Team Cerberus

Disassembler

1) Program description:

The idea of the program is to write a disassembler (inverse assembler). The program prompts user for a starting and ending address in memory, after that the user enters starting and ending addresses for the section of memory to be disassembled. I/O checks for errors and if address are correct, sends address in memory to OP-Code. OP-code part decodes the word (if legal), and send the EA information to the EA part. EA part of the program decodes the EA fields and, if legal, prepares the operands to display. Once the instruction is displayed, the process repeats itself. All the illegal code is sent to the "bad code" section.

We used Scrum as our software development methodology we had meeting twice a week. We also constantly communicated using email, Google drive, Facebook and phone messaging. Whenever there were debugging issues we would not hesitate to call each other to figure out the problem and clear it as quickly as possible.

We separated the program into main 3 parts. The first part was I/O (input/output). The I/O person dealt with inputting the test data into the program and outputting it once it has been decoded. The second part was Operation Decode. This Op-code person dealt with reading each instruction from memory and determining what kind of operation it is. Once the operation has been determined, the last part come in; the Effective Addressing. The Effective Address person received the data from the Op-code person and determined the rest of the requirements to be decoded.

2) Specification:

The I/O person welcomes the user by displaying a welcome message. It asks the user for the starting address. The address that the user enters needs to be in hexadecimal format. In order to make sure that it is read as a hexadecimal value, the I/O person accepts the input by the user as a string, and reads through each byte (each character is 1 byte long in ASCII) as a character, stores it as its equivalent hexadecimal value. After storing it as a hexa value, it asks the user for the ending address and does the same thing with the ending address. After saving the hex value for the ending address as well, the I/O person checks if the the ending address is greater than or equal to the starting address. If it is less than the starting address, it displays error in range and ends the program. The user will have to re-run the program to start over. The I/O person also displays an error if the user makes an invalid entry that cannot be read as a hexadecimal value like - '80GH'. The I/O person keeps count of the number of instructions printed in memory address at the end of the program, I.O person is calling this address as INST_COUNT. Before printing any instruction, it checks if the value insideINST_COUNT if it reached 10. If it reached 10 - it displays a message asking the user to press enter to continue to read the next instructions. Once the user presses enter - it sets the value in INST_COUNT to 0 again to keep count of the next 10 instructions. If the value inside INST_COUNT is less than 10, it prints the memory address of the instruction. Since the address always has to be printed in the 32 bit format - a mask is set in the beginning of 'F0000000'. This mask is Anded with the starting value of every instruction. After the anding we would get the 1st byte of the instructions. LSR of 4 bit is executed on the mask at every new loop - to get the next byte value. The value itself is also LSR by 28 bits after the 1st loop and then by 24, then, 20(keep subtracting by 4 every time we go back to the loop). We keep a count to see how many times this loop is executed. The total number of times this loop needs to go on is 8 - because we have to display 8 byte value. Once the count reaches 8, we display the memory address. Then the program loops to the decoder - if the

decoder is able to decode it goes to the loop for printing the instruction. If the op code/EA person cannot decode then they loop to bad code section. The I/O person displays data in the bad code section, and displays the word value inside the memory location in hexadecimal format just the way we displayed the memory address but this time since it is a word alue our mask begins at 'F000'. To ensure that it is a word value, the I/O person checks the difference between the starting address of the instruction and the address where the op code/EA person ended decoding. If it is less than 2, the I/O person adds to it to make the difference equal to 2. Once the difference is equal to 2 it displays the word value inside it. Once the instructions are displayed till the end of the memory address the user requested, the I/O person asks the user if he/she would like to continue. If the user presses 'Y' or 'y', the I/O person asks for the starting address again. If the user presses 'N' or 'n', then the program ends. For all the inputs that the user will enter - the I/O person considered both the upper and lower case scenarios. So the user can input in any case and it would not change the way the program would work.

The I/O person has now passed the new starting address of the instructions to the OP decoder. Here the program will look at the starting address stored in a address register and read only the first byte to be able to identify what type of instruction it is. The program then looks at only the first four bits of data of the assembled code and subroutines to the section that contains the matching four bits (ex. SECT_0000). These section contain multiple possible opcodes that the instruction could be, so the program has to look at the next 4 bits of the assembled code to see which opcode the instruction actually is (ex. NEG_SEC). In our program the MOVE and MOVEA opcode is special because the size of the instruction is used to determine which section it should be placed in (ex MOVEB_SEC or MOVEAB_SEC).

Once the program has jumped to the correct subroutine section it will then store the letters to spell out the opcode into a location in memory (ex. ‘N’ ‘E’ ‘G’). After it has stored the correct opcode spelling in memory, the program will prepare the instruction pointer to point to the correct byte that the EA section has to decode, and also create a new pointer that points to where the extra data of the instruction would be stored if the EA section needs to decode it for absolute data or immediate data.

