Disassembler Project Description

by Team A: Orpal Nagra, Yuunbum Lim, Gihwan (Kris) Kwon

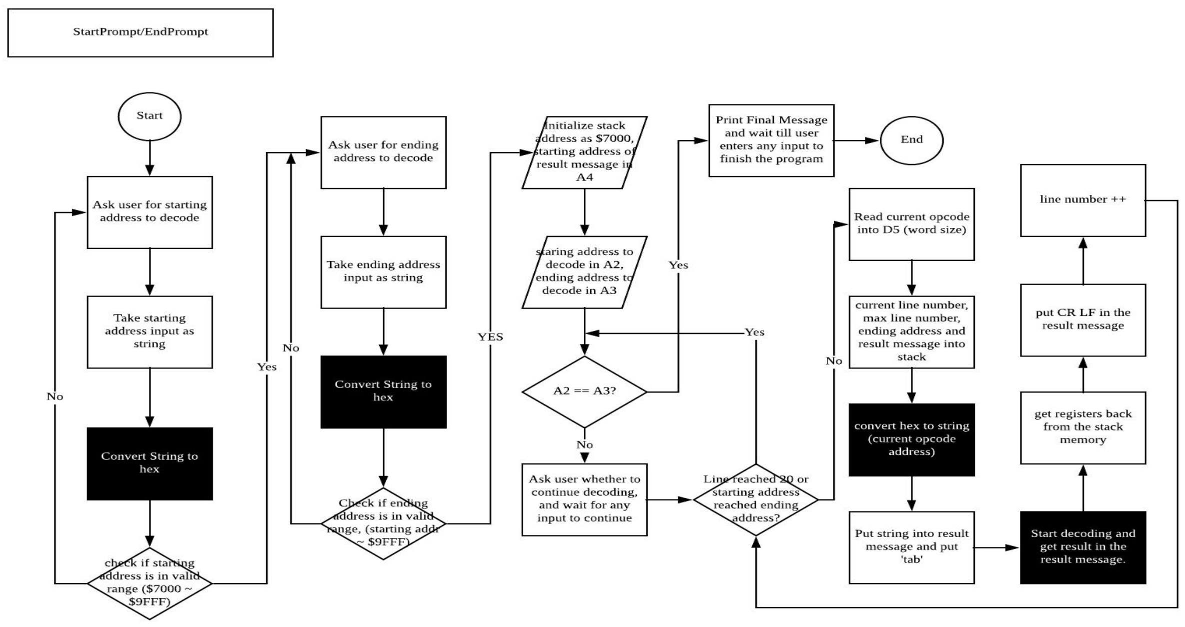
Our Approach

After forming teams, we met and gathered: project requirements, a high level approach and design, project goals, and plan of attack. This is a big project and requires careful planning and execution due to the tight time schedule and the complexity of the components involved. What came out of this? The foundational pieces of the project- I/O, Opcode Decoding, EA- and how to approach them.

I/O

The heart of the project is the I/O. This is what allows us to interact with the user, take an input and translate it into a format from which we can decode, and present the output to the user. It is the most important part of the end experience for the user.

- **Display of line**
  - Memory location
  - Opcode
  - Operand
- **I/O Flow**
  - The user is prompted to enter the starting address- allowed range is from $7000 to $9FFF. Convert to hex and verify range.
  - The user is prompted to enter the ending address- input must be > starting address and < = $9FFF. Convert to hex and verify range.
  - Prompt user to press enter to begin decoding
  - Decode opcodes
  - Print results as user presses enter
  - End program
- **I/O Flowchart**


StartPrompt/EndPrompt
Start
Ask user for starting address to decode
Take starting address input as string
Convert String to hex
check if starting address is in valid range ($7000 ~ $9FFF)
No
Yes
Ask user for ending address to decode
Take ending address input as string
Convert String to hex
Check if ending address is in valid range, (starting addr ~ $9FFF)
No
YES
Initialize stack address as $7000, starting address of result message in A4
staring address to decode in A2, ending address to decode in A3
A2 == A3?
Yes
No
Ask user whether to continue decoding, and wait for any input to continue
Line reached 20 or starting address reached ending address?
Yes
No
Print Final Message and wait till user enters any input to finish the program
End
Read current opcode into D5 (word size)
current line number, max line number, ending address and result message into stack
convert hex to string (current opcode address)
Put string into result message and put 'tab'
Start decoding and get result in the result message.
get registers back from the stack memory
put CR LF in the result message
line number ++

Opcode Decoding

After taking the input, decoding for the supported operations occurs. This section acts like a filter- the bits are checked based on their format and branched to the appropriate functions to decipher. If the bits don't fall into one of the specifications, they are not branched and an invalid operation is printed to the result in the format "7000 DATA $WXYZ.". An example of the way this works is a like a coffee filter- the operands that are supported pass through the filter and others are kept out and branched to invalid. This filter gets more and more precise as the functions are passed down.

- **Opcode Flow**
    - Check first four bits of opcode (first level filter)- branch to appropriate functions that have the same first four bits
    - Check next four bits of opcode (second level filter)- branch to functions that use the same eight bits
    - Branch to final functions (last level of filter)- decode based on differences found for the opcodes in their respective formats
    - Add the decoded message to the result string
- **Opcodes that the disassembler supports**
    - RTS, NOP
    - JSR, LEA
    - OR, ORI, EOR
    - BCLR
    - CMP, CMPI
    - MOVE, MOVEA, MOVEM
    - NEG

    - MULS, DIVS
    - SUB, SUBQ
    - ADD, ADDA
    - BRA, BCC, BCS, BVC, BGE, BLT
    - ASL, ASR
    - LSL, LSR
    - ROL, ROR

Effective Addressing (EA)

Part of the decoding includes getting the effective address for the operands that have it. This is used to help determine the source, destination, or both for most operands. The EA is unique to individual operands and they allow different types of modes for the effective address. For example, some EA functions don't allow the postincrementing address register (An)+, while others do.

- **EA Flow**
    - Utilize bit masking and shifting to get the mode and register bits stored in D1 and D0
    - Check mode to determine what the source or destination register is
    - Add appropriate characters to result string
    - Get register number from D0 and add to result string
        - For word or long, convert string to hex and print out
- **Addressing modes that the disassembler supports**
    - Data Register Direct: Dn
    - Address Register Direct: An
    - Address Register Indirect: (An)
    - Address Register Indirect with Post incrementing: (A0)+
    - Address Register Indirect with Pre incrementing: -(SP)
    - Immediate Data: #
    - Absolute Long Address: (xxx).L
    - Absolute Word Address: (xxx).W

Notes

- We have also attached a PDF file containing detailed flowcharts to demonstrate the flow of the program, including flowcharts for the supported operations