ISA Project

In this project we implemented an SIMP processor simulator (resemble to MIPS processor but simpler) in C language.

First, we created a program(asm.exe) that translates an assembler code to machine code and writes it to the main memory (txt file in this program).

The second program(sim.exe) simulates the instructions from the main memory.

Assembler(asm.exe):

Input arguments: command line receives 3 input arguments:

1. program.asm- the assembler code
2. memin.txt – an empty file
3. XXXX.asm- a code assembly file

output: writes to mem.txt the new memory following the given assembler code.

Functionality:

The program passes twice over the assembly code. In The first pass we perform the translation of the assembly code to the ISA conventions of SIMP and mapping all the labels. The first pass prints the memory to a temporary file named "write.txt".

In the second pass, the program reads from "write.txt", replace the label's names with their PC address and prints it to "mem.txt".

Structures:

```
typedef struct Label label;
struct Label {
	char name[MAX_lEN_NAME];//field name - contains label's name
	int adress;//field address - contains the PC address the label was found
	label* next;// this field is for dealing with collision in the hashtable
};

typedef struct HashTable {
	label** items;// items is an array of pointers to labels
	int size;// array's size is 'size' as the user decide
}HashTable;
```

In this program we used 'hashtable' technique for saving all the labels with a minimal time complex and minimal memory allocation.

Main functions:

```
void PassOne(FILE* f1, HashTable* table)
```

- This function performing the first pass on the assembly code. Pass one function is calling other function in order to extract and clean all the relevant parts from the assembly code. Pass one function is also create the mapping (using the hash table structure) of the labels and the labels addresses (PC). In our project, handling of ".word" instruction perform during the pass one.

- The output of this function is a temporary file and the hash-table, for second pass uses.

```
void PassTwo(FILE* memin, HashTable* table)
```

- This function performing the second pass on the assembly code. At this point, the assembly code is already according to the memory format which accepted in SIMP, except for the labels (The labels remains as it appears in the assembly code, for now).
  Pass two replace the names of the labels with their PC addresses according to the hash table map which created in the first pass.
- The function output is the memin file.

Simulator (sim.exe):

Input arguments: command line receives 12 input arguments:

1. sim.exe- simulator program
2. memin.txt- memory before the simulator.
3. diskin.txt- initial memory of the hard disk.
4. irq2in.txt- as described in the project instruction.
5. memout.txt- empty file
6. regout.txt- empty file
7. trace.txt- empty file
8. hwregtrace.txt- empty file
9. cycles.txt- empty file
10. leds.txt- empty file
11. display.txt- empty file
12. diskout.txt- empty file

This program simulates SIMP processor

Functionality:

We used 'w+' mode for 'memout.txt' and 'diskout.txt' in order to allow the program read and write simultaneously to those text files. First, the program duplicates the 'memin.txt' and 'diskin.txt' to 'memout.txt' and 'diskout.txt'. Then, the program reading from 'memout.txt' the instructions and implement them. The program starts from PC=0 (PC is the number of line in 'memout') and the PC changes every instruction regarding to the specific instruction that was read.

The output files are elaborated in the previous section (as inputs 4-11). These files are describing the processor actions during the program as required in the project instructions.

Important variables:

```
int PC // the current PC the program is reading the instruction
int PC_next // the next PC the program need to read from.
```

Main functions:

```
void Simulator(FILE* Memout,FILE *trace, FILE *leds, FILE *diskout,FILE *hwregtrace, FILE
*regout, FILE *cycles, FILE *display, FILE *irq2in);
```

- This is the major function. In this function the program reading from 'memout' the instructions using a 'while' loop, and breaks when the program encounter a 'HALT' instruction. Each iteration of the main loop simulates one clock cycle: implements one instruction and checks for interrupts. PC promoting or lunching to interrupt handler performed in this function as well.

```
void Instructions_0_to_13_opcode(int R[], int opcode, int rd, int rs, int rt, int PC,
int* PC_next);
```

```
void Instructions_lw_sw(int R[], int opcode, int rd, int rs, int rt, int PC, int*
PC_next, FILE* Memout);
```

```
void IO_Instructions(FILE* hwregtrace, FILE* leds, FILE* diskout, FILE* memout, FILE*
display, int opcode, int R[], int IORegister[], int rd, int rs, int rt, int* PC_next, int
clock_cycle);
```

- These three functions perform the instruction. Each clock cycle one of these functions operates and computes according to the instructions and conventions of SIMP processor. The result output of these functions restore in the memory (sw ins), read from the memory (lw ins), or changing the current values of the registers.

```
void routine_timer(int IORregister[]);
void routine_file(FILE *irq, int IORregister[], int clock, int* num);
void routine_disk(int IORegister[], int* timerdisk);
```

- These three functions handling the routine tests required in every clock cycle for generating interrupts, when its needed- by promoting the relevant counters and IO registers.