Intermediate Representation

Prof. James L. Frankel
Harvard University

Version of 3:37 PM 28-Nov-2023

Benefits

- Compiling *i* languages for *j* instruction sets
- Direct code generation would require *i*j* compilers
- Compiling *i* languages to IR code and generating machine code from IR for *j* instruction sets requires *i+j* compiler components

Three Address Code

- Three Address Code
  - Up to two sources
  - One destination
  - One operation
- i = j+k*l could be represented as
  - r0 = k * l
  - r1 = j + r0
  - i = r1
- Or as
  - r0 = k * l
  - i = j + r0
- Think of the r*n* names as representing the names of either temporaries or registers
  - We will name our registers with the letter **r** followed by an integer (counting from **0**)

Quadruples Representation

- Three address code can be represented by quadruples
- Our result/destination will always be in the second field in a quadruple
  - (operation, result/destinationAddress, operand_1, operand_2)
- The three address code for
  - r0 = k * l
  - r1 = j + r0
  - i = r1
- Can be represented in quadruples as
  - (multSignedWord, r0, k, l) # we will **not** allow identifiers to appear in
  - (addSignedWord, r1, j, r0) # these IR nodes
  - (assignWord, i, r1) # assignWord will **not** be one of our IR opcodes
- Not all quadruples have four fields

Triples Representation

- Three address code can be represented by triples
- The result/destination in triples is not encoded as an operand
    - Instead the result of a triple is expressed by referring to the triple indirectly
    - (operation, operand$_1$, operand$_2$)
    - Modification of a user's object is always explicit
        - (assignWord, destinationAddress, operand)
- The three address code for
    - r0 = k * l
    - r1 = j + r0
    - i = r1
- Can be represented in triples as
    - 0: (multSignedWord, k, l)
    - 1: (addSignedWord, j, (0))
    - 2: (assignWord, i, (1)) # assignWord will **not** be one of our IR opcodes
- Not all triples have three fields

Our Implementation

- We will be generating quadruples
- Both signed and unsigned variants of operations in our quadruples will exist (when they are different)
  - For example, a comparison for equality does *not* have signed variants
- Store quadruples in a ***doubly-linked*** linked list

- Traverse the AST and generate IR code at each node that represents executable operations
- Recursively generate IR code for subtrees
- To each AST node that has a value, add an attribute that is the name of the temporary/register that contains that value

Generating Code for Subtrees

- We’d like to generate IR code for subtrees in the AST without knowing how the result is going to be used
- We don’t know if the result of evaluating a subtree will be used as an lvalue or as an rvalue
    - If an operator **can** result in an lvalue, return the result as an lvalue *and* remember that it is an lvalue
        - We can always derive an rvalue from an lvalue by loading the value at the address given by the lvalue
    - If an operator **cannot** result in an lvalue, return the result as an rvalue *and* remember that it is an rvalue
- We remember the lvalue-ness or rvalue-ness of the derived code with a tag in the AST subtree as we generate IR code
- We will represent an lvalue by the address of the referent

Deriving an rvalue from an lvalue (1 of 3)

- Some operators require lvalues and others allow both rvalues and lvalues
- If we have an lvalue and want an rvalue, derive the rvalue by performing a **load** operation

- A reference to a name (identifier) – other than an enumeration constant – can be either an lvalue or an rvalue
  - §7.3.1: The name of a variable declared to be of arithmetic, pointer, enumeration, structure, or union type evaluates to an object of that type; the name is an lvalue expression
- For i = j, where all variables are ints, we generate the following IR code
- Reference to name i generates an lvalue
  - (addressOf, r0, i)   # r0 is an lvalue for i; r0 -> i
- Reference to name j generates an lvalue
  - (addressOf, r1, j)   # r1 is an lvalue for j; r1 -> j

Deriving an rvalue from an lvalue (2 of 3)

