Assembling-Disassembling

A program is written in assembly language (or even in a high level language). Then it is converted into (binary) machine code.
What is involved in this translation?
Because it is possible to refer to labels before they are declared, assemblers are usually multi-pass.
Bruce Hutton's assembler is composed of the following passes:

• Lexical analysis and parsing.

The input is analysed into tokens and constructs, and a tree is built, representing the structure of the program.

• Collection of declarations.

A treewalk is performed, to determine the names and nesting of blocks, and the identifiers declared within each block. The mapping of block names to blocks, for the list of blocks used by a block occurs in this pass. A consequence of this is that blocks must be declared before they are used.

• Mapping of identifiers to declarations.

A treewalk is performed to map all identifier applications to identifier declarations. Essentially this pass looks up the tables generated by the previous pass.

• Address generation.

A treewalk is performed to determine the offset of every statement from the base of its section, and the values of all identifiers (possibly as offsets from the base of a section). For local sections, this requires the calculation of the initial offset for the section. As a consequence, it must be defined in terms of constants and offsets of labels in previous local sections. Similarly, expressions are computed when they are needed to indicate the size of data (the expression in a space allocation statement, or an array declaration).

• Determination of the address of each section.

The code and data start at addresses that depend on whether the code is PAL, kernel, or user code. The constant and global table follows immediately after the code.

• Code generation.

A treewalk is performed to generate code. At this stage, all identifiers must be defined, in terms of absolute addresses.

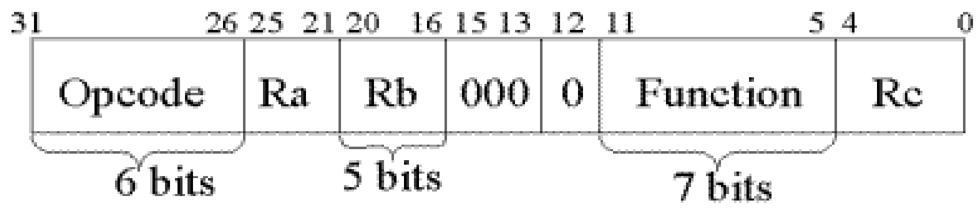
Each pass generates errors, with the offending construct indicated, and a line number. The line number is often one line after the real error.

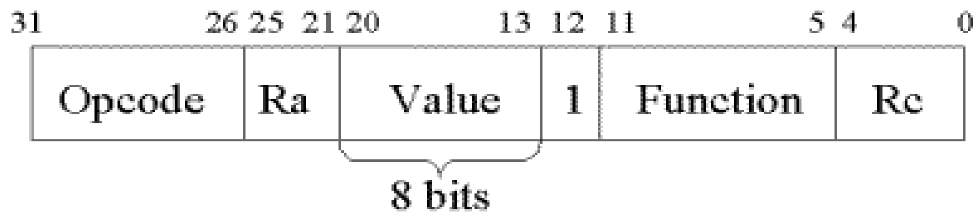
You have to know the opcode and function codes of each instruction. For example, the format, opcode and function code is indicated below for some of the common instructions.
The opcode and the function code uniquely define an instruction.

Name	Format	Opcode	Function code
addq	Operate	0x10	0x20
subq	Operate	0x10	0x29
mulq	Operate	0x13	0x20
sra	Operate	0x12	0x3c
lda	Memory	0x8	
ldq	Memory	0x29	
ldbu	Memory	0xa	
stq	Memory	0x2d	
beq	Branch	0x39	
bne	Branch	0x3d	

Integer operate instructions

The operate format is for instructions performing integer register to integer register operations, e.g. addq, subq. The operate format allows the specification of one destination operand and two source operands. One of the source operands can be a literal value. The two formats are distinguished by bit 12. If one of the source operand is a literal value, bit 12 is set to 1; otherwise, bit 12 is 0. It can be seen that each register field consists of 5 bits. This is because there are 32 integer registers. Thus, 5 bits is sufficient to hold the number denoting a register. In the diagrams below, `ra` and `rb` are the source registers' fields and `rc` is the destination register's field.
Integer operate instructions have the following format:


31
26 25
21 20
16 15 13
12
11
5 4
0
Opcode
Ra
Rb
000
0
Function
Rc
6 bits
5 bits
7 bits


31
26 25
21 20
13
12
11
5 4
0
Opcode
Ra
Value
1
Function
Rc
8 bits

Suppose we have the instruction “addq $a0, $t0, $t2;”. The identifiers a0, t0, t2 are symbolic names for registers 16, 1 and 3 (decimal), so we could write the instruction as “addq $16, $1, $3;”.
Moreover, the literal flag must be 0, so the fields for the instruction are:

