We shall understand the intricacies of an architecture hypothetized by a classmate of mine, Sathyam (and so, the hypothetical machine that works under this architecture shall be called the “Sathyam Machine” in his honour).

In this architecture, the word size is 16 bits. This means that each instruction/data occupies 16 bits. There are 8 general purpose registers (R0 to R7). The general Instruction format is as follows:

xxxxxx xxxMM xxxMM

Opcode Op2(dest) Op1(source)

The above is the general form of the machine code corresponding to an instruction. The first part is the opcode for a given mnemonic (such as ADD, SUB, etc.). This opcode is 6 bits. Each of the operands that follow the opcode take 5 bits (thus the total size of the instruction is 6 + 5 + 5 = 16 bits). The x’s in the operand indicate the Register number (if any), while MM indicates the Mode. We have the following modes:

00 => Register has operand

01 => Register has pointer to operand

10 => Immediate value (this value is stored in the next 16 bits immediately following this instruction. Assume the immediate value is always specified in hexadecimal).

The first source operand (Op1) uses 00, 01 and 10 modes. The second source operand (same as Destination or Op2) uses 00 and 01 modes. Op2 cannot use variables or addresses of them. For example: MOV A, #10 is illegal since Op2 is a variable. The destination/Op2 can only be a register or pointer to it. Thus, MOV R1, #10 is fine, and so also MOV \*R1, #10.

The following table lists some mnemonics along with their opcodes and usage: (note that X and Y could be registers or immediate values according to situation).

|  |  |  |
| --- | --- | --- |
| **OPCODE** | **MNEMONIC** | **USAGE** |
| 000000 | NEG (negation) | NEG X, Y => X = ~Y |
| 000001 | AND (logical AND) | AND X, Y => X = Y&X |
| 000010 | XOR | XOR X, Y => X = Y^X |
| 000011 | OR | OR X, Y => X = Y|X |
| 000100 | LSR (logical right shift) | LSR X, Y => X = Y >> X |
| 000101 | LSL (left shift) | ASL X, Y => X = Y<<X |
|  |  |  |
| 001010 | HLT (halt/stop) | HLT |
| 001101 | MOV (move word) | MOV X, Y => X = Y |
|  |  |  |
| 001111 | JMP | JMP label |
| 010000 | ADD | ADD X, Y => X = Y+X |
| 010001 | SUB | SUB X, Y => X = Y-X |

Notice that HLT does not take any operands. Thus, the Op1 and Op2 fields for HLT instruction are always all zeroes.

Whenever we have an immediate value as Op1, its first three bits (the bits indicating the register part) are all zero. Thus, if we consider MOV R1, #10, the last 5 bits would be: 00010.

Also, notice that JMP has a slightly different syntax. For the moment we will ignore JMP, and return to it later.

An Assembler is a software that takes in an Assembly Language program as input, and produces as output the machine language equivalent. Let us now consider some examples of Assembly language programs and generate their equivalent machine codes.

MOV R3, #25

HLT

An Assembler will convert it to the following:

0011 0101 1000 0010 (0)

0000 0000 0010 0101 (2)

0010 1000 0000 0000 (4)

Or, equivalently, in hexadecimal:

3582

0025

2800

Let us understand how the above conversion happened. We take each instruction and convert it to machine code according to the rules specified above. First, we have MOV R3, #25. We mentioned earlier that every instruction has the general format of xxxxxx xxxMM xxxMM, where the first 6 bits represent the opcode, the next 5 bits are for the destination (or Op2), and the last 5 bits are for the source (Op1). Here, the first 6 bits is the opcode for MOV instruction. Looking at the table, we observe that the opcode for MOV is 001101. Thus, these are the first 6 bits. Next, we must represent R3. Since the register R3 has the operand, therefore, the mode is represented by 00. And R3 is represented by 011. Thus the next 5 bits become 01100. Finally, we must represent Op1. We observe that Op1 is #25, which is an immediate value. In this case, the register is not used. Thus the xxx part of xxxMM for Op1 is 000. And, because this is an immediate operand, therefore, MM is 10. Thus, these 5 bits are: 00010. Therefore, putting all this together, we get the first instruction to be: 0011010110000010. Or, to put them in groups of four, as we have done above, we have: 0011 0101 1000 0010. Now, we have mentioned earlier that in case of an immediate value, the next 16 bits are used to denote the immediate value. Thus, the next 16 bits will be the binary representation of 25 (REMEMBER that this is in hexadecimal: all immediate values are assumed to be in hexadecimal). Hence, the next 16 bits are 0000 0000 0010 0101. Finally, the last instruction is HLT. This instruction takes no operands. Therefore, only its opcode is taken, while the remaining bits are all set to zero. Hence, this instruction translates to 0010 1000 0000 0000. Thus, we have converted our first assembly language program into binary.

