms of space. There are multiple ways to optimize it (looking at it quickly, I see about five). Name two optimizations that could be applied to this code. The optimizations should be different in nature (don’t just say you could apply one optimization to multiple instructions). You may not use more than the space given to describe your optimizations.

Many answers were appropriate, including:

- MOVE.L #0, D0 ==> CLR.L D0
- ADD/MOVE ==> ADDQ/MOVEQ
- use post-increment, not indexed addressing
- use byte operations on D0
- ...

**d)** This code is quite silly because it goes through the work of computing values at runtime that we actually know at “assembly time.” How could we create a similar table that needs no runtime initialization (you may not use this as one of your optimizations for part c)? You may not use more than a few lines for your answer.

```

ThreesTable DC.L 1, 3, 9, 27, ..., 14348907

```

### Problem 5 – Addressing Modes (20 points)

Given the following initial conditions, show the destination address and 32-bit value (in hex) after the execution of these instructions.

Notes:

- There is an ACSII chart on page C-5 of the M68000 reference manual if you need it.
- “doh” (defined below) is  $-\$20$  off of A5.
- **Every instruction is executed independently** with the following initial conditions:

0000 0000 to	????
0000 E100	
0000 E102	E11A
0000 E104	0000
0000 E106	E114
0000 E108	BABA
0000 E10A	600D
0000 E10C	F00D
0000 E10E	E10C
0000 E110	B000
0000 E112	AAAA
0000 E114	5000
0000 E116	BEEE
0000 E118	BAD1
0000 E11A to	DEAD
FFFF FFFF	

D0	0000 0004
D1	00FF 7000
A1	0000 7100
A2	0000 E104
A3	0000 E0F8
A4	0000 E11A
A5	0000 1000
PC	0000 2000

```
doh      DS.L  4
m      DC.B  "DONUTS"
woohoo  EQU   $12
cerealPort EQU  $0000 E10C
```

	Dst EA in hex	32-bit value in hex beginning at dst EA.
MOVE.L mmmm(PC), doh(A5)	0000 0FE0 or 0000 0FEC	444F 4E55
MOVE.W mmmm(PC, D0.L), cerealPort	0000 E10C	5453 E10C
MOVE.B -(A4), woohoo(A3)	0000 E10A	D10D F00D
MOVE.W #woohoo, (A2)	0000 E104	0012 E114
MOVE.L woohoo(A1, D1.W), woohoo	0000 0012	AAAA 5000