Code Generation

Sungbin Jo (2021-13630), Kijun Shin (2021-15391)

SwiftSnuPL code generation implementation notes

The SwiftSnuPL compiler only supports the target triple `aarch64-linux-gnu`. Notably, the generated code uses a Linux ABI, with little endian semantics. For other platforms, linking to the aarch64 libc and running the resulting binary with user mode qemu is recommended. Detailed instructions for running binaries are noted below.

The SwiftSnuPL compiler uses a very simplistic model where every operand gets allocated a separate stack space, without any optimization attempts. Each IR instruction gets compiled to loads from the stack, a few aarch64 instructions that corresponds to the IR, and stores to the stack.

Because this is so simplistic, implementing itself is a breeze, with the checks for cases where immediate values don't fit in the instruction, and needs separate checks. There are quite a few shortcomings on this model: the biggest one is that successive operations operating on the same operand compiles to successive stores and loads on the same memory location. Properly fixing this problem would require some kind of register allocation scheme; a much simplistic approach could aim at least optimize useless loads by peeking the previous instruction and checking if an operand we needs already exists in a register, and optimizing the load into a move.

We have a version of this patch implemented, but due to correctness concerns, it is disabled by default.

Another would be that the compiler allocates every operand separately on the stack; this results in using a lot of stack space. A proper solution here would be doing some sort of liveness analysis on the IR, and allocating the same place on the stack. Unfortunately we could not implement this due to time constraints; we were hopefully out of schedule and had to aim the quickest path to completion.

Overall, the assembly generation part of the compiler was

Architecture

The code generator receives four inputs, as shown below:

```
class AssemblyGenerator {
    var instructions: [Resolver.Symbol: [IRGenerator.Instruction]]
    var allocations: [String: Int64]
    var stringLiterals: [String: [UInt8]]
    var globalVariables: Set<String>
}
```

The dictionary `instructions` is the direct output of the IR generator, each containing the instructions for each function.

The dictionary `stringLiterals` and `allocations` are a separate dictionary that has the information of the `.string` and `.allocation` operands in the instructions; the special handling of string literals is required as it is the only case in the language where literals must be compiled as pointers, instead of immediate values that can be directly embedded in the instructions.

Variables aren't contained directly, but it will be accessed when loop through main module/procedure/function instructions. `allocations` used to calculate size of some operands(IR). `stringLiterals` used to embed string literals on data section. Since string is not special type in SnuPL/2(it's just array of char), so embedding has to satisfy middleware structure of array.

The set `globalVariables` contains global variable names, separately taken from the `Resolver`; they are used to generate a symbol for each global in the data section.

Passing `globalVariables` to the code generator is not intended design; we realized only a bit too late that code can reference variables outside of scope, and that our IR representation do not have enough information about the nonlocal symbols. We would have liked a better model of representing them, e.g. for future extensions of the language that allows nested functions. Unfortunately time constraints forced us to do something that obviously works; we did not have enough time to rework the design.

If we had spent more thought designing the IR with a better representation of the nonlocal symbols, e.g. by also saving the scope depth of such symbol, we could have possibly implemented nested functions with access links, and implement global variables as stack variables of the `main` function; Unfortunately that was not possible.

Running through the instructions, collecting all operands, and with the `allocations` dictionary, the code generator allocates stack space to each operand, and store the stack locations into a dictionary called `stackMapping`.

Structure of assembly codes are related to binary file format. It starts with text section, where generated assembly code will placed. After that, data section located. We allocate space for global variables and string literals in here.

```
func generate() -> String {
    var assembly = """
```

```
        \t.text
        \(instructions.map(generate(symbol:instructions:)).joined(separator: "\n"))
        \t.data\n
        """
    for globalVariable in globalVariables { assembly += "\t.comm \(globalVariable),8\n" }
    for (key, value) in stringLiterals {
        assembly += """
            \(key):
            \t.word \(value.count + 1)
            \t.word 1\n
            """
        for char in value { assembly += "\t.byte \(char)\n" }
        assembly += "\t.byte 0\n"
    }
    return assembly
}
```

Each symbol in `instructions` represents a function or procedure; we iterate through the dictionary and generates assembly code for each symbol.

Code generation for a specific function starts by calculating the stack layout.

It collects all local operands of the instructions; combined with the `allocations` dictionary, the code generator allocates stack space to each operand, and store the stack locations into a dictionary called `stackMapping`. Also note that the ARM64 ISA requires the stack pointer to be always aligned to the 16bits boundary; we add padding to account that so that the stack size becomes a multiply of 16.

```
func generate(symbol: Resolver.Symbol, instructions: [IRGenerator.Instruction]) -> String {
    let operands = operands(in: instructions)
    var stackMapping: [IRGenerator.Operand: Int64] = [:]
    var stackSize: Int64 = 0
    for operand in operands {
        let size = size(of: operand)
        stackMapping[operand] = stackSize
        stackSize += size
    }
    if stackSize % 16 != 0 { stackSize += 16 - (stackSize % 16) }
    // assembly code generation...
```

Implementation Notes

Handling Big Constants To handle constants bigger than the limit of immediate values in one instruction, we have a separate helper function `embedImm` that generates assembly to split up the constant and moves them in pieces.

```
func embedImm(imm: Int64, scratch scratchRegister: String) -> String {
    if imm < 0 {
        return """
            \tmov \(scratchRegister), #\(imm % (1 << 16))
            \tmovk \(scratchRegister), #\((imm >> 16) % (1 << 16)), LSL 16
            \tmovk \(scratchRegister), #\((imm >> 32) % (1 << 16)), LSL 32
            \tmovk \(scratchRegister), #\((imm >> 48) % (1 << 16)), LSL 48
            """
    } else {
        var retasm = "\tmov \(scratchRegister), #\(imm % (1 << 16))\n"
        if imm >= (1 << 16) { retasm += "\tmovk \(scratchRegister), #\((imm >> 16) % (1 << 1
        if imm >= (1 << 32) { retasm += "\tmovk \(scratchRegister), #\((imm >> 32) % (1 << 1
        if imm >= (1 << 48) { retasm += "\tmovk \(scratchRegister), #\((imm >> 48) % (1 << 1
        return retasm
    }
}
```

Load/Storing Operands As mentioned upwards, each IR instruction loads and stores the involved operands; the helper function `withOperands` produces most of the load/store logic.