- Assignment operator
  - §7.9 (Assignment Expressions) Operands: Every assignment operator requires a modifiable lvalue as its left operand and modifies that lvalue by storing a new value into it.
  - The right operand is *treated* as an rvalue (*i.e.,* if the right operand is *not* already an rvalue, then a **load** is generated to convert the lvalue into an rvalue)
  - §7.9 Result: The result of an assignment expression is never an lvalue.
- Because the assignment operator requires its left operand to be an lvalue and because r0 is an lvalue already, it does not need to be changed
- But, the assignment operator treats its right operand as an rvalue and r1 is an lvalue; therefore we need to **load** the right operand by performing
  - (loadWord, r2, r1)   # r2 is an rvalue for j; r2 <- j

Deriving an rvalue from an lvalue (3 of 3)

- Now we have an lvalue in r0 for the lhs and an rvalue in r2 for the rhs of the assignment operator
- We can now generate code for the assignment operator
  - (storeWord, r0, r2)   # i <- j

- To summarize, for i = j, where all variables are ints, we generate the following IR code:
  - (addressOf, r0, i)   # r0 is an lvalue for i; r0 -> i
  - (addressOf, r1, j)   # r1 is an lvalue for j; r1 -> j
  - (loadWord, r2, r1)   # r2 is an rvalue for j; r2 <- j
  - (storeWord, r0, r2)   # i <- j

References to Identifiers in IR Code

- The references to identifiers in IR code are not simply the names used in the input C program
- The identifiers referred to in IR code are represented by pointers to the appropriate symbol table entry for the relevant identifier
  - This is required in order to uniquely identify the specific identifier
  - The specific identifier will include information about the storage duration of that identifier (is it static or local (auto)?)
  - Later, we will add additional information to identifier entries in symbol tables

Details of addressOf IR

- When executing (addressOf, r0, i)
  - i should be a designator for a user's variable
  - r0 should be a register name
  - (addressOf, r0, i) loads register r0 with the address of user variable i

- Addresses can be taken only of names that refer to memory locations (*i.e.,* user identifiers)
- Addresses cannot be taken of registers

Details of loadWord and storeWord IRs

- When executing (loadWord, r2, r1)
    - Both operands (r2 and r1) should be register names
    - r1 should contain the address for a word in memory that will be read
    - r2 will be loaded with the value that was present in the specified word in memory
    - (loadWord, r2, r1) reads a word from memory at the location given by register r1 and stores that word into register r2
- When executing (storeWord, r0, r2)
    - Both operands (r0 and r2) should be register names
    - r2 should contain the value to be stored in memory
    - r0 should contain the address for a word in memory that will be written
    - (storeWord, r0, r2) writes the value in register r2 into a word in memory at the location given by register r0

Operator results: lvalue or rvalue

- For i = j+k*l, where all variables are ints, are the intermediate results able to be lvalues or rvalues?
  - Keep in mind that a reference to a name (identifier) – other than an enumeration constant – can be either an lvalue or an rvalue
- Names (Identifier Reference)
  - §7.3.1: The name of a variable declared to be of arithmetic, pointer, enumeration, structure, or union type evaluates to an object of that type; the name is an lvalue expression
- Multiplication
  - §7.6.1: The result is not an lvalue
- Addition
  - §7.6.2: The result is not an lvalue
- Simple Assignment
  - §7.9 (Assignment Expressions): The result of an assignment expression is never an lvalue
  - §7.9.1 (Simple Assignment): The result is not an lvalue

Operator operands: lvalue or rvalue

- For i = j+k*l, where all variables are ints, are the intermediate operands required to be lvalues or rvalues?
- Multiplication
  - Both operands are treated as rvalues (*i.e.,* if the operand is *not* already an rvalue, then a **load** is generated to convert the lvalue into an rvalue)
- Addition
  - Both operands are treated as rvalues (*i.e.,* if the operand is *not* already an rvalue, then a **load** is generated to convert the lvalue into an rvalue)
- Assignment
  - §7.9: Every assignment operator requires a modifiable lvalue as its left operand and modifies that lvalue by storing a new value into it
  - The right operand is *treated* as an rvalue (*i.e.,* if the right operand is *not* already an rvalue, then a **load** is generated to convert the lvalue into an rvalue)

