Final Code Generation and Code Optimization
### Translating 3-address code to final code

<table>
<thead>
<tr>
<th>3-Address Code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = A[i] )</td>
<td><code>load i into reg_1</code>&lt;br&gt;<code>la reg_2, A</code>&lt;br&gt;<code>add reg_2, reg_2, reg_1</code>&lt;br&gt;<code>lw reg_2, (reg_2)</code>&lt;br&gt;<code>sw reg_2, x</code></td>
</tr>
<tr>
<td>( x = y+z )</td>
<td><code>load y into reg_1</code>&lt;br&gt;<code>load z into reg_2</code>&lt;br&gt;<code>add reg_3, reg_1, reg_2</code>&lt;br&gt;<code>sw reg_3, x</code></td>
</tr>
<tr>
<td><code>if x &gt;= y goto L</code></td>
<td><code>load x into reg_1</code>&lt;br&gt;<code>load y into reg_2</code>&lt;br&gt;<code>bge reg_1, reg_2, L</code></td>
</tr>
</tbody>
</table>

### Improving Code Quality: Peephole Optimization

- redundant instruction elimination, e.g.:

  
  \[
  \ldots \text{goto} \ L \Rightarrow \ L: \\
  \]

- flow-of-control optimizations, e.g.:

  
  \[
  \ldots \text{goto} \ L_1 \Rightarrow \ \ldots \\
  L_1: \ \text{goto} \ L_2 \quad L_1: \ \text{goto} \ L_2 \\
  \ldots \quad \ldots 
  \]
Improving Code Quality: Peephole Optimization

- algebraic simplifications, e.g.:
  - instructions of the form $x := x + 0$ or $x := x \times 1$ can be eliminated.
  - special case expressions can be simplified, e.g.: $x := 2 \times y$ can be simplified to $x := y + y$.

Improving Code Quality: Code Optimization

- Examine the program to find out about certain properties of interest ("Dataflow Analysis").
- Use this information to change the code in a way that improves performance. ("Code Optimization").
Improving Code Quality: *Code Optimization*

**Code Motion out of Loops**: if a computation inside a loop produces the same result for all iterations (e.g., computing the base address of a local array), it may be possible to move the computation outside the loop.

```plaintext
original code
for ( i=0; i < N; i++) {
    base = &a[0];
    crt = *(base + i);
}

optimized code
base = &a[0];
for ( i=0; i < N; i++) {
    crt = *(base + i);
}
```

**Improving Code Quality: *Code Optimization***

**Common Subexpression Elimination**: if the same expression is computed in many places (e.g., array address computations; results of macro expansion), compute it once and reuse the result.

```plaintext
original code
e1 = *(&a[0]+offset +i);
e2 = *(&a[0]+offset +j);

e1 = *(&a[0]+offset +i);
e2 = *(&a[0]+offset +j);

optimized code
tmp = &a[0]+offset;
e1 = *(tmp +i);
e2 = *(tmp +j);
```
**Improving Code Quality**: *Code Optimization*

Copy Propagation: If we have an intermediate code "copy" instruction `x := y`, replace subsequent uses of `x` by `y` (where possible).

```
y = ...  
x = y;  
b = x / 2;  
```

original code  

```
y = ...  
b = y / 2;  
```

optimized code

---

**Improving Code Quality**: *Code Optimization*

Dead Code Elimination: delete instructions whose results are not used.

```
if (1)  
  x = y;  
else  
  x = z;  
```

original code  

```
x = y;  
```

optimized code
Basics of Code Optimization and Machine Code Generation

• Construct Control Flow Graph (CFG) Representation for the Intermediate Code → Algorithm for building CFG

• Perform Data Flow Analysis to Collect Information Needed for Performing Optimizations → Variable Liveness Analysis

• Perform Optimizations and Generate Machine Code → Algorithm for Register Allocation

Basic Blocks and Flow Graphs

• For program analysis and optimization, it is usually necessary to know control flow relationships between different pieces of code.

• For this, we:
  – group 3-address instructions into basic blocks
  – represent control flow relationships between basic blocks using a control flow graph.
**Example:**

L1:  if $x > y$ goto L0  
    $t_1 = x + 1$  
    $x = t_1$  
L0:  $y = 0$  
    goto L1

**Definition**: A *basic block* is a sequence of consecutive instructions such that:

1. control enters at the beginning;
2. control leaves at the end; and
3. control cannot halt or branch except at the end.

