



Project preface

PREFACE

- **Project teams:** This project is designed to be worked on by a team of programmers. **Up to 3 students** may organize themselves into a project team.

If all else fails, let me know and I will assign you to a team. Any team of two students has a good chance of me adding another student to your team, whether you like it or not. **Your team should be made by the fourth week, and submit team information [here](https://canvas.uw.edu/courses/1032102/assignments/3068279) (https://canvas.uw.edu/courses/1032102/assignments/3068279).**

It is not uncommon for project teams to basically fall apart and self-destruct. Here is a **hand-out** (<https://canvas.uw.edu/courses/1032102/files/34058433/download?wrap=1>)  (<https://canvas.uw.edu/courses/1032102/files/34058433/download?wrap=1>)  (<https://canvas.uw.edu/courses/1032102/files/34058433/download?wrap=1>) from Professor Walter Freytag (Business School) that I encourage you to read and discuss among yourselves at the start of your project. It won't guarantee a successful project, but reading it can help you set reasonable expectations for you as a team and as individuals within that team.

You will make a team at the 4th week in the quarter. After two weeks, you will have a chance to change your team.

- 1) You can voluntarily leave your team. In that case, you can have a new team.
- 2) A group can fire a team member. In that case, the fired person has to work alone, without creating a new team.

After two weeks, you cannot change your team, and you will be a team no matter what.

Warning: Project can be graded individually if necessary. If any member has not contributed to the programming, and the concern is expressed in the evaluation report, then the individual will get deducted points. Please note that, your final project is graded based on your final product. Then, from the individual report, the person who significantly harmed the team's work will get point reduction.

- **Progress reports:** Working for a project in a team is not easy. There are many issues, but most importantly, you should manage your time well. In this class, I, as an instructor, have to force you to manage your times with weekly progress reports. Each week after the team has been formed, you will be asked to submit two types of progress reports. One for individual report, and the other is for group report. If you do not submit on time, you will lose points. Regardless, please use them wisely so that they can help you, not distract you. For the progress report guide, please click [here](https://canvas.uw.edu/courses/1032102/pages/progress-report). (<https://canvas.uw.edu/courses/1032102/pages/progress-report>)

- **Scope of the project:** This project is a substantial effort, but it really isn't the "ordeal" that some students turn it into. For each of you, it is about the effort required to do 2-3 homework problem sets. Please approach this systematically, and do not wait too long to get started.
- **Cheating:** This project has been given before. You may have friends who have taken this class before you and may still have a copy of their project. Copying the project, or part of the project, from a former student is cheating and will not be tolerated. **Don't even think about it!** There is a special program developed that scans every past project into a database. Your project will be compared to this database of every former project. Believe me, the program works. I can even tell if you just copied a few lines of code from someone else, who you copied from and what you copied. You can't fool the program by changing the comments or the variable names. Also, I have the right to change the grade of the person you copied from. Don't do it. It isn't worth it. **Do not test on me!!**

There is one exception to the above rule. If you are repeating this class, you may make use of **the particular part of the project code that you previously developed**. You may not reuse the project code developed by your former teammates. You may only use code that you wrote. You may not share the entire project contents with your new project team. That would be considered to be cheating on their part and cheating on your part for providing it. Note that I will detect the fact that you have reused code with my cheating program, but I will see that it was your previous work when I review it. However, if you did not write any code with your previous project team, you are not entitled to use any of the code written by your former team mates. Remember, I have your personal statements on file and I can check to see what code you wrote the last time around.

In summary, if, for any reason, you are repeating the class, you may only re-use the code that you personally wrote.

- **Testing:** I will use the Easy68K simulator to load your program into memory at \$1000. I will then load my test program somewhere else in memory. My program will be comprised of a set of the instructions and data that are given in this specification. The test program will include other instructions that you are not required to decode, data fields, blank space, etc. All of your output will be sent to the display and also logged to a file. I then compare your output with my listfile.

When I grade your program I check it in two ways. First, I run the program and use it as any user would. I test the user interface and check it for robust recovery from input errors. Next, I ask it to disassemble my test program located somewhere in memory.

For this phase of the testing I log all the screen output to a file. You can do this as well, and you should.

At this point all I/O between the keyboard and the screen will also be sent to this file. You can then review your own file and see all of your I/O. It saves lots of time.