Evaluation Example

- Quadruples code generated for i = j+k*l, where all variables are ints
  - (addressOf, r0, i) # r0 is an lvalue for i; r0 -> i
  - (addressOf, r1, j) # r1 is an lvalue for j; r1 -> j
  - (addressOf, r2, k) # r2 is an lvalue for k; r2 -> k
  - (addressOf, r3, l) # r3 is an lvalue for l; r3 -> l
  - (loadWord, r4, r2) # r4 is an rvalue for k; r4 <- k
  - (loadWord, r5, r3) # r5 is an rvalue for l; r5 <- l
  - (multSignedWord, r6, r4, r5) # r6 is an rvalue for k*l; r6 <- k*l
  - (loadWord, r7, r1) # r7 is an rvalue for j; r7 <- j
  - (addSignedWord, r8, r7, r6) # r8 is an rvalue for j+k*l; r8 <- j+k*l
  - (storeWord, r0, r8) # i <- j+k*l

Quadruple Operation Codes (1 of 5)

- addressOf

- loadWord
- loadHalfWord
- loadSignedHalfWord
- loadByte
- loadSignedByte
- storeWord
- storeHalfWord
- storeByte

Quadruple Operation Codes (2 of 5)

- multSignedWord
- multUnsignedWord
- divSignedWord
- divUnsignedWord
- remSignedWord
- remUnsignedWord

- addSignedWord
- addUnsignedWord
- subSignedWord
- subUnsignedWord

- leftShiftWord
- rightShiftSignedWord
- rightShiftUnsignedWord

Quadruple Operation Codes (3 of 5)

- ltSignedWord
- ltUnsignedWord
- leSignedWord
- leUnsignedWord
- geSignedWord
- geUnsignedWord
- gtSignedWord
- gtUnsignedWord
- eqWord
- neWord

Quadruple Operation Codes (4 of 5)

- bitwiseAndWord
- bitwiseXorWord
- bitwiseOrWord

Quadruple Operation Codes (5 of 5)

- unaryMinus
- unaryLogicalNegation
- unaryBitwiseNegation

- etc.

Numbering Temporaries/Registers

- When generating IR code, assume an *infinite* number of temporaries/registers
  - Every *new* IR result should be new temporary
  - There will be some cases when a temporary will need to be an IR result in more than one IR instruction
    - This will happen only with the &&, ||, and ? : operators
- We’ll deal with the constrained number of real registers in MIPS later in the code generation process

IR Code for Integral Literal

- For

  ```
  int i;
  i = 5;
  ```

- A reference to an integral literal will use constInt
- §7.3.2: Except for string constants, a literal expression is never an lvalue
- IR code

  ```
  (addressOf, r0, i)        # r0 is an lvalue for i; r0 -> i
  (constInt, r1, 5)         # r1 is an rvalue for 5; r1 <- 5
  (storeWord, r0, r1)       # i <- 5
  ```

IR Code for Address (unary prefix &) Operator

- For
  int i, *p;
  i = 5;
  p = &i;
- §7.5.6: The operand of & must be either a function designator or an lvalue designating an object. The usual conversions are not applied to the operand of the & operator, and its result is never an lvalue.
- For the Address operator, no code needs to be generated to convert the lvalue operand into an rvalue result
- IR code

• (addressOf, r0, i)	# r0 is an lvalue for i; r0 -> i
• (constInt, r1, 5)	# r1 is an rvalue for 5; r1 <- 5
• (storeWord, r0, r1)	# i <- 5
• (addressOf, r2, p)	# r2 is an lvalue for p; r2 -> p
• (addressOf, r3, i)	# r3 is an lvalue for i; r3 -> i
•	# r3 is an rvalue for &i
• (storeWord, r2, r3)	# p -> i (or, equivalently, p <- &i)

