Final Code Generation

Input to Code Generator:
- intermediate code program
- symbol table

Output of Code Generator: target program. This can be any of:
- assembly language program
- absolute machine-language program
- relocatable machine-language program
Translating 3-address code to final code

This is almost a macro expansion process. Examples:

<table>
<thead>
<tr>
<th>3-Address Code</th>
<th>MIPS assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>x = A[i]</td>
<td>load i into reg₁</td>
</tr>
<tr>
<td></td>
<td>la reg₂, A</td>
</tr>
<tr>
<td></td>
<td>add reg₂, reg₂, reg₁</td>
</tr>
<tr>
<td></td>
<td>lw reg₂, ( reg₂ )</td>
</tr>
<tr>
<td></td>
<td>sw reg₂, x</td>
</tr>
<tr>
<td>x = y+z</td>
<td>load y into reg₁</td>
</tr>
<tr>
<td></td>
<td>load z into reg₂</td>
</tr>
<tr>
<td></td>
<td>add reg₃, reg₁, reg₂</td>
</tr>
<tr>
<td></td>
<td>sw reg₃, x</td>
</tr>
<tr>
<td>if x &gt;= y goto L</td>
<td>load x into reg₁</td>
</tr>
<tr>
<td></td>
<td>load y into reg₂</td>
</tr>
<tr>
<td></td>
<td>bge reg₁, reg₂, L</td>
</tr>
</tbody>
</table>

The resulting code may not be very efficient, but can be improved via various code optimization techniques.
Improving Code Quality: Peephole Optimization

We can traverse the sequence of intermediate code instructions looking for sequences that can be improved. Examples of such improvements include:

- **redundant instruction elimination, e.g.:**

  ```
  ... goto L  \Rightarrow \ L:  
  L:  \Rightarrow \ ... 
  ...
  ```

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

  ```
  ... goto L1  \Rightarrow \ goto L2  
  ...  \Rightarrow \ ... 
  L1:  goto L2  \Rightarrow \ L1:  goto L2  
  ...  \Rightarrow \ ...
  ```

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


**Improving Code Quality**: *Register Allocation*

A value in a register can be accessed much more efficiently than one in memory, so we should try to keep (frequently used) values in registers.

**Local Register Allocation**: considers only small segments of code ("basic blocks"): 

- If an expression will be used soon after it is evaluated, try to compute it into an unused register.

- If there are no free registers, we can either compute into memory (if addressing modes allow), or free up a register by storing its contents into memory. Choose the register cheapest to store to memory and least likely to be accessed soon.

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

  - **Copy Propagation**: If we have an intermediate code “copy” instruction ‘x := y’, replace subsequent uses of x by y (where possible).

  - **Dead Code Elimination**: delete instructions whose results are not used.
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
    t1 = x+1
    x = t1
L0: y = 0
    goto L1

```
L1: if x>y goto L0
    t1 = x+1
    x = t1
L0: y = 0
    goto L1
```
**Basic Blocks**

**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>( i = 1 )</td>
<td>1</td>
</tr>
<tr>
<td>(3)</td>
<td>√</td>
<td>( t1 = 4 \cdot i )</td>
<td>2</td>
</tr>
<tr>
<td>(4)</td>
<td></td>
<td>( t2 = a[t1] )</td>
<td>2</td>
</tr>
<tr>
<td>(5)</td>
<td></td>
<td>( t3 = 4 \cdot i )</td>
<td>2</td>
</tr>
<tr>
<td>(6)</td>
<td></td>
<td>( t4 = b[t3] )</td>
<td>2</td>
</tr>
<tr>
<td>(7)</td>
<td></td>
<td>( t5 = t2 \cdot t4 )</td>
<td>2</td>
</tr>
<tr>
<td>(8)</td>
<td></td>
<td>( t6 = \text{prod} + t5 )</td>
<td>2</td>
</tr>
<tr>
<td>(9)</td>
<td></td>
<td>( \text{prod} = t6 )</td>
<td>2</td>
</tr>
<tr>
<td>(10)</td>
<td></td>
<td>( t7 = i + 1 )</td>
<td>2</td>
</tr>
<tr>
<td>(11)</td>
<td></td>
<td>( i = t7 )</td>
<td>2</td>
</tr>
<tr>
<td>(12)</td>
<td></td>
<td>if ( i \leq N ) 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 if 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:

L1: \( \text{prod} = 0 \)
    \( i = 1 \)
L2: \( t1 = 4 \times i \)
    \( t2 = a[t1] \)
    \( t3 = 4 \times i \)
    \( t4 = b[t3] \)
    \( t5 = t2 \times t4 \)
    \( t6 = \text{prod} + t5 \)
    \( \text{prod} = t6 \)
    \( t7 = i + 1 \)
    \( i = t7 \)
    \( \text{if } i \leq N \text{ goto L2} \)
Representing Basic Blocks in a Flow Graph

Many different representations are possible, with different tradeoffs. One possibility:

```c
struct bblk_struct {
    int bblno;
    instruction *first_inst;
    instruction *last_inst;
    struct bblk_struct *preds, *succs;
    struct bblk_struct *prev, *next;

    /* + information computed during analysis */
}
```
Global Register Allocation by Graph Coloring

- When a register is needed but all registers are in use, a register has to be freed by storing its contents in memory ("spilling").

