You must submit the following items:

1. The source files of the program you wrote (files with .c or .h extension).
2. The compile .makefile file must be with the gcc compiler and flags: -Wall -ansi -pedantic. All messages sent by the compiler should be sifted so that the program will be folded without any comments or warnings.

The imaginary computer and the assembly language We will now define the assembly language and the imaginary computer model, for this project.

"hardware:"

The computer in the project consists of a processor, registers and RAM. Some of the memory is used As a stack.

The processor has 32 general registers, in the names:. $ 0, $ 1, $ 2 .. $ 31 The size of each register is 32 bits.

The least significant bit will be specified as bit number 0, and the most significant bit as number 31.

The memory size is 2^{25} cells, in addresses ,0-(2^{25}-1), and each cell is 8-bit in size (Byte). An address in memory can be represented in 25-bits (as a number without a sign).

This computer only works with positive and negative integers. Real numbers are not supported.

Arithmetic is done using the complementary method for 2.

There is also support for characters, which are represented by ascii code.

Machine instructions:

This computer has a variety of instructions, each instruction consists of an action and operands. The number of operands depends on the type of instruction.

Each instruction is encoded in a code known as 32-bit. The least significant bit of the instruction coding will be specified as bit no. 0, and the most significant bit as no. 31.

An instruction occupies four consecutive bytes in the computer memory, and is stored in a small-endian method, ie bits 0-7 of the instruction are in the house with the lowest address, bits 8-15 are the next home, followed by bits, 16-23, while bits 24-31 are in the highest address. most. When referring to the address Of memory instruction, this is always the lowest home address.

Each instruction has an action code, also called an opcode, that identifies the action being performed by the instruction. Some instructions have a secondary identification code, called a funct.

The instructions are divided into 3 types: type instructions R, type instructions I, and type instructions J.

Below is a table listing all the instructions, by type, and with their identification codes:

opcode (In decimal base)	funct (In decimal base)	instructions type	instructions name
0	1	R	add
0	2	R	sub
0	3	R	and
0	4	R	or
0	5	R	nor
1	1	R	move
1	2	R	mvhi
1	3	R	mvlo
10		I	addi
11		I	subi
12		I	andi
13		I	ori
14		I	nori
15		I	bne
16		I	beq
17		I	blt
18		I	bgt
19		I	lb
20		I	sb
21		I	lw
22		I	sw
23		I	lh
24		I	sh
30		J	jmp
31		J	la
32		J	call
63		J	stop

Note: In assembly syntax in assembly language, the action name is always written in lower case.

Machine instruction structure:

We will now discuss in more detail the syntactic structure of each instruction in the assembly language, and the manner in which the instruction is encoded in the machine code.

Type R instructions:

Type R instruction is encoded as follows:

opcode	rs	rt	rd	funct	not in use
31 26	25 21	20 16	15 11	10 6	5 0

In type R instructions all operands are collectors. Some of the instructions have two operands, and some have three operands.

The R instruction set includes the following instructions:

- **Arithmetic and logical instructions:** add, sub, and, or, nor.
- **Copy instructions:** move, mvhi, mvlo.

Arithmetic and logical instructions of type R:

The add, sub, and, or, nor instructions have three operands, and all three are collectors. All arithmetic / logical instructions of type R have the same code of operation and are 0 (ie the opcode field contains 0). In order to distinguish between the different instructions, there is another field called funct which is unique to each instruction (see the instruction table above).

We will demonstrate the use of Arithmetic and logical instructions, and how to encode them:

add $3, $19, $8

In encoding this instruction, the opcode field will contain the action code 0 (common to all type Arithmetic and logical R instructions) and in the funct field the code unique to add. The rs field will contain the number of the collector used as the first connected (in this example, 3) The rt field will contain the number of the collector used as the second connected (in this example 19), and the rd field will contain the number of the result register (in this example 8). The 0-5 bits in the instruction encoding are not used, and will contain zeros.

Similar to add, sub, and, or, nor instructions are also used.

Note: There is no denying that the same collector will be used by more than one operand (for example: or $1, $2, $1).

Copy instructions of type R:

the move, mvhi, mvlo instructions have two operands, and all two are collectors. All type R copy instructions have the same action code and it is 1 (i.e. the opcode field contains 1) In order to distinguish between the different instructions, there is a unique funct field for each of them (see the instruction table above).

We will demonstrate the use of copy instructions, and how to encode them:

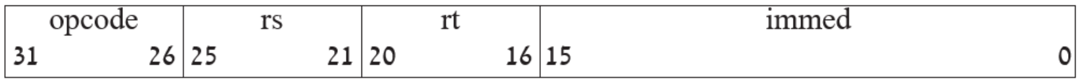
move $23, $2

In encoding this instruction, the opcode field will contain the action code 1(common to all R copy instructions). And the funct field the unique code for move. The rs field will contain the number of the source register of the copy (in this example 2) and the rd field will contain the number of the target register (in this example 23) the rt field, and the 0-5 bits are not used and will contain zeros.

Similar to move, mvhi, mvlo instructions are also used.

<u>Type I instructions:</u>

Type I instruction is coded as follows:



opcode	rs	rt	immed
31 26	25 21	20 16	15 0

Each type I instruction has a unique opcode (see instruction table above).

The I instruction set includes the following instructions:

- **Arithmetic and logical instructions:** addi, subi, andi, ori, nori.
- **Conditional branching instructions:** beq, bne, blt, bgt.
- **Instructions for loading and keeping in memory:** lb, sb, lw, sw, lh, sh.

Type I arithmetic and logical instructions:

We will demonstrate the use of instructions, and how to encode them:

addi $9, -45, $8

In encoding this instruction, the opcode field will contain the action code of addi, field rs will contain the number of the collector that is first connected (in this example 9) the rt field will contain the number of the result register (in this example 8), and the immed field will contain the constant that The second connected (in this example the number -45).

Similar to <u>addi</u>, <u>subi, andi, ori, nori</u> instructions are also used.

There is no denying that the same collector will be used in the two operands rs, rt.

Conditional branching instructions:

The jump target is encoded in the immed field, using the distance to the label address from the current address (branch instruction address). The distance can be positive or negative, but can not exceed the limits of a 16-bit number in the complementary to 2 method.

We will demonstrate the use of instructions, and how to encode them:

blt $5, $24, loop

In encoding this instruction, the opcode field will contain the blt action code, the rs field will contain the number of the collector used as the left operand of the comparison operation (in this example 5), and the rt field will contain the number of the collector used as the right operand (in this example 24).

