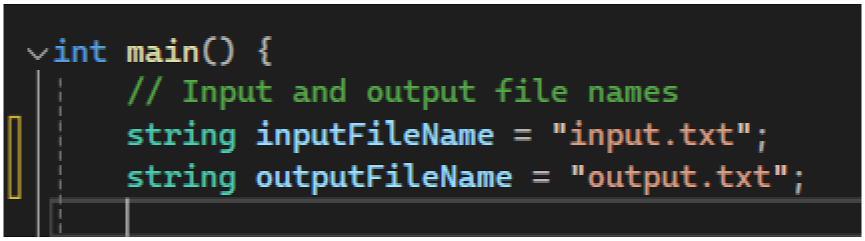
Assembly Instructions

To run code on the processor, the `main.cpp` file is provided, which takes the code written in assembly and converts it into a script understandable by the processor.

To translate the required file, you need to specify:

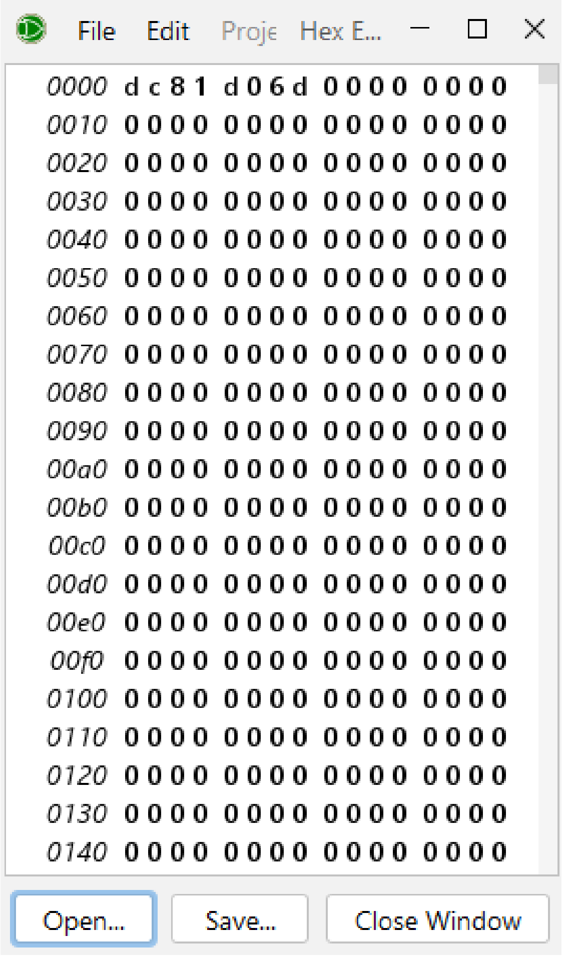
- The name of the file to be translated.
- The name of the file where the translated code will be saved.



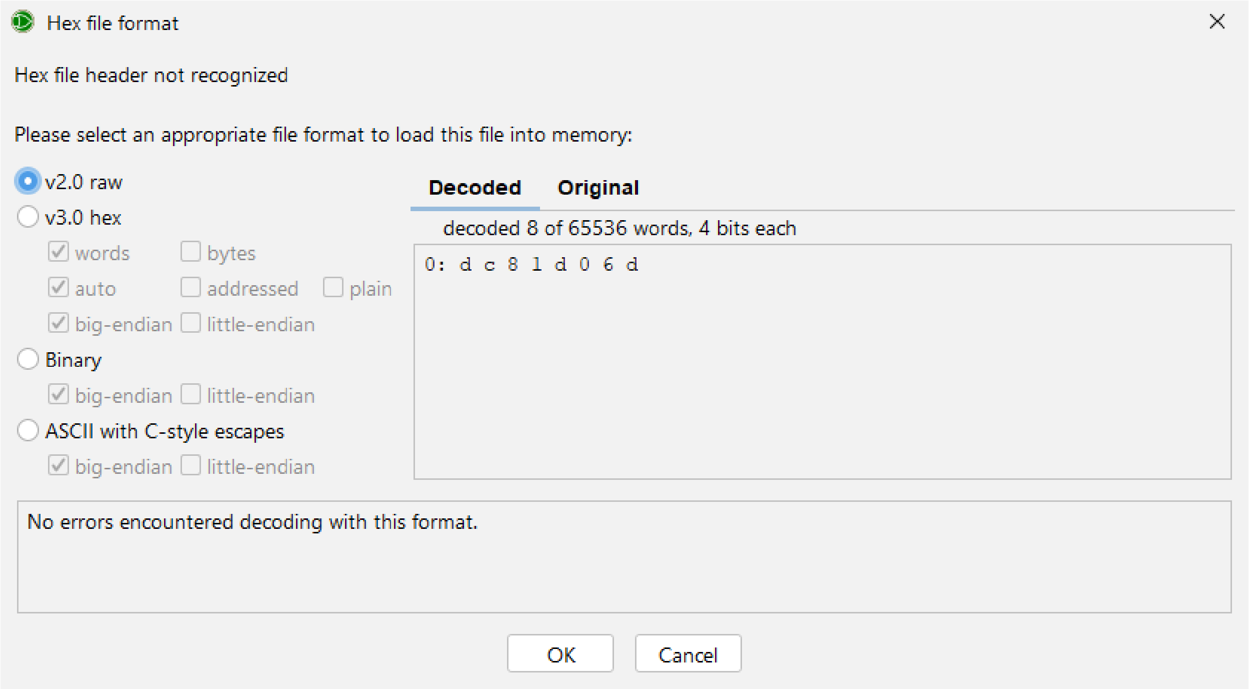
```
int main() {
    // Input and output file names
    string inputFileName = "input.txt";
    string outputFileName = "output.txt";
```

For proper translation, each command must be written on a single line (commands with explanations are listed in the table). Avoid punctuation marks and unnecessary symbols.

