Compsys - A5

Christian R (srw812), Nicolai N (dwx635) og Andreas J (cmn945)

December 22, 2021

Theoretical

Long or pointer

- rsp: Pointer. Used to allocate bytes in the stack
- r11: Pointer. Stores the return address.
- rdx: Long
- rax: Long
- rsi: Long
- rcx: Pointer
- rdi: Long
- r8: Long

Call and return

There is no function-calls since there is no "call" keyword in either of the functions. However, both functions have a return-statement : "ret %r11".

Program progress

Function LFB0:
The function starts with a procedure instruction, since it allocates 8 bytes in the stack. Then it encounters a condition, if the conditions holds it will come by the return instruction. If the condition does not hold, it loops until the condition is true.

Function LFB1:
The function starts with a procedure instruction like the first function. It then does some arithmetic by adding two numbers where one of the numbers is also scaled with 8. It then jumps to a condition, where if it holds it will go the return instruction, otherwise it will go to another condition. If the next condition does not hold, it will add 8 to three numbers and then continue.

Conditions

There are three conditions

- cble %rdx, %rax, .L4
- cbbe %r8, %rdi, .L10
- cbge %rax, %rcx, .L7

cble translates to <= for signed values, cbbe translates to <= for unsigned values, and cbge translates to >= for signed values. The first condition compares to longs, the second condition compares either two positive longs or two addresses, the third compares two longs.

C-code

```
int LFB0{
    {long x, long y} *rdx;
    long i = 0;
    long retval = 0;
    for (i; i <= rdx){
        retval = x+i*8;
        storeval = retval;
        retval = y+i*8;
        i++;
    }
    return retval
}

int LFB1{
    long x,y,z;
    long a,b,c;
    long retval = 0;
    long r8;
    r8 = x + b * 8;
    if (r8<= x){
        return retval;
    }
    else{
        *y = b;
        *z = c;
        if (b>=c){
            c = *x;
        }
        else{
            b = *x;
            x += 8;
            y += 8;
            z += 8;
        }
    }
}
```

Report

Running the code

The code can be run by first navigating to the src/ directory and running "make" in the terminal. Then our simulator can be run with "./sim hex-file startpoint [tracefile]". hex-files are generated from any valid .prime file using prasm, startpoint is the address of the first instruction point wished to be executed in byte-form. If wishing to compare against a tracefile, that can be generated using "prun hex-file startpoint [-tracefile TRACE]"
to run our tests you stay in the src directory and run the test script test.sh, ie "bash test.sh" on linux, here it is important to note that it may not work out of the box.
I have my prasm and prun appended to my system path and they work fine in the terminal by themselves, however for some reason the test script does not like this and cannot find them, so we had to resort adding the entire path of where those executables are located in my machine, which is unlikely to be the same as on your machine, and so you will have to add your own location to the script. As a less intrusive alternative we have decided to include the corresponding .hex, .sym, .trc and .out files, inside the test_runs directory, generated from running the test script ourselves and these can be used to manually view our results if wished. We will also go into depth about our tests and their results in the Testing section later

Implementation

For this assignment we had 3 sub tasks which needed to be solved for the simulator.
Firstly we had to figure out the size of the given instruction, for this the simulator had already done much of the heavy-lifting in figuring out which type of instruction has been received through a series of bools, which compare the major opcode (first 4 bits) against a lookup table of possible values
By definition we know that only 1 of these bools will be true and therefore we figured out the length using the "use_if" function from arithmetic.h, which would only return the given val for the true bool and a val of 0 for everything else. This allowed us to add together all the values returned from our series of use_if calls and then we'd essentially be adding together a bunch of 0's and one non-zero val, which would be the length of our instruction
We also for this had to figure out the byte length of each possible instruction. This was done with the lookup table in encoding.txt. any 8 bit long part counts as 1 byte (including a,d,s,c,z,v as bits as that is what they will be in the real encoding), and pp...32...pp/ii...32...ii is 32 bit long, which counts as 4 bytes. Adding that together for each instruction we now had

the byte length for each potential instruction

our second task was to make the instructions which accessed the memory be decoded properly, this included load, store and conditional jumps
We were given the bools is_load, is_store and is_conditional which needed to be made to properly calculate whether they needed to be true or not
There are 2 instructions which load from memory: movq (s), d and movq i(s), d
and 2 that store in memory: movq d, (s) and movq d, i(s)
We already have 2 different bools to figure out whether these are our potential instructions: is_reg_movq_mem for movq (s), d and movq d, (s), as well as is_imm_movq_mem for movq i(s), d and movq d, i(s)
We then match these against their minor op codes using boolean and which will return true if both the major opcode and minor opcode were matched and end as true through a boolean or on the 2 matching results for load and store respectively.

For checking for a conditional jump we have 2 potential instructions we need to check for: cb<c> s,d,p and cb<c> $i,d,p
for checking if it's the first one we need 3 pieces of information, since contional jumps are not as simple with their minor opcode as the others since it can be many potential values depending on which conditional we want to use, and so the easier solution is to simply check that it is not any of the other instructions with the same major opcode. This would be call and jmp. To assist in this we added the bool is_jump which matches if the major opcode and minor opcode match what we would expect for the jmp instruction. Now we have bools checking if the instruction is call or jmp, and so we can match the boolean nots of these against is_cflow with boolean ands, then match this with is_imm_cbranch with boolean or so we will in the end get a true if it is either of those instructions

For the third task we had to implement instructions which change the flow of the program (jmp, ret, call and cbcc), however we have run out of time to do this task and so it is not implemented

Testing

For testing we have a test script test.sh, which compares the simulation of the .prime files in src/tests against the same files run with prun
We have 5 primary test files
test_insn_len.prime tests tests whether we decode the instruction length properly. This is done by adding instructions that match each of the bools which we use to check the length. The file is not meant to be a program that actually does anything specific, the only thing we made sure to do was, make sure that instructions that can mess with the flow of the program, do not actually do this as they are not implemented and the test

returns success
test_load.prime tests whether the 2 load instructions decode properly by simply having a file that runs both instructions and it returns success
test_store.prime does the same as test_load.prime except we also add 2 values we load into the registers, so we don't just test loading in whatever we happen to find in the registers and could test with different values being loaded in, returns success
test_cflow_1.prime tests reg->reg conditional. it places 0 and 3 in rax and rdx respectively then runs a loop where it adds 1 to rax and checks if rax is less than rdx, if it is then it loops back, adds 1 and tries again until it is greater than or equal before coming to a stop. This does not return success, which we assume to be because we have not implemented the third part of the assignment, which is supposed to properly implement conditionals.
test_cflow_2.prime tests imm->reg conditional. it places 0 in rax, then runs the loop where it adds 1 to rax then checks if 3 is greater than or equal to rax, if it is then it loops back, adds 1 and tries again until it returns false. Again it does not return success, which is likely the same reason as above
test.prime does not formally test anything, but is a copy of a program given in one of the lectures, which we used throughout the implementation to test before we had been given tests and we've decided just to leave it there for good measure since it also returns success, so it doesn't hurt.