CSS-422 : Hardware and Computer Organization

Jim Hogg

Agenda

- Project Description
- Organization : Teams
- Deliverables
- Advice : How to Succeed
- Technical Direction
- Grading

Project Description

Description

Design, Build and Test a **Disassembler** for 68000 Code in 68000 Assembler Code

68k Assembler

Assembly Code

```
        ORG     $100
a:      DC.L    3        ; first argument
b:      DC.L    4        ; second argument

        ORG     $200
main:   BSR     aplus3b ; call subroutine
        MOVE.L  D0, D7  ; show result
        SIMHALT

aplus3b:                ; result = a + 3*b
        MOVE.L  a, D0   ; a
        MOVE.L  b, D1   ; b
        MULS    #3, D1  ; 3b
        ADD.L   D1, D0  ; a + 3b
        RTS             ; done!

        END main
```

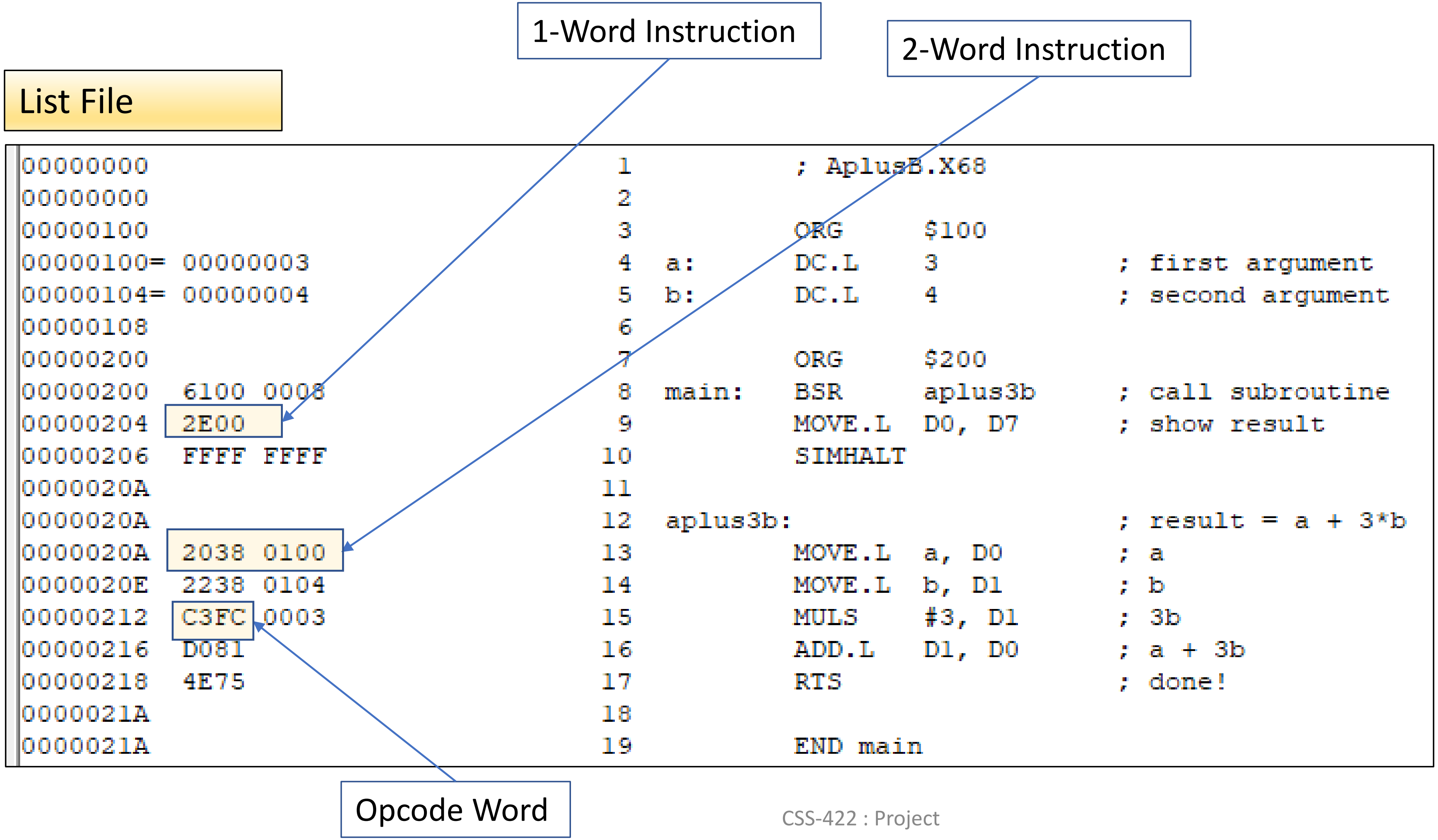
Assembler converts Assembly Code into Binary Code

