Compilers

CS414-2017S-08

Code Generation

David Galles

Department of Computer Science
University of San Francisco
Next Step: Create actual assembly code.

Use a tree tiling strategy:
- Create a set of tiles with associated assembly code.
- Cover the AST with these tiles.
- Output the code associated with each tile.

As long as we are clever about the code associated with each tile, and how we tile the tree, we will create correct actual assembly.
Our compiler will produce MIPS code
- RISC code is easier to generate than x86
- You are more familiar with RISC assembly (from Architecture and/or cs220)
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw rt, &lt;offset&gt; (base)</td>
<td>Add the constant value (&lt;offset&gt;) to the register base to get an address. Load the contents of this address into the register rt. rt = M[base + &lt;offset&gt;]</td>
</tr>
<tr>
<td>sw rt, &lt;offset&gt; (base)</td>
<td>Add the constant value (&lt;offset&gt;) to the register base to get an address. Store the contents of rt into this address. M[base + &lt;offset&gt;] = rt</td>
</tr>
<tr>
<td>add rd, rs, rt</td>
<td>Add contents of registers rs and rt, put result in register rd</td>
</tr>
</tbody>
</table>
## MIPS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub rd, rs, rt</td>
<td>Subtract contents of register rt from rs, put result in register rd</td>
</tr>
<tr>
<td>addi rt, rs, &lt;val&gt;</td>
<td>Add the constant value &lt;val&gt; to register rs, put result in register rt</td>
</tr>
<tr>
<td>mult rs, rt</td>
<td>Multiply contents of register rs by register rt, put the low order bits in register LO, and the high bits in register HI</td>
</tr>
<tr>
<td>div rs, rt</td>
<td>Divide contents of register rs by register rt, put the quotient in register LO, and the remainder in register HI</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>mflo rd</td>
<td>Move contents of the special register LOW into the register rd</td>
</tr>
<tr>
<td>j &lt;target&gt;</td>
<td>Jump to the assembly label &lt;target&gt;</td>
</tr>
<tr>
<td>jal &lt;target&gt;</td>
<td>Jump and link. Put the address of the next instruction in the Return register, and then jump to the address &lt;target&gt;. Used for function and procedure calls</td>
</tr>
<tr>
<td>jr rs</td>
<td>Jump to the address stored in register rs. Used in conjunction with jal to return from function and procedure calls</td>
</tr>
</tbody>
</table>
### MIPS

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>slt rd, rs, rt</td>
<td>if $rs &lt; rt$, $rd = 1$, else $rd = 0$</td>
</tr>
<tr>
<td>beq rs, rt, &lt;target&gt;</td>
<td>if $rs = rt$, jump to the label &lt;target&gt;</td>
</tr>
<tr>
<td>bne rs, rt, &lt;target&gt;</td>
<td>if $rs \neq rt$, jump to the label &lt;target&gt;</td>
</tr>
<tr>
<td>blez rs, &lt;target&gt;</td>
<td>if $rs \leq 0$, jump to label &lt;target&gt;</td>
</tr>
<tr>
<td>bgtz rs, &lt;target&gt;</td>
<td>if $rs &gt; 0$, jump to label &lt;target&gt;</td>
</tr>
<tr>
<td>bltz rs, &lt;target&gt;</td>
<td>if $rs &lt; 0$, jump to the label &lt;target&gt;</td>
</tr>
<tr>
<td>bgez rs, &lt;target&gt;</td>
<td>if $rs \geq 0$, jump to the label &lt;target&gt;</td>
</tr>
</tbody>
</table>
MIPS processors use 32 different registers
We will only use a subset for this project
(Though you can increase the number of registers used for temporary values fairly easily)
<table>
<thead>
<tr>
<th>Mnemonic Name</th>
<th>SPIM Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FP</td>
<td>$fp</td>
<td>Frame Pointer – Points to the top of the current activation record</td>
</tr>
<tr>
<td>$SP</td>
<td>$sp</td>
<td>Stack Pointer – Used for the activation record (stack frame) stack</td>
</tr>
<tr>
<td>$ESP</td>
<td></td>
<td>Expression Stack Pointer – The expression stack holds temporary values for expression evaluations</td>
</tr>
<tr>
<td>$result</td>
<td>$v0</td>
<td>Result Register – Holds the return value for functions</td>
</tr>
</tbody>
</table>
## 08-8: Registers

<table>
<thead>
<tr>
<th>Mnemonic Name</th>
<th>SPIM Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$return</td>
<td>$ra</td>
<td>Return Register – Holds the return address for the current function</td>
</tr>
<tr>
<td>$zero</td>
<td>$zero</td>
<td>Zero Register – This register always has the value 0</td>
</tr>
<tr>
<td>$ACC</td>
<td>$t0</td>
<td>Accumulator Register – Used for calculating the value of expressions</td>
</tr>
<tr>
<td>$t1</td>
<td>$t1</td>
<td>General Purpose Register</td>
</tr>
<tr>
<td>$t2</td>
<td>$t2</td>
<td>General Purpose Register</td>
</tr>
<tr>
<td>$t3</td>
<td>$t3</td>
<td>General Purpose Register</td>
</tr>
</tbody>
</table>
Long expressions – a * b + c * d * foo(x) – will require us to store several temporary values.

Since expressions can be arbitrarily long, we will need to store an unlimited number of partial solutions:

- Can’t always use registers – not enough of them
- Use a stack instead
For now, we will use an entirely different stack than the one for activation records

- Make debugging easier
- Later on, we can combine the stacks
08-11: **Tree Tilings**

- We will explore several different tree tiling strategies:
  - Simple tiling, that is easy to understand but produces inefficient code
  - More complex tiling, that relies less on the expression stack
  - Modifications to complex tilings, to increase efficiency
08-12: Simple Tiling

- Based on a post-order traversal of the tree
- Cover the tree with tiles
  - Each tile associated with actual assembly
- Emit the code associated with the tiles in a left-to-right, post-order traversal of the tree
• **Expression Trees**
  • The code associated with an expression tree will place the value of that expression on the top of the expression stack
08-14: Expression Trees

• Constant Expressions
  • Constant(5)
    • Push the constant 5 on the top of the expression stack
08-15: Expression Trees

- Constant Expressions
  - How can we push the constant $x$ on top of the expression stack, using MIPS assembly?
08-16: Expression Trees

• Constant Expressions
  • How can we push the constant $x$ on top of the expression stack, using MIPS assembly?

  ```assembly
  addi $t1, $zero, x
  sw $t1, 0($ESP)
  addi $ESP, $ESP, -4
  ```
Expression Trees

- Arithmetic Binary Operations

```
+  
Constant(3)  Constant(4)
```

- Instead of using a single tile to cover this tree, we will use three.
What should the code for the + tile be?