The immed field will contain the distance from the current instruction address (blt) to the instruction at the loop address. This distance must be calculated by the assembler himself. For example, suppose the current blt instruction is at 300, and the loop address is 200. The immed field will encode to -100.

Similar to <u>blt</u>, <u>bne, blt, bgt</u> instructions are also used.

Instructions for loading and keeping in memory:

We will demonstrate the use of instructions, and how to encode them:

```
lh $9, 34, $2
```

In encoding this instruction, the opcode field will contain the lh action code, the rs field will contain the collector number used to calculate the address (in this example 9), the collector rt will contain the collector number to which the value will be loaded (in this example 2), and the immed field will contain the offset (In this example 34). Note: Note that there is a difference in the order of the operands, between the instruction syntax in the assembly language and the binary coding.

Similar to lh, lb, bw instructions are also used.

another example:

```
sw $7, -28, $18
```

In encoding this instruction, the opcode field will contain the sw instruction code, the rs field will contain the collector number used to calculate the address (in this example $7), the rt field will contain the collector number to be stored in memory (in this example $18), and the immed field will contain the offset (In this example -28).

Note: Note that there is a difference in the order of the operands, between the instructions syntax in the assembly language and the binary coding.

Similar to sw, sb, sh instructions are also used.

Type J instructions:

Type J instruction is coded as follows:

opcode	reg	address
31 26	25	24 0

Each type J instruction has a unique opcode (see instruction table above).

There are four such instructions, and they are: stop, call, la, jmp.

Jmp instruction:

In encoding this instruction into machine language, we will place in the opcode field the jmp action code (see the action table above) in the reg field Commissioner 0, to indicate that the jump target is given by means of a label. If the main label is defined in the current source file (not external) then we will set its address in the address field (0-bit encoding is the least significant bit of the address). If the label is external, then we will place in the address zero field, because the real address is not known to the assembler (the field coding will be completed at the linking stage of the plan building process, **which you are not required to implement**).

In the second syntax form of the jmp instruction, a jump is made to the address located in the register, whose number is encoded in the address field (i.e. address will contain a number between 0-31 in 25-bit width). Note: The address space in the imaginary computer is the domain, ,[0..2^{25}-1] and the collector should contain a value that does not exceed this domain.

We will demonstrate the use of this form of syntax. Suppose the $7 collector contains an address we would like to jump to, then write:

jmp $7

In encoding this machine language instruction, we will place the jmp action code in the opcode field. In the reg field Commissioner 1, to indicate that the jump destination is given by means of a collector. In the address field we will place the collector number (in this example 7).

In fact, 0-4 bits will contain the collector number, and 5-24 bits will contain zeros.

la instruction:

The la (load address) instruction loads the $0 address of a particular label. We emphasize that the label can be defined in the current source file, or in another source file (external label).

The syntax of the instruction la in the assembly language is:

la label

label indicates the label whose address you want to load to the $0 register.

Note: In instruction la can not use a collector other than $0.

We will demonstrate the use of this instruction. Suppose that in a defined memory a variable named num1 (a variable found in the address labeled num1), and we want to load the variable address into the register, $ 0 then we write:

la num1

In encoding this instruction into machine language, we will place in the opcode field the operation code of la. The reg field is not used in this instruction, so it will always be set to 0. If the label num1 is defined in the source file (not external) then we will set its address in the address field. If the label is external, then we will set zeros in the address field (with the same meaning as in the jmp instruction).

call instruction:

An instruction that causes a jump to another place in the program to continue running, and saves the next instruction address (the instruction that follows the call). The instruction is mainly used for reading a routine. The syntax of this instruction in the assembly language is:

call label

A jump is made to an instruction located at the address indicated by the label label, while the instruction address following the call is stored in the $ 0 register. We emphasize that the label can be defined in the current source file, or in another source file (external label).

Note: In instruction call can not use a collector other than $0.

We will demonstrate the use of this instruction. Suppose there is a instruction program with a myfunc label, which is the first instruction of a routine, and we want to call a routine. Then we will write:

call myfunc

In encoding this instruction into machine language, we will place the call code of action in the opcode field. The reg field is not used in this instruction, so it will always be set to 0. If the myfunc label is set in the source file (not external) then we will set its address in the address field. If the label is external, then we will set zeros in the address field (with the same meaning as in the jmp instruction).

As mentioned, the instruction address after the call (return address from the routine) is stored in the $ 0 register to return from the routine, the jmp instruction must be used in its second form (see above)

stop instruction:

This provision has no operands and is intended to stop the program. In encoding this instruction into machine language, we will place the operation code of stop in the opcode field, and zeros will be set in all other fields.

Assembly structure in assembly language:

An assembly language program is made up of statements. An original file in an assembly language consists of rows that contain sentences of the language, with each sentence appearing in a separate line. That is, the sentence is separated by a sentence in the source file using the '\n' character (new line).

The length of a line in the source file is 80 characters at most (not including the character '\n').

There are four types of sentences (lines in the source file) in Assembly language, and they are:

Type of sentence	General Explanation
An empty sentence	This is a line that contains only white characters (whitespace), meaning only spaces And tabs. There may be no character in the line (except the \ n 'character), meaning the line is empty.
Comment sentence	This is a line where the first character that is not a white character is ';' Comma-comma). The assembler must completely ignore this line.
Guidance sentence	This is a sentence that instructs the assembler what to do when operating on the source plan. There are several types of prompt statements. A prompt statement may result in variable memory allocation and initialization of the program, but it does not produce a coding of machine instructions to be executed when the program is run.
Instructional sentence	This is a statement that produces a coding of machine instructions to execute when running the program. The sentence consists of the name of the instruction (action) that the processor must perform, and the operands of the instruction.

We will now go into more detail about the different types of sentences.

Guidance sentence:

A prompt line sentence has the following structure:

A definition of a label can appear at the beginning of a sentence. The label has a valid syntax that will be described later.

The label is optional.

The prompt name then appears. After the prompt name, parameters will appear (the number of parameters according to the prompt).

Name of prompt begins with '.' (Period) followed by lowercase letters only.

There are several types (names) of prompt line sentences, and they are:

1. '.dh', '.dw', '.db'

   The parameters of the directive sentences '.dh', '.dw', '.db' are valid integers (one or more) separated by the character ',' (comma) for example:

   .db 7, -57, 17, +9.dw 120056
   .dh 0, -60431, 1700, 3, -1

   Note that commas do not have to be adjacent to numbers. Spaces and tabs can appear between a number and a comma and a number in any quantity (or not at all), but the comma must appear between the numbers. Also, no more than one comma must appear between two numbers, nor a comma after the last number or before the first number.

