Assignment 4 - Software I, Summer 2003 (0368-2157-20)

`http://www.cs.tau.ac.il/~efif/courses/software1`

Due: Sep. 5, 2003

In this assignment you are asked to create a simple development environment for a degenerate CPU. The development environment consists of 1 awk script, namely *gen*, 2 programs, namely *asm*, and *sim*, and a single makefile for all.
The *gen* awk script generates C source-files that specify the instruction set of the CPU based on an input text file . The *asm* program uses the code generated by the *gen* program to read a text file that contains a program written in symbolic assembly, and converts it to a sequence of instructions. It writes the instruction sequence into a text file. Finally, the *sim* program reads the program file produced by the *asm*, and executes the instructions it contains sequentially. It also uses the code generated by the *gen* program.
The CPU consists of at most 256 registers. Each register can accommodate an **int**. The instruction word consists of 32 bits divided into 4 fields as follows:

opcode - the operation code

operand$_0$ - the index of the register to hold the first operand if applicable

operand$_1$ - the index of the register to hold the second operand if applicable

result - the index of the register to hold the result if applicable

The set of operations the CPU can perform is known in advanced, and listed below. Let R denote the CPU register file of 256 registers. Let o_0 and o_1 denote the 2 operand fields, and let r denote the result field.

add - $R[r] \leftarrow R[o_0] + R[o_1]$

sub - $R[r] \leftarrow R[o_0] - R[o_1]$

mul - $R[r] \leftarrow R[o_0] * R[o_1]$

div - $R[r] \leftarrow R[o_0]/R[o_1]$

in - read an **int** from standard input into $R[r]$

out - write the **int** in $R[o_0]$ to standard output

Ex 6.1 gen

The length of each field in bits, the position of the fields within the instruction word, and the possible values the opcode field may contain are all specified in a text file, possibly edited by a non-programmer in a fixed format, provided as input to the *gen* script. This script generates two C source-files, namely `inst.h` and `inst.c` as follows.

There are 2 types of statements in a legal input file. A *field* statement specifies a field in the instruction word, and a *value* statement specifies an optional value the last specified field may contain.
A *field* statement starts with the **field** keyword, followed by the field name, followed by the field starting position in the instruction word in bits, followed by the field length in bits. A *value* statement starts with the **value** keyword, followed by a mnemonic name, followed by the corresponding value itself in hexadecimal format.
For each *field* statement in the input file the *gen* program must generate 3 directive statements that specify the starting position of the field in bits, the length of the field in bits, and the field mask. The 3 directives are written into `inst.h`. For example:
Input file:

```
field opcode   0  8
field operand0 8  8
field operand1 16 8
field result   24 8
```

Output file:

```
#define OPCODE_POS      0
#define OPCODE_LEN      8
#define OPCODE_MASK     0x000000ff

#define OPERAND0_POS    8
#define OPERAND0_LEN    8
#define OPERAND0_MASK   0x0000ff00

#define OPERAND1_POS    16
#define OPERAND1_LEN    8
#define OPERAND1_MASK   0x00ff0000

#define RESULT_POS      24
#define RESULT_LEN      8
#define RESULT_MASK     0xff000000
```

For each *value* statement in the input file the *gen* program must generate 1 directive statement in the `inst.h` file that specifies the value of the option. For a set of field options, a directive that specifies the number of options in the set is generated as well. For example,
Input file:

```
value add 0x1
value in  0x3
value out 0x4
```

Output file:

```
#define OPCODE_ADD      0x1
#define OPCODE_IN       0x3
#define OPCODE_OUT      0x4
#define NUM_OPCODES     3
```

In addition, the *gen* script must generate an array of opcodes initialized with all possible opcodes and write it into the `inst.c` file. An element in the array is a structure that consists of the opcode mnemonic name and the corresponding value. For example:

```
Opcode Opcodes[] = {
  {"add", OPCODE_ADD},
  {"in", OPCODE_IN},
  {"out", OPCODE_OUT}
};
```

The *gen* script must insert the statement that includes `inst.h` in front of the `inst.c` file, and the definition of the `Opcode` struct, and the declaration of the `Opcodes` array as extern into the `inst.h` file. Finally, the code in the `inst.h` file must be embedded within `ifndef, define, endif` pragmas as listed below, to protect it from being compiled more than once.

```
#ifndef INST_H
#define INST_H
 code
#endif
```

As a convention the name of any specification input file ends with the "`.t`" suffix (for text). Suppose that `spec.t` contains the examples above. Typing the command below will produce `inst.h` and `inst.c` as specified.

```
gen spec.t
```

Ex 6.2 asm

The *asm* program reads a text file that contains source code in symbolic assembly, and converts it to a sequence of instruction words. It writes the instruction words into a text file at the same order they appear in the input file, each word occupying a single line. As a convention the name of any input file ends with the "`.s`" suffix (for symbolic assembly). By default the output file name has the same basename as the input file name, and ends with the "`.e`" suffix (for executable).
In symbolic assembly a comment starts with the '#' symbol at the beginning of the line, and ends at the end of the line. Each statement represents a single instruction, starts with the mnemonic name of the operation, and ends at the end of the line.
Each one of the 4 binary operations are followed by o_0, o_1, and r in this order. The *in* operation is followed by r, and the *out* operation is followed by o_0.
For example, suppose that an input file `prog.s` contains:

```
# A simple example
in 2
in 3
add 2 3 4
out 4
```

Given that the opcodes of *in*, *add*, and *out* are 0x3, 0x1, 0x4 respectively, typing the command:

```
asm prog.s
```

results with the file `prog.e` containing:

```
0x02000003
0x03000003
0x04030201
0x00000404
```

The `asm.c` source-code file includes `inst.h` and uses the directives in it, as well as the opcodes defined in the global array in `inst.h`. Link `asm.o` with `inst.o` to generate *asm*.

Ex 6.3 sim

The *sim* program reads the executable produced by the *asm* program and simulates its execution. For example, executing the program above:

```
sim prog.e
10
20
```

results with:

```
30
```

The *sim* program also uses `inst.h` generated by *gen*, but it doesn't have to be linked with `inst.o`.

Ex 6.4 makefile

Provide a `makefile` that supports the following commands:

make gen - generates `gen`

make asm - generates `asm`

make sim - generates `sim`

make *prog*.e - applies `asm` on the source file ***prog*.s** to generate the executable ***prog*.e**, where ***prog*** stands for the base name of an input file.

make clean - removes all the object and executable files, and `inst.h` and `inst.c`

Assume that the makefile variable $SPECFILE contains the name of the input file to *gen*. Place the statement below at the top of the makefile to set it to `inst.h` by default.

```
SPECFILE ?=spec.t
```

Make sure that all dependencies are accounted in the makefile, so that when a certain file is touched, all files that depend on it, but no other files, are rebuilt. For example, making `prog.e` in a clean state, starts a chain reaction where *gen* is executed to generate `inst.h` and `inst.c`, then *asm* is compiled, linked, and executed to generate `prog.e`.

1 Additional Instructions

The spec. file must specify all 4 fields. Otherwise, you will get compilation errors when compiling `asm` or `sim`. This is acceptable. Naturally, you may verify that they are specified already in `gen`, and exit with an error code, in case one or more is missing.
Not all 6 opcodes must be specified. This deficiency has no effect on `gen`. `asm` and `sim`, on the other hand, must be prepared to handle all operations (`asm` must distinguish between `in`, `out`, and all the rest, as they have different arguments). A good solution is to embed the code that processes a given operation within `#ifdef,#endif` pair as follows:

```
#ifdef OPCODE_ADD
(process OPCODE_ADD)
#endif
```