Code Generation for the x86 ISA

- We want to retarget our recursive-descent code generator to generate x86 assembly language rather than SaM assembly language.

Code Generation for the x86 ISA

- We want to retarget our recursive-descent code generator to generate x86 assembly language rather than SaM assembly language.
- Major differences
  - Instruction set design: 0-operand (SaM) vs. 2-operand (x86).
  - Storage state: Memory only (LIFO) vs. random-access memory + general-purpose registers.
  - Addressing modes: Base + Offset only vs. richer set.
  - Run-time stack organization: Growing upwards vs. growing downwards.
  - Argument and result passing conventions: Stack-only vs. GPR + stack.

Code Generation for the x86 ISA

- We want to retarget our recursive-descent code generator to generate x86 assembly language rather than SaM assembly language.
- Major differences
  - Instruction set design: 0-operand (SaM) vs. 2-operand (x86).
  - Storage state: Memory only (LIFO) vs. random-access memory + general-purpose registers.
  - Addressing modes: Base + Offset only vs. richer set.
  - Run-time stack organization: Growing upwards vs. growing downwards.
  - Argument and result passing conventions: Stack-only vs. GPR + stack.
- Two possible approaches
  - Write a post-pass translator to convert SaM ASM to x86 ASM.
  - Re-implement the recursive-descent code generator to directly generate x86 ASM.

Baseline Code Generator

- To focus on building a simple correct code generator, make the following simplifying assumptions.
  - All local variables have a “home location” on the stack frame.
  - Access local variables using B+O mode with `%rsp` as the base register.
  - Reserve GPRs `%rbp`, `%rsp`, `%rax`, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` for their standard usages according to the ARM64 ABI.
  - Use other GPRs as scratchpad registers for expression evaluation.
  - Prefer to use caller-saved registers over callee-saved registers.
  - Separate memory access from data-processing.

Baseline Code Generator

- To focus on building a simple correct code generator, make the following simplifying assumptions.
  - All local variables have a “home location” on the stack frame.
  - Access local variables using B+O mode with `%rsp` as the base register.
  - Reserve GPRs `%rbp`, `%rsp`, `%rax`, `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8`, and `%r9` for their standard usages according to the ARM64 ABI.
  - Use other GPRs as scratchpad registers for expression evaluation.
  - Prefer to use caller-saved registers over callee-saved registers.
  - Separate memory access from data-processing.

- Things to consider.
  - Keep arguments register-only, or also have a “home location” on the stack frame?
  - Save/restore all callee-saved registers in prologue/epilogue?
  - Design a custom (simplified but non-standard) linkage for LiveOak?

Some Terminology

- Basic block
  - A maximal single-entry single-exit sequence of statements.
  - Control enters at the beginning of the block.
  - Control leaves at the end of the block without halting or branching except possibly at the end.

Some Terminology

- Basic block
  - A maximal single-entry single-exit sequence of statements.
  - Control enters at the beginning of the block.
  - Control leaves at the end of the block without halting or branching except possibly at the end.
- Definitions and uses
  - The statement `x = expn` is said to define `x` and use all the variables in `expn` (which may include `x`).

Some Terminology

- Basic block
  - A maximal single-entry single-exit sequence of statements.
  - Control enters at the beginning of the block.
  - Control leaves at the end of the block without halting or branching except possibly at the end.
- Definitions and uses
  - The statement `x = expn` is said to define `x` and use all the variables in `expn` (which may include `x`).
- Liveness
  - A name in a basic block is live at a given point if its value at that point is used at a later point in the program, possibly in another basic block.
  - Highly conservative approximation: All named program variables are live at the exit point of a basic block.

Tracking Variable Mappings

- Two descriptors used *within the compiler* to keep track of locations of variables and temporaries.
  - Register descriptor (RD): What is in each register.
    - Empty (with a small caveat) at the beginning of a basic block.
    - At any point in the basic block, contains the values of zero or more literals, identifiers, or temporaries.
  - Address descriptor (AD): The location(s) where the current value of a name can be found at run-time.
    - Register, memory location (stack frame or static area), or some combination of these.
    - Memory location is considered “home”.
    - Can be maintained in symbol record or AST node record.
    - Maintain a modified flag (MF) to indicate when a variable is defined by a statement, to trigger the saving of the register-resident value to home location at the end of the basic block.

