SWE-514: Computer Systems

Project Document

December 27, 2021

Author	*Student ID*
Ali Kenan Yağmur	2021719165
Mümtaz Eralp Ünver	2021719162

Contents

1 Assembler **3**
- 1.1 Identifying Labels . . . 3
- 1.2 Addressing Modes Detection . . . 3
- 1.3 Combine All with Detected OpCodes . . . 4

2 Executor **6**
- 2.1 Global Variables . . . 6
- 2.2 Utility Functions . . . 6
- 2.3 Main Loop and OpCode Functions . . . 7

3 Exceptions **8**

1 Assembler

First of all, we have listed the steps to be done in the assembly part of the project as follows:

- Identifying the labels and assigning the appropriate addresses to the labels.
- Correct detection of addressing modes
- Detection of operation codes.
- Neatly combining the specified fields and converting them to machine code.

1.1 Identifying Labels

In order to determine the Label points, we first divided each line into instructions and concluded that those that do not have **'NOP'** and **'HALT'** OpCodes are LABELs.

Afterwards, it was deemed appropriate to trace the assembly code twice in order to keep the data that the detected LABELs correspond to the required addresses and be called at the relevant times.

```
################## PASS-1 ######################
#
# First pass for determining the labels and their addresses.
#
counter = 0
for line in lines:

    # Split the lines for spaces and get the instructions.
    instruction = line.split(" ")

    # Instruction halt does not require an address we should skip it.
    if instruction[0] == 'HALT':
        continue

    # check the NOP instruction
    if instruction[0] == 'NOP':
        counter = counter + 1
        continue
    # Here we check the labels
    if len(instruction) < 2:
        # If we find the label drop the ':' seperator from it
        # And add it to the labels dictionary.
        labels[line[:-1]] = hex(counter*3)[2:]
    counter = counter + 1
```

1.2 Addressing Modes Detection

In this section, it is determined which addressing mode the relevant operands correspond to and calculation of the necessary hexadecimal equivalents. Here our strategy:

- Starts with apostrophe → immediate
- Starts with square brackets
  - If contains register names: → regmem
  - If not register names: → memory
- Operand is register name → register
- Operand is label name → immediate
- Remaining things → immediate

```
# - Check the operands and get their addressing mode names.
# - From their addressing modes compute the operand bits as hex.
# Args:
#   operand (string)
#
# Returns:
#   addressingMode (string): key of dict(addressingModes)
#   convertedHex (string): hexcode of converted operand
def convertOperand(operand):
    addressingMode = 'nill'
    convertedHex = ''
    # Check for immediate chars
    if (operand.startswith("'")):
        # Seperate it from their apostrophes and convert it to hexadecimal
        addressingMode = addressingModes['immediate']
        convertedHex = hex(ord(operand[1:-1]))[2:]

    # Check for memory related addreses
    elif (operand.startswith("[")):
        addressingMode = addressingModes['memory']
        withoutBrackets = operand[1:-1]
        # They can refered with register. Check here.
        if withoutBrackets in ['A', 'B', 'C', 'D', 'E']:
            convertedHex = registers[withoutBrackets]
            addressingMode = addressingModes['regmem']
        else:
            convertedHex = withoutBrackets
    # Check here for register addressing modes.
    elif (operand in ['A', 'B', 'C', 'D', 'E']):
        addressingMode = addressingModes['register']
        convertedHex = registers[operand]
    # Check remaining immediate data.
    elif (operand.isdigit()):
        addressingMode = addressingModes['immediate']
        convertedHex = operand
    # Lastly they are labels.
    else:
        addressingMode = addressingModes['immediate']
        convertedHex = labels[operand]
    return addressingMode,convertedHex
```

1.3 Combine All with Detected OpCodes

In this part of our assembly parser, we combined the operand and addressing modes determined beforehand and translated into hexadecimal with our properly specified instructions and translated them into machine language.

In order for the relevant parts to be combined properly, we converted the hexadecimal parts we calculated earlier to binary as follows and added them end-to-end:

	OpCode	Addressing Mode	Operand
Hex	1C	1	0003
Partition	xxxxxx	xx	xxxxxxxxxxxxxxxx
Binary	011100	01	0000000000000011
Result	710003		

Table 1: Combiner

Here is related code:

```
################## PASS-2 ####################
#
# If we encounter with label we will use the labeled addresses for them.
#
counter = 0
for line in lines:
    # Split the lines for spaces and get the instructions.
    instruction = line.split(" ")

    # Check the opcodes with operand
    if len(instruction) > 1:
        # Get the opcode from its name
        opcode = opCodes[instruction[0]]
        # Get the operand and addressing mode as an hex.
        addr, operand = convertOperand(instruction[1])
    else:
        # Check for 'HALT' instruction
        if instruction[0] == 'HALT':
            opcode = opCodes[instruction[0]]
            # Manually gave the addressing mode and operand
            addr, operand = '0','0000'
        elif instruction[0] == 'NOP':
            opcode = opCodes[instruction[0]]
            addr, operand = '0','0000'
        else:
            # Check for label instruction
            continue

    # Converting the hexadecimals to binary is more convenient.
    # Because we concat two binary strings with different lengths.
    resultBinary = opCodeToBin(opcode) + modeToBin(addr) + operandToBin(operand)
    decimal = int(resultBinary, 2)
    # Output list contains with filled zero hexadecimals.
    output.append(str(hex(decimal)[2:]).zfill(6))
    counter = counter + 1
```

After all these works, the output of this python script is read in the same way as the name of the relevant file, and the binary output of the part ".bin" is created.

