Assignment 1

Computer Systemer
Department of Computer Science
University of Copenhagen

Pelle Rubin Galløe <hqr241@alumni.ku.dk>,
Mads B. Christensen <rsv422@alumni.ku.dk>,
Julian S. W. Wulff <bkx655@alumni.ku.dk>

Version 1;
Due: Sunday, September 22

1 Introduction

This is a technical report for Computer Systemer B5-6, 2019. In this project we have been assigned to complete a simulator for a minor instruction set, defined over a the language `x86prime`, which is more or less is a subset of the x86_64 assembler language. Furthermore the report contains the solution to a theoretical assignment regarding the simulation of a cache. The target audience for this report is the teachers assistants of the course. The solution is developed in `C`, with a set of restrictions defined in the assignment.

The solution ships with a `Makefile` such that the finished simulator compiles by running the following command in the terminal at the location of the code.

```
$ make sim
```

When the code has compiled the simulator should be able to simulate running a `x86prime` files. In order to simulates this, the prime file has to be encoded into hexadecimal format which can be done with the program *prasm* by executing the following command (*prasm* should be installed and added to `PATH`):

```
$ prasm file_name.prime
```

Now the hexadecimal formatted *x86prime* program can be simulated. This is done be running the following command where 0 is the starting address for where the program should start:

```
$ ./sim file_name.hex 0
```

In order to check the correctness of the simulator it is possible to run the simulator with a generated trace fileby adding it as an third argument to the above command. This trace file is generated with the program *prun*which simulates *x86prime* programs. The trace file is generated by running the following command (*prun* should be installed and added to `PATH`):

```
$ make prun file_name.hex start_label -tracefile file_name.trc
```

The process of testing the correctness of the simulator is automated and can be done by running the `test.sh` script, that tests all the files in the tests folder. Further information and discussion about the tests can be found in section 3.

2 Implementation

In the implementation of the simulator we have tried to follow the proposed micro architecture of the simple one cycle processor from the assignment paper, using the functions serving as all the building blocks of the processor such as the ALU and address generator.

The simulator consist of some global state variables that are set when the program is run and preserved between cycles. Most relevant variables includes the registers, the memory and the instruction pointer for our simulator. After these variable are set the simulator continues with a while loop which does most of the simulation, and runs until the simulated program is set to terminate by the return (ret) instruction. Each of the loops reads one line of the *x86prime* instruction, hereby performing one instruction per loop.

Other than the before mentioned parts of the program the simulator also includes a long list of macros which are used to map the encoding of *x86prime*. A selection of these macros can be seen in listing 1 as an example. In order to finish the simulator we had to implement the macros that encodes the calculations for the `LEAQ` and `MOVQ` instructions.

```
// Major op codes
#define RETURN 0x0
#define REG_ARITHMETIC 0x1
...
#define CALL 0xE

// minor opcodes
#define COPY 0x1
#define REG_SCALE 0x2
    ...
#define REG_IMM_MEM_COPY 0xD
```

Listing 1: Selection of macros

Each time the while loop runs, the program starts by reading the instruction at a given address returned either by the IP selector or as the starting address when the simulator is run. Each of the instructions is then read from the memory as 10 bytes, which are the maximal instruction size. The instruction is then send to the decoder. We finished the implementation of the decoder pretty straight forward by adding Boolean variables acting as control signals and the datapaths such the target address. The control signals ranged from signals determining type of instruction, whether the instruction should read or write either to memory, write to a register, determining instruction size, determining what to include in address generation and lastly reading immediate values and target address values. A selection of the control signals and their implementation is listed in listing 2.

```
/*** DECODE ***/
val reg_z = pick_bits(4, 4, inst_bytes[2]);

// Decode memory load and store
bool is_load = (is(COPY, minor_op) || is(IMM_ADD_REG, minor_op)) &&
               (is_reg_movq_mem || is_imm_movq_mem);

// Decode address generation operations
bool use_z = is_leaq3 || is_leaq7;

// determine instruction size
bool size2 = is_return || is_reg_arithmetic || is_reg_movq || is_reg_movq_mem ||
             is_leaq2;

val ins_size =
    or (or (use_if(size2, from_int(2)), use_if(size3, from_int(3))),
        or (use_if(size6, from_int(6)),
            or (use_if(size7, from_int(7)), use_if(size10, from_int(10)))));

bool use_imm = is_imm_arithmetic || is_imm_movq || is_imm_movq_mem ||
               is_leaq6 || is_leaq7 || is_imm_cbranch;

bool reg_wr_enable = is_reg_movq || is_reg_arithmetic || is_imm_arithmetic ||
                     is_imm_movq || is_load || is_leaq2 || is_leaq3 ||
                     is_leaq6 || is_leaq7;
```