- Code for entire tree needs to push *just* the final sum on the stack
- Code for Constant Expressions push constant values on top of the stack
08-19: **Expression Trees**

- Code is emitted in a post-order traversal of the tree
  - When code for + is executed
    - The values of the left and right sub-expressions are stored on the stack
    - Right sub-expression is on the top of the stack
Expression Trees

- Code is emitted in a post-order traversal of the tree
  - When code for + is executed
    - The values of the left and right sub-expressions are stored on the stack
    - Right sub-expression is on the top of the stack
- Pop the left and right sub-expressions of the stack
- Add their values
- Push the result on the stack
08-21: Expression Trees

• Arithmetic Binary Operations

• Code for a “+” tile:

```
lw $t1, 8($ESP)     % load first operand
lw $t2, 4($ESP)     % load the second operand
add $t1, $t1, $t2   % do the addition
sw $t1, 8($ESP)     % store the result
add $ESP, $ESP, 4   % update the ESP
```
• Complete Assembly:
08-23: Expression Trees

\[
\begin{array}{c}
+ \\
\text{Constant(3)} \quad \text{Constant(4)}
\end{array}
\]

```
addi $t1, $zero, 3       # Load the constant value 3 into the register $t1
sw  $t1, 0($ESP)         # Store $t1 on the top of the expression stack
addi $ESP, $ESP, -4     # update the expression stack pointer
addi $t1, $zero, 4      # Load the constant value into 4 the register $t1
sw  $t1, 0($ESP)         # Store $t1 on the top of the expression stack
addi $ESP, $ESP, -4     # update the expression stack pointer
lw  $t1, 8($ESP)         # load the first operand into temporary $t1
lw  $t2, 4($ESP)         # load the second operand into temporary $t2
add $t1, $t1, $t2       # do the addition, storing result in $t1
sw  $t1, 8($ESP)         # store the result on the expression stack
add $ESP, $ESP, 4       # update the expression stack pointer
```
Register Trees
  - Register(FP)
  - We will cover this tree with a single tile
  - Code for the tree needs to store the register on top of the stack
08-25: Expression Trees

- Register Trees
  - Register(FP)

```
sw  $FP, 0($ESP)  % Store frame pointer
addi $ESP, ESP, -4  % Update the ESP
```
Tiling the tree:

- Register(FP)  Constant(8)
08-27: Expression Trees

- Tiling the tree:

```
- Register(FP)  Constant(8)
```
sw $FP, 0($ESP)  % Store frame pointer on the top of the expression stack
addi $ESP, $ESP, -4 % Update the expression stack pointer
addi $t1, $zero, 8  % Load the constant value 8 into the register $t1
sw $t1, 0($ESP)    % Store $t1 on the top of the expression stack
addi $ESP, $ESP, -4 % update the expression stack pointer
lw $t1, 8($ESP)    % load the first operand into temporary $t1
lw $t2, 4($ESP)    % load the second operand into temporary $t2
sub $t1, $t1, $t2  % do the subtraction, storing result in $t1
sw $t1, 8($ESP)    % store the result on the expression stack
add $ESP, $ESP, 4  % update the expression stack pointer
Relational Operations

- Relational operators – <, >, =, etc – produce boolean values
- Assembly code for a relational operator tile needs to put a 0 or 1 on the expression stack

If we tile the tree with one tile / tree node:
08-30: Expression Trees

- Relational Operations
  - Relational operators – <, >, =, etc – produce boolean values
  - Assembly code for a relational operator tile needs to put a 0 or 1 on the expression stack

```
Constant(3)  Constant(4)
```

Tile for $>$ should:

- Pop the left & right operands off the top of the stack
- Push a 1 on the stack if the first operand is greater than the second operand
- Push a 0 on the stack otherwise
  - Remember that the second operand is on the top of the stack
Tile for $>$:
- Store the left and right operands in registers $t1$ and $t2$
- $t1 > t2$ iff $t2 < t1$
  - Use slt
- Store result on the top of the stack
08-33: Expression Trees

- Tile for >:

  ```
  lw  $t1, 8($ESP)
  lw  $t2, 4($ESP)
  slt $t1, $t2, $t1
  sw  $t1, 8($ESP)
  addi $ESP, $ESP, 4
  ```
• Relational Operations
  • $\leq$, $\geq$
    • SimpleJava uses all integer operands
    • $x \leq y$ iff $(x-1) < y$
08-35: Expression Trees

- Relational Operations
  - $==$,
  - $!=$
    - Can’t use $slt$ easily
    - Use $beq$ instead
Expression Trees

- Relational Operations
  - ==, !=
    - Store the left and right operands of == in registers $t1$ and $t2$
    - If $t1 == t2$, jump to a code segment that stores 1 on top of the stack.
    - Otherwise, store a 0 on the top of the stack
08-37: Expression Trees

• Code for `==` tile:

```
lw $t1, 8($ESP)
lw $t2, 4($ESP)
beq $t1, $t2, truelab
addi $t1, 0
j endlab
truelab:
    addi $t1, 1
endlab:
    sw $t1, 8,($ESP)
    addi $ESP, $ESP, 4
```
08-38: Expression Trees

- Boolean Operations
  - AND, OR, NOT
  - Take as operators boolean values
  - Return boolean values
  - Pop off the operands, push value back on the stack
If we use 0 for false, 1 for true

- Implement (NOT x) as 1-x
- Implement (x OR y), (x AND y) in a similar fashion to <, >, etc.
  - Use s1t to calculate return value

If we use 0 for false, non-zero for true

- Implement (x OR y) as x + y
- Implement (X AND y) as x*y
- Use s1t for NOT
Expression Trees

- Memory Accesses
  - Memory node is a memory dereference
    - Pop operand into a register
    - Dereference register
    - Push result back on stack
08-41: Expression Trees

- Memory Accesses

lw $t1, 4($ESP)
lw $t1, 0($t1)
sw $t1, 4($ESP)
08-42: Expression Trees

- Memory Example

```
Memory
  /
- Register(FP)  Constant(12)
```
Expression Trees

- Memory Example

```
Memory
  -
  Register(FP)  Constant(12)
```
08-44: Expression Trees

```
sw $FP, 0($ESP)  % Store frame pointer on the top of the expression stack
addi $ESP, $ESP, -4  % Update the expression stack pointer
addi $t1, $zero, 12  % Load the constant value 12 into the register $t1
sw $t1, 0($ESP)  % Store $t1 on the top of the expression stack
addi $ESP, $ESP, -4  % Update the expression stack pointer
lw $t1, 8($ESP)  % Load the first operand into temporary $t1
lw $t2, 4($ESP)  % Load the second operand into temporary $t2
sub $t1, $t1, $t2  % Do the subtraction, storing result in $t1
sw $t1, 8($ESP)  % Store the result on the expression stack
add $ESP, $ESP, 4  % Update the expression stack pointer
lw $t1, 4($ESP)  % Pop the address to dereference off the top of
                 % the expression stack
lw $t1, 0($t1)  % Dereference the pointer
sw $t1, 4($ESP)  % Push the result back on the expression stack
```
08-45: Expression Trees

• Function calls
  • Pop off all actual parameters of the Expression Stack
  • Push actual parameters onto activation record stack
  • Jump to the start of the function (jal)
  • After function returns, push $Result register onto the Expression Stack
• Function calls

Call("Foo")

Constant(9)    Constant(3)
08-47: **Expression Trees**

- Function calls

![Expression Tree Diagram]

- Call("Foo")
- Constant(9)
- Constant(3)
08-48: Expression Trees

- Code for the Call tile

```assembly
lw    $t1    4($ESP)
sw    $t1    0($SP)
lw    $t1    8($ESP)
sw    $t1    -4($SP)
addi  $SP,   $SP,  -8
addi  $ESP,  $ESP,  8
jal   foo
addi  $SP,   $SP,   8
sw    $result,0($ESP)
addi  $ESP,  $ESP,  -4
```
• Statement Trees
• The code associated with a statement tree implements the statement described by the tree
08-50: **Statement Trees**

