Compiler Design

Associated Supplemental Materials

Generating x86

David Galles

Chapter 8

Code Generation For x86

Chapter Overview

Once we have created an assembly tree, the next step is to create actual assembly code for a specific architecture. We will generate assembly by "tiling" the abstract assembly tree. We will create a set of "tiles", each of which can cover a small portion of an abstract assembly tree. Each tile will be associated with a segment of the actual assembly code which implements the section of the assembly tree covered by the tile. To generate the actual assembly for a specific assembly tree, we will first cover the tree with tiles, making sure that all portions of the tree are covered, and no tiles overlap. Then we will output the assembly associated with each tile. We will also need to use a slightly different stack frame for x86 assembly than was described in chapter 9 for MIPS.

8.1 Stack Frames For x86

The stack layout for x86 assembly is slightly different than the stack layout in MIPS. The two major changes are:

- The return address for functions is stored on the stack, and not in a register. The return address needs to be the first item in the activation record for a function.
- The stack pointer needs to point to the last element on the stack, and not to the next free element of the stack. At no time should elements beyond the stack pointer be modified (that is, we should never change the memory contents of (SP - 4) directly). When using MIPS assembly, we do all stack manipulation ourselves, and can set up the SP any way that we like. In x86 assembly, there are special instructions that deal directly with the stack, and so we need to follow specific conventions.

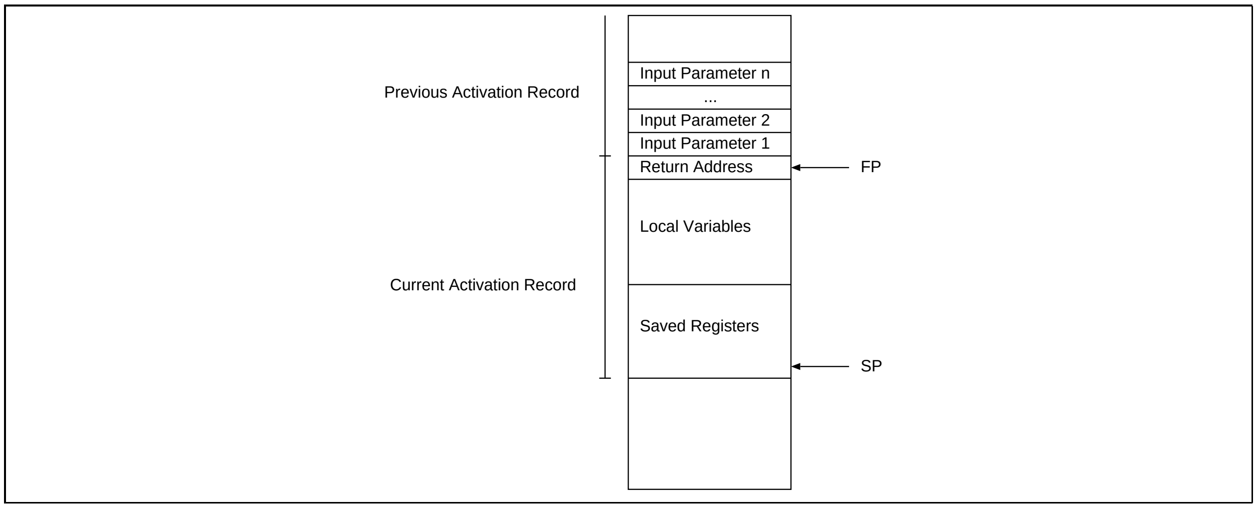


Figure 8.1: The stack layout for x86.

Register	Description
EAX	Accumulator & General Purpose Register
EBX	General Purpose Register
ECX	General Purpose Register
EDX	General Purpose Register
ESP	Stack Pointer
EBP	Base Pointer – we will use it as a Frame Pointer
EIP	Instruction Pointer
flags	Flag register, only accessed indirectly

Table 8.1: Some of the registers in x86 assembly.

The stack layout for x86 assembly can be found in Figure 8.1. The "saved registers" portion of the stack contains the saved values of registers from the previous function. For the basic code generator, we will only need to save the old value of the Frame Pointer.

8.2 Target Assembly Code

We will be using a subset of x86 assembly and registers for code generation, as described below.

8.2.1 x86 Registers

The x86 registers that we will use are in Figure 8.1. The registers EAX-EDX are general purpose registers, though even the general purpose registers have some specialized purposes. Multiplication and division, for instance, can only be done through the EAX register, and a multiplication or division will overwrite any value stored in the EDX register. ESP is a standard stack pointer, and we will use EBP as a frame pointer (though assembly language programmers often use this register for different purposes). EIP is the instruction pointer, which points to the next assembly language instruction to be executed by the CPU. The flags register is a set of status bits that can only be set indirectly. Some of the status bits in the flags register are:

Overflow 1 if the last arithmetic operation caused an overflow.

Sign Contains the sign of the result of the last arithmetic operation.

Zero 1 if the last arithmetic operation produced a zero result, 0 otherwise.

Instruction		Description
add	dest, source reg, reg/mem/const mem, reg/const	dest = dest + source.
sub	dest, source reg, reg/mem/const mem, reg/const	dest = dest - source.
inc	dest reg/mem	dest = dest + 1

mul	source
	reg/mem
		AX = AX * source (low order bits)
		DX = AX * source (high order bits)

div	source
	reg/mem
		AX = AX / source (truncated)
		DX = AX `mod` source

push	source reg/mem/const	ESP = ESP+4, MEM[ESP] = source
pop	source reg/mem	source = MEM[ESP], ESP = SP-4
call	addr reg/mem	ESP = ESP+4, MEM[ESP] = EIP, EIP = addr
ret		IP = MEM[ESP], ESP = ESP - 4
mov	dest, source reg, reg/mem/const mem, reg/const	dest = source

Table 8.2: A subset of x86 assembly statements for arithmetic and memory manipulation.

Carry 1 if the last add produced a carry.

We will only use the sign and zero flags, and those will only be used indirectly through conditional jump and set operations, defined below. For clarity, this chapter will only covers the 32 bit versions of the registers defined in 386 and later processors, though 8 bit and 16 bit versions also exist.

8.2.2 x86 Instructions

The subset of x86 instructions that we will use are in Figures 8.2 and 8.3. Our code generator will only take advantage of a small portion of the x86 instruction set. Writing a compiler to effectively use the complete x86 instruction set is extremely complicated, and well beyond the scope of this text.[1] The instructions can take registers, memory locations, or constants as arguments, as noted in the table above. For instance, the add instruction can take as operands a two registers, a register and a memory location, a register and a constant, a memory location and a register, and a memory location and a constant. The only combination not allowed is two memory locations – no instruction allows access to two memory locations. We will denote a memory location by using square brackets []. Thus, EAX stands for the register EAX, and [EAX] stands for the address stored in register EAX. For example, the instruction:

```
mov EAX, [EBP]
```

Moves the contents of the memory location pointed at by EBP to the EAX register. A memory location can be an assembly language label, a constant, a register, or the sum of a register and a constant or two registers. For example the instruction:

```
mov EAX, [EBP-4]
```

[1]Note that even gcc typically only uses a small subset of x86 assembly instructions when compiling under linux.

Instruction		Description
cmp	dest, source reg, reg/mem/const mem, reg/const	Comparison Sets flags register, to be used before a jump or set instruction
je	addr mem	Jump-Equal set EIP = addr if the last comparison was =
jne	addr mem	Jump-Not-Equal set EIP = addr if the last comparison was $\neq$
jg	addr mem	Jump-Greater set EIP = addr if the last comparison was $>$
jge	addr mem	Jump-Greater-Equal set EIP = addr if the last comparison was $\geq$
jl	addr mem	Jump-Less set EIP = addr if the last comparison was $<$
jle	addr mem	Jump-Less-Equal set EIP = addr if the last comparison was $\leq$
jmp	addr mem	Jump set EIP = addr unconditionally
sete	dest reg/mem	Set-Equal if the last comparison was =, then set dest = 1, else set dest = 0
setne	dest reg/mem	Set-Not-Equal if the last comparison was $\neq$, then set dest = 1, else set dest = 0
setg	dest reg/mem	Set-Greater if the last comparison was $>$, then set dest = 1, else set dest = 0
setge	dest reg/mem	Set-Greater-Equal if the last comparison was $\geq$, then set dest = 1, else set dest = 0
setl	dest reg/mem	Set-Less if the last comparison was $<$, then set dest = 1, else set dest = 0
setle	dest reg/mem	Set-Less-Equal if the last comparison was $<=$, then set dest = 1, else set dest = 0

Table 8.3: A subset of x86 assembly statements that manipulate and use the flag register.

Moves the contents of the memory location 4 bytes away from the location pointed at by EBP to the EAX register.

The conditional jump instructions are used in conjunction with the comparison cmp operation. The operation cmp dest, source subtracts dest from source, setting the appropriate flags, and discards the result. The conditional jump instructions then use the flags register to determine when to jump. For instance, the Jump-Equal command je will jump if the zero flag is 1, and the jle will jump if the zero flag is 1 or the sign flag is negative. As an example, we can jump to the assembly language label foo1 if the value stored in EAX is greater than the value stored in EBX with the following instructions:

```
cmp EAX, EBX
jg [foo1]
```

The set instructions work in the same way as the conditional jump instructions.

Function values are returned by setting the accumulator register, EAX.