IR Code for Indirection/Dereference (unary prefix *) Operator

- For
  ```
  int i, j, *p;
  i = 5;
  p = &i;
  j = *p;
  ```
- §7.5.7: If the pointer points to an object, then the result is an lvalue referring to the object
- The operand is *treated* as an rvalue (*i.e.,* if the operand is *not* already an rvalue, then a **load** is generated to convert the lvalue into an rvalue)
- For the Indirection/Dereference operator, no code needs to be generated to convert the rvalue operand into an lvalue result
- IR code

(addressOf, r0, i)	# r0 is an lvalue for i; r0 -> i
(constInt, r1, 5)	# r1 is an rvalue for 5; r1 <- 5
(storeWord, r0, r1)	# i <- 5
(addressOf, r2, p)	# r2 is an lvalue for p; r2 -> p
(addressOf, r3, i)	# r3 is an lvalue for i; r3 -> i
	# r3 is an rvalue for &i
(storeWord, r2, r3)	# p -> i (or, equivalently, p <- &i)
(addressOf, r4, j)	# r4 is an lvalue for j; r4 -> j
(addressOf, r5, p)	# r5 is an lvalue for p; r5 -> p
(loadWord, r6, r5)	# r6 is an rvalue for p; r6 <- p
	# r6 is an lvalue for *p
(loadWord, r7, r6)	# r7 is an rvalue for *p; r7 <- *p
(storeWord, r4, r7)	# j <- *p

IR Code for Addition of an Integer to a Pointer

- For

  int i, j, *p;
  i = *(p+j);

- IR code /* In my comments, read “->” as ***points to*** and read “<-” as ***gets*** */
  - (addressOf, r0, i) # r0 is an lvalue for i; r0 -> i
  - (addressOf, r1, p) # r1 is an lvalue for p; r1 -> p
  - (addressOf, r2, j) # r2 is an lvalue for j; r2 -> j
  - (loadWord, r3, r1) # r3 is an rvalue for p; r3 <- p
  - (loadWord, r4, r2) # r4 is an rvalue for j; r4 <- j
  - (constInt, r5, 4) # r5 is an rvalue for 4 (*i.e.,* sizeof(int)); r5 <- 4
  - (multSignedWord, r6, r4, r5) # r6 is an rvalue for j*4; r6 <- j*4
  - (addSignedWord, r7, r3, r6) # r7 is an rvalue for (p+j); r7 <- (p+j)
  - # r7 is an lvalue for *(p+j)
  - (loadWord, r8, r7) # r8 is an rvalue for *(p+j); r8 <- *(p+j)
  - (storeWord, r0, r8) # i <- *(p+j)

IR Code for Subscript Operator with Arrays

- For
  ```
  int i, j, a[100];
  i = a[j];
  ```
- The above is syntactic sugar for
  ```
  i = *(a+j);        /* All subscript operators should be translated into * and + in the AST */
  ```
- Type checking will convert the above into
  ```
  i = *((int *)a+j);
  ```
- For a cast from an array to a pointer, no code needs to be generated to convert the lvalue operand into an rvalue result
- IR code /* In my comments, read "->" as ***points to*** and read "<-" as ***gets*** */
  ```
  • (addressOf, r0, i)                # r0 is an lvalue for i; r0 -> i
  • (addressOf, r1, a)                # r1 is an lvalue for a; r1 -> a[0]
  •                                   # r1 is an rvalue for (int *)a; r1 -> a[0]
  • (addressOf, r2, j)                # r2 is an lvalue for j; r2 -> j
  • (loadWord, r3, r2)                # r3 is an rvalue for j; r3 <- j
  • (constInt, r4, 4)                 # r4 is an rvalue for 4 (i.e., sizeof(int)); r4 <- 4
  • (multSignedWord, r5, r3, r4)      # r5 is an rvalue for j*4; r5 <- j*4
  • (addSignedWord, r6, r1, r5)       # r6 is an rvalue for ((int *)a+j); r6 <- ((int *)a+j)
  •                                   # r6 is an lvalue for *((int *)a+j)
  • (loadWord, r7, r6)                # r7 is an rvalue for *((int *)a+j); r7 <- *((int *)a+j)
  • (storeWord, r0, r7)               # i <- *((int *)a+j)
  ```

