Structuur van Computerprogramma's II
Academic Year 2025-2026
Week 12 - Execution Model - Assignment

Clément Béra, Maarten Vandercammen, Robbe De Greef, Elisa Gonzalez Boix
Vrije Universiteit Brussel

The goal of this session is to implement a stack machine that is able to execute programs written in a minimal language called *SmiLang*, in which programs consist of sequences of local variable declarations and calls to functions (more precisely, calls to the + or - arithmetic operators, or to user-defined functions). Your task will be to implement the necessary operations on the stack machine to execute SmiLang programs.

Please have a look at the accompanying "SmiLang" document for further information on the SmiLang language and its instruction set.

1 SmiLang Stack Machine Setup

Figure 1 shows how SmiLang programs are executed by its stack machine. The SmiLang source code is translated to SmiLang instructions which are then executed one by one by the stack machine to produce a result. The figure shows this process for a code example implementing a function that returns `1`.

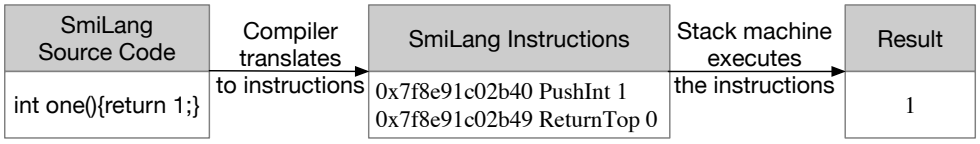

Figure 1: Compilation and execution flow

You can download a zip file containing a skeleton of a SmiLang stack machine and a list of SmiLang programs from Canvas. More concretely, the zip contains the following files:

- `makefile`**: allows you to compile the project with make,**
- `stackMachine.c`**: contains the skeleton of the SmiLang stack machine,**
- encoder.h and encoder.c: define the SmiLang instruction set and its encoding in bytes.
- examples.smi.c and examples.smi.h: contain a list of SmiLang programs, manually encoded to SmiLang instructions which can then be executed by the stack machine.

Out of all these files, you will only need to (1) use the makefile to run the stack machine and (2) complete the implementation of the `stackMachine.c` file.

Your task is to implement the functions in the `stackMachine.c` file to execute SmiLang instructions. The `stackMachine.c` file already contains code with all the logic needed for internal workings of the stack machine and some debugging functions.

The main goal is to implement the stack machine incrementally. To this end, eleven SmiLang sample programs are provided with an increasing level of difficulty. Each program encodes a SmiLang program,

program 1 being the easiest to execute and 11 the hardest. The level of difficulty implies the implementation of one or more instructions.

To implement the SmiLang instructions, you only have to implement one single C function per instruction. A template is provided for each of these functions in the section "Stack machine instructions" of the `stackMachine.c` file. For example, for the operation `PushTemp`, this template is provided:

```
void executePushTemp(unsigned char tempIndex){
        printf("Should be implemented: PushTemp\n");
        assert(0);
}
```

By default, the template just stops the execution of the instruction (e.g `PushTemp` in the code snippet) and logs in the console that the instruction has not been implemented. You will need to delete this code and implement the stack machine operation instead.

1.1 Running the Setup

The main function of `stackMachine.c` file tests in order all the eleven SmiLang sample programs by printing in the console `"Example X"`, then executing the SmiLang program on the stack machine, comparing the result against the expected result and printing in the console if the program was executed as expected or not.

To run the sample SmiLang programs provided, you need to `make` the project (a makefile is already provided) and run the executable produced from the command line.

The main function of `stackMachine.c` basically calls the `testExample` helper function which is the one that initiates the execution of a SmiLang program by calling the `executeFunction` which actually asks the stack machine to execute a given SmiLang function.