8.3 Simple Tiling

We will first describe a simple approach to tree tiling, which is easy to describe and understand, but produces inefficient code. We will then move on to more complicated tiling strategies, which will produce more efficient code. The simple tiling is based on a post-order traversal of the abstract assembly tree. That is, after we cover the tree with tiles, we will emit the code for each of the tiles in a post-order fashion. First, we will recursively emit the tiles for each of the subtrees of an assembly tree, from left to right (with the exception of function calls, as described below). Then, we will emit the code for the tile covering the root of the tree. Examples follow in the next few sections.

8.3.1 Simple Tiling of Expression Trees

We will start with the tiles for expression trees. The code associated with the tile for an expression tree will place the value of that expression on the top of the stack. We will start with very simple expressions, and move to more complicated expressions.

Constant Expressions

The simplest expression is a constant. Consider the following expression tree:

Constant(5)

The code associated with this tile needs to place a 5 on the top of the stack. This can be accomplished with the push operation, as follows:

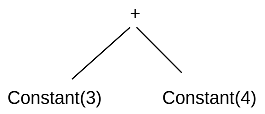
```
push 5          ; Push 5 on the top of the stack
```

In general, the tile for any constant value x is:

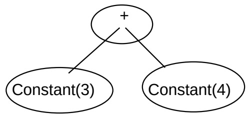
```
push x          ; Store x on the top of the stack
```

Arithmetic Binary Operations

We now look at a slightly more complicated expressions – binary operations. Consider the following expression tree:



Instead of trying to cover this entire tree with one tile, we will use three. The constants 4 and 5 can each be covered by a constant tile, as above, and we will use a + tile to cover the +.



What should the code for the + tile be? Recall that we are emitting the code for the tiles in a post-order fashion – that is, first emitting the code for the left subtree, then the code for the right subtree, then the code for the root. Thus, we can assume that the values of the left and right hand operands of the + are already on the stack when the code for the + tile is emitted. Thus all we need to do is pop off those values, do the sum, and push the result back on the stack. The code for the + tile is thus:

```
pop  EAX,           ; Pop the second operand into the accumulator, EAX
add [ESP], EAX      ; Do the addition, storing the result on the stack
```

and the code for the entire expression is:

```
push 4             ; Store the 4 on the stack
push 5             ; Store the 5 on the stack
pop  EAX           ; Pop the 5 into the accumulator, EAX
add [ESP], EAX     ; Do the addition, storing the result on the stack
```

Code for other arithmetic operators is similar. This seems like an awful lot of code for such a simple expression. In section 8.4 we will look at a different tiling strategy that generates more terse code.

Registers

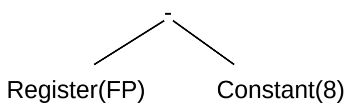
Consider the expression tree

Register(FP)

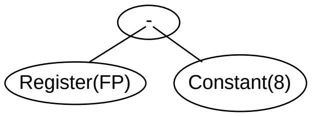
We will cover this tree with a single tile. The code for this tile needs to push the value of the frame pointer onto the top of the expression stack. This can be easily done with the code:

```
push EBP         ; Push the register EBP on the stack
```

Combining some of the tiles we've already discussed, the tree



can be tiled as follows:

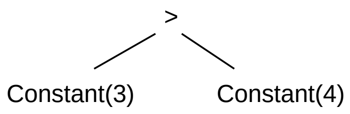


producing the assembly code:

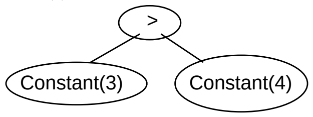
```
push  EBP               ; Store the register EBP on the stack
push  8                 ; Store the constant 8 on the stack
pop   EAX                ; Pop the second operand (8) into the accumulator, EAX
sub  [ESP], EAX         ; Do the subtraction, storing the result on the stack
```

Relational Operations

Relational operators, such as <, >, and =, produce boolean values, which are represented internally as 0 for false, and 1 for true. We can use the various set instructions to set a register or memory location to 0 or 1, based on a relational operation. For example consider the tree:



We can tile this tree as follows:



The code for the > tile needs to pop the values of the operands off the top of the stack, and store 1 on the top of the stack if op1 > op2, and 0 on the top of the stack otherwise. We can do this with a combination of cmp and setg, as follows:

```
pop   EAX         ; Pop the second operand to the accumulator
cmp  [ESP], EAX   ; Compare first operand (on stack) with second operand
setg [ESP]        ; Place the proper boolean value on the stack
```

Other relational operators can be tiled in a similar fashion.

Boolean Operations

Boolean operators – &&, ||, and ! – are very similar to arithmetic operators, and are implemented in a similar fashion. If we use 1 to represent true, and 0 to represent false, then (!x) is equivalent to (1-x). The && tile can be implemented with the code:

```
pop   EAX          ; Pop the second operand into the accumulator
add   EAX, [ESP]   ; Add the two operands, storing the result in the accumulator
cmp   EAX,  1      ; Compare the value of the two operands to 1
setg [ESP]         ; Set result to 1, if sum of operands is > 1
```

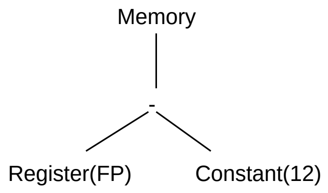
Of course, we could also choose to represent false with 0, and true with any non-zero value, in which case we would need to change our code for boolean operations.

Memory Accesses

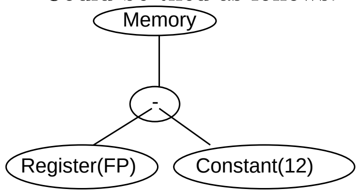
A Memory node is just a memory dereference. To implement a memory dereference, we need to pop the address from the top of the stack, dereference this address to find the value stored there, and then push this value on the top of the stack. The code for a Memory node is thus:

```
mov  EAX, [ESP]     ; Load the address to dereference off the top of the stack
mov  EAX, [EAX]     ; Dereference the pointer
mov [ESP], EAX      ; Replace the result back on the stack
```

Thus, the following expression tree (corresponding to a simple local variable)



Could be tiled as follows:



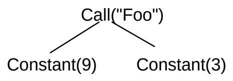
which results in the following assembly:

```
push  EBP             ; Push the Frame Pointer (EBP) on the stack
push  12              ; Push the constant 12 on the stack
pop   EAX             ; Pop the second operand (12) into the accumulator, EAX
sub  [ESP], AX        ; Do the subtraction, storing the result on the stack
mov   EAX, [ESP]      ; Pop the address to dereference off the top of the stack
mov   EAX, [EAX]      ; Dereference the pointer
mov  [ESP], EAX       ; Push the result back on the stack
```

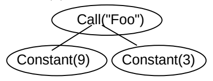
The end result of this code is that the value that was in memory location (FP - 12) is pushed onto the top of the stack.

Function Calls

To implement a function call, we need to copy all of the parameters of the function onto the top of the stack and then jump to the start of the function. When the function returns, we need to pop the arguments off the top of the stack, and push the return value of the stack. *To make the parameters appear in the proper place on the stack, we will need to make a slight break from our left-to-right, postorder traversal when emitting code, and emit the code for parameters to a function call in right-to-left order.* This is the only deviation from emitting code for tiles in a straightforward postorder fashion. Consider the following expression tree, which results from the function call foo(9,3):



We can tile this tree as follows:



The assembly language associated with the Call tile is:

```
call [foo]            ; Make the function call
add   ESP,  8         ; Pop the input parameters off the stack
push  EAX             ; Push the result of the function on the stack
```

Thus the assembly for the complete expression tree is:

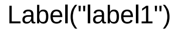
```
push  3              ; Push the 2nd parameter on the stack
push  9              ; Push the 1st parameter on the stack
call [foo]           ; Make the function call
add   ESP,  8        ; Pop the input parameters off the stack
push  EAX            ; Push the result of the function on the stack
```

8.3.2 Simple Tiling of Statement Trees

While the assembly code for expression trees pushes the value of the expression on the top of the stack, the assembly code for statement trees implements the statement described by the tree. We now look at tiles for statement trees.

Label Trees

The subtree

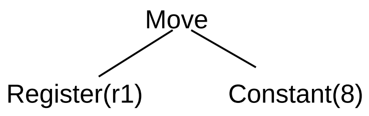


can be covered in a single tile. The code for that tile is just a label:

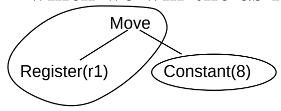
```
label1:
```

Move Trees

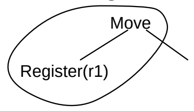
Trees that move data around, Moves, are a little more complicated. The left-hand side of a Move needs to be either a register or a memory location. Thus Moves will be the only simple tiles that cover two tree nodes – both the MOVE node and either a Memory node or a register node. Let's look at a register move first. Consider the tree (where r1 is any register – AX, BX, BP, etc):



which we will tile as follows:



The register move tile:



has the following associated code:

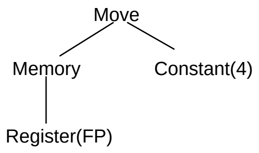
```
pop  r1          ; Pop the value off the top of the stack, into r1
```

Thus the complete expression tree would be associated with the code:

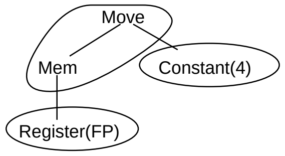
```
push 8           ; Push the constant 8 on the top of the stack
pop  r1          ; Pop top of stack into register r1
```

While the above code is verbose, it is correct. In the next section we will look at more sophisticated tiling that will produce more efficient code.

