UNIVERSITY of WASHINGTON

CSS 422 Hardware and Computer Organization

Project Overview

Professor: Yang Peng

The slides are re-produced by the courtesy of

Dr. Arnie Berger and Dr. Wooyoung Kim

Project Description

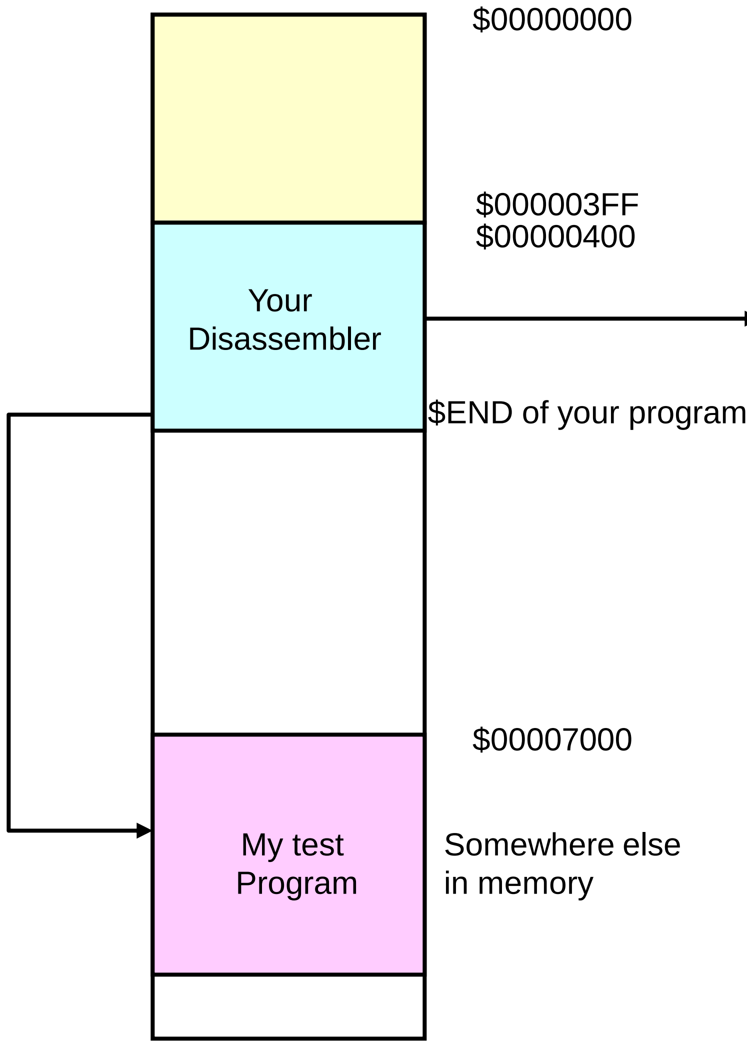
- See the canvas for the project description
  - https://canvas.uw.edu/courses/1408528/pages/project-description

- Progress reports
- Confidential evaluation
- Specification (how to program, etc.)
- Deliverable (what to submit, when, how)
- Simulator issues and Easy68k bug report (reported by students from previous class)
- Grading standards
- Required op-code and EA
- Addendum (additional information, will be continuously updated)

What is a disassembler?

- Disassembler (also called an ***Inverse Assembler***):
  - **Scans** a section of memory, and
  - **Attempts to convert** the memory's contents to a listing of valid assembly language instructions, because of "1:1 mapping"
- Most disassemblers cannot recreate symbolic, or label information
- Disassemblers **can be easily fooled** by not starting on an instruction boundary
- How it works:
  - The disassembler program **parses the op-code word** of the instruction and then **decides how many additional words** of memory need to be read in order to complete the instruction
  - If necessary, reads additional instruction words
  - The disassembler program **prints out the complete instruction** in **ASCII-readable format**
    - Converts binary information to readable Hex

What is a disassembler?



