Last Time

• Code Templates
  - Ideal code sequences for each language feature (what we'd like to be generating)
  - There is more than one sequence for each language feature
  - **Operand conditions** select which template fits each situation best
  - ISTs encode all of the templates and their operand conditions into a single decision tree

• Subscripting and Field Selection
  - Code templates for **arrays** and **subscripts** are critical to the quality of the generated code
  - Templates dependent on scope and kind of array variable
    - static, automatic or reference parameter

• Generating Code for Expressions
  - **Operand Stack** stores operands to be tested in operand conditions
  - Choices on operands encode IST in S/SL code
  - **Temporary** mechanism keeps track of registers (temporaries) used to hold run time intermediate results (ES)
Next

• Next we look at generating code for statements
  • If statements and conditional execution
  • Loops
  • Case and switch statements

• To understand how to implement decisions in if statements, while loops and so on, we need first to look at the representation of Boolean expressions (conditions) as code templates
Boolean Expressions

• Two ways to represent boolean expressions

  **Data Model** - represent result of Boolean conditions as an actual bit value (actually a byte)

  **Control Model** - represent result of Boolean conditions as the state of execution (where we are in the code)

• For both we will be using the example:

  \[ a = b \ or \ c = d \]

  which has a postfix representation of:

  \[ a \ b = c \ d = \ or \]
Data Model of Boolean Expressions

• In the **data model**, we use a byte (or a bit in some circumstances) to explicitly represent the result of each comparison and relational expression as a data value.

• Typically, **true** is represented by 1, **false** by 0 (or sometimes false by 0, true by nonzero, or false by 0, true by hex FF, or so on).

• The bit manipulation instructions (**and**, **or**, etc.) of the target machine are used to implement the values.
Data Model of Boolean Expressions

<table>
<thead>
<tr>
<th>Postfix</th>
<th>x86 machine code*</th>
<th>(*simplified)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b =</td>
<td>cmpl a,b</td>
<td></td>
</tr>
<tr>
<td></td>
<td>je L1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movb $0,%eax</td>
<td>0 = false</td>
</tr>
<tr>
<td></td>
<td>jmp L2</td>
<td></td>
</tr>
<tr>
<td>L1:</td>
<td>movb $1,%eax</td>
<td>1 = true</td>
</tr>
<tr>
<td>L2:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c d =</td>
<td>cmpl c,d</td>
<td></td>
</tr>
<tr>
<td></td>
<td>je L3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>movb $0,%ebx</td>
<td>0 = false</td>
</tr>
<tr>
<td></td>
<td>jmp L4</td>
<td></td>
</tr>
<tr>
<td>L3:</td>
<td>movb $1,%ebx</td>
<td>1 = true</td>
</tr>
<tr>
<td>L4:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>orb %ebx,%eax</td>
<td>bitwise or</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if a=b or c=d then</td>
<td>cmpb %eax,$0</td>
<td>if false</td>
</tr>
<tr>
<td></td>
<td>je else</td>
<td>go to else</td>
</tr>
</tbody>
</table>
Data Model of Boolean Expressions

- The data model is very general, easy to generate, and works well, but it does not make very good code

- For example, if an assembly language programmer were coding

  ```
  if a=b or c=d then
  ...
  ```

- She would typically come up with something more like this:

  ```
  cmpl a,b
  je then
  cmpl c,d
  je then
  jmp else
  ```

- *Five* instructions instead of the *13* on the previous slide
Control Model of Boolean Expressions

• The difference is that, in the assembly language solution, the value of the relational conditions is never actually represented as data - rather, it is implicit in the flow of the code

  cmpl a,b
  je then
  cmpl c,d  // here a≠b, therefore a=b or c=d is true
  je then    // only if c=d
  jmp else   // here a≠b and c≠d,
  // therefore a=b or c=d is false

then: ...  // here a=b or c=d

else: ...   // a≠b and c≠d

• If we are in the true branch of the code, then the result was true, and if we are in the false branch of code, then the condition was false

• What happens if the value is assigned to a boolean variable?  
  -> then we convert the state to a data value  
    (but most often it is used only as an if or while condition)
Control Model of Boolean Expressions

• Infix operator "clues" for logical operators are needed in the postfix expressions if we are to implement code templates for the control model, because there is branching code that must be generated at that point

• **Example**: For the condition

  
  a = b or c = d

  Which in postfix is

  
  a b = c d = or

• The **or** operator comes too late for use to emit the appropriate branching code to save the state in the code flow - so we have extra **infix** operators from the semantic phase to help us:

  
  a b = infixor c d = or

• These appear at the **original** location of the operator in the source code - the **postfix** operator is added as well, giving the code generator opportunities to generate code at both points
Condition Descriptors

• In order to implement code templates for the control model, we need to be able to keep track of which branch labels are used in our code to represent the true state, and which the false state.

• A special form of data descriptors called condition descriptors are used to represent conditions and their labels - these have three parts:
  - a branch condition (remembers the condition tested, e.g., equal, not equal, greater than, etc.)
  - a list of true labels (labels for the place we jump to if the condition is true)
  - a list of false labels (labels for the place we jump to if the condition is false)

• We need a list of labels because while we generate, we must invent labels to branch to for each comparison before we know where that really is or what it will be the same as.

