• Reminder: HW1 due today
• HW2 on course webpage. Due March 5
  • You will implement:
    • A simulator for X86lite machine code
    • An assembler
    • A loader
  • You can expect this assignment to require more time than HW1. Start early!
• You may work individually or in pairs
Each instruction has at most three operands ("three-address code")

\[
\begin{align*}
\text{<instr> :=} & \text{let } \text{<uid> = <operand> } \text{<op> <operand>;} \\
                    & \text{load } \text{<uid> = <var>;} \\
                    & \text{store } \text{<var> = <operand>;}
\end{align*}
\]

\[
\begin{align*}
\text{<operand> :=} & \text{<uid> | <integer>} \\
\text{<op> :=} & + \mid *
\end{align*}
\]
Control Flow
Concrete syntax

\[
\begin{align*}
\text{<control>} & ::= \text{br <label>} && \text{Branch} \\
\text{cbr <cc> <operand> <label> <label>} & \quad \text{Conditional branch} \\
\text{return <operand>} & \quad \text{Return} \\
\text{<cc>} & ::= \text{EqZ} \mid \text{LeZ} \mid \text{LtZ} \\
\text{<block>} & ::= \text{<instr><block>} \mid \text{<control>} \\
\text{<program>} & ::= \text{<program><label>: <block>} \mid \text{<block>}
\end{align*}
\]
int sum_upto(int n) {
    int sum = 0;
    while (n > 0) {
        sum += n;
        n--;
    }
    return sum;
}
Control Flow Graphs (CFG)

```c
int sum_upto(int n) {
    int sum = 0;
    while (n > 0) {
        sum += n;
        n--;
    }
    return sum;
}
```

![Control Flow Graph](image)
• Control flow graphs are a graphical representation of the control flow through a procedure
• A basic block is a sequence of instructions that
  1. Starts with an entry, which is named by a label
  2. Ends with a control-flow instruction (br, cbr, or ret)
     • the terminator of the basic block
  3. Contains no interior labels or control flow instructions
• A control flow graph (CFG) for a procedure $P$ is a directed, rooted graph where
  • The nodes are basic blocks of $P$
  • There is an edge $BB_i \rightarrow BB_j$ iff $BB_j$ may execute immediately after $BB_i$
  • There is a distinguished entry block where the execution of the procedure begins
• CFG models all program executions
  • Every execution corresponds to a path in the CFG, starting at entry
    • Path = sequence of basic blocks $B_1, \ldots, B_n$ such that for each $i$, there is an edge from $B_i$ to $B_{i+1}$
    • Simple path = path without basic blocks
  • (But not vice-versa!)
• CFG models all program executions
  • Every execution corresponds to a path in the CFG, starting at entry
    • Path = sequence of basic blocks $B_1, \ldots, B_n$ such that for each $i$, there is an edge from $B_i$ to $B_{i+1}$
    • *Simple* path = path without basic blocks
  • (But not vice-versa!)
• Simple application: **dead code elimination**
  1 Depth-first traversal of the CFG
  2 Any *unvisited node* is removed
• Graph structure used extensively in optimization (data flow analysis, loop recognition, ...
Why basic blocks?

• Control flow graphs may be defined at the instruction-level rather than basic-block level
• However, there are good reasons for using basic blocks
  • More compact
  • Some optimization passes ("local" optimizations) operate at basic block level
Constructing a CFG

• Traverse statements in IR from top to bottom
  • Find leaders
    • First statement
    • First statement following a label
  • Basic block = leader up to (but not including) next leader

• Can also construct CFG directly from AST
Generating code from a CFG

- Simple strategy: terminator always compiles to return / jump / conditional jump
  - “Fall-through” semantics of assembly blocks is never used

- A covering set of traces is a set of traces such that
  - Each trace is simple (loop free)
  - Each basic block belongs to a trace
- Basic algorithm: depth-first traversal of the CFG
  - If at least one successor is unvisited, elide jump and place the successor next in sequence
  - If all successors are visited, terminate branch
Generating code from a CFG

• Simple strategy: terminator always compiles to return / jump / conditional jump
  • “Fall-through” semantics of assembly blocks is never used

• More efficient strategy: elide jumps by ordering blocks appropriately
  • A covering set of traces is a set of traces such that
    • Each trace is simple (loop free)
    • Each basic block belongs to a trace
Generating code from a CFG

• Simple strategy: terminator always compiles to return / jump / conditional jump
  • “Fall-through” semantics of assembly blocks is never used

• More efficient strategy: elide jumps by ordering blocks appropriately
  • A covering set of traces is a set of traces such that
    • Each trace is simple (loop free)
    • Each basic block belongs to a trace

• Basic algorithm: depth-first traversal of the CFG
  • If at least one successor is unvisited, elide jump and place the successor next in sequence
  • If all successors are visited, terminate branch