Outline

1. Loop Optimizations
2. Dominators
3. Loop-Invariant Computations
4. Induction Variables
5. Array-Bounds Checks
6. Loop Unrolling
Loop Optimizations

- Loops are everywhere
  ⇒ worthwhile target for optimization
Loops

**Definition: Loop**

A loop with header $h$ is a set $S$ of nodes in a CFG such that

- $h \in S$
- $(\forall s \in S)$ exists path from $s$ to $h$
- $(\forall s \in S)$ exists path from $h$ to $s$
- $(\forall t \notin S) (\forall h \neq s \in S)$ there is no edge from $t$ to $s$

**Special loop nodes**

- A loop entry node has a predecessor outside the loop.
- A loop exit node has a successor outside the loop.
Example Loops

1
2
3
Example Loops

18-1b
Example Loops
18-1c
Example Loops
function isPrime (n: int) : int =
  (i := 2;
   repeat j := 2;
     repeat if i*j==n
       then return 0
         else j := j+1
     until j=n;
   i := i+1;
   until i==n;
  return 1)
Reducible Flow Graphs

- Arbitrary flow graphs: Spaghetti code
- Reducible flow graphs arise from structured control
  - if-then-else
  - while-do
  - repeat-until
  - for
  - break (multi-level)
Irreducible Flow Graphs

18-2a: Not a loop

Diagram of a flow graph showing nodes 1, 2, and 3 with directed edges.
Irreducible Flow Graphs

18-2b: Not a loop
Irreducible Flow Graphs

18-2c: Not a loop

- Reduces to 18-2a: collapse edges \((x, y)\) where \(x\) is the only predecessor of \(y\)
- A flow graph is irreducible if exhaustive collapsing leads to a subgraph like 18-2a.
Outline

1. Loop Optimizations
2. Dominators
3. Loop-Invariant Computations
4. Induction Variables
5. Array-Bounds Checks
6. Loop Unrolling
Dominators

- Objective: find loops in flow graph
- Assumption: each CFG has unique start node $s_0$ without predecessors

**Domination relation**

A node $d$ **dominates** a node $n$ if every path from $s_0$ to $n$ must go through $d$.

- Remark: domination is reflexive
Let $n$ be a node with predecessors $p_1, \ldots, p_k$ and $d \neq n$ a node. If $(\forall i) \ d \ \text{dominates} \ p_i$, then $d \ \text{dominates} \ n$ and vice versa. Let $D[n]$ be the set of nodes that dominate $n$.

**Domination equation**

\[
D[s_0] = \{s_0\} \quad D[n] = \{n\} \cup \bigcap_{p \in \text{pred}[n]} D[p]
\]

- Solve by fixpoint iteration
- Start with $(\forall n) \ D[n] = N$ (all nodes in the CFG)
- Watch out for unreachable nodes
Immediate Dominators

**Theorem**

Let $G$ be a connected graph. If $d$ dominates $n$ and $e$ dominates $n$, then either $d$ dominates $e$ or $e$ dominates $d$.

- **Proof:** by contradiction
- **Consequence:** Each node $n \neq s_0$ has one immediate dominator $idom(n)$ such that
  1. $idom(n) \neq n$
  2. $idom(n)$ dominates $n$
  3. $idom(n)$ does not dominate another dominator of $n$
The dominator tree is a graph where the nodes are the nodes of the CFG and there is an edge \((x, y)\) if \(x = idom(y)\).

- **back edge** in CFG: from \(n\) to \(h\) so that \(h\) dominates \(n\)
Natural Loop

The natural loop of a back edge \((n, h)\) where \(h\) dominates \(n\) is the set of nodes \(x\) such that

- \(h\) dominates \(x\)
- exists path from \(x\) to \(n\) not containing \(h\)

\(h\) is the header of this loop.
Nested Loops

Nested Loop

If $A$ and $B$ are loops with headers $a \neq b$ and $b \in A$, then $B \subseteq A$. Loop $B$ is nested within $A$. $B$ is the inner loop.