IR Code for Accessing Multidimensional Arrays (1 of 5)

- For,

```
int matrix[5][6];         # matrix is a 5-by-6 array of int
matrix[1][3] = 99;
```

IR Code for Accessing Multidimensional Arrays (2 of 5)

- In C, remember that a multidimensional array is stored in memory in ***row-major order***, so the elements of matrix are stored as:

```
matrix[0][0]
matrix[0][1]
matrix[0][2]
matrix[0][3]
matrix[0][4]
matrix[0][5]
matrix[1][0]
matrix[1][1]
matrix[1][2]
matrix[1][3]
matrix[1][4]
matrix[1][5]
matrix[2][0]
...
```

IR Code for Accessing Multidimensional Arrays (3 of 5)

- Starting with our example,

```
int matrix[5][6];         # matrix is a 5-by-6 array of int
matrix[1][3] = 99;
```

- is syntactic sugar for,

```
int matrix[5][6];         # matrix is a 5-by-6 array of int
*((*(matrix+1))+3) = 99;
```

IR Code for Accessing Multidimensional Arrays (4 of 5)

- Once we apply type checking to,

```
int matrix[5][6];         # matrix is a 5-by-6 array of int
*((*(matrix+1))+3) = 99;
```

- we have,

```
int matrix[5][6];         # matrix is a 5-by-6 array of int
*( (* int)( *((pointer to array of 6 ints)matrix+1) )+3) = 99;
```

IR Code for Accessing Multidimensional Arrays (5 of 5)

- Generating code for,

```
int matrix[5][6];        # matrix is a 5-by-6 array of int
*( (* int)( *((pointer to array of 6 ints)matrix+1) )+3) = 99;
```

- the result is,

```
(addressOf, r0, matrix)     # r0 is an lvalue for matrix; r0 -> matrix[0][0]
                            # r0 is an rvalue for (pointer to array of 6 ints)matrix; r0 -> matrix[0][0]
(constInt, r1, 1)           # r1 is an rvalue for 1; r1 <- 1
(constInt, r2, 24)      # r2 is an rvalue for 24 (6*sizeof(int)) (i.e., sizeof(array of 6 int)); r2 <- 24
(multSignedWord, r3, r1, r2)         # r3 is an rvalue for 1*24; r3 <- 1*24
(addSignedWord, r4, r0, r3) # r4 is an rvalue for ((pointer to array of 6 ints)matrix+1)
                            # r4 is an lvalue for *((pointer to array of 6 ints)matrix+1)
                            # r4 is an rvalue for (* int)(*((pointer to array of 6 ints)matrix+1))
(constInt, r5, 3)           # r5 is an rvalue for 3; r5 <- 3
(constInt, r6, 4)           # r6 is an rvalue for 4 (i.e., sizeof(int)); r6 <- 4
(multSignedWord, r7, r5, r6)         # r7 is an rvalue for 3*4; r7 <- 3*4
(addSignedWord, r8, r4, r7) # r8 is an rvalue for ( (* int)( *((pointer to array of 6 ints)matrix+1) )+3)
                           # r8 is an lvalue for *( (* int)( *((pointer to array of 6 ints)matrix+1) )+3)
(constInt, r9, 99)          # r9 is an rvalue for 99; r9 <- 99
(storeWord, r8, r9)         # *( (* int)( *((pointer to array of 6 ints)matrix+1) )+3) <- 99
```

