Assignment 2

HW2: X86lite

You are expected to be working in a group of two people for this assignment. Please submit a team.txt file specifying your team members with one matriculation number and full name in your group per line, use the following syntax (no dashes, no spaces in the matriculation number):
12123123, Grace Hopper
12312324, Tommy Flowers

Overview

In this project you will implement an assembler and simulator for a small, idealized subset of the X86-64 platform that will serve as the target language for the compilers we build in later projects. This project will continue to help get you up to speed with OCaml programming -- we'll need a few more constructs not used in HW1. You will also implement a non-trivial assembly-language program *by hand* to familiarize yourself with the workings of the X86 architecture.

Getting Started

To get started, download the project source files (and, if you use Eclipse, create a project whose executable is `main.native` or `main.byte`). The files included in `hw02.zip` are briefly described below. Those marked with ∗ are the only ones you should need to modify while completing this assignment.

Note: You need to add the `nums` library (which provides the `Big_int` implementation) to compile this project. If you use OCamlBuild manually, you can compile the project from the command line by doing `ocamlbuild -lib nums main.native` The provided `Makefile` is appropriately configured.

```
Makefile          - builds main.native, also supports targets 'test' and 'zip'

util/assert.ml(i) - the assertion framework
gradedtests.ml    - graded test cases that we provide
main.ml           - the main test harness

x86/x86.mli      - the X86lite interface
x86/x86.ml       - the X86lite instruction set implementation

int64_overflow.ml(i) - library for working with int64 values

*simulator.ml     - where your interpreter and assembler code (Parts I and II) should go
*team.txt
```

Part I: The Simulator

X86lite assembly code is organized into labeled blocks of instructions, which might be written in concrete syntax as shown below.

```
        .text
fac:
        subq    $8, %rsp
        cmpq    $1, %rdi
        jle     exit
        movq    %rdi, (%rsp)
        decq    %rdi
        callq   fac
        imulq   (%rsp), %rax
        addq    $8, %rsp
        retq
exit:
        movq    $1, %rax
        addq    $8, %rsp
        retq
        .globl  main
main:
        movq    $5, %rdi
        callq   fac
        retq
```

This code has three blocks, labeled `fac`, `exit`, and `main`. The code at labels `fac` and `exit` implements a recursive version of the familiar factorial function. The code at `main` calls factorial with the immediate value 5.

In this part of the project you will implement a simulator for the X86lite platform, but rather than using the concrete syntax shown above, you will execute programs that have been converted to machine code and layed out in the memory of an idealized X86lite machine:

```
[| ...
   InsB0 (Subq,  [Imm (Lit 8L); Reg Rsp]);  InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Cmpq,  [Imm (Lit 1L); Reg Rdi]);  InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (J Le,  [Imm (Lit 72L)]);          InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Movq,  [Reg Rdi;      Ind2 Rsp]); InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Decq,  [Reg Rdi]);                InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Callq, [Imm (Lit 0L)]);           InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Imulq, [Ind2 Rsp;     Reg Rax]);  InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Addq,  [Imm (Lit 8L); Reg Rsp]);  InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Retq,  []);                       InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Movq,  [Imm (Lit 1L); Reg Rax]);  InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Addq,  [Imm (Lit 8L); Reg Rsp]);  InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Retq,  []);                       InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Movq,  [Imm (Lit 5L); Reg Rdi]);  InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Callq, [Imm (Lit 0L)]);           InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   InsB0 (Retq,  []);                       InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag; InsFrag
   ...
|]
```

This is just an OCaml array of `sbyte`s, "symbolic" bytes where `InsB0` represents the first byte of an instruction and seven subsequent `InsFrag`s represent the remaining seven bytes. While in a real machine each fragment would encode meaningful information about the instructions, this approach hides the details of a specific encoding and aids in debugging. The actual encoding of X86 instructions in particular is notoriously complicated, and as we mentioned in class, variable in length. We will assume a fixed-length, 8-byte encoding of X86lite for our simulator, representing instructions in memory as `sbyte`s. Fetching and decoding an instruction will simply involve reading the contents of its first byte, ignoring the following `InsFrag`s.

The OCaml datatype used for instructions is defined in the provided `x86.ml`, and the X86Lite Specification gives the full details about the meaning of each instruction.

Read (or at least skim) the X86Lite Specification now. You might want to correlate the various parts of the X86lite machine with the datatypes defined in `x86.ml`.

The X86lite specification is written from the point of view of actual X86 hardware, except for the behavior of labels, which are "resolved" by another program, the assembler (and linker/loader). Your simulator can assume that this has already been done, so instruction operands will not contain labels. In the memory image for the factorial example above, you can see that calls using the label `fac` and jumps using `exit` have been replaced with literal immediate operands `0L` and `72L`.