- Label Trees
  - We just need to output the label
- Tree: `Label("Label1")`
- Associated code:

```python
Label1:
```
Move Trees

- Left-hand side of move must be a MEMORY node or a REGISTER node
- MOVE tiles cover two nodes
  - MOVE node
  - Left child (MEMORY node or REGISTER node)
Move Trees (Moving into Registers)

- Move
  - Register(r1)
  - Constant(8)
Move Trees (Moving into Registers)

- The code for the MOVE tile needs to:
  - Pop the value to move off the stack
  - Store the value in the appropriate register
Move Trees (Moving into Registers)

```
lw    $r1, 4($ESP)
addi  $ESP, $ESP, 4
```
08-55: **Statement Trees**

- Move Trees (Moving into Registers)

```assembly
addi $t1, $zero, 8
sw $t1, 0($ESP)
addi $ESP, $ESP, -4
lw $r1, 4($ESP)
addi $ESP, $ESP, 4
```
• **Move Trees (Moving into MEMORY locations)**

```
Move
   Memory
   Constant(4)
   Register(FP)
```
Move Trees (Moving into MEMORY locations)

- The code for the MOVE tile needs to:
  - Pop the value to move off the stack
  - Pop the destination of the move off the stack
  - Store the value in the destination
08-58: Statement Trees

- Move Trees (Moving into MEMORY locations)

```
lw $t1, 8($ESP)
lw $t2, 4($ESP)
sw $t2, 0($t1)
addi $ESP, $ESP, 8
```
08-59: Statement Trees

- Move Trees (Moving into MEMORY locations)

```
sw $FP, 0($ESP)          % Store the frame pointer on the expression stack
addi $ESP, $ESP, -4     % Update the expression stack pointer
addi $t1, $ZERO, 4      % Put constant 4 into a register
sw $t1, 0($ESP)          % Store register on the expression stack
addi $ESP, $ESP, -4     % Update expression stack pointer
lw $t1, 8($ESP)          % Store the address of the lhs of the move in a register
lw $t2, 4($ESP)          % Store value of the rhs of the move in a register
sw $t2, 0($t1)           % Implement the move
addi $ESP, $ESP, 8      % update the expression stack pointer
```
08-60: Statement Trees

- Jump Trees
  - Jump trees modify the flow of control of the program
  - Can be implemented with a single j instruction
08-61: Statement Trees

- Jump Trees
  - Tree: jump("jumplab")
  - Code:

j jumplab
Conditional Jump Trees
- Evaluate the expression
- Jump if the expression != 0
08-63: Statement Trees

- Conditional Jump Trees

```
CondJump("jumplab")
<
Constant(3)  Constant(4)
```
Conditional Jump Trees

```
CondJump("jumplab")
<
Constant(3)  Constant(4)
```
Statement Trees

- Conditional Jump Trees

CondJump("jumplab")
08-66: Statement Trees

- Conditional Jump Trees

```
lw   $t1, 4($ESP)  
addi  $ESP, $ESP, 4
bgtz  $t1, jumplab
```
Statement Trees

CondJump("jumptab")

<

Constant(3)  Constant(4)
addi $t1, $zero, 3  %--
sw  $t1, 0($ESP)   % Tile for const(3)
addi $ESP, $ESP, -4 %--
addi $t1, $zero, 4  %--
sw  $t1, 0($ESP)   % Tile for const(4)
addi $ESP, $ESP -4 %--
lw  $t1, 8($ESP)   %--
lw  $t2, 4($ESP)   %
slt  $t1, $t1, $t2 % Tile for <
sw  $t1, 8($ESP)   %
addi $ESP, $ESP, 4 %--
lw  $t1, 4($ESP)   %--
addi $ESP, $ESP, 4 % Tile for CJUMP
bgtz $t1, jumplab  %--
Sequential Trees

After we have emitted code for the left and right subtrees, what do we need to do?
08-70: **Statement Trees**

- **Sequential Trees**
  - After we have emitted code for the left and right subtrees, what do we need to do?
  - Nothing!
    - Sequential trees have no associated code
Empty Statement Trees
- No action is required
- No code associated with tile for empty trees
**Improved Tiling**

- Tiling we’ve seen so far is correct – but inefficient
  - Generated code is much longer than it needs to be
  - Too heavy a reliance on the stack (main memory accesses are slow)
- We can improve our tiling in three ways:
08-73: **Improved Tiling**

- Decrease reliance on the expression stack
- Use large tiles
- Better management of the expression stack
  - Including storing the bottom of the expression stack in registers
• Decrease reliance on the expression stack
  • Every expression is stored on the stack – even when we do not need to store partial results
  • Instead, we will only use the stack when we need to store partial results – and use registers otherwise
Code for expression trees will no longer place the value on the top of the expression stack.

Instead, code for expression trees will place the value of the expression in an accumulator register (ACC).

Stack will still be necessary (in some cases) to store partial values.
Accumulator Register

- Constant trees
  - Code for a constant tree needs to place the value of the constant in the accumulator register
  - Can be accomplished by a single assembly language instruction
08-77: **Accumulator Register**

- Constant trees
  - Tree: Constant(15)
  - Code:
08-78: **Accumulator Register**

- Constant trees
  - Tree: Constant(15)
  - Code:

```
addi $ACC, $zero, 15
```
08-79: Accumulator Register

• Register trees
  • Code for a register tree needs to move the contents of the register into the accumulator register
  • Can also be accomplished by a single assembly language instruction
Accumulator Register

- Register trees
  - Tree: Register(r1)
  - Code:
08-81: Accumulator Register

- Register trees
  - Tree: Register(r1)
  - Code:

```assembly
addi $ACC, $r1, 0
```
08-82: Accumulator Register

- Binary Operators (+, -, *, etc)
  - Slightly more complicated
  - Can no longer do a simple postorder traversal
    - Emit code for left subtree – stores value in ACC
    - Emit code for right subtree – stores value in ACC – overwriting old value
  - Oops!
• Binary Operators (+, -, *, etc)
  • Use an INORDER traversal instead
    • Emit code for left subtree
    • Store this value on the stack
    • Emit code for the right subtree
    • Pop value of left operand off stack
    • Do the operation, storing result in ACC
08-84: Accumulator Register

- Binary Operators (+, -, *, etc)

```
+   
/ 
Constant(3) Constant(4)
```
Binary Operators (+, -, *, etc)

- Emit code for left subtree
- Push value on stack
- Emit code for right subtree
- Do arithmetic, storing result in ACC
08-86: **Accumulator Register**

- Binary Operators (+, -, *, etc)

```assembly
+<code for left operand>
sw $ACC, 0($ESP)
addi $ESP, $ESP, -4
<code for right operand>
lw $t1, 4($ESP)
addi $ESP, $ESP, 4
add $ACC, $t1, $ACC
```
Accumulator Register

- Binary Operators (+, -, *, etc)

```
addi $ACC, $zero, 3
sw $ACC, 0($ESP)
addi $ESP, $ESP, -4
addi $ACC, $zero, 4
lw $t1, 4($ESP)
addi $ESP, $ESP, 4
add $ACC, $t1, $ACC
```
08-88: **Accumulator Register**