Listing 2: Selection of decoder parts

After implementing the decoder we had to connect the ALU and address generator with the decoder in order to let the simulator perform arithmetic instructions such as `ADDQ`, `SUBQ` and address generation for the differtent `LEAQ` instructions. This was done pretty straight forward be using the the included address generator block and ALU block. With this connect it was necessary to implement the result selector such that it only choose the result returned specified by the decoder. A selection of how the functions of the above mentioned was implemented can be seen in the listing 3.

```
// perform arithmic calculations
val arithmetic_result = alu_execute(minor_op, reg_out_a, op_b);

// Address generator
val agen = address_generate(reg_out_z, reg_out_b, sext_imm_i, op_v, use_z,
                            use_s, use_imm);

/*** RESULT SELECT ***/
val datapath_result =
    or
    (use_if(is_reg_movq || is_imm_movq, op_b),
     or (use_if(is_load, mem_out),
         or (use_if(is_imm_arithmetic || is_reg_arithmetic, arithmetic_result),
             use_if(is_leaq2 || is_leaq3 || is_leaq6 || is_leaq7,
                    agen_result))));
```

Listing 3: Selection of result selector implementation functions

In order to implement control flow instructions such as JMP, CALL and conditional jumps, we had to connect the decoder to the comparator block and the comparator block to the ip selector. This was achieved with the functions shown in listing 4

Lastly we connected the decoder to the memory enabling it to read and write to the memory.

3 Testing

In order to check the correctness of our simulator we have implemented a dozen of tests. These tests include one prime program for each of the instructions in the *x86prime* instruction set. Furthermore the tests include a couple of longer prime programs with a mix of instructions serving as simulations of real world programs, fx some programs to calculate the Fibonacci number in different ways. In our individual tests for each of the instructions we made sure to use the MOVQ instruction with an immediate value in order to fill the registers with values. This ensures that our tests returns truly. If the tests was not performed with values in the registers this could mean the the test with return with false positives.

```
// address of succeeding instruction in memory
val pc_incremented = add(pc, ins_size);

// Perform CFLOW comparision
bool cb = comparator(minor_op, reg_out_a, op_b);

// IP select
val pc_next =
    or (or (use_if(is_jmp, address_p),
            use_if((is_cflow || is_imm_cbranch) && cb, address_p)),
        or (use_if(is_return, reg_out_b),
            use_if(is_reg_arithmetic || is_reg_movq || is_reg_movq_mem ||
                       is_imm_arithmetic || is_imm_movq || is_imm_movq_mem ||
                       is_leaq2 || is_leaq3 || is_leaq6 || is_leaq7,
                   pc_incremented)));
```

Listing 4: Implementation of the IP selector

When all of our tests are run using the script mentioned in the introduction section we get all our test back with success.

4 Theoretical assignment - simulation of cache

In this assignment we where asked to answer a couple of questions regarding simulation of a cache. In following subsections we will answer these questions

4.1 Question 1

From the assignment text we a given a machine that is byte-addressed with 32-bit addresses. From this we conclude that the number of physical address bits in the main memory (m) is 32. Furthermore we are given that the machine is equipped with a 2-way associative cache with the size of 32 kilobyte, and that the cache's block size is 16 bytes. From this we conclude that the number of lines per set in the cache (E) is 2, since the cache is 2-way associative. With this information we filled out the table shown in table 4 and calculated how many bits are used for the *set index*, *block offset* and *tag bits*. By looking at table 4 in the appendix, we see that the 18 first bits are used as tag bits, the next 10 bits are used as set index and the last 4 bits are used for the block offset.

4.2 Question 2

Since no replacement strategy has been defined in the assignment sheet we will in the following question use *least recently used* as our replacement strategy. Given a cold cache, the following data stream:

0x0, 0xF00000, 0x0, 0xF0000C, 0xC00004, 0xE00008, 0xF00004, 0xC00000, 0x8, 0x10, 0x4.

Would result in:
miss, miss, hit, hit, miss, miss, miss, miss, miss, miss, hit. A table of how we got to this point can be seen in table 4 in the appendix.

