Final Code Generation and Code Optimization
Final Code Generation

- **src** → **front end**
- **pgm** → **front end**
- **symbol table**
- **optimizer**
- **final code generator** → **target**
- **intermediate code**
## 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>\n<code>la reg_2, A</code>\n<code>add reg_2, reg_2, reg_1</code>\n<code>lw reg_2, (reg_2)</code>\n<code>sw reg_2, x</code></td>
</tr>
<tr>
<td>( x = y+z )</td>
<td><code>load y into reg_1</code>\n<code>load z into reg_2</code>\n<code>add reg_3, reg_1, reg_2</code>\n<code>sw reg_3, x</code></td>
</tr>
<tr>
<td>( \text{if } x \geq y \text{ goto } L )</td>
<td><code>load x into reg_1</code>\n<code>load y into reg_2</code>\n<code>bge reg_1, reg_2, L</code></td>
</tr>
</tbody>
</table>
Improving Code Quality: Peephole Optimization

- redundant instruction elimination, e.g.:
  
  \[
  \ldots \quad \text{goto } L \quad \Rightarrow \quad L: \quad \ldots \n  \]

- flow-of-control optimizations, e.g.:
  
  \[
  \ldots \quad \text{goto } L1 \quad \Rightarrow \quad \ldots \quad \text{goto } L2
  \\
  L1: \quad \text{goto } L2 \quad \ldots \quad L1: \quad \text{goto } L2
  \\
  \ldots \quad \ldots
  \]
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.

original code

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

optimized code

```c
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.

Original code:

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

Optimized code:

```c
tmp = a[0] + offset;
e1 = *(tmp + i);
e2 = *(tmp + j);
```
Copy Propagation: If we have an intermediate code "copy" instruction `x := y`, replace subsequent uses of `x` by `y` (where possible).

\[
\begin{align*}
y &= \\
x &= y; \\
b &= x / 2;
\end{align*}
\]

original code

\[
\begin{align*}
y &= \\
b &= y / 2;
\end{align*}
\]

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  
    \[ t1 = x+1 \]  
    \[ x = t1 \]  
L0: \[ y = 0 \]  
    goto L1  

⇒

L1: if $x > y$ goto L0  
    \[ t1 = x+1 \]  
    \[ x = t1 \]  
L0: \[ y = 0 \]  
    goto L1
**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 up to, but not including, the next leader (or end of function).
/* dot product: \[ prod = \sum_{i=1}^{N} a[i] \times b[i] \] */

<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>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*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*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*t4</td>
<td>2</td>
</tr>
<tr>
<td>(8)</td>
<td></td>
<td>t6 = prod+t5</td>
<td>2</td>
</tr>
<tr>
<td>(9)</td>
<td></td>
<td>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: \( t1 = 4*i \)
    \( t2 = a[t1] \)
    \( t3 = 4*i \)
    \( t4 = b[t3] \)
    \( t5 = t2*t4 \)
    \( t6 = \text{prod}+t5 \)
    \( \text{prod} = t6 \)
    \( t7 = i+1 \)
    \( i = t7 \)
    if \( i \leq N \) goto L2

B1:
  \( \text{prod} = 0 \)
  \( i = 1 \)

B2:
  \( t1 = 4*i \)
  \( t2 = a[t1] \)
  \( t3 = 4*i \)
  \( t4 = b[t3] \)
  \( t5 = t2*t4 \)
  \( t6 = \text{prod}+t5 \)
  \( \text{prod} = t6 \)
  \( t7 = i+1 \)
  \( i = t7 \)
  if \( i \leq N \) goto B2
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 $\Rightarrow$ 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 - 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.
  \[ \Rightarrow \text{a given basic block may be part of many different live ranges.} \]
- A given variable may have several different live ranges.
2 Live Ranges of x

Start

define x
define y

use x

use x

use y

use x

End

1 Live Range of y

Start

define x
define y

use y

use x

use y

define x

use x

use x

End
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.
D = A + 1
read B
D = D + B

D = A + 1
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.
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 the
    for each node i in list in reverse order do
        Add i & its edges back to G;
        choose color for i
    endfor
End if
A kept in Memory

Spill A in Memory

{empty graph}
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

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

  we have:

  -- \( x \) is not live
  -- \( y \) and \( z \) are live

Live Before \( I = ( \text{Live After } I - \{ x \} ) \cup \{ y, z \} \)
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:

- \(\text{IN}[B]\)
- \(\text{OUT}[B]\)
- \(\text{GEN}[B] = \{X\}\)
  - B makes X live on entry to B
- \(\text{KILL}[B] = \{Y\}\)
  - B makes Y not live on entry to B
Initialization:
\[ \text{IN}[B] = \emptyset \text{ for all } B \]
\[ \text{OUT}[B] = \begin{cases} \text{all globals} & \text{if } B \text{ is an exit block} \\ \emptyset & \text{otherwise} \end{cases} \]

of a function
other than main()

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

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

- \[ \text{IN}[B] = (\text{OUT}[B] - \text{KILL}[B]) \cup \text{GEN}[B], \text{ where} \]
  \[ \text{GEN}[B] = \{ v : \text{variable } v \text{ is read before being written} \} \]
  \[ \text{KILL}[B] = \{ v : \text{variable } v \text{ is defined in } B \} \]

Since a flow graph may have cycles, we need to iterate this step until there is no change to any IN or OUT set.
define x
define y

use x
use x
use y
use x
use y

Start

GEN[1] = -
KILL[1] = x,y

GEN[2] = x
KILL[2] = -

GEN[3] = x
KILL[3] = -

GEN[4] = y
KILL[4] = -

GEN[5] = x
KILL[5] = -

GEN[6] = y
KILL[6] = x

GEN[7] = x
KILL[7] = -

GEN[8] = x
KILL[8] = -

End
OUT[5] = {}
OUT[7] = {}
OUT[8] = IN[8]

\[\text{OUT}(b) = \bigcup_{s \text{ in } \text{Succ}(b)} \text{IN}(s)\]
\[\text{IN}(b) = (\text{OUT}(b) − \text{KILL}(b)) \bigcup \text{GEN}(b)\]
IN[1] = OUT[1] - {x,y}  
OUT[5] = {}  
OUT[7] = {}  
IN[8] = OUT[8] U {x}  
OUT[8] = IN[8]
\[
\text{OUT}(b) = \bigcup_{s \text{ in } \text{Succ}(b)} \text{IN}(s)
\]
\[
\text{IN}(b) = (\text{OUT}(b) - \text{KILL}(b)) \bigcup \text{GEN}(b)
\]

