The Front End

1. assumes the presence of an infinite number of registers to hold temporary variables.
2. introduces inefficiencies in the source to IR translation.
3. does a direct translation of programmer’s code.
4. does not create pseudo-assembly tuned to the target architecture.
   - Not scheduled for machines with non-unit latency.
   - Not scheduled for wide-issue machines.

The Back End

1. Maps infinite number of virtual registers to finite number of real registers → register allocation
2. Removes inefficiencies introduced by front-end → optimizer
3. Removes inefficiencies introduced by programmer → optimizer
4. Adjusts pseudo-assembly composition and order to match target machine → scheduler

Research and development in back end is growing rapidly.
- EPIC Architectures
- Binary re-optimization
- Runtime optimization
- Optimizations requiring additional hardware support
Optimization

for i := 0 to 10
do a[i] = x;
  ADDI r1 = r0 + 0

LOAD r2 = M[FP + a]
ADDI r3 = r0 + 4
MUL r4 = r3 * r1
ADD r5 = r2 + r4
LOAD r6 = M[FP + x]
STORE M[r5] = r6
ADDI r1 = r1 + 1
BRANCH r1 <= 10, LOOP

Loop invariant code removal...

Register Allocation

for i := 0 to 10
do a[i] = x;
  ADDI r1 = r0 + 0
LOAD r2 = M[FP + a]
ADDI r3 = r0 + 4
LOAD r6 = M[FP + x]

LOAD r2 = M[FP + a]
ADDI r3 = r0 + 4
LOAD r6 = M[FP + x]

LOOP:
MUL r4 = r3 * r1
ADD r5 = r2 + r4
STORE M[r5] = r6
ADDI r1 = r1 + 1
BRANCH r1 <= 10, LOOP

Uses 6 virtual registers, only have 5 real registers...

Scheduling

| 1   | ADDI   | r1 = r0 + 0 |
| 2   | LOAD   | r2 = M[FP + x] |
| 3   | ADDI   | r3 = r0 + 4 |
| 4   | LOAD   | r4 = M[FP + x] |
|     | LOOP:  |               |
| 1   | MUL    | r5 = r3 * r1 |
| 2   | ADD    | r5 = r2 + r5 |
| 3   | STORE  | M[r5] = r4 |
| 5   | ADDI   | r1 = r1 + 1 |
| 6   | BRANCH | r1 <= 10, LOOP |

| 1   | ADDI   | r1 = r0 + 0 |
| 2   | LOAD   | r2 = M[FP + x] |
| 3   | ADDI   | r3 = r0 + 4 |
| 4   | LOAD   | r4 = M[FP + x] |
|     | LOOP:  |               |
| 1   | MUL    | r5 = r3 * r1 |
| 2   | ADD    | r5 = r2 + r5 |
| 3   | ADD    | r5 = r2 + r5 |
| 4   | STORE  | M[r5] = r4 |
| 5   | BRANCH | r1 <= 10, LOOP |

Multiply instruction takes 2 cycles...
Control Flow Analysis

Control Flow Analysis determines how instructions are fetched during execution.

- Control Flow Graph - graph of instructions with directed edge $I_i \rightarrow I_j$ iff $I_j$ can be executed immediately after $I_i$.

Control Flow Analysis Example

```plaintext
r1 = 0

LOOP:
  r1 = r1 + 1
  r2 = r1 & 1
  BRANCH r2 == 0, ODD
  r3 = r3 + 1
  JUMP NEXT

ODD:
  r4 = r4 + 1

NEXT:
  BRANCH r1 <= 10, LOOP
```
Basic Blocks

- **Basic Block** - run of code with single entry and exit.
- Control flow graph of basic blocks more convenient.
- Determine by the following:
  1. Find *leaders*:
     (a) First statement
     (b) Targets of conditional and unconditional branches
     (c) Instructions that follow branches
  2. Basic blocks are leader up to, but not including next leader.

Basic Block Example

```plaintext
r1 = 0

LOOP:
    r1 = r1 + 1
    r2 = r1 & 1
    BRANCH r2 == 0, ODD

    r3 = r3 + 1
    JUMP NEXT

ODD:
    r4 = r4 + 1

NEXT:
    BRANCH r1 <= 10, LOOP
```

Domination Motivation

**Constant Propagation:**

```
          r1 = 4
            ↘
              r2 = r1 + 5
                ↘
                  r2 = r1 + 5
                    ↘
                      r1 = 4
                        ↘
                          r2 = r1 + 5
                ↘
                  r2 = 9
```
Dominator Analysis

- Assume every Control Flow Graph (CFG) has start node $s_0$ with no predecessors.
- Node $d$ dominates node $n$ if every path of directed edges from $s_0$ to $n$ must go through $d$.
- Every node dominates itself.
- Consider:

  ![Dominator Analysis Diagram]

- If $d$ dominates each of the $p_i$, then $d$ dominates $n$.
- If $d$ dominates $n$, then $d$ dominates each of the $p_i$.