**Identifying basic blocks**:

1. Determine the set of *leaders*, i.e., the first instruction of each basic block:
   (a) The first instruction of the function is a leader.
   (b) Any instruction that is the target of a branch is a leader.
   (c) Any instruction immediately following a (conditional or unconditional) branch is a leader.
2. For each leader, its basic block consists of itself and all instructions upto, but not including, the next leader (or end of function).
Example

/* dot product: \[ \text{prod} = \sum_{i=1}^{N} a[i] \cdot b[i] \] */

<table>
<thead>
<tr>
<th>No.</th>
<th>leader?</th>
<th>Instruction</th>
<th>basic block</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>✓</td>
<td>\text{prod} = 0</td>
<td>1</td>
</tr>
<tr>
<td>(2)</td>
<td></td>
<td>\text{i} = 1</td>
<td>1</td>
</tr>
<tr>
<td>(3)</td>
<td>✓</td>
<td>\text{t1} = 4*i</td>
<td>2</td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td>\text{t2} = a[t1]</td>
<td>2</td>
</tr>
<tr>
<td>(5)</td>
<td></td>
<td>\text{t3} = 4*i</td>
<td>2</td>
</tr>
<tr>
<td>(6)</td>
<td></td>
<td>\text{t4} = b[t3]</td>
<td>2</td>
</tr>
<tr>
<td>(7)</td>
<td></td>
<td>\text{t5} = t2\cdot t4</td>
<td>2</td>
</tr>
<tr>
<td>(8)</td>
<td></td>
<td>\text{t6} = \text{prod} + \text{t5}</td>
<td>2</td>
</tr>
<tr>
<td>(9)</td>
<td></td>
<td>\text{prod} = \text{t6}</td>
<td>2</td>
</tr>
<tr>
<td>(10)</td>
<td></td>
<td>\text{t7} = \text{i+1}</td>
<td>2</td>
</tr>
<tr>
<td>(11)</td>
<td></td>
<td>\text{i} = \text{t7}</td>
<td>2</td>
</tr>
<tr>
<td>(12)</td>
<td></td>
<td>\text{if i} \leq N \text{ goto (3)}</td>
<td>2</td>
</tr>
</tbody>
</table>

Control Flow Graphs

Definition: A flow graph for a function is a directed graph \( G = (V, E) \) whose nodes are the basic blocks of the function, and where \( a \rightarrow b \in E \) iff control can leave \( a \) and immediately enter \( b \).

The distinguished initial node of a flow graph is the basic block whose leader is the first instruction of the function.
Constructing the flow graph of a function:

1. Identify the basic blocks of the function.
2. There is a directed edge from block \( B_1 \) to block \( B_2 \) if
   a. there is a (conditional or unconditional) jump from the last instruction of \( B_1 \) to the first instruction of \( B_2 \); or
   b. \( B_2 \) immediately follows \( B_1 \) in the textual order of the program, and \( B_1 \) does not end in an unconditional jump.

Predecessors and Successors: if there is an edge \( a \rightarrow b \) then \( a \) is a predecessor of \( b \), and \( b \) is a successor of \( a \).

---

Example:

\[ \begin{align*}
L1: \quad & \text{prod} = 0 \\
& i = 1 \\
L2: \quad & t1 = 4*i \\
& t2 = a[t1] \\
& t3 = 4*i \\
& t4 = b[t3] \\
& t5 = t2*t4 \\
& t6 = \text{prod}+t5 \\
& \text{prod} = t6 \\
& t7 = i+1 \\
& i = t7 \\
& \text{if } i \leq N \text{ goto } L2
\end{align*} \]

\[ \begin{align*}
B1: \quad & \text{prod} = 0 \\
& i = 1 \\
B2: \quad & t1 = 4*i \\
& t2 = a[t1] \\
& t3 = 4*i \\
& t4 = b[t2] \\
& t5 = t2*t4 \\
& t6 = \text{prod}+t5 \\
& \text{prod} = t6 \\
& t7 = i+1 \\
& i = t7 \\
& \text{if } i <= N \text{ goto } B2
\end{align*} \]
**Improving Code Quality**: Register Allocation

- **Rationale**
  - A value in a register can be accessed much more efficiently than one in memory
- **Liveness Analysis to build Live Ranges**
  - Identifies durations for which each variable could benefit from using a register
- **Perform Register Allocation**
  - CPU has limited registers → keep frequently used values in registers

---

**Variable Liveness**

**Definition**: A variable is *live* at a point in a program if it *may* be used at a later point before being redefined.