In the content of the file, for example, there are instructions for each 3 bytes listed below:

080041
0D0003
080039
0D0002
080004
0D0004
380000
710003
090003
0E0002
190003
190002
190002
1D0004
380000
500015
6D0005
710005
040000

2 Executor

2.1 Global Variables

In this part of the project, first of all we created some global variables for storing necessary data like:

- **program_counter** : Store the current instruction memory address.
- For proper conversion we decided to store important signed and unsigned values as integer:
  - **unsignedMAX** = 65535
  - **signedMAX** = 32767
  - **signeMIN** = -32768
- Condition Codes.
  - **zf = 0 , cf = 0 , sf = 0**
- **registers** : Stores current register data.
- **stack_pointer** : Stores the current stack pointer value as integer.
- **memory** : Stores the current memory address-value as dictionary. In Table-2 we can see the hierarchy of the our executor program memory.

0x0000	INSTRUCTION-1
0x0003	INSTRUCTION-2
.	.
.	.
.	.
0x0027	INSTRUCTION-(N-1)
0x002A	INSTRUCTION-(N)
0x002D	DATA-1
0x002F	DATA-2
0x0031	DATA-3
0x0033	DATA-4
.	.
.	.
0xFFFD	STACK TOP
0xFFFF	STACK BOTTOM

Table 2: Memory Hierarchy

2.2 Utility Functions

After initialization of the global variables we created some utility functions for relevant conversions like:

```
#######----- UTILITY FUNCTIONS -----#######

# Convert unsigned integer to signed.
def utos(unsigned):
    signed = unsigned - 2**16
    return signed

# Convert signed integer to unsigned
def stou(signed):
    unsigned = signed + 2**16
    return unsigned


```

```
# Convert hexadecimal instruction bytecodes to the binary representations
def hexToBinary(num):
    scale = 16
    numOfBits = 24
    return bin(int(num, scale))[2:].zfill(numOfBits)

# Seperate the OPCODE - MODE - OPERAND from the binary representations
def parseBits(num):
    binary = hexToBinary(num)
    opcode = binary[0:6]
    mode = binary[6:8]
    operand = binary[8:]
    return opcode, mode, operand

# Convert binary to hexadecimal with filled up to 1 byte.
def binaryToHex(binary):
    decimal = int(binary, 2)
    return hex(decimal)[2:].zfill(2).upper()

# Convert decimal to hexadecimal with filled up to 2 bytes.
def decimalToHex(decimal):
    if decimal < 0: return hex(stou(decimal))[2:].zfill(4)
    return hex(decimal)[2:].zfill(4)

# Convert hexadecimal to integer number.
def hexToDecimal(hexa):
    return int(hexa, 16)

# ----------------------------------- #
```

These utility functions mainly used everywhere in our executor script. All the calculations done in decimal integers and then converted to their hexadecimal representations and if necessary saved in hexadecimal format.

2.3 Main Loop and OpCode Functions

The most important part of our implementation and this project. Every opcode has its own function. After related binary file read line by line, the executor starts with **program_counter** = '0000' as hexadecimal value and goes that memory location read the instruction from there.

After the instruction comes from the related memory address, it will be parsed into fields which we described in assembler section.

- First six bits are opcodes. [0:6]
- After opcodes, next two bits are addressing modes. [6:8]
- Then the last 16 bits are the operands. [8:]

After these partition for simplicity we convert the binary partitions to the decimal integer values and make the calculations in this condition.

Since all of the OpCode functions are in the source code with their explanations and will take up a lot of space in this report, we thought it would be more appropriate to examine the source code.

In the main loop section of the code, every instruction will be parsed and related opcode function will be called here. Every opcode function can change the global variables according to the execution order.

If the **'HALT'** instruction comes for the execution. Main loop will end and the program exit.

3 Exceptions

1) This program should not be tested with wrong inputs.

2) This program does not have any syntax check mechanism. If any wrong typed input were given, scripts can work in unexpected behaviour.

3) If the given input have an infinite loop behaviour, user should exit or kill the process manually. (Can use SIGINT or SIGKILL)

4) This code just simulates the given instruction set sequentially. The user should not expect a behaviour like pipelined execution.