Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

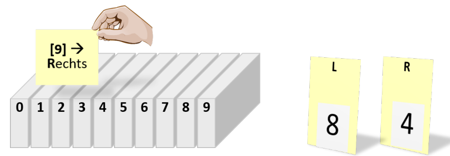
Übungen zur Vorlesung Informatik (D-MATH / D-PHYS) HS 2016
Dozent: B. Gärtner
http://lec.inf.ethz.ch/ifmp/2016

Bonus Exercise **22.11. – 6.12.2016 (23.59)**

Bonus Exercise: Do-It-Yourself Stack Processor Simulator.

Introduction

In the first lecture of this course, it was demonstrated – using cardboard boxes and paper sheets – how a computer executes a program. A set of paper sheets carrying the instructions were created (assembled). The paper sheets (instructions and data) were stored in cardboard boxes (memory). In multiple turns, instructions were fetched from a box, read (decoded), and executed. Temporary results were displayed on two hands. This has been repeated until the program was finished and the result was available.



In this project, you will automate this process by writing a C++ program for simulating a fully-fledged virtual CPU (Central Processing Unit, or simply processor) that is able to execute programs in machine language. In the following, we refer to this virtual stack-based processor[1] as StackCPU$_{16}$. You will understand how a processor works, what machine language is, and how machine language can be displayed in human readable assembler format.

We have split the work into two subtasks.

(a) **Disassembly:** your program receives instructions in hexadecimal form and translates them into a human readable form. This part helps you to understand the machine instructions.

(b) **Simulation:** your program simulates a system consisting of a processor, memory, and in- and output. This simulated system executes the machine instructions by applying their effect.

Although the long and technical descriptions in parts of this project may look frightening at first, be assured that it is not as difficult as it may appear. We believe that this project is fun and hope you enjoy working on it!

Before starting, **please read carefully the following guidelines regarding this bonus exercise:**

- This is a graded exercise, and the programs are graded automatically.
- Maximal reachable points are 100. You can hand-in multiple versions, the submission with the highest score counts. If you get n points, you get a bonus of $\frac{1}{8} \cdot \frac{n}{100}$ of a grade.
- Cheating is not rewarded: if you tune your submission to the provided test inputs, you will not receive all points. We will run your program with hidden test inputs as well.
- Hand-in using the online submission system, no other forms of hand-ins will be accepted.
- Start early. If you have a problem on the day of submission, we may not be able to solve it.
- Before sending a mail with a question to this exercise, check the course homepage for most recent information.
- The ETH Disciplinary Code applies to this bonus exercise as it constitutes part of your final grade. It is not allowed to share any (hand)written or electronic (partial) solutions with any of your fellow students. We are obligated to report any violations of the Code. The only exception we make is that we encourage you to verbally discuss the task with your fellow students.
- We reserve the right to invite you to an oral examination. This can be triggered randomly or by a similarity check and thus does not necessarily imply suspicion.

ETH Codeboard link: https://codeboard.ethz.ch/ifmp16Bonus

[1] The Java Virtual Machine and Microsoft's Common Language Runtime are prominent examples of stack-based virtual architectures.

Note: There are gaps in the specification that lead to undefined behavior for incorrect machine language input (an illegal instruction, a division by zero, overflow or underflow, memory out of bounds,...). You may assume that the machine language input is correct, and all our test inputs satisfy this assumption. Still, try to protect against errors in the input, or in your own code, for example by using assertions!

(a) Disassembly

A program is provided at the standard input as a decimal integer count n followed by a series of n instructions encoded as *hexadecimal* numbers. Disassembly is the translation of a stream of instruction values into a human-readable format. The reverse is called assembly.

Remark: Reading an unsigned integer `i` in hexadecimal form from standard input can be accomplished with `std::cin >> std::hex >> i`. Once switched to hex mode, the input stream stays in this mode until it is switched back to decimal mode with `std::cin >> std::dec`. In case of doubt, it is a good idea to simply switch each time you read. The same is true for output. Writing an unsigned integer `x` in hexadecimal form to the standard output can be accomplished with `std::cout << std::hex << x` (this does not write the leading `0x`).