- Memory Expression Trees
  - ACC points to a memory location
  - Load the contents of that memory location into the ACC
08-89: Accumulator Register

- Memory Expression Trees
- Code for a Memory tile:

```assembly
lw  $ACC, 0($ACC)
```
08-90: Accumulator Register

- Memory Example

```
Memory

- Register(FP)
  - Constant(12)
```
08-91: Accumulator Register

- Memory Example

```
Memory
  ├── Register(FP)
  │    ├── Constant(12)
```

**08-92: Accumulator Register**

- **Memory Example**

```
addi $ACC, $FP, 0     % Store the FP in the accumulator
sw  $ACC, 0($ESP)     % Store left operand on expression stack
addi $ESP, $ESP, -4  % Update expression stack pointer
addi $ACC, $zero, 12 % Store constant 4 in the accumulator
lw  $t1, 4($ESP)      % Pop left operand off expression stack
addi $ESP, $ESP, 4   % Update expression stack pointer
sub  $ACC, $t1, $ACC  % Do the addition
lw  $ACC, 0($ACC)     % Dereference the ACC
```
Accumulator Register

- Register Move Statement Trees
  - Almost the same as Register expressions
  - Move value to an arbitrary register, instead of ACC
Accumulator Register

- Register Move Statement Trees
  - Almost the same as Register expressions
  - Move value to an arbitrary register, instead of ACC

\[
\text{addi } \$r1, \$\text{ACC}, 0
\]
• Memory Move Statement Trees
  • Calculate the source & destination of the move
  • Like operator expressions, will need to store values on stack
  • Once source & destination are stored in registers, can use a `sw` statement
08-96: Accumulator Register

- Memory Move Statement Trees
08-97: **Accumulator Register**

- Memory Move Statement Trees

```assembly
<code for left subtree (destination)>
sw  $ACC,  0($ESP)
addi $ESP, $ESP,  -4
<code for right subtree (value to move)>
lw  $t1,  4($ESP)
addi $ESP, $ESP,   4
sw  $ACC,  0($t1)
```
08-98: **Accumulator Register**

- Memory Move Example

![Diagram](image_url)
08-99: Accumulator Register

• Memory Move Example

- Memory
- Constant(4)
- Register(FP)
- Move
Accumulator Register

- Memory Move Example

```
addi $ACC, $FP, 0       % Code for Register(FP) tile
sw    $ACC, 0($ESP)    % Store destination on expression stack
addi $ESP, $ESP, -4    % Update expression stack pointer
addi $ACC, $zero, 4    % Code for Constant(4) tile
lw    $t1, 4($ESP)     % Load destination into a register
addi $ESP, $ESP, 4     % Update expression stack pointer
sw    $ACC, 0($t1)     % Implement the move
```
Function & Procedure Calls
- Move arguments to the call stack as they are computed
  - No need to use the expression stack at all
08-102: Accumulator Register

- **Function Calls**

```assembly
sw $ACC, -$4n-4($SP)   % Store first argument on the call stack
<Code for second argument>
sw $ACC, -$4n-8($SP)   % Store second argument on the call stack
...
<Code for nth argument>
sw $ACC, 0($SP)        % Store nth argument on the call stack
addi $SP, $SP, -$4n    % Update call stack pointer
jal foo                % Make the function call
addi $SP, $SP, $4n     % Update call stack pointer
addi $ACC, $result, 0  % Store result of function in accumulator
```
08-103: **Accumulator Register**

- **Procedure Calls**

  ```assembly
  sw  $ACC, <-4n-4>($SP) % Store first argument on the call stack
  <Code for second argument>
  sw  $ACC, <-4n-8>($SP) % Store second argument on the call stack
  ...
  <Code for nth argument>
  sw  $ACC, 0($SP) % Store nth argument on the call stack
  addi $SP, $SP, <-4n> % Update call stack pointer
  jal foo % Make the function call
  addi $SP, $SP, <4n> % Update call stack pointer
  ```
08-104: Accumulator Register

- Conditional Jumps
  - No temporary values need to be saved
  - Jump if ACC is not zero
Conditional Jumps

- No temporary values need to be saved
- Jump if ACC is not zero

<Code for conditional expression>
```
bgtz $ACC, jumplab
```
Accumulator Register

- Jumps, labels, sequential statements
  - Do not have subtrees with values
  - Code is the same as previously defined
Instead of covering a single node for each tile, cover several nodes with the same tile.

As long as the code associated with the larger tile is more efficient than the code associated with all of the smaller tiles, we gain efficiency.
08-108: Larger Tiles Example

- Memory Move Expression

```
  Move
 /     \
|      |
Memory  Constant(7)
```

```
  
   Register(FP)  Constant(4)
```


08-109: Larger Tiles Example

- Standard Tiling

```
Tile 1
  Register(FP)
```

```
Tile 2
  Constant(4)
```

```
Tile 3
  -
```

```
Tile 4
  Constant(7)
```

```
Tile 5
  Move
```

```
Memory
```

![Diagram](https://via.placeholder.com/150)
Standard Tiling

addi $ACC, $FP, 0  % code for tile 1
sw  $ACC, $ESP, 0  % code for tile 3
addi $ESP, $ESP, -4  % code for tile 3
addi $ACC, $zero, 4  % code for tile 2
lw  $t1, 4($ESP)  % code for tile 3
addi $ESP, $ESP, 4  % code for tile 3
sub  $ACC, $t1, $ACC  % code for tile 3
sw  $ACC, $ESP, 0  % code for tile 5
addi $ESP, $ESP, -4  % code for tile 5
addi $ACC, $zero, 7  % code for tile 4
lw  $t1, 4($ESP)  % code for tile 5
addi $ESP, $ESP, 4  % code for tile 5
sw  $ACC, 0($t1)  % code for tile 5
08-111: Larger Tiles Example

- Memory Move Expression

```
    Move
   /   \
Memory  Constant(7)
   |
   -
  /  |
Register(FP)  Constant(4)
```
08-112: Larger Tiles Example

- Using Larger Tiles

```
Tile 2
Memory

Move

Constant(7)

Tile 1

Register(FP)

Costant(4)
```
• Code for Tile 2?
Larger Tiles Example

- Code for Tile 2?

`sw $ACC, -4($FP)`
08-115: Larger Tiles Example

Tile 2
Memory

Move

Tile 1
Constant(7)

Register(FP)

Costant(4)

addi $ACC, $zero 7 % tile 1
sw $ACC, -4($FP) % tile 2
Can get a huge saving using larger tiles

Especially if tile size is geared to functionality of the actual assembly

- _sw_ Stores the value in a register in a memory location pointed to by an offset off a different register
- Tile that takes full advantage of this functionality will lead to efficient assembly
08-117: Larger Tiles

- Design tiles based on the actual assembly language
- Take advantage of as many feature of the language as possible
- Create tiles that are as large as possible, that can be implemented with a single assembly language instruction
  - Plus some extra instructions to do stack maintenance
08-118: Larger Tiles – Examples
08-119: Larger Tiles – Examples

```
sw r1 -x(r2)
```
08-120: Larger Tiles – Examples

Move
Memory
Subtree

Register(r2)
Costant(x)
Move

Memory

Subtree

Register(r2)

Costant(x)

