CMPE 230 - Project 2 Documentation

Karahan Sarıtaş (2018400174)
Cahid Arda Öz (2019400132)

June 4, 2021

Contents

1 Problem Statement

We are asked to implement an Assembler and an execution simulator for a hypothetical CPU called CPU230. CPU230 consists of an accumulator register (A), four general purpose registers (B, C, D, E), a program counter (PC), a stack pointer (S) and flags (ZF, CF, SF). Each register consists of 16 bits. Our CPU fetches the instructions from a 64KB memory segment.

2 Solution

First, we designed our CPU and Memory. Upon creating a Memory object, one should load it with the necessary amount of bytes, 64KB in our project. Memory simply reads a memory cell or writes into it. It takes its instructions from CPU to read and write. CPU is like the brain of the system. It holds all the registers, fetches the instructions and implements them until it encounters a ”HALT” instruction. After fetching the instructions from Memory, CPU decodes the instructions and checks the opcode. For each opcode, different operation is performed. CPU has a Memory object as one of its fields. It utilizes this Memory to read and write.

To perform the operations such as ADD, INC, SUB, DEC, SHL, SHR etc. we created a new class named Misc. Unlike our classes CPU, Memory and Assembler, Misc doesn't represent an object. We use it as a storage of different operations. When we have to perform one of the arithmetic operations given below, we call the necessary function from Misc. It is also able to switch from binary representation to hexadecimal representation and vice versa. When necessary, it can switch from unsigned binary representation to two's complement.

We then designed our Assembler. It simply iterates through the .asm file and converts each instruction into a hexadecimal number representing 3 bytes. To store the memory addressed marked by labels, it iterates the .asm file twice. After generating all instructions, it produces the .bin file to be processed by CPU.

Lastly, we created cpu230assemble.py and cpu230exec.py classes. These are simply the executable classes and we use these to run our program. They perform the main tasks such as assembling, creating a memory, loading it and running the CPU.

3 Implementation

Our implementation consists of three objects:

1. Memory, CPU and Assembler
2. A ”static” class named as Misc
3. Main classes to run the program, named as cpu230assembler and cpu230exec.

First, Assembler is utilized and a binary file is produced from .asm file. Then a Memory object is created and loaded with this binary file. A CPU object is created and previously created Memory is given to it.

Once the program is loaded, CPU follows the fetch-decode-execute cycle using the Memory. During the execution, CPU uses Misc functions to perform necessary arithmetic operations. Program terminates when it encounters a ”HALT” instruction and the .txt output is generated.

Now, let's have a deeper look at the methods of our classes.

3.1 Assembling

Assembler consists of two main methods and some helper methods. Main methods are assemble() and write(). Assemble() simply parses the .asm file and generates corresponding instructions in hexadecimal format. If during this procedure, it encounters with a problem such as a syntax error or a semantic error (incompatible addressing mode etc.) it stops the parsing and return false. Then cpu230assemble halts the program and prints out an error message. If there is no problem with the .asm file, cpu230assemble calls the write() method of Assembler and a binary file is generated.

In addition to these main methods, Assembler has helper methods, namely generateInstruction(), toInstruction(), register(), toOperand(), processHexadecimal(), and toOpcode()

1. assemble(): Iterates through the .asm file twice. At first iteration, stores the memory addresses marked by labels. At second iteration, parses each line and produces instructions in hexadecimal format. Instructions are stored in a list.

2. write(): After assemble() is called, this method is called by cpu230assemble. It simply writes the instructions stored in the instruction list into a binary file.

3. generateInstruction(opr, operation): It takes two strings representing the operation and the operand. Using other helper methods toOperand() and toOpcode(), it creates corresponding 6-bit opcode, 2-bit addressing mode and 16-bit operand. Then it calls toInstruction() to switch from 24-bit binary representation to 6-character hexadecimal format. Then stores the instruction in instruction list.

4. toInstruction(opcode, addressingMode, operand): Takes opcode, addressingMode and operand in binary format and returns corresponding hexadecimal instruction.

5. register(opr): It takes the operand and tries to identify which register it is. This method is called when we are looking for a register. If we cannot identify the register, register() simply returns false and Assembler stops processing.

6. toOperand(opr): It takes the operand and returns its binary format. Operand can be a register, a memory address, an immediate data. An immediate data can be a hexadecimal number, a character bounded by quotation marks and a label.

