

Project overview

Professor: Wooyoung Kim

The slides are re-produced by the courtesy of Dr. Arnie
Berger

Project description

- See the canvas for the project description:

<https://canvas.uw.edu/courses/925117/pages/project-description>

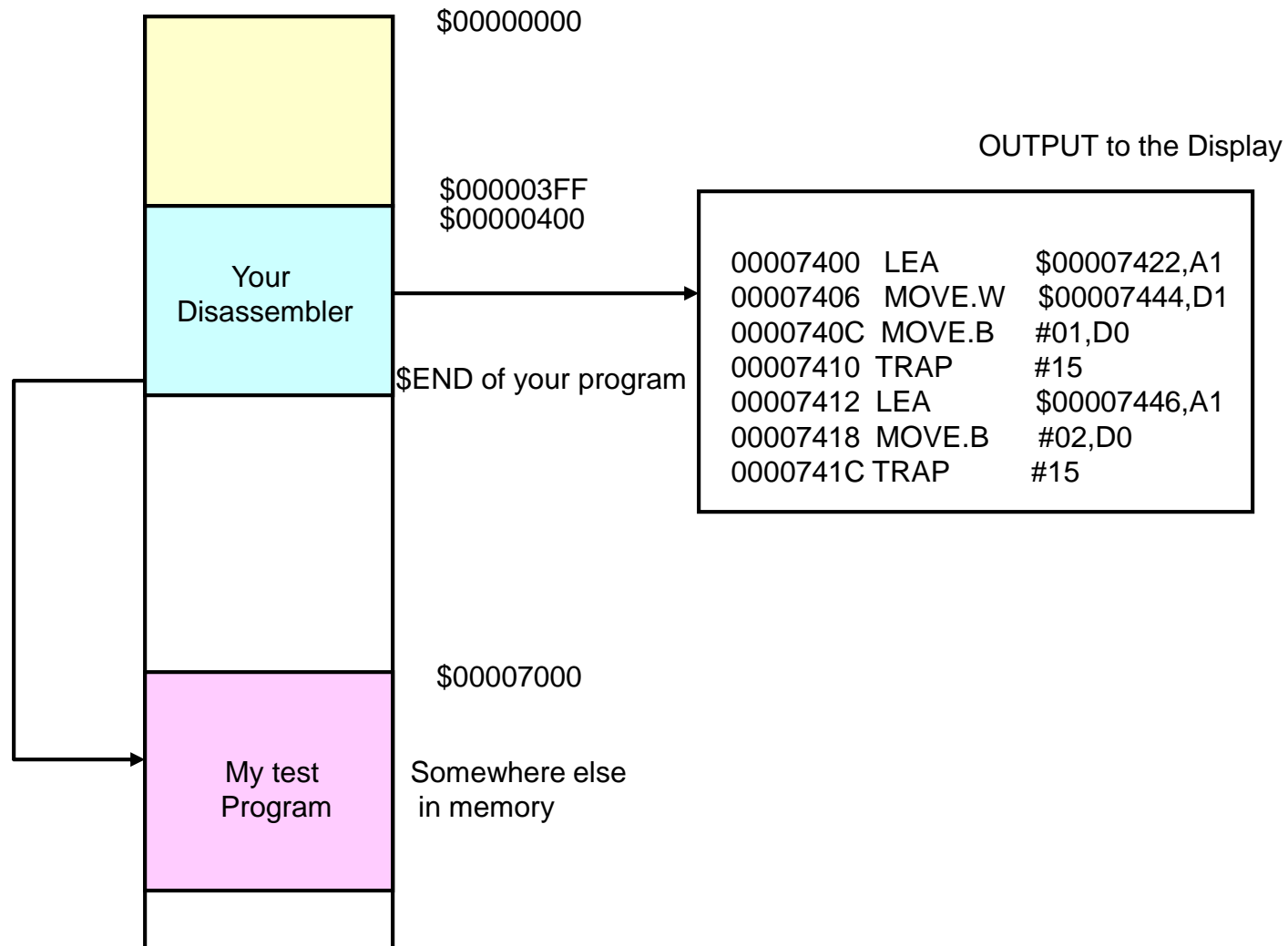
This slide is available online

- Progress reports (format, etc.)
- Confidential Evaluation Reports (description of the report, format)
- 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
- Most disassemblers cannot recreate symbolic, or label information
- Disassemblers can be easily fooled by not starting on an instruction boundary
- How it works:
 - 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
 - Prints out the complete instruction in ASCII-readable format
 - Converts binary information to readable Hex

What is a disassembler?



What is a disassembler?

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

start     ORG      $400      *Starting address
          NOP
          MOVE.W   #STACK,SP *Initialize the stack pointer
          MOVE.B   #$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)
- Also contains detailed memory location information not found in source file, 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

Testing your code

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

- Teams of 2 or 3, no larger
- Only two students have done it alone
- Two groups out of 19 has failed in one of the previous classes
- Get an early jump on this project. Don't wait! You still have a final exam to prepare for
- Plan, plan, plan: Do not write code until you know what you are doing
- Develop your API's before you write code
- Think about back-ups and version control
- Develop a test program early!
- Test thoroughly, do incremental development
- Develop a schedule in MS Project or Excel: Use it!
- Don't neglect your write-up
- Meet regularly to synch-up your code and do a status check face-to-face. Don't depend exclusively on e-mail.

