CPU/Assembler Documentation

CPU Name:

I took CS 382 and all I got was this sucky CPU

By: Dylan Faulhaber and Jack Patterson

Partner Contributions

- **Dylan Faulhaber**
  - Made the virtual CPU in logism.
- **Jack Patterson**
  - Coded the assembler in python.
- **Together**
  - Created simple assembly program and wrote documentation.

Central Processing Unit

- **Registers**
  - 1 Program Counter register
    - Cannot be accessed from the assembler.
  - 4 General Purpose registers
    - Referred to as r0, r1, r2, r3
- **Possible Instructions**
  - STR
    - Stores value in register into data memory
    - STR Rt Rn Rm
      - Rt = register that has the data to be loaded
      - Rn = register that has the address of where in memory to load data
      - Rm = register that has the offset for the memory address
  - LDR
    - Loads value from memory into register
    - LDR Rt Rn Rm
      - Rt = register that the data from memory will be loaded into
      - Rn = register that has the address of memory where the data is being loaded from
      - Rm = register that has the offset for the memory address
  - ADD
    - Adds two numbers together and saves answer into register
    - ADD Rt Rn Rm
      - Rt = register that the result of the addition will be stored in
      - Rn = register that has the first number to be added
      - Rm register that has the second number to be added
    - Rt = Rn + Rm
  - SUB
    - Subtracts two numbers together and saves answer into register
    - ADD Rt Rn Rm
      - Rt = register that the result of the addition will be stored in

        - Rn = register that has the first number to be added
        - Rm register that has the second number to be added
    - Rt = Rn – Rm

- **Binary Encoding**
    - Each instruction has a binary encoding of 8 bits (1 byte).
        - NN MM QQ RR
            - Bits NN
                - OpCode
                - 2 bits are needed for the OpCode since there are four instructions, which means we can refer to them with binary codes: 00, 01, 10, 11.
            - Bits MM
                - Rt
                - 2 bits are need for Rt because there are four registers, which means we can refer to them with binary codes: 00, 01, 10, 11.
            - Bits QQ
                - Rn
                - 2 bits are need for Rn because there are four registers, which means we can refer to them with binary codes: 00, 01, 10, 11.
            - Bits RR
                - Rm
                - 2 bits are need for Rm because there are four registers, which means we can refer to them with binary codes: 00, 01, 10, 11.
    - NOTE: There are no immediate numbers allowed. All instructions use registers.
    - OpCode Binary References
        - 00 → STR
        - 01 → LDR
        - 10 → ADD
        - 11 → SUB
    - Register Binary References
        - 00 → Reg 0
        - 01 → Reg 1
        - 10 → Reg 2
        - 11 → Reg 3

Assembler Program

- **Instruction Writing Conventions**
  - All files of assembly instructions must have the **.txt** extension.
  - Comments
    - Can be done with a preceding //
    - Must be done on the same line of instruction. Cannot be on their own line
  - Two main sections: **.text** and **.data**.
    - .text:
      - Must be first section in the document
      - Contains all instructions (STR, LDR, ADD, SUB)
      - Only spaces are allowed, no commas or tabs
      - Instructions must be all upper case
      - Registers are referred to as **r0**, **r1**, **r2**, **r3**
        - Must use a lower case r
      - No immediate values
    - .data:
      - Contains all data that will be saved in memory at start of program
      - Only spaces are allowed, no commas or tabs
      - Only supports numbers of size byte
      - No name corresponds with the entries in this section, it is required to add only the **.byte** directive and its value separated by a space
      - Only one value is allowed for each directive
  - Example Instruction text file

```
1   .text:
2       LDR r1 r0 r0
3       ADD r2 r1 r1 // add 1+1
4       ADD r2 r1 r1 // add 2+1
5       ADD r2 r1 r1 // add 3+1
6       STR r1 r0 r0
7       SUB r1 r2 r1 // subtract 3-1
8       STR r2 r1 r0
9       LDR r2 r0 r1
10
11
12  .data:
13      .byte 1
14
```

- **Python (Assembler) - How it works**
    - The assembler was built in python. It works by iteratively parsing each line of the data.
    - The assembler provides support for comments, some error checking, and the separation of **.data** and **.text** sections into two different image files.
    - It is required that the python file is in the same directory as the **asm.txt** file which has the instructions in it. The two image files will be auto generated by the assembler and saved into the same directory. Once the two image files are generated, they can then be loaded into Logisim Evolution with the **text.txt** file being used for the instruction memory and the **data.txt** file being used for the data memory.
    - It works by first reading the **.text** section, and then the **.data** section. There are four functions that handle parsing and inserting the **.text** and **.data** information into their image files (**data.txt** for the **.data** section and **text.txt** for the **.text** section). They are:
        - Main
            - The main function handles most of the parsing for the **.text** section. It starts by reading each line of the **asm.txt** file. It first checks for comments and omits them from being read by the program. The function will then check if the **.data** section is hit and will then run the function responsible for parsing the **.data** section. If not, it then translates the line into its corresponding hex value and appends this hex value to a list.
        - addToInstructionsFile
            - A list is supplied to the function which will be turned into a matrix based on the number of lines needed. Once the correct matrix is created, the function then inserts the data in the matrix into the file (**text.txt**) with a simple for loop.
        - parseDataSection
            - This function works by creating a dictionary that stores both the directive and value. This ensures the data is valid before use. If the directive is not **.byte**, the function will break and return an error message. Once this is done, the **addToDataFile** function is called.
        - addToDataFile
            - This function performs a similar process to **addToInstructionFile**, it splits the hex values so there is no **0x** at the beginning. Once this is done, it does the same matrix splitting to ensure a proper file format with proper line breaks. It then adds all the values into the **.data** file. Since this is the last function, if the program runs successfully, a message is displayed stating that the files have been created.

- The **.text** section:
  - At each line, it splits the operation and the registers with a space as the delimiter. Once this is done, the assembler translates the operation **(ADD, SUB, LDR, STR)** into its corresponding **opcode** (binary), and the destination register / operand registers into their corresponding binary representations. The assembler then concatenates all these binary values to get an 8-bit binary sequence. Once this 8-bit sequence is generated, it is converted into hexadecimal. For example: the cycle from instruction to hex is: **LDR r1 r0 r0 → 01010000 → 50**.
  - 50 is inserted into a list and when there are more than 16 hex values, the list is split into a matrix, so the program knows to perform a line break maintaining the required format for the image file. At each line insertion, the program looks up the corresponding line value in a dictionary corresponding to a line number indicator in the program. For example, the hex values for the line numbers could be: **(00, 10 … f0)**. They are then inserted at the beginning of each line.

- The .data section:
  - Once all the instructions in the **.text** section are translated into an image file, the **.data** section is handled. It is important to notice that because the parser functions for the **.data** are called immediately from the **.text** parser, the **.data** section must be put at the bottom of the file. The directive must come before the value.