Loop-nest Tree

1. Compute the dominators of the CFG
2. Compute the dominator tree
3. Find all natural loops with their headers
4. For each loop header $h$ merge all natural loops of $h$ into a single loop $\text{loop}[h]
5. Construct the tree of loop headers such that $h_1$ is above $h_2$ if $h_2 \in \text{loop}[h_1]$}

- Leaves are innermost loops
- Procedure body is pseudo-loop at root of loop-nest tree
A Loop-Nest Tree

```
1
  6, 7, 11, 12

2
  3, 4

5
  10

8
  9
```
Loop optimizations need CFG node before the loop to move code out of the loop

⇒ add preheader node like $P$ in example
Outline

1. Loop Optimizations
2. Dominators
3. Loop-Invariant Computations
4. Induction Variables
5. Array-Bounds Checks
6. Loop Unrolling
Loop-Invariant Computations

- Let \( t \leftarrow a \oplus b \) be in a loop.
- If \( a \) and \( b \) have the same value for each iteration of the loop, then \( t \) always becomes the same value.
- \( \Rightarrow \) repeated computation of the same value
- Goal: Hoist this computation out of the loop
- Approximation needed for “loop invariant”
The definition \( d : t \leftarrow a_1 \oplus a_2 \) is loop-invariant for loop \( L \) if, for each \( a_i \), either

1. \( a_i \) is a constant,
2. all definitions of \( a_i \) that reach \( d \) are outside \( L \), or
3. only one definition of \( a_i \) reaches \( d \) and that definition is loop-invariant.

**Algorithm: Loop-Invariance**

1. Identify all definitions whose operands are constant or from outside the loop
2. Add loop-invariant definitions until fixpoint
Suppose $t \leftarrow a \oplus b$ is loop-invariant.
Can we hoist it out of the loop?

<table>
<thead>
<tr>
<th>L₀</th>
<th>L₀</th>
<th>L₀</th>
<th>L₀</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
</tr>
<tr>
<td>$L₁$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$i \leftarrow i + 1$</td>
<td></td>
<td>$i \leftarrow i + 1$</td>
<td></td>
</tr>
<tr>
<td>$t \leftarrow a \oplus b$</td>
<td></td>
<td>$t \leftarrow a \oplus b$</td>
<td></td>
</tr>
<tr>
<td>$M[i] \leftarrow t$</td>
<td></td>
<td>$M[i] \leftarrow t$</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$L₂$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x \leftarrow t$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Suppose $t \leftarrow a \oplus b$ is loop-invariant.
Can we hoist it out of the loop?

<table>
<thead>
<tr>
<th>$L_0$</th>
<th>$L_0$</th>
<th>$L_0$</th>
<th>$L_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
<td>$t \leftarrow 0$</td>
</tr>
<tr>
<td>$L_1$</td>
<td>$L_1$</td>
<td>$L_1$</td>
<td>$L_1$</td>
</tr>
<tr>
<td>$i \leftarrow i + 1$</td>
<td>if $i \geq N$ goto $L_2$</td>
<td>$i \leftarrow i + 1$</td>
<td>$i \leftarrow i + 1$</td>
</tr>
<tr>
<td>$t \leftarrow a \oplus b$</td>
<td>$t \leftarrow a \oplus b$</td>
<td>$M[i] \leftarrow t$</td>
<td>$M[i] \leftarrow t$</td>
</tr>
<tr>
<td>$M[i] \leftarrow t$</td>
<td>$M[i] \leftarrow t$</td>
<td>goto $L_1$</td>
<td>if $i &lt; N$ goto $L_1$</td>
</tr>
<tr>
<td>if $i &lt; N$ goto $L_1$</td>
<td>$t \leftarrow 0$</td>
<td>$M[j] \leftarrow t$</td>
<td>$M[j] \leftarrow t$</td>
</tr>
<tr>
<td>$L_2$</td>
<td>$L_2$</td>
<td>$L_2$</td>
<td>$L_2$</td>
</tr>
<tr>
<td>$x \leftarrow t$</td>
<td>$x \leftarrow t$</td>
<td>$x \leftarrow t$</td>
<td>$x \leftarrow t$</td>
</tr>
</tbody>
</table>

| yes | no | no | no |
Hoisting

Criteria for hoisting

A loop-invariant definition \( d : t \leftarrow a \oplus b \) can be hoisted to the end of its loop’s preheader if all of the following hold