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
- Underestimating effort required
 - Waiting too long to start
- Poor division of responsibilities among team members
- Lost project
 - No back-up or version control
- 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!

How to organize

- Disclaimer: This is one way of several possible ways to organize your teams
- Team Roles
 - I/O Person: Handles all inputs from the user and displays to the screen
 - Op Code Person: Handles decoding the OP-Codes and passing EA information to EA person
 - EA Person: Decodes Effective Addresses

General program flow

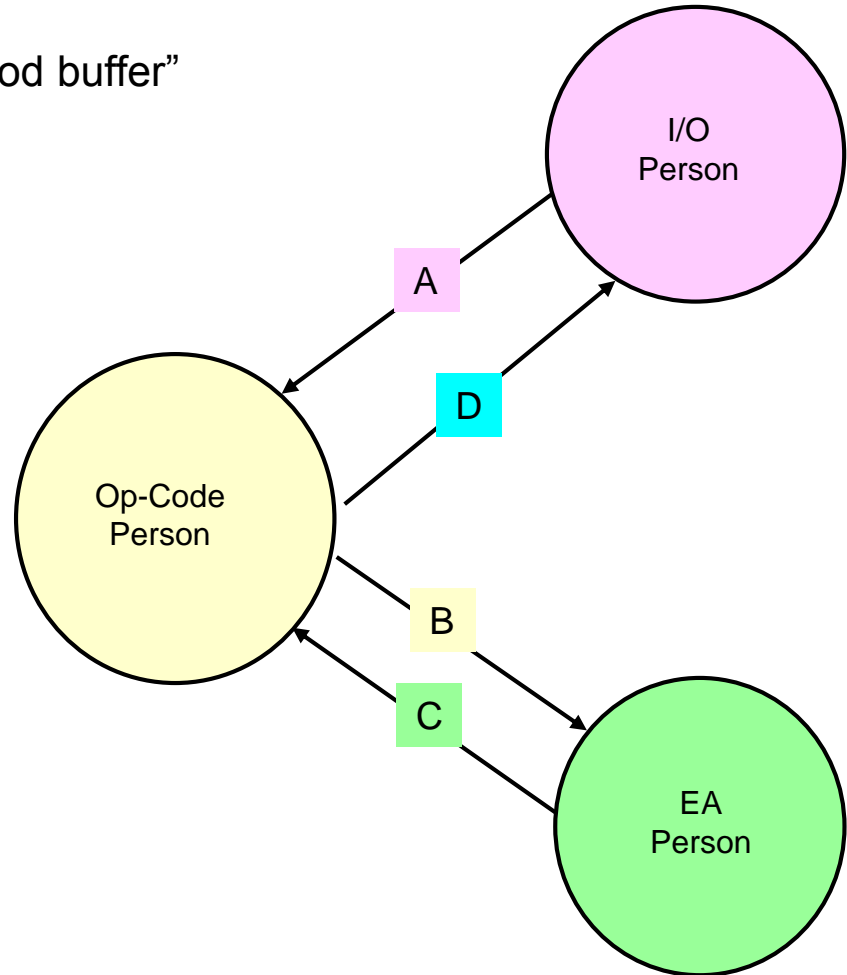
1. I/O person prompts user (me) for a starting and ending address in memory
2. User enters starting and ending addresses for region of memory to be disassembled.
3. I/O person checks for errors and if address are correct, prepares display buffer and sends address in memory to OP-Code person.
4. Op-code person can either decode word to legitimate instruction or cannot.
 1. If word in memory cannot be decoded to legitimate instruction, I/O person 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 person
5. EA person decodes 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

General responsibilities

- Individual responsibilities
 - Op-code person: Decodes op-code
 - Generally the strongest coder on the team
 - EA Person: Decodes effective addresses
 - Uses EA field information passed on by Op-code person
 - I/O Person: Interfaces to user
 - Decodes inputs from user
 - Formats and displays disassembled code
- Group responsibilities
 - Decide on roles
 - Design algorithm, coding conventions and parameter passing rules
 - Design test program
 - Meet to integrate and test
 - Test, test, test!
 - Do write-up

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
- 35 op-codes and 8 EA are required
- See the list on canvas,
<https://canvas.uw.edu/courses/925117/pages/required-opcodes>

Instructions:

MOVE, MOVEA, MOVEM
ADD, ADDA, ADDQ
SUB, SUBA, SUBI
MULS, DIVS
LEA
AND, ORI
NEG
EOR, EORI
NOT
LSR, LSL
ASR, ASL
ROL, ROR
BTST
CMP, CMPI, CMPA
Bcc (BEQ, BNE, BLT, BHI)
BRA, JSR, RTS

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