In order to disassemble instructions, they must be decoded first. Decoding means splitting the 32-bit values into its components: opcode (8 bits) and operand (24 bits). The instruction layout is described in Appendix A.3. An opcode uniquely identifies an instruction. The opcodes of StackCPU$_{16}$ are summarized in Section A.5 of the Technical Specification on page 6. The meaning of the operand varies between different instructions.

Decoding Examples:
The instruction value `0x21000000` should be decoded into opcode=`0x21`=33 and operand=0 (corresponding to instruction `sub`). The instruction value `0x3200010A` should be decoded into opcode=`0x32`=50, operand = `0x10A`=266 (corresponding to instruction `const 266`).

Decode and disassemble instructions read from standard input.

- When reading the command `disassemble` from standard input, enable disassembly mode.
- Read the number of instructions n.
- Output each of the n instructions in the format `<mnemonic> <operand>` followed by a newline character. `<mnemonic>` should be replaced by the name (such as `add` or `jmp`) of the opcode encoded by the first 8 bits of the instruction. Depending on the opcode, `<operand>` can be empty. Otherwise, `<operand>` should be replaced by the decimal signed integer value encoded by the last 24 bits of the instruction. Provide spaces between name and operand.
- For illegal opcodes, i.e. opcodes that are not part of the instruction set, output `data` followed by the hexadecimal instruction value.
- Output `end` at the end of the disassembly.

Examples:

input	output	input	output	input	output
disassemble		disassemble		disassemble	
5		10		10	
0x3200000f	const 15	0x32000000	const 0	0x320000ff	const 255
0x12000000	out	0x32000001	const 1	0x32000100	const 256
0x3200000a	const 10	0x20000000	add	0x31000000	store
0x13000000	outchar	0x26000000	dup	0x32000100	const 256
0x1000000	hlt	0x26000000	dup	0x30000000	load
	end	0x12000000	out	0x12000000	out
		0x32000009	const 9	0x3200000a	const 10
		0x41000009	jeq 9	0x13000000	outchar
		0x40000001	jmp 1	0x1000000	hlt
		0x1000000	hlt	0xff	data 0xff
			end		end

(b) Simulation

Implement the processor that executes the machine instructions. Figure 1 depicts on a high level how your processor processes machine instructions. The processing of an instruction is divided into three phases (remember the cardboard example) and repeats until the device is instructed to halt:

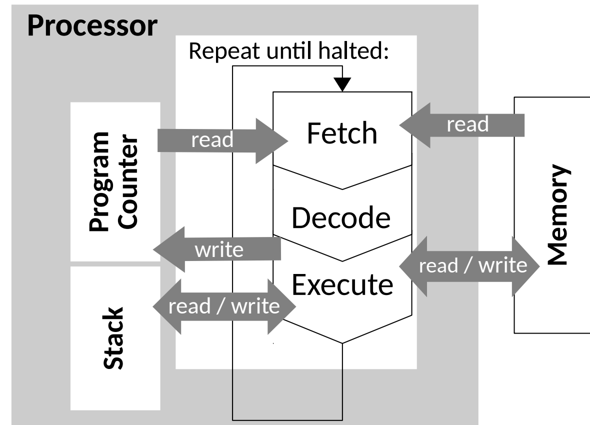


Figure 1: Processor execution instructions

Fetch: The instruction to be executed is loaded from memory. The program counter *PC* holds the location,i.e. the memory address, of the instruction to be executed next. Upon start up of the processor the PC must be set to the address of the first instruction of a program. This is always `0`.

Decode: The instruction fetched in the previous phase is decoded into its components opcode and operand. An opcode uniquely identifies an instruction and its format, i.e. number and type of operands. The memory layout of a machine instruction is presented in Appendix A.3.

Execute: Each instruction has effects. The execution of an instruction means to apply these effects. In our processor, the effect can be a modification of memory content, the modification of the operand stack, an input, or output. Moreover, each instruction modifies the PC: With the execution of an instruction the PC is either set to the next instruction or, in the case of a branch instruction, to the branch target if the branch is taken.

Your program of this sub task must implement these phases with the following algorithm

1. When reading the command `simulate` from standard input, enable simulation mode.
2. Store the machine instructions present at standard input to StackCPU_{16} memory consecutively starting at address `0`.
3. Set the program counter (`PC`) to address `0`.
4. Fetch instruction at `PC` from memory.
5. Decode the instruction.
6. Execute the instruction: apply its effect, as specified in Appendix A.5.
7. If not halted goto 4, exit the processor simulator program otherwise.

