Addendum

Some additional thoughts on the 68K Disassembler Project

0. How to start:

Right now, this project might seem like a pretty formidable task. It certainly has been for many students in the past. It is also one of the best exercises that I, or any other instructor teaching computer architecture, can think of to show you how it all really works. Also, when you're out interviewing for a job, take along this project to show that you really do know some 68000 assembly language programming.

Where do you begin? First, forget about what language your programming this in. I could ask you to write it in C++, C# or Java and it would be just as challenging. Work on the algorithm. How do you decode an instruction? Focus on that. Also, as soon as possible create your own test program. Make it a killer. Don't make it easy, that won't stress your code enough. For example, If your program can disassemble itself, then my test program is a snap.

A killer test program does not have to be a gigantic test program that covers every possible combination of op-codes and addressing modes. Use your insight into how your program works to create test cases that will stress it.

Some hints for creating your test program:

1. Locate it in different regions of memory. If it runs in one place and crashes your code in another, then you have lurking bugs.
2. Insert random words of data in between valid instructions.
3. Add instructions and effective addressing modes that you are not required to decode. For example, add an effective address mode that you are required to decode with an op-code that you are not required to decode, and vice versa.
4. Many instructions have different forms under special conditions, test all these conditions. For example, the branch instructions have a different form if the branch is short (8-bit) versus long (16-bit).
5. Make sure you test the arithmetic and logic instructions in all of their variations. For example:

ADD.W D0,<EA> and ADD.W <EA>,D0

create special challenges for correctly decoding and printing.

6. Practice good Software Engineering test methodologies. For example, *regression testing* is a great way to make sure that fixing one bug doesn't introduce another one.
7. Keep good logs of the errors, how to reproduce them and who owns the repair.

Once you begin to build your program it is a good idea to build it in stages, rather than go for the whole enchilada in one shot. By that I mean that the worst scenario of all is to be integrating your code at 4am on the day its due. As soon as a skeleton of the I/O code is finished, start to disassemble your program. Assume for a moment that your program can't disassemble anything yet. If you've structured it properly, then you should report DATA for every word of memory. That's OK. You're actually quite far along. Now, add something simple, like NOP. You should see NOP in you output, and the rest are DATA statements. If you build it up this way, and always, always keep tight control of what version you are working on, then you can progress in well-managed stages.

Don't underestimate the importance of keeping track of what version you are working on. It is know that lots of project teams crash and burn because they handed in the wrong version to me, or they lost two days of work because the wrong version was purged. Save everything until you get your grade. Every new version goes in a new folder. In the real world, if you accidentally kill a source file you can be shown the door. **You are highly suggested to use Bitbucket or Github, and make your project private.**

Don't underestimate the importance of team dynamics. The most common reason is that one or more of the team members fail to meet their commitments and the other team members are stuck. Other situations involve teams where there are clashes of wills and a failure to compromise. In any case, before you formally organize as a team, you should read the **handout provided by Professor Freytag (https://canvas.uw.edu/courses/1408528/files/67695508/download?wrap=1)** and discuss it among yourselves. If you can't come to an agreement about how you will manage yourselves without a boss standing over you with a whip, then maybe your not in the right team.

I. Project:

I strongly urge you to organize your team with one team member as the designated tester and one team member as the keeper of the subroutines and API's. The tester should be concerned with all the QA issues of the project. This means tasks such as bounds testing, human interface, code inspection, coding standards, appearance of the final document.

This is a major assembly language project. It has several important coding challenges. Among them are:

1- Inputting numbers in ASCII format.

2- Converting the ASCII numbers to binary and converting the binary numbers to ASCII.

3- Building look-up tables and using indexed addressing modes.

4- Understanding the instruction coding of a mainstream microprocessor.

5- Error and bounds checking.

6- Developing competence in a wider group of instructions.

Almost all disassemblers that I'm familiar with are built using some type of a tables. The table could be a look-up table or a jump table. In developing your program you should begin by studying the op-codes and building a map. Start with the simple op-codes, like ADD.B D0,D1 and see how they work.