It adds assembly code for loading and storing the operands on the top and the bottom of the provided code; an example use case of the function is below.

```
case .unary(let op, let destination, let source):
    assembly += withOperands(
        load: [source], to: ["x8"],
        store: [destination], from: ["x8"],
        scratch: "x10"
    ) {
        switch op {
        case .neg:
            return """
            mov x9, xzr
            subs x8, x8, x9
            """
        // handle other cases...
        }
    }
```

The arguments `load` and `to` represents the operand/register pair that will get loaded to. Likewise, the `store` and `from` represents the operand/register pair that will get saved from.

The `scratch` register is required for calculating the stack position of an operand in the case where the stack location is bigger than 256 and does not fit in the limit of immediate values.

The `offset` parameter, not used above, is for when the stack pointer is changed for some reason. This is required when passing more than 8 arguments; to pass the arguments on the stack, the stack must be expanded, messing up the stack location.

Note the branch on the size of `stackOffset`: it selects the appropriate instruction that allows the format.

```
func withOperands(
    load sourceOperands: [IRGenerator.Operand] = [], to sourceRegisters: [String] = [],
    store destinationOperands: [IRGenerator.Operand] = [], from destinationRegisters: [Strin
    scratch scratchRegister: String, offset stackExpanded: Int64 = 0, body: (() -> String)?
) -> String {
    var assembly = ""
    for (operand, register) in zip(sourceOperands, sourceRegisters) {
        switch operand {
        case .symbol(let symbol):
            let stackOffset = stackMapping[operand]! + stackExpanded
            if symbol.isGlobal {
                assembly += """
                    adrp \(register), \(symbol.token.string)
                    add \(register), \(register), :lo12:\(symbol.token.string)
                    ldr \(register), [\(register)]
                    """
            } else {
                if stackOffset < 256 {
                    assembly += "ldr \(register), [sp, #\(stackOffset)]"
                } else if stackOffset < 4096 {
                    assembly += """
                        add \(register), sp, #\(stackOffset)
                        ldr \(register), [\(register)]
                        """
                } else {
                    assembly += embedImm(imm: stackOffset, scratch: scratchRegister);
                    assembly += """
                        add \(register), sp, \(scratchRegister)
                        ldr \(register), [\(register)]
                        """
                }
            }
        // handle other cases...
        }
    }
    if let body = body { assembly += body() }
    for (operand, register) in zip(destinationOperands, destinationRegisters) {
        switch operand {
        // handle other cases...
        }
    }
    return assembly
}
```

Get List of Every Operands in Procedure/Function Procedure/Function has instruction list made by middleware. By loop through it, controlling case by case, set of operands are made like this:

```
func operands(in instruction: IRGenerator.Instruction) -> Set<IRGenerator.Operand> {
    switch instruction {
    case .move(let destination, let source), .unary(_, let destination, let source): return
    case .binary(_, let destination, let source1, let source2): return [source1, source2, de
    case .parameter(let destination, _): return [destination]
    case .jump: return []
    case .branch(_, let source1, let source2): return [source1, source2]
    case .call(let destination, _, let arguments):
        var operands: Set<IRGenerator.Operand> = []
        for argument in arguments { operands.insert(argument) }
        if let destination = destination { operands.insert(destination) }
        return operands
    case .return(let value): if let value = value { return [value] } else { return [] }
    case .load(let destination, let source, _), .store(let source, let destination, _):
        return [source, destination]
    case .label: return []
    }
}
```

Cross-compiling `aarch64-linux-gnu` code in x86_64 environments

For building, `binutils` for assembling and linking, and `libc` for the `aarch64-linux-gnu` target triple is required.

The easiest way for running the produced binaries is by running them with QEMU user mode.

Assuming an Ubuntu 20.04 environment, the following commands produce an `aarch64-linux-gnu` binary and run it with `qemu-user`.

```
$ # Install required packages
$ sudo apt-get install binutils-aarch64-linux-gnu libc-dev-arm64-cross
$
$ # Compile, assemble, and link the binary
$ ./SwiftSnuPL compile -S main.mod > main.s
$ ./SwiftSnuPL stdlib -S > libSnuPL.s
$ aarch64-linux-gnu-as -o main.o main.s
$ aarch64-linux-gnu-as -o libSnuPL.o libSnuPL.s
$ aarch64-linux-gnu-ld -o main \
  -dynamic-linker /lib/ld-linux-aarch64.so.1 \
  /usr/aarch64-linux-gnu/lib/crt1.o \
  /usr/aarch64-linux-gnu/lib/crti.o \
  -lc libSnuPL.o main.o \
  /usr/aarch64-linux-gnu/lib/crtn.o
$
$ # Run the binary with QEMU user mode
$ qemu-aarch64 -L /usr/aarch64-linux-gnu ./main
```