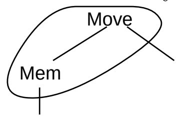
Now, let's look at a memory move. Consider the following tree:



This tree can be tiled as follows:



The memory move tile:



has the following associated code:

```
pop  EAX,             ; Pop the value to move into a register
mov [ESP], EBX        ; Implement the move
```

Thus the entire statement tree can be implemented with the following code:

```
push  EBP             ; Push the Frame Pointer EBP (destination) onto the stack
push  4               ; Push the constant 4 (value to move) on the stack
pop   EAX             ; Pop the value to move into a register
mov  [ESP], EBX        ; Implement the move
```

Jump Trees

Jump statement trees, such as:

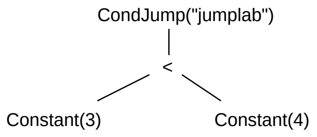
Jump("label")

Can be tiled with a single tile. The code for this tree is just a jump instruction:

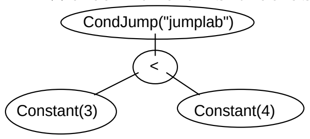
```
jmp [label]
```

Conditional Jump Trees

Consider the following statement tree:



We can tile this tree as follows:



What should the code for the cjump tile be? If the value of the expression subtree is true (that is, equal to 1), we want to jump to the label "jumplab", otherwise we don't need to do anything. The code that we emit for the conditional jump's expression subtree places the value of the expression on the top of the stack. Thus, we need to pop the expression off the top of the stack, and jump to the label "jumplab" if the expression is true. Thus, the cjump tile has the following associated assembly:

```
pop EAX                ; Pop the value on the top of the stack
cmp EAX,  0            ; Compare expression with 0
jne  [jumplab]         ; Jump if the popped value is not false
```

Thus the entire statement tree is implemented with the code:

```
push 3
push 4
pop  EAX               ; Pop second operand into the accumulator
cmp  [ESP], EAX        ; Compare first operand (on stack) with second operand
setg [ESP]             ; Place the proper boolean value on the stack
pop  EAX               ; Pop the boolean value into the accumulator
cmp  EAX,  0           ;  Compare expression with false (zero)
jne  [jumplab]         ;  Jump if the popped value is not false
```

Granted, this is an awful lot of code for a simple statement. Section 8.4 will cover a more efficient tiling strategy.

Sequential Trees

Sequential trees are an interesting special case – after we have created the code for the left subtree and the right subtree, what do we need to do? Nothing! Thus a sequential tile covers just the seq node, and there is no code associated with the tile.

Procedure Calls

Procedure calls are very similar to function calls. The only difference is that there is no return result for a procedure. So, after the procedure ends, we do not need to copy the result onto the expression stack.

8.4 More Sophisticated Tiling

The above tiling strategy will indeed work, and produce correct code. The code that is produced, however, will be inefficient. We will look at three strategies for improving the efficiency of our generated code. The first involves a method for limiting the use of the expression stack by using an accumulator register. The second involves creating larger tiles which cover a larger section of the assembly tree. Finally, the third strategy will streamline the accumulator stack operations.

8.4.1 Accumulator Register

We will modify the code associated with expression tiles, so that instead of placing the value of the expression tree on the top of the expression stack, the value of the expression tree will be placed in the accumulator register, AX. We will still need to use the stack to store old values of the accumulator.

Tiling Constants using an Accumulator Register

Using the accumulator register method, the code associated with some expression trees will be quite simple. The code associated with the constant expression Constant(15) needs to place 15 in the accumulator register. This can be done with a single assembly language statement, as follows:

```
mov  EAX, 15
```

Tiling Registers Using an Accumulator Register

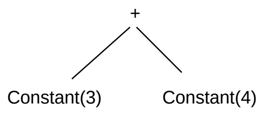
The code associated with a register expression, Register(r1), needs to move the value of r1 into the accumulator register, and this can also be done with a single assembly language statement:

```
mov  EAX, r1
```

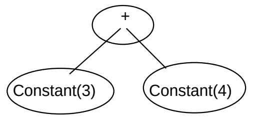
Tiling Binary Operators Using an Accumulator Register

There is a slight problem with binary operations. If we execute the code to place the left operand in the accumulator register, and then execute the code to place the right operand in the accumulator register, we will overwrite the value of the left operand with the value of the right operand. What can we do? After executing the code to place the one of the operands in the accumulator, we can save that value on the stack, and then execute the code to place the other operand in the accumulator.

Of course, the code generation is now more complicated – instead of emitting the code for each tile in a post-order traversal of the tree, we will have to do a combination in-order/post-order traversal – first emitting code for one subtree, then emitting code to save the value, then emitting code for the other subtree, and finally emitting code for the root. Because of the way x86 operations are implemented, we will want to first emit code for the *right* subtree, save the value, and then emit code for the *left* subtree, before emitting code to implement the root tile. Consider a simple binary expression tree, like:



We can tile this tree just as before:



but now the code for the + tile is:

```
<code for *right* operand>
push  EAX                  ; Push *right* operand on the stack
<code for *left* operand>
```

```
pop   EBX                  ; Pop the *right* operand from the stack
add   EAX,  EBX            ; Do the addition
```

The complete code for that tree would then be:

```
mov   EAX,  4             ; Store constant 4 (right operand) in the accumulator
push  EAX                 ; Push *right* operand on the stack
mov   EAX,  3             ; Store constant 3 (left operand) in the accumulator
pop   EBX                 ; Pop right operand into a register
add   EAX, EBX            ; do the operation
```

The code associated with other operator tiles can be converted to use the accumulator register in a similar fashion.

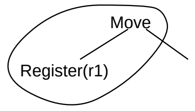
Tiling Memory Accesses Using an Accumulator Register

A Memory node is just a memory dereference. To implement a Memory node, then, we need to move the data stored at the address pointed to by the accumulator into the accumulator. The code for a Memory node (using the accumulator register) would be:

```
mov EAX, [EAX]    ; Store the contents of the memory location pointed
                  ;  to by AX in AX
```

Tiling Moves Trees Using an Accumulator

Tiling register moves using an accumulator is easy – we just need to move the accumulator into the appropriate register. Thus the register move tile:



thus has the following associated code:

```
mov  EAX, r1         ;  Store the value of register r1 in the accumulator
```

Tiling memory moves is a little more complicated, since we need to save the old value of the accumulator on the expression stack. In a sense, memory moves are just a binary operator on two expressions (though an operator that has a side effect instead of returning a value). Thus the code emitted for a memory move will be similar to that emitted for a binary expression. First, we will emit the code that places the left-hand side of the move (the address to move) into the accumulator. Next, we store this address on the stack. After that, we can safely emit code which places the value to move in the accumulator. Finally, we can pop the destination from the stack, and implement the move, as follows:

```
<code for left subtree (move destination)>
push EAX                ; Push destination of move on stack
<code for right subtree (value to move)>
pop  EBX                ; Push the destination from the stack
mov [BX], AX            ; Implement the move
```

An extended example of a move expression using an accumulator appears in section 8.4.2.

Tiling Function & Procedure Calls Using an Accumulator

Tiling a function call using an accumulator is very similar to the naive tiling for a function call. There are two main differences. The first is that arguments must be moved explicitly to the call stack from the accumulator, and the second is that the result does not need to be stored on the stack, but can remain in the accumulator.

```
<Code for nth argument>
push EAX                  ; Store the nth argument on the stacj
<Code for (n-1)st argument>
push EAX                  ; Store (n-1)st argument on the stack
...
<Code for 1st argument>
push EAX                  ; Store 1st argument on the call stack
call  [foo]               ;  Make the function call
add   SP, 4n              ;  Update call stack pointer
```

Procedures calls are identical.

Tiling Conditional Jumps Using an Accumulator

Unlike operator expressions and moves, conditional jumps do not require a temporary value to be saved. Instead, we only need to do a jump if the value stored in the accumulator is true – that is, not equal to zero. The code is thus:

```
<Code for conditional expression>
cmp   EAX, 0
jne  [jumplabel]
```

Tiling Jumps, Labels and Sequential Statements Using and Accumulator

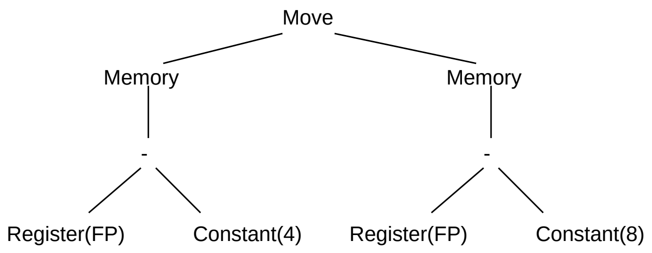
Jumps, labels, and sequential statements do not rely on the stack, so the code associated with their tiles is unchanged.

8.4.2 Larger Tiles

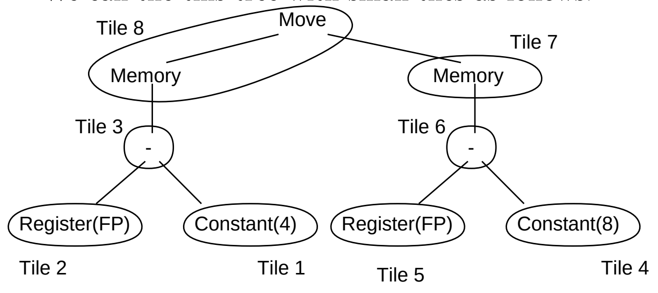
Another approach to creating more efficient code is to use bigger tiles – that is, tiles that cover larger sections of the assembly tree.

