Prelude

- What is a qubit?
  - Quantum bit
- Why quantum computing?
  - “Superposition” can search solutions to a problem simultaneously
  - 3 bits: 1 of 8 possible values
  - 3 qubits = all 8 values, with probabilities
- Is it fundamentally more powerful?
  - No. Just massively parallel.

Practical?

Today

- Code generation

High-level IR

- High-level language constructs
  - Array accesses, field accesses
  - Complex control flow
    - Loops, conditionals, switch, break, continue
  - Procedures: callers and callees
  - Arithmetic and logic operators
    - Including things like short-circuit && and ||
- Tree structure
  - Arbitrary nesting of expressions and statements

Low-level IR

- Instructions for an abstract machine
- Single operation assignments

\[ x = y \text{ op } z \]

- Operations:
  - Arithmetic: add, sub, etc.
  - Comparisons: >, <, >=, <=
  - Logical operations: & | ~
  - Also: unary operations
Low-level IR

- Load and store

  \[ x = *p \quad x = \text{load } p \]
  \[ *p = y \quad \text{store } *p = y \]

  Note: we must build addresses using arithmetic

- Simple control-flow

  \[
  \text{label1: goto label1 }
  \]
  \[
  \text{if\_goto } x, \text{ label1 }
  \]
  \[
  \text{Jump to label1 if } x \text{ has non-zero value }
  \]

HIR to LIR example

\[
\begin{align*}
\text{if } c == 0 \{ \\
\text{while } c < 20 \{ \\
\quad c = c + 2 \\
\}
\} \\
\text{else} \\
\quad c = n * n + 2 \\
\text{goto end }
\text{lab1: }
\]
\[
\text{if\_goto } t1, \text{ lab1 }
\]
\[
t2 = n * n \\
t3 = c >= 20 \\
\text{if\_goto } t3, \text{ end }
\]
\[
c = c + 2 \\
\text{goto lab1 }
\text{end: }
\]

Lowering

- How do we translate from high-level IR to low-level IR?
  
  HIR is complex, with nested structures
  
  LIR is low-level, with everything explicit
  
  Need a systematic algorithm

- Idea:
  
  Define translation for each AST node, assuming we have code for children
  
  Come up with a scheme to stitch them together
  
  Recursively descend the AST

Lowering scheme

- Define a \textit{generate} function
  
  For each kind of AST node
  
  Produces code for that kind of node
  
  May call generate for children nodes

- How to stitch code together?
  
  Generate function returns a temporary (or a register) holding the result
  
  Emit code that combines the results

Lowering expressions

- Arithmetic operations

  \[
  \text{expr1 op expr2 }
  \]

  \[
  \begin{align*}
  t1 &= \text{generate}\text{(expr1)} \\
  t2 &= \text{generate}\text{(expr2)} \\
  r &= \text{new temp}() \\
  \text{emit} \left( r = t1 \text{ op } t2 \right) \\
  \text{return } r \\
  \end{align*}
  \]

  Generate code for left and right children, get the registers holding the results

  Obtain a fresh register name

  Emit code for this operation

  Return the register to generate call above

- Logic operations

  \[
  \begin{align*}
  \text{What about } && \text{ and } ||?
  \quad \text{In C and Java, they are “short-circuiting”}
  \quad \text{Need control flow…}
  \end{align*}
  \]
Short-circuiting $||$

- If $\text{expr1}$ is true, don’t eval $\text{expr2}$

$$
\text{expr1} || \text{expr2}
$$

```
E = new_label()
E = new_label()
t1 = generate(expr1)
emit( r = t1 )
emit( if goto t1, E )
t2 = generate(expr1)
emit( r = t2 )
emit( E: )
return r
```

Details...

```
E = new_label()
r = new_temp()
t1 = exp1
emit( r = t1 )
emit( if goto t1, E )
t2 = expr2
emit( r = t2 )
emit( E: )
return r
```

Details...

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

Helper functions

- **emit()**
  - The only function that generates instructions
  - Adds instructions to end of buffer
  - At the end, buffer contains code
- **new_label()**
  - Generate a unique label name
  - Does not update code
- **new_temp()**
  - Generate a unique temporary name
  - May require type information

```
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if not goto t1, E )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

Short-circuiting $&&$

```
N = new_label()
E = new_label()
r = new_temp()
t1 = generate(expr1)
emit( r = t1 )
emit( if goto t1, N )
et( goto N )
emit( N: )
t2 = generate(expr2)
emit( r = t2 )
emit( E: )
return r
```

Array access

- Depends on abstraction
- OR:
  - Emit array op
  - Lower later