Progress report

Progress report guideline

You should have a regular meeting, at least once a week for this project. Since you will start the project right after you form the group, you should have at least 5 meetings in this quarter.

To have a successful team work, it is really important that you keep your progresses posted to your team members as well as to me.

Therefore, I ask you to write a progress report every week; so that I can see how you are doing. I will collect two types of progress reports each week. One is for a group report, and the other is for an individual report.

Your individual report focus on what YOU (individually) have done for this project. As a team, each might have different roles and you want to write in detail about your job.

Team report can collect all the individual report, or report the overall works as a team. I want to see how the individual works can be combined in harmony in this team report.

Please do not copy team report to your individual report. If you copied the team report to your individual report, you do not get any credits.

In the assignments, you will see “progress report” submission details, and should submit two types of progress reports each week. If you do not submit on time, you will lose points for each report.

For the format, please find the following guidelines.

=====

Heading:

- Date: Date the memo is sent
- To: Usually, name and position of the reader but, in this course, write, for example, “CSS 422, Winter 2016”
- From: Name and position of the writer
- Subject: A clear phrase that focuses the reader's attention on the subject of the memo

Work Completed:

Explain what work has been done during the reporting period. You can arrange chronologically, or divide it into each task you have done. But be consistent. For each week, you have to submit a required task report.

Problems:

Explain any problems you have encountered. This problem usually should be reported to your team member if it is an individual report.

Work Scheduled:

Enumerate all the works scheduled. It is recommended to write in the order of priority. This part can be repeated if you did not finish it.

Self Evaluation:

Evaluate your progress so far.

Supports:

You have to provide any supporting documents/codes to prove your work. For the team report, each support will be specified. For the individual report, you should provide your codes that have been written in that week.

Confidential evaluation

Confidential evaluation report

You may work in teams of up to three people. Each of the team members **usually shares the same grade, but there can be exceptions.**

In addition to the required deliverable that are discussed, below, each member of the project team **must individually submit a confidential statement** briefly describing:

1. What you did on the project
2. What the other team members did on the project
3. An estimate of the percentage of the project that you contributed
4. An estimate of the percentage of the project that each of your team mates completed

Please note that you are not simply answering the above questions. You should write a report that describes the above questions.

If any individual student fails to submit this statement, they will receive a 5 point deduction of their grade.

Your individual grade on the project may be different from the project grade as a whole. If other team members report that someone is not doing their fair share, then I will use that information, along with your statements to adjust the grades of the individual team members. **Thus, if any member or members of a project group feels that the other team member(s) are not doing, or did not do, their fair share of the work the student should contact the instructor as soon as possible.**

Specification

Specification:

1) Write an inverse assembler (disassembler) that will convert a memory image of instructions and data back to 68000 assembly language and output the disassembled code to the display. You will not be required to disassemble all of the instructions and addressing modes. The list of instructions and addressing modes is given at [Required Opcodes \(https://canvas.uw.edu/courses/1032102/pages/required-opcodes\)](https://canvas.uw.edu/courses/1032102/pages/required-opcodes) (<https://canvas.uw.edu/courses/1032102/pages/required-opcodes-w14>) page. Note that I'm not going to fill the memory with garbage!

2) If you want to see how a disassembler works, just take one of your homework problems and load it in memory at an address after your program. Then open a memory window and see the code in memory. You can also view it as disassembled code in the simulator.

3) DO NOT USE THE TRAP 60 FACILITY OF THE SIMULATOR. You must completely develop your own disassembler algorithm. If I suspect that you used TRAP 60 I will use a search tool that I designed to scan your source code for the TRAP 60 calls. **You may only use the I/O Trap functions up through 14.** Trap task 15 and higher may not be used. Note that you can call TRAP #15, for I/O. But you cannot use trap task #15. That is, you **cannot call, MOVE.B #15, D0, TRAP #15.**

4) Your program should be written from the start in 68000 assembly language. Do not write it in C or C++ and then cross-compile it to 68000 code. It is really easy to tell when you've written it in C and it probably won't save you very much time. When you are working at the bit level C is just structured assembly language (or so they say).

5) Your program should be ORG'ed at \$1000.

6) At startup, the program should display whatever welcome messages you want to display and then prompt the user for the **starting location and the ending location** (in hexadecimal format) of the code to be disassembled. The program should scan the memory region and output the memory addresses of the instructions and the assembly language instructions contained in that region to the display. You should be able to

actually disassemble your own program to the display! By the way, in the past, some student have actually embedded some test code in their programs. **I strongly urge you not to do it!**

7) The display should show one screen of data at a time, hitting the **ENTER** key should display the next screen of information.