1. \( d \) dominates all loop exits at which \( t \) is live-out
2. there is only one definition of \( t \) in the loop
3. \( t \) is not live-out at the loop preheader

Attention: arithmetic exceptions, side effects of \( \oplus \)

Condition 1 often prevents hoisting from while loops: transform into repeat-until loops.
Outline

1. Loop Optimizations
2. Dominators
3. Loop-Invariant Computations
4. Induction Variables
5. Array-Bounds Checks
6. Loop Unrolling
Consider

\[
\begin{align*}
  s & \leftarrow 0 \\
  i & \leftarrow 0 \\
  L_1 : & \text{ if } i \geq n \text{ goto } L_2 \\
  j & \leftarrow i \cdot 4 \\
  k & \leftarrow j + a \\
  x & \leftarrow M[k] \\
  s & \leftarrow s + x \\
  i & \leftarrow i + 1 \\
  \text{goto } L_1 \\
  L_2
\end{align*}
\]
Induction Variables

Consider

\[
\begin{align*}
  s & \leftarrow 0 \\
  i & \leftarrow 0 \\
  L_1 : & \text{ if } i \geq n \text{ goto } L_2 \\
  j & \leftarrow i \cdot 4 \\
  k & \leftarrow j + a \\
  x & \leftarrow M[k] \\
  s & \leftarrow s + x \\
  i & \leftarrow i + 1 \\
  \text{goto } L_1 \\
\end{align*}
\]

before

\[
\begin{align*}
  s & \leftarrow 0 \\
  i & \leftarrow 0 \\
  L_1 : & \text{ if } k' \geq c \text{ goto } L_2 \\
  k' & \leftarrow a \\
  b & \leftarrow n \cdot 4 \\
  c & \leftarrow a + b \\
  L_1 : & \text{ if } k' \geq c \text{ goto } L_2 \\
  x & \leftarrow M[k'] \\
  s & \leftarrow s + x \\
  k' & \leftarrow k' + 4 \\
  \text{goto } L_1 \\
\end{align*}
\]

after
**Induction Variables**

- **Induction-variable analysis:**
  identify induction variables and relations among them

- **Strength reduction:**
  replace expensive operation (e.g., multiplication) by cheap operation (e.g., addition)

- **Induction-variable elimination:**
  remove dependent induction variables
A basic induction variable is directly incremented (e.g., $i$)

A derived induction variable is computed from other induction variables (e.g., $j$ and $k$)

- $j = a_j + i \cdot b_j$ with $a_j = 0$ and $b_j = 4$
  $\Rightarrow j$ described by $(i, a_j, b_j)$
- $k = j + c_k$ with loop-invariant $c_k$
  $\Rightarrow k$ described by $(i, a_j + c_k, b_j)$

The basic induction variable $i$ described by $(i, 0, 1)$

A linear induction variable changes by the same amount in every iteration.
Non-linear Induction Variables

\[ s \leftarrow 0 \]

\[ \text{L}_1 : \quad \text{if } s > 0 \text{ goto } \text{L}_2 \]
\[ i \leftarrow i + b \]
\[ j \leftarrow i \cdot 4 \]
\[ x \leftarrow M[j] \]
\[ s \leftarrow s - x \]
\[ \text{goto } \text{L}_1 \]

\[ \text{L}_2 : \quad i \leftarrow i + 1 \]
\[ s \leftarrow s + j \]
\[ \text{if } i < n \text{ goto } \text{L}_1 \]
Non-linear Induction Variables

before

\[
\begin{align*}
  s & \leftarrow 0 \\
L_1: & \quad \text{if } s > 0 \text{ goto } L_2 \\
  i & \leftarrow i + b \\
  j & \leftarrow i \cdot 4 \\
  x & \leftarrow M[j] \\
  s & \leftarrow s - x \\
  \text{goto } L_1 \\
L_2: & \quad i \leftarrow i + 1 \\
  s & \leftarrow s + j \\
  \text{if } i < n \text{ goto } L_1
\end{align*}
\]

after

\[
\begin{align*}
  s & \leftarrow 0 \\
  j' & \leftarrow i \cdot 4 \\
  b' & \leftarrow b \cdot 4 \\
  n' & \leftarrow n \cdot 4 \\
L_1: & \quad \text{if } s > 0 \text{ goto } L_2 \\
  j' & \leftarrow j' + b' \\
  j & \leftarrow j' \\
  x & \leftarrow M[j] \\
  s & \leftarrow s - x \\
  \text{goto } L_1 \\
L_2: & \quad j' \leftarrow j' + 4 \\
  s & \leftarrow s + j \\
  \text{if } j' < n' \text{ goto } L_1
\end{align*}
\]
Detection of Induction Variables