Move Trees

Consider the following assembly tree, which implements a simple assignment statement to a local variable:



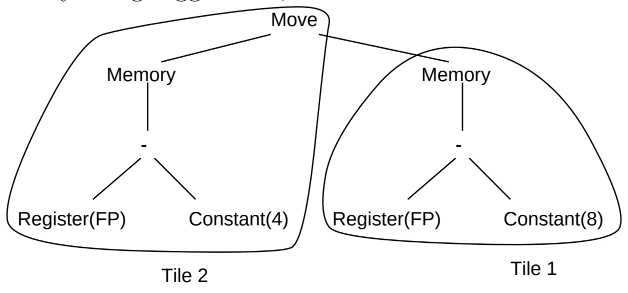
We can tile this tree with small tiles as follows:



to get the following code (this example uses the accumulator register described in the previous section):

```
mov   EAX,  4          ; Tile 1
push  EAX              ; Tile 3
mov   EAX,  EBP        ; Tile 2
pop   EBX              ; Tile 3
sub   EAX,  EBX        ; Tile 3
push  EAX              ; Tile 8
mov   EAX,  8          ; Tile 4
push  EAX              ; Tile 6
mov   EAX,  EBP        ; Tile 5
pop   EBX              ; Tile 6
sub   EAX,  EBX        ; Tile 6
mov   EAX, [EAX]       ; Tile 7
pop   EBX              ; Tile 8
mov  [EBX], EAX        ; Tile 8
```

By using bigger tiles, we can cover this tree with only 2 tiles, as follows:



Both tiles 1 and two in the above tiling are much larger that any tiles that we have seen previously. Do these larger tiles need to be associated with larger code segments? No! For tile 1, we must load into the accumulator the data stored at the memory location that is offset 8 from the frame pointer, which can be done in a single instruction:

```
mov EAX, [EBP - 8]
```

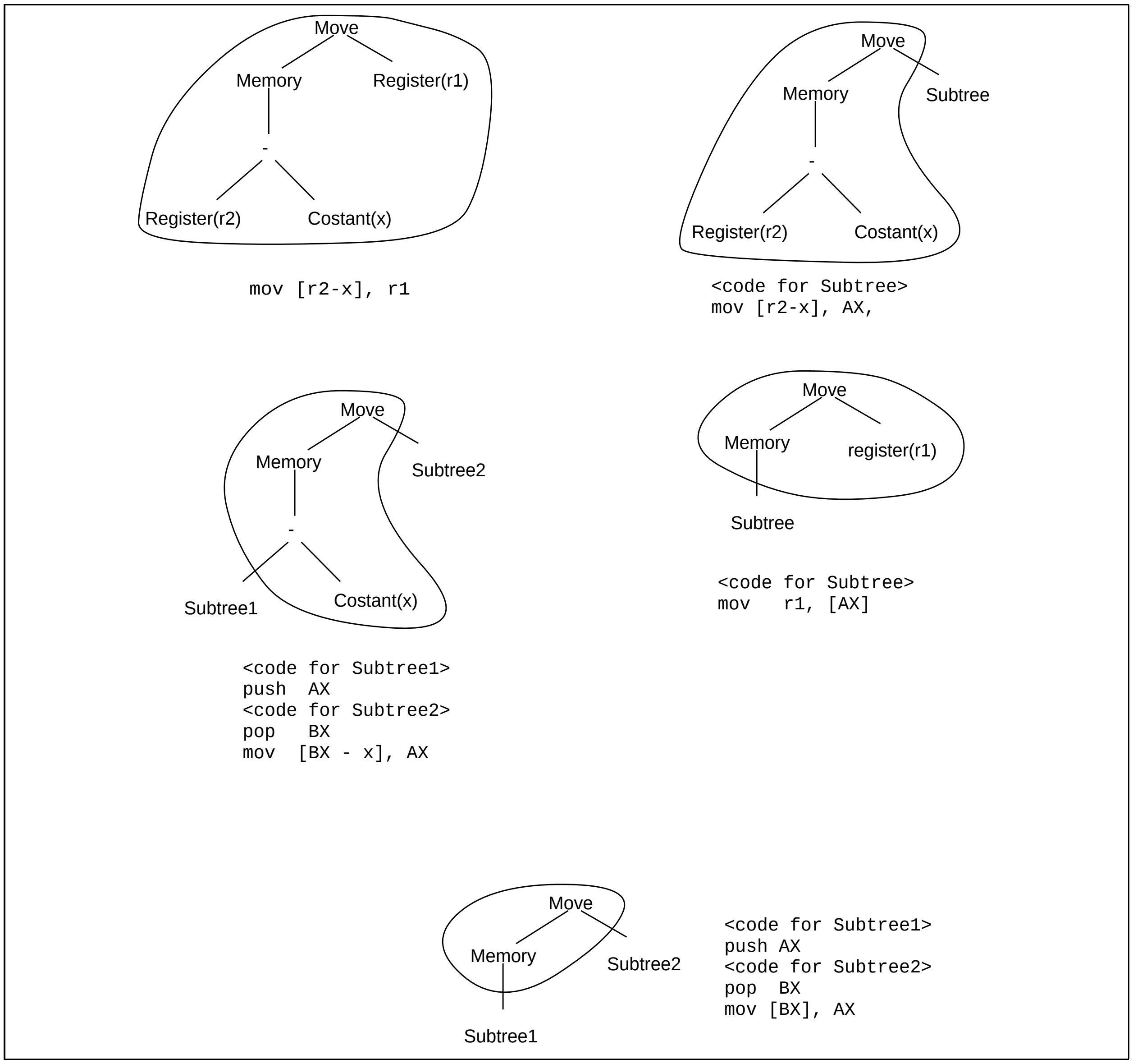
Likewise, for tile 2 we need to store the value in the accumulator into a memory address that is a constant offset from the frame pointer, which can also be done in a single instruction:

```
mov [EBP - 4], EAX
```

Thus the entire expression tree can be implemented with:

```
mov  EAX,     [EBP - 8]   ; code for tile 1
mov [EBP - 4], EAX        ; code for tile 2
```

We get a huge savings by using larger tiles – partially because we did do need to save values on the expression stack as we had before, and partially because we are taking full advantage of the `mul` assembly language instruction. How do we decide how large the tiles should be? We want a good match between the tiles and the generated assembly language instructions. That is, we want each tile to do basically the work of a single assembly language instruction (plus whatever extra work needs to be done to save values on the expression stack). Let's look a little closer at the Move instruction. The Move instruction can either store the value of a register, constant, or memory location into a register, or store the value of a constant or a register into a memory location. A memory location can be either the value in a register, the value in a register plus some constant, or the value in a register plus the value in another register. Some example tiles based on this instruction are in Figure 8.2.



```
mov [r2-x], r1
```

```
<code for Subtree>
mov [r2-x], AX,
```

```
<code for Subtree1>
push  AX
<code for Subtree2>
pop   BX
mov  [BX - x], AX
```

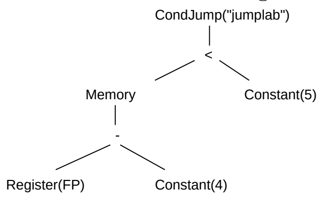
```
<code for Subtree>
mov   r1, [AX]
```

```
<code for Subtree1>
push AX
<code for Subtree2>
pop  BX
mov [BX], AX
```

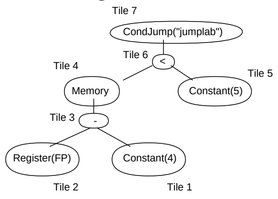
Figure 8.2: Various tiles to implement a memory move, along with associated code.

Conditional Jumps

Let's look at some larger tiles for conditional jumps. Consider the following expression tree:



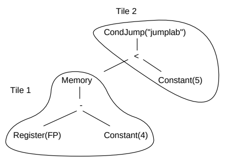
Using the small tiles approach, we get the following tiling:



Which results in the following assembly

```
mov   EAX,  4     ; Tile 1
push  EAX         ; Tile 3
mov   EAX,  EBP   ; Tile 2
push  EBX         ; Tile 3
sub   EAX, EBX    ; Tile 3
mov   EAX, [EAX]  ; Tile 4
push  EAX         ; Tile 6
mov   EAX,  5     ; Tile 5
pop   EBX         ; Tile 6
cmp   EBX, EAX    ; Tile 6
sgt   EAX         ; Tile 6
cmp   EAX, 0      ; Tile 7
jlt   [jumplab]   ; Tile 7
```

We can tile this tree with only two tiles, as follows:



Which has the associated code:

```
mov  EAX, [EBP - 4]    ; Tile 1
cmp  EAX, 5            ; Tile 2
jlt  [jumplab]         ; Tile 2
```

We have combined the conditional jump and the < 5 into one tile, as well as combining all of the nodes for the local variable [EBP - 4] into one tile.