7. toOpcode(operation): It takes an operation and returns the corresponding binary format.

8. processHexadecimal(hex): This method is used to process hexadecimal with 1, 2, 3, 4 or 5 digits. It ignores the initial zeros and returns a 4-digit hexadecimal value.

3.2 Execution

CPU fetches instruction from its memory and performs necessary actions. With the help of Memory, it can read and write into the memory cells. It updates the flags after each operation accordingly. It consists of two main methods, runProgram() and writeFile(). These methods are called by cpu230exec. CPU has other methods, namely getNextInstruction(), decodeInstruction(), pushStack(), popStack(), getRegister(), setRegister(), setFlagsWithList(), read(), write(), and runInstruction().

1. runProgram(): As stated above, this method utilizes fetch–decode–execute cycle to execute the instructions. It calls getNextInstruction(), runs it with runInstruction() and does the same thing for next instruction.

2. writeFile(): After executing the whole binary file, writeFile() method is called under the cpu230exec. It creates a .txt file and writes the results of PRINT operations into it.

3. getNextInstruction(): It calls the readInstruction() method of Memory to fetch the instruction and increments the program counter (PC) by 3.

4. decodeInstruction(instruction): Since we are provided with 6-digit hexadecimal instructions, we have to decode them into opcode, addressing mode and operand. decodeInstruction() is used for this purpose. It is called just after getting the instruction from Memory.

5. pushStack(word): It writes the given character to the memory cell pointed to by the Stack Register (S), then decrements it by 2. (From high address to low address)

6. popStack(): It returns the last added element to the stack. Stack Register is incremented by 2.

7. getRegister(i): According to the given parameter i, it returns the corresponding register. Each register is labeled with a digit from 0 to 5 for simplicity. If i is not in this boundary, it returns Stack Register.

8. setRegister(i, operand): First, it finds the related register according to the given parameter i. And then loads the operand onto it.

9. setFlagsWithList(flags): It takes a list of flags, and updates the flags with that list. If length of the flags is 2, it represents SF and ZF. If it is 3, it represents SF, ZF and CF.

10. read(addressingMode, operand): According to the addressing mode, it reads the relevant memory cell. If addressing mode is 00, operand is an immediate data. If it is 01, operand is a register. If it is 10, operand is a register and we should read the memory address pointed to by that register. If it is 11, operand is a memory address and we simply read that address to get the real operand to use in our operation.

11. write(addressingMode, operand, toWrite): It takes an addressing mode, an operand and content to be written. According to the addressing mode, it finds the memory address or register and writes the content onto it by using write() method of Memory.

12. runInstruction(instruction): It takes an instruction, decodes it and performs the operation. It uses Misc functions to perform arithmetic operations. When necessary it sets flags, reads from register/memory or writes onto register/memory by calling appropriate methods.

Memory simply represents our RAM. It consists of 1-byte memory cells. For our project, cpu230exec creates a Memory consisting of 64KB and loads it with the binary file produced by our Assembler. It consists of four methods, namely loadProgram(), write(), read(), readInstruction().

1. loadProgram(file): It takes a binary file and generates n many memory slots. In our project, n is 65536 (64KB). It reads the file and fills the memory slots one by one.

2. write(address, operand): It takes a decimal address and a decimal operand. Then it converts the operand into hexadecimal. Then writes this hexadecimal value to the given operand.

3. read(address, nBytes): It takes an address and number of bytes to read. If number of bytes to read is just one, it just reads the address and returns the content of it. If it is bigger than one, it starts from the beginning of the address and reads the memory cells one by one. Then it returns the concatenated content.

4. readInstruction(address): This methods calls the read() method with nBytes = 3.

3.3 Reading Input

To implement "READ" call, we used default input() method from python. Whenever a "READ" call is made, execution stops and the program starts waiting. User has to enter a single character and press enter to continue execution.

3.4 Miscellaneous

Misc.py is used as a storage of different functions for decimal-binary-hexadecimal conversions and arithmetic operations. These functions are called by CPU. Misc.py includes decToBin(), hexToDec(hex), hexToBin(), binToHex(), twos_comp(), addToHex(), subFromHex(), notOperation(), xorOperation(), andOperation(), orOperation(), shrOperation(), shlOperation(), and addOperation().

1. decToBin(dec): Converts a decimal number to a binary number.

