Code Generation
Code Generator

- Severe requirements imposed
  - Output must be correct and high quality
  - Code generator should run efficiently
- Generating optimal code is undecidable
  - Must rely on heuristic
  - Choice of heuristic very important
- Details are dependent on target language and operating system
- Certain generic issues are inherent in the design of basically all code generators

Input to Code Generator

- The input to the code generator consists of:
  - Intermediate code produced by the front end (and perhaps optimizer)
    - Remember that intermediate code can come in many forms
    - We are concentrating on three-address code but several techniques apply to other possibilities as well
  - Information in the symbol table (used to determine runtime addresses of data objects)
- The code generator typically assumes that:
  - Input is free of errors
  - Type checking has taken place and necessary type-conversion operators have already been inserted
Output of Code Generator

- The target program is the output of the code generator
- Can take a variety of forms
  - Absolute machine language
  - Relocatable machine language
    - Can compile subprograms separately
    - Added expense of linking and loading
  - Assembly language
    - Makes task of code generation simpler
    - Added cost of assembly phase

Memory Management

- Compiler must map names in source code to addresses of data objects at run-time
  - Done cooperatively by front-end and code generator
  - Code generator uses information in symbol table
- If machine code is being generated:
  - Labels in three-address statements need to be converted to addresses of instructions
  - Process is analogous to backpatching
Instruction Selection

- Obviously this depends on the nature of the instruction set
- If efficiency is not a concern, instruction selection is straightforward
  - For each type of three-address statement, there is a code skeleton outlines target code
  - Example, \( x := y + z \), where \( x, y, \) and \( z \) are statically located, can be translated as:
    
    \[
    \begin{align*}
    \text{lw} & \; \$t0, \; y \\
    \text{lw} & \; \$t1, \; z \\
    \text{add} & \; \$t2, \; \$t0, \; \$t1 \\
    \text{sw} & \; \$t2, \; x \\
    \end{align*}
    \]

Instruction Selection

- Often, the straight-forward technique produces poor code:
  
  \[
  \begin{align*}
  a & := b + c \\
  d & := a + e
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{lw} & \; \$t0, \; b \\
  \text{lw} & \; \$t1, \; c \\
  \text{add} & \; \$t2, \; \$t0, \; \$t1 \\
  \text{sw} & \; \$t2, \; a \\
  \text{lw} & \; \$t0, \; a \\
  \text{li} & \; \$t1, \; 1 \\
  \text{add} & \; \$t2, \; \$t0, \; \$t1 \\
  \text{sw} & \; \$t2, \; d
  \end{align*}
  \]

- A naïve translation may lead to correct but inefficient target code:
  
  \[
  \begin{align*}
  a & := a + 1
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{lw} & \; \$t0, \; a \\
  \text{li} & \; \$t1, \; 1 \\
  \text{add} & \; \$t2, \; \$t0, \; \$t1 \\
  \text{sw} & \; \$t2, \; a
  \end{align*}
  \]
Registers

- Some registers have to be used by the system:
  - Base registers
  - Stack pointer
  - Current frame pointer
  - Global area pointer

Register Allocation

- Instructions are usually faster if operands are in registers instead of memory
- Efficient utilization of registers is important in generating good code
- Register allocation selects the set of variables that will reside in registers
- A register assignment phase picks the specific register in which a variable resides
- Finding an optimal assignment of registers to variables is difficult
  - The problem is NP-complete
  - Certain machines require register-pairs for certain operators and results
Items Kept in Registers

- Value frequently used
  - Even if they cross basic block boundaries
- Loop values
  - Especially inner loop values

Usage Count

- How can you decide what to keep in a register?
- Usage count
  \[ \sum_{blocks \, B \, in \, L} use(x, B) + 2 \times live(x, B) \]
  - Where \( use(x, B) \) is the number of times \( x \) is used in \( B \) prior to a definition
  - \( live(x, B) \) is 1 if \( x \) is live on exit from \( B \) and is assigned a value in \( B \), and is 0 otherwise
Graph Coloring

- Intermediate language is generated assuming an unlimited number of symbolic registers
- Register allocation phase maps the unlimited registers onto the real registers
- Uses a uniform approach
- Idiosyncrasies of the machine are entered in a uniform manner in the interference graph

Graph Coloring

- All registers are considered to be part of a uniform pool
- All computations compete on an equal basis for these registers
- Attempt to eliminate loads and stores
- As much as possible is done in registers
  - Automatic scalars
  - Parameters
Graph Coloring

- It is the job of the code generator and optimizer to take advantage of the unlimited number of registers allowed in the intermediate code.
- If it is not possible to map the unlimited number of registers onto the real registers, spill code must be added to the intermediate code.
- Graph coloring does a better job than hand coders can do.