8) The program should be able to realize when it has an illegal instruction (i.e, data), and be able to deal with it until it can find instructions again to decode. Instructions that cannot be decoded, either because they do not disassemble as op codes or because you aren't able to decode them should be displayed as:

```
1000 DATA $WXYZ
```

where \$WXYZ is the hexadecimal number that couldn't be decoded. **Your program should not crash because it can't decode an instruction.** Remember, it is perfectly legal to have data and instructions interspersed, so it is very possible that you will hit data, and not an instruction.

9) Address displacements or offsets should be properly displayed as positive or negative numbers, but a better grade will be achieved if you actually calculate the address of the branch and display that value. For example:

```
1000 BRA -7 *Branch back 7 bytes
```

is acceptable, but:

```
1000 BRA 993 * Branch to address 993
```

is better.

10) You should do a line by line disassembly, displaying the following columns:

a- Memory location b- Op-code c- Operand

11) When it completes the disassembly, the program should prompt the user to disassemble another memory image, or prompt the user to quit.

Deliverables

Deliverable: You must submit item 1 and item 2 in [Final Project](https://canvas.uw.edu/courses/1032102/assignments/3068262), (<https://canvas.uw.edu/courses/1032102/assignments/3068262>) and item 3 to [Confidential Individual Evaluation](https://canvas.uw.edu/courses/1032102/assignments/3068259) (<https://canvas.uw.edu/courses/1032102/assignments/3068259>) (Do not zip! please)

Note: Your deliverable will be up to 90%. 5 % are for the progressive reports for the last 5 weeks (team and individual). The rest 5% are reserved for your individual evaluation documentation.

1. Source code: Please submit your source file (.X68) AND listing file (.L68). Your source file name should be, <Team name>_W16.X68
2. Documentation. Please include all the following sections and make it as one documentation.

1) **Program description:** This is a write-up of 1 to 2 pages about your program. Describe your design philosophy, flow chart, any algorithms that you were especially proud of, any algorithms that you got from other sources (copied, downloaded, etc.). Using canned routines is OK but you MUST cite your use of them. Remember, canned library routines are very easy to spot. Also, the more routines that you appropriate from other sources, the more expectations I'll have for the coding of your part of the program. The program description should also cover any limitations that you are aware of.

2) **Specification:** 1~2 pages. This is a simple list of what your program does.

3) **Test Plan:** 1~2 pages. This is a description of how you tested the program. It should also contain a description of your team's coding standards. If you have, please include your testing files as well.

4) **Exception report:** This is your opportunity to describe problems that you've encountered but couldn't fix, or chose not to fix. Anything that you feel deviates from your intended program. Also, this is where you can describe what you were able to complete in the time allotted versus what the assignment asked for. This should definitely include the results of your testing if you found defects but didn't fix them. In an ideal situation, I should be able to just read your documentation and your source listing and give you a grade, without needing to run the program. Of course, I will run the program, but I hope that you get the idea.

5) **Team assignments and report:** A description of how you organized your team's tasks. That is, "Who did what and how". You should specify the amount of the coding, as a percentage, the member did in the project. This information is VERY important as it will be a source for the separate grading.

3. **Confidential evaluation report:**

<https://canvas.uw.edu/courses/1032102/pages/confidential-evaluation>) The separate individual evaluation is described in the **Confidential Evaluation page** <https://canvas.uw.edu/courses/1032102/pages/confidential-evaluation>). Each member of the team submits their own evaluation. If you do not submit your own evaluation, you will receive an automatic 5 point deduction from your grade on the project. No exceptions!

Simulator issues

Simulator Issues

There may be some issues with using the Easy68K as the project's only simulator package that I'm not yet aware of. In general, the simulator is very forgiving when it comes to displaying the disassembled lines of code on its terminal screen. However, what might look OK to you on the screen, may actually appear as one long line of text in the log file because you neglected to add carriage return <CR> and line feed <LF> characters to the end of your output lines. Every line that you send to the screen should be terminated with the ASCII codes for CR and LF.

It is to your advantage to print out the output file. Easy68K has a logging function that will capture everything that was output to the display. If your print-out is messed up, mine will also be messed up.