**Example**:

```
\[ x = 1 \]
\[ y = y \cdot x \]
\[ z = z + 1 \]
\[ y = x + y \]
\[ x = x + 1 \]
\[ x = 2 \]
```
**Live Ranges**

**Definition:** A *live range* is an isolated and connected group of basic blocks in which a variable is live.

- Usually, a live range begins at a definition point of a variable and ends at its last uses.
- Different variables may have different live ranges. (⇒ a given basic block may be part of many different live ranges.)
- A given variable may have several different live ranges.
**Global Register Allocation**: considers the entire body of a function or procedure:

- Tries to keep frequently accessed values in registers, esp. across loops.
- Uses loop nesting depth as a guide to frequency of access: variables in the most deeply nested loops are assumed to be accessed the most frequently.

```
read A
D = A+1
read B
D = D+B

read C
D = D+C

print A,D
```

Register Interference Graph:
- **nodes**: live ranges
- **edges**: live ranges overlap
- *k-coloring*, where *k* is the number of registers
## Attempt n-coloring

Color the interference graph using R colors where R is the number of registers.

**Observation:** If there is a node n with < R neighbors, then no matter how the neighbors are colored, there will be at least one color left over to color node n.

Remove n and its edges to get $G'$
Repeat the above process to get $G''$

......

If an empty graph results, R-coloring is possible. Assign colors in reverse of the order in which they were removed.

## Attempt Coloring Contd..

**Input:** Graph G

**Output:** N-coloring of G

While there exists n in G with < N edges do
   Eliminate n & all its edges from G; list n
End while

If G is empty then
   for each node i in list in reverse order do
      Add i & its edges back to G;
      choose color for i
   endfor
End if
Liveness Analysis and Live Range Construction

• Global Analysis
  → Finds what variables are live at basic block boundaries

• Local Analysis
  → Finds what variables are live at all points within basic blocks

• Build Live Ranges
Computing Liveness Information (within a basic block)

Suppose we know which variables are live at the exit from the basic block. Then:

- Scan backwards from the end of the block. At the point immediately before an instruction

\[ I : x := y \oplus z \]

we have:

- \( y \) and \( z \) are live; and
- \( x \) is not live (unless \( x = y \) or \( x = z \)).

Input a \( \rightarrow \) \{ \}
Input b \( \rightarrow \) \{ a \}
y = a + b \( \rightarrow \) \{ a, b \}
y = y - 1 \( \rightarrow \) \{ y, a \}
x = a + 1 \( \rightarrow \) \{ x, y \}
Print \( x + y \) \( \rightarrow \) \{ \}

Computing Liveness Information (dataflow analysis)

We compute \( \text{IN}[B] \) and \( \text{OUT}[B] \), the sets of variables that are live at the beginning and end of each basic block, respectively, in a flow graph, as follows:

**Initialization:**
\[
\begin{align*}
\text{IN}[B] &= \emptyset \text{ for all } B \\
\text{OUT}[B] &= \left\{ \begin{array}{ll}
\text{all globals} & \text{if } B \text{ is an exit block of a function}
\emptyset & \text{otherwise}
\end{array} \right.
\end{align*}
\]

**Propagation:** For each non-exit block \( B \):

- \( \text{OUT}[B] = \bigcup_{B \in \text{successors}(B)} \text{IN}[B'] \)

- \( \text{IN}[B] = (\text{OUT}[B] - \text{KILL}[B]) \cup \text{GEN}[B] \), where

\[
\begin{align*}
\text{GEN}[B] &= \{ v : \text{variable } v \text{ is read before being written} \} \\
\text{KILL}[B] &= \{ v : \text{variable } v \text{ is defined in } B \}
\end{align*}
\]