```
\[
\begin{align*}
\text{GEN}[1] &= \text{KILL}[1] = x, y \\
\text{GEN}[2] &= x \\
\text{GEN}[3] &= x \\
\text{GEN}[4] &= y \\
\text{GEN}[5] &= x \\
\text{GEN}[6] &= y \\
\text{GEN}[7] &= x \\
\text{GEN}[8] &= x
\end{align*}
\]
```

```
\[
\begin{align*}
\text{KILL}[1] &= x, y \\
\text{KILL}[2] &= - \\
\text{KILL}[3] &= - \\
\text{KILL}[4] &= - \\
\text{KILL}[5] &= - \\
\text{KILL}[6] &= x \\
\text{KILL}[7] &= - \\
\text{KILL}[8] &= - \\
\end{align*}
\]
```

```
\[
\begin{align*}
\text{IN}(b) &= \text{OUT}(b) - \text{KILL}(b) + \text{GEN}(b) \\
\text{OUT}(b) &= \bigcup_{s \text{ in } \text{Succ}(b)} \text{IN}(s)
\end{align*}
\]
```

```
\[
\begin{align*}
\text{IN}(b) &= (\text{OUT}(b) - \text{KILL}(b)) \bigcup \text{GEN}(b) \\
\end{align*}
\]
```
2 Live Ranges of x

Start

define x
define y

use x

use x

use x

use y

define x
use y

use x

use x

use x

use x

End

1 Live Range of y

Start

define x
define y

use x

use x

use x

use y

define x
use y

use x

use x

use x

End
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 = ∪
             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
Sample Problems for Review
1. Input X
2. Input Y
3. X=X+Y
4. If Z<0 go to 7
5. X=X+1
6. Go to 8
7. X=X-1
8. Y=Y+1
9. T=X+Y
10. If Z==T go to 4
11. Output Z

1. Input X
2. Input Y
3. X=X+Y
4. If Z<0 go to 7
5. X=X+1
6. Go to 8
7. X=X-1
8. Y=Y+1
9. T=X+Y
10. If Z==T go to 4
11. Output Z
Input X
Input Y
If X < Y go to L1
Z = X + Y
X = Y
Go to L2

L1: Z = X - Y
X = Y
Go to L2

L2: Output X
Output Y
Output Z

Input X
Input Y
If X < Y go to L1

L1: Z = X - Y
X = Y
Go to L2

L2: Output X
Output Y
Output Z

Input X
Input Y
If X < Y go to L1
Z = X + Y
X = Y
Go to L2
If $X < Y$ go to L1

L1:
- $Z = X - Y$
- $X = Y$

OUT[2] = {X, Y, Z}

L2:
- Output X
- Output Y
- Output Z

OUT[4] = {}
LIVE RANGES OF X, Y and Z
LIVE RANGES OF X, Y and Z
INTERFERENCE GRAPH
REGISTER ALLOCATION: R1, R2, R3

REMOVE DEGREE<3
X1, X2, Z; Y

COLOR IN REVERSE ORDER
Y  R1
Z  R2
X2  R3
X1  R2 or R3
REGISTER ALLOCATION: R1, R2

REMOVE DEGREE<2
X1; spill Y; X2, Z

COLOR IN REVERSE ORDER
Z    R1
X2    R2
X1    R1 or R2
0 Main () {
    Int a, b;
    
    1 F() {
        Int a, c;
        2 Call G();
    }
    
    1 G() {
        Int a, e;
        2 H() {
            Int a, d;
            3 Call F();
        }
        
        2 Call H();
    }
    
    1 Call F();
}

Main \rightarrow F \rightarrow G \rightarrow H \rightarrow F

c-d \rightarrow 1-1 2-1 2-2 3-1

Access Links

Control Links

Access b in H traverse 3-1=2 links then at offset at of 1 find b
CONSTRUCT

if x < y then
  <otherstatements>
elseif a > b then
  <otherstatements>
  ...
elseif c == d then
  <otherstatements>
else
  <otherstatements>
endif

Question:
Provide SEMANTIC RULES that generate code and finally place it in attribute <S>.code
if $x < y$ then
  <otherstatements>
elseif $a > b$ then
  <otherstatements>