There should only be the printable ASCII character set, CR and LF in your I/O to the screen.

There will be an automatic deduction of 10 points from your grade if I have to go into the log file and create a printable output file because you forgot to add the proper formatting control characters or you added non-printable characters, such as a beep, to your output.

Easy68K bugs

It seems like Easy68k simulator program contains several bugs. Here are the list of Easy68k bugs which are captured by students who have taken this class previously. I strongly urge you to read this first so that you do not have to struggle with this bug when it comes to debugging your own program.

1. A MOVE.L operation with direct data going into a data register with the direct data being less than 5 bits causes the EASY68k to perform a MOVEQ instead of a MOVE.L.

2. Similar bugs can be found with most commands which have a "Q" variant. The 68K assembler seems to automatically use the less memory and processor intensive version of the command when possible. I assume this is to maximize the system's efficiency and speed.

3. The Easy68k Manual shows that a MOVE operation with the destination being an address register is an invalid operation, and should be a MOVEA operation.

However, the Easy68k compiler gives no warning or errors when trying to perform a MOVE with the destination being an address register. Because the Easy68k shows that this kind of operation is invalid, the program will display a MOVEA in place of the MOVE when displaying the data.

4. Our inverse assembler will start to act irregularly and will start to display error messages for valid op-code data if a large amount of data/operations are sent through the inverse assembler. We encountered this problem when trying our test routine that tested all possible combinations of valid data for op-codes. We first thought it was our inverse assembler that was causing the irregular performance, but if you repeatedly perform the same action/operation, such as NOP, through the Easy68k assembler for countless times the Easy68k starts to act irregular, which results in our inverse assembler acting irregular as well. The test routine that tested all possible combinations of valid data was around 5,500 lines of code.

Grading Standards for Team Projects

items		Points
Program	Opcode+EA	50
	I/O	20
Documentation		20
Confidential Individual Evaluation		5
Progress reports		5
Total		100

Most cases, a team will share the same grade, but there will be exceptions if necessary.

Each team member's project point is first given based on final product. Then, based on the confidential evaluation report, members whose contribution are too low, will have additional point reduction.

The program will be carefully tested for all required op-code and EAs. Here are some examples where points are deducted.

- 1) Program cannot assemble into listing file: program point is 0 out of 70
- 2) Program crashes on non-required opcode or EA: -5
- 3) Program crashes on each required opcode or EA: -3 each
- 4) Each required op-code or EA which is not properly disassembled: -2 each
- 5) Program does not print the address of each instruction: -5
- 6) Program does not print results as one screen at a time: -5
- 7) Program does not have options to restart or finish: -5

Required opcodes

Required op-codes and addressing modes

Below are the list of instructions and addressing modes assigned for this project.

Effective Addressing Modes:

1. Data Register Direct
2. Address Register Direct
3. Address Register Indirect
4. Immediate Data
5. Address Register Indirect with Post incrementing
6. Address Register Indirect with Pre decrementing
7. Absolute Long Address
8. Absolute Word Address

Instructions:

1. MOVE
2. MOVEQ
3. MOVEM
4. ADD
5. ADDA
6. ADDQ
7. SUB
8. SUBA
9. SUBI
10. MULS
11. DIVS
12. LEA
13. CLR
14. AND
15. ANDI
16. EOR
17. EORI
18. LSR
19. LSL
20. ASR
21. ASL
22. ROL
23. ROR
24. BCHG
25. CMP

- 26. CMPA
- 27. CMPI
- 28. Bcc (BCC, BGT, BLE, BVS)
- 29. JSR
- 30. RTS

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. My test program is usually around 300 bytes in length, yet very few disassemblers can get through it without stumbling.

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.



According to Dr. Berger's note, "past experience has shown that a good organization for a three-member team might be:"

- One person handling instruction decode
- One person handling effective addresses
- One person all I/O to the screen and user interface issues

You don't have to do it this way. It has just worked out well in the past.

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.

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/1032102/files/34058433/download?wrap=1) (<https://canvas.uw.edu/courses/1032102/files/34058433/download?wrap=1>)  (<https://canvas.uw.edu/courses/1032102/files/34058433/download?wrap=1>)  (<https://canvas.uw.edu/courses/1032102/files/34058433/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 you're 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 table. 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 not necessary 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, 1011XXXXXXXXXXXX, 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 than 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.