Selecting Tiles

Once we have added larger tiles, there will be multiple ways to tile the same assembly language tree. Naturally, we will want to pick a tiling strategy that produces the most efficient assembly code possible. Assuming that each tile is associated with approximately the same number of assembly language instructions, we want to pick a tiling that has the smallest possible number of tiles. We will use the following greedy strategy

- Pick the largest possible tile that covers the root of the assembly tree

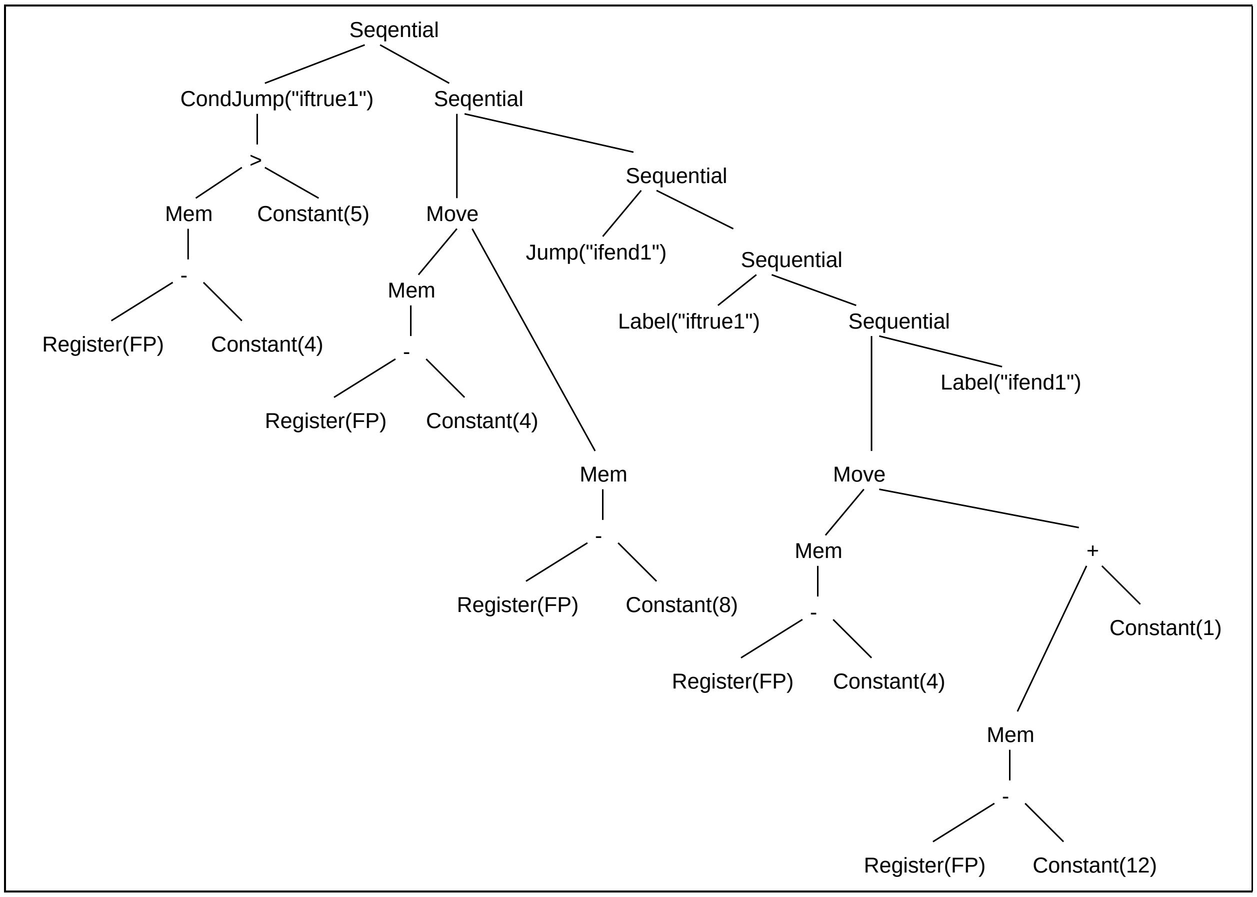


Figure 8.3: Abstract assembly for the simpleJava statement `if (x > 5) x = z + 1; else x = y;`, assuming x has offset 4, y has offset 8, and z has offset 12.

- Recursively tile the untiled subtrees

Will this strategy always produce a valid tiling? That is, will we ever get stuck, not being able to tile the rest of the tree because we made poor choices early on? As long as we include all the small tiles along with the larger tiles in the set we use for tiling, we will never paint ourselves into a corner. Why? We can cover any subtree with tiles if we are allowed to cover each node with a single tile!

8.5 Extended Example

We now look at an extended example of generating code from an assembly tree. The assembly tree is generated from the following simpleJava code:

```
if (x > 1)
   x = y;
else
   x = z + 1;
```

Assuming that x has the offset 4 off the Frame Pointer, y has the offset 8 off the Frame Pointer, and z has the offset 12 off the Frame Pointer, the abstract assembly for this statement is in Figure 8.3.

8.5.1 Code Generation Example Using Simple Tiling

Using the simplest tiling strategy, the assembly tree in Figure 8.3 will be tiled as in Figure 8.4

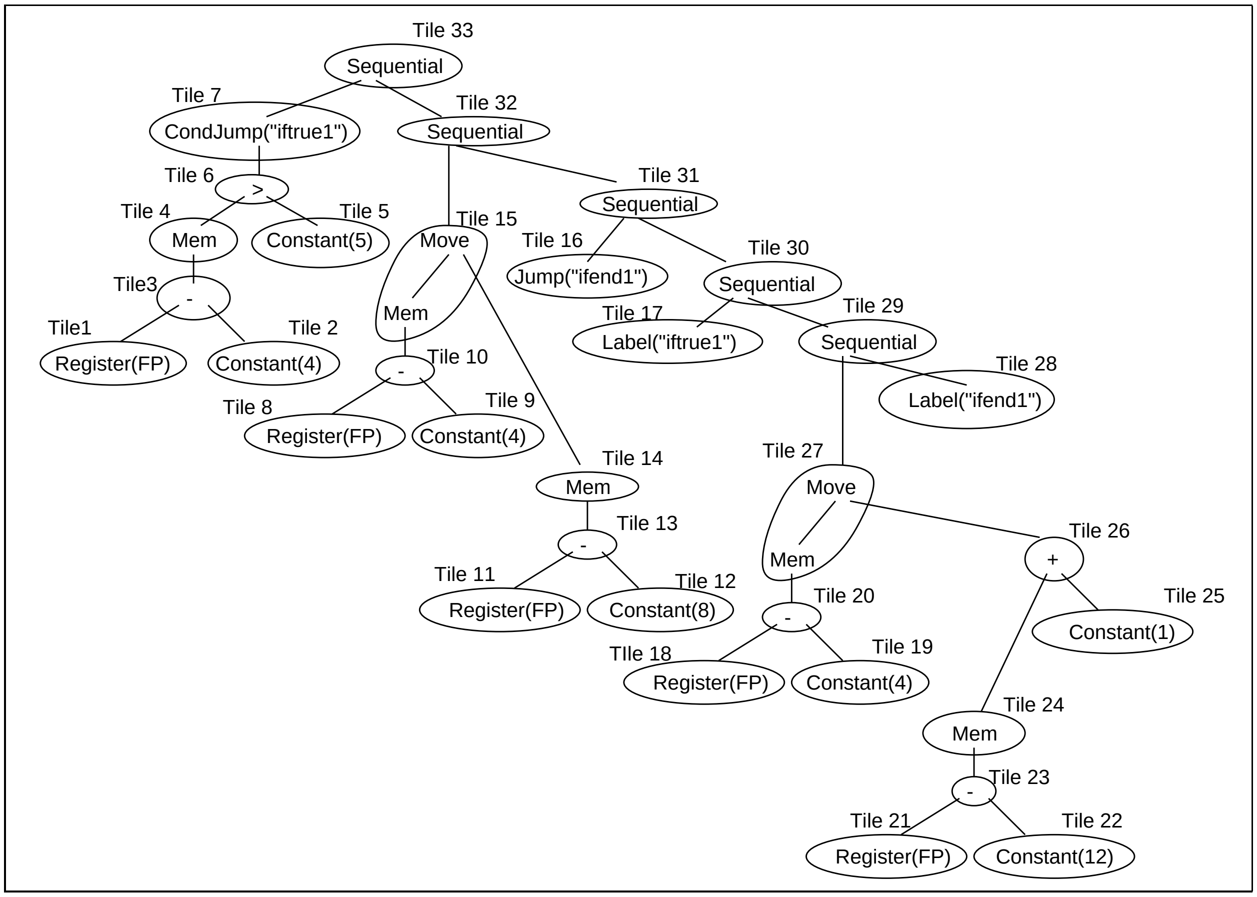


Figure 8.4: Simple tiling of abstract assembly for the simpleJava statement `if (x > 5) x = z + 1; else x = y;`, assuming x has offset 4, y has offset 8, and z has offset 12.