Notation and Definitions

- **Graph coloring**
  - The assignment of a color to each of its nodes in such a manner that if two nodes are adjacent, they have different colors.
- A coloring of a graph is said to be an *n*-coloring if it does not use more than *n* different colors.
- **Chromatic number** of a graph
  - Minimal number of colors, in any, of its coloring
    - The least *n* for which there is an *n*-coloring.
- Determining if G is *n*-colorable is NP-complete.
Notation and Definitions

- **Use-def chain**
  - Linkage of uses and definitions of variable in a program

- A value $A$ is *live* at point $P$ in a program if $A$ could be used along some path in the flow graph starting at point $P$.

- **Basic block**
  - Sequence of consecutive statements that may be entered only at the beginning, and when entered, are executed in sequence without halt or possibility of a branch (except at the end of the basis block)

Notation and Definitions

- **Flow graph**
  - Directed graph such that the successor relationships of basic blocks are portrayed

- **Register interference graph**
  - Data structure used in this procedure
  - There is an edge in the graph if the live tracks of variables overlap

- Coalescing or combining nodes
  - Eliminate source to target copy operations
  - Also known as propagation in the literature
Approach for Graph Coloring

- For each procedure (function) in the source program, an interference graph is constructed
- Nodes in the interference graph stand for machine registers and for all computations in the procedure that reside in machine registers
- Edges stand for register interference

Approach for Graph Coloring

- If the chromatic number of graph is less than or equal to the number of machine registers, register allocation has been achieved
  - The register assigned to a computation is one that has a different color than its neighbors
- If the chromatic number is greater, spill code must be introduced to store and reload registers in order to obtain a program whose chromatic number is less
Graph Coloring Algorithm

```c
int ColorGraph ( graph G, nodes N )
{
    if ( N is empty )
        return TRUE;
    if ( none of the nodes in N have fewer than #colors neighbors in G )
        return FALSE;
    Select register R from nodes that have less then #colors neighbors;
    B = Colorgraph( ( G – those interferences involving R ), ( N – R ) );
    if ( B )
        Coloring(R) = an arbitrary color selected from those colors that have
                not been assigned to neighbors of R;
        return B;
} /* ColorGraph */
```

Interference Graph Example

```
live-in: k, j
  g := mem[j+12]
  h := k-1
  f := g*h
  e := mem[j+8]
  m := mem[j+16]
  b := mem(f)
  c := e+8
  d := c
  k := m+4
  j := b
live-out: d, k, j
```

Source Code
## Interference Graph Example

<table>
<thead>
<tr>
<th>Stack</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>2</td>
</tr>
<tr>
<td>f</td>
<td>3</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
</tr>
<tr>
<td>m</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>g</td>
<td>3</td>
</tr>
<tr>
<td>h</td>
<td>2</td>
</tr>
<tr>
<td>j</td>
<td>1</td>
</tr>
</tbody>
</table>

## Procedures and Functions

- The activation records can be implemented in two different ways:
  - Static allocation
    - Fortran
  - Stack allocation
    - C, C++, Java
- What are the advantages and disadvantage of the two different methods?
Procedures and Functions

- When a procedure or function is called, there are a number of hidden actions that occur:
  - Setting up activation record
  - Transmission of parameters
  - Creation of linkages for nonlocal referencing
  - Other “housekeeping” activities

Hidden Activities

- Prologue code
  - Inserted at the start of the code for the procedure/function

- Epilogue code
  - Inserted at end of procedure/function
  - Return results
  - Free storage for the activation record
Procedure/Function Calls

- Look up procedure/function in symbol table
- Expressions for the actual parameters must be evaluated
- Ensure each actual parameter conforms to the formal parameter
- Generate code to push the value of the parameter on the run-time stack

Procedure/Function Returns

- Generate epilogue code
- Generate the return instruction
  - For functions, this involves moving the return value to the appropriate location
Creating the Activation Records

- Built in two different places
  - Procedure/function calls
    - Push values of actual parameters on stack
  - Beginning of procedure/function call
    - Push the return address on stack
    - Push the frame pointer on stack
    - Set location of new frame pointer
    - Allocate space for local variables

Reclaiming the Activation Record

- Epilogue code should:
  - Put the old value of frame pointer back into the proper register
  - Put the return address into the proper register
  - Deallocation the space for the parameters
    - Increment the value of the stack pointer by the size of the parameter area
Basic Blocks

