1 Project 4

Due: Nov 19, before midnight.

This document first provides the aims of this project, followed by a discussion of its background. It then lists the requirements as explicitly as possible. It then hints at how these requirements can be met. Finally, it describes exactly what needs to be submitted.

1.1 Aims

The aims of this project are as follows:

- To expose you to machine-language.
- To make you write a `Makefile` which supports linking with non-standard external libraries.
- To give you further experience with writing non-trivial C programs.

1.2 Background

This project requires to write a simple simulator for the `y86` ISA described in section 4.1 of the text. Though the authors of the text provide source code for such a simulator, the constraints provided for this project do not make it easy to reuse that source code easily. In any case, writing such a simulator is a useful exercise.

1.3 Requirements

Update the `master` branch of your github repository with a directory `submit¬ /prj4-sol` such that that typing `make` within that directory will build an executable `y86-sim` which is a simulator for the Y86 machine described in Section 4.1 of the text:

The `y86-sim` program can be invoked with the following arguments:

- An option `-i`. If specified, it should force the simulator is to run interactively, pausing after each simulation step.
- An option `-l`. If specified, then the simulator is not run; instead an assembly listing is produced on standard output.
- An option `-v`. If specified, it should force the simulator to output (on stdout) all y86 state changes since the last simulation step.

- An option `-V`. If specified, it should force the simulator to output (on stdout) all CPU registers and state changes since the last simulation step. If both `-v` and `-V` are specified, then `-V` overrides `-v`.
- The names of one or more `.ys` files containing y86 assembly language code.
- Zero or more integer parameters. If specified, those parameters will be loaded into the high memory of the simulated Y86 machine The simulation is started with register `rdi` pointing to the parameters and register `rsi` set to the number of parameters provided.

You are being provided with a library which supports the components of a Y86 machine as well as a Y86 assembler. You are also being provided with a C `main¬()` function which handles all the command-line parsing and sets up a properly initialized instance of a simulated `y86` computer with all the `.ys` programs specified on the command line assembled and loaded into memory. All that you really need to do is fully implement the following function (specified in ysim.h with a skeleton implementation in ysim.c

```
/** Execute the next instruction of y86. Must change status of
 *  y86 to STATUS_HLT on halt, STATUS_ADR or STATUS_INS on
 *  bad address or instruction.
 */
void step_ysim(Y86 *y86);
```

1.4 Provided Libraries

You have been provided with `libcs220` and `liby86` libraries in the course lib directory. The specification headers for the modules provided by these libraries are in the course include directory and the source code in the course src directory.

You may assume that the environment in which your program will be compiled and run will have these libraries available within a `$HOME/cs220` directory.

1.4.1 libcs220

This is a trivial library which provides help with memory allocation and error reporting:

- Provides checked versions `mallocChk()`, `reallocChk()`, and `callocChk¬()` of the memory allocation routines which wrap the standard memory allocation routines with the program exiting on failure. The specification file is in memalloc.h.

- Provides routines for reporting errors using `printf()`-style format strings with one modification: if the format string ends with `:`, then `strerror¬ (errno)` is appended to the error-message. The specification file is in errors.h.

1.4.2 liby86

This library provides a `y86` module which supports the components of a Y86 machine and a `yas` module which is an assembler for Y86 code.

The y86 Module

The specification for this module is in the y86.h header file and implementation in y86.c. This module defines the state of a y86 computer.

- It contains 14 registers with numbers given by the `Register` **enum** `REG¬ _RAX`, `REG_RCX`, ..., `REG_R14`.
- It contains a condition code with 3 condition bits corresponding to the overflow flag, sign flag and zero flag at bit positions `OF_CC`, `SF_CC` and `ZF_CC` respectively.
- It contains a machine status which determines whether the machine is running or stopped intentionally or because of an error.
- It contains a memory with size specified at initialization time.

The machine is accessed as an ADT using operations which allow reading/writing registers, memory, the PC, status and condition codes. Note that any operation involving an address can set the machine status to `STATUS_ADR` if the address is invalid; it is a good idea to check the status after any such operation.

The yas Assembler Module

The specification for this module is in the yas.h header file and implementation in yas.c.

This module provides an assembler for the y86 ISA. It provides load-and-go functionality to directly assemble code into the memory of a simulated y86 computer. It also provides functionality to provide an assembly listing.

This module is accessed by the code you are being provided with and you should not need to even look at these files (except out of general curiosity).