   These prompt line statements guide the assembler to allocate space in the data image where the values of the parameters will be stored, and to advance the data counter, depending on the number of values and their size. Each data defined by the '.db' prompt sentence takes up a single byte. A statistic defined by the prompt '.dw' takes four bytes (a word). A figure defined by '.dh' occupies two bytes (half-word).

Note: Note that each number must be of a suitable size, not to exceed the representation limits of the defined data type.

If a label is defined in the landing page, then this label receives the data counter value (before the promotion) and is inserted into the symbol table. This makes it possible to refer to a particular place in the data image through the label name (actually, this is a way of defining a variable name).

For example, if we write:

```
XYZ:  .dh 0, -60431, 1700, 3, -1
```

Then 10 consecutive bytes (bytes) will be allocated in the data image, and any two bytes (half-words) are initialized with one of the numbers that appear in the prompt (in order) The XYZ label is identified with the first home address.

XYZ can be treated as a variable that is a set of 5 half-word-sized organs.

2. '.asciz'

For '. asciz' prompt one parameter, which is a valid string. The string characters are encoded according to the corresponding ascii values, and inserted into the data image in their order, each character in a separate byte. At the end of the string will be added the character '\0' (the numeric value 0), which marks the end of the string. The data counter of the assembler will be promoted accordingly along the length of the string (plus one space for the ending character). If a label is defined in the prompt line, then this label receives the data counter value (before the promotion) and is inserted into the symbol table.

For example, if we write:

```
STR:    .asciz "hello world"
```

Then 12 consecutive bytes will be assigned in the data image, starting with the above string characters (including the ending zero). The STR label is associated with the address of the first character.

STR can be treated as a variable that is a string of characters (maximum) length 11.

3. '.entry'

The '.entry' prompt has one parameter, and is the name of a label defined in the current source file (that is, a label that receives its value in that file). Purpose of entry entry. Is to characterize this label in a way that will allow assembly code contained in other source files to use it (as an instruction operand).

For example, the lines:

```
            .entry HELLO
  HELLO:    add $1,$2,$5
```

Notify the assembler that there may be a reference in another source file to the HELLO label defined in the current file.

Note: A label defined at the beginning of the entry line. Is meaningless and the assembler **ignores** this label (the assembler may issue a warning message).

4. '.extern'

The '.extern' prompt has one parameter, and is the name of a label that is not defined in the current source file. The purpose of the instruction is to inform the assembler that the label is defined in another source file, and that the assembly code in the current file uses the label.

Please note that this directive complies with the '.entry' directive. Which appears in the file where the label is defined. In the linking phase, a match will be made between the value of the label, as specified in the machine code of the file that defined the label, and the coding of the instructions that use the label in other files (the linking phase is not relevant to this project).

For example, the '.extern' prompt that matches the '.entry' prompt from the previous example would be:

Note: The same label cannot be defined in the same file as both entry and extern (in the examples above, the HELLO label). The whole idea is to allow instruction in one file to use data from another file, or to jump to instruction in another file.

Note: A label defined at the beginning of the .extern line is meaningless. The assembler ignores this label (the assembler may issue a warning message).

Instructional sentence

A instruction sentence consists of the following parts:

1. **Optional** label.
2. The name of the action.
3. Operands, depending on the type of operation (as explained for each operation above)

If a label is defined in the instruction bar, then it will be inserted into the symbol table. The label value will be the first byte address of the instruction within the code image that the assembler builds.

The name of the action will always be written in lower case. After the operation name, the operands will appear, depending on the type of operation. The name of the operation and the first operand must be separated by spaces and / or tabs (one or more).

When there are several operands, the operands are separated by a ',' (comma) character. There does not have to be an attachment of the operands to the comma. Any amount of spaces and / or tabs on either side of the comma is legal.

Characterization of fields in assembly language sentences

label:

A label is a symbol that is defined at the beginning of a instruction sentence or at the beginning of a prompt sentence that defines variables. A valid label begins with an alphabetic letter (uppercase or lowercase), followed by a series of alphabetic letters (uppercase or lowercase) and / or numbers. The maximum length of a label is 31 characters.

Definition of a label ends in the character ':' (colon). This character is not part of the label, but only a sign indicating the end of the definition. The ':' character must be attached to the label without spaces.

The same label must not be defined more than once (of course in different lines). Lowercase and uppercase letters are considered different from each other.

For example, the labels defined below are legal labels.

```
hEllo:
x:
He78902:
```

Note: Reserved words of the assembly language (i.e. name of action or prompt) cannot be used as a label name either. For example: the add, asciz symbols cannot be used as labels, but the Add, ASCiz symbols are valid labels.

The label gets its value depending on the context in which it is defined. A label defined in a prompt line will receive the current data counter value of the data counter, while a label defined in a instruction line will receive the current instruction counter value.

Attention: In an instruction sentence it is permissible to use an operand which is a label defined in a later row in the source file. It is also permissible in the instruction statement to use the operand which is an icon that is not defined as a label in the current file, as long as the icon is characterized as external (using the .extern prompt in the current file). This issue will be clarified later, in the discussion on the realization of the assembler.

A number:

A valid number begins with an optional sign: '_' or '+' followed by a series of digits with a decimal base. For example: 76, _5, +123 are valid numbers. There is no support for our assembly language in representation on a basis other than decimal, and there is no support for incomplete numbers.

String:

A valid string is a series of visible (printable) ascii characters, surrounded by double quotes (quotes are not considered part of the string) Example of a valid string: "hello world".

Assembler with two transitions

The assembler receives as input a file containing a program in the assembly language, and its job is to convert the source code into machine code, and build an output file containing the machine code.

Note: The assembler does not execute the program, but only translates it into binary. Thus, the source code must be syntactically correct, but it does not matter to the assembler (and to our project) what the plan is supposed to do and whether it will work correctly at all.
For example: The assembler receives the following program in assembly language:

```
MAIN:   add     $3,$5,$9
LOOP:   ori     $9,-5,$2
        la      val1
        jmp     Next
Next:   move    $20,$4
        bgt     $4,$2,END
        la      K
        sw      $0,4.$10
        bne     $31,$9,LOOP
        call    val1
        jmp     $4
END:    stop
STR:    .asciz  "aBcd"
LIST:   .db     6,-9
        .dh     27056
.entry  K
K:      .dw     31,-12
.extern val1
```

The translation of the source program in the machine code example is shown below. For each line in the source code, the binary encoding of the instruction or data is displayed, along with the address in memory where the code will be loaded for execution.