The file returned by the assembler must be uploaded to the processor via the "open..." button, selecting the format as "v2.0 raw."



```
0000 dc81 d06d 0000 0000
0010 0000 0000 0000 0000
0020 0000 0000 0000 0000
0030 0000 0000 0000 0000
0040 0000 0000 0000 0000
0050 0000 0000 0000 0000
0060 0000 0000 0000 0000
0070 0000 0000 0000 0000
0080 0000 0000 0000 0000
0090 0000 0000 0000 0000
00a0 0000 0000 0000 0000
00b0 0000 0000 0000 0000
00c0 0000 0000 0000 0000
00d0 0000 0000 0000 0000
00e0 0000 0000 0000 0000
00f0 0000 0000 0000 0000
0100 0000 0000 0000 0000
0110 0000 0000 0000 0000
0120 0000 0000 0000 0000
0130 0000 0000 0000 0000
0140 0000 0000 0000 0000
```



Hex file header not recognized

Please select an appropriate file format to load this file into memory:

- ● v2.0 raw
- ○ v3.0 hex
  - ☑ words □ bytes
  - ☑ auto □ addressed □ plain
  - ☑ big-endian □ little-endian
- ○ Binary
  - ☑ big-endian □ little-endian
- ○ ASCII with C-style escapes
  - ☑ big-endian □ little-endian

decoded 8 of 65536 words, 4 bits each

```
0: d c 8 1 d 0 6 d
```

No errors encountered decoding with this format.

Command Table

#	Command	Syntax	Description
1	LOAD	LOAD From To	Loads data from a memory address (specified in From register) to another register (To).
2	STORE	STORE From To	Stores data from a register (From) into a memory address (specified in To register).
3	ADD	ADD A B To	Adds data from registers A and B, saving the result in To.
4	SUB	SUB A B To	Subtracts data in register B from A, saving the result in To.
5	MUL	MUL A B To	Multiplies data from registers A and B, saving the result in To.
6	SHIFTR	SHIFTR P1 P2 To	Shifts data in register P1 to the right by the amount in P2, saving the result in To.
7	DIV	DIV A B To	Divides data in register A by B, saving the result in To.
8	SHIFTL	SHIFTL P1 P2 To	Shifts data in register P1 to the left by the amount in P2, saving the result in To.
9	NOT	NOT P To	Performs a NOT operation on data in register P, saving the result in To.
10	AND	AND A B To	Performs an AND operation on data from registers A and B, saving the result in To.

11	OR	OR A B To	Performs an OR operation on data from registers A and B, saving the result in To.
12	XOR	XOR A B To	Performs an XOR operation on data from registers A and B, saving the result in To.
13	NAND	NAND A B To	Performs a NAND operation on data from registers A and B, saving the result in To.
14	NOR	NOR A B To	Performs a NOR operation on data from registers A and B, saving the result in To.
15	JUMP	JUMP BR	Jumps to the line specified in register BR.
16	LI	LI To INT	Loads an integer (INT, between 0 and 255) into register To.
17	BREQ	BREQ P1 P2 BR	Jumps to the line in register BR if the data in registers P1 and P2 are equal.
18	BRNE	BRNE P1 P2 BR	Jumps to the line in register BR if the data in registers P1 and P2 are not equal.

Additional Functionality

Features such as timers and GPIO are accessed by writing to and reading from specific registers.

The processor has 16 available registers. However, some of them are reserved for specific functionalities.

GPIO ports:

- **Input:** 4 bits
- **Output:** 4 bits

Registers

Register number	Purpose
0	OCA register - If a number is written to this register, the timer will be set to the value of this register and when it reaches this value, it will signal the IFOCA register.
1	OCB register - If a number is written to this register, the timer will be set to the value of this register and when it reaches this value, it will signal the IFOCB register.
2-5	Output registers - If you write the number 1 to these registers, we will have a logical high on the corresponding pin, and for any other value, a logical low.

6-9	Information read pins - These registers have been replaced with direct input pins, so it is impossible to write anything to them, only to retrieve information. 1 means logical high and 0 means logical low, they do not provide other numerical values.
10	IFO register - If a number is written to this register, when the timer reaches its maximum value, the code will jump to the line written in this register.
11	IFOCA register - If the timer reaches the OCA value, the code will jump to the line written in this register. (If the OCA register is empty, this register can be used freely)
12	IFOCB register - If the timer reaches the OCB value, the code will jump to the line written in this register. (If the OCB register is empty, this register can be used freely)
13	Final register - If you write the number 1 to this register, it means the end of the code and the counter will not increase any more. For any other value, this register will work as a buffer.
14-15	Free-use registers - These registers do not have any specific purpose and can be used freely in your code.

Tests

1. Simple loop _ the value of the register increases by one until it reaches 60 and then is output to the operating memory at address “1” (60 in hexadecimal is 3c).
   simple loop _ assembly.txt
   simple loop.txt
2. LED test _ the LEDs are turned on and off cyclically.
   LED _ assembly.txt
   LED.txt
3. Timer test _ the code The timer is set on the seventh line, then performs the operation of outputting the answer to the operating system and returns to the seventh line, thus completing the code (the answer is 100 in hexadecimal is 64 and is output to address number 5).
   timer _ assembly.txt
   timer.txt

The given tests are placed in the assembler file.