Notice that towards the right of the binary equivalent, we have some numbers in parenthesis. These are NOT THE OUTPUT OF THE ASSEMBLER!!!! These numbers are merely for our reference, and these indicate the address of the instruction. Every byte has an address, and since each instruction occupies two bytes, therefore, each line has address at multiples of two. Actually, the assembler keeps track of the address of each instruction as it is generating the machine code. The “Program Counter” is used for this. The Program Counter is a special variable (strictly speaking, it is a register, but you don't have to confuse yourself about this) whose value is the address of the instruction being executed. Thus, at the beginning of the program, PC (Program Counter) holds zero, indicating that the first instruction is at address zero. Subsequently, PC keeps incrementing by two to indicate that the next instructions are to be executed.

Translate the following programs into machine code, and verify that you have got the concepts right.

MOV R1, #0100

MOV \*R1, #25

HLT

The binary equivalent of this is:

0011 0100 1000 0010 (0)

0000 0001 0000 0000 (2)

0011 0100 1010 0010 (4)

0000 0000 0010 0101 (6)

0010 1000 0000 0000 (8)

Equivalently, in hexadecimal, we have:

3482

0100

34A2

0025

2800

Next, try this:

MOV R1, #0100

MOV \*R1, #10

MOV R2, #0102

MOV \*R2, #15

MOV R3, #0104

ADD \*R2, \*R1

MOV \*R3, \*R2

HLT

The binary equivalent is:

0011 0100 1000 0010 (0)

0000 0001 0000 0000 (2)

0011 0100 1010 0010 (4)

0000 0000 0001 0000 (6)

0011 0101 0000 0010 (8)

0000 0001 0000 0010 (A)

0011 0101 0010 0010 (C)

0000 0000 0001 0101 (E)

0011 0101 1000 0010 (10)

0000 0001 0000 0100 (12)

0100 0001 0010 0101 (14)

0011 0101 1010 1001 (16)

0010 1000 0000 0000 (18)

Or equivalently, in hexadecimal,:

3482

0100

34A2

0010

3502

0102

3522

0015

3582

0104

4125

35A9

2800

Let us now look at two new concepts: DC and DS. DC stands for “Declare Constant”. It is just like the declaration of a constant in a programming language at a higher level. It initializes a variable to an immediate value, and this variable’s address is the point of declaration at the program. For example, consider:

A DC 25

When we convert this to machine code, we get:

0000 0000 0010 0101 (0)

We observe that the address 0 got filled with the binary representation of 25 (which is in hexadecimal). Again, we assume that the value specified for DC is always in hexadecimal. If this statement (A DC 25) was present at some other location in the assembly code, then, when the assembler reaches this line, it associates with the current address in its machine code (which is being maintained by the Program Counter) the value 25. Later on, if we refer to A, the assembler will replace A with 25 in its machine code, as if we were specifying an immediate value. Note that it is also possible to refer to A even before the DC statement. This is called as forward reference. The assembler will still resolve this. For example:

MOV R3, A

… (assume there are further lines of assembly code)

A DC 25

Now, the assembler, when converting to machine code will be able to do:

0011 0101 1000 0010 (0)

0000 0000 0010 0101 (2)

…

0000 0000 0010 0101 (10) (assume)

It is able to forward reference and make out that A refers to 25 even though the DC statement came much later. Also, if we were to use &A, it will make out that &A should refer to 10.

To get the concept clearer, let us convert the following assembly code to machine code:

A DC 25

MOV R7, &A

MOV R1, \*R7

HLT

The corresponding machine code:

0000 0000 0010 0101 (0)

0011 0111 1000 0010 (2)

0000 0000 0000 0000 (4)

0011 0100 1001 1101 (6)

0010 1000 0000 0000 (8)

In Hex:

0025

3782

0000

349D

2800

DS stands for “Define Storage”. The statement: A DS 2 reserves 2 words of memory, filling them all up with zeroes. Referring to A will give zero, while referring to address of A will give the address of the first word of this store of memory. For example:

… (assume some lines of assembly code)

A DS 2

MOV R3, &A

…

Converting to machine code, we get:

…

0000 0000 0000 0000 (4) (assume)

0000 0000 0000 0000 (6)

0011 0101 1000 0010 (8)

0000 0000 0000 0100 (A)

We notice here that because we said A DS 2, two words (i.e, 16\*2=32 bits) got allocated, and contains all zeroes. Later, when we reference &A, we have got the binary representation of 4, since the address of the first word of this allocated block is 4.

Let us now lastly look at the concept of JMP. If we observe the table given in the beginning, we notice that JMP is used as: JMP label. This label is any arbitrary identifier, and this label can be declared just before any assembly language instruction. Essentially JMP instruction is identical to the goto statement in C. Thus, we could have something like:

MOV R1, #10

A: MOV R2, #20

…

JMP A

Notice above that the second instruction begins with the label. This is the rule. Whenever we are declaring a label, it should come at the beginning of an instruction, and the name of the label should be followed by colon (:) and a space, following which the actual instruction is there. Assume the address of the second instruction shown above is 4. Then, the JMP statement will translate to the following:

0011 1100 0000 0000

0000 0000 0000 0100

JMP has opcode as 001111, and these constitute the first 6 bits. The remaining all bits are set to zero in this instruction, i.e., we assume no Op1 or Op2. However, the immediately following instruction will be the address of the label which the JMP refers to.