1.5 Provided Project Files

The files prj4-sol directory contains the following files:

ysim Module Specification in ysim.h and skeleton implementation in ysim.c. You will need to add code to `ysim.c` to implement your project..

main.c This contains a `main()` function for your program which handles all the details of processing the command-line arguments.

1. It creates a new instance of a y86 computer:
2. It assembles the `.ys` assembly files specified on the command line. If the command line parameters specified the `-l` option requesting a listing, then it outputs an assembly listing on standard output and exits. Otherwise it loads the machine code corresponding to the `.ys` files into the memory of the y86 instance.
3. If the command line arguments include any integer parameters, then those parameters are loaded into high memory with the `rdi` register set up as an `Word argc` count and the `rsi` register set up as a `Word argv[]` pointer.

   Note that the code in `main.c` does not check whether loading the integer parameters into high memory will overwrite any of the code loaded from the `.ys` files. This should not be a problem for the small programs being run by this project.
4. Starts simulating the loaded code by stepping the simulator by repeatedly calling the `step_ysim()` function you are required to write until the machine `status` changes to something other than `STATUS_AOK`. Between each simulation step it dumps machine state if the `-v` or `-V` command-line options were specified, or pauses if the `-i` command-line option was specified.
5. Dumps the state of the machine and changes on standard output and terminates.

README A README file which must be submitted along with your project. It contains an initial header which you must complete (replace the dummy entries with your name, B-number and email address at which you would like to receive project-related email). After the header you may include any content which you would like read during the grading of your project. If your project is not complete, document what is not working in the `README`.

Example `.ys` files in directory extras These example files show complete y86 programs which should be runnable by your simulator. Gold outputs for running each file through your simulator without any options are shown in the corresponding `.out` files. (The main-sum.out gold output was produced using the additional argument produced using `$(seq 1 10)`).

Test shell script test.sh This shell script compares the output of running your simulator on one or more `.y86` files with the corresponding `.out` gold output. It is simply invoked with the paths to one-or-more `y86` files. Examples:

```
$ D=$HOME/cs220/projects/prj4/extras
```

```
$ $D/test.sh $D/halt.ys
$ $D/test.sh $D/*.ys
```

1.6 Hints

The following points are worth noting:

- Y86 is actually Y86-64, a 64-bit machine. This is reflected in the **typedef**'s used for `Word` and `Address` in y86.h. You should strive to write your code in a manner which does not depend on these sizes. For example, your code should not assume that the length of a `irmovq` is 10; instead it should use an expression like `1 + sizeof(Byte) + sizeof(Word)` for the length. That enables retargeting of the simulator to say a Y86-32.
- Your simulator will need to simulate a basic instruction cycle:
  1. Fetch instruction specified by program counter `pc` from memory.
  2. Execute the instruction just fetched, updating CPU and memory as required by the semantics of the instruction.
  3. Set the program counter `pc` to the address of the next instruction to be executed. Except for control transfers, this will be the address of the next sequential instruction
- While developing your project in your `submit/prj4-sol` directory, you will frequently be using the test files in the extras directory located at path `~/cs220/projects/prj4/extras`. Instead of typing that path repeatedly, it may be useful to setup a shell variable `D=$HOME/cs220/projects/¬ prj4/extras` and then use `$D` in your commands to refer to the `extras` directory.

The following points are not prescriptive in that you may choose to ignore them as long as you meet all the project requirements.

1. Read the Y86 description given in section 4.1 of the textbook. Also understand the y86 ADT you have been provided with (note that you should not need to use the `get_memory_pointer_y86()` or `dump_changes_y86()` operations.
2. Start your project in a manner similar to how you start a lab. Set up a `prj4` branch and copy the provided prj4-sol directory into your `i220?/¬ submit` directory in that branch.
3. Create a `Makefile` for your project. The `Makefile` should be set up to build `main.o` and ysim.o‘ and link them along with the `cs220` and `y86` libraries into the executable `y86-sim`.
   - The interface to the libraries in the include directory will be needed to compile the source code. The `-I` option to gcc can be used to

specify the directory those interfaces. You can set that up in your `Makefile` by defining the make variable `CPPFLAGS` as follows (the following assumes that make variable `COURSE` is set to `cs220`):

```
CPPFLAGS = -I $$HOME/$(COURSE)/include
```

This variable is understood by the implicit compilation commands built into make.

- The link command will need to tell gcc which libraries to use. You can ensure that by having the link command use the `LDFLAGS` make variable defined as follows:

```
LDFLAGS = -L $$HOME/$(COURSE)/lib -l cs220 -l y86
```

Note that the library specification **must** follow the names of any object files which use those libraries.