OUTPUT to the Display

00007400	LEA	$00007422,A1
00007406	MOVE.W	$00007444,D1
0000740C	MOVE.B	#01,D0
00007410	TRAP	#15
00007412	LEA	$00007446,A1
00007418	MOVE.B	#02,D0
0000741C	TRAP	#15

What is a disassembler?

- Source file contains symbolic names for numerical values, comments, symbol names for memory locations (variables)

- Source file **does not contain detailed memory location information**

```
NUM1    EQU       $AA            *First number
NUM2    EQU       $55            *Second Number
stack   EQU       $7000          *Stack pointer
temp    EQU       $1000          *Memory variable

        ORG       $400           *Starting address
start   NOP
        MOE.W     #STACK,SP      *Initialize the stack pointer
        MOVE.B    #$D7,D0        *Load D0 with D7
        MOVE.B    #NUM1,D1       *Load first number
        MOVE.B    #NUM2,D2       *Load the second number
        MOVEA.W   #temp,A0       *Load temp address
        MOVE.B    D1,(A0)+       *Save it
        MOVE.B    D0,(A0)        *Save next
        SUBA.W    #$0001,A0      *Store address
        ASR.W     (A0)           *Shift it
        MOVE.W    (A0),D7        *Get it back
        BRA       start          *Go back and do it again
        END       $400           *End of code
```

What is a disassembler?

- List file contains symbolic names for numerical values, comments, symbol names for memory locations (variables)
- **List file contains detailed memory location information** not found in source file, including line numbers, other cross-reference information, and object code

```
1   000000AA      NUM1:             EQU       $AA          ;*First number
2   00000055      NUM2:             EQU       $55          ;*Second Number
3   00007000      STACK:            EQU       $7000        ;*Stack pointer
4   00001000      TEMP:             EQU       $1000        ;*Memory variable
5
6   00000400                        ORG       $400         ;*Starting address
7   00000400 4E71         START:    NOP
8   00000402 3E7C7000               MOVE.W    #STACK,SP    ;*Initialize the stack pointer
9   00000406 103C00D7               MOVE.B    #$D7,D0      ;*Load D0 with D7
10  0000040A 123C00AA               MOVE.B    #NUM1,D1     ;*Load first number
11  0000040E 143C0055               MOVE.B    #NUM2,D2     ;*Load the second number
12  00000412 307C1000               MOVEA.W   #TEMP,A0     ;*Load temp address
13  00000416 10C1                   MOVE.B    D1,(A0)+     ;*Save it
14  00000418 1080                   MOVE.B    D0,(A0)      ;*Save next
15  0000041A 90FC0001               SUBA.W    #$0001,A0    ;*Store address
16  0000041E E0D0                   ASR.W     (A0)         ;*Shift it
17  00000420 3E10                   MOVE.W    (A0),D7      ;*Get it back
18  00000422 60DC                   BRA       START        ;*go back and do it again
19  00000400                        END       $400         ;*end of code
```

What is a disassembler?

- What the same memory region would look like if displayed by an inverse assembly program
- Displays memory addresses and instructions at that address
- **All symbolic information and comments are lost!**

```
00000400    NOP
00000402    MOVE.W     $7000,SP
00000406    MOVE.B     #$D7,D0
0000040A    MOVE.B     #$AA,D1
0000040E    MOVE.B     #$55,D2
00000412    MOVEA.W    $1000,A0
00000416    MOVE.B     D1,(A0)+
00000418    MOVE.B     D0,(A0)
0000041A    SUBA.W     #$0001,A0
0000041E    ASR.W      (A0)
00000420    MOVE.W     (A0),D7
00000422    BRA        $00000400
```

Important! Testing Your Code!

Remember this page! You will check this page many times!

Assume that you have your disassembler program ready.