For ORI, SUBI, EORI, CMPI, NEG, NOT, ADDQ, ASR, ASL, LSR,LSL, ROL and ROR after identifying the op-code word the program separates the 2 bites that identify the size of the data by shifting and comparing. If there is no specific size for that instruction, the program identifies that it’s not a valid entry and jumps to “bad code” section. After identifying the size of the data, the program moves on to the effective address part.

In effective address the program for ORI, SUBI, EORI, CMPI, NEG, NOT, ADDQ identifies the mode of the data by shifting and manipulating to separate the 3 bits needed after copying it to another data register (not to lose it for further manipulation). After the mode is identified by the 3 bits, the program then moves on to the last 3 bits to identify the register. When looking at the mode and register if it is an absolute data or immediate data then the program looks at the extra data pointer and reads the correct amount of bytes it needs to spell out the contents of the instruction, while moving the extra data pointer along.

Some of the other op-codes required a bit different approach. MOVE and MOVEA had a different syntax. Although the main approach was the same, other functions were required to decode the op-code word and size.

The op-codes DIVS, SUB, SUBA, EOR, CMP, CMPA, MULS, AND, ADD, ADDA, LEA, BTST, MOVE, MOVEA required separate functions to identify the register. As before the approach was similar in the sense of separating the bits needed by manipulating the binary data and identifying whatever was required. But there was a different logic involved.

Some of the special cases were MOVEM, ASR, ASL LSR, LSL, ROL and ROR. These required a lot of additional coding and manipulation.

We also had a chart that included all the address registers and data registers used not to override each other's data.

Dn/An	
D2	Store Ending address

D4	Store Starting address
D7	Error Flag
A6	point to read the instruction - by end point to new instruction
A5	Address memory location of printer (started _____)
D5	How many bytes to read from the address memory location pointer
A2	Pointer to where data is stored (if data is stored)
A4	Starting point of current instruction - change to A6 later
D4	Keep track of length of each instruction in memory
D6	store length (temporarily) for printing. Anyone using this initialize to 0 first
D3	store number of instructions printed
A3	data register for size comparison

3) Test Plan:

We decided to first test if op-codes and effective address work together. A lot of testing has been done while combining op-codes and effective addresses. After these two parts worked pretty well together, we started testing it with the I/O code.

We wrote a separate test code to test the program after we were done. The test code had all the required op-codes and some additional ones. We had some issues with the immediate address part, that were resolved during testing sessions and combining all the codes together.

We had simple coding standards: write clean code and comment mainly the parts that can be confusing. Also one of our main goals was to reuse as much code as possible to avoid redundancy.

4) Exception report:

One of the bugs is that MOVE prints out as MOVEA if the destination is an address register. So it jumps straight to MOVEA. Same thing happens with the SUB opcode. If the destination is address register, it jumps straight to SUBA. Another case with SUB is that for immediate data, it jumps straight to SUBI.

When using MOVEM, we print out the data register first, then the address register, even if in the test code it's the other way, as the order doesn't really matter.

Another one is that for MULU the size doesn't matter, as it's not identified in the machine code, therefore there's now way for us to figure out the size and print it.

We have a view problems that we encountered:

1) When running the test program it has to start at 7000 (no '$' required). If you test it anywhere else you will get bad data for some of the commands later on.
2) Also it should be noted that we have included the opcode SUBQ and ANDI so that it causes some of the other required opcodes to run properly!

We were able to complete all the opcodes we were required to finish, but we did not get to thoroughly test if the bad code get's placed into the bad section correctly. The only bad code we have tested so far is: SWAP, NOP, CLR. Also our program does not identify bad code if they are too similar to the other required opcodes such as EXG, which is very similar to AND.

5) Team assignments and report:

We divided the program into three main parts: The I/O, Op-code and EA. Pallakh was responsible for the I/O, Stuti for Op-code and Ruzanna for EA. Since op-code and EA of the program are tightly connected Stuti and Ruzanna coded most of the EA part together. We did a few online peer-review sessions to code and connect the code using collabedit.com. At the same time we discussed any bugs and issues that arose with Pallakh. After the EA and Op-code section were working perfectly with each other, Pallakh took the combined code and added her I/O part and tested the program. If any bugs arose while she was testing the whole program she would contact Stuti to review the problem.

We tried to separate the tasks evenly (Stuti 33%, Pallakh 33%, Ruzanna 33%), but if somebody needed help the other group members were there to help out. Op-code was the middle part, so Stuti seemed to be most aware of what is happening on other two sides of the code. In the end, the final percentages were as follows: Pallakh did her 33%, Stuti did her ~36% with some parts of helping out Ruzanna, who did ~30% of the work.