Address	Instruction	Commented Disassembly			Meaning in C++
	16				
0	0x3200000f	const	15	; load i for ...	
1	0x30000000	load		; ... decrement	
2	0x26000000	dup		; ... output	
3	0x26000000	dup		; ... test	int i = 10;
4	0x32000000	const	0	; if i == 0	while (i != 0) {
5	0x4100000e	jeq	14	; jump to halt	
6	0x12000000	out		; output i	std::cout << i;
7	0x32000001	const	1	; decrement i	--i;
8	0x21000000	sub		;	
9	0x3200000f	const	15	;	
10	0x31000000	store		; store i	
11	0x3200000a	const	10	; output newline	std::cout << "\n";
12	0x13000000	outchar			
13	0x40000000	jmp	0	; jump to start	}
14	0x1000000	hlt		; stop program	
15	0xa	data	0xa	; variable i	
16		end			

Figure 2: Output numbers from 10 to 1. Instruction stream with disassembly and meaning in C++

Example of the fetch, decode and execute phase:

Assume the processor was loaded with the instruction stream displayed in Figure 2 and has already executed some steps such that the program counter PC has the value 4.

Fetch: In the fetch phase the processor retrieves the instruction from simulated memory located at address PC. Here this is mem[PC] $=$ mem[4] $=$ 0x32000000.

Decode: The processor decodes the machine instruction 0x32000000 $= 50 \cdot 2^{24} + 0$: opcode $=$ 50, operand $=$ 0. According to the specification, opcode 50 corresponds to a `const` instruction.

Execute: After decoding, the processor is ready to execute the instruction, i.e. it applies the effect of the `const` operation and pushes the value `0` to the operand stack. Another effect of the instruction is the advancing of the program counter by 1.

The processor continues with the fetch phase again: The next instruction to be executed is `0x4100000e` corresponding to `jeq 14`. The effect of this instruction is to compare the two top most values on the stack and to jump to 14 in case they are equal. The processor continues with this scheme until it has reached a `hlt` instruction.

Example test runs:

input	output	input	output	input	output
simulate	15	simulate	123456789	simulate	255
5		10		9	
0x3200000f		0x32000000		0x320000ff	
0x12000000		0x32000001		0x32000100	
0x3200000a		0x20000000		0x31000000	
0x13000000		0x26000000		0x32000100	
0x1000000		0x26000000		0x30000000	
		0x12000000		0x12000000	
		0x32000009		0x3200000a	
		0x41000009		0x13000000	
		0x40000001		0x1000000	
		0x1000000			

A Specification of the StackCPU$_{16}$ System

A.1 System Memory

Instructions and data of the simple CPU are stored in memory. The system memory of our simulated machine has a total capacity of 256 KB. It is 32-bit word addressed. Thus there are $2^{18}/4 = 2^{16}$ different addresses.

(32bit)	(32bit)	(32bit)	(32bit)	(32bit)	(32bit)	(32bit)	(32bit)
0	1	2	...	...	$2^{16}-3$	$2^{16}-2$	$2^{16}-1$

For the implementation of the memory you can assume that an `unsigned int` has at least 32 bits.

A.2 Operand Stack and Program Counter

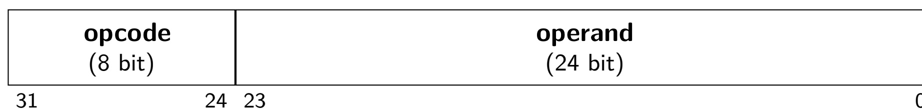
The StackCPU$_{16}$ machine is stack-based architecture. It features an *operand stack* for storing results of computations. It is used by machine instructions such as arithmetics or comparisons, or for short term storage of temporary values. All stack entries on this simulated processor are 32-bits wide. The operand stack has at least 16 entries.

The processor has a *register* called the program counter (PC). This register points to the address of the next instruction to be executed. This register is implicitly modified during the execution of any instruction.

You may for example implement the PC register using an `unsigned int`, and the operand stack using a vector of `int`, or `std::stack<int>`.