Since a flow graph may have cycles, we need to iterate this step until there is no change to any \( \text{IN} \) or \( \text{OUT} \) set.
\begin{align*}
\text{IN}[1] &= (\text{OUT}[1] - \text{KILL}[1]) \cup \text{GEN}[1] = \text{OUT}[1] - \{x, y\} \\
\text{OUT}[1] &= \text{IN}[2] \cup \text{IN}[4] \\
\text{IN}[2] &= (\text{OUT}[2] - \text{KILL}[2]) \cup \text{GEN}[2] = \text{OUT}[2] \cup \{x\} \\
\text{OUT}[2] &= \text{IN}[3] \cup \text{IN}[4] \\
\text{IN}[3] &= (\text{OUT}[3] - \text{KILL}[3]) \cup \text{GEN}[3] = \text{OUT}[3] \cup \{x\} \\
\text{OUT}[3] &= \text{IN}[3] \cup \text{IN}[5] \\
\text{IN}[4] &= (\text{OUT}[4] - \text{KILL}[4]) \cup \text{GEN}[4] = \text{OUT}[4] \cup \{y\} \\
\text{OUT}[4] &= \text{IN}[5] \cup \text{IN}[6] \\
\text{IN}[5] &= (\text{OUT}[5] - \text{KILL}[5]) \cup \text{GEN}[5] = \text{OUT}[5] \cup \{x\} \\
\text{OUT}[5] &= \{} \\
\text{IN}[6] &= (\text{OUT}[6] - \text{KILL}[6]) \cup \text{GEN}[6] = (\text{OUT}[6] - \{x\}) \cup \{y\} \\
\text{OUT}[6] &= \text{IN}[7] \cup \text{IN}[8] \\
\text{IN}[7] &= (\text{OUT}[7] - \text{KILL}[7]) \cup \text{GEN}[7] = \text{OUT}[7] \cup \{x\} \\
\text{OUT}[7] &= \{} \\
\text{IN}[8] &= (\text{OUT}[8] - \text{KILL}[8]) \cup \text{GEN}[8] = \text{OUT}[8] \cup \{x\} \\
\text{OUT}[8] &= \text{IN}[8]
\end{align*}

\text{OUT}(b) = \bigcup_{s \in \text{Succ}(b)} \text{IN}(s)

\text{IN}(b) = (\text{OUT}(b) - \text{KILL}(b)) \cup \text{GEN}(b)
\[
\begin{align*}
\text{IN}[1] &= \text{OUT}[1] - \{x,y\} \\
\text{OUT}[1] &= \text{IN}[2] \cup \text{IN}[4] \\
\text{IN}[2] &= \text{OUT}[2] \cup \{x\} \\
\text{OUT}[2] &= \text{IN}[3] \cup \text{IN}[4] \\
\text{IN}[3] &= \text{OUT}[3] \cup \{x\} \\
\text{OUT}[3] &= \text{IN}[3] \cup \text{IN}[5] \\
\text{IN}[4] &= \text{OUT}[4] \cup \{y\} \\
\text{OUT}[4] &= \text{IN}[5] \cup \text{IN}[6] \\
\text{IN}[5] &= \text{OUT}[5] \cup \{x\} \\
\text{OUT}[5] &= \{} \\
\text{IN}[6] &= (\text{OUT}[6] - \{x\}) \cup \{y\} \\
\text{OUT}[6] &= \text{IN}[7] \cup \text{IN}[8] \\
\text{IN}[7] &= \text{OUT}[7] \cup \{x\} \\
\text{OUT}[7] &= \{} \\
\text{IN}[8] &= \text{OUT}[8] \cup \{x\} \\
\text{OUT}[8] &= \text{IN}[8]
\end{align*}
\]
Algorithm for solving data flow equations:
For each block B do
  if B is the exit block then
    OUT[B] = set of global variables
    IN[B] = (OUT[B] – KILL[B]) U GEN[B]
  else
    OUT[B] = IN[B] = { }
  endif
Endfor
DONE = false
While not DONE do
  DONE = true;
  for each B which is not the exit block do
    new = U
    B′ ∈ SUCC(B)
    if new ≠ OUT[B] then
      DONE = false;
      OUT[B] = new;
      IN[B] = (OUT[B] – KILL[B]) U GEN[B]
    endif
  Endfor
Endwhile