Once you can decode and display simple op codes and operands, then add more to your table. It is much better to get a sub-set working properly and then, as your program comes together, add more of the instructions and addressing modes until you've got it all.

The biggest error that students seem to make with this project is not to completely test their disassembler. Don't test your code with a test program that you know it will pass, test it with one that stresses it as much as possible. I would go so far as to recommend that your test program is one of the first things that you create. You are much better off having your program decode a subset of the required instructions, but doing that subset well, then to claim that you decode it all, and have it fail 25% of the time.

There are no hidden mysteries to this project, you already know at the outset the instructions and addressing modes that you are responsible for.

Please don't even think about any assembly language coding until you understand how to decode an instruction and then build up the complete instruction line so that you can output it to the screen. It is OK to try small test algorithms to make sure your ideas are sound, but don't start hacking code until you are clear on what you are doing.

It is unnecessary to do any symbolic decoding. Just display the displacement field or memory address.

II. Instruction decoding

1. Consider the op-codes for most instructions. If you look at the 4 most significant bits, (DB12-DB15) you can group them into 16 categories. Consider the following table:

Bits 15 through 12	Operation
0000	Bit manipulation/MOVEP/Immediate
0001	Move Byte
0010	Move Long
0011	Move Word
0100	Miscellaneous
0101	ADDQ/SUBQ/Scc/DBcc
0110	BSR,BRA,Bcc
0111	MOVEQ
1000	OR/DIV/SBCD
1001	SUB/SUBX
1010	Unassigned
1011	CMP/EOR
1100	AND/MUL/ABCD/EXG
1101	ADD/ADDA/ADDX
1110	Shift/Rotate
1111	Special/Reserved

2. Suppose we read an op code, 1011XXXXXXXXXXX, where X can be anything. The first task is to isolate the pattern 1011 as the rightmost bits in a register with all zeroes in every other bit position. The reason is that we can make use of one of the addressing modes of the 68K, address register indirect with index and displacement. The example program should show you what I mean:

```
****************************************
* Example of using a jump table to decode an instruction
*
****************************************


* System equates


stack           EQU     $A000
example         EQU     %1101111001100001    * I made up bits 0 to 11
shift           EQU     12                   * Shift 12 bits


* Program starts here


                ORG     $400
start           LEA     stack,SP        *Load the SP
                LEA     jmp_table,A0    *Index into the table
                CLR.L   D0              *Zero it
                MOVE.W  #example,D0     *We'll play with it here
                MOVE.B  #shift,D1       *Shift 12 bits to the right
                LSR.W   D1,D0           *Move the bits
*
* Consider the next instruction. Why do we have to multiply the index
```

* by 6? How many bytes does a single jump table entry require?

```
        MULU    #6,D0     *Form offset
        JSR     0(A0,D0)  *Jump indirect with index

jmp_table   JMP     code0000
        JMP     code0001
        JMP     code0010
        JMP     code0011
        JMP     code0100
        JMP     code0101
        JMP     code0110
        JMP     code0111
        JMP     code1000
        JMP     code1001
        JMP     code1010
        JMP     code1011
        JMP     code1100
        JMP     code1101
        JMP     code1110
        JMP     code1111
```

*The following subroutines will get filled in as you decode the instructions . For *now, just exit gracefully.

```
code0000    STOP    #$2700
code0001    STOP    #$2700
code0010    STOP    #$2700
code0011    STOP    #$2700
code0100    STOP    #$2700
code0101    STOP    #$2700
code0110    STOP    #$2700
code0111    STOP    #$2700
code1000    STOP    #$2700
code1001    STOP    #$2700
code1010    STOP    #$2700
```

* Next we put in the next level of decoding. I just stuck this BRA

* instruction here so it would look different. If this was your real

* code, you would decode to the next level. Perhaps this would be

* another jump table to the 8 possible op-codes at the next level.

```
code1011    BRA     code1011
```

```
code1100    STOP    #$2700
code1101    STOP    #$2700
code1110    STOP    #$2700
code1111    STOP    #$2700

        END   $400
```

