A2: Simulering af x86 delmængde

Bjørn Christian Vedel Bennetsen (qfr834) &
Kalle Kromann (zmr905)

1. Oktober 2017

1 Compilation and testing

The program can be compiled using make inside the `src/` folder. It will however also be compiled when running the tests using the bash scripts, if the binary is not already up to date. We have written test-program for each of the commands in the given `x86`-subset. To make things easier for us, we have written some bash-scripts to run and tests our simulation. The shell scripts `run_sim.sh` and `test_all.sh` can be executed inside the `src`-dir that excluding the simulator files contains `test_programs/`, `test_all.sh` and `architecture_tools/`.
`run_sim.sh` takes a `x86`-program without the extension and the path to the `architecture_tools/` directory.
It assembles the program using the given assembler in `architecture_tools/asm`, to produce a `.o` executable. Then it executes the compiled executable with the reference simulator in `architecture_tools/sim` along with a name for a trace-file. We then run the test-program using our own simulator along with the trace-file produced by the reference simulator and thus validate our `x86`-program by checking register and memory write for each cycle of the simulation.
The validation happens with the methods `validate_reg_wr` and `validate_mem_wr` that on an unsuccessful validation, returns a **"validation error"**. If the all the validations was successful the simulation exits, and prints **"Done"**. We use this way of validating our programs in the bash-script **test_all.sh**. We loop over all the test programs and runs them with `run_sim.sh`, then we pipe the output into the UNIX-built-in `egrep` with

the argument "Done", then further piping the output into `wc -l` and then, 0 is printed if the validation did not complete along with a **"validation error"**. and 1 is printed if the test was successful.

We decided to make at least 1 test-program for each of the commands, which purpose is to test that the command is functional. All the test-programs can be found in `test_programs/`.

Some of the test-programs also uses other commands, thus they also test these. This can also be useful to indicate what command could be functioning wrong, if two tests programs contains the same command, and fails.

All of our test-programs gets validates successful, except the program:
`test_programs/opq/opq_cmpq_1_neg.x86`, which purpose is to test the comparison-command `cmpq %ra %rb`, when a negative value is in `%rb`. The comparison should set the `cc.sf`-flag to true, which is what our simulator is doing, but not what the reference simulator does, therefore the test fails and a trace validation error is printed. We have seen an issue on the `compsys-e2017-sim/`-repo which addressed this problem of the reference simulator.

2 Design and implementation

The solution is pretty straightforward and follows the example datapath diagram pretty closely, with a few differences. We have split most of our multiplexors (the `or(val, val)` functions) into separate statements, assigning the same variable multiple times to improve readability of the code.

When we designed the program we designed one function at a time, making new variables as they were needed, or reassigned already existing variables if that was needed. We use a lot of different booleans as control-bits for the multiplexors. We also do a lot of boolean algebra, to allow multiple control bits for one multiplexor. For example Push and Call both uses the operand a as -8, so we use `is_Push || is_Call` as the control bit.

We use the convention that the variables reg_a and reg_b are source and destination registers respectively. That means we must flip the reading of the registers if the instruction

is `movq D(%rB),%ra` because it is the only instruction where the encoding is this way. Operand a is used as the datapath result, and can be either what reg_a contains, immediate values or what is used in the ALU as the first operand. Operand b is also used in the ALU as the second operand, and it's value is also used if there is need to write to memory with an address in reg_b.

We use the supplied ALU function `alu_execute` for our ALU operations. This function returns both the result of the arithmetic and the needed condition codes. Because the ALU runs every clock and therefore will return random condition codes every time we use a simple macro to parse the condition codes, only updating the existing codes if the instruction was in fact an arithmetic operation. The macro just evaluates the following boolean statement:
`(is_arith && result) || ((is_arith^cur) && !is_arith)`
where `is_arith` is whether or not it is an arithmetic operation, `result` is the new condition code and `cur` is the current condition code. The xor part ensures that if the instruction is arithmetic but the result was false, and the current code is true the new condition code should be false. We make sure it doesn't work the other way (code is false, but is_arith is true, should be false to keep the current condition code) by using the !is_arith in the end. We also use the ALU using the supplied add() function a couple of places.

The stack is handled in such a way that we assume register 4, %rsp, contains the stack pointer, so it is up to the programmer/codegenerator to initialize the stack before using it, by setting the register to the highest address in memory, e.g. 1024 in our case. When pushing to the stack we write to the memory of the current stack pointer minus 8, because the data is written towards the end of memory. The stack uses fixed size elements of size 8, because it writes immediates of type uint64_t which are 8 bytes long and can be negative, so we need every byte to represent the sign. When popping from the stack we just increment the stack pointer by 8, and don't overwrite the existing memory with zeros or something, as it will just be overwritten when it is needed again.