The assembler builds the machine code of the program to suit memory loading **starting from address 100 (decimal).**

Note the jumps in 4 in the address column each time you encode an instruction, since the encoding consists of 32 bits, which occupy 4 consecutive memory cells.

Address (decimal)	**Source Code**			**Machine Code (binary)**
0100	MAIN:	add	$3,$5,$9	000000 00011 00101 01001 00001 000000
0104	LOOP:	ori	$9,-5,$2	001101 01001 00010 1111111111111011
0108		la	val1	011111 0 0000000000000000000000000
0112		jmp	Next	011110 0 0000000000000000001110100
0116	Next:	move	$20,$4	000001 10100 00000 00100 00001 000000
0120		bgt	$4,$2,END	010010 00100 00010 0000000000011000
0124		la	K	011111 0 0000000000000000010011101
0128		sw	$0,4,$10	010110 00000 01010 0000000000000100
0132		bne	$31,$9,LOOP	001111 11111 01001 1111111111100100
0136		call	val1	100000 0 0000000000000000000000000
0140		jmp	$4	011110 1 0000000000000000000000100
0144	END:	stop		111111 0 0000000000000000000000000
0148	STR:	.asciz	"aBcd"	01100001
0149				01000010
0150				01100011
0151				01100100
0152				00000000
0153	LIST:	.db	6,-9	00000110
0154				11110111
0155		.dh	27056	0110100110110000
0157	K:	.dw	31,-12	00000000000000000000000000011111
0161				11111111111111111111111111110100

We will now describe how the assembler will perform the machine code translation in a systematic way.

The assembler maintains a table listing all the action names of the corresponding instructions and binary codes (opcode, funct), similar to the table we presented earlier. Therefore, it is easy to translate the name of the operation into binary: when an instruction line is read from the source file, look for the appropriate binary code in the table.

Even operands that are registers or instantaneous constants can be encoded directly into binary. The same goes for data that is defined and initialized using prompt lines.

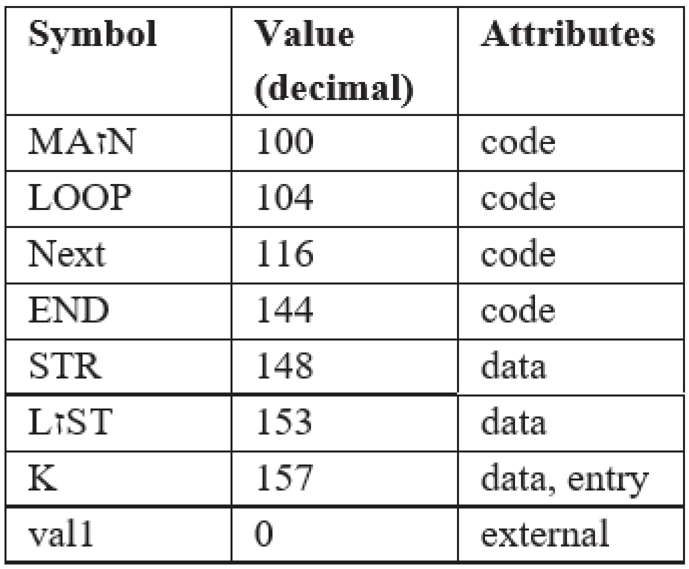
The main difficulty is to encode binary operands that use symbols (labels). The assembler must know the numerical value (address) that each symbol represents. However, unlike the codes of the action names, which are fixed and known values, the inscriptions in the memory of the symbols used in the program are unknown, until the entire source program was scanned and all the definitions of the symbols were discovered.

For example, in the source code above, the assembler cannot know that the symbol END should be associated with address 144 (decimal), and that the symbol K should be associated with address 157, but only after all the program lines have been read.

Therefore the treatment of the assembler is separated into two stages. In the first stage, all the symbols in the source plan are identified and the numerical values associated with them are determined. In the second stage, the symbols in the operands of the program instructions are replaced with their numerical values, so that the operands can be coded to binary.

Performing these two steps involves two scans (called "transitions") of the lines in the source program. In the first transition, the assembler builds a table of symbols and inserts into it each symbol defined in the source plan, along with the numeric value (address in memory) associated with it.
In the second transition, the assembler uses the table of symbols to convert the lines in the source code to machine code.
Women notice that at the beginning of the second transition the values of all the symbols should already be known. If there is a symbol used in the operand, but the symbol is not in the symbol table, the encoding cannot be completed.

For the example we presented, the table of symbols is given below. Each symbol in the table also has attributes that will be explained later. The order of the rows in the table does not matter (here the table is in the order in which the symbols in the source plan were defined).



Symbol	Value (decimal)	Attributes
MAIN	100	code
LOOP	104	code
Next	116	code
END	144	code
STR	148	data
LIST	153	data
K	157	data, entry
val1	0	external

Attention: As mentioned, the role of the assembler, on its two aisles, is to translate source code into machine language code. At the end of the assembler operation, the program is not yet ready to be loaded into memory for execution. The machine code must go to the link / load stages, and only then to the execution stage (these stages are not part of the project).

The first transition

In the first transition rules are required to determine which address will be associated with each symbol. The basic principle is to count the places in memory, which are occupied by the instructions. If each instruction is loaded in memory to the place that follows the previous instruction, such a count will indicate the address of the next instruction. The count is done by the assembler and is held in the instruction counter (IC). The initial value of the IC is 100 (decimal), so the machine code of the first instruction is constructed so that it is loaded into memory starting at 100. The IC is incremented by 4 after each instruction line (as the number of cells occupied by the instruction coding). Thus the IC always indicates the next free space in the memory.

Similarly, the assembler counts the places in memory that the data occupy (defined by instructions). The count is held in the data counter (DC) (see more details below, in the discussion of the assembler's work process).

As mentioned, in order to encode the instructions in machine language, the assembler holds a table, which has a suitable encoding for each operation (opcode and in some operations also a funct). While translating to machine code, the assembler replaces each action name in its encoding.

When the assembler encounters a label that appears at the beginning of the line, he knows that it is preceded by a definition of a label, and then he assigns to it an address _ the current value of the IC or DC. This is how each label gets its response in the line where it is defined. The label is inserted into the table of symbols, which contains in addition to the name of the label also the address and other characteristics. When there is a reference to the label in the operand of any instruction, the assembler can retrieve the appropriate address from the symbol table.