A.3 Instruction Format and Encoding

All instructions are 32-bit wide and made up of the following components: Operation code (opcode) and at most one operand. Opcode and operand are encoded at fixed locations in the instruction, but occupy different sizes as you can see in the following figure:



An opcode (the first two hex nibbles of the hexadecimal instruction value) uniquely identifies an instruction. The operand can hold a *signed* integer value, encoded by 24 bits (the last six hex nibbles of the hexadecimal instruction value), in two's complement representation [2], see Section A.5. If an instruction does not use an operand, its value is ignored by the processor.

You may for example implement instructions using `unsigned int`.

A.4 Instruction Set Properties

The StackCPU$_{16}$ virtual processor provides a Load-Store architecture, i.e. memory can be only accessed by load and store operations. All other operations operate on the operand stack or on immediate values.

The instruction set consists of general, stack, memory, arithmetic, and branch instructions. General instructions are used for input, output, memory access and system control. The arithmetic instructions implement mathematical operations (+,-,*,div,mod). They all work on 32-bit signed integers and operate on the operand stack. The behavior of the operations is supposed to correspond exactly to the behavior of C++ operators. Branch instructions can make the processor continue processing at a given location. There is one unconditional branch and branches that are triggered by a condition. Branch targets are absolute memory addresses.

[2] Let v be the nonnegative binary number formed by the 24 bits; if the highest of the 24 bits is 0, the encoded value is v, otherwise $v - 2^{24}$. For example, `0x00000a` = `0xa` represents the decimal value 10, while `0xffffff` is -1.

A.5 StackCPU$_{16}$ Instruction set

Legend:

c	denotes a constant and is encoded as signed integer operand
PC	stands for the program counter
S	stands for the operand stack
l, r, v, m	denote 32-bit signed integer values

Mnemonic and operands	Opcode	Description / Effect
General instructions		
`hlt`	0x1	Halt the system (exit simulator).
`in`	0x10	Read a decimal integer value v from standard input `std::cin`, and push v onto S. Then increment PC.
`inchar`	0x11	Read a character value v from standard input `std::cin`, and push v onto S. Then increment PC.
`out`	0x12	Pop the top value v from S, and write v to standard output as decimal integer. Then increment PC.
`outchar`	0x13	Pop the top value v from S ,and write v to standard output as character. Then increment PC.
Arithmetic instructions		
`add`	0x20	Pop value r from S. Pop value l from S. Push value $l+r$ onto S. Then increment PC.
`sub`	0x21	Pop value r from S. Pop value l from S. Push value $l-r$ onto S. Then increment PC.
`mul`	0x22	Pop value r from S. Pop value l from S. Push value $l \cdot r$ onto S. Then increment PC.
`div`	0x23	Pop value r from S. Pop value l from S. Push value l/r (integer division!) onto S. Then increment PC.
`mod`	0x24	Pop value r from S. Pop value l from S. Push value $l\%r$ (rest of integer division) onto S. Then increment PC.
`neg`	0x25	Pop value r from S. Push value $-r$ onto S. Then increment PC.
Stack Management		
`const c`	0x32	Push the integer value c onto S. Then increment PC.
`dup`	0x26	Pop value r from S. Push value r onto S twice. Then increment PC.
Memory operations		
`load`	0x30	Pop value m from S. Push the 32-bit signed integer value r stored at memory location m to S. Then increment PC.
`store`	0x31	Pop value m from S. Pop value v from S. Store v at memory location m as 32-bit signed integer value. Then increment PC.
Branch instructions		
`jmp c`	0x40	Set program counter PC to c.
`jeq c`	0x41	Jump if equal: Pop value r from S. Pop value l from S. If $l = r$ then assign c to PC, otherwise increment PC.
`jne c`	0x42	Jump if not equal: Pop value r from S. Pop value l from S. If $l \neq r$ then assign c to PC, otherwise increment PC.
`jls c`	0x43	Jump if less: Pop value r from S. Pop value l from S. If $l < r$ then assign c to PC, otherwise increment PC.
`jle c`	0x44	Jump if less or equal: Pop value r from S. Pop value l from S. If $l \leq r$ then assign c to PC, otherwise increment PC.