Lecture 5-6

COMPILER DESIGN
Announcements

• HW2: X86lite
  – Requires lots of work, start early
  – Missing a team partner, check in Moodle
  – How to register, will be announced in Moodle
"Lowering" an AST to ASM Code

```
.text
.globl program
program:
pushq %rbp
movq %rsp, %rbp
pushq %rdi
pushq %rsi
popq %rax
popq %r10
addq %r10, %rax
pushq %rax
pushq %rdx
popq %rax
popq %r10
imulq %r10, %rax
pushq %rax
popq %rax
popq %rbp
retq
```

```
(X1 + X2) – X3
```

```
Var “X1”
Var “X2”
Add
Sub
Var “X3”
```

```
(X1 + X2) – X3
```
see compile.ml in lec04.zip

DIRECTLY GENERATING X86
Directly Translating AST to Assembly

• For simple languages, no need for intermediate representations
  – e.g. the arithmetic expression language from before

\[(X1 + X2) - X3\]

• Main idea: **maintain invariants**
  – e.g. code emitted for a given expression computes the answer into rax

• Key challenges
  – storing intermediate values needed to compute complex expressions
  – some instructions use specific registers (e.g., shift)
One Simple Strategy

• Compilation “emits” instructions into an instruction stream

• To compile an expression
  – Recursively compile its sub-expressions
  – Process the results

• Invariants
  – Compilation of an expression yields its result in \texttt{rax}
  – Argument \((X_i)\) is stored in a dedicated operand
  – Intermediate values are pushed onto the stack
  – Stack slot is popped after use (so the space is reclaimed)

• Resulting code is wrapped to comply with cdecl calling conventions

• See \texttt{compile.ml} from lec04.zip
INTERMEDIATE REPRESENTATIONS
Why do something else?

• We followed a simple *syntax-directed* translation
  – Input syntax uniquely determines the output
    • No complex analysis or code transformation is done
  – It works fine for simple languages
Why do something else?

But …

• The resulting code quality is poor

\[(X1 - X1) + 3\]
Why do something else?

But …

• Richer source language features are hard to encode
  – Structured data types
  – Objects
  – First-class functions
  – …
Why do something else?

But …

- It is hard to optimize the resulting assembly code
  - The representation is too concrete
    - e.g., it has committed to using certain registers and the stack
  - Only a fixed number of registers
  - Some instructions restrict where operands are located

```
.text
.globl _program
_program:
pushq %rbp
movq %rsp, %rbp
movq %rdi, %rax
pushq %rax
movq %rdi, %rax
imulq $-1, %rax
popq %r10
addq %r10, %rax
pushq %rax
pushq %rax
movq $3, %rax
popq %r10
addq %r10, %rax
popq %rbp
retq
```
Why do something else?

But …

• Retargeting the compiler to a new architecture is hard
  – Target assembly code is hard-wired into the translation

C  IR  x86  Java Byte-code  Arm

Rust

OCaml

egin{verbatim}
.globl _program
_program:
pushq  %rbp
movq  %rsp, %rbp
movq  %rdi, %rax
pushq  %rax
movq  %rdi, %rax
imulq  $-1, %rax
popq  %r10
addq  %r10, %rax
pushq  %rax
movq  %rax
addq  %r10, %rax
popq  %r10
retq
end{verbatim}
But …

• Control-flow is not structured
  – Arbitrary jumps from one code block to another
  – Implicit fall-through makes sequences of code non-modular
(i.e. can’t rearrange sequences of code easily)

factorial:
  pushq %rbp
  movq %rsp, %rbp
  cmpq $1, %rdi
  jg .GT1
  .LE1:
    movq $1, %rax
    jmp .EXIT
  .GT1:
    pushq %rdi
    subq $1, %rdi
    call factorial
    popq %rdi
    imulq %rdi, %rax
  .EXIT:
    movq %rbp, %rsp
    popq %rbp
    ret
Intermediate Representations (IR’s)

- **Abstract machine code**
  - Hides details of the target architecture
  - Allows machine independent code generation and optimization
Multiple IR’s

- Goal: get program closer to machine code without losing the information needed to do analysis and optimizations
- In practice, multiple intermediate representations might be used (for different purposes)
What makes a good IR?

- Easy translation target (from the level above)
- Easy to translate (to the level below)
- Narrow interface
  - Fewer constructs means simpler phases/optimizations
What makes a good IR?