String Constants

- Each string constant is stored in memory with an appended null character, '\0'
  - This can be accomplished in SPIM by using the .asciiz assembler directive
- A string constant consisting of *n* characters has type array of *n*+1 char
- §2.7.4: If a string constant appears anywhere except as an argument to the address operator **&**, an argument to the **sizeof** operator, or as an initializer of a character array, then the usual array conversions come into play, changing the string from an array of characters to a pointer to the first character in the string
- Identical string constants are allowed to share the same memory

IR Code for String Literal

- For
```
char *p;
p = "Hello, world";
```
- After type checking, it becomes
```
char *p;
p = (char *)"Hello, world";
```
- At execution time, string literals need to be stored in the .data segment
  - So that string literals can be defined in the .data segment and later referred to in the .text segment, each string literal needs to have a compiler-generated label added as an attribute in the assembly code
  - To accomplish these goals, your compiler needs to build a table of string literals with associated compiler-generated labels
  - No IR code needs to be generated for the .data segment string declaration, but when assembly code is generated, the string table should be printed in the assembly language output file. Here is an example of how the string above would be output:
```
        .data
_StringLabel_1:
        .asciiz         "Hello, world"
```
- IR code for: p = (char *)"Hello, world";
```
(addressOf, r0, p)                              # r0 is an lvalue for p; r0 -> p
(addressOf, r1, _StringLabel_1)                 # r1 is an lvalue for "Hello, world“; r1 -> “Hello, world”
                                                # r1 is an rvalue for (char *) "Hello, world"
(storeWord, r0, r1)                             # p -> “Hello, world”
```

IR Code for Casts

- For

```
int i, j;
i = (int)(char)j;
```

- IR code

```
• (addressOf, r0, i)                # r0 is an lvalue for i; r0 -> i
• (addressOf, r1, j)                # r1 is an lvalue for j; r1 -> j
• (loadWord, r2, r1)                # r2 is an rvalue for j; r2 <- j
• (castWordToByte, r3, r2)          # r3 is an rvalue for (char)j; r3 <- (char)j
• (castSignedByteToWord, r4, r3)    # r4 is an rvalue for (int)(char)j;
                                    #          r4 <- (int)(char)j
• (storeWord, r0, r4)               # i <- (int)(char)j
```

Label Definitions

- Each label definition will be a degenerate quadruple (it has just two fields)
- A definition of a label named "theNewLabel" would be generated as
    - (label, theNewLabel)

- However, prefixes need to be added to user declared labels to guarantee that they are unique in the emitted MIPS assembly language program
    - This will be shown in the following slides

IR Code for goto

- For
```
void f(void) {
  …
  goto errorDetected;
  …
errorDetected:
  …
}
```
- IR code
  - …
  - (goto, _UserLabel_f_errorDetected)
  - …
  - (label, _UserLabel_f_errorDetected)
  - …
- The label prefix is formed using the function name because labels are unique per function

IR Code for if

- For

```
int i;
…code₁…
if(i)
    …code₂…
…code₃…
```

- IR code
  - …code$_1$…
  - (addressOf, r0, i)
  - (loadWord, r1, r0)
  - (gotoIfFalse, r1, _GeneratedLabel_1)
  - …code$_2$…
  - (label, _GeneratedLabel_1)
  - …code$_3$…

IR Code for while

- For

```
int i;
…code₁…
while(i)
   …code₂…
…code₃…
```

- IR code
  - …code$_1$…
  - (label, _GeneratedLabel_1)
  - (addressOf, r0, i)
  - (loadWord, r1, r0)
  - (gotoIfFalse, r1, _GeneratedLabel_2)
  - …code$_2$…
  - (goto, _GeneratedLabel_1)
  - (label, _GeneratedLabel_2)
  - …code$_3$…

IR Code for other C statements

- Use the models above to design IR sequences for the other C statements
  - if … else
  - do
  - for
  - break
    - Reminder: Binds to the innermost loop – **while**, **do**, or **for** – (or **switch**)
    - Leaves the construct to which it is bound
  - continue
    - Reminder: Binds to the innermost loop – **while**, **do**, or **for**
    - Goes to the next iteration of the construct to which it is bound
- Additional information will be furnished later on how to deal with calling functions and returning from functions
  - return