Corresponding Binary Code, starting at address $200

```
0000 0003 0000 0004 6100 0008
2E00 FFFF FFFF 2038 0100 2238
0104 C3FC 0003 D081 4E75
```

We often say Machine Code instead of Binary Code

68k Assembler



List File

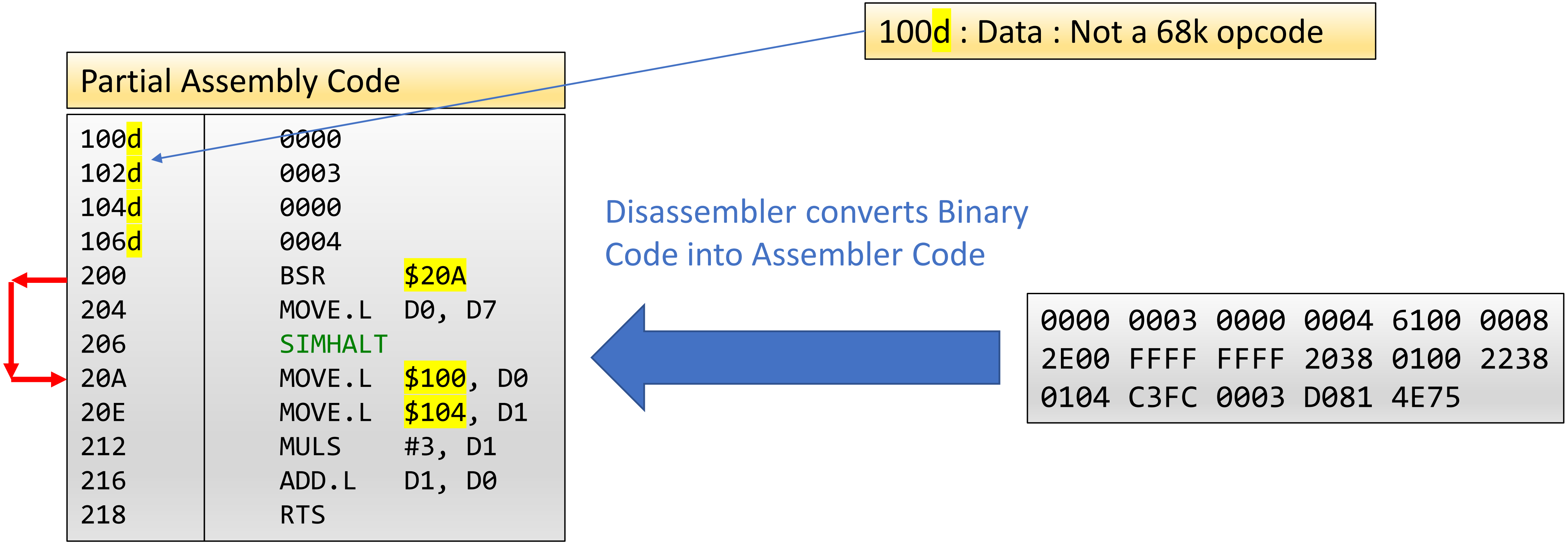
1-Word Instruction

2-Word Instruction

Opcode Word

```
00000000                         1          ; AplusB.X68
00000000                         2
00000100                         3          ORG     $100
00000100= 00000003               4  a:      DC.L    3              ; first argument
00000104= 00000004               5  b:      DC.L    4              ; second argument
00000108                         6
00000200                         7          ORG     $200
00000200  6100 0008              8  main:   BSR     aplus3b        ; call subroutine
00000204  2E00                   9          MOVE.L  D0, D7         ; show result
00000206  FFFF FFFF             10          SIMHALT
0000020A                        11
0000020A                        12  aplus3b:                       ; result = a + 3*b
0000020A  2038 0100             13          MOVE.L  a, D0          ; a
0000020E  2238 0104             14          MOVE.L  b, D1          ; b
00000212  C3FC 0003             15          MULS    #3, D1         ; 3b
00000216  D081                  16          ADD.L   D1, D0         ; a + 3b
00000218  4E75                  17          RTS                    ; done!
0000021A                        18
0000021A                        19          END main
```