An instruction can also refer to a symbol that has not yet been defined so far in the program, but will be defined only later in the program.
For example, the following is a jump instruction for which is associated with the label A, but the label is defined only in the continuation of the code:

```
        jmp A
        .
        .
        .
    A: …..
```

When the assembler arrives at the jump instruction (jmp A), he has not yet encountered the label definition A and of course does not know the address associated with the label. Therefore the assembler can not build the binary encoding of the operand A. See below how this problem is solved.

In any case, it is always possible in the first transition to partially build the machine code of the instruction. For example, you can encode the opcode field, the funct field (in the R-format instruction), as well as all the fields that contain collector numbers. In arithmetic / logical instructions in format I, the immed field can also be encoded in the first transition.

The second transition

We have seen that in the first transition, the assembler cannot construct the machine code of operands that use symbols that have not yet been defined. Only after the assembler has gone through the entire program so that all the symbols have already entered the symbol table can the assembler complete the machine code.

To do this, the assembler performs an additional transition (second transition) on each source file, and builds the machine code of each operand which is a value of an icon, using the values of the icons from the icon table.

Specifically, the immed field must be updated in a conditional branching instruction type I, and the address field in a type J instruction.

At the end of the second transition, the program will be translated in its entirety into machine code.

Separation of instructions and data

The program distinguishes between two types of content: instructions and data. The machine code should be arranged so that there is a separation between the data and the instructions. Separating the instructions and data into different sections of memory is a better method than linking the data settings to the instructions that use them.

One of the dangers of not separating the instructions from the data is that sometimes, due to a logical error in the program, the processor may try to "execute" the data as if it were legal instructions. For example, an error that can cause such a phenomenon is incorrect branching. The program will of course not work properly, but most of the time the damage is more severe because a hardware anomaly is created as soon as the processor performs an illegal operation.

Our assembler must separate, in the machine code it produces, the data segment and the instruction segment. That is, **the output file (in the machine code) will have a separation of instructions and data into two separate sections**, although **the input file does not have to have such a separation**. The assembler's algorithm is described below, detailing how to perform the separation.

Error detection in the source program

The assembler must perform a careful parsing of the source program in the assembly language, and be able to detect and report a variety of errors, such as: undefined action name, incorrect number of operands, type of operand that is not suitable for operation, non-existent collector number, use of undefined symbol , A symbol defined more than once, an unusually sized numeric value, and other errors.

The assembler will report the errors via messages to the standard stdout output. Every error is caused (usually) by a certain line in the source code. Each error message must also indicate the line number in which the error was detected (the number of lines in the file starts with 1).

Example: If line # x in the input file contains two operands in the instruction that should have only a single operand, the assembler will give an error message such as "Line x: extraneous operand"

Example: If line # y in the input file contains an unknown prompt name, the assembler will give an error message such as: "Line y: unrecognized directive <directive-name>"

Attention: The assembler does not stop its operation after the first error is found, but continues to go through the input to detect more errors, if any. Of course there is no point in building the output files if errors are detected (the machine code cannot be completed anyway).

We will now describe in detail the work process of the assembler. Later, a skeletal algorithm for the first and second transitions will be presented.

The assembler maintains two arrays, hereinafter referred to as the instruction image (code) and the data image. In the instruction set, the assembler builds the encoding of the machine instructions read during the transition to the source file. In the data set, the assembler enters the encoded data read from the source file (i.e. the data
In .db, .dw, .dh) prompt lines.

The assembler uses two counters, called IC (Instruction Counter) and DC (Data Counter). These counters indicate the next available space in the instruction set and the data set, respectively. Each time the assembler starts to go

through a source file, the IC counter gets a starting value of 100, and the DC counter gets a starting value of 0.
The initial value IC = 100 is set so that the program's machine code matches the memory load (for running) starting from address 100.

In addition, the assembler maintains a table, in which all the labels encountered by the assembler during the transition to the source file are collected. This table is called the table of symbols.
Each symbol is saved in the table by the name of the symbol, its numeric value, and other properties (one or more), such as the location in the memory image (data or code), and the type of visibility of the symbol (external or entry).

In the first transition the assembler builds the table of symbols, and the skeleton of the memory image (the set of instructions and the set of data separately).

The assembler reads the source file line by line, and acts according to the type of line (instruction, prompt, or blank line / note).

1. Blank line or note line:

   The assembler ignores the line and moves on to the next line.
2. Instruction line:

   The assembler analyzes the line and deciphers what the instruction is.

   If the assembler finds at the beginning of the instruction line also a definition of a label, then the defined label is inserted into the table of symbols. The value of the symbol in the table is IC, and the property is code.

   The assembler inserts into the memory image a line for the instruction being read, containing the IC address, and the fields of the machine code according to the type of instruction (R, I, J). Any field that can be encoded already, will get the appropriate encoding (see the first transition description above).

   Finally, the assembler adds 4 to the value of the IC, since each instruction will be encoded into 4 memory cells (32 bits).

3. Prompt line:

   When the assembler reads in the source file a prompt line, it acts according to the type of prompt, as follows:

   I. prompt .db, .dw, .dh.

      The assembler reads the list of numbers, inserts each number into the data set (in binary encoding) and in the number of bytes depending on the type of prompt statement. And promotes the DC data pointer in-in the total amount of bytes that all the numbers consumed.

      If the '.data' line also defines a label, then the label is inserted into the icon table. The value of the label is the value of the DC data counter before entering the numbers in the array. The property of the label is data.

   II. prompt .asciz

      The handling of '. asciz' is similar to '.db', except that the ascii codes of the characters are the ones inserted into the data set (each character in a separate byte). Finally, the character '\ 0' (indicating the end of a string) is inserted into the data set. The DC counter is preceded by the +1 string length (the character ending the string also takes up space).

      The treatment with the label defined in the asciz prompt is the same as the treatment done with the prompt .db, dw, .dh

III. prompt .enrty

This is an instruction for the assembler to characterize the given label as an operand as an entry in the symbol table. When producing the output files (see below), the label will be recorded in the entries file.

Note: It is not considered an error if the source file contains more than one .entry directive with the same label as the operand. The extra shows do not add anything, but neither do they interfere.

IV. prompt .extern

This is a declaration of an icon (label) defined in another source file, and the current file uses it. The assembler puts the symbol that appears as an operand in the symbol table, with the value 0 (the real value is unknown, and will be determined only at the link stage) and with the external property it is not known in which file the symbol definition is, and it is not relevant for the assembler.