Our ML-level interpreter's representation of the X86lite machine state is given by the following type:

```
type flags = { mutable fo : bool
             ; mutable fs : bool
             ; mutable fz : bool
             }

type regs = quad array

type mem = sbyte array

type mach = { flags : flags
            ; regs : regs
            ; mem : mem
            }
```

The memory and register files are simulated by OCaml-level (mutable) arrays of `sbyte`s and `quad`s (OCaml 64-bit integers), respectively. The three condition flags are mutable boolean fields; all of the state is bundled together in a record (see IOC Chapter 8.1 for more about OCaml's record types). The main differences between the interpreter and the environment in which real X86 programs are executed include:

- **Memory:** Our simulator will use only 64K bytes of memory. The part of the heap simulated is the block of highest addressable memory locations -- in `simulator.ml`, this block is bounded from below by `mem_bot` and from above by `mem_top`. We will not model requesting memory from the operating system: you can assume the entire 64K address space has been paged in before execution of the program starts. We will also not model any of the restrictions on alignment or code layout related to memory paging.

- **Symbolic instruction encoding:** As described at the beginning of the section, we will assume a fixed-length, 8-byte instruction encoding by representing instructions symbolically in memory. The behavior of programs that read or manipulate `sbyte`s representing instructions as data is not specified. Yor simulator may raise an error or assume some default behavior: we will not test these cases.

- **Operand restrictions:** The X86Lite specification mentions several restrictions on the operands of various instructions. For example `leaq` can only take an indirect memory operand. Your simulator is not required to detect invalid operands, and may raise an exception or choose some convenient behavior. In other words, your simulator may implement a superset of the X86lite specification by executing instructions with invalid operands. We will only test your simulator with programs that conform to the restrictions in the specification.

- **Termination and system calls:** Normally, a program will terminate by notifying the operating system using a system call (e.g. `exit` on POSIX systems). We will not simulate system calls, so instead we use a sentinel address outside of our address space, `exit_addr`, to indicate that a program has terminated. The provided `run` function will call the `step` function until `%rip` contains `exit_addr`. To achieve this, you should begin execution with `exit_addr` on the top of the stack, so that executing `RETQ` without first pushing something else on the stack will terminate the program.

Provided Code

- `sbyte` serialization
- Machine state and X86 instruction datatypes
- The `Int64_overflow` module

Tasks

Complete the implementation in the `simulator.ml` file, some parts of which are given to you. We recommend that you do things in this order:

- First, as an exercise in condition codes, implement the `interp_cnd` function.
- Second, as another simple warm-up, implement the `map_addr` function, which maps X86lite addresses (represented as `quad` values) into `Some` OCaml array index (or `None` if the address is not in the legal address space).
- Third, implement the interpretation of operands (including indirect addresses), since this functionality will be needed for simulating instructions.
- Finally, implement the `step` function, which simulates the execution of a single instruction by modifying the machine state passed as an argument.

Hints:

- We have provided a module for performing 64-bit arithmetic with overflow detection. You may find this useful for setting the status flags.
- You'll probably want a function that sets the three condition flags after a result has been computed.
- Groups of instructions share common behavior -- for example, all of the arithmetic instructions are quite similar. You should

factor out the commonality as much as you can in order to keep your code clean.
- You will probably want to develop small test cases to try out the functionality of your interpreter. See `gradedtests.ml` for some examples of how to set up tests that can look at the final state of the machine.

Tests

We will grade this part of the assignment based on a suite of tests. Some of them are available for you to look at in `gradedtests.ml`, the rest of them we reserve for our own cases. We will also stress-test your interpreter on a number of "big" programs.
The provided `Makefile` can be used to run the test-suite via `make test`.
To help other teams debug their interpreters, you are encouraged to share "microbenchmark" test cases by posting them to the indicated thread on [Moodle](). These should be short (2-3 instruction) programs that test various functional aspects of the interpreter. We will not use these tests in our grading. You may add such test cases to the test suite defined in `studenttests.ml`

Part II: X86lite Assembler and Loader

Writing machine code directly is difficult and error-prone, even using our symbolic representation of instructions. The example factorial program in the previous section is written as a set of instructions for an assembler, a program that can automate much of the process for us. The primary functionality of the assembler for the purposes of this project includes the translation of human-readable mnemonics for instructions into machine code, and the translation of symbolic labels that appear in the assembly program into addresses understood by the machine.
Rather than working with a concrete syntax as in the above example, we will use the abstract syntax defined in `x86.ml`:

```
[ text "fac"
       [ Subq,  [~$8; ~%Rsp]
       ; Cmpq,  [~$1; ~%Rdi]
       ; J Le,  [~$$"exit"]
       ; Movq,  [~%Rdi; Ind2 Rsp]
       ; Decq,  [~%Rdi]
       ; Callq, [~$$"fac"]
       ; Imulq, [Ind2 Rsp; ~%Rax]
       ; Addq,  [~$8; ~%Rsp]
       ; Retq,  []
       ]
; text "exit"
       [ Movq,  [~$1; ~%Rax]
       ; Addq,  [~$8; ~%Rsp]
       ; Retq,  []
       ]
; gtext "main"
       [ Movq,  [~$n; ~%Rdi]
       ; Callq, [~$$"fac"]
       ; Retq,  []
       ]
]
```

As you can see, the correspondence between the abstract syntax and the concrete syntax is quite close. The file `x86.ml` and its corresponding interface `x86.mli` together provide the basic definitions for the creating and manipulating X86lite abstract syntax -- the main types you should be aware of are `lbl`, `reg`, `operand`, `cnd`, `ins`, and. Each of these corresponds fairly directly to a concept from the X86lite spec.