68k **Dis**Assembler



100d : Data : Not a 68k opcode

Partial Assembly Code

```
100d      0000
102d      0003
104d      0000
106d      0004
200       BSR       $20A
204       MOVE.L    D0, D7
206       SIMHALT
20A       MOVE.L    $100, D0
20E       MOVE.L    $104, D1
212       MULS      #3, D1
216       ADD.L     D1, D0
218       RTS
```

Disassembler converts Binary Code into Assembler Code

```
0000 0003 0000 0004 6100 0008
2E00 FFFF FFFF 2038 0100 2238
0104 C3FC 0003 D081 4E75
```

Our DisAssembler won’t be able to recover variable names, function names, or comments (Real disAssemblers, or debuggers, *can* do this, using an auxiliary *Symbols* or *Debug* file)

Project : Details

Given a block of binary data, in memory:

1. Convert opcode word to mnemonic
2. Work out how many additional words of memory, if any, are needed for the operands
3. Work out addressing modes
4. Extract operands
5. Complete the assembly instruction
6. goto step 1

Notes:

- You must write the disassembler in 68000 Assembly Code
- You must write all the code yourselves – don't use code found elsewhere!
- You won't be able to recreate variable names or subroutine names
- Be robust against disassembling *data*

Organization

Project Teams

- Group Project – 2-4 students per group. This mimics real life in a software company
- Choose your group. Be careful! All members of the group will receive the same grade
- If you don’t self-select by Thursday this week, I will form groups from unassigned students (randomly, in Canvas)

I have set up 15 Groups in Canvas, and given them names of 68k instructions:

- ADD, AND, BEQ, BNE, BSR, CMP, EOR, FABS, JMP, MOVE, MULS, NEG, NOP, RTS, TRAP

So figure out whom you want to work with, choose one of the above groups, and add yourselves to it.
Groups are limited to 4 students. (I can set up more if 15 is not enough)

Typical Project Milestones

1. Team meets to discuss and set expectations and team values
2. Team decides who does what
3. Create schedule: wild guess! Include “contingency”
4. Build test program or programs - incrementally
5. Decide APIs
6. IO : start by just displaying machine code as raw hex
7. Disassemble NOP, RTS
8. Disassemble more opcodes and address modes
9. Test against TestDasm.X68
10. Gather all deliverables

How to Break up the Development Work

You can divide the work among the team however you choose. Here are 3 alternatives to consider.

IO Dev: handle input from user and display to screen

Opcode Dev: handle each opcode

Address Dev: handle all addressing modes

Each Dev does some of the opcodes

Each Dev does some of the addressing modes

Work together on IO

Partition the API functions among team members

Milestones

You should plan several “milestones” for the project – where you integrate the group’s code and test it (and save it somewhere safe!)

Probably weekly, but decide what works best

For example:

- IO routines written; other APIs documented in code, but empty; preliminary tests written
- Scan the test file (Test.X68) and write out the raw hex
- Disassemble simple instructions: NOP, JSR, etc
- Disassemble more instructions (agree list in advance)
- Error handling
- Remaining instructions (eg: MOVEM)
- Check against Test
- Gather final report

The class project aims to be a foretaste of your future life in professional Software Development:

- Working in small teams is common, but not easy. It introduces you to: communicating within the team and with your *manager*; ensuring everyone is working equally hard; coping with inter-personal disputes; making the most of each other's skills
- It may not be obvious what is actually required of the project (although the spec for this one is fairly well-defined)
- Time is unyielding. You will fall behind schedule. Plan how to catch up; or to cut some functionality (eg: "Our project doesn't support MULS or JSR")

Depending on how teams progress, I might provide hints during the project, so all teams cross the finishing line

Deliverables

Deliverables

Source Code

Files for the 68000 Assembler code for your Disassembler, called: <TeamName>.X68 or <TeamName>.zip if multiple files

Test Results

Text file showing the results of running your Disassembler over the file TestDasm.X68, into a file TestDisAsm.txt
(I will issue TestDasm.X68 later. Meantime, build your own test programs)

Project Report

A report called <TeamName>.docx
This report must contain some content for every section. See the next slide for sections

Progress Reports

Report from each project team, every 2 weeks, explaining work done

Presentation

Present final results via Zoom

See later slides for details of each item

Final Project Report : Sections

- Team
  - List each team member
  - Who did what?
  - How you did source control, shared code, integration
- Description
  - 1 page: design, any neat algorithms used
- Specification
  - 1 page saying what your program does
  - How it works – its design
- Test Plan
  - 1 page. How many tests? What do they cover?
- Problems
  - Any parts you did not complete?
  - Any known bugs remaining in the final checkin
- Schedule
  - List of tasks with time estimate and owner

Progress Reports

Due Date:	Thursdays, 2 weeks apart (see Canvas calendar)
To:	me, plus your team members
From:	Team Name
Progress:	Work done in last 2 weeks Problems encountered Lessons learned? Are you on-schedule? If not, how far behind, and your plan to catch up Link to any code, project report or test results

Don’t go into crazy detail. Aim for at most 1 page!