Note: It is not considered an error if the source file contains more than one external prompt with the same label as the operand. The extra shows do not add anything, but neither do they interfere.

Note: In a instruction operand, it is allowed to use an icon that will be defined later in the file (whether directly by setting a label, or if indirectly by landing extern). In the entry of the entry prompt it is allowed to use an icon which will be defined later in the file by setting a label.

At the end of the first transition, the assembler updates in the symbol table each symbol that is characterized as data, by adding (100) + IC (decimal) to the value of the symbol. This is because in the overall image of the machine code, the data image is separated from an instruction image, and all data is required to appear in the machine code after all the instructions.
A data type icon is a label in the data image, and the update adds to the value of the icon (i.e. its address in memory) the total length of the instruction image, plus the code start address of the code, which is 100.

The icon table now contains the values of all the icons needed to complete the memory image (except values of external icons).

In the second transition, the assembler completes using the symbol table the encoding of all the instructions that have not yet been fully encoded in the first transition.

Skeleton algorithm of the assembler

To sharpen the understanding of the assembler's work process, we will present below a skeletal algorithm for the first and second transitions.

Attention: There is no obligation to use this particular algorithm.

As mentioned, we divide the machine code image into two parts: the instruction image (code), and the data image (data). Each part will have a separate counter: IC (instruction counter) and DC (data counter).

We will construct the machine code to suit memory loading starting from 100.

At each transition start reading the source code from the beginning.

First transition

1. IC -> 100, DC -> 0.
2. Read the following line from the source file. If the source file runs out, go to 17.
3. If this is a comment line or a blank line, return to 2
4. Is the first field in a row a label? If not, go to 6.
5. Turn on flag "has a symbol setting."
6. Is this a data storage prompt line, that is, is it a .dh, .dw, .db, or .asciz prompt? If not, go to 9.
7. If there is a symbol definition (label), enter it in the symbol table with the data property The value of the symbol will be DC. (If the icon is not a valid label, or the icon is already in the table, an error must be reported).
8. Identify the type of data, and encode it in the memory image of the data. Also record in the data image the current value of the DC data counter, as the memory address of (the first home of) the data defined in the current row. Then, increase the DC by adding the total length of the defined data. Return to 2.
9. Is this an .extern prompt or an .entry prompt? If not, go to 12.
10. If this is a .entry prompt return to 2 (the prompt will be handled in the second transition).
11. If this is an .extern prompt, insert the icon that appears as an operand of the prompt into the symbol table with the value 0, and with the external property (if the symbol is not a valid label, or the symbol is already in the table without the external property an error must be reported). Return to 2.
12. This is a line of instruction. If there is a symbol definition (label) enter it in the symbol table with the .code property, the value of the symbol will be IC (if the symbol is not a valid label, or the symbol is already in the table, an error must be reported).
13. Look for the name of the action in the action names table, and if it is not found, then report an error.
14. Analyze the structure of the instruction operands according to the type of instruction. If an error is detected, it should be reported.
15. Build the binary encoding of the instruction, in full or in part as much as possible (what cannot be encoded now, must be completed in the second transition) Add the encoding to the memory image of the instruction. Also write in the instruction image the current value of the IC as the memory address of the added instruction.
16. Update IC > IC +4. go to 12.
17. The source file is called in its entirety. If errors were found in the first transition, stop here (there will be no second transition and no output files will be built).
18. Save the final values of IC and DC (we will call them ICF and DCF). We will use them to build the output

files, after the second transition.

19. In the table of symbols, update the value of each symbol that is characterized as data, by adding the value ICF (see explanation below).
20. Update in the memory image of the data the addresses of all the data (as recorded in step 8) Adding the ICF value to each address.
21. Start a second transition.

First transition

1. Read the following line in the source code. If the source code runs out, go to 9.
2. If this is a comment line or a blank line, return to 1
3. If the first field in the row is an icon (label) skip it.
4. Is this a prompt line that is not a .entry prompt? If so, return to .1
5. Is this a .entry prompt? If not, go to 7.
6. In the symbols table, add the entry property to the properties of the symbol that appears as an operand of the prompt (if the symbol is not in the symbol table, an error must be reported). Return to 1.
7. This is a line of instruction. Complete the missing binary encoding of the instruction in the memory image using the table of symbols. In J-instruction (except stop) the operand label address must be encoded.
8. This is a line of instruction. Complete the missing binary encoding of the instruction in the memory image using the table of symbols. In J-instruction (except stop) the operand label address must be encoded.

   In conditional branching instruction, the distance from the current instruction address to the destination address (the label address of the destination operand) must be calculated and coded. And insert this distance into the appropriate field in the instruction encoding. If a symbol that is not in the table of symbols is required, or in the case of conditional branching if a symbol characterized as external is required, an error must be reported and return to 1.
9. The source code is scanned in its entirety. If errors were found in the second transition, stop here (no output files will be built).
10. Build the output files (more details below).

We will run this algorithm on the example program we saw earlier, and present the binary code obtained in the first and second transition.

Below is the sample plan again.

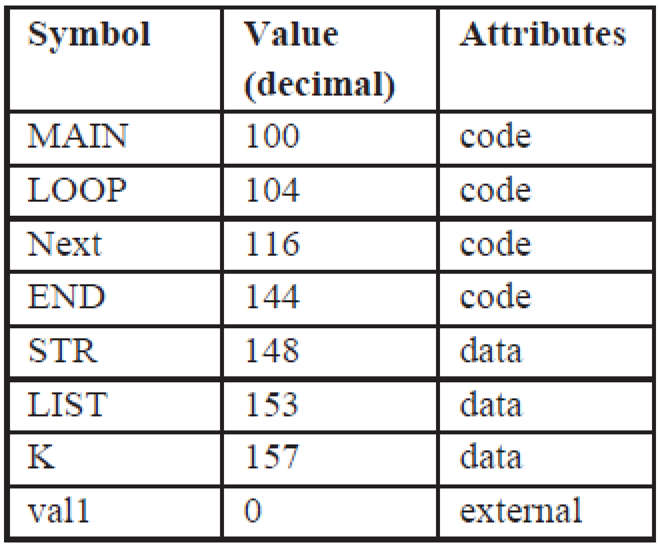
```
MAIN:   add     $3,$5,$9
LOOP:   ori     $9,-5,$2
        la      val1
        jmp     Next
Next:   move    $20,$4
        bgt     $4,$2,END
        la      K
        sw      $0,4.$10
        bne     $31,$9,LOOP
        call    val1
        jmp     $4
END:    stop
STR:    .asciz  "aBcd"
LIST:   .db     6,-9
        .dh     27056
.entry  K
K:      .dw     31,-12
.extern val1
```

We will make a first transition on the code above, and build the table of symbols. We will also encode in this transitionage the whole picture of the data, as well as parts of the picture of the instructions, as far as possible at this stage.