<code for Subtree>
sw $ACC, -x(r2)
08-122: Larger Tiles – Examples

![Diagram showing relationships between Move, Memory, Subtree1, Subtree2, and Constant(x).]
08-123: Larger Tiles – Examples

Move
Memory
Subtree1
Costant(x)
Subtree2

<code for Subtree1>
sw $ACC, 0($ESP)
addi $ESP, $ESP
<code for Subtree2>
lw $t1, 4($ESP)
addi $ESP, $ESP, 4
sw $ACC, -x($t1)
08-124: Larger Tiles – Examples

- Move
- Memory
- register(r1)
- Subtree
<code for Subtree>
sw  $r1, -x($ACC)
08-126: Larger Tiles – Examples

Diagram:
- Move
- Memory
- Subtree1
- Subtree2
08-127: Larger Tiles – Examples

Subtree1
<code for Subtree1>
sw $ACC, 0($ESP)
addi $ESP, $ESP
<code for Subtree2>
lw $t1, 4($ESP)
addi $ESP, $ESP, 4
sw $ACC, 0($t1)

Move
Memory
Subtree2
Larger Tiles

- Larger tiles are better
- Why design small tiles as well as large tiles?
08-129: Larger Tiles

- Larger tiles are better
- Why design small tiles as well as large tiles?
  - Might not always be able to use largest tile
- Why design a range of tile sizes
Larger Tiles

- Larger tiles are better
- Why design small tiles as well as large tiles?
  - Might not always be able to use largest tile
- Why design a range of tile sizes
  - Most efficient tiling of any tree
• Conditional Jump Trees

\[
\text{CondJump("jumplab") = Constant(3) Constant(4)}
\]
08-132: Larger Tiles

- Conditional Jump Trees

```
CondJump("jumplab")
```

```
Tile 1
Constant(3)

Tile 3
= 4

Tile 2
Constant(4)

Tile 4
```
08-133: Larger Tiles

```
addi $ACC, $zero 3          % Tile 1
sw   $ACC, 0($ESP)          % Tile 3
addi $ESP, $ESP, -4        % Tile 3
addi $ACC, $zero, 4        % Tile 2
lw    $t1, 4($ESP)          % Tile 3
addi $ESP, $ESP, 4         % Tile 3
beq   $t1, $ACC, truelab1  % Tile 3
addi $ACC, $zero, 0        % Tile 3
j     endlab1              % Tile 3

truelab1:                   % Tile 3
  addi $ACC, $zero, 1       % Tile 3

endlab1:                    % Tile 3
  bgtz $ACC, jumplab        % Tile 4
```
08-134: Larger Tiles

- We are doing two conditional jumps
  - Set the boolean value, 0 or 1
  - Implement the actual jump
- Using larger tiles, we can remove one of them:
  - Jump directly to “jumplab” if the expression is true
Larger Tiles

- Conditional Jump Trees

```
Cjump("jumplab") = Constant(3) Constant(4)
```

Diagram:

```
Tile 1  Constant(3)  --- Cjump("jumplab") --- Constant(4)  --- Tile 3
```

Tile 2
Larger Tiles

Conditional Jump Trees

```
addi $ACC, $zero 3  % Tile 1
sw  $ACC, 0($ESP)   % Tile 3
addi $ESP, $ESP, -4 % Tile 3
addi $ACC, $zero, 4 % Tile 2
lw   $t1, 4($ESP)   % Tile 3
addi $ESP, $ESP, 4 % Tile 3
beq  $t1, $ACC, jumplab % Tile 3
```
08-137: Larger Tiles

- Conditional Jump Trees
  - For some conditional jump trees, we can do even better

```
CondJump("jumplab") < Constant(0)
```
Conditional Jump Trees

For some conditional jump trees, we can do even better

```
CondJump("jumplab")
<
Constant(0)
<Code for left subtree>
bltz $ACC, jumplab
```
08-139: Larger Tiles

- Given a range of tiles, we have a choice as to which tile to pick.
- How do we decide which tiles to use, to minimize the total number of tiles?
  - Under the assumption that each tile uses about the same amount of assembly.
08-140: Tiling the Tree

- Greedy Strategy:
  - Cover the root of the tree with the largest possible tile
  - Recursively tile subtrees
- Are we always guaranteed that we will find a tiling this way (that is, can we ever get stuck?)
Greedy Strategy:
- Cover the root of the tree with the largest possible tile
- Recursively tile subtrees

Are we always guaranteed that we will find a tiling this way (that is, can we ever get stuck?)
- We can always find a tiling this way – provided we include all unit-sized tiles in our tile set
Optimizing Expression Stack

- We spend more operations than necessary manipulating the expression stack.
- Streamlining stack operations will save us some assembly language instructions (and thus some time).
08-143: Constant Stack Offsets

- Every time we push an item on the Expression Stack, need to increment the $ESP
- Every time we pop an item off the Expression Stack, need to increment the $ESP
- We know at compile time how deep the stack is
  - Can use a constant offset off the $ESP
  - Never need to change the $ESP
Constant Stack Offsets

- Pushing an expression:
  
  ```
  sw $ACC, 0($ESP)
  addi $ESP, $ESP, -4
  ```

- Popping an expression:
  
  ```
  addi $ESP, $ESP, 4
  lw $ACC, 0($ESP)
  ```
08-145: **Constant Stack Offsets**

- Pushing an expression:
  
  ```assembly
  sw  $ACC, <offset>($ESP)
  ```
  (decement <offset> by 4)

- Popping an expression:
  (increment <offset> by 4)

  ```assembly
  lw  $ACC, <offset>($ESP)
  ```
Example:

```
<table>
<thead>
<tr>
<th>Tile 1</th>
<th>Tile 2</th>
<th>Tile 3</th>
<th>Tile 4</th>
<th>Tile 5</th>
<th>Tile 6</th>
<th>Tile 7</th>
<th>Tile 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
<td>Mem</td>
</tr>
<tr>
<td>Register(FP)</td>
<td>Register(FP)</td>
<td>Register(FP)</td>
<td>Register(FP)</td>
<td>Register(FP)</td>
<td>Register(FP)</td>
<td>Register(FP)</td>
<td>Register(FP)</td>
</tr>
<tr>
<td>Constant(xoffset)</td>
<td>Constant(xoffset)</td>
<td>Constant(xoffset)</td>
<td>Constant(xoffset)</td>
<td>Constant(xoffset)</td>
<td>Constant(xoffset)</td>
<td>Constant(xoffset)</td>
<td>Constant(xoffset)</td>
</tr>
<tr>
<td>Constant(9)</td>
<td>Constant(8)</td>
<td>Constant(7)</td>
<td>Constant(6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
08-147: Constant Stack Offsets

```assembly
addi $ACC, $zero, 9 % Tile 1
sw $ACC, 0($ESP) % Tile 7 -- pushing a value on the
addi $ESP, $ESP, -4 % Tile 7 expression stack
addi $ACC, $zero, 8 % Tile 2
sw $ACC, 0($ESP) % Tile 6 -- pushing a value on the
addi $ESP, $ESP, -4 % Tile 6 expression stack
addi $ACC, $zero, 7 % Tile 3
sw $ACC, 0($ESP) % Tile 5 -- pushing a value on the
addi $ESP, $ESP, -4 % Tile 5 expression stack
addi $ACC, $zero, 6 % Tile 4
addi $ESP, $ESP, 4 % Tile 5 -- popping a value off the
lw $t1, 0($ESP) % Tile 5 expression stack
sub $ACC, $t1, $ACC % Tile 5
addi $ESP, $ESP, 4 % Tile 6 -- popping a value off the
lw $t1, 0($ESP) % Tile 6 expression stack
sub $ACC, $t1, $ACC % Tile 6
addi $ESP, $ESP, 4 % Tile 7 -- popping a value off the
lw $t1, 0($ESP) % Tile 7 expression stack
sub $ACC, $t1, $ACC % Tile 7
lw $ACC, xoffset($FP) % Tile 8
```
08-148: Constant Stack Offsets