Advice

Why Projects Fail

Poor Testing

- Incomplete test program; fails to find all bugs

Poor Planning

- Underestimate time required
- End up writing code, abandoning and starting over
- Waiting too long to start

Poor Project Management

- Must be self-directed: I won’t nag you!
- Poor or uneven division of responsibilities
- No backup or version control
- Late integration

Borrowing

- Several solutions available on the web. Don’t use them! Devise your own

How to Succeed

- Start early! Ask me questions or advice if you get stuck
- Plan! Figure out what you’re doing before you write code
- Design your APIs first
- Write a sequence of tests. Progress from easy to hard
- Don’t write the entire disassembler before testing. Write it in phases, and test that each phase works before going on to the next. (Incremental development)
- Develop a schedule: who/what/when
- Track the schedule to know how late you are running
- Remember to document the source code
- Meet with your group to sync up on Zoom (“Agile Development”)

More Advice

- Study M68000 Programmer's Reference Manual – not all of it – just the instructions you need
- Hand-assemble examples of the 20 required 68k opcodes, and 8 addressing modes, to understand what's going on
- Build a test script:
  - include examples of all 20 required opcodes
  - Include examples of all 8 addressing modes
  - Include examples of all .B, .W and .L sizes
- Use the script to test your Disassembler. Start with NOP, the simplest. Add support for more opcodes and addressing modes incrementally. Start testing early!

- Do NOT try to disassemble *all* possible 68k instructions: there are 100s!
- Do NOT try to disassemble *all* possible addressing modes. Cover only the 8 modes asked for

Technical Direction

Required Opcodes and Addressing Modes

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

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

Disassembler : Program Flow

1. IO subroutines prompt user for start and end address in memory (use hex)
2. User enters start and end addresses
3. I/O subroutines check for errors
4. Opcode subroutines disassemble word, or cannot:
    - If word is a valid opcode, pass address info to AddressMode subroutines
    - If word is invalid opcode, display the raw data in hex – tag with “d” for data
5. ‘Address’ subroutines disassemble the Effective Address field or fields
6. Repeat steps 4 and 5

Project : Specification

- Using 68000 Assembler Code, write a disassembler for 68000 machine code. Do NOT write some other language, such as C, and cross compile to 68000 assembly code!
- Support only the opcodes and addressing modes shown in the next slide
- ORG your Disassembler at $1000
- For IO, you can use only Trap 15, with Task IDs 0 to 14. No others
- Ask user for start and end addresses of the block to disassemble. Specify addresses in hex
- If program encounters invalid opcode, report the raw data, move to next word in the test, and try again
- The BRA instruction should display the absolute address of their target (rather than offset from current memory location). Eg: 60F2 should disassemble to, for example, BRA 00000104
- The display should have 3 columns: Address, Opcode, Operands

How to Test Your Disassembler

1. Launch EDIT68K. Use “Open File” to load your test file (eg: Test.X68)
2. Assemble your test code (F9). Fix any bugs until it assembles cleanly. This will create Test.S68

1. Launch EDIT68K. Use “Open File” to load your Disassembler. Let’s call it “DASM” for short
2. Assemble your DASM (F9). Fix any bugs until it assembles cleanly
3. Choose the “Execute” button on the “Assembler Status” popup
4. Click “File|Open Data” and choose your test file (eg: Test.S68)

5. See where the “data” is loaded
6. Run DASM
7. Should show all disassembled data onscreen

Grading

You should write tests to make sure DASM works as expected

At some point into the project, I will issue a test program called **TestDasm.X68**. You should run your DASM over this file and save the results to **<TeamName>-Results.txt**. Those results will form a large part of your final project grade.

Rubric

Items	Points
Disassembler • Opcodes • Address • IO • Quality – eg, comments!	 • 25 • 30 • 5 • 5
Project Report • Team • Description • Specification • Test Plan • Problems • Schedule	 • 3 • 5 • 5 • 5 • 3 • 5
Progress Reports	9
Total	100

Penalty Points

IO

- If program does not print address of every instruction: -1 point
- If program does not handle illegal user input (invalid addresses): -1 point

Opcodes

- If program crashes on test script: -5 points
- If program disassembles wrong opcode: -1 points (per opcode)

Addresses

- If program disassembles wrong effective address: -1 point (per opcode)