Additional Quadruple Operation Codes

- constInt

- Narrowing casts:
    - castWordToHalfWord
    - castWordToByte
    - castHalfWordToByte
- Widening casts:
    - castUnsignedHalfWordToWord
    - castSignedHalfWordToWord
    - castUnsignedByteToHalfWord
    - castSignedByteToHalfWord
    - castUnsignedByteToWord
    - castSignedByteToWord

- label
- goto
- gotoIfFalse
- gotoIfTrue

Compiler Generated Labels

- Entry point
  - main
  - Your compiler needs to check that a function with the name “main” has been defined
- String literals
  - _StringLabel_*integer*
- Label declared by the user
  - _UserLabel_*functionName_userLabel*
- Label for if, while, etc.
  - _GeneratedLabel_*integer*
- User declared file scope identifiers (global variables & functions)
  - SPIM does *not* allow labels to be the same as opcodes
  - Therefore, all global identifiers (except for “main,” which needs to be unaltered) should have a prefix of _Global_

Short-Circuit Operators

- The &&, ||, and ? : operators are more complicated than the other operators
    - They have so-called short-circuit behavior in which some operands are evaluated and others are not necessarily evaluated
- The generated IR needs to use conditional goto's to produce the correct behavior
- Keep in mind that the operands of && and || that are evaluated and the first operand of ? : need to be checked to see if they are *true* or *false* as specified by the C Programming Language

Static Single Assignment Form and Additional Complexity in Dealing with Short-Circuit Operators

- It may be apparent to the diligent student that the technique that we are using to generate code ***will assign to each temporary only once***
- This is called **Static Single Assignment** Form or **SSA**
- **SSA** enhances the compiler's ability to perform optimizations
    - Less analysis needs to be performed to determine whether an optimization is allowed

- Unfortunately, the &&, ||, and ? : operators violate the SSA principle
    - Either the true or false expression result of these operators needs to be assigned to the resultant temporary

How to Generate Code for the Short-Circuit Operators

- There are usually two approaches taken to generate code for the short-circuit operators
    - Assign to the temporary that holds the result of the operator more than once
    - Use a new IR instruction called phi (pronounced like “fee”), or ϕ, to select one of the two results, as appropriate

- Keep in mind that only one of the two operand values will actually be utilized as the result of the short-circuit operator

The phi, or ϕ, IR instruction

- (phi, r2, r0, r1)   # either r0 or r1 is assigned to r2

- The phi IR instruction is quite clever
    - It “knows” whether the first or the second operand is the one that needs to be assigned to its result based on the flow of code preceding the phi IR
    - Using the phi IR instruction maintains SSA in the IR

- Of course, when it comes to generating real assembly/machine code, a register *will be assigned to more than once*

Should I Use the phi IR Instruction?

- It is slightly more work to use the phi IR instruction
  - This is because when assembly code is generated in PS6, renaming of temporaries is required, as follows:
  - The operands to the phi IR where they are used as results of other IRs are both renamed to be the name of the result of the phi IR
  - The phi IR has no code generated for it
- Feel free to use either approach

References to Identifiers in IR

- All references to identifiers in IR should be represented by pointers to the identifier in the appropriate symbol table
- Even though we have written identifiers in IR nodes as strings, this does not furnish sufficient information to resolve the name to the correct identifier in the input program
- This procedure should be followed for both file scope (global) variables and for function and block scope (local) automatic variables

- References to identifiers should appear only in addressOf IR nodes

Automatic Variables

- Automatic (function & block scope) variables will be stored on the stack using a stack frame
- Information about calling conventions follows
  - IR for calling functions and returning from functions
  - IR for passing parameters and returning a result
  - Allocation of storage for automatic variables on the stack
  - Accessing local variables