Dominator Analysis

- If $d$ dominates each of the $p_i$, then $d$ dominates $n$.
- If $d$ dominates $n$, then $d$ dominates each of the $p_i$.
- $Dom[n] =$ set of nodes that dominate node $n$.
- $N =$ set of all nodes.
- Computation:
  1. $Dom[s_0] = \{s_0\}$.
  2. for $n \in N - \{s_0\}$ do $Dom[n] = N$
  3. while (changes to any $Dom[n]$ occur) do
  4. for $n \in N - \{s_0\}$ do
  5. $Dom[n] = \{n\} \cup (\cap_{p \in prof[n]} Dom[p])$.

Dominator Analysis Example

<table>
<thead>
<tr>
<th>Node</th>
<th>$Dom[n]$</th>
<th>$Dom[n]$</th>
<th>$I Dom[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1-12</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Immediate Dominator/Dominator Tree

- Immediate dominator used in constructing dominator tree.
- Dominator Tree:
  - efficient representation of dominator information
  - used for other types of analysis (e.g. control dependence)
- $s_0$ is root of dominator tree.
- Each node $d$ dominates only its descendants in tree.
- Every node $n$ ($n \neq s_0$) has exactly one immediate dominator $IDom[n]$.
- $IDom[n] \neq n$
- $IDom[n]$ dominates $n$
- $IDom[n]$ does not dominate any other dominator of $n$.
- Last dominator of $n$ on any path from $s_0$ to $n$ is $IDom[n]$.

Immediate Dominator Example

![Dominator Tree Diagram]

<table>
<thead>
<tr>
<th>Node</th>
<th>$Dom[n]$</th>
<th>$IDom[n]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1,2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1,2,3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1,2,4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>1,2,5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>1,2,4,6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1,2,7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1,2,5,8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>1,2,5,8,9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1,2,5,8,9,10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1,2,7,11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>1,2,12</td>
<td></td>
</tr>
</tbody>
</table>

Post Dominator

- Assume every Control Flow Graph (CFG) has exit node $x$ with no successors.
- Node $p$ post-dominates node $n$ if every path of directed edges from $n$ to $x$ must go through $p$.
- Every node post-dominates itself.
- Derivation of post-dominator and immediate post-dominator analysis analogous to dominator and immediate dominator analysis.
- Post-dominators will be useful in computing control dependence.
- Control dependence will be useful in many future optimizations.
Loop Optimization

- Large fraction of execution time is spent in loops.
- Effective loop optimization is extremely important.
- First step in loop optimization → find the loops.
- A loop is a set of CFG nodes $S$ such that:
  1. there exists a header node $h$ in $S$ that dominates all nodes in $S$.
     - there exists a path of directed edges from $h$ to any node in $S$.
     - $h$ is the only node in $S$ with predecessors not in $S$.
  2. from any node in $S$, there exists a path of directed edges to $h$.
- A loop is a single entry, multiple exit region.

Examples of Loops

Back Edges

- Back-edge - flow graph edge from node $n$ to node $h$ such that $h$ dominates $n$
- Each back-edge has a corresponding natural loop.
Natural Loops

- Natural loop of back-edge \((n, h)\):
  - has a loop header \(h\).
  - set of nodes \(X\) such that \(h\) dominates \(x \in X\) and there is a path from \(x\) to \(n\) not containing \(h\).
- A node \(h\) may be header of more than one natural loop.
- Natural loops may be nested.

Loop Optimization

- Compiler should optimize inner loops first.
  - Programs typically spend most time in inner loops.
  - Optimizations may be more effective → loop invariant code removal.
- Convenient to merge natural loops with same header.
- These merged loops are not natural loops.
- Not all cycles in CFG are loops of any kind (more later).

Loop Optimization

**Loop invariant code motion**
- An instruction is loop invariant if it computes the same value in each iteration.
- Invariant code may be hoisted outside the loop.

```
ADDI  r1 = r0 + 0
LOAD  r2 = M[FP + a]
ADDI  r3 = r0 + 4
LOAD  r6 = M[FP + x]

LOOP:
  MUL  r4 = r3 * r1
  ADD  r5 = r2 + r4
  STORE M[r5] = r6
  ADDI  r1 = r1 + 1
  BRANCH r1 <= 10, LOOP
```
Loop Optimization

- **Induction variable analysis and elimination** - \( i \) is an induction variable if only definitions of \( i \) within loop increment/decrement \( i \) by loop-invariant value.

- **Strength reduction** - replace expensive instructions (like multiply) with cheaper ones (like add).

  ```
  ADDI r1 = r0 + 0
  LOAD r2 = M[FP + a]
  ADDI r3 = r0 + 4
  LOAD r6 = M[FP + x]
  
  LOOP:
  MUL r4 = r3 * r1
  ADD r5 = r2 + r4
  STORE M[r5] = r6
  
  ADDI r1 = r1 + 1
  BRANCH r1 <= 10, LOOP
  ```

---

Non-Loop Cycles

- Loops are instances of *reducible* flow graphs.
  - Each cycle of nodes has a unique header.
  - During reduction, entire loop becomes a single node.

- Non-Loops are instances of *irreducible* flow graphs.
  - Analysis and optimization is more efficient on reducible flow graphs.
  - Irreducible flow graphs occur rarely in practice.
    - Use of structured constructs (e.g. if-then, if-then-else, while, repeat, for) leads to reducible flow graphs.
    - Use of goto’s *may* lead to irreducible flow graphs.
  - Irreducible flow graphs can be made reducible by *node-splitting*. 

---