In addition to the instructions covered in the spec, X86lite assembly programs can contain label declarations and data consisting of either 64-bit words or zero-terminated strings. Each label declaration also has a visibility modifier, though these will only be used in later projects. X86lite assembly programs are represented using the following types defined in `x86.ml`:

```
type data = Asciz of string
          | Quad of imm

type asm = Text of ins list
         | Data of data list

type elem = { lbl: lbl; global: bool; asm: asm }

type prog = elem list
```

The `elem` type represents a section of an assembly program beginning with a label that contains either a list of instructions or a list of data. Each piece of data in a data section is a 64-bit value or a string. We can access the contents of each of these sections via offsets from their associated labels. X86lite assembly programs are lists of labeled `elem` blocks.

Your goal in this part of the assignment is to translate an `X86.prog` into an initial machine state that can be executed by your simulator. While this is not the simplest way to execute an X86lite program, it is meant to illustrate some of what the system assembler, linker, and loader will do to the assembly your compiler will generate in future projects.

This part of the project will involve serializing instructions and data into `sbyte`s, laying out the program in memory, resolving labels to addresses, and initializing the machine state. We can split this into two phases, assembling and loading. The assembler will do most of the work, outputting an executable that the loader will use to generate an initial machine state:

```
type exec = { entry    : quad
            ; text_pos : quad
            ; data_pos : quad
            ; text_seg : sbyte list
            ; data_seg : sbyte list
            }
```

An executable contains the following fields:

- **entry:** The entry point of the program, the address in memory of the first instruction executed.
- **text_pos, data_pos:** The address at which the following memory segments should be loaded.
- **text_seg, data_seg:** The assembled code for the text and data sections of the assembly program, with symbolic labels resolved.

Unlike an assembly program represented as an `X86.prog`, an object file has a single, contiguous segment of memory containing instructions and a single contiguous segment containing data. This is not strictly necessary to execute the program, but real systems often keep executable code in sections of memory that are guaranteed to be read-only by the virtual memory system for security purposes. Also, our executables contain neither declarations nor uses of labels. The provided functions to convert instructions and data to `sbyte`s guarantee this invariant.

Executable File Specification

We will require very specific output from your assembler and loader. Though programs may still execute correctly using other layouts, uniform outputs are necessary for testing purposes. The `text_seg` and `data_seg` fields of the executable should consist of the serialized contents of the `Text` and `Data` sections of the assembly program in the order that they appear, without any extra padding or extraneous `sbyte`s. Use the supplied `sbytes_of_ins` and `sbytes_of_data` functions. The `text_pos` field must be exactly 0x400000, the lowest addressable byte in the simulator. The `data_pos` field must contain the address immediately following the end of the text segment in memory. The `entry` field must contain the address of the first instruction after the label `"main"` in the assembly program.
The `assemble` function should raise an `Undefined_symbol` exception if it encounters a label that is not declared in the source program, or if `"main"` is not declared.

Loader Specification and Memory Layout

The `load` function should initialize a machine state given an executable file by copying the contents of `text_seg` and `data_seg` to the load addresses specified in `text_pos` and `data_pos`. It should initialize the instruction pointer to the address in `entry`, and the stack pointer to the highest legal memory address of our simulator. The contents of memory at the highest address should be the sentinel `exit_addr`.

Tasks

- Fill out the `assemble` function, which creates an `obj` given an `X86.prog`. First, calculate the address where text and data should be loaded according to the memory layout described above. Then, to resolve forward references to labels, you will need to traverse the assembly program and construct a **symbol table** to record the absolute address of each label definition you encounter. The last step is to traverse the assembly program a second time, outputting `sbyte`s for each instruction and data element you encounter. You will need to use your symbol table to replace labels, which can occur in instruction operands or `Quad` data, with their addresses.
- Fill out the `load` function, which creates an initial machine state given an object file. You will need to create an initial memory and copy the segments of the object file to their specified load addresses. You will also have to initialize the machine registers, setting `%rip` and `%rsp` appropriately. Lastly, you will need to initialize the stack to contain the sentinel `exit_addr` described in the previous section.

Grading

Projects that do not compile will receive no credit!
Your team's grade for this project will be based on:

- Implementing the X86lite simulator, assembler and loader in Parts I and II , graded via our test cases (including some withheld from the source we distributed).

hw2.zip 5 October 2021, 12:48 PM

Submission status

Submission status	No attempt
Grading status	Not graded
Due date	Thursday, 21 October 2021, 3:59 PM
Time remaining	11 days 16 hours
Last modified	-
Submission comments	Comments (0)

Add submission

You have not made a submission yet.

HW1 Overview + OcaIDE

Jump to...

Exercise 1 - OCaml Basics