**Basic Induction Variable**
Variable $i$ is a basic induction variable in loop $L$ with header $h$ if all definitions of $i$ in $L$ have the form $i \leftarrow i + c$ or $i \leftarrow i - c$ where $c$ is loop-invariant.

**Derived Induction Variable**
Variable $k$ is a derived ind. var. in the family of $i$ in loop $L$ if

1. There is exactly one definition of $k$ in $L$ of the form $k \leftarrow j \cdot c$ or $k \leftarrow j + d$ where $j$ is an induction variable in the family of $i$ and $c$, $d$ are loop-invariant;

2. if $j$ is a derived induction variable in the family of $i$, then
   - only the definition of $j$ in $L$ reaches (the def of) $k$
   - there is no definition of $i$ on any path between the definition of $j$ and the definition of $k$

3. If $j$ is described by $(i, a, b)$, then $k$ is described by $(i, a \cdot c, b \cdot c)$ or $(i, a + d, b)$, respectively.
Strength Reduction

- Often multiplication is more expensive than addition
  ⇒ Replace the definition $j \leftarrow i \cdot c$ of a derived induction variable by an addition

**Procedure**
- For each derived induction variable $j \sim (i, a, b)$ create new variable $j'$
- After each assignment $i \leftarrow i + c$ to a basic induction variable, create an assignment $j' \leftarrow j' + c \cdot b$
- Replace assignment to $j$ with $j \leftarrow j'$
- Initialize $j' \leftarrow a + i \cdot b$ at end of preheader
Example Strength Reduction
Induction Variables $j \sim (i, 0, 4)$ and $k \sim (i, a, 4)$

before

\[
\begin{align*}
  s & \leftarrow 0 \\
  i & \leftarrow 0 \\

  L_1 : & \text{ if } i \geq n \text{ goto } L_2 \\
  j & \leftarrow i \cdot 4 \\
  k & \leftarrow j + a \\
  x & \leftarrow M[k] \\
  s & \leftarrow s + x \\
  i & \leftarrow i + 1 \\
  & \text{ goto } L_1 \\
  L_2 & 
\end{align*}
\]

after

\[
\begin{align*}
  s & \leftarrow 0 \\
  i & \leftarrow 0 \\
  j' & \leftarrow 0 \\
  k' & \leftarrow a \\

  L_1 : & \text{ if } i \geq n \text{ goto } L_2 \\
  j & \leftarrow j' \\
  k & \leftarrow k' \\
  x & \leftarrow M[k] \\
  s & \leftarrow s + x \\
  i & \leftarrow i + 1 \\
  j' & \leftarrow j' + 4 \\
  k' & \leftarrow k' + 4 \\
  & \text{ goto } L_1 \\
  L_2 & 
\end{align*}
\]
Further apply constant propagation, copy propagation, and dead code elimination

Special case: elimination of induction variables that are
- not used in the loop
- only used in comparisons with loop-invariant variables
- useless

Useless variable

A variable is **useless** in a loop $L$ if
- it is dead at all exits from $L$
- it is only used in its own definitions

Example After removal of $j$, $j'$ is useless
Almost useless variable

A variable is almost useless in loop $L$ if

- it is only used in comparisons against loop-invariant values and in definitions of itself and
- there is another induction variable in the same family that is not useless.