Let us convert the following to machine code to understand all these we have talked about:

C DS 1

MOV R1, &A

MOV R2, &B

MOV R3, &C

MOV R4, \*R1

MOV R5, \*R2

JMP GREAT

MOV \*R3, R5

HLT

GREAT: MOV \*R3, R4

HLT

A DC 25

B DC 30

The equivalent machine code is:

0000 0000 0000 0000 (0)

0011 0100 1000 0010 (2)

0000 0000 0001 1110 (4)

0011 0101 0000 0010 (6)

0000 0000 0010 0000 (8)

0011 0101 1000 0010 (A)

0000 0000 0000 0000 (C)

0011 0110 0000 0101 (E)

0011 0110 1000 1001 (10)

0011 1100 0000 0000 (12)

0000 0000 0001 1010 (14)

0011 0101 1011 0100 (16)

0010 1000 0000 0000 (18)

0011 0101 1011 0000 (1A)

0010 1000 0000 0000 (1C)

0000 0000 0010 0101 (1E)

0000 0000 0011 0000 (20)

In hexadecimal:

0000

3482

001E

3502

0020

3582

0000

3605

3689

3C00

001A

35B4

2800

35B0

2800

0025

0030

Assemblers can be of two varieties: either a Two-Pass Assembler, or a One-Pass Assembler. A Two-Pass Assembler generates the machine code for a given assembly language after going through it twice, whereas a One-Pass Assembler scans through the Assembly Code only once in order to generate the machine code. For now, we will understand how a Two-Pass Assembler works (the following explanation is an oversimplification of reality! There are lots of other complications in a Two-Pass Assembler which I have not handled; this is only to help you get a gist of how its basic functionality is).

A Two-Pass Assembler maintains (among other things) a data structure known as a Symbol Table. A Symbol Table is basically a table that maps a symbol to an address. Thus, it has two fields: Symbol and Address. To better understand how a Two-Pass Assembler works, let us see its working in the context of an example. Consider the following assembly code:

MOV R7, &A

MOV R1, \*R7

HLT

A DC 25

Now, when a Two-Pass Assembler is given this program, it scans it twice. The first time it scans, it is only looking for symbols, so as to fill up its symbol table. Here, Symbols refer to variables and labels. Thus, when it first scans this program, it first of all finds the symbol A in the very first line, which is undefined. So, it puts the symbol A into its symbol table. However, it cannot yet put the address field for this A now, because it doesn’t know the address of the location where A is defined. This is a case of Forward Reference. The symbol (in this case, A) is used before it is defined. The assembler at this point cannot resolve the address of A. So, it moves on to the next instruction. At the same time, most importantly, it maintains the PC (Program Counter) value. Thus, in the first instruction, PC would have been 0. When it comes to the second instruction, PC value is updated to become 4 (this is because at location 2, we are supposed to have the address of A as an immediate value stored). In this manner, it keeps track of the current PC value at any instant of time. The reason this is important is because when it finally finds the definition of A in the last line, it must now fill up the Address field in the Symbol Table corresponding to entry A. And to fill up that address field, it must have kept track of PC so that when it reaches the definition of A, it can obtain the address to be the current PC value. Note that the PC value must be appropriately managed. For example, if we get the instruction B DS 3, then, the PC must be incremented by 6 (since DS stores 3 words of space). In case of DC, we can also additionally store the value (in this case 25) in the symbol table as another column. In case we get a label, the label and the current PC value are stored in the Symbol Table as the Symbol and Address Fields respectively. Thus, in our example here, at the end of the first pass, our Symbol Table has the following:

|  |  |  |
| --- | --- | --- |
| **Symbol** | **Address** | **Value(in case of DC)** |
| A | 8 | 25 |
|  |  |  |

Note that the address and values are in hexadecimal (as usual). In case there were any labels or DS statements, they would also have been stored as symbol-address pair (no value).

Now, in the second pass, the assembler happily converts the program to machine code. Now there is no problem of forward reference! When it sees the A in the first line, it can resolve the address of A by consulting its symbol table which it has constructed in the first pass! The important point to note here is that in the first pass, NO MACHINE CODE IS GENERATED!!!!! In the first pass, only the symbol table is constructed. It is only in the second pass that the machine code is generated.

**PROBLEM STATEMENT:**

Write a C program which prompts the user to enter a filename. The user enters a filename. This file contains an Assembly language program (conforming to the above specifications of Sathyam machine). Your C program prints as output the hexadecimal equivalent of the machine code of this input assembly code. For example, assume File1.txt contains the following:

MOV R1, #0100

MOV \*R1, #25

HLT

Then, upon running the C program, the following should happen (the red font indicates user input):

Enter filename: File1.txt

The machine code is:

3482

0100

34A2

0025

2800

Note that your C program should handle DC, DS, JMP statements, and thus should implement a Two-Pass Assembler.