Field	opcode	regA	regB	padding	Literal flag	function	regC
Hex	**0x10**	**0x10**	**0x1**	**0x0**	**0x0**	**0x20**	**0x3**
Binary	**010000**	**10000**	**00001**	**000**	**0**	**0100000**	**00011**

Grouping the bits in lots of 4 gives
0100 0010 0000 0001 0000 0100 0000 0011
Writing it in hexadecimal gives the instruction code as the number 0x42010403.
Consider the instruction “subq $t5, 1;”.
Due to operate instruction format, it has to be expended to three operands, and replacing the symbolic name of registers by their numbers, gives “subq $6, 1, $6;”.
This is an integer operate format, with a literal as second operand, so the fields for the instruction are:

Field	opcode	regA	Literal value	Literal flag	function	regC
Hex	**0x10**	**0x6**	**0x1**	**0x1**	**0x29**	**0x6**
Binary	**010000**	**00110**	**00000001**	**1**	**0101001**	**00110**

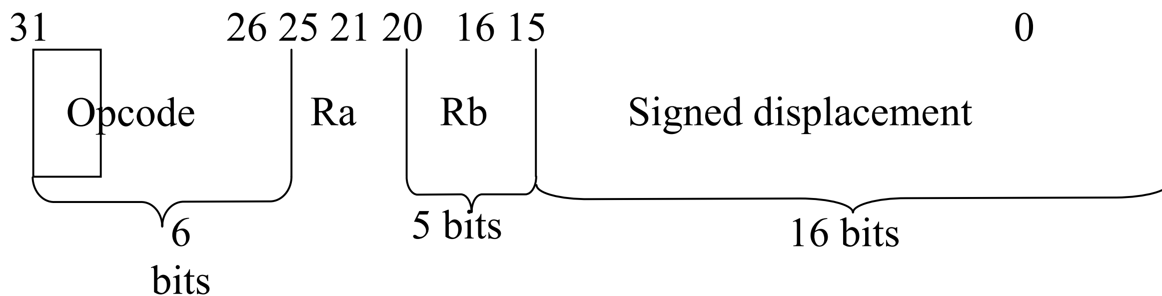
Grouping the bits in lots of 4:
0100 0000 1100 0000 0011 0101 0010 0110
Writing it in hexadecimal, gives the instruction code as the number 0x40c03526.
In fact, the computer must perform the translation in reverse order. Given the instruction in internal form, it must be able to determine the opcode and operands, so that it can execute the instruction.
For example, suppose we have an instruction 0x4cf5540e.
Writing this in binary, gives 0100 1100 1111 0101 0101 0100 0000 1110.
The 6 bit opcode is 010011, 0x13 in hexadecimal, which represents an integer operate instruction.
Moreover bit 12 is 1, so the instruction has a literal for the second operand.
Splitting it up into the relevant fields, gives

Field	opcode	regA	Literal value	Literal flag	function	regC
Binary	**010011**	**00111**	**10101010**	**1**	**0100000**	**01110**
Hex	**0x13**	**0x7**	**0xaa**	**0x1**	**0x20**	**0xe**
Decimal		**7**	**170**			**14**

Opcode 0x13, and function code 0x20 represent the mulq instruction. The Instruction is “mulq $7, 170, $14;”, or using symbolic names for registers, “mulq $t6, 170, $s5;”.

Memory access instructions

Memory access instructions have the following format:



The displacement is a signed two’s complement number.
Suppose we have the instruction “lda $sp, +10($sp);”.

Field	opcode	regA	regB	Displacement
Hex	0x8	0x1e	0x1e	0xa
Binary	001000	11110	11110	0000000000001010

(Decimal 10 is hexadecimal 0xa and binary 1010.)
Grouping the bits in lots of 4 gives
0010 0011 1101 1110 0000 0000 0000 1010
Writing it in hexadecimal, gives the instruction code as the number 0x23de000a.
Suppose we have the instruction “lda $sp, -10($sp);”.

Field	opcode	regA	regB	Displacement
Hex	0x8	0x1e	0x1e	0xfff6
Binary	001000	11110	11110	1111111111110110

(The decimal -10 is represented as a two’s complement number by writing decimal 10 in binary as 0000000000001010, taking the one’s complement 1111111111110101, add 1 to get 1111111111110110. You MUST take into account the number of bits used to store the value.)
Grouping the bits in lots of 4 gives
0010 0011 1101 1110 1111 1111 1111 0110.
Writing it in hexadecimal, gives the instruction code as the number 0x23defff6.