An almost useless variable can be made useless by rewriting the comparisons to use the related induction variable.
Coordinated induction variables

Let $x \sim (i, a_x, b_x)$ and $y \sim (i, a_y, b_y)$ be induction variables. $x$ and $y$ are coordinated if

$$(x - a_x)/b_x = (y - a_y)/b_y$$

throughout the execution of the loop, except during a sequence of statements of the form $z_i \leftarrow z_i + c_i$ where $c_i$ is loop-invariant.
Rewriting Comparisons

Let $j \sim (i, a_j, b_j)$ and $k \sim (i, a_k, b_k)$ be coordinated induction variables. Consider the comparison $k < n$ with $n$ loop-invariant. Using $(j - a_j)/b_j = (k - a_k)/b_k$ the comparison can be rewritten as follows

$$b_k(j - a_j)/b_j + a_k < n$$
$$\Leftrightarrow$$
$$b_k(j - a_j)/b_j < n - a_k$$
$$\Leftrightarrow$$

$$\begin{cases} 
  j < (n - a_k)b_j/b_k + a_j & \text{if } b_j/b_k > 0 \\
  j > (n - a_k)b_j/b_k + a_j & \text{if } b_j/b_k < 0
\end{cases}$$

where the right-hand sides are loop-invariant and their computation can be hoisted to the preheader.
Rewriting Comparisons

Restrictions

1. \((n - a_k)b_j\) must be a multiple of \(b_k\)
2. \(b_j\) and \(b_k\) must both be constants or loop invariants of known sign
Outline

1. Loop Optimizations
2. Dominators
3. Loop-Invariant Computations
4. Induction Variables
5. Array-Bounds Checks
6. Loop Unrolling
Array-Bounds Checks

- Safe programming languages check that the subscript is within the array bounds at each array operation.
- Bounds for an array have the form $0 \leq i < N$ where $N > 0$ is the size of the array.
- Implemented by $i <_u N$ (unsigned comparison).
- Bounds checks redundant in well-written programs ⇒ slowdown.
- For better performance: let the compiler prove which checks are redundant.
- In general, this problem is undecidable.
Conditions for Bounds Check Elimination

1. There is an induction variable $j$ and loop-invariant $u$ used in statement $s_1$ of either of the forms
   - if $j < u$ goto $L_1$ else goto $L_2$
   - if $j \geq u$ goto $L_2$ else goto $L_1$
   - if $u > j$ goto $L_1$ else goto $L_2$
   - if $u \geq j$ goto $L_2$ else goto $L_1$
   where $L_2$ is out of the loop

2. There is a statement $s_2$ of the form
   - if $k <_u n$ goto $L_3$ else goto $L_4$
   where $k$ is an induction variable coordinated with $j$, $n$ is loop-invariant, and $s_1$ dominates $s_2$

3. There is no loop nested within $L$ containing a definition of $k$

4. $k$ increases when $j$ does: $b_j/b_k > 0$
Objective: test in the preheader so that $0 \leq k < n$ everywhere in the loop

Let $k_0$ value of $k$ at end of preheader

Let $\Delta k_1, \ldots, \Delta k_m$ be the loop-invariant values added to $k$ inside the loop

$k \geq 0$ everywhere in the loop if
- $k \geq 0$ in the loop preheader
- $\Delta k_1 \geq 0 \ldots \Delta k_m \geq 0$
Let $\Delta k_1, \ldots, \Delta k_p$ be the set of loop-invariant values added to $k$ on any path between $s_1$ and $s_2$ that does not go through $s_1$.

$k < n$ at $s_2$ if $k < n - (\Delta k_1 + \cdots + \Delta k_p)$ at $s_1$

From $(k - a_k)/b_k = (j - a_j)/b_j$ this test can be rewritten to $j < b_j/b_k(n - (\Delta k_1 + \cdots + \Delta k_p) - a_k) + a_j$

It is sufficient that $u \leq b_j/b_k(n - (\Delta k_1 + \cdots + \Delta k_p) - a_k) + a_j$ because the test $j < u$ dominates the test $k < n$

All parts of this test are loop-invariant!
Array-Bounds Checking Transformation

- Hoist loop-invariants out of the loop
- Copy the loop $L$ to a new loop $L'$ with header label $L'_h$
- Replace the statement "if $k < u \ n$ goto $L'_3$ else goto $L'_4$" by "goto $L'_3$"
- At the end of $L$'s preheader put statements equivalent to
  if $k \geq 0 \land \Delta k_1 \geq 0 \land \cdots \land \Delta k_m \geq 0$
  and $u \leq b_j/b_k(n - (\Delta k_1 + \cdots + \Delta k_p) - a_k) + a_j$
  goto $L'_h$ else goto $L_h$
This condition can be evaluated at compile time if
1. all loop-invariants in the condition are constants; or
2. \( n \) and \( u \) are the same temporary, \( a_k = a_j, b_k = b_j \) and no \( \Delta k \)'s are added to \( k \) between \( s_1 \) and \( s_2 \).