• Then as we discover that two or more represent the same state (e.g., the then part of an if), we combine them into one list - then they all label the same place when emitted.
Condition Descriptors

• Example:

  \texttt{cmpl a,b} \quad \text{locally generate code for } a=b \\
  \texttt{je L1} \quad \text{don't know where we're going yet, so guess a new label L1}

  \ldots

  \texttt{cmpl c,d} \quad \text{locally generate code for } c=d \\
  \texttt{je L2} \quad \text{don't know where we're going yet, so guess a new label L2}

  \ldots

  

  \text{Boolean or tells us L1 and L2 are in the same set}

  \ldots

  \texttt{L1:} \quad \text{eventually find out true part should go here, so whole list of equivalent labels goes here}

  \texttt{L2:}
Condition Descriptors

• Condition descriptors take the form:

\[(\text{condition}, \text{true labels}, \text{false labels})\]

• As we generate code, we use them as operands in the Operand Stack to keep track of the conditions and labels associated with the branches we have generated to represent the result of the corresponding Boolean expressions

• Example:

<table>
<thead>
<tr>
<th></th>
<th>Operand Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{a} \text{b} = \text{cmpl} \text{a,b}</td>
<td>(equal, \varepsilon, \varepsilon)</td>
</tr>
<tr>
<td>\text{infixor} \text{je} \text{L1}</td>
<td>(\varepsilon, \text{L1}, \varepsilon)</td>
</tr>
<tr>
<td>\text{c} \text{d} = \text{cmpl} \text{c,d}</td>
<td>(equal, \varepsilon, \varepsilon)</td>
</tr>
<tr>
<td>or no code</td>
<td>(equal, \text{L1}, \varepsilon)</td>
</tr>
</tbody>
</table>
What the Heck is Going On?

Operand Stack

\[
a \ b = \ cmp \ a,b \quad (equal, \ v, \ v) \quad \text{keeping track of fact that comparison is already done, but we need to test equal}
\]

\[
\text{infixor} \ \ beq \ L1 \quad (v, L1, v) \quad \text{test equal now done, keep track of where it goes}
\]

\[
c \ d = \ cmp \ c,d \quad (equal, v, v) \quad (v, L1, v) \quad \text{2nd comparison is done, but we need to test equal}
\]

\[
\text{or} \quad \no\ code \quad (equal, L1, v) \quad \text{combine or'ed conditions - still need to check equal on 2nd comparison, and L1 already represents true}
\]
Conditions - Data vs Control Model

• If there are no side effects in expressions, then the two models are equivalent.

• However, if you have a Boolean or other function that has a side effect called in the condition, then they are not the same.

• In our example \((a = b \text{ or } c = d)\), in the control model, if \(a\) and \(b\) are equal, the second condition is never checked.

• If either \(c\) or \(d\) has a side effect (e.g. is a function call that changes global variables), this may not be desirable (depends on the language).

• PT has no functions, so it uses the control model, which is much more efficient.
Conditions - Data vs Control Model

• In PT and other control model implementations, if the first part of an or returns true, then the second part is not evaluated, speeding things up (conditional evaluation).

• Similarly if the first part of an and condition evaluates to false, then the second part is not evaluated, speeding things up.

• In order to understand how to implement all conditions, we only need to generate code for the boolean operators or and not.

• All other logical operators can be represented using these two operators (remember DeMorgan?)

\[ A \text{ and } B \equiv \text{ not } ((\text{not } A) \text{ or } (\text{not } B)) \]

• Some languages, e.g. C, provide the programmer with both models.

\[ A \& B \quad (data \ model) \]
\[ A \&\& B \quad (control \ model, \ with \ conditional \ evaluation) \]
Control Model Templates

• Therefore we have three abstract machine instructions for which we need code templates to manipulate conditions: \texttt{infixor, or} and \texttt{not}

• The \texttt{or} operation has the general postfix form: \texttt{c1 infixor c2 or}

• Suppose that the condition descriptors resulting from \texttt{c1} and \texttt{c2} are:
  \[ c_1 = (c_1, t_1, f_1) \]
  \[ c_2 = (c_2, t_2, f_2) \]

• Then the code template for the \texttt{infixor} … \texttt{or} language feature is:

\[
\begin{align*}
\textit{c1} & \quad \textit{code for c1} & (c_1, t_1, f_1) \\
\textit{infixor} & \quad \textit{bc1 L_n} & (\varepsilon, t_1+L_n, \varepsilon) \\
& \quad \textit{f1:} & (\varepsilon, t_1+L_n, \varepsilon) \\
\textit{c2} & \quad \textit{code for c2} & (c_2, t_2, f_2) \\
\textit{or} & \quad \textit{(c2, t_1+t_2+L_n, f_2)} & (c_2, t_1+t_2+L_n, f_2)
\end{align*}
\]
Control Model Templates

• The general not operation

\[ c_1 \text{ not} \]

has the simple template:

\[ c_1 \quad (\text{code for } c_1) \quad (c_1, t_1, f_1) \]

\[ \text{not} \quad (\neg c_1, f_1, t_1) \]

(Note that the label sets have been swapped!)

• Thus, the not operator requires no code to be emitted - and as a result, the strategy of not implementing the and operator and instead using DeMorgan's laws has no runtime cost

• That is, we get the best code for and without actually having to implement it!
Summary

- In order to understand how to generate code for statements, we first need to understand how to handle Boolean conditions.

- Two models for evaluation of Boolean expressions:
  - the *data model* implements Boolean conditions explicitly, as 1/0 results - simple but not very good code.
  - the *control model* represents Boolean conditions in the control state (code location) in a set of conditional branches - more complicated, but much better code.

- Control model implemented using special operands called *condition descriptors* to represent control state.

- Next week:
  - Quiz #4 (Text ch. 17-19 inclusive, lectures 23-26)
  - USAT & TA course evaluations
  - Finish code generation, then: Virtual machine *interpreters*