- A basic block is a sequence of statements such that:
  - Flow of control enters at the beginning of the basic block
  - Flow of control leaves at the end of the basic block
  - No possibility of halting or branching except at end

- A name is *live* at a given point if its value will be used again in the program

- Each basic block has a first statement known as the *leader* of the basic block

Partitioning Code into Basic Blocks

- Algorithm must determine all leaders:
  - The first statement is a leader
  - Any statement that is the target of a conditional or unconditional goto is a leader
  - Any statement immediately following a goto or unconditional goto is a leader

- A basic block:
  - Starts with a leader
  - Includes all statements up to but not including the next leader
Basic Block Example

begin
    prod := 0;
i := 1;
do
    begin
        prod := prod + a[i] * b[i]
    end
while i <= 20
end

(1) prod := 0
(2) i := 1
(3) t1 := 4 * i
(4) t2 := a[t1]
(5) t3 := 4 * i
(6) t4 := b[t3]
(7) t5 := t2 * t4
(8) t6 := prod + t5
(9) prod := t6
(10) t7 := i + 1
(11) i := t7
(12) if i <= 20 goto 3
(13) ...

Transformations on Basic Blocks

- A basic block computes a set of expressions
  - The expressions are the values of names that are live on exit from the block
  - Two basic blocks are equivalent if they compute the same set of expressions
- Certain transformations can be applied without changing the computed expressions of a block
  - An optimizer uses such transformations to improve running time or space requirements of a program
  - Two important classes of local transformations:
    - Structure-preserving transformations
    - Algebraic transformations
Example Transformations

- **Algebraic Transformations:**
  - Statements such as \( x := x + 0 \) or \( x := x \times 1 \) can be safely removed
  - Statement \( x := y^2 \) can be safely changed to \( x := y \times y \)

- **Common subexpression elimination:**
  - Suppose the statement \( x := y + z \) appears in a basic block and \( x \) is dead (i.e. never used again)
  - This statement can be safely removed from the code

- **Dead-code elimination**
  - Suppose the statement \( x := y + z \) appears in a basic block and \( x \) is dead (i.e. never used again)
  - This statement can be safely removed from the code

Example Transformations

- **Renaming of Temporary variables**
  - Suppose the statement \( t := b + c \) appears in a basic block and \( t \) is a temporary
    - We can safely rename all instances of this \( t \) to \( u \), where \( u \) is a new temporary
    - A normal-form block uses a new temporary for every statement that defines a temporary

- **Interchange of two independent adjacent statements**
  - Suppose we have a block with two adjacent statements:
    - \( t1 := b + c \)
    - \( t2 := x + y \)
  - If \( t1 \) is distinct from \( x \) and \( y \) and \( t2 \) is distinct from \( b \) and \( c \), we can safely change the order of these two statements
Flow Graphs

- A graphical representation of basic blocks that is useful for optimization
- Nodes represent computation
  - Each node represents a single basic blocks
  - One node is distinguished as initial
- Edges represent flow-of-control
  - There is an edge from $B_1$ to $B_2$ if and only if $B_2$ can immediately follow $B_1$ in some execution sequence:
    - True if there is an conditional or unconditional jump from the last statement of $B_1$ to the first statement of $B_2$
    - True if $B_2$ immediately follows $B_1$ and $B_1$ does not end with an unconditional jump
  - $B_1$ is a predecessor of $B_2$, $B_2$ is a successor of $B_1$

Representations of Flow Graphs

- A basic block can be represented by a record consisting of:
  - A count of the number of quadruples in the block
  - A pointer to the leader of the block
  - The list of predecessors and successors
- Alternative is to use linked list of quadruples
- Explicit references to quadruple numbers in jump statements can cause problems:
  - Quadruples can be moved during optimization
  - Better to have jumps point to blocks
Loops and Flow Graphs

A loop in a collection of nodes in a flow graph such that:
- All nodes in the collection are *strongly connected*
  - There must be a path of length one or more connecting any two nodes in the collection
- The collection of nodes must have a unique *entry* node
  - The only way to reach a node in the loop from a node outside the loop is to go through entry
- A loop that contains no other loops is called an *inner* loop

Next-Use Information

The use of a name in a three-address statement is defined as follows:
- Suppose statement $i$ assigns a value to $x$
- Suppose statement $j$ has $x$ as an operand
- We say that $j$ uses the value of $x$ computed at $i$ if:
  - There is a path through which control can flow from statement $i$ to statement $j$
  - This path has no intervening assignments to $x$
- For each three-address statement that is of the form $x := y \text{ op } z$
  - We want to determine the next uses of $x$, $y$, and $z$
  - For now, we do not consider next uses outside of the current basic block
Determine Next-Use