Calling Functions with No Parameters and No Return Value

- For

```
void functionName(void);
…
functionName();
…
```

- IR code

```
(call, _Global_functionName)
```

Calling Functions with Parameters, but with No Return Value

- For

```
void functionName(int a, int b);
…
int i, j;
functionName(i, j);
…
```

- IR code

```
…
(addressOf, r0, i)
(loadWord, r1, r0)
(parameter, 0, r1)
(addressOf, r2, j)
(loadWord, r3, r2)
(parameter, 1, r3)
(call, _Global_functionName)
…
```

Accessing a Called Function’s Return Value

- For

```
int functionName(void);
…
int i;
i = functionName();
…
```

- IR code

```
(addressOf, r0, i)
(call, _Global_functionName)
(resultWord, r1)
(storeWord, r0, r1)
```

Definition of a Function

- For

```
void functionName(void) {
  …
}
```

- IR code

```
(procBegin, _Global_functionName)
…
(procEnd, _Global_functionName)
```

- Executing the procEnd IR operation is the only way to return to the caller

Returning a Value From a Function

- For

```
int functionName(void) {
  int i;
  …
  return i;
  …
}
```

- IR code

```
(procBegin, _Global_functionName)
…
(addressOf, r0, i)
(loadWord, r1, r0)
(returnWord, r1)
(goto, _GeneratedLabel_1)
…
(label, _GeneratedLabel_1)
(procEnd, _Global_functionName)
```

Summary of IR Operation Codes for Procedures/Functions

- Opcodes used in procedure/function definitions
  - procBegin
  - procEnd
  - returnWord
  - returnHalfWord
  - returnByte

- Opcodes used in procedure/function references
  - call
  - parameter
  - resultWord
  - resultHalfWord
  - resultByte

Summary of IR Operation Codes for Memory Access Operations

Memory Address Opcode:

- addressOf

Memory Access Opcodes:

- loadWord
- loadHalfWord
- loadSignedHalfWord
- loadByte
- loadSignedByte
- storeWord
- storeHalfWord
- storeByte

Summary of IR Operation Codes for Multiplicative, Additive, and Shifting Operations

Multiplicative Opcodes:
- multSignedWord
- multUnsignedWord
- divSignedWord
- divUnsignedWord
- remSignedWord
- remUnsignedWord

Additive Opcodes:
- addSignedWord
- addUnsignedWord
- subSignedWord
- subUnsignedWord

Shifting Opcodes:
- leftShiftWord
- rightShiftSignedWord
- rightShiftUnsignedWord

Summary of IR Operation Codes for Comparison Operations

Inequality Comparsion Opcodes:

- ltSignedWord
- ltUnsignedWord
- leSignedWord
- leUnsignedWord
- geSignedWord
- geUnsignedWord
- gtSignedWord
- gtUnsignedWord

Equality Comparison Opcodes:

- eqWord
- neWord

Summary of IR Operation Codes for Bitwise Operations

Bitwise Opcodes:

- bitwiseAndWord
- bitwiseXorWord
- bitwiseOrWord

Summary of IR Operation Codes for Unary Operations

Unary Opcodes:

- unaryMinus
- unaryLogicalNegation
- unaryBitwiseNegation

Summary of IR Operation Codes for Integral Literal and Type Casting Operations

Integral Literal Opcode:

- constInt

Narrowing Cast Opcodes:

- castWordToHalfWord
- castWordToByte
- castHalfWordToByte

Widening Cast Opcodes:

- castUnsignedHalfWordToWord
- castSignedHalfWordToWord
- castUnsignedByteToHalfWord
- castSignedByteToHalfWord
- castUnsignedByteToWord
- castSignedByteToWord

Summary of IR Operation Codes for Labels, Branching, and Temporary Remapping

Label Definition Opcode:

- label

Branching Opcodes:

- goto
- gotoIfFalse
- gotoIfTrue

Temporary Remapping Opcode:

- phi

IR for Invoking System Calls (to be described later in the MIPS Assembly Language slides)

- Opcode used to invoke a system call
  - syscall