Code Generation for Basic Blocks

- Base case, X∈{**num,** `y`}:
  - If $AD(X)$ ∋ some register R, return R.
  - Otherwise, $R = \text{getreg}()$.
    - $AD(X) = AD(X) \cup \{R\}$; $RD(R) = RD(R) \cup \{X\}$; $MF(X) = \text{false}$.
    - Emit the instruction `MOV<x> val-or-memloc(X), <R>`.
    - Return R.

Code Generation for Basic Blocks

- Base case, X∈{**num,** y}:
  - If $AD(X)$ ∋ some register R, return R.
  - Otherwise, $R = \text{getreg}()$.
    - $AD(X) = AD(X) \cup \{R\}; RD(R) = RD(R) \cup \{X\}; MF(X) = \text{false}.$
    - Emit the instruction `MOV<x> val-or-memloc(X), <R>`.
    - Return R.
- Subtree $T = \text{op}(T_1, T_2)$:
  - $R_1 = \text{codegen}(T_1); R_2 = \text{codegen}(T_2)$.
  - Emit the instruction `<OP><x> <R1>, <R2>` (i.e., $R_2 \leftarrow R_2 \text{ op } R_1$).
  - $AD(T_2) = AD(T_2) \setminus \{R_2\}; RD(R_2) = RD(R_2) \setminus \{T_2\}; MF(T_2) = \text{true}.$
  - If $\text{istemp}(T_1)$:
    - $AD(T_1) = AD(T_1) \setminus \{R_1\}; RD(R_1) = RD(R_1) \setminus \{T_1\}.$
  - Return R_2.

Code Generation for Basic Blocks

- Base case, X∈{**num,** y}:
  - If $AD(X)$ ∋ some register R, return R.
  - Otherwise, $R = \text{getreg}()$.
    - $AD(X) = AD(X) \cup \{R\}; RD(R) = RD(R) \cup \{X\}; MF(X) = \text{false}.$
    - Emit the instruction `MOV<x> val-or-memloc(X), <R>`.
    - Return R.
- Subtree $T = \text{op}(T_1, T_2)$:
  - $R_1 = \text{codegen}(T_1); R_2 = \text{codegen}(T_2).$
  - Emit the instruction `<OP><x> <R1>, <R2>` (i.e., $R_2 \leftarrow R_2 \text{ op } R_1$).
  - $AD(T_2) = AD(T_2) \setminus \{R_2\}; RD(R_2) = RD(R_2) \setminus \{T_2\}; MF(T_2) = \text{true}.$
  - If $\text{istemp}(T_1)$:
    - $AD(T_2) = AD(T_2) \setminus \{R_2\}; RD(R_2) = RD(R_2) \setminus \{T_2\}.$
  - Return R_2.
- What should you do if op is the assignment operator?
- At end of basic block, store values of live variables defined by statements within the basic block to their home locations.

The Register Allocator: getreg()

- For now, we will use a simple yet generally effective register allocator.
  - Initialize the register map so that the appropriate registers contain the input parameters of the method.
  - Scan the list of all registers to find r whose $RD(r)$ is empty. Return r.
  - If no such register exists:
    - Select a register s to spill.
    - Generate code to store its contents back to memory, if needed.
    - Return s.

The Register Allocator: getreg()

- For now, we will use a simple yet generally effective register allocator.
  - Initialize the register map so that the appropriate registers contain the input parameters of the method.
  - Scan the list of all registers to find r whose $RD(r)$ is empty. Return r.
  - If no such register exists:
    - Select a register s to spill.
    - Generate code to store its contents back to memory, if needed.
    - Return s.

- Also consider not loading constants to registers, but using them as immediate operands of data-processing instructions instead.
  - This is what `val-or-memloc(X)` is intended to convey.