The assembly for the tiling in Figure 8.4 is the following:

```
        push  EBP                 ; Tile 1
        push  4                   ; Tile 2
        pop   EAX                 ; Tile 3
        sub  [ESP],  EAX          ; Tile 3
        mov   EAX,  [ESP]         ; Tile 4
        mov   EAX,  [EAX]         ; Tile 4
        mov  [ESP],  EAX          ; Tile 4
        push  5                   ; Tile 5
        pop   EAX                 ; Tile 6
        cmp  [ESP], EAX           ; Tile 6
        setg [ESP]                ; Tile 6
        pop   EAX                 ; Tile 7
        cmp   EAX, 0              ; Tile 7
        bne   [iftrue1]           ; Tile 7
        push  EBP                 ; Tile 8
        push  4                   ; Tile 9
        pop   EAX                 ; Tile 10
        sub  [ESP],  EAX          ; Tile 10
        push  EBP                 ; Tile 11
        push  8                   ; Tile 12
        pop   EAX                 ; Tile 13
        sub  [ESP],  EAX          ; Tile 13
        mov   EAX,  [ESP]         ; Tile 14
        mov   EAX,  [EAX]         ; Tile 14
        mov  [ESP],  EAX          ; Tile 14
        pop   EAX,                ; Tile 15
        mov  [ESP],  EBX          ; Tile 15
        jmp  [ifend]              ; Tile 16
iftrue1:                          ; Tile 17
        push  EBP                 ; Tile 18
        push  4                   ; Tile 19
        pop   EAX                 ; Tile 20
        sub  [ESP],  EAX          ; Tile 20
        push  EBP                 ; Tile 21
        push  4                   ; Tile 22
        pop   EAX                 ; Tile 23
        sub  [ESP],  EAX          ; Tile 23
        push  EBP                 ; Tile 24
        push  8                   ; Tile 25
        pop   EAX                 ; Tile 26
        add  [ESP],  EAX          ; Tile 26
        pop   EAX                 ; Tile 27
        mov  [SP], EAX            ; Tile 27
ifend:                            ; Tile 28
                                  ;  No code for tiles 29 -- 33
```

Whew! That's quite a bit of assembly for a relatively simple statement. As we will see in the next sections, this code can be optimized quite a bit. As verbose as it is, the above code does have some nice properties. The code for every type of tile is exactly the same each time that tile type appears. For example the code for a subtraction tile (tiles 3, 10, 13, and 20) is identical, regardless of where the subtraction appeared in the abstract assembly. Also, the code for different tiles is not interleaved, and the code for the tiles is output in a simple post-fix fashion.

8.5.2 Code Generation Example Using Accumulator

If we use an accumulator (but still only use very small tiles), the assembly tree in Figure 8.3 will be tiled as in Figure 8.4.

The assembly for the tiling in Figure 8.4, if we use an accumulator, is the following:

```
        mov  EAX, 4           ; Tile 2
        push EAX              ; Tile 3
        mov  EAX, EBP         ; Tile 1
        pop  EBX              ; Tile 3
        add  EAX, EBX         ; Tile 3
        mov  EAX, [EAX]       ; Tile 4
        push EAX              ; Tile 6
        mov  EAX, 5           ; Tile 5
        pop  EBX              ; Tile 6
        cmp  EBX, EAX         ; Tile 6
        setg EAX              ; Tile 6
        cmp  EAX, 0           ; Tile 7
        jne  [iftrue1]        ; Tile 7
        mov  EAX, 4           ; Tile 9
        push EAX              ; Tile 10
        mov  EAX, EBP         ; Tile 8
        pop  EBX              ; Tile 10
        add  EAX, EBX         ; Tile 10
        push EAX              ; Tile 15
        mov  EAX, 4           ; Tile 12
        push EAX              ; Tile 13
        mov  EAX, EBP         ; Tile 11
        pop  EBX              ; Tile 13
        add  EAX, EBX         ; Tile 13
        mov  EAX, [EAX]       ; Tile 14
        pop  EBX              ; Tile 15
        mov  [EBX], EAX       ; Tile 15
        jmp  [ifend1]         ; Tile 16
iftrue1:                      ; Tile 17
        mov  EAX, 4           ; Tile 19
        push EAX              ; Tile 20
        mov  EAX, EBP         ; Tile 18
        pop  EBX              ; Tile 20
        add  EAX, EBX         ; Tile 20
        push EAX              ; Tile 27
        mov  EAX, 1           ; Tile 25
        push EAX              ; Tile 26
        mov  EAX, 4           ; Tile 22
        push EAX              ; Tile 23
        mov  EAX, EBP         ; Tile 21
        pop  EBX              ; Tile 23
        add  EAX, EBX         ; Tile 23
        mov  EAX, [EAX]       ; Tile 24
        pop  EBX              ; Tile 26
        add  EAX, EBX         ; Tile 26
        pop  EBX              ; Tile 27
        mov  [EBX], EAX       ; Tile 27
ifend1:                       ; Tile 28
                              ; No code for tiles 29-33
```

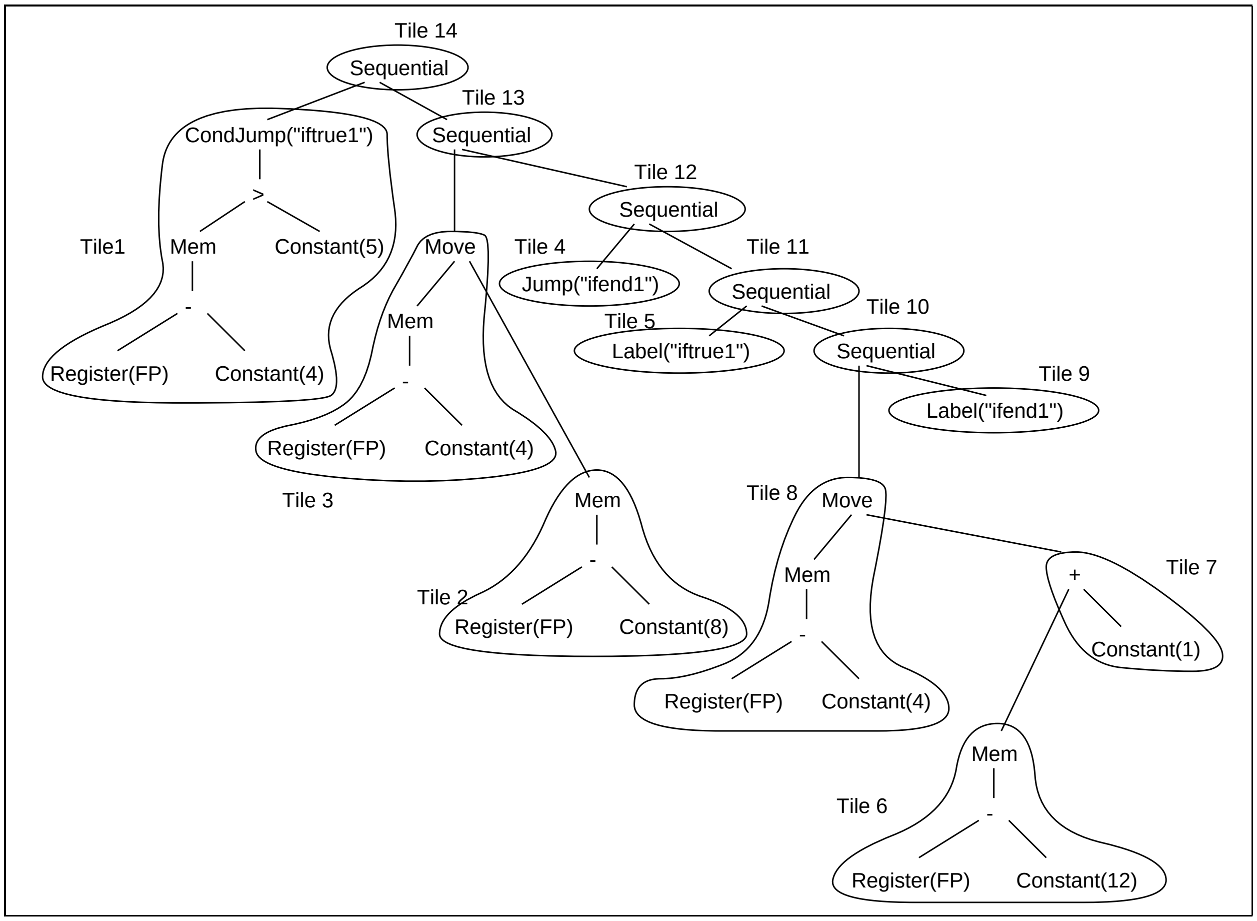


Figure 8.5: Better tiling of the abstract assembly for the simpleJava statement `if (x > 5) x = z + 1; else x = y;`, assuming x has offset 4, y has offset 8, and z has offset 12.

This generated code, while no more terse, is slightly better, since we are using registers instead of main memory more often for storing partial values. The real gains in efficiency come from using bigger tiles.

8.5.3 Code Generation Example Using Larger Tiles

If we use larger tiles to cover the tree, then the assembly tree in Figure 8.3 can be tiled as in Figure 8.5.

The assembly for the tiling in Figure 8.5 is the following:

```
        cmp  [EBP - 4], 5           ; Tile 1
        jgt  [iftrue1]              ; Tile 1
        mov   EAX,     [EBP - 8]    ; Tile 2
        mov  [EBP - 4]  EAX         ; Tile 3
        jmp  [ifend1]               ; Tile 4
iftrue1:                            ; Tile 5
        mov   EAX,     [EBP - 12]   ; Tile 6
        add   EAX,      1           ; Tile 7
        mov  [EBP - 4], EAX         ; Tile 8
ifend1:                             ; Tile 9
                                    ;  No code for tiles 10 -- 14
```

8.6 Code Generation in Java

8.6.1 Project Definition

For your final programming assignment, you will write a code generator that produces x86 assembly code. For this project you will need to:

- Create a set of tiles to cover the abstract assembly tree. Be sure that your tile set includes small tiles that can cover each single node in the assembly tree (with the exception of move nodes).
- Create the assembly code to go with each tile.
- Write a Java Visitor class that traverses the Abstract Assembly Tree, tiling the tree and emitting the proper assembly code.