```
addi $ACC, $zero, 9           % Tile 1
sw   $ACC, 0($ESP)            % Tile 7 -- pushing on expression stack
addi $ACC, $zero, 8           % Tile 2
sw   $ACC, -4($ESP)           % Tile 6 -- pushing on expression stack
addi $ACC, $zero, 7           % Tile 3
sw   $ACC, -8($ESP)           % Tile 5 -- pushing on expression stack
addi $ACC, $zero, 6           % Tile 4
lw    $t1, -8($ESP)           % Tile 5 -- popping from expression stack
sub  $ACC, $t1, $ACC          % Tile 5
lw    $t1, -4($ESP)           % Tile 6 -- popping from expression stack
sub  $ACC, $t1, $ACC          % Tile 6
lw    $t1, 0($ESP)            % Tile 7 -- popping from expression stack
sub  $ACC, $t1, $ACC          % Tile 7
lw    $ACC, xoffset($FP)      % Tile 8
```
Using constant offsets off the $ESP words well most of the time

There are problems with function calls, however

\[ x = y + (z + \text{bar}(2)) \]
## Constant Stack Offsets

### x = y + (z + bar(2))

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $ACC, -8($FP)</td>
<td>Load y into the ACC</td>
</tr>
<tr>
<td>sw $ACC, 0($ESP)</td>
<td>Store value of y on the expression stack</td>
</tr>
<tr>
<td>lw $ACC, -12($FP)</td>
<td>Load z into the ACC</td>
</tr>
<tr>
<td>sw $ACC, -4($ESP)</td>
<td>Store value of z on the expression stack</td>
</tr>
<tr>
<td>addi $ACC, $zero, 2</td>
<td>Load actual parameter (2) into ACC</td>
</tr>
<tr>
<td>sw $ACC, 0($SP)</td>
<td>Store actual parameter on the stack</td>
</tr>
<tr>
<td>addi $SP, $SP, -4</td>
<td>Adjust the stack pointer</td>
</tr>
<tr>
<td>jal bar</td>
<td>Make the function call</td>
</tr>
<tr>
<td>addi $SP, $SP, 4</td>
<td>Adjust the stack pointer</td>
</tr>
<tr>
<td>addi $ACC, $result, 0</td>
<td>Store result of function call in the ACC</td>
</tr>
<tr>
<td>lw $t1, -4($ESP)</td>
<td>Load stored value of z into t1</td>
</tr>
<tr>
<td>add $ACC, $t1, $ACC</td>
<td>Do the addition z + bar(2)</td>
</tr>
<tr>
<td>lw $t1, 0($ESP)</td>
<td>Load stored value of y into t1</td>
</tr>
<tr>
<td>addi $ACC, $t1, $ACC</td>
<td>Do the addition y + (z + bar(2))</td>
</tr>
<tr>
<td>sw $ACC, -4($FP)</td>
<td>Store value of addition in x</td>
</tr>
</tbody>
</table>

### What’s wrong with this code?
What’s wrong with this code?

When we call the function, constant offset is -8.
  - There are 2 expressions stored beyond the top of the $ESP

In the body of the function, constant offset is ..
What’s wrong with this code?

When we call the function, constant offset is -8.
  - There are 2 expressions stored beyond the top of the $ESP

In the body of the function, constant offset is 0!

If `bar` uses the expression stack, it will clobber the values we’ve stored on it!
08-153: Constant Stack Offsets

• Problem:
  • Function calls expect constant offset to be 0 at start of the function
  • Actual constant offset may not be 0
    • May be arbitrarily large (why?)

• Solution:
08-154: **Constant Stack Offsets**

- **Problem:**
  - Function calls expect constant offset to be 0 at start of the function
  - Actual constant offset may not be 0
    - May be *arbitrarily large* (why?)

- **Solution:**
  - Before a function call, decrement the $ESP by constant offset
    - Constant offset is now 0 again
  - After the function call, increment the $ESP again
Constant Stack Offsets

\[ x = y + (z + \text{bar}(2)) \] (Corrected)

```
lw $ACC, -8($FP)    % Load y into the ACC
sw $ACC, 0($ESP)    % Store value of y on the expression stack
lw $ACC, -12($FP)   % Load z into the ACC
sw $ACC, -4($ESP)   % Store value of z on the expression stack
addi $ACC, $zero, 2 % Load actual parameter (2) into ACC
sw $ACC, 0($SP)     % Store actual parameter on the stack
addi $SP, $SP, -4   % Adjust the stack pointer
addi $ESP, $ESP, -8 % ** Adjust the expression stack pointer **
jal bar              % Make the function call
addi $ESP, $ESP, 8   % ** Adjust the expression stack pointer **
addi $SP, $SP, 4     % Adjust the stack pointer
addi $ACC, $result, 0 % Store result of function call in the ACC
lw $t1, -4($ESP)     % Load stored value of z into t1
add $ACC, $t1, $ACC  % Do the addition z + \text{bar}(2)
lw $t1, 0($ESP)      % Load stored value of y into t1
addi $ACC, $t1, $ACC % Do the addition y + (z + \text{bar}(2))
sw $ACC, -4($FP)     % Store value of addition in x
```
Like to store temporary values in Registers instead of in main memory

Can’t store all temporary values in Registers
  • Arbitrarily large number of temporary values may be required
  • Limited number of registers

Can store some temporary values in registers
08-157: **Using Registers**

- Store the bottom of the expression stack in registers
  - For small expressions, we will not need to use main memory for temporary values
  - Retain the flexibility to handle large expressions
- Bottom $x$ elements of the expression stack in registers
Using Registers

Example:

- Use two temporary registers $r2 & $r3
- If we only need two temporary values, use $r2 and $r3
- When more values are required, use stack
Using Registers
Using Registers

- Constant stack offsets (no registers)