The reverse way

Suppose we have the instruction 0x23deffe0. In binary it transforms to:
0010 0011 1101 1110 1111 1111 1110 0000
The opcode is 0x8, so it is a lda instruction. Splitting it up into fields gives:

Field	opcode	regA	regB	Displacement
Hex	0x8	0x1e	0x1e	0xffe0
Binary	001000	11110	11110	1111111111100000

In other words, “lda $sp, -0x20($sp);”. (We can determine the negative number the displacement corresponds to by taking the two’s complement, to get a positive number. Alternatively, 0xffe0 can be substracted from 0x10000.)

Branch instructions

Branch instructions are a little more complex, because the displacement stored in the instruction is relative to the PC, at the time at which the instruction is executed (after the PC has been incremented to point to just after the instruction), and the displacement is counted in longwords (in other words, the low two bits of the byte displacement are discarded), because all instructions must be longword aligned.

Suppose we have an instruction “bne $s1, label1;”, at address 0x80023c, and label1 correponds to address 0x80027c.
The PC will be 0x800240 at the time the instruction is executed. So the address to branch to is 0x80027c - 0x800240 = +0x3c bytes away. Dividing this by 4 (Displacement size is a multiple of longword size) gives us a displacement of +0xf. The opcode for bne is 0x3d, and register s1 is register 10 (decimal):

Field	opcode	regA	Displacement/4
Hex	0x3d	0xa	0xf
Binary	111101	01010	000000000000000001111

Grouping the bits in lots of 4 gives
1111 0101 0100 0000 0000 0000 0000 1111

Writing it in hexadecimal, gives the instruction code as the number 0xf540000f.
Suppose we have an instruction "beq $v0, label2;" at address 0x80025c, and label2 corresponds to address 0x80022c.
The PC will be 0x800260 at the time the instruction is executed. So the address to branch to is 0x80022c - 0x800260 = -0x34 bytes away (0x7fffcc, when written as a 23 bit 2's complement number).
Dividing this by 4 gives us a displacement of -0xd (0x1ffff3, when written as a 21 bit 2's complement number). The opcode for beq is 0x39, and register v0 is register 0:

Field	opcode	regA	Displacement/4
Hex	0x39	0x0	-0xd(0x1ffff3)
Binary	111001	00000	111111111111111110011

(There are various ways of performing the arithmetic. One way is to do everything in binary. Another way is to do it in hexadecimal. Negative numbers come out as numbers with.f's on the left. When the data is packed in the displacement field, the extra bits are discarded.)
Grouping the bits in lots of 4 gives
1110 0100 0001 1111 1111 1111 1111 0011
Writing it in hexadecimal, gives the instruction code as the number 0xe41ffff3.

The reverse way

Suppose we have the instruction 0xe6000003, at address 0x800200. In binary this correspond to 1110 0110 0000 0000 0000 0000 0000 0011.
The opcode is 0x39, so it is a beq instruction.
Splitting it up into fields gives:

Field	opcode	regA	Displacement/4
Hex	0x39	0x10	0x3
Binary	111001	10000	000000000000000000011

The destination adress is 4 * 0x3 + 0x800204 = 0x800210, giving the instruction "beq $a0, 0x800210;". If the address 0x800210 has a label, the symbolic label can replace it.

Assembling

Convert the instructions below to hexadecimal code.

`mulq $16, $17, $18 (reg 16 is 0x10)`

31 26	25 21	20 16	15 13	12	11 5	4 0
010011	10000	10001	000	0	0100000	10010
Opcode	Ra	Rb	**padding**	LF	**Function**	Rc
0x13	0x10	0x11	**0x0**	**0x0**	**0x20**	0x12

0x4e110412

`addq $19, $20`

31 26	25 21	20 16	15 13	12	11 5	4 0
010000	10011	10100	000	0	0100000	10011
Opcode	Ra	Rb	**padding**	LF	**Function**	Rc
0x10	0x13	0x14	**0x0**	**0x0**	**0x20**	0x13

0x42740413

`addq $21, 0x34, $1`

31 26	25 21	20 13	12	11 5	4 0
010000	10101	00110100	1	0100000	00001
Opcode	Ra	Literal	**LF**	**Function**	**Rc**
0x10	0x15	0x34	**0x1**	**0x20**	**0x1**

0x42a69401

`subq $2, 12`

31 26	25 21	20 13	12	11 5	4 0
010000	00010	00001100	1	0101001	00010
Opcode	Ra	Literal	**LF**	**Function**	**Rc**
0x10	0x2	0xc	**0x1**	**0x29**	**0x2**

0x40419522