Creating the tiles and associated code should be fairly straightforward, given the above notes – though some of the advanced tilings are a little subtle (the difference between x - 1 and 1 - x when trying to use the largest possible tiles, for instance). How can we create a Visitor to handle the tree tiling? The tree tiling visitor will be similar to the ASTVisitor used in semantic analysis, thought the code generation visitor will need to do some more type casting, and even make some use of the Java "instance of" operator. Each visitor function will check to see which tile should be used to cover the root of the current tree (this is where the "instance of" operator might come in handy), call the Accept methods of the various subtrees, and emit the code for the root tile. Some example Visitor methods are in Figure 8.6.

Consider the method `VisitMove` from Figure 8.6. To tile a Move tree, we must first determine if the left-hand side of the move is a register or a memory location. If it is a register, we first call the Accept method of the right-hand side of the move, which will emit assembly code that places the value to move into the register into the accumulator. We can then emit code to move the accumulator into the proper register. If the left-hand side of the move is a memory location, then we need to call the accept method of the left-hand side of the move to calculate the proper address, call the visitor of the right-hand side of the move to calculate the value to move, and then implement the move itself. Unfortunately, we do need to use typecasting and "instance of" to accomplish the tiling.

The largest difficulty for this assignment will not be creating the code generator, but debugging it. Writing and debugging a code generator that creates unoptimized assembly should take 5-10 hours. Optimizing the code generator can take an additional 10 hours, depending upon which optimizations you attempt.

8.6.2 Project "Gotcha's"

- Debugging assembly – especially machine generated assembly – can be quite difficult. Start small – build a code generator that can create unoptimized assembly for a small subset of simpleJava. Test and debug this simple code generator before expanding it to cover all of simpleJava
- Creating a code generator that just uses small tiles is easier that creating a code generator that uses complex tiles – but debugging assembly created from large tiles is much easier than debugging assembly created from small tiles.

8.6.3 Provided Files

The following files are provided in the Code Generation segment of the web site for the text:

- **AST*.java** Java files for the abstract syntax tree.
- **ASTVisitor.java** Visitor interface for ASTs.
- **AAT*.java** Java files for the abstract assembly tree.
- **AATVisitor.java** Visitor interface for AATs.

```
Object VisitMemory(AATMemory expression)
/* This method only generates very simple tiles.                    */

   expression.mem().Accept(this);
   emit("mov " + Register.ACC() + ", [" + Register.ACC() + "]");
   return null;
}

Object VisitConstant(AATConstant expression) {
  emit("mov " + Register.ACC() + "," + expression.value());
  return null;
}

Object VisitMove(AATMove statement) {
/*  This method only generates very simple tiles.                 */
   if (statement.lhs() instanceof AATRegister) {
      statement.rhs().Accept(this);
      emit("mov " + ((AATRegister) statement.lhs()).register() +
           "," +  Register.ACC() );
   } else {
      ((AATMemory) statement.lhs()).mem().Accept(this);
      emit("push " + Register.ACC() + ")
      statement.rhs().Accept(this);
      emit("pop " + Register.Tmp1());
      emit("mov [" + Register.ACC() + "], " + Register.Tmp1());
   }
}
```

Figure 8.6: Some code generation visitor methods

- **Type.java, ClassType.java, ArrayType.java, IntegerType.java, BooleanType.java, Void-Type,java** Java files to implement internal representation of types.
- **FunctionEnvironment.java, TypeEnvironment.java, VariableEnvironment.java, FunctionEntry.java, VariableEntry.java, TypeEntry.java** Java files to implement environments
- **sjc.java** A main program (for simple java compiler)
- **ASTPrint.java** Methods that print out the abstract syntax tree, to assist in debugging.
- **AATprint.java** Methods that print out the abstract assembly tree, to assist in debugging.
- **Register.java** Class that manipulates registers.
- **test1.sjava – testn.sjava** Various simpleJava programs to test your abstract syntax tree generator. Be sure to create your own test cases as well!

```
void generateExpression(AATexpression tree) {
  switch (tree->kind) {
  case AST_MEM:
    generateMemoryExp(tree);
    break;
  case AST_CONSTANT:
    emit("mov %s,%d",ACC(),tree->u.constant);
    break;
    ...  /* More cases for generateExpression */
  }
}

void generateMemoryExpression(AATexp tree) {
  /* This code only generates small, simple tiles  */
  generateExpression(tree->u.mem);
  emit("mov %s,[%s] ",ACC(),ACC());
}
```

Figure 8.7: The beginnings of the function genExp, from the file codegen.c.

8.7 Code Generation in C

8.7.1 Project Definition

For your final programming assignment, you will write a code generator that produces x86 assembly code. For this project you will need to:

- Create a set of tiles to cover the abstract assembly tree. Be sure that your tile set includes small tiles that can cover each single node in the assembly tree (with the exception of move nodes).
- Create the code to go with each tile.
- Write C code that tiles the abstract assembly tree, and emits the proper assembly code.

Creating the tiles and associated code should be fairly straightforward, given the above notes – though some of the advanced tilings are a little subtle (the difference between x - 1 and 1 - x when trying to use the largest possible tiles, for instance). How can you create write a C program to do the actual tiling and emit the correct code? You will need to write a suite of mutually recursive functions to tile the tree. The two main functions are generateExpression, which generates code for an expression tree, and generateStatement, which generates code for a statement tree. Each of these functions will check to see which is the largest tile that will cover the root of the tree, make recursive calls to tile the subtrees, and emit code for the root tile. A skeleton of codegen.c is available in the assignment directory, pieces of which are reproduced in Figures 8.7 – 8.8

The function emit works exactly like printf (except the statements are printed to the file specified by the command line argument, instead of to standard out)

8.7.2 Project Difficulty

The largest difficulty for this assignment will not be creating the code generator, but debugging it. Writing and debugging a code generator that creates unoptimized assembly should take 5-10 hours. Optimizing the code generator can take an additional 10 hours, depending upon which optimizations you attempt.

```
void generateStatement(AATstatement tree) {
  switch (tree->kind) {
  case AAT_JUMP:
    emit("jmp [%s]",tree->u.jump);
    break;
  case T_seq:
    genStm(tree->u.seq.left);
    genStm(tree->u.seq.right);
    break;
  case T_move:
    generateMove(tree);
   ...  /* more cases for statements  */
  }
}

void generateMove(AATstatement tree) {

  /* This code only generates small, simple tiles  */

  if (tree->u.move.lhs->kind == AAT_REGISTER) {
    genExp(tree->u.move.rhs);
    emit("mov  %s,%s", tree->u.move.lhs->u.reg, Acc());
  } else {
    genExp(tree->u.move.rhs);
    emit("push %s", Acc());
    genExp(tree->u.move.lhs->u.mem);
    emit("pop %s\" Temp1());
    emit("mov [%s], %s", Temp1(), Acc());
  }
}
```

Figure 8.8: The beginnings of the function generateStatement, from the file codegen.c.

8.7.3 Project “Gotcha’s”

- Debugging assembly – especially machine generated assembly – can be quite difficult. Start small – build a code generator that can create unoptimized assembly for a small subset of simpleJava. Test and debug this simple code generator before expanding it to cover all of simpleJava
- Creating a code generator that just uses small tiles is easier that creating a code generator that uses complex tiles – but debugging assembly created from large tiles is much easier than debugging assembly created from small tiles.

8.7.4 Provided Files

The following files are provided in the Assembly Tree Generation segment of the web site for the text:

- **AST.h, AST.c** Abstract syntax tree files.
- **AAT.h, AAT.c** Abstract Assembly files.
- **type.h, type.c** Internal representations of types.
- **environment.h, environment.c** C files to implement environments.
- **sjc.c** Main program (simple java compiler)
- **ASTPrint.java** Methods that print out the abstract syntax tree, to assist in debugging.
- **AATprint.java** Methods that print out the abstract assembly tree, to assist in debugging.
- **MachineDependent.h** Machine dependent constants
- **register.h, register.c** Files for manipulating registers.
- **codegen.c** A sample skeleton code generator file
- **library.s** Assembly code for library functions (print, println, read), along with some setup code to make SPIM happy.
- **test1.sjava** – **testn.sjava** Various simpleJava programs to test your abstract syntax tree generator. Be sure to create your own test cases as well!

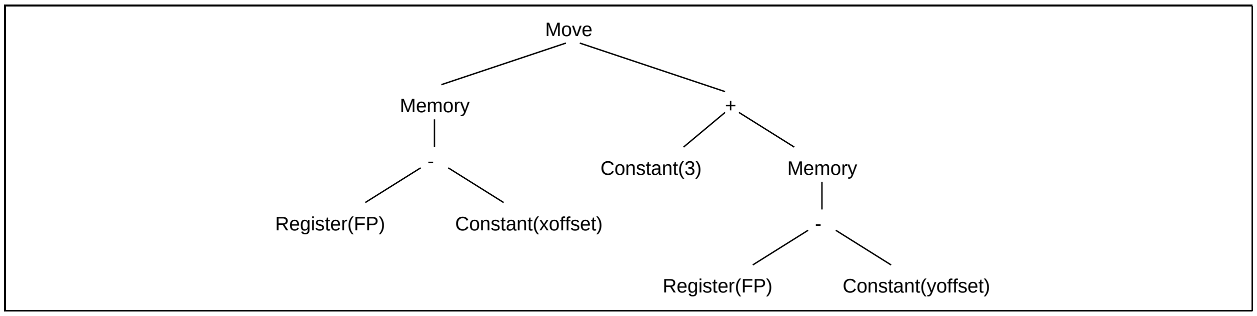


Figure 8.9: Assembly tree for `x = 3 + y`.