```assembly
addi $ACC, $zero, 9                 % Tile 1
sw $ACC, 0($ESP)                    % Tile 7 -- pushing on expression stack
addi $ACC, $zero, 8                 % Tile 2
sw $ACC, -4($ESP)                   % Tile 6 -- pushing on expression stack
addi $ACC, $zero, 7                 % Tile 3
sw $ACC, -8($ESP)                   % Tile 5 -- pushing on expression stack
addi $ACC, $zero, 6                 % Tile 4
lw $t1, -8($ESP)                    % Tile 5 -- popping from expression stack
sub $ACC, $t1, $ACC                 % Tile 5
lw $t1, -4($ESP)                    % Tile 6 -- popping from expression stack
sub $ACC, $t1, $ACC                 % Tile 6
lw $t1, 0($ESP)                     % Tile 7 -- popping from expression stack
sub $ACC, $t1, $ACC                 % Tile 7
lw $ACC, xoffset($FP)               % Tile 8
```
Bottom of expression stack in registers

addi $ACC, $zero, 9 % Tile 1
addi $r2, $ACC, 0 % Tile 7 -- pushing on expression stack
addi $ACC, $zero, 8 % Tile 2
addi $r3, $ACC, 0 % Tile 6 -- pushing on expression stack
addi $ACC, $zero, 7 % Tile 3
sw $ACC, 0($ESP) % Tile 5 -- pushing on expression stack
addi $ACC, $zero, 6 % Tile 4
lw $t1, 0($ESP) % Tile 5 -- popping from expression stack
sub $ACC, $t1, $ACC % Tile 5
addi $t1 $t3, 0 % Tile 6 -- popping from expression stack
sub $ACC, $t1, $ACC % Tile 6
addi $t1, $t2, 0 % Tile 7 -- popping from expression stack
sub $ACC, $t1, $ACC % Tile 7
lw $ACC, xoffset($FP) % Tile 8
08-162: Using Registers

- If we store the bottom of the expression stack in registers, we have a problem with function calls:
  - \( x = \text{foo}(a,b) + \text{foo}(c,d) \)
- What can we do?
If we store the bottom of the expression stack in registers, we have a problem with function calls:

- \( x = \text{foo}(a,b) + \text{foo}(c,d) \)

What can we do?

- On a function call, push all registers onto the expression stack, and update the expression stack pointer.
- After a function call, pop all values back into the registers, and update the expression stack pointer.
Using Registers
08-165: Using Registers

```
addi $ACC, $zero, 9  % Tile 1
addi $r2, $ACC, 0   % Tile 7 -- pushing on expression stack
addi $ACC, $zero, 8 % Tile 2
addi $r3, $ACC, 0   % Tile 6 -- pushing on expression stack
addi $ACC, $zero, 7 % Tile 3
sw  $ACC, 0($ESP)   % Tile 5 -- pushing on expression stack
sw  $r2, -4($ESP)   % Tile 4 -- Push register on ESP
sw  $r3, -8($ESP)   % Tile 4 -- Push register on ESP
addi $ESP, $ESP, -12 % Tile 4 -- Update ESP
jal  foo            % Tile 4
addi $ACC, $result, 0 % Tile 4
addi $ESP, $ESP, 12 % Tile 4 -- Update ESP
lw  $r2, -4($ESP)   % Tile 4 -- Pop register off ESP
lw  $r3, -8($ESP)   % Tile 4 -- Pop register off ESP
lw  $t1, 0($ESP)    % Tile 5 -- popping from expression stack
sub $ACC, $t1, $ACC % Tile 5
addi $t1 $t3, 0    % Tile 6 -- popping from expression stack
sub $ACC, $t1, $ACC % Tile 6
addi $t1 $t2, 0    % Tile 7 -- popping from expression stack
sub $ACC, $t1, $ACC % Tile 7
lw  $ACC, xoffset($FP) % Tile 8
```
if (x > 1)
    x = z + 1;
else
    x = y;

• x has offset 4
• y has offset 8
• z has offset 12
08-167: Extended Example
Extended Example – Basic

