Creating a simulator for KUE-chip2

Hiroaki Yamamoto

May 25, 2012

1 Requirements

- Basically, this simulator works as an interpretor for a specified machine language file.
- The simulator needs machine language file to input program area.
- A user can input data into data area by choosing a file.
  In addtion to this, the user can input initial state for registers/flags.
- To avoid infinite loop, the user can specify step-limitation.
- The result must be stored into the specified file name as comma separated values.
- Those above options are provided via command line options. The details are section 2.

2 Options

Table 1: Command line options

Option	Alias	Argument	Attribute	Default value	Description
`--initial-state`	-ii	File Name	Optional		Sets an initial registers/flags state
`--data-area`	-d	File Name	Optional		Sets data into data area.
`--step`	-s	Number	Optional		Sets step-limitation
`--output`	-o	Number	Optional	result.csv	Sets an output file.
`--input`	-i	File Name	Needed		Sets a machine language file.

Note that `--input` option is Default Option. i.e. If you execute the simulator with the following, the command line argument is interpreted as `--input`:

```
./simulator program.bin
```

3 Data formats

3.1 Program data (Machine language file)

The program data is stored as it is. Note that the data is **BINARY data hand-assembled**.

3.2 Data area file

The data area file is stored as it is. Uninitialized data is interperted as 0xff. The file size is smaller than 256 bytes, each extra area is set to 0xff.

3.3 Initial Sates for Registers/flags

The data format is the following:

Table 2: Data structure of Initial States for Refisters/flags

0-7	8-15	16-24	25	26	27	28	29-32
ACC	IX	IBUF	CF	VF	NF	ZF	0

The unit of Table.2 is bit.

3.4 Output file

Output file is stored into a file as Comma Separeted Values with the following schema:

```
Object code;Mnemonic code;PC;ACC;IX;CF;VF;NF;ZF;IN;OUT;MEMORY
```

4 Data Structure

The data structure passing each function is the following.

```
struct buf{
    unsigned char bits;
    int flag;
} typedef buf;

struct data{
    unsigned short obj_code;
    unsigned char *mnemonic_code;
    unsigned char pc,acc,ix;
    unsigned char flags=0x0X;
    buf in,out;
    unsigned char memory[256];
    int memory_changed;
    unsigned char modified_addr;
    unsigned char prev;
```

```
    unsigned char now;
} typedef data;
```

Note that `mnemonic_code` must be allocated dinamically. Moreover, the data will be passed as a pointer to reduce memory usage. "flags" variable has values below 4 bits, and the structure is the following:

3			0
CF	VF	NF	ZF

5 Output

The format to output to stdout is the following:

```
{Object Code} {Mnemonic code} [PC:{PC} ACC:{ACC} IX{IX}\
 CF={CF} VF={VF} ZF={ZF} NF={NF} IN={IN} OUT={OUT} MEMORY={memory diff}
```

string is a variable corresponding to a register/flags.

Note that memory diff mus be a diff of memory. For example, if the address 0x10 before modified was 0xff and changed to 0x00, the memory_diff syntax is the following:

```
{address}:{the value before modified}->{the value after modified}
```

The meanings of other variables are the following:

Table 3: The meaning of each variable

{PC}	Program Counter
{ACC}	Accumulator
{IX}	Index Register
{CF}	Carry Flag
{VF}	Overflow Flag
{ZF}	Zero Flag
{NF}	Negative Flag
{IN}	IBUF
{OUT}	OBUF

Note that those above variables are hexadecimal, not string.