If you have problems setting up the `Makefile`, review *Lab 1* or any `make` documentation on the web

Once you have set up your `Makefile` correctly, you should be able to build the project. Running the executable without any arguments should produce a usage message:

```
$ ./y86-sim
usage: ./y86-sim [-s] [-v] [-V] YAS_FILE_NAMES... INT_INPUTS...
           -l:  produce assembler listing only
           -s:  single-step program
           -v:  verbose: dump changes
                  after each instruction
           -V:  very verbose: dump all registers
                  after each instruction
```

If you get an error about missing libraries, your `LD_LIBRARY_PATH` is not set up correctly (it should be set up if you followed the directions provided at the start of the semester). Try

```
$ export LD_LIBRARY_PATH=$HOME/cs220/lib
$ ./y86-sim
```

to solve the problem.

You should also be able to get assembly listings:

```
$ ./y86-sim ~/cs220/projects/prj4/extras/asum.ys -l
```

4. Get started on your `step_ysim()` function. Read the pc and then fetch the opcode byte addressed by it; whenever you read/write memory, you need to check for an address error using something like:

```
if (read_status_y86(y86) != STATUS_AOK) return;
```

Extract the base opcode from the fetched byte by extracting its high nybble. Do a switch on this base opcode. For now, simply add a `default` case for unhandled instructions. Have this default case simply set the machine status to `STATUS_INS` to indicate a bad instruction.

Run your simulator on any `ys` file. For example,

```
$ ./y86-sim ~/cs220/projects/prj4/extras/halt.ys
```

The machine should stop with status printed as `BAD_INS`.

5. Add a case for the `halt` instruction which simply sets the machine's status to `STATUS_HLT`. Run the above command again. You should see that the machine stops with status set to `HLT`.

6. Add a case for a `nop` instruction which simply advances the pc. Verify your output by running your simulator on the `nops-halt.ys` file.

   You can verify that your output matches the provided gold outputs by using the provided test.sh script.

   ```
   $ D=$HOME/cs220/projects/prj4/extras
   $ $D/test.sh $D/halt.ys $D/nops-halt.ys
   ```

   The script should succeed silently if your output matches the gold output.

7. Implement the `irmovq` instruction by adding a `case` in the `switch`. You will need to read the next byte following the opcode byte to get the register specifier. Extract the register from the low nybble of the fetched register specifier. Then fetch the next word from memory as the immediate operand. Finally set the register to the fetched immediate operand and update the pc to point to the next instruction.

   Add a test program to verify that your implementation is correct. Start out with the `halt.ys` program and add a `irmovq` instruction before the `halt` to move some distinctive value into a register, say `rax`. Verify that the register has that distinctive value when the program simulation stops.

8. Add support for `call` and `ret` instructions. The former will need to get the address of the function being called from the word following the opcode byte. The return address will need to be pushed on to the stack before the pc is updated to transfer control to the called function.

   Implement `ret` to transfer control to the address popped from the stack.

   You should now be able to run the basic.ys test.

9. Add code to implement the memory move instructions `mrmovq` and `rmmovq`. You should now be able to run the mem-mov.ys test.

10. Complete the `check_cc()` function in the provided `ysim.c`. This will allow you to implement the `cmovXX` (which includes `rrmovq`) instructions. You should now be able to run the mov.ys test.

11. Add support for the `OP1` instructions by completing the provided `op1¬ ()` function and its support functions `set_add_arith_cc()`, `set_sub_¬ arith_cc()`, `set_logic_op_cc()`. The code for the condition codes for the different `OP1` instructions follows from the meaning of the condition codes.

    Addition Zero flag set if the result is zero, sign flag set if the result is negative, overflow flag set if the operands have the same sign and the sign of the result is different from the sign of the operands.

    Subtraction Zero flag set if the result is zero, sign flag set if the result is negative, overflow flag set if the operands have the opposite sign and the sign of the result is different from the sign of the first operand.

    Logical Operations `andq` and `xorq` Zero flag set if the result is zero, sign flag set if the result is negative, overflow flag always reset.

    You should now be able to run the abs-diff.ys test.

12. Implement the `pushq` and `popq` instructions. Be sure to implement the correct behavior when the operand for those instructions is `rsp` as specified in problems 4.7 and 4.8 of the text. You should now be able to run the pushq-rsp.ys and popq-rsp.ys tests.

13. Implement the remaining instructions. Use the existing tests or add in your own tests to verify.

14. Iterate the previous steps until all requirements are met.

1.7 Submission

Submit as per previous projects.