OK, so we can index to a subroutine through the jump table. We also know that there are 4 possible op-codes that have 1011 as the 4 most significant bits. They are:

a. CMP b. CMPA c. EOR d. CMPM

Where do we go from here? Now we have to consider bits 6,7 and 8. This gives us 8 possible combinations. Look at the following table:

Bits 6,7,8	Instruction
000	CMP.B
001	CMP.W
010	CMP.L
011	CMPA.W
100	EOR.B
101	EOR.W
110	EOR.L
111	CMPA.L

Since we don't have to decode the CMPM instruction, this is as far as we would have to take it. We now know that the op-code is, the size of the operation, bits 9,10 and 11 give us the data or address register involved in the operation. The last piece of information that we need is provided by bit positions 0 through 5, the effective address bit field.

If we had to decode CMPM, then we would have to go a bit further. Bits 6,7 and 8 of the CMPM instruction could be 100, 101 or 110. So how do we distinguish it from EOR.B,

EOR.W or EOR.L? The answer is that CMPM always has bits 3,4 and 5 as 001. If we look at the effective address tables, we see that 001 refers to a mode 1 operation, the effective address is an address register. But, checking the EOR instruction, we see that

EOR.X Dn,<ea> does not allow <ea> to be an address register, so there is no ambiguity, we just had to check another level down to make sure.

This general algorithm will work for most cases. There are some special case instructions, such as MOVE, that you'll have to handle differently.

III. Effective address calculation

The 16-bit op-code provides all that you need to know about decoding an instruction. Remember that an instruction could contain as many as 5 16-bit words. For example, the instruction:

MOVE.L $AAAAAAAA,$55555555

moves the contents of memory location $AAAAAAAA to memory location $55555555.

This instruction in memory looks like: 23F9, AAAA,AAAA,5555,5555

That's a total of 5 words of memory. If the instruction started at memory location $1000, then the next instruction would have to start at memory location $100A. Thus, once you decode the op-code, the next task is to figure out the effective address so you can advance your instruction pointer to the next instruction in memory.

This is why someone should have the task of figuring out the effective address data. It is really crucial to making the disassembler work.

IV. Hints on decoding immediate instructions

How to avoid a potential problem with the immediate addressing mode

"Thanks to a former CSS422 student, Chuck Bond",

There is a potential nasty bug that can get into your disassembler. The problem has to do with the instructions that have a source operand that represents immediate data. These instructions would be ADDI, SUBI, CMPI, ORI, ANDI and so forth. All of these instructions share a common trait that the source operand is implied by the instruction. The destination operand for these instructions is an effective address, defined by the six-bit effective address field.

Let's look at some real code to see what the problem is. Consider the following instructions:

00000400 067955550000AAAA ADDI.W #$5555,$0000AAAA

00000408 06B9AAAA55550000FFFE ADDI.L #$AAAA5555,$0000FFFE

00000412 0640AAAA ADDI.W #$AAAA,D0

00000416 Next Instruction

The ADDI.W instruction takes 4 words: $0679, $5555, $0000, $AAAA

The ADDI.L instruction takes 5 words: $06B9, $AAAA, $5555, $0000, $FFFE

The ADDI.W instruction takes 3 words: $0640, $AAAA

Notice that the first two instructions have an absolute long address (mode = 111, reg = 001) as the effective addressing mode. This means that the op-codes that use an immediate operand are actually longer then 1 word in length, they may be two or three words long.

How does this affect you? Simple, suppose that your algorithm works by placing the address of the op-code word in an address register and then decoding the op code word. Next, you pass the register as pointer to the effective address algorithm, but you leave the pointer where it is. The person doing the effective address reads the op-code word and masks out everything but the last 6 bits. Thus, they don't know that this is an immediate operand instruction.

Consider the instruction at address $0408. They decode the op-code and see that it is ADD.L. They pass the pointer (pointer = $0408) to the effective address person and the effective address person reads the op-code word, $06B9, and isolates the effective address field by **ANDing** $06B9 with $003F, resulting in $0039 ($39 = % 0 0 1 1 1 0 0 1).