```
sw $FP, 0($ESP)     % Tile 1
addi $ESP, $ESP, -4 % Tile 1
addi $t1, $zero, 4  % Tile 2
sw $t1, 0($ESP)     % Tile 2
addi $ESP, $ESP, -4 % Tile 2
lw  $t1, 4($ESP)    % Tile 3
lw  $t2, 8($ESP)    % Tile 3
sub $t1, $t2, $t1   % Tile 3
sw $t1, 8($ESP)     % Tile 3
addi $ESP, $ESP, 4  % Tile 3
lw  $t1, 4($ESP)    % Tile 4
lw  $t1, 0($t1)     % Tile 4
sw $t1, 4($ESP)     % Tile 4
addi $t1, $zero, 5  % Tile 5
sw $t1, 0($ESP)     % Tile 5
addi $ESP, $ESP, -4 % Tile 5
lw  $t1, 8($ESP)    % Tile 6
lw  $t2, 4($ESP)    % Tile 6
slt $t1, $t2, $t1   % Tile 6
addi $ESP, $ESP, 4  % Tile 6
```
Extended Example – Basic

lw $t1, 4($ESP)     % Tile 7
addi $ESP, $ESP, 4  % Tile 7
bgtz $t1, iftrue1   % Tile 7
sw $FP, 0($ESP)     % Tile 8
addi $ESP, $ESP, -4 % Tile 8
addi $t1, $zero, 4  % Tile 9
sw $t1, 0($ESP)     % Tile 9
addi $ESP, $ESP, -4 % Tile 9
lw $t1, 4($ESP)     % Tile 10
lw $t2, 8($ESP)     % Tile 10
sub $t1, $t2, $t1   % Tile 10
sw  $t1,  8($ESP)  % Tile 10
addi $ESP, $ESP,  4  % Tile 10
sw  $FP,  0($ESP)   % Tile 11
addi $ESP, $ESP,  -4 % Tile 11
addi $t1,  $zero,  8  % Tile 12
sw  $t1,  0($ESP)   % Tile 12
addi $ESP, $ESP,  -4 % Tile 12
lw  $t1,  4($ESP)   % Tile 13
lw  $t2,  8($ESP)   % Tile 13
sub  $t1,  $t2,  $t1 % Tile 13
sw  $t1,  8($ESP)   % Tile 13
addi $ESP, $ESP,  4  % Tile 13
lw  $t1,  4($ESP)   % Tile 14
lw  $t1,  0($t1)    % Tile 14
sw  $t1,  4($ESP)   % Tile 14
lw  $t1,  8($ESP)   % Tile 14
lw  $t2,  4($ESP)   % Tile 15
sw  $t2,  0($t1)    % Tile 15
addi $ESP, $ESP,  8  % Tile 15
j   Ifend1        % Tile 16
iftrue1:

    sw $FP, 0($ESP)            % Tile 17
    addi $ESP, $ESP, -4       % Tile 18
    addi $t1, $zero, 4        % Tile 19
    sw $t1, 0($ESP)           % Tile 19
    addi $ESP, $ESP, -4       % Tile 19
    lw $t1, 4($ESP)           % Tile 20
    lw $t2, 8($ESP)           % Tile 20
    sub $t1, $t2, $t1         % Tile 20
    sw $t1, 8($ESP)           % Tile 20
    addi $ESP, $ESP, 4        % Tile 20
    sw $FP, 0($ESP)           % Tile 21
    addi $ESP, $ESP, -4       % Tile 21
    addi $t1, $zero, 8        % Tile 22
    sw $t1, 0($ESP)           % Tile 22
    addi $ESP, $ESP, -4       % Tile 22
    lw $t1, 4($ESP)           % Tile 23
    lw $t2, 8($ESP)           % Tile 23
    sub $t1, $t2, $t1         % Tile 23
    sw $t1, 8($ESP)           % Tile 23
addi $ESP, $ESP, 4  % Tile 23
lw  $t1,  4($ESP)  % Tile 24
lw  $t1,  0($t1)  % Tile 24
sw  $t1,  4($ESP)  % Tile 24
addi $t1, $zero, 4  % Tile 25
sw  $t1,  0($ESP)  % Tile 25
addi $ESP, $ESP, -4  % Tile 25
lw  $t1,  4($ESP)  % Tile 26
lw  $t2,  8($ESP)  % Tile 26
add  $t1,  $t2,  $t1  % Tile 26
sw  $t1,  8($ESP)  % Tile 26
addi $ESP, $ESP, 4  % Tile 26
lw  $t1,  8($ESP)  % Tile 27
lw  $t2,  4($ESP)  % Tile 27
sw  $t2,  0($t1)  % Tile 27
addi $ESP, $ESP, 8  % Tile 27
Ifend1:  % Tile 28
% No code for tiles 29 -- 33
08-174: **Extended Eg. – Accumulator**
addi $ACC, $FP, 0 % Tile 1
sw $ACC, 0($ESP) % Tile 3
addi $ESP, $ESP, -4 % Tile 3
addi $ACC, $zero, 4 % Tile 2
lw $t1, 4($ESP) % Tile 3
addi $ESP, $ESP, 4 % Tile 3
sub $ACC, $t1, $ACC % Tile 3
lw $ACC, 0($ACC) % Tile 4
sw $ACC, 0($ESP) % Tile 6
addi $ESP, $ESP, -4 % Tile 6
addi $ACC, $zero, 5 % Tile 5
lw $t1, 4($ESP) % Tile 6
addi $ESP, $ESP, 4 % Tile 6
slt $ACC, $ACC, $t1 % Tile 6
bgtz $ACC, iftrue1 % Tile 7
addi $ACC, $FP, 0 % Tile 8
sw $ACC, 0($ESP) % Tile 10
addi $ESP, $ESP, -4 % Tile 10
addi $ACC, $zero, 4 % Tile 9
lw  $t1, 4($ESP) % Tile 10
sub $ACC, $t1, $ACC % Tile 10
sw $ACC, 0($ESP) % Tile 15
addi $ESP, $ESP, -4 % Tile 15
addi $ACC, $FP, 0 % Tile 11
sw $ACC, 0($ESP) % Tile 13
addi $ESP, $ESP, -4 % Tile 13
addi $ACC, $zero, 8 % Tile 12
lw  $t1, 4($ESP) % Tile 13
addi $ESP, $ESP, 4  % Tile 13
sub $ACC, $t1, $ACC % Tile 13
lw  $ACC, 0($ACC)  % Tile 14
lw  $t1, 4($ESP)  % Tile 15
addi $ESP, $ESP, 4  % Tile 15
sw $ACC, 0($t1)  % Tile 15
iftrue:
    addi $ACC, $FP, 0         % Tile 18
    sw  $ACC, 0($ESP)          % Tile 20
    addi $ESP, $ESP, -4       % Tile 20
    addi $ACC, $zero, 4       % Tile 19
    lw   $t1, 0($ESP)         % Tile 20
    addi $ESP, $ESP, 4        % Tile 20
    sub  $ACC, $t1, $ACC      % Tile 20
    sw  $ACC, 0($ESP)          % Tile 27
    addi $ESP, $ESP, -4       % Tile 27
    addi $ACC, $FP, 0         % Tile 21
    sw  $ACC, 0($ESP)          % Tile 23
    addi $ESP, $ESP, -4       % Tile 23
    addi $ACC, $zero, 12      % Tile 22
    lw   $t1, 4($ESP)          % Tile 23
    addi $ESP, $ESP, 4        % Tile 23
    add  $ACC, $t1, $ACC      % Tile 23
    lw   $ACC, 0($ACC)         % Tile 24
    sw  $ACC, 0($ESP)          % Tile 26
  j  ifend1                   % Tile 16
addi $ESP, $ESP, -4  % Tile 26
addi $ACC, $zero  1  % Tile 25
lw  $t1, 4($ESP)  % Tile 26
addi $ESP, $ESP,  4  % Tile 26
add  $ACC, $t1,  $ACC  % Tile 26
lw  $t1, 4($ESP)  % Tile 27
addi $ESP, $ESP, 4  % Tile 27
sw  $ACC, 0($t1)  % Tile 27
ifend1:  % Tile 28

% No code for tiles 29 -- 33
08-179: Extended Eg. – Large Tiles
08-180: Extended Eg. – Large Tiles
lw $ACC, -4($FP)       % Tile 1
sw $ACC, 0($ESP)        % Tile 3
addi $ESP, $ESP, -4     % Tile 3
addi $ACC, $zero, 5     % Tile 2
lw  $t1, 4($ESP)        % Tile 3
addi $ESP, $ESP, 4      % Tile 3
slt  $ACC, $ACC, $t1    % Tile 3
bgtz $ACC, iftrue1      % Tile 3
lw  $ACC, -8($FP)       % Tile 4
sw  $ACC, -4($FP)       % Tile 5
j    ifend             % Tile 6

iftrue1:                % Tile 7
lw  $ACC, -12($FP)      % Tile 8
addi $ACC, $ACC, 1     % Tile 9
sw  $ACC, -4($FP)       % Tile 10
ifend:                  % Tile 11
                         % No code for tiles 12 -- 16
08-182: Optimized Expression Stack

lw $ACC, -4($FP)       % Tile 1
addi $t2, $ACC, 0     % Tile 3
addi $ACC, $zero, 5   % Tile 2
addi $t1, $t2, 0      % Tile 3
slt $ACC, $ACC, $t1   % Tile 3
bgtz $ACC, iftrue1    % Tile 3
lw $ACC, -8($FP)      % Tile 4
sw $ACC, -4($FP)      % Tile 5
j ifend               % Tile 6

iftrue1:               % Tile 7
lw $ACC, -12($FP)     % Tile 8
addi $ACC, $ACC, 1    % Tile 9
sw $ACC, -4($FP)      % Tile 10

ifend:                 % Tile 11
                       % No code for tiles 12 -- 16
Tiles 2 and 3 could be covered by a single tile:

```
addi $ACC, $ACC, -x
bgtz $ACC, jumplab
```
08-184: Further Optimizations

lw $ACC, -4($FP)  % Tile 1
addi $ACC, $ACC, -5  % Tile 2/3
bgtz $ACC, iftrue1  % Tile 2/3
lw $ACC, -8($FP)  % Tile 4
sw $ACC, -4($FP)  % Tile 5
j ifend  % Tile 6

iftrue1:
  lw $ACC, -12($FP)  % Tile 8
  addi $ACC, $ACC, 1  % Tile 9
  sw $ACC, -4($FP)  % Tile 10

ifend:  % Tile 11
  % No code for tiles 12 -- 16
• Implementing in Java
  • Implement AATVisitor to do code generation
    • Don’t always call “accept” on children
    • Sometimes call “accept” on grandchildren, great-grandchildren, etc.
  • Will need to use “instance of” (slightly more ugly than semantic analysis)
Don’t always call “accept” on children
- Use “instance of” to decide which “tile” to use
- Call “accept” to tile the subtrees
- Output code with the “emit” function
08-187: **Implementation Details**

- Start will small tiles, slowly adding larger tiles to make the code more efficient
- Unfortunately, generated code using small tiles is hard to debug
- Plan on spending at least 50% of your time debugging rather than coding
  - Much more so with this project than with prior projects!
- Hopefully all early bugs have been worked out, so you only need to deal with getting codegen to work without going back and modifying SemanticAnalyzer.java, BuildTree.java, etc.
  - It’d be nice to win the lottery, too.