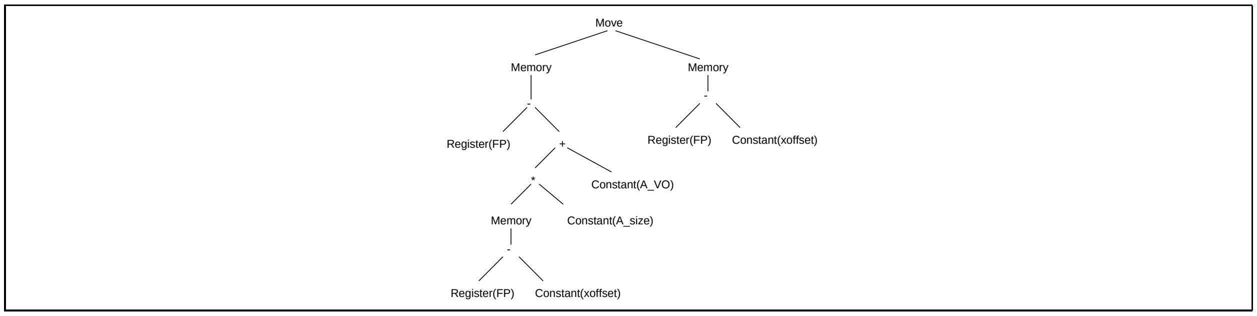


Figure 8.10: Assembly tree for `A[x] = x`.

8.8 Exercises

1. Tile the assembly tree in Figure 8.9 and show the emitted code, noting which instructions are associated with which tiles.
2. Tile the assembly tree in Figure 8.10 and show the emitted code, noting which instructions are associated with which tiles.
3. Tile the assembly tree in Figure 8.11 and show the emitted code, noting which instructions are associated with which tiles.
4. Tile the assembly tree in Figure 8.12 and show the emitted code, noting which instructions are associated with which tiles.
5. Tile the assembly tree in Figure 8.13 and show the emitted code, noting which instructions are associated with which tiles.

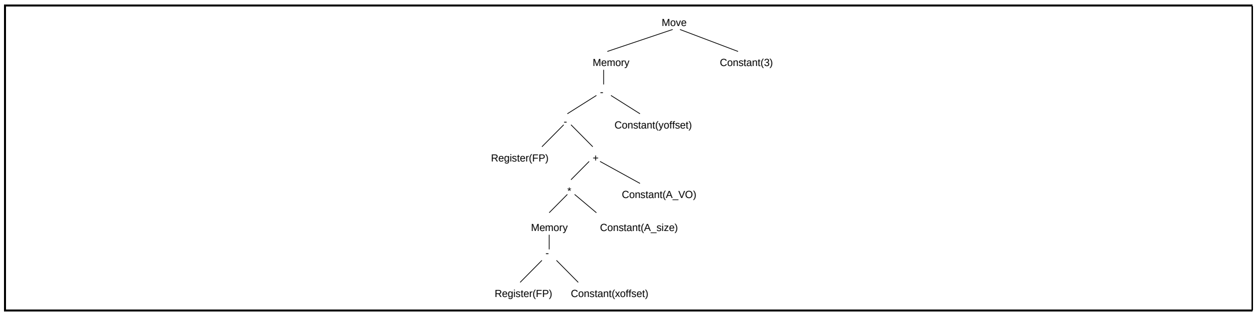


Figure 8.11: Assembly tree for `A[x].y = 3`.

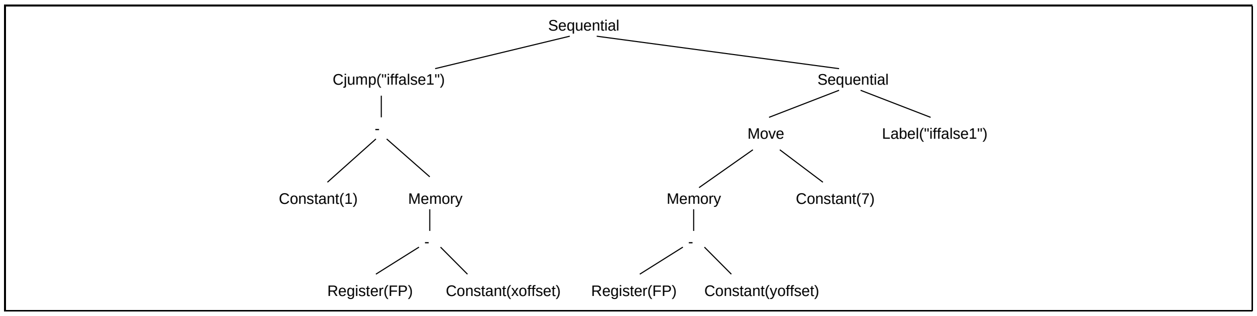


Figure 8.12: Assembly tree for `if (x) y = 7`.

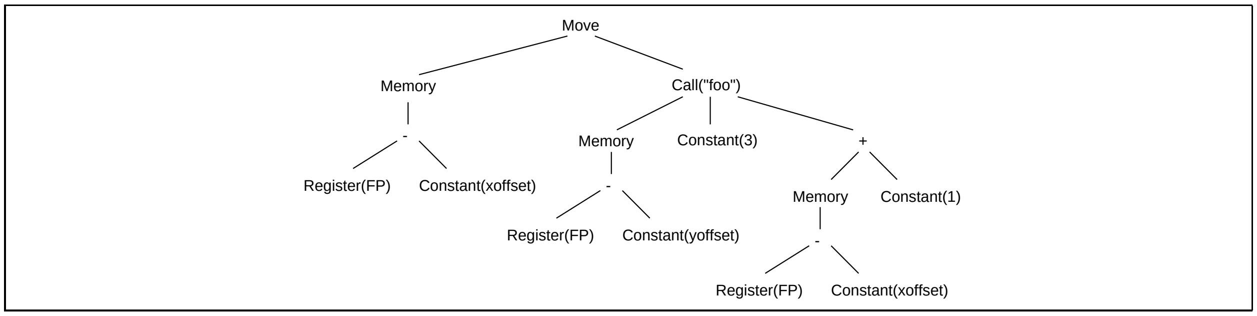


Figure 8.13: Assembly tree for `x = foo(y,3,x+1)`.

6. Tile the assembly tree in Figure 8.14 and show the emitted code, noting which instructions are associated with which tiles.

7. Consider the (fictional) load and save word assembly language instruction:

   ```
   lasw  <offset1>(base1), <offset2>(base2)
   ```

   Which has the following semantics: Add constant offset1 to the contents of register base1 to get address1. Add constant offset2 to the contents of register base2 to get address2. Store the contents of memory location address2 at memory address1.

   Give a set of tiles based on this instruction, along with the code associated with each tile. Be sure to make your tile set as diverse as possible, so that the instruction can be used to its full extent.

8. The following questions concern using a static offset off the expression stack pointer when function calls appear in an expression.

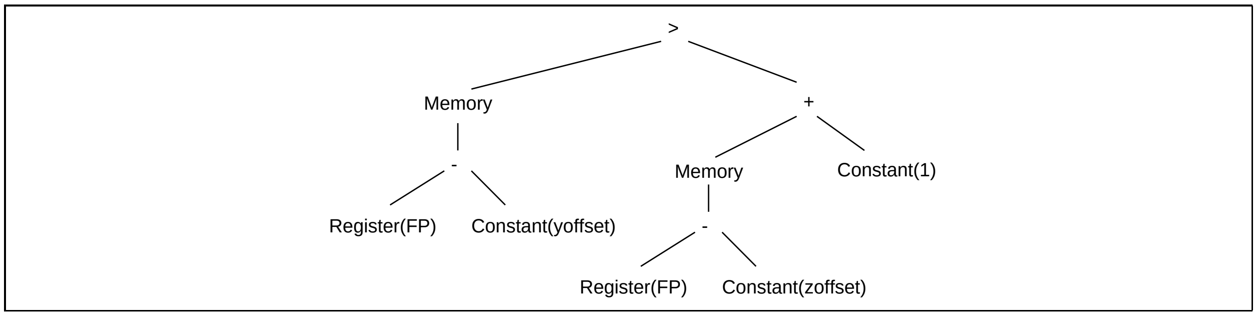


Figure 8.14: Assembly tree for `x = y > z + 1`.

(a) We might need to adjust the expression stack pointer on a jal, but we will never need to adjust the expression stack pointer on a j, bne, or any other jump. Why not?

(b) Professor Shadey has a solution to the static offset problem that does not require any modification to the expression stack pointer. His code generator will make two passes – on the first pass, it will find out the static offset that occurs each time a function is called. When he generates the code for that function, he sets the initial constant offset to the maximum of all those values. For example, if there are 3 calls to the function foo, and at the first `jal foo` the static offset is 0, at the second `jal foo` the offset is -8, and at the third `jal foo` the static offset is -4, then he will set the static offset to -8 instead of 0 at the beginning of the function foo. Under what conditions will Professor Shadey's solution work? Under what conditions will his solution fail? Be specific!

(c) Consider the assembly code that is generated for each of the following statements. For which of the statements must the expression stack be modified (and expression stack registers saved) before the jal? Explain.

   i. tmp = plus(3,4)
   ii. tmp = 1 + plus(x, y-1) /* inside the function plus */
   iii. tmp = x + times(x,y-1) /* inside the function times */
   iv. tmp = exp(x-1) + exp(x-1) /* inside function exp */