To execute a SmiLang function, the stack machine first sets up the stack pointers with invalid values. Actually, the stack machines employs a common marker which are used in C and assembler to mark invalid values, namely `0xdeadbeef` and `0xbadf00d`. Here is an initialized stack:

```
0x10f9920b8     fp      0xdeadbeef
0x10f9920b0     ip      0xbadf00d
```

The base of the stack, `0xbadf00d` is currently used as the return ip and will be set to the return value once the function returns.

Once set up, the stack machine then keeps executing instructions, which will use the stack to store its intermediate values. The stack machine stops when the frame pointer is set back to the invalid value through the `ReturnTop` instruction.

1.2 Debugging Functions

To help you understand what is going on, the stack machine skeleton provides debugging functions. Some debugging functions are already automatically called if the `DEBUG` flag in `stackmachine.c` is turned on.

The first debugging function is `void printFunction(Function *function)`, which takes a pointer to a SmiLang function as parameter, and prints both the instructions of the given function and the corresponding source code. For example:

```
> Instructions:
0x7fba76d00510          PushArg -2
0x7fba76d00512          PushArg -1
0x7fba76d00514          Sub
0x7fba76d00515          ReturnTop 2
> Source:
int minus(int arg1, int arg2){return arg1 - arg2;}
```

This can be used to know what SmiLang program is being executed, but also to know what instruction is being executed by comparing the instructions addresses to the current instruction pointer (ip global variable).

The second debugging function is `void printStack()`. Using sp and fp, it prints the current stack. The current sp and fp, if pointing to a value on stack, are printed prefixed by a left arrow. If the fp is valid, the stack is annotated with fp and ip to mark the frame pointer and return instruction pointer to the different stack frames present on stack. For example:

```
> Stack:
0x10f992110     fp      0x10f9920d8     <- sp   <- fp
0x10f992108     ip      0x7fba76d003ab
0x10f992100             7
0x10f9920f8             3
0x10f9920f0             3
0x10f9920e8             2
0x10f9920e0             6
0x10f9920d8     fp      0x10f9920b8
0x10f9920d0     ip      0x7fba76d002bb
0x10f9920c8             7
0x10f9920c0             5
0x10f9920b8     fp      0xdeadbeef
0x10f9920b0     ip      0xbadf00d
```

2 Assignment

The goal of the assignment is to implement all the stack operations of SmiLang (Pop, Plus, Minus, PushArg, PushTemp, StoreTemp, DefineTemp, ReturnTop, PushInt and Call) such that the stack machine can execute the eleven SmiLang sample programs provided with increasing level of difficulty.

The main function starts by testing the execution of the first SmiLang program. The execution should fail since some instructions used by the function are not yet implemented. Your task consists of implementing the required instructions to make the test pass, recompile and re-run the main. After this, the main function tests the execution of the next SmiLang program, which should fail and for which some instructions need to be implemented, and so on. You will keep implementing instructions until all the SmiLang programs provided can be executed by the stack machine.

To implement the requested instructions you only need to use the global C variables sp and fp, which are the stack pointer and frame pointer of the stack machine. The type of sp and fp is `(IntType *)`, `IntType` being the integer type manipulated by the stack machine.

Important: Do not write code aside from the "Stack machine instructions" section of the `stackMachine.c` file, or change **any** of the function signatures.

3 Going Further

Congratulations! If you reach here you have managed to complete the implementation of the instruction set for SmiLang. Let us take you further on the exploration of the SmiLang stack machine. What follows describes three possible extensions to the current machine. They independent from each other but ordered with increasing level of difficulty.

3.1 Stack Overflow Detection

So far, we have assumed that the stack was infinite: pushing values on the stack has been assumed to always work and never overflow from the memory reserved for the stack. In practice, the program needs to allocate a fixed amount of memory for the stack, and if the program pushes too many values, the stack overflows. In

the file `stackMachine.c`, a constant is defined to set the memory reserved for the stack (`#define StackSize 500`). Try to change that value from 500 (500 times the size of a pointer) to 10. What happens now when you run the examples? Which examples work correctly, which do not and why?