```
E = new_label()
r = new_temp()
a = generate(expr1)
o = generate(expr2)
emit( o = a * size )
emit( r = load a )
return r
```

Type Information from the symbol table
Statements
- Simple sequences
  ```
  statement1;
  statement2;
  ... 
  statementN;
  ```
- Conditionals
  ```
  if (expr) 
  statement;
  ```

Loops
- Emit label for top of loop
- Generate condition and loop body
  ```
  while (expr) 
  E = new_label()
  t = generate(expr) 
  emit( ifnot goto t, E )
  generate(statement) 
  emit( goto T )
  emit( E: )
  ```

Function call
- Different calling conventions
  ```
  x = f(expr1, 
  expr2, 
  ...);
  a = generate(f) 
  foreach expr_i 
  ti = generate(expr_i) 
  emit( push ti )
  emit( call_jump a )
  emit( x = get_result )
  ```

For loop
- How does “for” work?
  ```
  for {expr1, expr2, expr3} 
  statement
  ```

Assignment
- Problem
  - Difference between right-side and left-side
    - Right-side: a value
    - Left-side: a location
  - Example: array assignment
    ```
    A[i] = B[j]
    ```

Special generate
- Define generate for l-values
  - lgenerate returns register contains address
  - Simple case: also applies to variables
  ```
  r = new_temp()
  a = generate(expr1) 
  o = generate(expr2) 
  emit( o = o * size )
  emit( a = a + o ) 
  emit( r = load a )
  return r
  ```

Assignment

- Use lgenerate for left-side
- Return r-value for nested assignment

```plaintext
expr1 = expr2;
l = lgenerate(expr2);
emit(store *l = r);
return r;
```

At leaves

- Depends on level of abstraction

  - `generate(v)` – for variables
    - All virtual registers: return v
    - Strict register machine: emit(r = load v)
    - Lower level: emit(r = load base + offset)
    - *base* is stack pointer, *offset* from symbol table
    - Note: may introduces many temporaries
  
  - `generate(c)` – for constants
    - May return special object to avoid r = #

Generation: Big picture

```plaintext
def generate(ASTNode node):
    reg r;
    switch (node.getKind()) {
        case BIN:
            t1 = generate(node.getLeft());
            t2 = generate(node.getRight());
            r = new_temp();
            emit(r = t1 op t2);
            break;
        case NUM:
            r = new_temp();
            emit(r = node.getValue());
            break;
        case ID:
            r = new_temp();
            o = symtab.getOffset(node.getID());
            emit(r = load sp + o);
            break;
    }
    return r
```

Code Shape

- Definition
  - All those nebulous properties of the code that impact performance & code “quality”
  - Includes:
    - Code for different constructs
    - Cost, storage requirements & mapping
    - Choice of operations
    - Code shape is the end product of many decisions

- Impact
  - Code shape influences algorithm choice & results
  - Code shape can encode important facts, or hide them

An example:

- What if x is 2 and z is 3?
- What if y + z is evaluated earlier?
- The “best” shape for x + y + z depends on context
- There may be several conflicting options
Order of evaluation

- Ordering for performance
  - Using associativity and commutativity
    - Very hard problem
  - Operands
    - Op1 must be preserved while op2 is computed
    - Emit code for more intensive one first
- Language requirements
  - Sequence points:
    - Places where side effects must be visible to other operations
  - C examples:
    - \( f() + g() \) may be executed in any order
    - \( f() || g() \) must be executed first
    - \( f(i++) \) argument to \( f \) must be \( i+1 \)

Code Generation

- Tree-walk algorithm
  - Notice: generates code for children first
  - Effectively, a bottom up algorithm
  - So that means....
- Right, syntax directed translation
  - Can emit LIR code in productions
  - Pass registers in $$, $1, $2, etc.
  - Tricky part: assignment

One-pass code generation

Goal: \( \text{Expr} ( \text{ $$} \times \text{ $$} ) \)

\( \text{Expr} + \text{Term} \)
- \( x = \text{new_temp}(); \)
- \( \text{emit}( r = \$1 + \$2 ); \)
- \( \text{$$} = r; \)
- \( \text{Expr} - \text{Term} \)
- \( x = \text{new_temp}(); \)
- \( \text{emit}( r = \$1 - \$2 ); \)
- \( \text{$$} = r; \)
- \( \text{Term} \times \text{Fact} \)
- \( x = \text{new_temp}(); \)
- \( \text{emit}( r = \$1 \times \$2 ); \)
- \( \text{$$} = r; \)
- \( \text{Term} / \text{Fact} \)
- \( x = \text{new_temp}(); \)
- \( \text{emit}( r = \$1 - \$2 ); \)
- \( \text{$$} = r; \)

Fact: \( \text{ID} \)
- \( x = \text{new_temp}(); \)
- \( o = \text{symtab.getOffset}(\$1); \)
- \( \text{emit}( r = \text{load sp + o }); \)
- \( \text{$$} = r; \)
- \( \text{NUM} \)
- \( x = \text{new_temp}(); \)
- \( \text{emit}( r = \$1 ); \)
- \( \text{$$} = r; \)

Next time...

- Generating more efficient code
- Some machine-specific examples