Effective address person then thinks that the long word address following the op-code word is $AAAA5555 because effective address person doesn't know that it is an immediate instruction, and that $AAAA and $5555 are the immediate operands and not the destination address. Also, effective address person returns the pointer pointing to $0000 as the next instruction boundary ($040E), rather than $D179 ($0412).

What do you do about it? Well, one solution is to handle the immediate source operand as part of the op-code and pass to the effective address person the address of the operand following the op-code word and also, the effective address field in a register, because it gets lost otherwise.

This is a nice way to solve it because if the effective address is a data register, then the pointer does not get advance anymore and the effective address person returns the pointer as it was. This is shown in the following snippet.

```
00000412 0640AAAA                ADDI.W    #$AAAA,D0
```

The “op-code portion of this instruction is $0640, $AAAA. The pointer, when it’s passed to the effective address person, is pointing to $0416. Since the EA is a data register, there are no more words or memory needed for this instruction, so the pointer is returned unchanged.

V. Printing the instruction

Even the simplest instruction must always take at least 1 word of memory. Even a NOP (do nothing) take a total of 16-bits. Thus, we can get a lot of the display system working by starting the disassembler so that you can print the current address that you are pointing to in memory, the word “DATA” and the 4 hexadecimal digits that represent the word at that address. If you can do this without crashing, you’ve got a lot accomplished.

But the program is supposed to do much more than that. How do we print out anything to the screen? For that information I suggest you check on the TRAP #15 instruction in your text. We’ll also discuss it in class and your final homework will give you lots of practice. For now, let’s just suppose that you can print a line of text, located somewhere in memory, to the screen.

So, if you have a line buffer set up in memory, you can build the buffer with information as you get it, and then when you’re ready, you can print the buffer to the display. Here’s an example:

1- You know what address in memory you’re point to, so you can put that address into the start of the buffer.

2- You then need the character for a TAB or space, so you can put that in next.

3- If you don't know the op-code, you can add the ASCII characters for D, A, T, A and then add another space or TAB.

4- If you wrote DATA, then you must write out the ASCII code for the 4 hexadecimal digits representing the word of data at that address.

5- Then you can send out a newline character ($0A, $0D) and start again.

But suppose you can decode it. Then at step #3, above, you would write the op-code. For example, in the sample code we tried above, bits 6, 7 and 8 would have told us the op-code. If bits 6, 7, 8 equal 000, then you could immediately write CMP.B and a space.

Now the instruction CMP.B is written in assembly language as CMB.B <ea>,Dn. Thus, we have to figure out the effective address so we can display it, then we can add the comma, then we can decode the register from bits 9, 10, 11.

VI. Organizing the project

Before you write one line of code, you should meet to set-up your coding conventions. This is common practice in most industrial settings. In your case, its crucial because you will each be writing different pieces of the program and you need to be able to put it together and make it work. For example, here are some issues that you need to consider:

- How are parameters passed into subroutines?
- How are parameters returned from subroutines?
- How do you signal an error?
- How do you signal success?
- How do you document what each subroutine is supposed to do?
- What does an API for a subroutine look like?
- How do you control your source code? (Don't underestimate this one)
- What is your development schedule?
- How do you know if you are on track or in trouble?
- What are the milestones?
- How will you test the program?

- How will you do the write-up?

I suggest that one of the first things that you do on your project team is to have someone build the test program. This becomes your reference for the balance of the project. As your decoding algorithm improves, you will be able to decode more and more of the test program.

Another important task for your early meetings is to come up with a realistic project schedule that you all buy into. Believe me, it doesn't make any sense to develop a schedule that nobody is willing to commit to. I missed getting a raise because of that one. As you begin to get into the project, you should plan on adjusting the schedule to reflect reality. If things are falling behind, then have a fallback plan. Redo the schedule based on completing a subset of the assignment, but doing that subset well.

Again, a good plan might be to decode the NOP first. This is simple, there are no effective addresses to worry about. Success is measurable if you see your NOP in a sea of DATA statements.