- **Example**: Source language have “while”, “for”, “foreach” loops (and possibly more variants)
  - IR might have only “while” loops and sequencing
  - Translation eliminates “for” and “foreach”

\[
\begin{align*}
\text{[for(pre; cond; post) \{body\}]} &= \\
\text{[pre; while(cond) \{body; post\}]} \\
\end{align*}
\]

- Here the notation \([cmd]\) denotes the “translation / compilation” of cmd
High-level IR’s

- Abstract syntax + new node types not generated by the parser
  - e.g., type checking information or disambiguated syntax nodes
- Typically preserves the high-level language constructs
  - Structured control flow, variable names, methods, functions, etc.
  - May do some simplification (e.g. convert `for` to `while`)
- Allows high-level optimizations based on program structure
  - e.g., inlining “small” functions, reuse of constants, etc.
- Useful for semantic analyses like type checking
IR’s at the extreme

• Low-level IR’s
  – Machine dependent assembly code + extra pseudo-instructions
    • e.g. a pseudo instruction for interfacing with garbage collector or memory allocator (parts of the language runtime system)
    • e.g. (on x86) an imulq instruction that doesn’t restrict register usage
  – Source structure of the program is lost
    • Translation to assembly code is straightforward
  – Allows low-level optimizations based on target architecture
    • e.g. register allocation, instruction selection, memory layout, etc.

• What’s in between?
Mid-level IR’s: Many Varieties

- Intermediate between AST (abstract syntax) and assembly
- May have unstructured jumps, abstract registers or memory locations
- Convenient for translation to high-quality machine code
  - Example: all intermediate values might be named to facilitate optimizations that attempt to minimize stack/register usage

\[(X1 + (X2 - X3)) \times X4\]

\[
t1 = X2 - X3 \\
t2 = X1 + t1 \\
t3 = t2 \times X4
\]
Mid-level IR’s: Many Varieties

• Many examples
  – Triples: OP a b
    • Useful for instruction selection on X86 via “tiling”
  – Quadruples: a = b OP c (“three address form”)
  – SSA: variant of quadruples where each variable is assigned exactly once
    • Easy dataflow analysis for optimization
    • e.g. LLVM: industrial-strength IR, based on SSA
  – Stack-based
    • Easy to generate
    • e.g. Java Bytecode
Growing an IR

• Develop an IR in detail… starting from the very basic

• Start: a very simple IR for the arithmetic language
  – Very high level
  – No control flow

• Goal: A simple subset of the LLVM IR
  – LLVM = “Low-level Virtual Machine”
  – Used in HW3+

• Add features needed to compile rich source languages
SIMPLE LET-BASED IR
Eliminating Nested Expressions

- Fundamental problem
  - Compiling complex & nested expression forms to simple operations

\[
((1 + X4) + (3 + (X1 \times 5)))
\]

- Idea: name intermediate values, make order of evaluation explicit
  - No nested operations
Translation to SLL

• Given

\[
\text{Add}(\text{Add}(\text{Const } 1, \text{Var } X4), \\
\text{Add}(\text{Const } 3, \text{Mul}(\text{Var } X1, \\
\text{Const } 5)))
\]

• Translate to this desired SLL form

```ocaml
define tmp0 = add 1L varX4 in
define tmp1 = mul varX1 5L in
define tmp2 = add 3L tmp1 in
define tmp3 = add tmp0 tmp2 in
    tmp3
```

• Translation makes the order of evaluation explicit
• Names intermediate values
• Note: introduced temporaries are never modified
see: ir-by-hand.ml, ir<X>.ml

INTERMEDIATE REPRESENTATIONS
Intermediate Representations

• IR1: Expressions
  – simple arithmetic expressions, immutable global variables

• IR2: Commands
  – global *mutable* variables
  – commands for update and sequencing

• IR3: Local control flow
  – conditional commands & while loops
  – *basic blocks*

• IR4: Procedures (top-level functions)
  – local state
  – call stack
Basic Blocks and CFGs

- A sequence of instructions that is always executed starting at the first instruction and always exits at the last instruction
  - Starts with a label that names the *entry point* of the basic block.
  - Ends with a control-flow instruction (e.g. branch or return) the “link”
  - Contains no other control-flow instructions
  - Contains no interior label used as a jump target

- Basic blocks can be arranged into a *control-flow graph (CFG)*
  - Nodes are basic blocks
  - There is a directed edge from node A to node B if the control flow instruction at the end of block A *might* jump to the label of block B
factorial:
  pushq  %rbp
  movq  %rsp, %rbp
  cmpq  $1, %rdi
  jg .GT1

.LE1:
  movq  $1, %rax
  jmp .EXIT

.GT1:
  pushq  %rdi
  subq  $1, %rdi
  call factorial
  popq  %rdi
  imulq  %rdi, %rax

.EXIT:
  movq  %rbp, %rsp
  popq  %rbp
  ret
Basic Blocks and CFGs

factorial:
  pushq %rbp
  movq %rsp, %rbp
  cmpq $1, %rdi
  jg .GT1

.LE1:
  movq $1, %rax
  jmp .EXIT

.GT1:
  pushq %rdi
  subq $1, %rdi
  call factorial
  popq %rdi
  imulq %rdi, %rax

.EXIT:
  movq %rbp, %rsp
  popq %rbp
  ret
Coming up ...

Low-Level Virtual Machine (LLVM)