We will mark "?" In the picture of the instructions in each field that cannot be coded in the first transition.

Address (decimal)	**Source Code**	**Machine Code (binary)**
0100	MAIN: add $3,$5,$9	000000 00011 00101 01001 00001 000000
0104	LOOP: ori $9,-5,$2	001101 01001 00010 1111111111111011
0108	la val1	011111 0 ?
0112	jmp Next	011110 0 ?
0116	Next: move $20,$4	000001 10100 00000 00100 00001 000000
0120	bgt $4,$2,END	010010 00100 00010 ?
0124	la K	011111 0 ?
0128	sw $0,4,$10	010110 00000 01010 0000000000000100
0132	bne $31,$9,LOOP	001111 11111 01001 ?
0136	call val1	100000 0 ?
0140	jmp $4	011110 1 0000000000000000000000100
0144	END: stop	111111 0 0000000000000000000000000
0148	STR: .asciz "aBcd"	01100001
0149		01000010
0150		01100011
0151		01100100
0152		00000000
0153	LIST: .db 6,-9	00000110
0154		11110111
0155	.dh 27056	0110100110110000
0157	K: .dw 31,-12	00000000000000000000000000011111
0161		11111111111111111111111111110100

The table of symbols after the first transition is:



Symbol	Value (decimal)	Attributes
MAIN	100	code
LOOP	104	code
Next	116	code
END	144	code
STR	148	data
LIST	153	data
K	157	data
val1	0	external

We will now make the second transition. Use the symbol table to complete the encoding of places marked with "?". The binary code in its final form here is identical to the code presented at the beginning of the topic "Assembler with two transitions."

Address (decimal)	Source Code			Machine Code (binary)
0100	MAIN:	add	$3,$5,$9	000000 00011 00101 01001 00001 000000
0104	LOOP:	ori	$9,-5,$2	001101 01001 00010 1111111111111011
0108		la	val1	011111 0 0000000000000000000000000
0112		jmp	Next	011110 0 0000000000000000001110100
0116	Next:	move	$20,$4	000001 10100 00000 00100 00001 000000
0120		bgt	$4,$2,END	010010 00100 00010 0000000000011000
0124		la	K	011111 0 0000000000000000010011101
0128		sw	$0,4,$10	010110 00000 01010 0000000000000100
0132		bne	$31,$9,LOOP	001111 11111 01001 1111111111100100
0136		call	val1	100000 0 0000000000000000000000000
0140		jmp	$4	011110 1 0000000000000000000000100
0144	END:	stop		111111 0 0000000000000000000000000
0148	STR:	.asciz	"aBcd"	01100001
0149				01000010
0150				01100011
0151				01100100
0152				00000000
0153	LIST:	.db	6,-9	00000110
0154				11110111
0155		.dh	27056	0110100110110000
0157	K:	.dw	31,-12	00000000000000000000000000011111
0161				11111111111111111111111111110100

The table of symbols after the second transition is:

Symbol	Value (decimal)	Attributes
MAIN	100	code
LOOP	104	code
Next	116	code
END	144	code
STR	148	data
LIST	153	data
K	157	data, entry
val1	0	external

At the end of the second transition, if no errors are detected, the assembler builds the output files (see below), which contain the binary code and additional information for the linking step.

As stated, the linking and loading stages **are not to be realized** in this project, and will not be discussed here.

Input files and operating the assembler

When running the assembler, it must be transitioned through one or more of the arguments line (command list list of input file names).
These are text files, including syntactic programs of the assembly language defined in this project.

The name of a file containing an assembly language program must have the extension. ".As" For example, hello.as is a valid name.

For example: Suppose our assembler program is called an assembler, then the following command line:

Will run the assembler on the three input files that are the ergomatics at the command prompt.

The assembler works separately on each input file in the list of arguments, one after the other. If an input file does not open, you should print an error message and move on to the next file. If the input file went through the assembly process without errors, the assembler builds output files for it.

```
assembler test.as myprog.as  hello.as
```

Assembler output files

- An object file, containing the machine code.
- An externals file, with details of all the places (addresses) in the machine code in which an icon is declared as external (an icon that appeared as an operand of the .extern prompt, and is characterized in the symbol table as external).
- An entries file, with details about each icon declared as an entry point (an icon that appeared as an operand of the .entry prompt, and is characterized in the icon table as an entry).

If there is no .extern prompt in the source file, the assembler **does not create** the externals output file.
If the source file does not have any .entry prompt, the assembler **does not create** the entries output file.

The names of the output files are based on the name of the input file, as it appeared in the command prompt, plus an appropriate extension: the extension ".ob" for the object file, the extension ".ent" for the entries file, and the extension ".ext" for the externals file .

For example, running the assembler using the command line: assembler myprog.as will create an output file myprog.ob, as well as output files myprog.ext and myprog.ent as long as there are .extern or .entry instructions in the source file.

We will now present the formats of the output files. Examples will be given later.

Object file format

This file contains the memory image of the machine code, in two parts: the instruction image first, followed by the data image.

Remember, the assembler encodes the instructions so that the instruction image matches the load starting from address 100 (decimal) in memory. Note that only at the end of the first transition do you know the total size of the instruction image. Because the data image is after the instruction image, the size of the instruction image affects the addresses in the data image. This is why it was necessary to update in the table of symbols, at the end of the first transition, the values of the symbols characterized as data (remember, in the skeletal algorithm presented above, in step 19, we added the ICF value to each such symbol). In the second transition, in completing the encoding, the updated values of the symbols, which are adapted to the full and final structure of the memory image, are used.

The assembler can now write the entire memory image into an output file (the object file). The file is made up of lines of text in the format below.

The first line in the object file is a "header", which contains two numbers in a decimal base: the first is the total length of the instruction image (ie the amount of memory cells), and the second is the total length of the data image (the amount of memory cells). Between the two numbers separates one space.
Recall that in the first transition, in step 18 in the algorithm, the values ICF and DCF were preserved. The total length of the instructions is ICF-100, and the total length of the data image is DCF.