- Scan each basic block from end to beginning
  - At start, record, if known, which names are live on exit from that block
  - Otherwise, assume all non-temporaries are live
  - If algorithm allows certain temporaries to be live on exit from block, consider them live as well
- Whenever reaching a three address statement at line i with the form x := y op z
  - Attach statement i to information in symbol table regarding the next use and liveliness of x, y, and z
  - Next, in symbol table, set x to "not live" and "no next use"
  - Then set y and z to "live" and the next uses of y and z to i
- A similar approach is taken for unary operators

Reusing Temporaries

- It is sometimes convenient during optimization for every temporary to have its own name
- Space can be saved, however, by reusing temporary names
- Two temporaries can be packed into the same location if they are not live simultaneously
A Simple Code Generator

- The book describes a simple code generator
- Generates target code for a sequence of three-address statements
- Considers statements one at a time:
  - Remembers if any of the operands are currently in registers
  - Takes advantage of that if possible
- Assumes that for each operator in three-address code there is a corresponding target-language operator
- Also assumes that computed results can be left in registers as long as possible, storing them only:
  - If their register is needed for another computation
  - Just before a jump, labeled statement, or procedure call

Register Allocation

- We can use graph coloring, as previously described, to assign temporaries to specific registers
Code-Generation Algorithm

- For each three-address statement $x := y \text{ op } z$ perform the following actions:
  - Get value in register (done using graph coloring)
  - If the value of $y$ is not already in $L$, generate the instruction MOV $y'$, $L$ to place a copy of $y$ in $L$
  - Generate the instruction OP $z'$, $L$ where $z'$ is the chosen current location of $z$
- The actions for unary operators are analogous
- A simple assignment statement is a special case

Code Generation Example

- Consider the statement $d := (a - b) + (a - c) + (a - c)$
- This may be translated into the following three-address code:
  
  $t := a - b$
  $u := a - c$
  $v := t + u$
  $d := v + u$

  - Assume that $d$ is live at end of block
  - Assume that $a$, $b$, and $c$ are always in memory
  - Assume that $t$, $u$, and $v$, being temporaries, are not in memory unless explicitly stored with a sw instruction
Code Generation Example

<table>
<thead>
<tr>
<th>Statements</th>
<th>Generated Code</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>registers empty</td>
</tr>
<tr>
<td>$t := a - b$</td>
<td>lw $t0, a</td>
<td>$t0 contains a</td>
</tr>
<tr>
<td></td>
<td>lw $t1, b</td>
<td>$t1 contains t</td>
</tr>
<tr>
<td></td>
<td>sub $t1, $t0, $t1</td>
<td></td>
</tr>
<tr>
<td>$u := a - c$</td>
<td>lw $t2, c</td>
<td>$t0 contains u</td>
</tr>
<tr>
<td></td>
<td>sub $t0, $t0, $t2</td>
<td>$t1 contains t</td>
</tr>
<tr>
<td>$v := t + u$</td>
<td>add $t1, $t1, $t0</td>
<td>$t0 contains u</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$t1 contains v</td>
</tr>
<tr>
<td>$d := v + u$</td>
<td>add $t0, $t1, $t0</td>
<td>$t0 contains d</td>
</tr>
<tr>
<td></td>
<td>sw $t0, d</td>
<td></td>
</tr>
</tbody>
</table>

Directed Acyclic Graphs (Dags)

- A dag for a basic block is a directed acyclic graph such that:
  - Leaves represent the initial values of name
    - Labeled by unique identifiers, either variable names or constants
    - Operator applied to name determines if l-value or r-value is needed; usually it is r-value
  - Interior nodes are labeled by an operators
  - Nodes are optionally also given a sequence of identifiers (identifiers that are assigned its value)
- Useful for implementing transformations on and determining information about a basic block
Using Dags

- A dag can be automatically constructed from code using a simple algorithm
- Several useful pieces of information can be obtained:
  - Common subexpressions
  - Which identifiers have their values used in the block
  - Which statements compute values that could be used outside the block
- Can be used to reconstruct a simplified list of quadruples
  - Can evaluate interior nodes of dag in any order that is a topological sort (all children before parent)
  - Heuristics exist to find good orders
  - If dag is a tree, a simple algorithm exists to give optimal order (order leading to shortest instruction sequence)