2. hexToDec(hex): Converts a hexadecimal number to a decimal number.

3. hexToBin(hex): Converts a hexadecimal number to a binary number.

4. binToHex(hex): Converts a binary number to a hexadecimal number.

5. twos_comp(val, bits): Converts a binary number to a signed decimal number.

6. addToHex(value, add): Adds the decimal add to the hexadecimal value and returns hexadecimal number.

7. subFromHex(value, sub): Subtracts the decimal sub from the hexadecimal value and returns the hexadecimal number.

8. notOperation(dec): Performs not operation on the decimal number, sets the flags accordingly and returns them to the CPU.

9. xorOperation(dec1, dec2): Performs xor operation on the decimal numbers, sets the flags accordingly and returns them to the CPU.

10. orOperation(dec1, dec2): Performs or operation on the decimal numbers, sets the flags accordingly and returns them to the CPU.

11. shrOperation(dec1): Shifts the decimal one bit right, sets the flags accordingly and returns them to the CPU.

12. shlOperation(dec1): Shifts the decimal one bit left, sets the flags accordingly and returns them to the CPU.

13. addOperation(dec1, dec2): Performs add operation on the decimal numbers, sets the flags accordingly and returns them to the CPU.

4 Testing

To test our program, we crated several testcases. Following is an example from our pool of testcases:

```
LOAD 1c2
STORE B
LOAD 159
STORE C
LOAD 198
STORE D
LOAD 1c7
STORE E

SHR D
SHR D
SHR D
PRINT A
PRINT B
PRINT C
PRINT D
PRINT E
PRINT A
PRINT B
PRINT C
PRINT D
PRINT E
DEC 17f
DEC 17f
DEC 17f
DEC 17f
DEC 17f
DEC 0c6
DEC 0c6
DEC 0c6

HALT
```

The testcase consists of 2 segments. First segment is where we store random values to all registers. Range of random values is [0-(2^9-1)]. Reasoning behind this range is we wanted store values close to the values which have corresponding ascii characters.

4.1 Flag Test

We wanted to take our testcases further. In its current state, our testcase can not test the flags in a comprehensive way. To do this, we designed the jump test:

```
TEST11:
    PRINT '#'
    JZ JZ11
        PRINT 'a'
        JZ11:
    JE JE11
        PRINT 'b'
        JE11:
    JNZ JNZ11
        PRINT 'c'
        JNZ11:
    ...
    JAE JAE11
        PRINT 'h'
        JAE11:
    JB JB11
        PRINT 'i'
        JB11:
    JBE JBE11
        PRINT 'j'
        JBE11:
    PRINT '#'
JMP LABEL11
```

The test starts when the code jumps to TEST11. Each jump operation is called one by one and a character is printed or not printed depending on whether the jump is performed. To call this test, we updated every line in the original testcases. Every instruction ”<operation><operand>” is updated in the following way:

```
    PRINT ' '
    PRINT '0'
<operation> <operand>
    JMP TEST11
    LABEL11:
```

4.2 Print Guard

We also wanted to be able to observe the contents of registers in our tests. To do this, we used the ”PRINT” statement. The problem with ”PRINT” is that the operand does not always have a corresponding ascii value. To overcome this issue, we wrote the Print Guard:

```
    PRINT '@'
    LOAD C
    CMP 7F
    JAE PRINT2
    CMP 20
    JB PRINT2
PRINT A
    PRINT2:
```

The Print Guard loads the given register (C in our case) to A, then compares it with 7Fh and 20h. "PRINT" statement is called only if the operand is in the range [32, 126]. This way, we are able to observe the contents of registers without causing any errors.

Combining the Print Guard with Flag Test using a test generation script; we were able to swiftly generate several cases, each with thousands of lines.

5 Conclusion

Looking at the testcases provided to us and created by us, we conclude that our program solves the given problem. There are some improvements that come to mind:

1. Since we added necessary syntactic and semantic checks one by one to our Assembler, it became a little bit messy after a point. We could have prepared a list of possible errors beforehand and design our Assembler accordingly.

2. We had to make several transitions from binary to hexadecimal, unsigned to signed and vice versa in Misc.py. To improve the readability, we made these transitions explicitly again and again at some points. Maybe we could have spent some more time to focus on how we can eliminate some of the redundant transitions without reducing the readability.