The following lines in the file contain the memory image, in ascending order of addresses. Each row has 5 fields: an address in memory followed by the contents of 4 bytes that are from this address, in ascending order from left to right in the row. The address will be recorded in a decimal base in four digits (including leading zeros). The contents of each house will be recorded in a two-digit cadenceimal basis (including leading zero). When the digits A-F are in capital letters.

Between spaces in a row separates one space.

The last row in the file can contain less than 4 bytes, as needed.

Machine instructions are encoded into 4 bytes, so each instruction will occupy an entire row in the object file. As said, the memory is organized in the little-endian method, so the 0-7 bits of the instruction will be the left byte in the order of bytes in a row, the 8-15 bits of the second byte left in the row, and so on.

The data, however, are encoded in lengths that are not necessarily double 4 bytes. However, the data must be entered into the object file in quartets of bytes in each row, without "holes", and while maintaining the correct order (ascending order of addresses in each row from left to right, and ascending order from row to row). See example below.

The entry file format

The entries file is made up of lines of text, one line for each symbol that is characterized in the table of symbols as an entry.

The name of the symbol appears in the row, followed by its value as determined in the symbol table (in a decimal base in four digits, including leading zeros). Between the two fields in a row there is one space. The order of the rows does not matter, because each row stands on its own.

The externals file format

The externals file is also made up of lines of text, a line for each type J instruction in the machine code that uses an icon that is characterized as external. Recall, a list of these instructions was constructed in the second transition (step 8 in the skeletal algorithm).

Each line in the externals file contains the name of the external symbol, followed by the instruction address in which it is required (in a four-digit decimal base, including leading zeros). Between the two fields in a row there is one space.

The order of the rows does not matter, because each row stands on its own.

Attention: There may be several addresses in the machine code where the address of the external icon is required, each such address will have a separate line in the externals file.

We will demonstrate the output generated by the assembler for a source file named ps.as given below.

```
;file ps.as
;sample source code

.entry  Next
.extern wNumber
STR:    .asciz  "aBcd"
MAIN:   add     $3,$5,$9
LOOP:   ori     $9,-5,$2
        la      val1
        jmp     Next
Next:   move    $20,$4
LIST:   .db     6,-9
        bgt     $4,$2,END
        la      K
        sw      $0,4,$10
        bne     $31,$9, LOOP
        call    val1
        jmp     $4
        la      wNumber
.extern val1
        .dh     27056
K:      .dw     31,-12
END     stop
.entry  K
```

Below is the full binary encoding (memory image) of the source file, at the end of the second transition.

Address (decimal)	**Source Code**	**Machine Code (binary)**
0100	MAIN: add $3,$5,$9	000000 00011 00101 01001 00001 000000
0104	LOOP: ori $9,-5,$2	001101 01001 00010 1111111111111011
0108	la val1	011111 0 0000000000000000000000000
0112	jmp Next	011110 0 0000000000000000001110100
0116	Next: move $20,$4	000001 10100 00000 00100 00001 000000
0120	bgt $4,$2,END	010010 00100 00010 0000000000011100
0124	la K	011111 0 0000000000000000010100001
0128	sw $0,4,$10	010110 00000 01010 0000000000000100
0132	bne $31,$9,LOOP	001111 11111 01001 1111111111100100
0136	call val1	100000 0 0000000000000000000000000
0140	jmp $4	011110 1 0000000000000000000000100
0144	la wNumber	011111 0 0000000000000000000000000
0148	END: stop	111111 0 0000000000000000000000000
0152	STR: .asciz "aBcd"	01100001
0153		01000010
0154		01100011
0155		01100100
0156		00000000
0157	LIST: .db 6,-9	00000110
0158		11110111
0159	.dh 27056	0110100110110000
0161	K: .dw 31,-12	00000000000000000000000000011111
0165		11111111111111111111111111110100

The table of symbols in the second transition final is:

Symbol	**Value (decimal)**	**Attributes**
wNumber	0	external
STR	152	data
MAIN	100	code
LOOP	104	code
Next	116	code, entry
LIST	157	data
val1	0	external
K	161	data, entry
END	148	code

Below is the contents of the sample output files.

The ps.ob file:

```
     52 17
0100 40 48 65 00
0104 FB FF 22 35
0108 00 00 00 7C
0112 74 00 00 78
0116 40 20 80 06
0120 1C 00 82 48
0124 A1 00 00 7C
0128 04 00 0A 58
0132 E4 FF E9 3F
0136 00 00 00 80
0140 04 00 00 7A
0144 00 00 00 7C
0148 00 00 00 FC
0152 61 42 63 64
0156 00 06 F7 B0
0160 69 1F 00 00
0164 00 F4 FF FF
0168 FF
```

The ps.ext file:

```
val1 0108
val1 0136
wNumber 0144
```

The ps.ent file:

```
Next 116
K 0161
```

Summary and general guidelines

- The size of the source plan provided as input to the assembler is not known in advance, and therefore the size of the machine code is also unpredictable. However, in order to facilitate the realization of the assembler, a maximum size may be assumed. Therefore it is possible to use arrays to store the machine code image only. Any other data structure (e.g. the symbol table), should be implemented efficiently and cost-effectively (e.g. through a linked list and dynamic memory allocation).
- The names of the output files should match the input file name, except for the extensions. For example, if the input file is prog.as then the output files that will be created are: prog.ent, prog.ext, prog.ob.
- The format of operating the assembler should be as required in the project, without any changes. That is, the user interface will be solely via the command line. In particular, the names of the source files will be transitioned to the assembler program as arguments (one or more) in the command line. Do not add interactive input menus, graphical windows of any kind, etc.
- Care should be taken to divide the assembly of the assembler into several modules (files in the C language) according to tasks. Do not concentrate tasks of different types in a single module. It is recommended to divide into modules such as: first transition, second transition, auxiliary functions (e.g., base translation, line syntax analysis), symbol table, memory map, fixed tables (such as action codes).
- Care must be taken to document the implementation fully and clearly, using detailed comments in the code.
- Excess white characters should be allowed in the assembly file input language. For example, if the instruction line has two operands separated by a comma, then before and after the comma it is allowed to have spaces and tabs in any quantity. Similarly, also before and after the name of the action. Empty rows are also allowed. The assembler will ignore unnecessary white outlines (i.e. skip them).
- The input (assembly code) may contain syntactic errors. The assembler must detect and report any incorrect lines in the input. Do not stop handling the input file after the first error is detected. As detailed as possible messages should be printed on the screen so that it is possible to understand what and where each error is.

  Of course, if an input file contains errors, there is no point in generating the output files (obj, ext, ent).