Assignment 2: Simulering af x86 delmængde

CompSys 2017
Department of Computer Science
University of Copenhagen

Peter, Tobias & Xueying

1. October 2017

1 Compiling and running the code

1. Unzip `src.zip` in the same directory as `architecture-tools` (this is critical).
2. Navigate to the `src` directory.
3. Type the command: `bash test.sh` in a UNIX-like command line. The test-script will automatically build `architecture-tools/asm`, `architecture-tools/sim`, `architecture-tools/trace` and `src/sim`. The script will then build and test the various `.x64` located in `src/tests`. Finally it will print the number of errors encountered.

2 Implementation

The general idea of this program is to simulate a x86 assembler. In order to recognise these different instructions in x86 machine code, we used different hex code as signal that indicates if it catches any instruction in the variable `major_op`. How the program works will be explained in greater detail in the coming sections.

2.1 NOP and HLT instruction

The NOP, HLT and (movq %a,%b) instruction was given to us, these instructions is however still being tested by our tests.

2.2 Move Instructions

Move instructions has four different kinds. As mentioned, the hex code is used as signal, once there is a match found in the input file, the program will send a corresponding signal further which empowers the program to write to memory or register, which is important since the move instructions function as the basic communications among register, memory and immediate bytes. The size of instructions varies from kind to kind, hence we sort instructions into three kinds: one using memory, one using registers, and the one using

none. By doing this, the input file can be correctly read after the current line. When the move instruction is executed then its result will be passed onto `datapath_result`.

2.3 Conditional Move Instructions

The conditional move instructions function as the normal move instructions when the given condition is fulfilled, they work otherwise like NOP instruction. The variable `minor_op` stores the type of operation that it takes then the program will use `eval_condition` function which takes this operation and the flag registers to verify the status of the condition.

2.4 Arithmetic Instructions

Arithmetic instructions enable the program to manipulate data. It can not only do mathematical adding and subtracting, but also bit-wise operations like `and` and `xor`. The variable `minor_op` stores the type of operation as earlier described, which will be later executed by `alu_execute` function on the data and pass to `datapath_result`. There is one exception among all the arithmetic instructions: compare instruction, being that it will write to the registers.

2.5 Jump Instructions

To perform any jump instruction, it requests a destination which is stored in immediate bytes. After extracting this information from immediate bytes, the pointer that points to next line will be assigned to the destination.
There are also conditional jump instructions that take a condition and will only performs when the condition is fulfilled. They are very similar to the previous conditional move instruction.

2.6 `CALL` & `RET`

With the `CALL` and `RET` instructions, we will be able to perform function calls. `CALL` decrements the stack pointer and saves the program counter to memory, before setting it to the program count of the function that has been called. Conversely, `RET` increments the stack pointer and retrieves the saved program counter from memory in order to continue reading from where the function was called.

2.7 `PUSH` & `POP`

In order to push the contents of a registry to the stack, the stack pointer is first decremented. The registry-value being pushed onto the stack is then saved to memory at the address indicated by the decremented stack pointer. On the other side, `POP` reads the value stored in memory at the address indicated by the stack pointer and stores this value in a specified registry. In addition to this, the stack pointer is incremented.

3 Testing

3.1 Important Notice:

Since our team is made up by people using different operating systems, the test script will give an error message and fail to generate the .out and .trace files. This is caused by the different `"end of line sequence"` changing the test script and test files `"end of line sequence"` to LF instead of CRLF will fix this problem.

3.2 Test script

To easily test our implementation, we have created a test script. The script will build all dependencies (such as the reference simulator and our own simulator), assemble all test-programs, generate .trace-files and run these with our simulator. Finally, the script will report back the number of failed tests. We have attempted to include tests for all possible instructions.

3.3 Failed tests

Late in our testing we managed to find two bugs related to `movq %a,i(%b)` and `movq i(%b),%a`. Due to the time constraints, we have not managed to fix these bugs, and therefore there will be times where these two instructions fail. Scenarios where these errors occur are included in our test.