Extension. Rather than crashing with a segmentation fault when a stack overflows, let us extend our stack machine so that it detects the stack overflow before it actually happens. If detected, the stack machine should abort the program execution and print on the command line "Stack overflow" instead of doing a segmentation fault. What do you need to change in the stack machine to detect stack overflows? Implement it and make sure the SmiLang programs behave as expected when the `StackSize` variable is set to 10.

Efficiency Considerations. If the stack machine needs to check for a stack overflow at the execution of each instruction that pushes a value on the stack, this would make it get slow due to the extra computation. To solve this problem, let us assume that the SmiLang compiler is changed so that each function now starts with the instruction `EnsureSlotsOnStack xx`, with xx the maximum number of slots the function will use on stack. How would you implement the execution of this instruction? With that instruction, does the stack machine still needs to check for stack overflow at each instruction pushing values on the stack? Why?

While developing a compiler, a developer reported a stack underflow error. How did it happen?

3.2 SmiLang Data Types

So far SmiLang only operates on data of type integer. The Smi integers are currently the same size as pointers (See in `stackMachine.c`, function `ensureMachineAsExpected`). They are defined as long long in 64 bits and int in 32 and 16 bits (See in `encoder.h`, section Type definition). This means that when pushing and popping off the stack, the stack pointer is always moved by 2/4/8 bytes in respectively 16/32/64 bits, disregarding if we are pushing and popping integers or pointers (frame pointer, instruction pointer).

Extension. Let us extend SmiLang to support `char` data types. This implies to introduce a new instruction `PushChar xx` which pushes the xx `char`, encoded in a single byte, on the stack. How would you implement that instruction? What is the memory consumption of a `char` on the stack? Would it work with the rest of the stack machine (Plus, Call, etc.) ? Do you need to re-implement some instructions to encode which operands/parameters are char and not Smis? Draw the memory used by the stack before and after the execution of the `PushChar` instruction.

Efficiency Considerations. Note that on modern processors, unaligned memory reads are much more expensive than aligned memory reads. With your implementation of `PushChar`, will the stack machine perform unaligned memory reads? What are the trade-offs between execution time (i.e. maximizing aligned over unaligned memory reads), memory consumption, and engineering time (changing the implementation of Plus/Sub/Call to encode if there are char operands/parameters)?

3.3 Calling Convention

During the lectures you have seen the disassembled C code for using gcc on a Unix-based system. In this setting the standard x86 calling conventions employed were cdecl (for 32-bit processors) and System V AMD64 ABI (on 64-bit ones). When implementing a Virtual Machine (or in our case a simple stack machine) the calling convention employed is not determined by the hardware architecture where the machine runs on. The calling convention implemented in SmiLang is actually derived from stdcall, the calling convention for 32-bit x86 processors on Windows.

The main difference between the two families of calling conventions is the following. With stdcall, the callee is responsible for cleaning the function arguments off the stack. Recall that the `ReturnTop` instruction encodes the number of arguments of the current function which needs to remove from the stack before pushing the returned value. If the SmiLang calling convention was derived from cdecl, then the `ReturnTop`

instruction would not specify the number of function arguments and when returning, the arguments of the function would still be on the stack, below the returned value.

Extension. Have a look back at the instructions generated for the `minus` function shown in Section 1.2. Draw on paper the state of the stack just before executing the ReturnTop instruction, just after executing the ReturnTop instruction with the stdcall-style SmiLang calling convention, and with a new calling convention cdecl-style. Have a look now at the instructions generated for calling the `minus` function and using its return value in the `multiTempArgs` function in both calling conventions. What would need to change to implement a cdecl style calling convention in the SmiLang stack machine?

Efficiency Considerations. Let us assume that the SmiLang compiler is changed so that after the call instruction a new instruction needs to be used to clean up the stack. Compare this solution with the current stdcall-style SmiLang calling convention. Would it be possible to change the stack machine to provide an efficient implementation of cdecl-style calling conventions without requiring a new instruction?