Graph coloring is a systematic way of allocating registers and managing spills.

- Graph coloring uses two passes:
  1. Target machine instructions are selected as though there are infinitely many symbolic registers.
  2. Physical registers are assigned to symbolic registers in a way that minimizes the cost of spills.

- **Basic Idea**: Construct an interference graph:
  
  **Nodes**: symbolic registers;
  
  **Edges**: there is an edge between nodes \( m \) and \( n \) if \( m \) and \( n \) are simultaneously “live.”

  Then “color” this graph using \( k \) colors (\( k = \) no. of available registers) such that adjacent nodes don’t get the same color.
Register Interference Graphs

- Nodes \( \sim \) variables or symbolic registers.
- There is an edge between two nodes if they can be simultaneously live.
- If there are \( k \) assignable registers, we try to \( k \)-color the interference graph.

This problem is NP-complete in general. The following heuristic works well in practice:

repeat

if a node \( n \) has fewer than \( k \) neighbors, we can always find a color for it. Delete \( n \) and its edges from the graph and try to \( k \)-color the resulting graph.

until either

- we get the empty graph: in this case, work backwards to produce a \( k \)-coloring of the original graph; or
- we get a graph where every node has \( \geq k \) neighbors: in this case, choose a node to spill to memory, delete this node from the graph, and repeat the above process.

General rule for spilling: avoid introducing code into inner loops.
Example of Register Allocation.

```
read A

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

D = A + 1
Read C
D = D + C

printf A, D
```

Number of registers = 3

```
A
|
B
V
D

B & C

null graph
```

Color in reverse order:

```
A R1
B
C
D R2

A
|
B
V
C
D

R1
R2
```
**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.
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:

Using liveness information:

1. If a variable $x$ is in a register $r$ at a program point, and $x$ is not live, then $r$ can be used for another variable without having to store $x$ to memory.

2. For constructing the interference graph for register allocation by graph coloring.
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
  \[ I : x := y \ op \ z \]

  we have:
  - \( y \) and \( z \) are live; and
  - \( x \) is not live (unless \( x = y \) or \( x = z \)).
Computing Liveness Information (dataflow analysis)

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

Initialization:

\[
IN[B] = \emptyset \text{ for all } B \\
OUT[B] = \begin{cases} 
\text{all globals} & \text{if } B \text{ is an exit block} \\
\emptyset & \text{otherwise}
\end{cases}
\]

Propagation: For each non-exit block $B$:

\[
\begin{align*}
OUT[B] &= \bigcup_{B' \in \text{successors}(B)} IN[B'] \\
IN[B] &= (OUT[B] - KILL[B]) \cup GEN[B], \text{ where} \\
GEN[B] &= \{ v : \text{variable } v \text{ is read before being written} \} \\
KILL[B] &= \{ v : \text{variable } v \text{ is defined in } B \}
\end{align*}
\]

Since a flow graph may have cycles, we need to iterate this step until there is no change to any $IN$ or $OUT$ set.
**LIVE VARIABLES**

\[ \text{IN}(B) \] - set of variables that are live at entry of Basic Block \( B \)

\[ \text{OUT}(B) \] - set of variables that are live at exit of Basic Block \( B \)

\[ \text{GEN}(B) = \{ \nu : \text{variable } \nu \text{ is used before being defined in } B \} \]

\[ \text{KILL}(B) = \{ \nu : \text{variable } \nu \text{ is defined in } B \} \]

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

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

**Algorithm:**

for each block \( B \) do

if \( B \) is the exit block then

\[ \text{OUT}(B) = \text{set of global variables} \]

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

else

\[ \text{OUT}(B) = \text{IN}(B) = \emptyset \]

endif

endfor

DONE = false

while not DONE do

DONE = true

for each \( B \) which is not the exit block do

new = \[ \bigcup_{B' \in \text{succ}(B)} \text{IN}(B') \]

if new \( \neq \text{OUT}(B) \) then

DONE = false

\[ \text{OUT}(B) = \text{new} \]

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

endif

endif

endwhile