1. Write a testing source code (testing.X68→ testing.S68)
   - List all the required opcode and EA
   - Any non-required opcodes to see if your program can catch it as invalid data
2. Run your disassembler program from the source file
3. Your program will open in the simulator program
4. In the simulator, go to **File→Open Data**
5. Choose the “testing.S68” file as a testing file
6. Then, the assembled testing file will be loaded into your memory
7. See where the “data” is loaded
8. Go to **Run→Log Start** to have a log file
9. Run your program, and give the starting and ending address when prompt ($7FC0 and $814F, for example)
10. Should show one screen of data at a time, hitting the ENTER key should display the next screen

Group Dynamics and Logistics

- 2 or 3 students per team (up to 4 in Autumn 2020)
- Get an early jump on this project. Don’t wait! You still have two exams, five assignments, and 14 exercises to prepare for!
- Plan, plan, plan! Do not write code until you know what you are doing!
- Back-ups and version control! You are required to use Github or Bitbucket!
- Integration!
- Develop a test program early!
- Test thoroughly, do incremental development!
- Develop a schedule and follow it!
- Meet regularly to sync-up your code and do a status check face-to-face. Don’t depend exclusively on emails!

Why Projects Fail

- **Insufficient testing**
  - Fail to find subtle bugs
  - Side effects due to word addressing
  - Incomplete test program
- Having to write too much code due to poor up-front planning
- **Team becomes dysfunctional**
  - Must be self-directed, no manager to beat you into submission
  - Poor division of responsibilities among team members
- **Underestimating** the needed effort and time
  - Waiting too long to start
- **Poor project management!!!**
  - No back-up or version control
  - Late code integration
- **Time management!!!**
- Caught cheating

Some Representative Milestones

1. Team is organized
2. Team meets to discuss and set expectations and team values
3. Team decides who does what
4. Development schedule is created
5. Test program is built
6. Team meets and decides on API's
7. I/O skeleton is complete, will display all memory as data
8. NOP is decoded
9. Other op-codes and effective address modes are added
10. Team meets regularly to check status, integrate SW
11. Begin abuse testing, start write-up
12. Complete personal statements
13. Complete all deliverables, pack everything up, cross your fingers and study for the final!

Team Organization 1

- Disclaimer: This is only ***one way out of many possible ways*** to organize your teams
- Team Roles based on **Functions/Libraries/Subroutines**
  - I/O Person: Handles all inputs from the user and displays to the screen – I/O Library
  - Op-Code Person: Handles decoding the OP-Codes and passing EA information to EA person – OPCode Library
  - EA Person: Decodes Effective Addresses – EA Library

Team Organization 2

- Disclaimer: This is only ***one way out of many possible ways*** to organize your teams

- Team Roles based on **workload**
  - Each member takes care of a certain amount of OpCode work
  - Each member takes care of a certain amount of EA work
  - Work together on I/O work – critical for testing and debugging

Team Organization 3

- Disclaimer: This is only ***one way out of many possible ways*** to organize your teams

- All team members work together on basic, common routines
- Team members divide the rest of work to balance workload

Team Organization 4

- Disclaimer: This is only ***one way out of many possible ways*** to organize your teams