The second case arises for instance with code like this:

```java
int u = a.length;
int i = 0;
while (i<u) {
    sum += a[i];
    i++;
}
```

assuming common subexpression elimination for \( a.length \)

Compile-time evaluation of the condition means to unconditionally use \( L \) or \( L' \) and o delete the other loop

Clean up with elimination of unreachable and dead code
Array-Bounds Checking Generalization

- Comparison of $j \leq u'$ instead of $j < u$
- Loop exit test at end of loop body: A test
  - $s_2 : \text{if } j < u \text{ goto } L_1 \text{ else goto } L_2$

where $L_2$ is out of the loop and $s_2$ dominates all loop back edges; the $\Delta k_i$ are between $s_2$ and any back edge and between the loop header and $s_1$

- Handle the case $b_j/b_k < 0$
- Handle the case where $j$ counts downward where the loop exit tests for $j \geq l$ (a loop-invariant lower bound)

- The increments to the induction variable may be “undisciplined” with non-obvious increment:

```c
while (i<n-1) {
    if (sum<0) { i++; sum += i; i++ } else { i += 2; }
    sum += a[i];
}
```
Loop Unrolling

- For loops with small body, much time is spent incrementing the loop counter and testing the exit condition.
- **Loop unrolling** optimizes this situation by putting more than one copy of the loop body in the loop.
- To unroll a loop $L$ with header $h$ and back edges $s_i \rightarrow h$:
  1. Copy $L$ to a new loop $L'$ with header $h'$ and back edges $s_i' \rightarrow h'$.
  2. Change the back edges in $L$ from $s_i \rightarrow h$ to $s_i \rightarrow h'$.
  3. Change the back edges in $L'$ from $s_i' \rightarrow h'$ to $s_i' \rightarrow h$. 
Loop Unrolling Example (Still Useless)

Before:

\[ L_1 : \]
\[
\begin{align*}
x & \leftarrow M[i] \\
s & \leftarrow s + x \\
i & \leftarrow i + 4 \\
\text{if } i < n \text{ goto } L_1 \text{ else } L_2
\end{align*}
\]

\[ L_2 \]

After:

\[ L_1 : \]
\[
\begin{align*}
x & \leftarrow M[i] \\
s & \leftarrow s + x \\
i & \leftarrow i + 4 \\
\text{if } i < n \text{ goto } L_1 \text{ else } L_2
\end{align*}
\]

\[ L_2 \]

before \hspace{2cm} after
Loop Unrolling Improved

- No gain, yet
- Needed: induction variable $i$ such that every increment $i \leftarrow i + \Delta$ dominates every back edge of the loop
  - each iteration increments $i$ by the sum of the $\Delta$s
  - increments and tests can be moved to the back edges of loop
- In general, a separate epilogue is needed to cover the remaining iterations because the unrolled loop can only do multiple-of-$K$ iterations.
Loop Unrolling Example

\[ L_1 : \quad x \leftarrow M[i] \]
\[ \quad s \leftarrow s + x \]
\[ \quad x \leftarrow M[i + 4] \]
\[ \quad s \leftarrow s + x \]
\[ \quad i \leftarrow i + 8 \]
\[ \text{if } i < n \text{ goto } L_1 \text{ else } L_2 \]

\[ L_2 \]

only even numbers

\[ L_1 : \quad x \leftarrow M[i] \]
\[ \quad s \leftarrow s + x \]
\[ \quad x \leftarrow M[i + 4] \]
\[ \quad s \leftarrow s + x \]
\[ \quad i \leftarrow i + 8 \]
\[ \text{if } i < n - 4 \text{ goto } L_1 \text{ else } L_2' \]

\[ L_2' : \quad \text{if } i < n \text{ goto } L_2 \text{ else } L_3 \]

\[ L_3 \]

with epilogue