4.3 Question 3

Given a secondary 4-way assosiative cache with the size of 256 kilobyte and a block size of 32 bytes, the following data stream:

0x0, 0xF00000, 0x0, 0xF0000C, 0xC00004, 0xE00008, 0xF00004, 0xC00000, 0x8, 0x10, 0x4, 0x, 0x0, 0xF00000, 0x0, 0xF0000C, 0xC00004, 0xE00008, 0xF00004, 0xC00000, 0x8, 0x10, 0x4.

Given a cold cache would result in:
miss, miss, hit, hit, miss, miss, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit, hit.

As with question 1 and 2 tables for the secondary cache's parameters and the data stream can be found in the appendix

4.4 Question 4

Given a miss-rate of 20% and a miss penalty on 10 clock cycles for the primary cache, and a miss-rate of 30% and some specified hit time we can calculate the average memory access time (AMAT). Since no hit time is specified in the assignment paper we assume that the hit time for the L1 cache is 4 clock cycles and the hit time for the L2 cache is 10 clock cycles.

We calculate the AMAT with the following function

$$AMAT = hit\ time\ L1 + miss\ rate\ L1 \cdot (hit\ time\ L2 + miss\ rate\ L2 \cdot miss\ penalty\ L2)$$
$$AMAT = 4 + 0.2 \cdot (10 + 0.3 \cdot 50) = 9 \text{ clock cycles}$$

5 Conclusion

We have finished the implementation of the *x86prime* simulator, which is now able to perform all of the instructions in the *x86prime* instruction set. To the best of our knowledge the correctness of the simulator is as it should be, which the results of our tests also point towards. In our implementation we have tried to follow the suggested micro architecture and only used built-in components which came as part of the simulator.

6 Appendix

Parameter	Description
Given parameters	
$E = 2$	Number of lines per set
$B = 16$	Block size (bytes)
$m = 32$	Number of physical (main memory) address bits
$C = 32768$	Cache size (bytes)
Derived quantities	
$S = \frac{C}{B \cdot E} = \frac{32768}{16 \cdot 2} = 1024$	Number of sets
$s = \log_2(S) = 10$	Number of set index bits
$b = \log_2(B) = 4$	Number of block offset bits
$t = m - (s + b) = 18$	Number of tag bits

Table 1: Cache parameters

	t = 18	s = 10	b = 4	h/m
0x0	0000000000	0000000000	0000	miss
0xF00000	1111000000	0000000000	0000	miss
0x0	0000000000	0000000000	0000	hit
0xF0000C	1111000000	0000000000	1100	hit
0xC00004	1100000000	0000000000	0100	miss
0xE00008	1110000000	0000000000	1000	miss
0xF00004	1111000000	0000000000	0100	miss
0xC00000	1100000000	0000000000	0000	miss
0x8	0000000000	0000000000	1000	miss
0x10	0000000000	0000000001	0000	miss
0x4	0000000000	0000000000	0100	hit

Table 2

Parameter	Description
Given parameters	
$E = 4$	Number of lines per set
$B = 32$	Block size (bytes)
$m = 32$	Number of physical (main memory) address bits
$C = 262144$	Cache size (bytes)
Derived quantities	
$S = \frac{C}{B \cdot E} = \frac{262144}{32 \cdot 4} = 2048$	Number of sets
$s = \log_2(S) = 11$	Number of set index bits
$b = \log_2(B) = 5$	Number of block offset bits
$t = m - (s + b) = 16$	Number of tag bits

Table 3: Secondary cache parameters

	t = 16	s = 11	b = 5	h/m
0x0	00000000	00000000000	00000	miss
0xF00000	11110000	00000000000	00000	miss
0x0	00000000	00000000000	00000	hit
0xF0000C	11110000	00000000000	01100	hit
0xC00004	11000000	00000000000	00100	miss
0xE00008	11100000	00000000000	01000	miss
0xF00004	11110000	00000000000	00100	hit
0xC00000	11000000	00000000000	00000	hit
0x8	00000000	00000000000	01000	hit
0x10	00000000	00000000000	10000	hit
0x4	00000000	00000000000	00100	hit
0x0	00000000	00000000000	00000	hit
0x0	00000000	00000000000	00000	hit
0xF00000	11110000	00000000000	00000	hit
0x0	00000000	00000000000	00000	hit
0xF0000C	11110000	00000000000	01100	hit
0xC00004	11000000	00000000000	00100	hit
0xE00008	11100000	00000000000	01000	hit
0xF00004	11110000	00000000000	00100	hit
0xC00000	11000000	00000000000	00000	hit
0x8	00000000	00000000000	01000	hit
0x10	00000000	00000000000	10000	hit
0x4	00000000	00000000000	00100	hit

Table 4