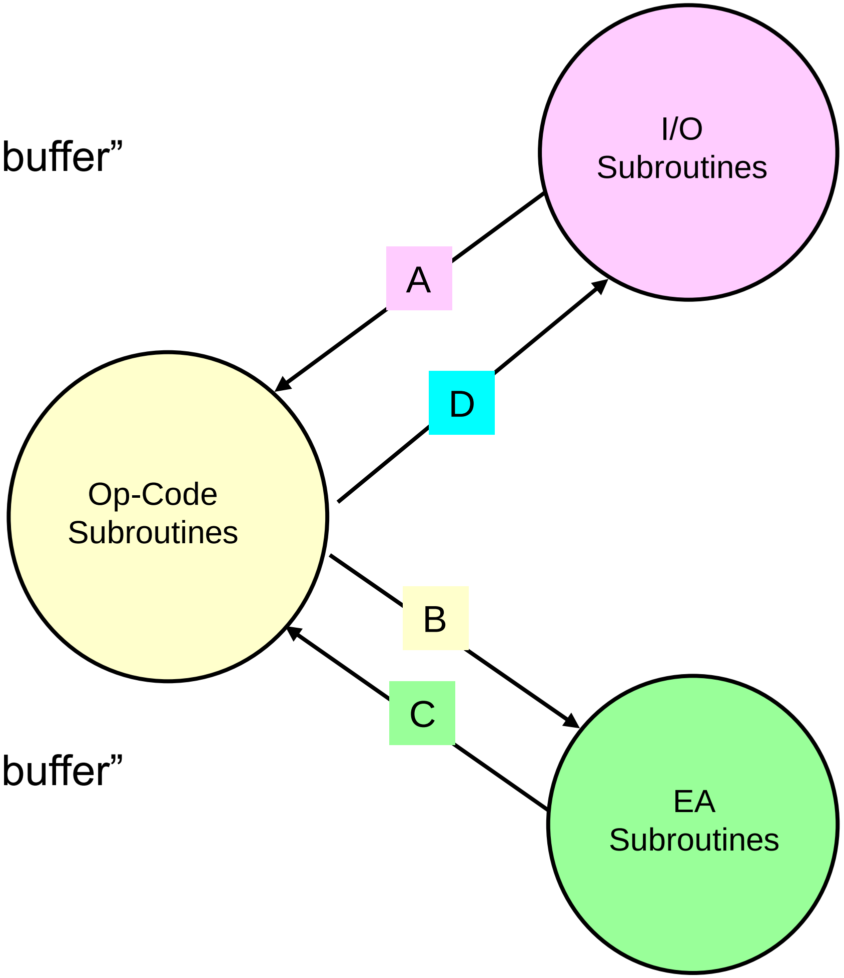
- Team Roles based on **subroutines/functions**
  - All team members sit together to define the needed functions and the APIs
  - All team members sit together to define how to parse in and out data between functions
  - Allocate different functions to different members to balance the workload

General Program Flow

1. I/O subroutines prompt user (me) for a starting and ending address in memory
2. User enters starting and ending addresses for region of memory to be disassembled
3. I/O subroutines check for errors and if address are correct, prepare the display buffer and send address in memory to Op-Code routines
4. Op-Code subroutines can either decode word to legitimate instruction or cannot.
   1. If word in memory cannot be decoded to legitimate instruction, I/O routines writes to screen: XXXXXXXX DATA YYYY, where XXXXXXXX is the memory address of the word and YYYY is the hex value of the word
   2. If it can be decoded, then it is prepared for display and the EA information is passed to the EA routines
5. EA subroutines decode EA field(s) and
   1. If EA cannot be decoded, signals this back, or
   2. Prepares operands for display
6. Once the instruction is displayed, process repeats itself

Parameter Passing

- A Parameters
  - Pointer to memory to decode
  - Pointer to next available space in “Good buffer”
  - Good/bad flag
- B Parameters
  - Memory pointer to next word after the op-code word
  - 6 bits from EA field of op-code word
  - Pointer to next available space in “Good buffer”
  - Good/bad flag
- C Parameters
  - Memory pointer to next word after the EA word
  - Pointer to next available space in “Good buffer”
  - Good/bad flag
- D Parameters
  - Memory pointer to next op-code word
  - Good/bad flag



Required Op-code and EA

- Not all op-codes/EA are required to disassemble
- See the list: https://canvas.uw.edu/courses/1408528/pages/required-opcodes

Autumn 2020 requirement

Instructions:
NOP
MOVE, MOVEM
ADD
SUB
MULS, DIVU
LEA
AND, NOT
LSL, LSR, ASL, ASR
Bcc (BLT, BGE, BEQ)
JSR, RTS
BRA

Effective Addressing Modes:
Data Register Direct
Address Register Direct
Address Register Indirect
Immediate Addressing
Address Register Indirect with Post-incrementing
Address Register Indirect with Pre-decrementing
Absolute Long Address
Absolute Word Address