elseif $c == d$ then
  <otherstatements>
else
  <otherstatements>
endif
……

INTERMEDIATE CODE

if $x < y$ go to L1
  go to L2
L1: <otherstatements>
  go to exitL
L2: If $a > b$ go to L3
  go to L4
L3: <otherstatements>
  go to exitL
L4: if $c == d$ go to L5
  go to L6
L5: <otherstatements>
  go to exitL
L6: <otherstatements>
exitL: ……. 
if <condt> then <otherstatements> <rest>

elseif <condt> then <otherstatements> <rest>

else <otherstatements> endif
if x < y go to L1
  go to L2
L1: <otherstatements>
  go to exitL
L2: <rest>

elseif <condt> then <otherstatements> <rest>

id relop id

elseif <condt> then <otherstatements> <rest>

id relop id

else <otherstatements> endif

if a > b go to L3
  go to L4
L3: <otherstatements>
  go to exitL
L4: <rest>

if c==d go to L5
  go to L6
L5: <otherstatements>
  go to exitL
L6: <rest>

L6: <otherstatements>

exitL: .......
if x < y go to L1
   go to L2
L1: <otherstatements>
   go to exitL
L2: <rest>
L2: If a > b go to L3
   go to L4
L3: <otherstatements>
   go to exitL
L4: <rest>
L4: if c==d go to L5
   go to L6
L5: <otherstatements>
   go to exitL
L6: <otherstatements>
exitL: ....
\[ \text{condt} \rightarrow \text{id}_1 \text{ relop id}_2 \ \{
\text{truelabel} = \text{newlabel}();
\text{condt}.\text{false}label = \text{newlabel}();
\text{condt}.\text{code} = \text{gen}(\text{"if id}_1\text{.place "relop" id}_2\text{.place "go to" truelabel} \\
\quad || \text{gen}(\text{"go to" condt}.\text{false}label) || \text{gen(truelabel:"})
\}\]

\[ \text{S} \rightarrow \text{if } \text{condt} \text{ then } \text{otherstatements} \\
\quad \{
\text{rest}.\text{if}false\text{label} = \text{condt}.\text{false}\text{label};
\text{rest}.\text{i}exit = \text{newlabel}();
\text{rest}.\text{i}code = \text{condt}.\text{code} || \text{otherstatements}.\text{code} || \\
\quad \text{gen}(\text{"go to" rest}.\text{i}exit)
\}
\quad \text{rest} \{ \text{S}.\text{code} = \text{rest}.\text{s}code \} \]
\[ \text{<rest}_1\text{>} \rightarrow \text{elsif <condt> then <otherstatements> } \]
\[
\{ \\
\quad \text{<rest}_2\text{>.icode} = \text{<rest}_1\text{>.icode} \mid \mid \text{gen(<rest}_1\text{>.ifalselabel "::") \mid \mid} \\
\quad \text{<condt>\text{.code} \mid \mid <otherstatements>\text{.code} \mid \mid \text{gen("go to" <rest}_1\text{>.iexit);}} \\
\quad \text{<rest}_2\text{>.ifalselabel} = \text{<condt>\text{.falselabel;}} \\
\quad \text{<rest}_2\text{.iexit} = \text{<rest}_1\text{.iexit}} \\
\}
\]
\[
\text{<rest}_2\text{>} \{ \text{<rest}_1\text{>.scode} = \text{<rest}_2\text{.scode} \}
\]

\[ \text{<rest}_1\text{>} \rightarrow \text{else <otherstatements> endif } \]
\[
\{ \\
\quad \text{<rest}_1\text{>.scode} = \text{<rest}_1\text{.icode} \mid \mid \text{gen(<rest}_1\text{>.ifalselabel "::")} \\
\quad \mid \mid <\text{otherstatements}\text{.code} \mid \mid \text{gen(<rest}_1\text{.iexit "::")}} \\
\}
\]